Kapitel 3:

Webserver-Frameworks: Django

Limitierungen der Low-Level-Entwicklung (1)

*Die Low-Level-Entwicklung mit PHP und SQL ist bzgl. **einer effizienten Softwareentwicklung** limitiert …*

- **Problem 1: Funktionalität wird durch primitive Operationen realisiert**
  - z.B. Formularverarbeitung: Jedes DB-Attribut ...
    - ... muss bei der Formularausgabe erzeugt und ggf. vorbelegt werden
    - … muss bei der Formulardatenverarbeitung geprüft und einem Datensatz-Attribut zugewiesen werden
  - Eine Erweiterung des Schemas führt zu vielen Anpassungen

- **Problem 2: Mischung von Darstellungs- und Kontroll-Ebene führt zu unübersichtlichen Strukturen**
  - Es ist oft schwer zu erkennen, was die wesentlichen Kontrollabläufe sind und ob alle Fälle vollständig behandelt sind

Limitierungen der Low-Level-Entwicklung (2)

Die Low-Level-Entwicklung mit PHP und SQL ist bzgl. ***der Sicherung der Datenintegrität*** *limitiert …*

- **Problem 3: Datenintegrität wird an vielen Punkten verteilt bzw. repliziert gesichert**
  - z.B. Feldlänge / Typ / Wertebereich eines Attributs eines editierbaren DB-Datensatzes ...
    - … ist im DB-Schema (SQL) festgelegt
    - … muss an HTML-Formular übergeben werden
    - ... muss bei POST-Daten geprüft werden
    - Eingabefehler sollten per Postback im Formular angezeigt werden
  - Änderung des Schemas führt dadurch zu vielen Anpassungen

Limitierungen der Low-Level-Entwicklung (3)

Die Low-Level-Entwicklung mit PHP und SQL ist bzgl. ***Schutz vor Sicherheitslücken*** ***limitiert …***

- **Problem 4: Sicherheitsprobleme müssen oft manuell und überall im Code verteilt gelöst werden**
  - vollständige Abschottung aller potentiellen Angriffsvektoren erforderlich
  - dies erfolgt auf sehr geringem Abstraktionsniveau, z.B.
    - SQL-Injections vermeiden bei textuellen Aufbau von SQL-Queries
    - XSS-Attacken vermeiden bei textuellem Aufbau von HTML-Ausgaben

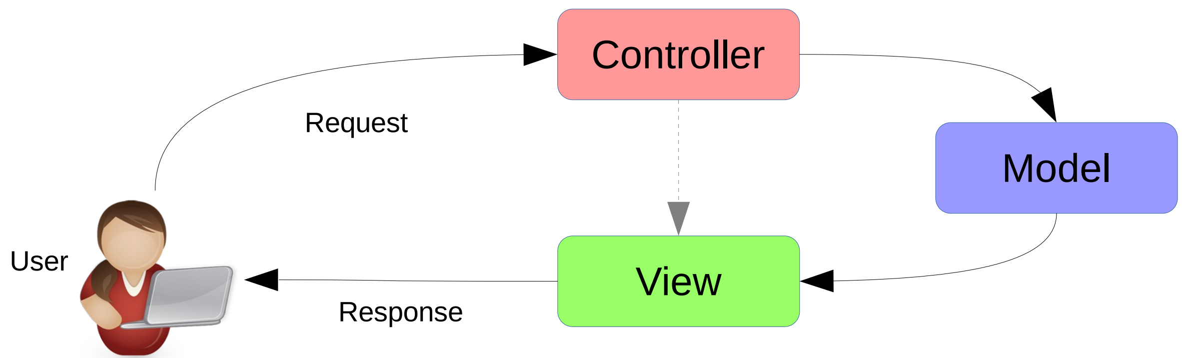
- **Problem 5: Mechanismen sind oft „Unsafe by default“**
  - Sicherheit bekommt man nur, wenn man ***alles richtig*** macht

Anforderungen

- **Wir brauchen ...**
  - **Trennung** von Modell, Präsentation und Steuerung (→ MVC)
    - Klarere Struktur der Applikation
    - Keine Durchmischung von Applikationslogik und Darstellung
  - **Abstrakte** Schema-Definition (Modell)
    - Schema-Eigenschaften werden nicht wiederholt sondern zentral definiert und gesichert (und überall benutzt)
  - Implementierung von Routine-Abläufen auf **hoher Ebene**
    - Automatisch Änderungsformulare zum Schema erzeugen, Antworten prüfen und speichern
  - **Sicherer** Umgang mit unsicheren Daten (u.a. Präsentation)
    - **Escape-by-Default** für alle unsicheren Inhalte (**Safe-by-Default**)
  - Mechanismen zur **zuverlässigen** Fehler- und Ausnahmebehandlung
    - z.B. keine unkontrollierten Ausgaben an den Nutzer (mitten in der Webseite) im Fehlerfall
  - ...

Hintergrund: MVC

- **Entwurfsmuster „Model – View – Controller“ (MVC)**
  - Idee: Klare **Trennung** von Modell, Präsentation und Steuerung
    - Das **Modell** („**model**“) hält und pflegt die **Daten**
      - Unabhängig von View und Controller
    - Die **Präsentation** („**view**“) stellt die Daten dem Benutzer dar
      - Daten werden aus dem Modell gelesen und dargestellt
    - Die **Steuerung** („**controller**“) verwaltet Modell und Präsentation
      - Sie verarbeitet und kontrolliert Änderungen und steuert auch z.B. die Einhaltung von Benutzerrechten (wer darf was sehen oder ändern)



Hintergrund: MVC

- **Entwurfsmuster „Model – View – Controller“ (MVC)**
  - Es gibt **viele Varianten** und offene Punkte dieses Modells
    - Siehe https://de.wikipedia.org/wiki/Model_View_Controller
  - *Zur Vertiefung*:
    - Wie sind die Abläufe in einem LAMP-Server hier zuzuordnen?
      - Was davon haben wir bisher davon explizit architekturell abgegrenzt?
      - Siehe Abschnitt „Serverseitige Webanwendungen“
    - Wie sind die Abläufe des Systems Webserver + Webclient hier einzuordnen?
      - Siehe Abschnitt „Zusammenspiel von Server und Browser bei Webanwendungen“
  - Viele Web-Frameworks setzen das Konzept um
    - Allerdings oft mit sehr unterschiedlichen Sichtweisen
      - Django (s.u.) hat z.B. eine eher unorthodoxe Sicht auf den Begriff „**View**“:
        - https://docs.djangoproject.com/en/4.2/faq/general/
        - https://django-book.readthedocs.io/en/latest/chapter01.html#the-mvc-design-pattern

Lösungsansätze (1)

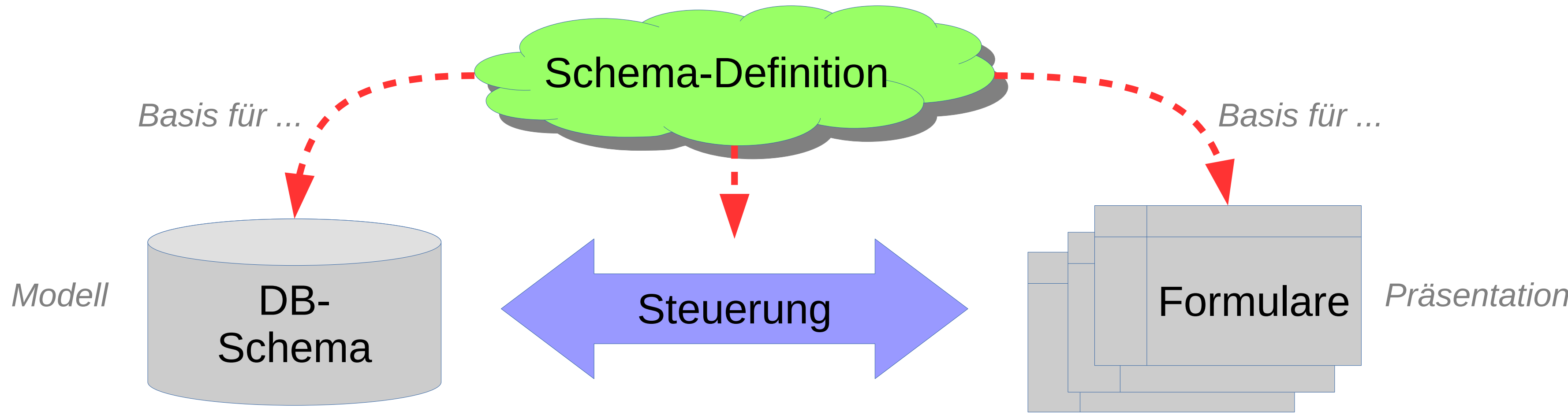
- **Trennung von Modell, Präsentation und Steuerung**
  - Separate Kontrollogik / Steuerung
    → Klare und verständliche Strukturen bei der Anfrageverarbeitung

    MVC: Controller
  - Mechanismen zur sicheren Fehlerbehandlung
    - z.B. keine unkontrollierten Ausgaben an den Nutzer im Fehlerfall
  - Nutzung von **Template-Engines** zur Präsentation

    MVC: View
    - Idee: Daten + **Template** → HTML-Response
    - Darstellungsaspekte (u.a. Design) werden im Template behandelt
    - Template- und Applikationsdesigner können getrennt / parallel arbeiten
    - Variable Designs durch umschaltbare Template-Sätze
  - Escape-by-Default für alle unsicheren Inhalte
    - Tagging (interne Markierung / Klassifizierung) von Strings
      - unsicher / sicher / bereits escaped
    - vermeidet Injection-Probleme (XSS, HTML-, SQL-Injection)

Lösungsansätze (2)

- **Nur eine Schema-Definitionen für Datenablage (Modell) und Formularbehandlung**
  - Beschreibung des Schemas auf hoher Abstraktionsebene



  - Generierung von DB-Schema und Formularen aus dem selben normativen Schema (Das DB-Schema könnte auch als normatives Schema dienen.)
  - Automatische Prüfung von Formulardaten auf Schema-Konformität
  - Bietet konsistenten / effizienten Mechanismus zu Verabeitungskette DB → Formular → Bearbeitung → Formular → DB

Lösungsansätze (3)

- **Applikationsstruktur, die Routineabläufe automatisiert**
  - „**Konvention vor Konfiguration**“
    - Softwaredesign-Paradigma („*Don´t Repeat Yourself*“)
    - http://de.wikipedia.org/wiki/Konvention_vor_Konfiguration
  - Hohes Abstraktionsniveau wo immer möglich, z.B.
    - Abstraktes Schema (einmal für Alles)
    - Abstrakte Filterung von Daten (keine komplizierten SQL-Joins)
    - Robuste URL-Analyse und -Synthese
    - Hierarchische HTML-Templates
  - Robustheit und Pflegbarkeit
    - Logging / Benachrichtigung des Administrators bei Fehlern
    - automatische Schema- und Datenmigration bei Upgrades
    - KISS („Keep it small and simple“)
  - Skalierbarkeit (Funktionen, Schema, Performanz)

Lösung: Web-Application-Frameworks

- **Web-Application-Frameworks**
  - Software, die die effiziente Entwicklung von Web-Applikationen unterstützt, vereinfacht und sicherer macht
    - Konzept: Die Mechansimen werden nur noch an den entscheidenden Punkten (Schemaentwurf, Applikationslogik, HTML-Design) angepasst
  - Realisieren die o.g. Lösungsansätze (mehr oder weniger)
    - Datenbankabstraktion, Schemamanagement, Template-Engine
    - **Scaffolding** (typische Abläufe sind sehr leicht realisierbar)
  - Es gibt eine große Zahl von ihnen, z.B.
    - ASP.NET — C# / Visual Basic (Microsoft)
    - AngularJS, NodeJS, Express — Javascript / Typescript (u.a. Google)
    - JSF — Java / JavaBeans (SUN/Oracle, IBM)
    - Zend Framework, Laravel, Symfony — PHP
    - **Django**, Flask — Python
    - Ruby on Rails — Rails
    - ...
      - siehe http://de.wikipedia.org/wiki/Liste_von_Webframeworks oder z.B. in einem MDN Artikel

Web-Application-Framework: Django

- **Django ist ein auf der Sprache Python basierendes Web-Application-Framework**
  - Integrierter Objektrelationaler Mapper (ORM)
    - Datenbank-Backend ist austauschbar (MySQL, Postgres, Oracle, SQLite, ...)
    - Schema-Definition und DB-Zugriff erfolgen High-Level, ohne Bedarf für SQL
  - Generiert zum Schema ein komplettes Administrations-Interface
    - Basierend auf Benutzerverwaltung mit fein steuerbaren Rechten
  - Mächtige Template-Sprache
    - Trennung von Datenaufbereitung und Darstellung (mit Vererbung zwischen Templates)
  - Flexibles URL-Management
    - Mustergesteuerte URL-Zerlegung und URL-Synthese
  - Sicher vor SQL-Injections, CSRF, Schutz vor XSS
  - ...

Programmiersprache Python

- **Python** Grundkonzepte
  - universelle Multiparadigmen-Sprache
    - imperativ, objektorientiert, funktional
  - dynamisch getypt (aber auch strikt getypt)
    - Typen sind an Daten / Objekte gebunden, nicht an Variablen
    - Ducktyping (→ Wikipedia)
  - Haupt-Ziele:
    - **Hohe Ausdrucksfähigkeit** (Probleme sind sehr kompakt lösbar)
    - **Sehr gute Lesbarkeit**
    - vielfältige Nutzbarkeit durch vielfältige Schnittstellen (Bibliotheken)
    - Erweiterbar durch Module
  - Programme müssen nicht explizit übersetzt werden
  - interaktiv nutzbar (Kommandos eingeben und direkt ausführen)
    - Python-Shell oder Erweiterung „ipython“

Programmiersprache Python

- **Beispiele**

Grundoperationen

```
[~] python3

>>> 1+2*3
7
>>> s = 'a' + "b"
>>> s
'ab'
>>> s == 'ab'
True
```

Strukturierte Anweisungen

```
>>> x = 3
>>> print(x)
3
>>> if x == 3:
...     print('x ist drei.')
... else:
...     print('x ist nicht drei.')
...
x ist drei.
```

Funktionen

```
>>> def max(a,b):
...     if a>b:
...         return a
...     else:
...         return b
...

>>> max(4,6)
6
```

Klassen, Methoden

```
>>> class C():
...     def f(self):
...         pass   // do nothing
... class D(C):
...     def f(self):
...         print("Ich bin D.")
...
>>> x = D()
>>> x.f()
Ich bin D.
```

Programmiersprache Python

- **Beispiele für Datentypen und Ausdrücke**

Tupel

```
>>> x = (0,1,2,3,4,5)
>>> x[3]
3
>>> x[-1]
5
>>> x[3:5]
(3, 4)
>>> x[3:4]
(3,)
>>> (3)
3
```

(3,) ist 1-Tupel
(3) ist Zahl

Listen

```
>>> x = [0,1,2,3,4,5]
>>> x[3]
3
>>> x.append('Y')
>>> x[3:]
[3, 4, 5, 'Y']
>>> x[:3]
[0, 1, 2]
>>> x[3:4]
[3]
```

Warum ist hier [3,] unnötig?

Dictionaries

```
>>> s = dict(a=1, b=2)
>>> s
{'a':1, 'b':2}
>>> s == {'a':1, 'b':2}
True
>>> s['a']
1
>>> s.get('c', 'unknown')
'unknown'
>>> s.items()
[('a', 1), ('b', 2)]
```

```
>>> '%s + %s = %s' % (1,2,3)
'1 + 2 = 3'

>>> a, b   =  '', 'sonst'
>>> a or b, bool(a), bool(b)
('sonst', False, True)

>>> x = 'ja' if a else 'nein'
>>> print(x)
'nein'
```

```
>>> [ x*x for x in [1,2,3] ]
[1, 4, 9]

>>> p = [ (x, x*x) for x in [1,2,3] ]
>>> p
[(1, 1), (2, 4), (3, 9)]

>>> dict(p)
{1: 1, 2: 4, 3: 9}
```

„List-Comprehension“

Programmiersprache Python

- **Python-Doku**
  - Zentrale Python-Weseite: https://www.python.org
    - Enthält auch eine interaktive Shell: https://www.python.org/shell/
    - Doku: https://docs.python.org/
    - Interakt. Tutorials: https://docs.python.org/3/tutorial/
      https://www.learnpython.org/
    - Weitere Tutorials: https://wiki.python.org/moin/BeginnersGuide/Programmers
  - Wikipedia-Seite (Überblick):
    http://de.wikipedia.org/wiki/Python_(Programmiersprache)
  - Für frühere Python-Version (Python 2.x):
    - Python 2 Quick-Reference (PQR): http://rgruet.free.fr
      - **Sehr nützliches Nachschlagewerk**, leider bisher kein Python 3 Update
      - Aktuell PQR 2.7: http://rgruet.free.fr/PQR27/PQR2.7.html

Django Quickstart

- **Django-Installation in Debian-Linux**

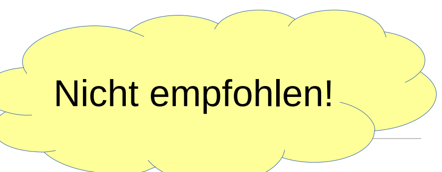
1. Grundlagen-Pakete installieren
   - Auf den Übungs-Servern (scilab-*) bereits erledigt
   - Als User root …

```
aptitude install ipython3 python3-pip gettext
    # Unterstützung für MySQL, WSGI, Sprachunterstützung
aptitude install python3-psycopg2 python-sqlite
    # Optional: Unterstützung für weitere DBMS (Postgres, SQLite)
aptitude install ipython3
    # Optional: Die Python-Shell iPython
```

2. Django installieren (2 Möglichkeiten)
   - Methode A: Als **Debian-Paket**
   - Methode B: Mit **PIP** → wählbar, z.B. Django **4.2**

– Wir benutzen hier Methode B (PIP)

Django Quickstart



- Alternative: **Django-Installation als Debian-Paket**
  - Als User root …

```
aptitude show python-django python3-django
    # Paket-Infos zu "python-django" und "python3-django" anschauen

aptitude install python3-django
    # "python-django" installieren (dauert ca. eine Minute)
```

  - Es gibt noch diverse weitere Pakete

```
aptitude search python3-django
    # ca. 50 Erweiterungs-Pakete zu Django werden aufgelistet
```

  - Nachteil: Debian-Pakete hier zum Teil nicht sehr aktuell

Django Quickstart

- Hintergrund: **Python Packet-Installer PIP**
  - PIP ist ein Python-Werkzeug um Python-Pakete zu installieren
    - Funktioniert Unabhängig von den Debian-Paketen
    - Bietet sehr große Zahl von Paketen, meist sehr aktuelle Versionen
  - ggf. zunächst **PIP** installieren: Als User root …

```
aptitude install python3-pip
    # Werkzeug pip installieren
```

    - Auf den Übungs-Servern (scilab-*) bereits erledigt

Django Quickstart

- Hintergrund: **Python Packet-Installer PIP**
  - PIP-Pakete können an 2 Orten installiert werden …
    - ins **System** (als User root)
    - in ein **Benutzer-Verzeichnis** (als regulärer Benutzer)

Ab jetzt am Besten als regulärer User

  - Bei Bedarf zunächst **PIP** Paket-Liste aktualisieren

```
pip3 install --upgrade pip --user
    # als User (oder als root ohne --user ins System installieren)
```

Immer `pip3` benutzen (nicht `pip`)

  - Paket-Index von PIP (Suche nach Paketen)
    - https://pypi.org/search/
    - z.B. zu Django:
      https://pypi.org/search/?q=Django
  - Dokumentation zu PIP:
    - https://pip.pypa.io/en/stable/user_guide/

Django Quickstart

- **Django-Installation mit PIP**
  - **Django** durch PIP installieren: Als regulärer User …

```
pip3 install Django==4.2 --user
    # Django Version 4.2.x in ~/.local/ installieren
```

    - PIP installiert so die Pakete im Benutzer-Verzeichnis `~/.local/`
  - Damit die Programme gefunden werden, muss `~/.local/bin` in den Programm-Suchpfad aufgenommen werden
    - Abhängig von der Shell
    - Für csh / tcsh: Eintrag `set path=( ~/.local/bin $path )`

      → systemweit in `/etc/csh.cshrc` (systemweit, als root)

      → oder benutzerlokal in `~/.cshrc`

      evtl. direkt nach der Installation 'rehash' um das neue Programm zu finden
    - Auf den Übungs-Servern (scilab-*) systemweit bereits erledigt
  - Danach kann man als User 'django-admin' aufrufen

Web-Application-Framework: Django

- **Django: Es gibt sehr Ausführliche Dokumentation**
  - Homepage: https://www.djangoproject.com/
    - Online-Doku Django 4.2: https://docs.djangoproject.com/en/4.2/
  - The Django Book: https://django-book.readthedocs.io/
    - Freies Online-eBook, Tutorial
  - Aktive Community
    - Viele (zentral und dezentral gepflegete) Module
    - z.B. Django-Snippets (Tricks & Erweiterungen)
      - https://djangosnippets.org/
    - z.B. Django im **Python Package Index** (https://pypi.org/)
      - https://pypi.org/search/?q=Django

Django Quickstart

- **Neues Projekt anlegen (als User)**
  - `mkdir django ; cd django`
    - # unser Verzeichnis für unsere Django-Projekte
  - `django-admin startproject test1`
    - # Wir legen ein neues Projekt "test1" an
  - `cd test1 ; dir -R`

```
# Schauen wir uns an …
test1                                <-- wir sind hier (aktuelles Verzeichnis)
       manage.py                     <-- unser Management-Programm
       test1                         <-- das Config-Verzeichnis des Projekts
             __init__.py
             settings.py             <-- hier macht man Einstellungen
             urls.py
             wsgi.py
```

  - `python3 ./manage.py`
    - chmod u+x ./manage.py # Script vorher ggf. noch ausführbar machen
    - Evtl. in der ersten Zeile „python“ gegen „python3“ austauschen
    - # ab jetzt rufen wir dieses Script nur noch mit „`./manage.py`“ auf

Django Quickstart (falls SQLite)

- **Django ist vorkonfiguriert für SQLite als Datenbank**
  - **SQLite** benötigt keinen eigenen Datenbank-Server
    - Die Daten werden in lokalen Dateien abgelegt
    - Siehe https://www.sqlite.org/docs.html
  - In einfachen Szenarien genügt SQLite zum Betrieb
    - Geringe Last, geringe Anforderungen an Zuverlässigkeit, ...
  - # Auszug aus **`test1/settings.py`**

```
DATABASES = { 'default': {
    'ENGINE': 'django.db.backends.sqlite3',
    'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
} }
```


  - `./manage.py migrate`
    - Aktualisiert / Erzeugt Datenbank-Schema
  - `./manage.py createsuperuser`
    - # Legt Admin-Account an (Name, Mailadresse und neues Passwort angeben)

Django Quickstart (falls SQLite)

- **Wie sieht die SQLite-Datenbank jetzt aus?**
  - `./manage.py dbshell`
    - # nur interessehalber mal anschauen …
      .tables

```
auth_group
auth_group_permissions
auth_permission
auth_user
auth_user_groups
auth_user_user_permissions
django_admin_log
django_content_type
django_migrations
django_session
```

    - # Inhalt der Benutzerdatenbank ...
      .schema **auth_user**
      SELECT * FROM auth_user;
    - Tipp: evtl. muss man als User root die sqlite3-Tools installieren

```
aptitude install sqlite3
    # Kommando-Frontend für sqlite3 installieren
```

DB-Shell, ruft Shell zum konfigurierten DB-Backend auf (Aufruf unabhängig vom DB-Backend)

Die Kommandos **in** der DB-Shell hängen aber natürlich vom DBMS ab (hier: **SQLite**)

Tipp: Mit den Kommandos „`.headers on`“ und „`.mode columns`“ wird die Ausgabe lesbarer. (Kann man auch in ~/.sqliterc schreiben.)

Django Quickstart (falls mySQL)

- **Alternative: z.B. mysql-Datenbank**
  - EDIT **test1/settings.py**
    - # Editiere folgende Zeilen um die Datenbank anzubinden:

```
DATABASES = { 'default': {
    'ENGINE':   'django.db.backends.mysql',
    'NAME':     'test1',
    'USER':     'lamp',
    'PASSWORD': 'xxx_MEIN_PASSWORT_xxx',   } }
```

  - mysql
    - CREATE DATABASE test1;
      # wir legen noch die o.g. DB an
      # ab jetzt benutzen wir statt "mysql" das Kommando "./manage.py dbshell"
  - ./manage.py migrate
    - Aktualisiert / Erzeugt Datenbank-Schema
    - Die SQL-Operationen kann man vorher mit „./manage.py sqlmigrate“ sehen
  - ./manage.py createsuperuser
    - # Legt Admin-Account an (Name, Mailadresse und neues Passwort angeben)

Django Quickstart (falls mySQL)

- **Wie sieht die mysql-Datenbank jetzt aus?**
  - `./manage.py dbshell`
    - # nur interessehalber mal anschauen …
      SHOW tables;

```
+----------------------------+
| Tables_in_test1            |
+----------------------------+
| auth_group                 |
| auth_group_permissions     |
| auth_permission            |
| auth_user                  |
| auth_user_groups           |
| auth_user_user_permissions |
| django_admin_log           |
| django_content_type        |
| django_migrations          |
| django_session             |
+----------------------------+
```

    - # Inhalt der Benutzerdatenbank ...
      DESCRIBE **auth_user**;
      SELECT * FROM auth_user \G

DB-Shell, ruft Shell zum konfigurierten DB-Backend auf (Aufruf unabhängig vom DB-Backend)

Die Kommandos **in** der DB-Shell hängen aber natürlich vom DBMS ab (hier: **mySQL**)

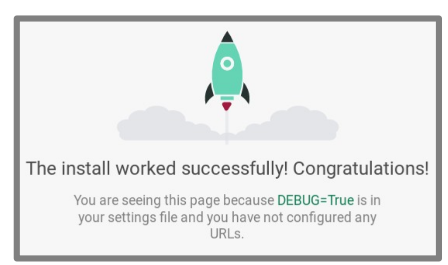
Django Quickstart

- **Start des Test-Servers auf Port 8000**
  - `./manage.py runserver 0.0.0.0:8000`
    - # Wir starten die Web-Applikation
      # Und können auf der Adresse des Servers mit einem Webbrowser
      # darauf zugreifen, z.B.
- **Web-Client-Zugriff auf den Test-Server**
  - z.B. http://scilab-0100.cs.uni-kl.de:8000/

    → Fehlermeldung
  - EDIT **test1/settings.py**
    - # Editiere folgende Zeilen um den Server-Hostnamen freizugeben, z.B.:

      ALLOWED_HOSTS = ['scilab-0100.cs.uni-kl.de']
  - Nochmal http://scilab-0100.cs.uni-kl.de:8000/

    → „*The install worked successfully!*“



Django Quickstart

- **Das Admin-Interface ist ein Django-App**
  - Man kann eigene Apps schreiben oder existierende nutzen
  - Auszug aus `test/settings.py`
    - ```
      INSTALLED_APPS = [
          'django.contrib.admin',
          'django.contrib.auth',
          'django.contrib.contenttypes',
          'django.contrib.sessions',
          'django.contrib.messages',
          'django.contrib.staticfiles',
      ]
      ```
  - Auszug aus `test1/urls.py`
    - ```
      urlpatterns = [
          path('admin/', admin.site.urls),
      ]
      ```
  - Zugriff auf Admin-Interface
    - z.B. http://scilab-0100.cs.uni-kl.de:8000/admin/

Django Quickstart

- **Wir legen das Datenschema als App „Prüfungsamt“ an**
  - `./manage.py startapp pruefungsamt`
    - # wir legen ein neues Applikationsskelett an
      # Es gibt jetzt neue Unterverzeichnisse und Dateien, u.a.:
      ```
      ~ / test1 / pruefungsamt / models.py
                                 admin.py
      ```
  - EDIT `pruefungsamt/models.py`
    - # Modell neu anlegen (s.u.)
  - EDIT `test1/settings.py`
    - `INSTALLED_APPS = ( ..., 'pruefungsamt', ...)`
  - `./manage.py makemigrations`
    - # Anpassung der Datenbank vorbereiten (→ pruefungsamt/migrations/0001_initial.py)
  - `./manage.py migrate`
    - Hintergrund: Anzeige der Migrationen und der dazu benutzten SQL-Statements:
      - `./manage.py showmigrations                  # zeigt Status der Migrationen`
      - `./manage.py sqlmigrate pruefungsamt 0001    # zeigt SQL-Statements`

Django Quickstart

- **pruefungsamt/models.py**

```
from django.db import models

class Student(models.Model):
    matnr = models.IntegerField(unique=True)
    name  = models.CharField(max_length=64)
    hoert = models.ManyToManyField('Vorlesung', blank=True)

class Professor(models.Model):
    persnr = models.IntegerField(unique=True)
    name   = models.CharField(max_length=64)

class Vorlesung(models.Model):
    vorlnr = models.IntegerField(unique=True)
    titel  = models.CharField(max_length=128)
    dozent = models.ForeignKey(Professor, null=True,
                               on_delete=models.SET_NULL)
```

  - Hintergrund-Info: Beachten Sie, dass ForeignKey / ManyToManyField als ersten Parameter eine Klasse oder deren Namen (String) haben kann. (Warum String?)

Django Quickstart

- **Wir legen das Admin-Interface zur App „Prüfungsamt“ an**
  - EDIT pruefungsamt/admin.py
    - # Admin-Definition anlegen (→ nächste Folie)

Django Quickstart

- **pruefungsamt/admin.py**

```
from .models import *
from django.contrib import admin


class Student_Admin(admin.ModelAdmin):
    pass

admin.site.register(Student, Student_Admin)


class Professor_Admin(admin.ModelAdmin):
    pass

admin.site.register(Professor, Professor_Admin)


class Vorlesung_Admin(admin.ModelAdmin):
    pass

admin.site.register(Vorlesung, Vorlesung_Admin)
```

Import der Modell-Klassen von **pruefungsamt** (.), damit wir *Student*, *Professor* und *Vorlesung* nutzen können

„`pass`“ ist ein Platzhalter, da wir hier (noch) nichts weiter angeben wollen

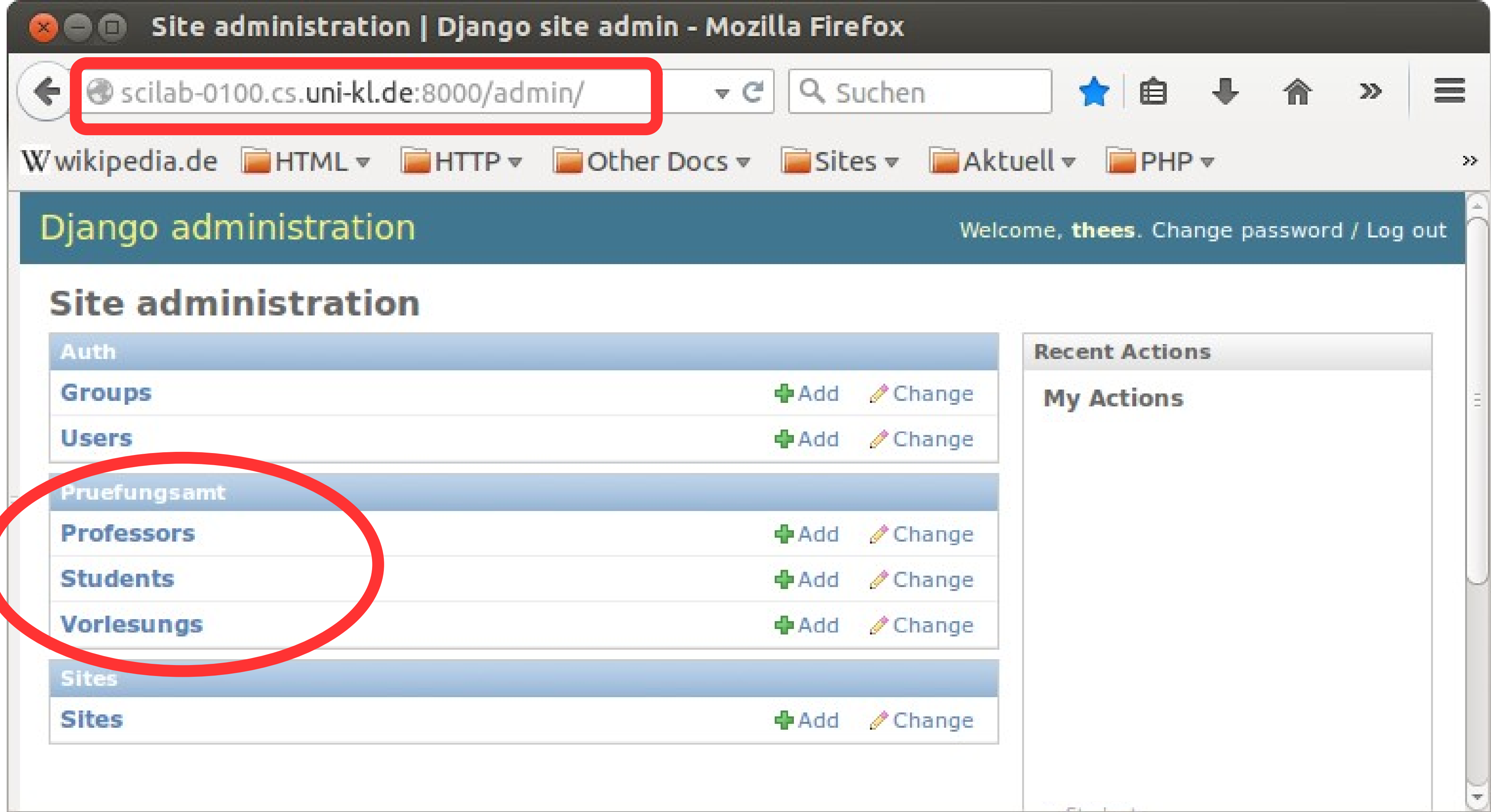
Django Quickstart

- **Danach jeweils den Web-Service starten ...**
  - `./manage.py runserver 0.0.0.0:8000`

- **Web-Client-Zugriff auf den Test-Server**
  - URL z.B. http://scilab-0100.cs.uni-kl.de:8000/admin/
  - Im Admin-Interface pflegen wir z.B. die Test-Daten ein

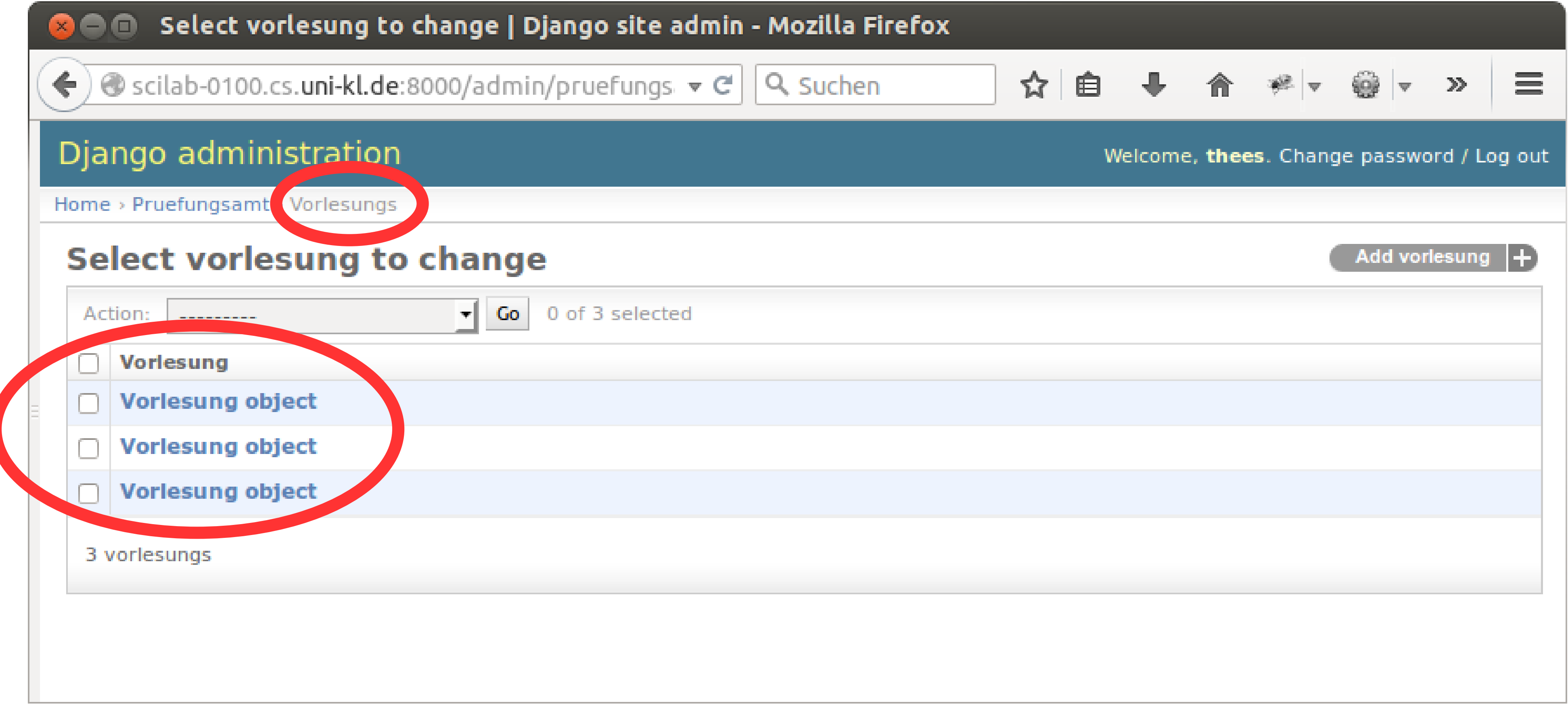
Django Quickstart

- **Die 3 Klassen sind im Admin-Interface zugreifbar**



Django Quickstart

- **Die Liste der Vorlesungen ist aber noch „*unschön*“**
  - Wir haben aber ja auch z.B. noch nicht gesagt, was wir an Attributen sehen wollen



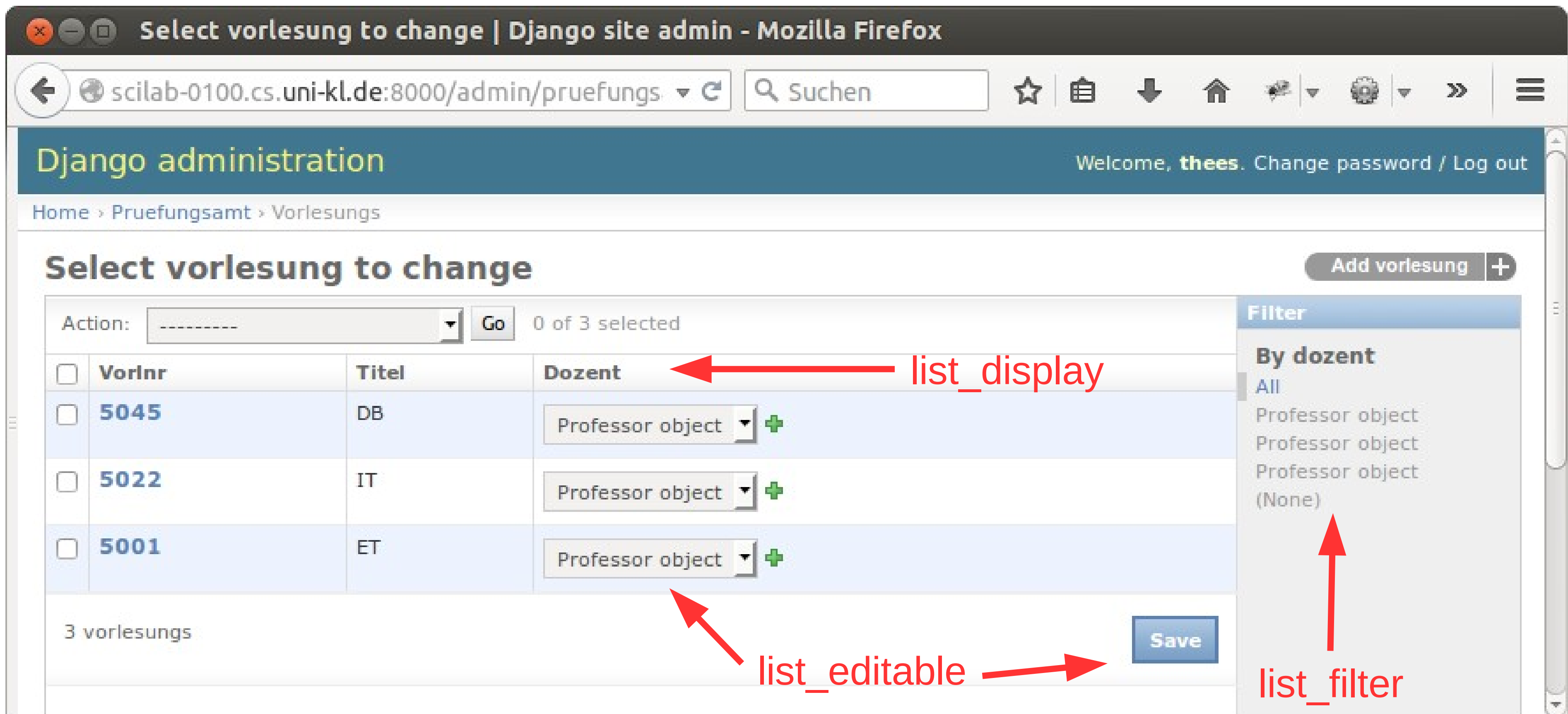
Django Quickstart

- **pruefungsamt/admin.py** (erweitert)

```
from .models import *
from django.contrib import admin


class Student_Admin(admin.ModelAdmin):
    list_display = ('matnr', 'name',)
    filter_horizontal = ('hoert',)

admin.site.register(Student, Student_Admin)


class Professor_Admin(admin.ModelAdmin):
    list_display = ('name', 'persnr', )

admin.site.register(Professor, Professor_Admin)


class Vorlesung_Admin(admin.ModelAdmin):
    list_display  = ('vorlnr', 'titel', 'dozent',)
    list_filter   = ('dozent',)
    list_editable = ('dozent',)

admin.site.register(Vorlesung, Vorlesung_Admin)
```

Django Quickstart

- **Die Liste der Vorlesungen ist fast perfekt**



  - Objekte sollten noch benannt werden (statt „*Professor object*“)
    - Lösung: Jedes Objekt sollte eine String-Darstellung liefern

Django Quickstart

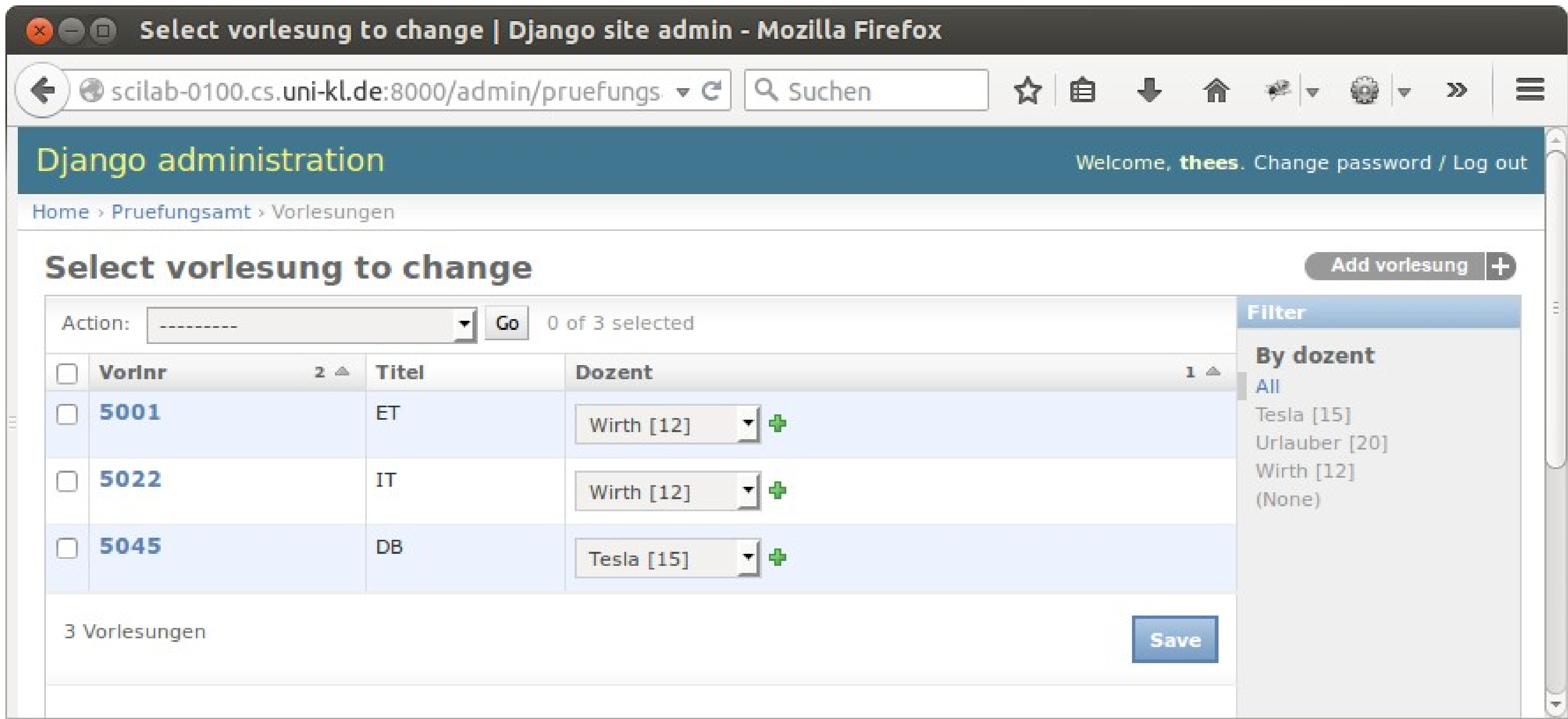
- **Wir ergänzen die Schema-Objekte:**
  - um eine Methode `__str__`, die einen lesbaren Text liefert
  - um eine Meta-Klasse, die Zusatzinformationen liefert
    - z.B. Name der Klasse in der Darstellung (singular / plural)
    - z.B. Standard-Reihenfolge bei Queries

- **`pruefungsamt/models.py`**

```
class Professor(models.Model):
    persnr = models.IntegerField(unique=True)
    name   = models.CharField(max_length=64)

    def __str__(self):
        return "%s [%s]" % (self.name, self.persnr)

    class Meta:
        verbose_name        = 'Professor'
        verbose_name_plural = 'Professoren'
        ordering            = ('name', 'persnr',)
```

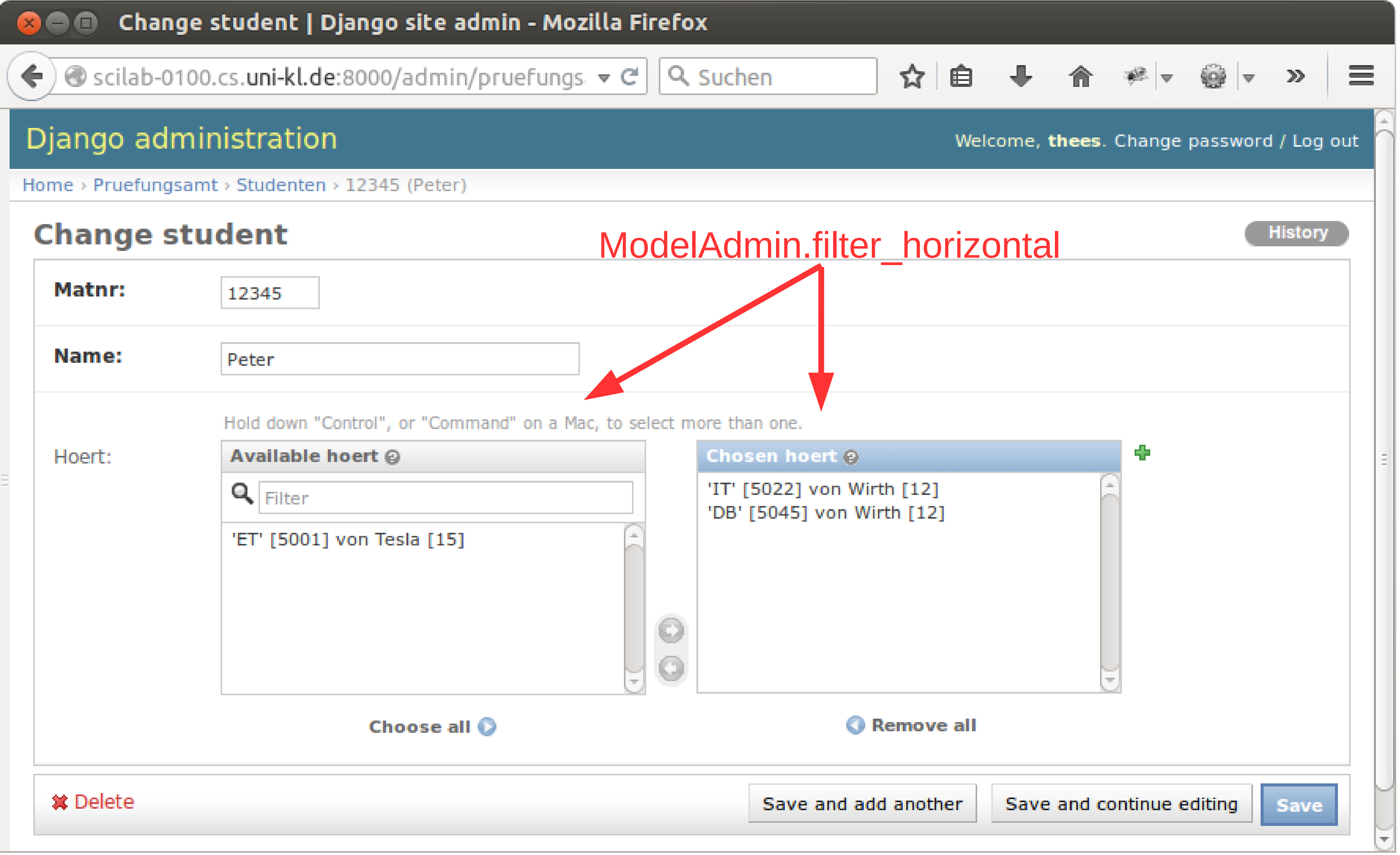
Django Quickstart

- **Die Liste der Vorlesungen ist jetzt fertig**



Django Quickstart

- **Und auch die Editier-Seite ist fertig**



Django Quickstart

- **Schema-Zugriff mit der Python-Shell (1)**
  - Man kann die Schema-Objekte in eigenen Programmen oder gar interaktiv in einer Python-Shell zugreifen
    - Zu letzterem rufen wir das Django-Kommando „shell“ auf
    - Wenn `ipython` installiert ist, wird dieses als Shell benutzt (mehr Komfort)
  - `./manage.py shell`

```
from pruefungsamt.models import *

s = Student()
s.name = 'Tester'
s.matnr = 123123
s.save()
```

  - Das neue Objekt ist jetzt dauerhaft in der Datenbank abgelegt
    - Wir könne es z.B. im Admin-Interface sehen

Django Quickstart

- **Schema-Zugriff mit der Python-Shell (2)**
  - ./manage.py shell
    - `from pruefungsamt.models import *`
    - `Student.objects.all()`

```
[<Student: 12345 (Peter)>, <Student: 25403 (Jonas)>, <Student:
26120 (Buche)>, <Student: 27103 (Fauler)>, <Student: 123123
(Tester)> ]
```

Alle Studenten (-Objekte)

  - Solche Query-Sets werden von Django in SQL-Queries umgesetzt
    - **Hintergrund:** Wie sieht die SQL-Anfrage dazu aus?

```
qs = Student.objects.all()
str(qs.query)
```

```
SELECT
    `pruefungsamt_student`.`id`    ,
    `pruefungsamt_student`.`matnr`,
    `pruefungsamt_student`.`name`
FROM `pruefungsamt_student`
ORDER BY (`pruefungsamt_student`.`name`  ASC ,
          `pruefungsamt_student`.`matnr` ASC )
```

Django Quickstart

- **Schema-Zugriff mit der Python-Shell (3)**
  - ./manage.py shell
    - `from pruefungsamt.models import *`
    - `Student.objects.filter(hoert__titel = 'ET')`

```
[<Student: 25403 (Jonas)>, <Student: 26120 (Buche)>]
```

Alle Studenten, die eine Vorlesung hören die den Titel „ET“ hat.

  - So kann man komplexe Anfragen stellen
    - **Hintergrund:** Wie sieht die SQL-Anfrage dazu aus?

```sql
SELECT
    `pruefungsamt_student`.`id`,
    `pruefungsamt_student`.`matnr`,
    `pruefungsamt_student`.`name`
FROM `pruefungsamt_student`
INNER JOIN `pruefungsamt_student_hoert`
   ON (`pruefungsamt_student`.`id` = `pruefungsamt_student_hoert`.`student_id`)
INNER JOIN `pruefungsamt_vorlesung`
   ON (`pruefungsamt_student_hoert`.`vorlesung_id` = `pruefungsamt_vorlesung`.`id`)
WHERE `pruefungsamt_vorlesung`.`titel` = 'ET'
ORDER BY (`pruefungsamt_student`.`name`  ASC ,
          `pruefungsamt_student`.`matnr` ASC )
```

Verständnisfrage:
Warum 2 Joins?

Django: Daten exportieren / importieren

- **Management-Funktionen um Daten zu exportieren**
  - Format: z.B. JSON (default)

```
./manage.py dumpdata pruefungsamt --indent 4 > pruefungsamt.json
```

pruefungsamt.json

```
[
{
    "model": "pruefungsamt.student",
    "pk": 1,
    "fields": {
        "matnr": 26120,
        "name": "Fichte",
        "hoert": [ 3, 1 ]
    }
},
{
    "model": "pruefungsamt.student",
    "pk": 2,
# ....
}
]
```

Liste von ...

Datensatz

Datensatz

Tipp: ggf. beim Editieren: vor dem „]“ darf in JSON kein Komma stehen

Django: Daten exportieren / importieren

- **Management-Funktionen um Daten zu exportieren**
  - Format: z.B. XML

```
./manage.py dumpdata pruefungsamt --format xml --indent 4 > pruefungsamt.xml
```

pruefungsamt.xml

```
<?xml version="1.0" encoding="utf-8"?>
<django-objects version="1.0">

    <object model="pruefungsamt.student" pk="1">
        <field name="matnr" type="IntegerField">26120</field>
        <field name="name" type="CharField">Fichte</field>
        <field name="hoert" rel="ManyToManyRel"
                   to="pruefungsamt.vorlesung">
                            <object pk="3"></object>
                            <object pk="1"></object>
         </field>
    </object>

    <object model="pruefungsamt.student" pk="2">
<!-- .... -->
    </object>
</django-objects>
```

XML-Element enthält ...

Datensatz

Datensatz

Django: Daten exportieren / importieren

- **Management-Funktionen um Daten zu importieren**
  - Format: z.B. JSON (default)

```
./manage.py loaddata pruefungsamt.json
```

  - Es darf keine ID-Kollisionen geben
    - Sinnvoll so vor allem zu Import in leere Datenbank (z.B. Testsysteme, Initialdaten bei Neuinstallation)
  - Ausblick: Alternative Variante **Natural Keys**
    - Idee: Statt künstlicher IDs „natürliche" Schlüssel für exportierte / importierte Daten verwenden (nur beim Export / Import, nicht in der Datenbank)
    - Beispiel: Matrikelnummer, Personalnummer, Vorname+Nachname, ...
    - Umsetzung: Schema-Funktionen **natural_key()** und **get_by_natural_key()** und bei **dumpdata** die Optionen **--natural-foreign** und **--natural-primary**
    - Mehr dazu: https://docs.djangoproject.com/en/4.2/topics/serialization/#natural-keys

Django Models

- **Feld-Typen** (z.B. CharField, IntegerField, …)
  - Instanzen der Klassen `django.db.models.XxxField`
  - Es gibt allgemeine Optionen für alle Feldtypen (Auszug)
    - siehe https://docs.djangoproject.com/en/4.2/topics/db/models/#field-options
    - **blank** = True (default False):
      - leere Eingabe bei Validierung erlauben (hat keine DB-Schema-Auswirkung!)
    - **null** = True (default False):
      - NULL in DB-Schema erlaubt (leere Eingaben werden als NULL modelliert)
    - **unique** = True (default False):
      - Attribut ist UNIQUE im DB-Schema, Duplikat-Werte nicht erlaubt (außer NULL)
    - **default** = 5 (*Wert* oder *Callable*):
      - Bestimmt Default-Wert im DB-Schema und Intialwert in Modell-Objekten
    - **primary_key** = True (default False):
      - Attribut ist Primärschlüssel im DB-Schema (nur einteilig möglich)
      - Der Primärschlüssel ist immer unter dem Alias-Namen „pk" ansprechbar
      - Wird kein Primärschlüssel angelegt, wird implizit ein IntegerField „id" erzeugt

nur zur Formular-Validierung

Bei blank mit unique ist null praktisch zwingend. (Übung: Warum?)

möglichst nicht verwenden

Django Models

- **Feld-Typen**
  - allgemeine Optionen *(Fortsetzung)*
    - **verbose_name** = 'Attributname'
      - Definiert den Anzeigenamen des Attributs, z.B. in Model-Forms
      - Default ist der Attribut-Name
      - Meistens kann diese Option auch als erster anonymer Parameter übergeben werden (Ausnahme: Relationen – s.u.)
    - **help_text** = `'...'`
      - Hilfe-Text der in Model-Forms beim Attribut angezeigt wird (z.B. im Admin-Interface)
    - **choices** = `( ('m', 'male'), ('f', 'female'), ('d', 'diverse'), )`
      - Liste von Paaren des Musters (Wert, Anzeigetext)
      - Legt fest, welche Werte in dem Attribut erlaubt sind
        - Im obigen Beispiel genügt ein Zeichen, also models.CharField(max_length=1, choices=...)
      - Die Anzeigetexte werden bei Formularen (Model-Forms) als Auswahlliste angezeigt
      - Zu einem Attribut geschlecht mit obigen choices wird automatisch eine Methode `get_geschlecht_display()` definiert, die den Anzeigetext liefert.
    - **editable** = False (default True):
      - Feld ist bei False in Model-Forms nicht editierbar (z.B. im Admin-Interface)

Django Models

- **Feld-Typen**
  - allgemeine Optionen *(Fortsetzung)*
    - **validators** = [EmailValidator, MinLengthValidator(15)]
      - Liste der Validatoren, die erfüllt sein müssen
        - hier im Beispiel: Es muss eine Email sein, die mindestens 15 Zeichen lang ist
      - siehe https://docs.djangoproject.com/en/4.2/ref/validators/#built-in-validators
      - z.B.
        - `MinValueValidator(n)`, `MaxValueValidator(n)`
          - Eingabe (Zahl) muss größer-/kleiner-gleich n sein
        - `MinLengthValidator(n)`, `MaxLengthValidator(n)`
          - Länge der Eingabe (Zeichenkette) muss größer-/kleiner-gleich n sein
        - `EmailValidator`
          - Eingabe muss Email sein
        - `URLValidator`
          - Eingabe muss URL sein (und Ziel ggf. existieren)
        - `RegexValidator`(re)
          - Eingabe muss zu regulärem Ausdruck re passen
          - z.B. `RegexValidator(r'^#[0-9a-f]{6}$')`
            → 6-stellige CSS-Hex-Farbangaben z.B. `#ffaa37`

Django Models

- **Feld-Typen** (Grundlegende Typen)

  siehe https://docs.djangoproject.com/en/4.2/ref/models/fields/#field-types

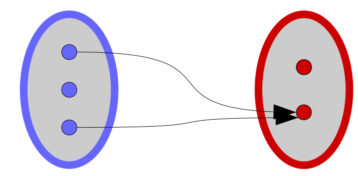
  - **CharField**
    - Zeichenkette, wird editiert in Formularfeld „<input type='text' ...>“ (einzeilig)
    - Verpflichtender Parameter max_length (Maximale Länge, für DB-Modell und Validierung)
  - **TextField**
    - Zeichenkette, wird editiert in Formularfeld „<textarea>...</textarea>“ (mehrzeilig)
    - Länge kann unlimitiert sein
  - **IntegerField**, **PositiveIntegerField**
    - Werte sind ganze (bzw. ganze positive) Zahlen
  - **BooleanField**, **NullBooleanField**
    - Werte True oder False bzw. True, False, Null

Django Models

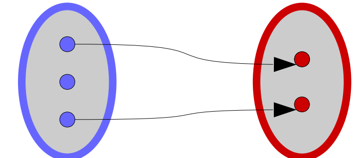
- **Feld-Typen** (komplexe Typen)
  - **DateField**, **TimeField**, **DateTimeField**
    - Datum- oder Zeitstempel-Felder
  - **EmailField**, **URLField**
    - Zeichenketten, die Emails oder URLs (ggf. mit existierendem Ziel) enthalten
  - **AutoField**
    - Spezielles IntegerField mit dem AUTO_INCREMENT-Verhalten von SQL
    - Das automatisch eingefügte id-Feld hat also die Definition
      `id = models.AutoField(primary_key=True)`
  - **FileField**, **ImageField**
    - Im Model-Form ein Eingabefeld, mit dem man Dateien (bzw. Bild-Dateien) hochladen kann
    - Zum FileField-Attribut `x` ist `x.path` der Dateisystem-Pfad der hochgeladenen Datei auf dem Server
      - siehe https://docs.djangoproject.com/en/4.2/topics/files/

Django Models: Relationen

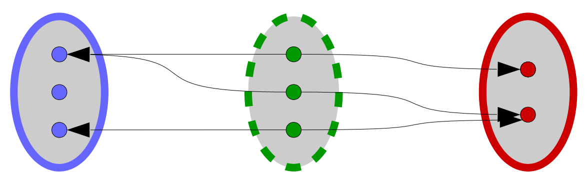
- **Feld-Typen** (Relationen)
  - **ForeignKey**(othermodel)



    - n:1-Relation zu othermodel (Klasse, Name einer Klasse oder 'self')
    - Die Relation wird als Fremdschlüssel in der DB-Tabelle realisiert.
  - **OneToOneField**(othermodel)



    - 1:1-Relation zu othermodel
    - Die Relation wird als Fremdschlüssel in der DB-Tabelle realisiert.
  - **ManyToManyField**(othermodel)



    - n:m-Relation zu othermodel
    - Da die Relation mit zwei n:1-Relationen realisiert wird, wird eine Modell-Hilfsklasse automatisch erzeugt (mit entsprechend eigener DB-Tabelle)
      - Man kann aber auch explizit eine solche Hilfsklasse anlegen und mit **ManyToManyField**(othermodel, **through**=eine_model_klasse) angeben.
      - Diese Hilfsklasse muss ForeignKey-Attribute zu beiden Klassen haben.

Django Models: Relationen

- **Relationen in Modell-Instanzen**
  - Beispiel: pruefungsamt/models.py

```
class Student(models.Model):
    matnr = models.IntegerField(unique=True)
    name  = models.CharField(max_length=64)
    hoert = models.ManyToManyField('Vorlesung', blank=True)


class Professor(models.Model):
    persnr = models.IntegerField(unique=True)
    name   = models.CharField(max_length=64)
```

  - ```
    class Vorlesung(models.Model):
        vorlnr = models.IntegerField(unique=True)
        titel  = models.CharField(max_length=128)
        dozent = models.ForeignKey(Professor, null=True,
                                   on_delete=models.SET_NULL)
    ```
  - Es gibt also …
    - Eine n:1-Beziehung Vorlesung.dozent zu Professor
    - Eine n:m-Beziehung Student.hoert zu Vorlesung
    ```

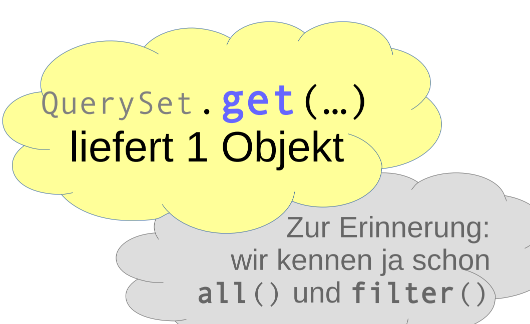
Django Models: Relationen

- **Relationen in Modell-Instanzen (n:1)**
  - ./manage.py shell
    - `from pruefungsamt.models import *`
    - `v = Vorlesung.objects.get(titel='DB')`
    - `v`

      <Vorlesung: 'DB' [5045] von Wirth [12]>
    - `d = v.dozent`
    - `d`

      <Professor: Wirth [12]>
    - `d.vorlesung_set.all()`

      [<Vorlesung: 'IT' [5022] von Wirth [12]>, <Vorlesung: 'DB' [5045] von Wirth [12]>]



  - Zur n:1-Relation Vorlesung.**dozent** → Professor ist *implitzit* die Rückrichtung Professor.**vorlesung_set** → Vorlesung als 1:n-Relation (liefert *Menge*, also *Query-Set*) zugreifbar.
  - Übungsfrage: Wie kann man das selbe Ergebnis direkt erhalten?
    - `Vorlesung.objects.filter(dozent=d)`

Django Models: Relationen

- **Relationen in Modell-Instanzen (n:m)**
  - ./manage.py shell
    - `from pruefungsamt.models import *`
    - `s = Student.objects.get(name='Fichte')`
    - `s`

      <Student: 26120 (Fichte)>
    - `s.hoert.all()`

      [<Vorlesung: 'ET' [5001] von Tesla [15]>, <Vorlesung: 'DB' [5045] von Wirth [12]>]
    - `v = s.hoert.all()[0]`
    - `v.student_set.all()`

      [<Student: 26120 (Fichte)>, <Student: 25403 (Jonas)>]
  - Zur n:m-Relation Student.**hoert** → Vorlesung ist *implitzit* die Rückrichtung Vorlesung.**student_set** → Student als n:m-Relation (liefert Menge, also *Query-Set*) zugreifbar.
  - Übungsfrage: Wie kann man das selbe Ergebnis direkt erhalten?
    - `Student.objects.filter(hoert=v)`

Django Models: Relationen

- **Relationen in Modell-Instanzen (1:1)**
  - 1:1-Relationen (OneToOneField) verhalten sich weitgehend wie n:1-Relationen.

    Es gibt prinzipiell nur zwei Unterschiede:
    - Es kann höchstens ein Objekt eine 1:1-Beziehung zu einem Zielobjekt haben
    - Die Rückrichtung ist entsprechend dann eindeutig (oder undefiniert).

      Daher hat das Implizit definierte Feld der Rückrichtung den Namen der Ausgangsklasse (diesmal ohne „_set“ am Ende) und enthält direkt einen Verweis auf das eine referenzierende Objekt.

      Gibt es kein referenzierendes Objekt, so löst der Zugriff eine Exception aus (DoesNotExist).
  - Verständnisfragen:
    - Angenommen, Vorlesung.dozent wäre folgendermaßen definiert

      ```
      dozent = models.OneToOneKey(Professor,   null=True,
                                               on_delete=…)
      ```
    - Welche Semantik hätte diese Schema-Variante?
    - Wie heißt hier also das implizit definierte Rück-Attribut?

Django Models: Relationen

- **Relationen in Modell-Instanzen**
  - Der Name des implizit erzeugten Rückrichtungs-Attributs kann man mit der Option **related_name** bei allen drei Relations-Typen ändern.
  - Beispiel:
    - ```
      class Vorlesung(models.Model):
          vorlnr= models.IntegerField(unique=True)
          titel = models.CharField(max_length=128)
          dozent= models.ForeignKey(Professor,null=True,on_delete=…,
                                    related_name='haelt_vorlesungen')
      ```
    - Die Menge der Vorlesungen, die Professor Wirth hält, erhält man mit
      ```
      p            = Professor.objects.get(name='Wirth')
      vorlesungen = p.haelt_vorlesungen.all()
      ```
  - Das Ändern dieses Attributnamens ist immer dann notwendig, wenn mehrere Relationen von einer Modell-Klasse zu einer anderen existieren
    - Die Rückrichtungen hätten sonst den selben Namen
    - **Beispiel**: Neben `hoert` könnte Student auch die ManyToMany-Beziehung `hiwi` zu Vorlesung haben. Beides würde die Rückrichtung `student_set` erzeugen.
      ```

Django Models: Relationen

- **Relationen in Modell-Instanzen**
  - Bei 1:1- und 1:n-Relationen **muss** mit der Option **on_delete** angegeben werden, was geschehen soll, wenn das referenzierte Objekt gelöscht wird.
  - **Werte** (Semantik analog zu SQL)
    - models.CASCADE
    - models.PROTECT
    - models.SET_NULL
    - models.SET_DEFAULT
    - models.SET(x) (x ist Wert oder Callable)
    - models.DO_NOTHING
  - Beispiel:

```
class Vorlesung(models.Model):
    # ...
    dozent = models.ForeignKey(Professor, null=True,
                               on_delete=models.SET_NULL)
```

Übungsfrage:
Warum gibt es kein
`on_update`?

Django Models: Metadaten

- **Durch die Meta-Klasse in einem Django-Modell können Eigenschaften des Modells gesteuert werden**
  - Meta-Attribute (Auszug)
    - **ordering** = `( '-matnr', 'name' )`
      - Definiert die Standard-Reihenfolge der Objekte als Tupel von Attributnamen
      - Steht vor dem Namen ein „**-**“, werden sie **absteigend** sortiert, (Standard: **aufsteigend**, also die kleinsten Werte zuerst)
      - Kann im Queryset mit order_by() wieder geändert werden (s.u.)
    - **unique_together** = (('name', 'geburtsort', 'geburtsdatum'), ('firma', 'pnr'))
      - Tupel von Tupeln von Attributnamen, deren Wert nicht tupelweise identisch sein dürfen.
      - Wird in das DB-Schema integriert
    - **verbose_name** = 'Vorlesung'

      verbose_name_plural = 'Vorlesungen'
      - Definiert die Benennung der Modell-Klasse im Admin-Interface
      - Default ist der Klassenname im Singular bzw. verbose_name+“s“ im Plural

Django Models: Metadaten

- **Beispiel: Zusatz-Angaben zur Klasse Professor**
  - eine Meta-Klasse
  - eine Methode `__str__`, die einen lesbaren Text liefert
    - Der Mechanismus ist Sicher gegen Injections, man muss hier nichts escapen, obwohl die Ausgabe z.B. im Admin-Interface benutzt wird
- **pruefungsamt/models.py**

```
class Professor(models.Model):
    persnr = models.IntegerField(unique=True)
    name   = models.CharField(max_length=64)

    def __str__(self):
        return "%s [%s]" % (self.name, self.persnr)

    class Meta:
        verbose_name        = 'Professor'
        verbose_name_plural = 'Professoren'
        ordering            = ('name', 'persnr',)
```

Django: QuerySet-API

- **Die QuerySet-API bietet Zugriff auf die Datenbank-Objekte**

  siehe https://docs.djangoproject.com/en/4.2/ref/models/querysets/

  – Ausgangspunkt ist z.B. die Komponente objects einer Modell-Klasse
    - z.B. `Vorlesung.objects`
  – Auf diese können wir eine **QuerySet-Methode** anwenden
    - z.B. `Vorlesung.objects.filter(dozent=d)`
    - Ergebnis ist **wiederum ein Queryset**
  – Dadurch können QuerySet-Methoden **verkettet** werden
    - z.B. `Vorlesung.objects.filter(dozent=d).filter(titel='ET')`

  – Präzisierung am Rande:
    - Die objects-Komponente liefert eigentlich kein QuerySet-Objekt, sondern ein Manager-Objekt. Dieser verhält sich aber weitgehend wie ein QuerySet.
    - Deshalb muss man `.all()` aufrufen um die Ergebnisse zu bekommen

Django: QuerySet-API

- **QuerySet-Methoden, die wieder QuerySets liefern**
  - **all**()
    - Erzeugt QuerySet aus einem Manager-Objekt: `Dozent.objects.all()`
  - **filter**(...)
    - Lässt alle Objekte durch, die die angegebenen Kriterien erfüllen
    - z.B. `Vorlesung.objects.filter(titel='ET')`
  - **exclude**(...)
    - Filtert alle Objekte aus, die die angegebenen Kriterien erfüllen
  - **order_by**(...)
    - Gibt Sortierkriterien an (analog zu ordering in Meta-Klasse)
    - z.B. `XYZ.objects.all().order_by('-matnr', '-name')`
  - **reverse**()
    - Kehrt Reihenfolge um
  - **distinct**()
    - Eliminiert Duplikate (bei komplexen Queries manchmal erforderlich)

Django: QuerySet-API

- **Filter-Kriterien** („Field-Lookups“)

  siehe https://docs.djangoproject.com/en/4.2/ref/models/querysets/#id4

  – Parameter für filter(), exclude() und get()

  – Direkter Paramtervergleich

    - `parametername__exact=Wert`
    - `parametername=Wert` (selbe Bedeutung wie ...`__exact`)
    - z.B. `Vorlesung.objects.filter(titel='ET')`

  – Wert-Relation (Zahlen oder Strings)

    - **`gt`** / **`gte`** / **`lt`** / **`lte`** (größer / größer-gleich / kleiner / kleiner-gleich)
      – z.B. `Person.objects.filter(iq__gte=120)`
    - **`in`** (ist Wert im Liste enthalten?)
      – z.B. `Wert.objects.filter(x__in=[1,2,3])`
    - **`range`** (liegt Wert in min-max-Bereich?)
      – z.B. `Wert.objects.filter(x__range=(1,3))`

Django: QuerySet-API

- **Filter-Kriterien** („Field-Lookups“)
  - Relationen: String-Vergleiche
    - `startswith` / `endswith` / `contains` / `exact`
      `istartswith` / `iendswith` / `icontains` / `iexact`
      - Enthält Sting (ggf. Case-insensitiv bei „i...“) den Wert?
    - `regex` / `iregex` (Matcht regulärer Ausdruck den Feldwert?)
  - Null-Test
    - `isnull` (ist der Wert SQL-NULL?)
      - z.B. `filter(owner__isnull=True)`
  - Filterkriterien über Relationen hinweg
    - Ist Attribut a eine Relation, so kann man mit a__b auf ein Attribut b der referenzierten Klasse zugreifen.
      - Das ist über mehrere Stufen möglich. Danach können Relationen kommen.
      - z.B. `Vorlesung.objects.filter(dozent__name='Wirth')`
      - z.B. `Vorlesung.objects.filter(dozent__name__startswith='W')`

Django: QuerySet-API

- **Filter-Kriterien**: Weiterführende Techniken
  - **F-Ausdrücke**: Attribute als Werte
    - Mit `models.F('attributname')` als Wert kann man auf andere Attribute Bezug nehmen
      - z.B. `Pers.objects.filter(gehalt__gt=F('vorgesetzte__gehalt'))`
        - Liefert Alle, die mehr verdienen als ihre jeweiligen Vorgesetzen
  - **Q-Ausdrücke**: Logische Verknüpfung von Bedingungen
    - Mit `models.Q(...)` kann man aus den übergebenen Filter-Krieterien **Q-Objekte** generieren, gewissermaßen *eingefrorene* Kriterien
      - z.B. `q = Q(name__startswith='Wi')`

        Nutzung z.B.: `Pers.objects.filter(q)`
    - Q-Objekte kann man logisch verknüpfen
      - „**&**“ (und), „**|**“ (oder), „**~**“ (not)

        Ergebnis ist jeweils wieder ein Q-Objekt
    - Beispiel:

```
– q1 = Q(name__startswith='Wi') &  Q(name__endswith='th')
– q2 = Q(name__startswith='Za') & ~Q(name__endswith='ze')
  Pers.objects.filter(q1 | q2)
```

    - *Verständnisfrage: Welche Kombinationen sind auch ohne Q-Objekt möglich?*

Django: QuerySet-API

- **QuerySet-Methoden, die keine QuerySets liefern**
  - **get**(...)
    - Liefert **genau ein** Objekt, dass die angegebenen Kriterien erfüllt
      - Existiert kein passendes, so wird eine DoesNotExist-Exception geworfen
      - Existieren mehrere, so wird eine MultipleObjectsReturned-Exception geworfen
    - Beispiel: `stud = Student.objects.get(matnr = 26120)`
  - **count**()
    - Liefert die Anzahl der Objekte im QuerySet
    - Beispiel: `studcount = Student.objects.all().count()`
  - **exists**()
    - Liefert True, wenn mindestens Objekt im Queryset existiert
    - Entspricht also `count()>0`, ist aber effizienter

Django: QuerySet-API

- **Modell-Objekt-Methoden**

  Die Modell-Instanzen haben zwei Methoden, um den Datensatz in der Datenbank zu erzeugen, zu aktualisieren und zu löschen.

  – Speichern eines neuen Objekts: `save()`

    - ```
      s = Student(matnr=26120, name='Peter')
      s.save()
      ```

  – Ändern eines existierenden Objektes: `save()`

    - ```
      s = Student.objects.get(matnr = 26120)
      s.name = 'Ben'
      s.save()
      ```

      – Unterschied zu oben: Hier ist `s.pk` vor den Aufruf von save() schon gesetzt

  – Löschen eines Objektes: `delete()`

    - ```
      s = Student.objects.get(matnr = 26120)
      s.delete()
      ```
      ```

Django: QuerySet-API

- **Manager-Methoden, die Objekte erzeugen**
  - **create**(...)
    - z.B. `s = Student.objects.create(matnr=1234, name='Peter')`
    - Identisch zu
      - `s = Student(matnr=1234, name='Peter')`
      - `s.save()`
  - **get_or_create**(... , defaults = {...})
    - Legt Objekt an, wenn es nicht schon existiert (anhand der vorderen Parameter)
    - Wenn es angelegt wird, werden die defaults-Werte zur Initialisierung genutzt
    - Rückgabewert ist das Paar (obj, created), wobei der Bool-Wert created anzeigt, ob das Objekt (obj) neu erzeugt wurde.
    - z.B. stud, was_created = `Student.objects.get_or_create(matnr=1234, defaults={'name': 'Peter'} )`
      - Der Name spielt also nur eine Rolle, wenn noch niemand mit der Matrikelnummer existiert (Verständnisfrage: Was wäre, wenn name direkt als Parameter übergeben worden wäre?)

Django: QuerySet-API

- **QuerySet-Methoden, die Objekte ändern**
  - **update**(...)
    - Alle Objekte im QuerySet werden aktualisiert wie angegeben
    - z.B. `Student.objects.filter(name='Ben').update(name='Bob')`
      - Alle Studenten mit dem Namen „Ben“ heißen danach „Bob“
    - Das ist effizienter als die Elemente einzeln zu aktualisieren

```
for stud in Student.objects.filter(name='Ben'):
    stud.name='Bob'
    stud.save()
```

    - Vorsicht bei solchen Massen-Updates! Was tut folgendes?
      `Student.objects.all().update(name='Bob')`

Django: QuerySet-API

- **QuerySet-Methoden, die Objekte ändern**
  - `delete()`
    - Alle Objekte im QuerySet werden gelöscht
    - z.B. `Student.objects.filter(matnr__in=[17,38,95]).delete()`
      - Alle Studenten mit den Matrikelnummern 17, 38 und 95 werden gelöscht.
    - z.B. `Student.objects.filter(matnr__lte=999).delete()`
      - Alle Studenten mit Matrikelnummern <= 999 werden gelöscht.
    - z.B. `Student.objects.filter(matnr=17).delete()`
      - „Alle“ Studenten mit den Matrikelnummern 17 werden gelöscht.
        - Hier nur ein Treffer möglich – warum?
    - Das ist effizienter als die Elemente einzeln zu löschen
      - ```
        for stud in Student.objects.filter(matnr__lte=999):
            stud.delete()
        ```
      - ```
        for stud in Student.objects.all():
            if stud.matnr <= 999:
                stud.delete()
        ```

Django: QuerySet-API

- **Mengenartige Operationen auf Relationen**
  - Ändern einer ManyToMany-Relation
    - ```
      s = Student.objects.get(matnr = 26120)
      v = Vorlesung.objects.get(titel = 'ET')
      ```
      ```
      s.hoert.add(v)       # eine Vorlesung mehr
      s.hoert.remove(v)    # eine weniger
      s.hoert.clear()      # gar keine Vorlesungen mehr
      s.hoert = [v]        # nur genau die diese Vorlesung
      ```
  - Ändern einer ForeignKey-Relation
    - Vom Foreign-Key-Attribut-Objekt aus („n:1"-Seite)
      - Objekt laden, Foreign-Key-Attribut zuweisen, speichern
      - Oder per update(...) des QuerySets
    - Von der Gegenseite aus („1:n"-Seite)
      - Analog zur ManyToMany-Relation:
      - ```
        p = Professor.objects.get(name='Wirth')
        p.vorlesung_set.add(v)
        ```

Django: QuerySet-API

- **ManyToMany-Relationen bei neu erzeugten Objekten**
  - Ein Auto-Increment-Primärschlüssel wird ggf. beim save gesetzt
    - z.B. der standardmäßig implizit gesetzte PK id
    - *Verständnisfrage*: Warum erst dann?
  - Wenn der PK (noch) nicht gesetzt ist, kann man z.B. keine ManyToMany-Beziehungen aufbauen
    - *Verständnisfrage*: Warum?
  - Folgende Beispiele scheitern also:
    - ```
      s = Student(matnr = 26120, name = 'Peter')     # neues Obj.
      v = Vorlesung.objects.get(titel = 'ET')
      s.hoert.add(v)
      ```
      Lösung: `s.save()`
    - ```
      p = Professor(matnr = 26120, name='Wirth')     # neues Obj.
      p.vorlesung_set.add(v)
      ```
      Lösung: `p.save()`

Django: QuerySet-API

- **Weitere QuerySet-Operationen und Eigenschaften**
  - Abruf eines Teils der QuerySet-Objekte
    - `Student.objects.all()[0:5]`
      - Ruft die ersten 5 Studenten ab (Index 0..4, gemäß jeweils gültiger Sortierung)
      - Entspricht der SQL-Option „LIMIT 5“
      - Wenn es weniger sind, entsteht kein Fehler
    - `Student.objects.all()[5:15]`
      - Ruft den 6. - 15. Studenten ab (Index 5..14, gemäß jeweils gültiger Sortierung)
      - Entspricht der SQL-Option „OFFSET 5 LIMIT 10“

Zur Erinnerung bzgl. Array-Bereichs-Zugriffen in Python:

```
>>> x = (0,1,2,3,4,5)
>>> x[3:5]
(3, 4)
```

Django: QuerySet-API

- **Low-Level-QuerySet-Operationen**
  - QuerySet-Methode **extra**(...)
    - Mit dieser Methode kann man bestimmte zusätzliche SQL-Bedingungen (select, where, order_by) in einen QuerySet integrieren.
    - Bei Nutzung expliziter Parameter-Übergabe sicher gegen SQL-Injections.
      - ```
        Entry.objects.extra(      where = ['headline=%s'],
                                  params= ['Lennon']     )
        ```
    - siehe https://docs.djangoproject.com/en/4.2/ref/models/querysets/#extra
  - Raw-SQL-Queries: Manager-Methode **raw**(...)
    - Mit dieser Methode kann man eine komplette SQL-Anfrage an die Datenbank senden. Die Verknüpfung mit anderen QuerySet-Methoden ist nur beim Abholen der Ergebnisse möglich.
    - Bei Nutzung expliziter Parameter-Übergabe sicher gegen SQL-Injections.
      - ```
        Entry.objects.raw( ['SELECT * FROM myapp_person
                      WHERE last_name = %s'], params=['Lennon']    )
        ```
    - siehe https://docs.djangoproject.com/en/4.2/topics/db/sql/

Django: QuerySet-API

- **Modell-Objekt-Methoden**
  - **`__str__()`**
    - Benutzerdefiniert, liefert eine Textdarstellung des Objekts
      - (In Django 2.x gab es dazu auch die Funktion `__unicode__()`)
  - **`get_XXX_display()`**
    - zu jedem Attribut mit Option „choices“ (s.o.) wird automatisch diese Methode (statt XXX den jeweiligen Attributnamen einsetzen) definiert.
    - Sie liefert den Klartext-Namen des aktuellen Coices-Wertes
  - Es gibt noch weitere Methoden, die wir später betrachten
    - `get_absolute_url()`
    - `clean()`, `clean_fields()`, `full_clean()`, `validate_unique()`

Django: Request-Handling

Zwischenstand: Wir wissen jetzt, wie man in Django …

- Ein Daten-Schema definiert
  - Modell-Klassen
- Daten abfragt
  - Query-Sets, filter, get, ...
- Daten ändert
  - save, update, delete

Nächstes Ziel: HTTP-Requests behandeln

- Request empfangen
- View-Methode bestimmen und aufrufen
- Response-Inhalt erzeugen
- Reponse abschicken

Django: Request-Handling

- **Request-Handling**

  Wird ein HTTP-Request an die Django-Webapplikation geschickt, laufen im wesentlichen folgende Schritte Ab:

1. Aus dem **HTTP-Request** (Header und Nutzlast) wird ein **Request-Objekt** erzeugt.
2. Die enthaltene **URL wird analysiert** und die (per „`*/urls.py`“) zugeordnete **View-Methode** und **Parameter** werden bestimmt
3. Die View-Methode (aus „`*/views.py`“) wird mit einem Request-Objekt und den o.g. Parametern **aufgerufen**
4. Die View-Methode verarbeitet den Request und liefert ein **Response-Objekt** als Ergebnis zurück
   - Die View-Methode kann alternativ auch mit einer Exception enden
5. Aus dem Response-Objekt werden Header und Nutzlast einer **HTTP-Response** gewonnen und per HTTP verschickt.

Django: Request-Handling

- **Request-Handling**
  - Vom Django-Nutzer müssen nur die URL-Regeln und die View-Methode definiert werden
  - Eine einfache **View-Methode** könnte so aussehen:
    - ```
      def show_dozenten_anzahl(request):
          count = Professor.objects.count()
          text = 'Wir haben %s Dozenten.' % count
          return HttpResponse(text)
      ```
    - Der Name der Funktion („show_dozenten_anzahl") ist frei gewählt.
    - Diesen Code schreiben wir in die Datei „test1/pruefungsamt/**views.py**"
  - Eine einfache URL-Regel könnte so aussehen:
    - ```
      urlpatterns = [
          path('dozenten_anzahl/', show_dozenten_anzahl),
      ]
      ```
    - Diesen Code schreiben wir in die URL-Datei „test1/test1/**urls.py**"
      - Zusätzlich brauchen wir: `from pruefungsamt.views import *`

Django: Request-Handling

- **Damit haben wir schon eine funktionierende Webseite**
  - Wenn man die URL

    `http://scilab-0100.cs.uni-kl.de:8000/dozenten_anzahl/`

    aufruft erhält man z.B. die Ausgabe

    „Wir haben 3 Professoren.“

  - Die übertragene Webseite ist aber keine valide HTML-Seite
    - nur ein Text ohne Tags und Struktur

  - Wir könnten zumindest zugeben, dass es kein HTML ist ….
    - ```
      def show_dozenten_anzahl(request):
          count = Professor.objects.count()
          text = 'Wir haben %s Dozenten.' % count
          return HttpResponse(text, content_type="text/plain")
      ```

    - Das war aber ja nicht was wir eigentlich wollten: **HTML**
      ```

Django: Request-Handling

- **Hintergrund: HttpRequest-Objekte**
  - Request-Objekte enthalten alle Informationen zum HTTP-Request
    - **path** URL-Pfad (z.B. „/dozenten_anzahl/“)
    - **method** „GET“ oder „POST“
    - **GET**
POST Die übergebenen GET- / POST-Parameter als Dictionary
    - **COOKIES** Cookies als assoziatives Array
    - **session** Ein lesbares und schreibbares assoziatives Array
      - Session-Daten können direkt zugewiesen werden
    - **user** Eingeloggter User (Objekt) oder AnonymousUser
      - Test z.B. über request.user.**is_authenticated**
    - **META** Dictionary mit Metadaten
      - z.B. HttpRequest.META['HTTP_REFERER']

  - siehe https://docs.djangoproject.com/en/4.2/ref/request-response/#httprequest-objects

Django: Request-Handling

- **Hintergrund**: **HttpResponse-Objekte**
  - Response-Objekte enthalten alle Informationen zum HTTP-Response. Daraus wird am Ende der Response als Text (Header + Payload)
    - **__init__**(content="", mimetype=None, status=200, content_type=DEFAULT_CONTENT_TYPE)
      - Konstruktor, content ist die (initiale) Payload
    - **status_code** Status-Code (lesbar/schreibbar)
    - **content** Payload (lesbar/schreibbar)
    - **__setitem__**(header, value) Header-Wert setzen
      __getitem__(header) Header-Wert lesen
      __delitem__(header) Header-Wert entfernen
    - **write**(content) Das Objekt kann wie eine Datei beschreiben werden
    - **set_cookie**(key, value="", max_age=None, expires=None, path="/", domain=None, secure=None, httponly=False)
    - **set_signed_cookie**(...)
      - Weitegehend wie set_cookie, Werte aber **kryptographisch signiert** (Diskussion)
  - siehe https://docs.djangoproject.com/en/4.2/ref/request-response/#httpresponse-objects

Django: Request-Handling

- **Hintergrund**: **HttpResponse-Objekt-Varianten**
  - Es gibt einige Spezialisierungen der HttpResponse-Klasse, die in erster Linie des Default-Status-Code ändern:
    - **HttpResponseNotFound**
      - 404-Response (Objekt / Seite nicht gefunden)
    - **HttpResponseForbidden**
      - 403-Response (Zugriff nicht erlaubt)
    - **HttpResponseServerError**
      - 500-Response (interner Fehler des Servers)
    - **HttpResponseRedirect**(url) bzw. **HttpResponsePermanentRedirect**(url)
      - 302 bzw. 301-Response mit angegebenem Umleitungsziel
      - Der Client wird auf die angegebene Seite umgeleitet (→ url in **Location-Header**)

Entspricht Setzen des **status**-Feldes in HttpResponse

  - Es gibt weitere HttpResponse-Varianten für komplexere Aufgaben
    - **FileResponse** → liefert Datei als Nutzlast zurück
    - **JsonResponse** → überträgt Datenstruktur als JSON-Nutzlast

Django: Request-Handling

- **HTML-Nutzlast in der Response** (Primitiver Ansatz)
  - Wir bauen schrittweise HTML auf:
    - ```
      def show_dozenten_anzahl(request):
          count = Professor.objects.count()
          html  = '<!DOCTYPE html><html><body>'
          html += 'Wir haben %s Dozenten.' % count
          html += '</body></html>'
          return HttpResponse(html)
      ```
  - Oder, um die Aufgaben der Strings klarer zu gliedern:
    - ```
      def show_dozenten_anzahl(request):
          count = Professor.objects.count()
          content = 'Wir haben %s Dozenten.' % count
          template = '<!DOCTYPE html><html><body>%s</body></html>'
          return HttpResponse(template % content)
      ```
    - Wir haben also ein **Muster** (engl. „*Template*“) der HTML-Seite, in die wir Inhalte einbauen.

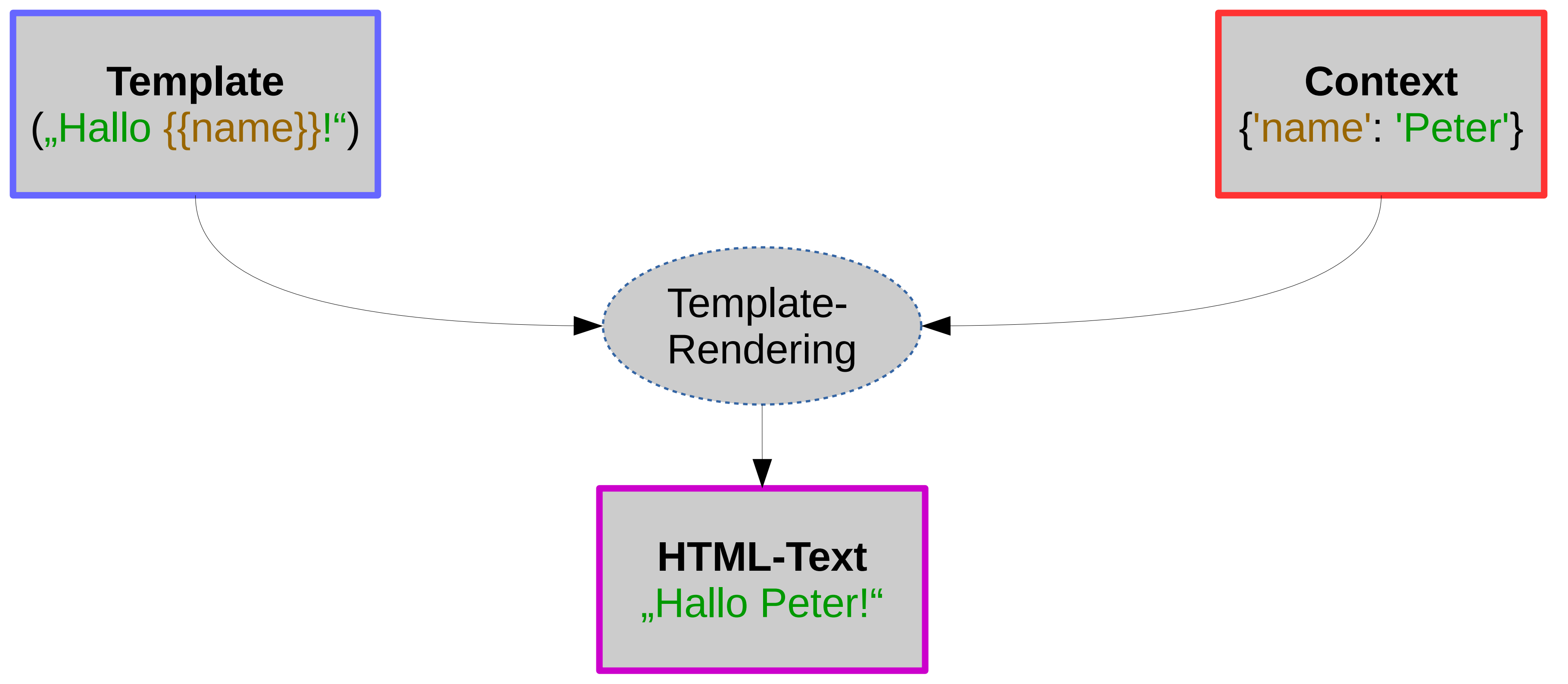
Django: Request-Handling

- **Das obige Verfahren zur HTML-Erzeugung funktioniert**
  - Es ist aber bald **unhandlich**
    - Komplexe HTML-Seiten werden so sehr unübersichtlich
    - Wir wollten doch Kontroll-Logik (View-Methode) und Darstellung (HTML) voneinender trennen …

- **Lösung: Template-Engine**
  - Django enthält zur Erzeugung der HTML-Seiten eine Template-Engine, die
    - von der **View-Methode** aufgerufen wird und von ihr **Daten** erhält,
    - die aus statischen Dateien (**Template-Dateien**) HTML-Templates liest
    - **Daten und Templates verknüpft** („**Rendering**“) und als Text zurück liefert
  - Überblick: https://docs.djangoproject.com/en/4.2/topics/templates/
    - Die Template-Engine ist **erweiterbar** und **nicht** auf HTML **spezialisiert**

Django: Template-Engine

- **Idee: Template + Context → HTML-Text**
  - **Template**: Muster der Darstellung (HTML-Muster)
  - **Context**: Daten die darin vorkommen können



Django: Template-Engine

Nur **Grundprinzip** – Normalerweise nutzen wir den vereinfachten **Shortcut** (s.u.)

- **Aufruf eines Templates** (Grundprinzip)
  - Wir bilden die obige View nun mit Templates nach:
    - ```
      from django.template import Context, loader

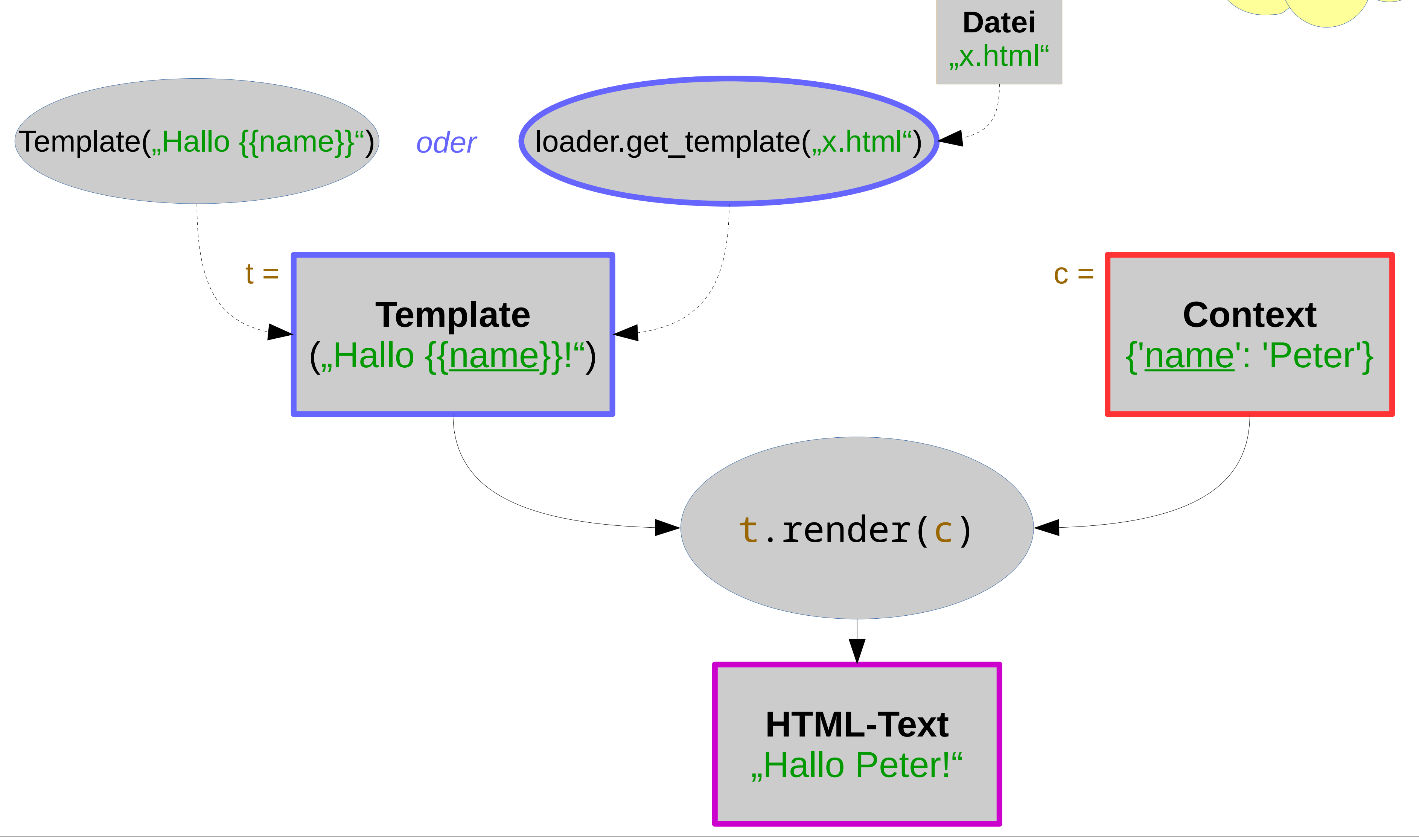
      def show_dozenten_anzahl(request):
          context = Context({
                'prof_count': Professor.objects.count(),
             })
          template =loader.get_template('dozenten_anzahl.html')
          return HttpResponse(template.render(context))
      ```
  - Was passiert hier?
    - In dem **Context**-Objekt **context** werden die anzuzeigenden Daten abgelegt
      - Sie werden dem Konstruktor als **Dictionary** übergeben
    - In **template** wird das **Template** aus Datei 'dozenten_anzahl.html' geladen
    - Dann wird `template.render(context)` aufgerufen.
      Dabei werden Template und Daten verknüpft.
      Ergebnis ist der **HTML-Text**, der als Response-Nutzlast zurückgeliefert wird.
      ```

Django: Template-Engine

- **Wie sieht das Template aus?**
  - Wir legen die Template-Datei in 'test1/pruefungsamt/**templates**/**dozenten_anzahl.html**' an
    - ```
      <!DOCTYPE html>
      <html>
        <body>
          Wir haben {{ prof_count }} Dozenten.
        </body>
      </html>
      ```
  - Die Template-Datei sieht aus wie eine normale HTML-Datei
  - Lediglich an der Stelle, an der die Daten eingebaut werden sollen, wird mit „`{{prof_count}}`“ auf den **übergebenen Kontext-Wert** Bezug genommen.
- **Das Ergebnis des Renderings ist wie erwartet**
  - Das Template ist aber viel übersichtlicher als zuvor.
      ```

Django: Template-Engine

- **Datenfluss der Template-Verarbeitung**



Django: Template-Engine

- **Rendering-Shortcut** (praktische Lösung)
  - Da fast alle View-Funktionen mit einem Template-Rendering enden, gibt es ein **Shortcut**, das das erleichtert.
  - Beispiel: Diese Views bewirken das Selbe:
    - ```
      from django.shortcuts import render

      def show_dozenten_anzahl(request):
          d = {'prof_count': Professor.objects.count(), }
          return render(request, 'dozenten_anzahl.html', d)
      ```
    - ```
      from django.template import Context, loader
      from django.http     import HttpResponse

      def show_dozenten_anzahl(request):
          d = {'prof_count': Professor.objects.count(), }
          context  = Context(d)
          template = loader.get_template('dozenten_anzahl.html')
          return HttpResponse(template.render(context, request))
      ```

Django: Template-Engine

- **Die Template-Engine kann mehr als Variablen einfügen**
  - Wir können z.B. Blöcke nur unter bestimmten Bedingungen ausgeben
    - ```
      <!DOCTYPE html>
      <html><body>
      Wir haben
      {% if prof_count == 0 %}
          keine
      {% else %}
          {{ prof_count }}
      {% endif %}
      Dozenten.
      </body></html>
      ```

      ⇒

      ```
      <!DOCTYPE html>
        <html><body>
          Wir haben
                3
          Dozenten.
       </body></html>
      ```
  - Die Textstruktur ist beliebig
    - u.a. ist keine Einrückung erforderlich (aber sie ist hilfreich beim Code-Lesen)
    - Leerzeichen und Einrückungen sind Teil der (HTML-Text) Ausgabe!
      ```

Django: Template-Engine

- **Von der Template-Engine interpretierte Elemente:**
  - **Ausdrücke**: „**{{ … }}**“
    - Ausdrücke sind z.B. die Namen der übergebenen Kontext-Variablen
    - Beispiel von oben: `{{ prof_count }}`
  - **Tags**: „**{% … %}**“
    - Tags sind strukturierende Elemente, die meist Text und andere Tags umfassen
    - Beispiele: if-else-endif, for-Schleifen, etc.
  - **Kommentare**: „**{# … #}**“
    - Kommentare werden ausgefiltert. Sie müssen in der selben Zeile enden.
    - Mehrzeilige Kommentare sind mit dem **comment**-Tag realiserbar

Django: Template-Engine

- **Tags können Klammern bilden** oder auch nicht:

- **Manche Tags stehen alleine für sich**
  - sie bilden **keine Klammern**
  - z.B. `{% url 'impressum' %}`, das eine URL ausgibt

- **Manche Tags bilden Klammern**
  - z.B. `{% if x==1 %} x is one {% endif %}`
  - Die Tags bilden meist Paare der Form **name** → **endname**
    - z.B. `{% comment %} … {% endcomment %}`
  - Die Klammerung kann aber auch komplexer und mehrstufig sein
    - z.B. `{% if a %} … {% elif b %} … {% else %} … {% endif %}`

Django: Template-Engine

- **Tags können (mehrere) Parameter haben**
  - z.B. `{% if prof_count %}Wir haben ...{% endif %}`

- **Tag-Parameterlisten können dabei auch feste Schlüsselwörter enthalten**
  - z.B. `{% for x in some_list %} … {{x}} … {% endfor %}`

- **Tags können auch *neue* Variablen definieren**
  - z.B. Schleifenvariablen oder „`forloop.` ...“ bei For-Schleifen
  - ```
    <ul>
      {% for x in some_list %}
        <li> Element Nr. {{forloop.counter}}: {{x}}
      {% endfor %}
    </ul>
    ```
    ```

Django: Template-Engine

- **Ausdrücke in Template-Tags**
  - Das Ergebnis des Ausdrucks in „`{{ … }}`“ *(z.B. der Variablen)* wird in den Ausgabe-Text eingefügt (und ggf. vorher escapt)
  - Als Ausdrücke können u.a. verwendet werden ...
    - übergebene Kontext-Daten „`{{prof_count}}`“
    - die von Tags erzeugten lokalen Variablen (z.B. Schleifenvariablen, s.o.)
    - Variablen können auf Objekt-Komponenten und -Methoden zugreifen z.B. wenn stud ein QuerySet ist: „`{{stud.count}}`“
  - Variablenwerte können mit **Filtern** modifiziert werden.
    - z.B. für name=“Test“ erzeugt „`{{name|upper}}`“ den Text „TEST“
    - Filter können auch kaskadiert werden: „`{{name|upper|linebr}}`“
  - An manche Filter kann auch ein **Parameter** übergeben werden
    - „`{{name|default:"N.N."|upper}}`“
    - Parameter können Variablennamen oder Stringliterale (in '...' oder "...") sein

Keine „()“ Klammern beim Methoden-aufruf!

Django: Template-Engine

- **Feature: Automatisches Escaping**
  - Django **escapt alle Variablen-Ausgaben** in der Template-Engine per Default.
    - Versucht ein Angreifer eine Injection über eine Eingabe, die z.B. HTML-Text enthält, so werden die zuverlässig vor Interpretation geschützt.
    - Die Zeichen <, >, ' (single quote), " (double quote) und & werden in ihre entsprechenden html-Zeichencodes umgewandelt.
    - Die Verwendung von `{{ evil_data }}` ist also sicher.
  - Das Escaping kann **gezielt abgeschaltet** werden
    - Mit dem Filter „**safe**": z.B. in `{{ good_data|safe }}` wird der Inhalt von data unverändert ausgegegeben (*Verständnisfrage: Wozu braucht man das?*)
    - In größeren Blöcken kann das Autoescaping abeschaltet werden.
      Einzelne Ausgaben können dann wieder mit dem Filter „escape" geschützt werden:

```
{% autoescape off %}
    Hallo {{good_data}}, du sagtest {{evil_data|escape}}
{% endautoescape %}
```

Django: Template-Engine

- **Feature: Vererbung von Templates**
  - HTML-Seiten innerhalb einer Website unterscheiden sich abseits des Inhalts meist nur wenig voneinander (einheitliches Grunddesign)
  - Die Templates bieten daher die Möglichkeit zur **Vererbung**
- **Grundidee**
  - Man definiert ein Basis-Template und markiert (benannt) alle Punkte, an denen die spezielleren Templates Änderungen machen können „`block`"-Tag
  - Die speziellernen Templates beerben ein Basis-Template („`extends`"-Tag) und modifizieren die Blöcke
    - Sie können dabei auf auch die geerbten Block-Inhalte zugreifen („block.super"-Variable)

  - https://docs.djangoproject.com/en/4.2/topics/templates/#template-inheritance

Django: Template-Engine

- **Vererbung von Templates**

basis.html

```
<!DOCTYPE html>
<html><head>
  <title>
    {% block title %}
        No Title
    {% endblock %}
    </title>
</head>
<body>
    <div id=content>
    {% block content %}
        No Content
    {% endblock %}
    </div>
</body>
</html>
```

special.html

```
{% extends "basis.html" %}

{% block title %}
    My Special Title
{% endblock %}

{% block content %}
    Bla
    Bla
    Bla
{% endblock %}
```

Django: Template-Engine

- **Template „**basis.html**“**

```
<!DOCTYPE html>
<html><head>
  <title>{% block title %}No Title{% endblock %}</title>
</head><body>
  <div id=content>
    <h1>{% block heading %}No Heading{% endblock %}</h1>
    {% block content %}This page has no content.{% endblock %}
  </div>
</body></html>
```

- **Template „**special.html**“**

```
{% extends "basis.html" %}
{% block title %}My Special Page{% endblock %}
{% block heading %}About my very special page{% endblock %}
{% block content %}
  Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod
  tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. ...
{% endblock %}
```

Django: Template-Engine

- **Template „basis.html“**

```
<!-- … -->
<menu>
  <ul>
  {% block menuitems %}
    <li>Home
    <li>About
  {% endblock %}
  </ul>
</menu>
<!-- … -->
```

- **Template „special.html“**

```
{% extends "basis.html" %}
{% block menuitems %}
    <li>First
    {{ block.super }}
    <li>Last
{% endblock %}
```

- Template „special.html“ entspricht:

```
<!-- … -->
<menu>
  <ul>
  {% block menuitems %}
    <li>First
    <li>Home
    <li>About
    <li>Last
  {% endblock %}
  </ul>
</menu>
<!-- … -->
```

Django: Template-Engine

- **Wo sucht Django nach Templates?**
  - In settings.py wird dazu eine Einstellung `TEMPLATES` gemacht.
  - *Default:*

```
TEMPLATES = [
  {
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'DIRS': [],
    'APP_DIRS': True,
    'OPTIONS': {
       'context_processors': [
          'django.template.context_processors.debug',
          'django.template.context_processors.request',
          'django.contrib.auth.context_processors.auth',
          'django.contrib.messages.context_processors.messages',
  ], }, }, ]
```

  - Die Voreinstellung `APP_DIRS` auf True sorgt dafür, dass in den Verzeichnissen „`template`" in der jeweiligen App gesucht wird.
  - Man sollte `DIRS` zusätzlich auf `['templates']` setzen.
    - Dadurch wird auch in dem Verzeichnis **`templates`** auf Projektebene (also parallel zu den App-Verzeichnissen) gesucht.
    - Hier kann man die **Basis-Templates** zentral ablegen.

Django: Template-Engine

- **Wie greift man auf statische Dateien zu?**
  - Grafik-Dateien, CSS-Dateien, etc. haben eine URL und einen Dateisystem-Pfad
  - Ihre **URL** ist in `STATIC_URL` in settings.py festgelegt. (Default: '/static/')
  - Auf die URL kann man mit dem Template-Tag `{% static RELATIVER_PFAD %}` zugreifen.
    - Vorher ist einmalig ein `{% load static %}` erforderlich.
  - Beispiel:

```
{% load static %}
<html>
 <head>
   <link rel=stylesheet href="{% static 'css/styles.css' %}">
```

Template-rendering

```
<html>
 <head>
   <link rel=stylesheet href="/static/css/styles.css">
```

Django: Template-Engine

- **Wo sucht Django nach statischen Dateien?**
  - Standardmäßig sind zwei File-Finders aktiviert:
    - AppDirectoriesFinder (sucht im Verzeichnis 'static' in jedem App-Verzeichnis)
    - FileSystemFinder (sucht in den Verzeichnissen aus STATICFILES_DIRS)
      - Default für ist STATICFILES_DIRS ist `[ ]`, also nirgends suchen
- **Global benötigte Dateien (z.B. für Basis-Templates)**
  - sollten zentral direkt im Projektverzeichnis unter „static/“ liegen
    - Ergänzen in `settings.py`: `STATICFILES_DIRS = [ 'static/' ]`
  - Dadurch wird bei `{% static 'css/styles.css' %}` gesucht in …
    - `PROJEKTVERZEICHNIS/static/css/styles.css`
    - `PROJEKTVERZEICHNIS/APPVERZEICHNIS/static/css/styles.css`
      für alle App-Verzeichnisse im Projekt.

Tipp: Mit manage.py kann man prüfen, wo eine Datei gefunden wird

- `./manage.py findstatic css/styles.css`

Django: Template-Engine

- **Vorführung**
  - Wir modifizieren jetzt eine der Design-Beispiele aus dem letzten Semester zu einem Basis-Template.

Django: Template-Engine

- **Django enthält viele weitere Tags und Filter**
  - Wir schauen uns jetzt die wichtigsten an:

    https://docs.djangoproject.com/en/4.2/ref/templates/builtins/

- **Ausblick:** Man kann die Template-Engine auch um eigene Tags und Filter ergänzen.
  - Diese können ggf. komplexe Aufgabe realisieren, z.B.
    - Daten aus der Datenbank holen
    - einen größeren Datensatz ausgeben
    - Texte umwandeln, übersetzen, ...
  - Es gibt auch vorgefertigte **Zusatz-Tags und Filter**
  - https://docs.djangoproject.com/en/4.2/howto/custom-template-tags/

Django: Request-Handling

- **Zurück zur Request-Verarbeitung**
  - Wir können mittlerweile Request-URLs auf Views mappen
  - Wir können auf GET- und POST-Parameter und Cookies zugreifen und sie in der View nutzen
- **Der URL-Dispatcher kann aber mehr ...**
  - Der URL-Dispatcher kann auch die URL nach unseren Regeln zerlegen und daraus View-Parameter extrahieren
    - Ziel: Sprechende URLs
    - z.B. `http://scilab-0123.cs.uni-kl.de:1234/professor/2/`
      - zum Zugriff auf Dozenten mit der id 2
      - Ziel nun: **id**=2 soll an die View übergeben werden
  - Dazu brauchen wir **Pfadangaben**, gegen die die URL-Pfade getestet werden.
    - z.B. `'professor/<int:id>/'` zur obigen URL

Django: URL-Parser

- **Semantische URLs: Argument-Übergabe als Pfad-Element**
  - z.B. URL-Pfade '/articles/2022/12/' oder '/mod/INF-00-31-M-3/'
- **Idee: Definition von „Pfad-Mustern“ in urls.py**

```
from django.conf.urls import path, include
import news.views

urlpatterns = [
   path('articles/<int:year>/',              news.views.year_archive ),
   path('articles/<int:year>/<int:month>/', news.views.month_archive),
   path('article/id-<int:id>/',              news.views.article_detail),
]
```

  - Siehe auch https://docs.djangoproject.com/en/4.2/topics/http/urls/#example
  - Wir geben jeweils in dem Pfad-Muster einen **Typ** (z.B. „int“) und einen **Namen** (z.B. „year“) an.
    - Die Typen („*Converter*“) definieren zulässige Zeichenketten („**int**“: Zahlen)
    - Der Name dient zur Übergabe an die View-Methode (z.B. „month_archive“)

Django: URL-Parser

- **Anwendung von Pfad-Mustern in urls.py (1)**

```
urlpatterns = [  # ...
    path('articles/<int:year>/<int:month>/',    news.views.month_archive),
]
```

  - Beim Zugriff der URL '/articles/2022/12/' resultiert folgender Aufruf:

```
news.views.month_archive(request, year=2022, month=12)
```

  - Die entsprechende View-Methode könnte so aussehen:

```
def month_archive(request, year, month):
    d = {
        'articles': Article.objects.filter(year=year, month=month),
    }
    return render(request, 'month_archive.html', d)
```

  - Und das Template „month_archive.html“ könnte im Kern so aussehen:

```
{% for article in articles %}
   <article>
      <h2>{{ article.title }}</h2>
      <div class=summary>{{ article.summary }}</div>
   </article>
{% endfor %}
```

Die for-Schleife durchläuft die Ergebnisse des Querysets articles.

Django: URL-Parser

- **Anwendung von Pfad-Mustern in urls.py (2)**

```
Urlpatterns = [  # ...
    path('article/id-<int:id>/', news.views.article_detail),
]
```

  - Beim Zugriff der URL '/article/id-123/' resultiert folgender Aufruf:

```
news.views.article_detail(request, id=123)
```

  - Die entsprechende View-Methode könnte so aussehen:

```
def article_detail(request, id):
    d = {
        'article': Article.objects.get(id=id),
    }
    return render(request, 'article_detail.html', d)
```

  - Und das Template „article_detail.html“ könnte im Kern so aussehen:

```
<article>
   <h2>{{ article.title }}</h2>
   <div class=summary>{{ article.summary }}</div>
   <div class=text>{{ article.text }}</div>
</article>
```

Hier nur ein Modell-Element article (daher keine Schleife)

Django: URL-Parser

- **Es gibt noch weitere Parameter-Typen** *(„Converter“)*
  - z.B. **str** (matcht Zeichenketten ohne „/“)
    - Beim Zugriff der URL '/mod/INF-00-31-M-3/' soll folgender Aufruf erfolgen:

      `modules.views.mod_detail(request, modnr='INF-00-31-M-3')`
    - Die Pfad-Regel müsste also so aussehen:

```
urlpatterns = [  # ...
    path('mod/<str:modnr>/', modules.views.mod_detail),
]
```

    - *Übung*: Warum ohne „/“? Definieren Sie mod_detail und ein Template.
  - Es gibt noch einige weitere Parameter-Typen
    - Die aber seltener gebraucht werden (slug, uuid, path)
    - Siehe https://docs.djangoproject.com/en/4.2/topics/http/urls/#path-converters
  - Man kann auch eigene Parameter-Typen definieren oder mit re_path präzisere Formatangaben definieren
    - Für beides brauchen wir **Reguläre Ausdrücke** (→ später)

Django: URL-Parser

- **Namen und Zusätzliche Parameter**
  - Der Path-Aufruf kennt zwei weitere(optionale) Argumente:
    `path(pattern, view, kwargs=None, name=None)`
  - Mit `name` kann man einen **Namen** für das URL-Pattern angeben
  - Mit `kwargs` kann man **zusätzliche Parameter** an die view übergeben:
    - ```
      urlpatterns = [
        path('vorlesung/<int:id>/',     views.show_vl, {'ext':False}),
        path('vorlesung/<int:id>/ext/', views.show_vl, {'ext':True}),
      ]
      ```
  - Dadurch kann man zusätzliche Parameter an die View übergeben:
    - Aufruf von URL-Pfad `/vorlesung/123/`
      → Aufruf `views.show_vl(request, id=123, ext=False)`
    - Aufruf von URL-Pfad `/vorlesung/123/ext/`
      → Aufruf `views.show_vl(request, id=123, ext=True)`
  - So kann die selbe View-Funktion **mehrmals** verwendet werden
    - Parameter ext steuert hier die Erzeugung einer Langfassung der Webseite
      ```

Django: URL-Parser

- **URL-Patterns können andere Pattern-Dateien einbetten**
  - Die projektweite urls.py referenziert meist App-lokale Regeln
    - Nehmen wir an, die Projektweite urls.py hat folgenden Inhalt:

```
from django.conf.urls import path, include
from django.contrib   import admin

urlpatterns = [
    path('pa/',     include('pruefungsamt.urls')),
    path('admin/',  admin.site.urls),
]
```

  - Die App-lokalen urls.py werden unter dem Pfad-Präfix eingebunden
    - Nehmen wir an, `pruefungsamt/urls.py` hat folgenden Inhalt:

```
urlpatterns = [
    path('professor/<int:id>/', views.show_prof ) ),
    path('professoren/',        views.list_profs) ),
]
```

    - Die App-lokalen Regeln behandeln nur noch auf den **Rest-Pfad** (hier ohne `'pa/'`)
    - Aufruf von URL-Pfad `/pa/professor/123/`

      → Aufruf `pruefungsamt.views.show_prof(request, id=123)`

Django: URL-Parser

- **Wir können so also URL-Pfade analysieren**
  - Wir erhalten aufzurufende Views und ggf. Parameter
  - Man kann das auch manuell aufrufen:
    - Wir gehen vom Beispiel auf der vorherigen Folie aus:

```
from django.core.urlresolvers import resolve
func, args, kwargs = resolve('/pa/professor/123/')

print([func, args, kwargs])
[ <function pruefungsamt.views.show_prof>, [], {'id':123} ]
```

  - Der Django-Server analysiert URL-Pfade eingehender Requests automatisch ...
    - und ruft die ermittelte View-Funktion mit den Parametern auf:
      - `func(request, *args, **kwargs)`
    - also konkret:
      - `pruefungsamt.views.show_prof(request, id=123)`
    - Der explizite Aufruf von resolve() ist nur selten nötig.

Django: URL-Parser

- **Auch der Rückweg ist möglich: URL-Synthese**
  - Wir geben eine View-Funktion (deren Name) und die Parameter vor und erhalten von der Funktion **reverse** einen URL-Pfad
    - ```
      from django.urls import reverse
      url = reverse(funcname, args=None, kwargs=None)
      ```
  - Wie zu erwarten liefert der Aufruf mit obigen Werten die Ausgangs-URL
    - ```
      reverse('show_prof', kwargs={'id': 123,} )
      ```
      → `/pa/professor/123/`
  - Statt des Funktionsnamen können wir auch übergeben ...
    - … die Funktion (Funktions-Referenz)
    - … den Namen der URL-Regel, die wir umkehren wollen (s.o.)
      - Das ist nützlich, wenn wir mehrere Pfade (und URL-Regeln) haben, die die selbe View-Funktion aufrufen, und eine bestimmte davon haben wollen.

Django: URL-Parser

- **URL-Synthese: *Wozu brauchen wir das?***
  - Hat ein Modell-Objekt eine „*Homepage*“, so kann man deren Pfad über die **Modell-Methode** **get_absolute_url()** angeben

```
class Professor(models.Model):
    persnr = models.IntegerField(max_length=10, unique=True)
    name   = models.CharField(max_length=64)

    def get_absolute_url(self):
        return reverse('show_prof', kwargs={'id': self.id,} )
```

    - Dadurch wird u.a. auf der *Objekt-Editier-Seite* des **Admin-Interface** ein Link „View on site“ zur angegebenen URL des Objekts angezeigt.
    - In Templates kann man diese URL ebenso verwenden:
      `<a href='{{ prof.get_absolute_url|urlencode }}'>{{ prof.name }}</a>`
  - Das **Template-Tag** {% **url** %} kann reverse-Lookups ausführen
    - Muster: `{% url name.of.view v1 v2 arg3=v3 arg4=v4 %}`
    - Analog zum obigen Beispiel:
      `<a href='{% url 'show_prof' id=prof.id %}'>{{prof}}</a>`

Django: Anwendungs-Beispiel

- **Beispiel: Basis-Template** (templates/base.html)

```
{% load static %}

<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}Example-Uni{% endblock %}</title>
</head>
<body>
    <div style="float:right;">
        <img src="{% static 'img/logo.png' %}"></div>
    <div style="background: #afa;">
      {% block menu %}
        [<a href="/">Home</a>]
      {% endblock %}
    </div>

    {% block content %}
      This page is under development.
    {% endblock %}
</body>
</html>
```

Ermöglicht Template-Tag static (s.u.)

Liefert URL-Pfad für statische Dateien

Django: Anwendungs-Beispiel

- **Beispiel: VL-Template** (pruefungsamt/templates/vl.html)

```
{% extends "base.html" %} {# vl.html #}

{% block title %}
  Vorlesung {{ vl }}
{% endblock %}

{% block menu %}
  {{ block.super }}
  [<a href="{% url 'list_vls' %}">Liste aller Vorl.</a>]
{% endblock %}

{% block content %}
  <h1>Vorlesung {{ vl }}</h1>
  <table border=1>
    <tr> <th>Nummer: <td>{{ vl.vorlnr }}
    <tr> <th>Name:   <td>{{ vl.titel }}
    <tr> <th>Dozent: <td>
        <a href="{{ vl.dozent.get_absolute_url }}">{{ vl.dozent }}</a>
  </table>
{% endblock %}
```

Django: Anwendungs-Beispiel

- **Beispiel: Globale Projekt-URL-Regeln** (test1/urls.py)

```
from django.urls import include, path
from django.contrib import admin
import pruefungsamt, test1.views

urlpatterns = [
    path('',              test1.views.homepage             ),
    path('pa/',           include(pruefungsamt.urls)       ),
    path('admin/',        include(admin.site.urls)         ),
]
```

- **Beispiel: App-URL-Regeln** (pruefungsamt/urls.py)

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

urlpatterns = [
    path('professor/<int:id>/',         views.show_prof, name='show_prof'),
    path('professoren/',                views.list_profs,name='list_profs'),
    path('vorlesung/<int:id>/',         views.show_vl,   name='show_vl'),
    path('vorlesungen/',                views.list_vls,  name='list_vls'),
]
```

Django: Anwendungs-Beispiel

- **Beispiel: VL-View** (aus pruefungsamt/views.py)

```
from django.http import HttpResponse
from django.shortcuts import render
from django.http import Http404
from models import *

# ...

def list_vls(request):
    # Zeige Liste aller Vorlesungen an
    vls = Vorlesung.objects.all()
    return render(request, 'vls_list.html', dict(vls=vls) )


def show_vl(request, id):
    # Zeige die Vorlesung mit der id an
    try:
        vl = Vorlesung.objects.get(id=id)
    except Vorlesung.DoesNotExist:
        raise Http404
    return render(request, 'vl.html', dict(vl=vl) )
```

Falls das Objekt nicht existiert ...

erzeuge Fehlerseite 404 (Not Found)

Django: Anwendungs-Beispiel

- **Beispiel: VL-Template** (aus pruefungsamt/templates/vls_list.html)
  - Zugriff auf **übergebene** Query-Sets `(vl)`

```
<table>
     {% for vl in vls %}
       <tr>
       <td><a href="{% url 'show_vl' vl.id %}">...
       <td>{{ vl.vorlnr }}
       <td><a href="{{ vl.dozent.get_absolute_url }}">...
     {% endfor %}
</table>
```

- **Beispiel: Prof-Template** (aus pruefungsamt/templates/prof.html)
  - Zugriff auf **indirekt** erreichbare Query-Sets `(prof.vorlesung_set)`

```
<h2>{{ prof.vorlesung_set.all.count }} Vorlesungen</h2>
<ul>
  {% for vl in prof.vorlesung_set.all %}
      <li><a href="{{ vl.get_absolute_url }}">{{ vl }}</a>
  {% empty %}
      <li>Keine Vorlesungen
  {% endfor %}
</ul>
```

Django: Formulare

- **Modell-Formulare**
  - Django unterstützt auch die Formular-Verarbeitung
    - Man kann z.B. Formular-Klassen ähnlich wie die Modell-Klassen aus einzelnen Feldern zusammen stellen
    - Oft werden aber gerade **Modell-Objekte** in **Formularen** bearbeitet
  - Idee: Wir erzeugen aus dem Modell auch HTML-Formulare
    - Zu einer gegebenen Formular-Klasse (z.B. Vorlesung) erzeugen wir nun in views.py eine Model-Form-Klasse (VorlesungForm):

```
from django.forms import ModelForm
from pruefungsamt.models import Vorlesung

class VorlesungForm(ModelForm):
     class Meta:
          model  = Vorlesung
          fields = ('titel', 'dozent',)
          # exclude = ()
```

    - Mit den Meta-Attributen „fields“ und „exclude“ kann man angeben, welche Attribute editierbar sein sollen (default: alle).
      - Hier sind nur Titel und Dozent editierbar

- **Modell-Formulare: Instanziierung**
  - Erzeugt man eine Instanz dieser Klasse und gibt sie als String aus, erhält man ein HTML-Formular:

```
from pruefungsamt.views import *
form = VorlesungForm()
print(str(form))
```

  - Erzeugt die Ausgabe:

```
<tr><th><label for="id_titel">Titel:</label>
    <td><input id="id_titel" type="text" name="titel" maxlength="128" />
<tr><th><label for="id_dozent">Dozent:</label>
    <td><select name="dozent" id="id_dozent">
                <option value="" selected="selected">---------</option>
                <option value="2">Tesla [15]</option>
                <option value="3">Urlauber [20]</option>
                <option value="1">Wirth [12]</option>
           </select>
```

  - Das Formular ist offensichtlich dafür gedacht, in einer Tabelle ausgegeben zu werden.
  - Es setzt auch die Beschränkungen aus dem Modell um
    - Feldlängen, verfügbare Foreign-Key-Ziele

Django: Formulare

- **Modell-Formulare: HTML-Template**
  - Wir legen nun ein neues Template „vl_edit.html“ an, das ein Formular (Variable **form**) in eine Tabelle ausgibt.

```
{% block content %}
  <h1>Vorlesung {{ vl }} bearbeiten</h1>
  <form method=POST action="">
     <table>
         {{ form }}
         <tr><th><td><input type="submit" value="Speichern" />
         {% csrf_token %}
     </table>
  </form>
{% endblock %}
```

- Formular-Methode ist POST und Ziel ist die Ursprungs-URL („**Postback**“)
- Wir fügen noch einen Submit-Button hinzu
  - damit wir das Formular später abschicken können.
- Wir ergänzen im Formular das Tag „`{% csrf_token %}`“
  - Auf dieses Tag kommen wir später zurück.

Django: Formulare

- **Modell-Formulare: App-URL-Regeln**
  - Nun fügen wir eine neue URL zu einer neuen View-Methode an
    - In pruefungsamt/**urls.py**

```
from django.conf.urls import include, path
import views

urlpatterns = [
    path('professor/<int:id>/',                views.show_prof   ),
    path('professoren/',                       views.list_profs  ),
    path('vorlesung/<int:id>/edit/',           views.edit_vl         ),
    path('vorlesung/<int:id>/',                views.show_vl         ),
    path('vorlesungen/',                       views.list_vls    ),
]
```

    - Sie soll es später ermöglichen, von der Ansicht (show_vl) einer Vorlesung aus eine Editier-Seite (**edit_vl**) zum selben Objekt aufzurufen.
    - Diese View-Methode legen wir als nächstes an.

Django: Formulare

- **Modell-Formulare: View-Methode**
  - Die komplette View-Methode zur Formularverarbeitung (Postback)

```
def edit_vl(request, id):
    try:
        vl = Vorlesung.objects.get(id=id)
    except Vorlesung.DoesNotExist:
        raise Http404

    # vl ist jetzt ein valides Objekt
    if request.method == 'POST':
        form = VorlesungForm(request.POST, instance=vl)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect(vl.get_absolute_url())
    else:
        form = VorlesungForm(instance=vl)

    # GET-Request oder Fehler im POST: Wir geben das Formular aus
    return render(request, 'vl_edit.html', dict(vl=vl, form=form) )
```

Analyse auf den nächsten Folien

Django: Formulare

- **Modell-Formulare: View-Methode** (GET-Pfad)
  - Betrachten wir nun **nur** den ersten **GET**-Aufruf der Seite
    - Dabei wird das Formular mit den **Objektdaten** initialisiert und zurück gegeben *(alles Unwichtige in der Darstellung entfernt)*

```
def edit_vl(request, id):
    vl = Vorlesung.objects.get(id=id)

    if request.method == 'POST':
        # ...
    else:
        form = VorlesungForm(instance=vl)

    # GET-Request: Wir geben das Formular aus
    return render(request, 'vl_edit.html', dict(vl=vl, form=form), )
```

    - Das Formular erhält das zu editierende Objekt vl als Initialisierungs-Parameter
    - Das resutlierende Formular-Objekt wird an das Template übergeben

Django: Formulare

- **Modell-Formulare: View-Methode** (POST-Pfad)
  - Betrachten wir nun **nur** die **POST**-Antwort nach der Bearbeitung
    - Dabei wird das Formular mit den **Objektdaten** und den **POST-Daten** initialisiert und mit **is_valid**() getestet:

```
def edit_vl(request, id):
    vl = Vorlesung.objects.get(id=id)

    if request.method == 'POST':
        form = VorlesungForm(request.POST, instance=vl)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect(vl.get_absolute_url())
```

    - Ist der Test erfolgreich (die Werte zulässig), wird das Formular abgespeichert
      - genauer: das Modell-Objekt wird abgespeichert, nachdem die Attribute aus dem Formular aktualisiert wurden
    - Am Ende wird der Web-Client mit einem Redirect auf die Heimatseite des Objekts zurück geschickt, von wo der auf die Editierseite gekommen war.

Django: Formulare

- **Modell-Formulare: View-Methode** (invalid-POST-Pfad)
  - Betrachten wir die **POST**-Antwort bei unzulässigen Eingaben
    - Dabei wird das Formular mit den **Objektdaten** und den **POST-Daten** initialisiert und (nun erfolglos) mit **is_valid()** getestet:

```
def edit_vl(request, id):
    vl = Vorlesung.objects.get(id=id)

    if request.method == 'POST':
        form = VorlesungForm(request.POST, instance=vl)
        if form.is_valid():
            # ...
    else:
        form = VorlesungForm(instance=vl)

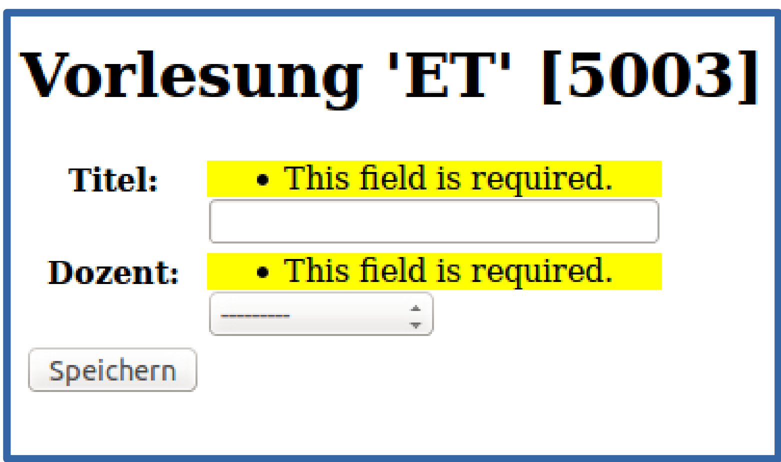
    # GET-Request: Wir geben das Formular aus
    return render(request, 'vl_edit.html', dict(vl=vl, form=form), )
```

    - Wir geben das Formular aus, als ob es aus dem ersten GET stammen würde
      - Allerdings ist nun eine Fehlermeldung für den Nutzer enthalten

Django: Formulare

- **Fehlermeldungen werden in das Formular integriert**
  - Beispiel (erzeugt von Django):

```
<form method=POST action="">
  <table>
    <tr><th><label for="id_titel">Titel:
        <td><ul class="errorlist">
               <li>This field is required.
            </ul>
            <input id="id_titel" type="text" name="titel"
                                            maxlength="128" />
    <tr><th><label for="id_dozent">Dozent:</label>
       ...
```



Django: Formulare

- **Wie entstehen Fehlermeldungen: Formular-Validierung**
  - Modell-Formulare stellen die Einhaltung der **Restriktionen** aus dem Modell sicher
    - Typ- und Längenbeschränkungen (IntegerField, max_length, ...)
    - Wertbeschränkungen (blank, null, …)
    - Zusätzliche Modell-Validatoren (validators / MaxValueValidator, …)
    - Beschränkungen zwischen anderen Daten (unique, unique_together, …)
      - Kann nicht anhand eines Objekts geprüft werden, erfordert Datenbank-Zugriff
  - Man kann auch **eigene Validierungen** hinzufügen
    - z.B. Formular-Methode **clean_*FIELDNAME*()** löst ggf. Exception aus

```
class VorlesungForm(ModelForm):
    # ...
     def clean_email(self):  # wird von Django aufgerufen, prüft email-Feld
          email = self.cleaned_data['email']
          if not email or not email.endswith('.rptu.de'):
               raise forms.ValidationError('Keine RPTU-Addresse angegeben')
          return email
```

    - Siehe https://docs.djangoproject.com/en/4.2/ref/forms/validation/

Django: Formulare

- **Der Formular-Validierungsmechanismus**
  - form.**is_valid()** löst verschiedene clean...()-Methodenaufrufe aus
    - zuerst in den Feldern, dann im Formular selbst
  - In diese kann man sich einklinken (s.o.), z.B.
    - field.**clean()** in den einzelnen Feldern des Formulars
    - form.**clean_*FIELDNAME*()** im Formular für jedes Feld *FIELDNAME*
    - form.**clean()** für das Formular
  - Wenn ein **Problem** gefunden wird, …
    - landet ein Eintrag in form.**_errors** und
    - form.**is_valid()** liefert **False**
  - Wenn **kein Problem** gefunden wird, …
    - landen die validierten Daten in form.**cleaned_data** und
    - form.**is_valid()** liefert **True**
    - In diesm Fall kann man z.B. das Modell-Formular mit save() abspeichern

Django: Formulare

- **Nochmal die Postback-View-Methode**
  - Mit dem Shortcut get_object_or_404() spart man den try-Block
    - Wirft die Http404-Exception wenn das Objekt nicht gefunden wird

```
def edit_vl(request, id):
    vl = get_object_or_404(Vorlesung, id=id)
    if request.method == 'POST':
        form = VorlesungForm(request.POST, instance=vl)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect(vl.get_absolute_url())
    else:
        form = VorlesungForm(instance=vl)
    return render(request, 'vl_edit.html', dict(vl=vl, form=form))
```

  - Wozu brauchen wir den request-Parameter bei render( )?
    - Bei der Erzeugung der Response kann darauf zurück gegriffen werden
      - z.B. im Template über die implizite Template-Variable {{ request }}
      - Das ist hier nötig wegen des **CSRF-Tokens** (`{% csrf_token %}` im Template)

Django: Formulare

- Zur Erinnerung: **Modell-Formulare ...**
  - Ermöglichen uns, auf Basis einer Modell-Klasse ein Formular
    - mit Initial-Werten zu versehen und als HTML-Formular auszugeben
    - aus GET- oder POST-Date die Antwort-Daten zu extrahieren und im Modell abzuspeichern
  - Dabei können wir auswählen, welche Attribute übertragen werden
- **Alternative: Individuell erzeugte Formulare**
  - Anstatt Formulare auf Basis einer Modell-Klasse zu erzeugen, können wir auch vollständig individuelle Formulare erzeugen
    - Beispiel: Login-Formular *(Datensatz wird ja nie so im Modell gespeichert)*
  - Dazu müssen wir die Formular-Felder beschreiben
    - Das geschieht ganz ähnlich wie die Definition eines Modells
    - Die Basisklasse des Formulars ist nun django.forms.**Form**
    - Die Felder stammen ebenfalls aus django.forms

Django: Formulare

- **Beispiel: Individuell erzeugtes Formular**
  - Hier ein eigenes individuelles Vorlesungs-Formular

```
from django import forms
class SpecialVorlesungForm(forms.Form):
    vorlnr = forms.IntegerField(label='Vorl.Nr.')
    titel  = forms.CharField(   label='Titel', max_length=128)
```

  - Zur **Erinnerung**: Modell-Formulare werden zweistufig definiert
    - Modell-Formular:

```
from django.forms import ModelForm
class VorlesungForm(ModelForm):
     class Meta:
          model  = Vorlesung
          fields = ('vorlnr', 'titel',)
```

    - Das zugrunde liegende Modell:

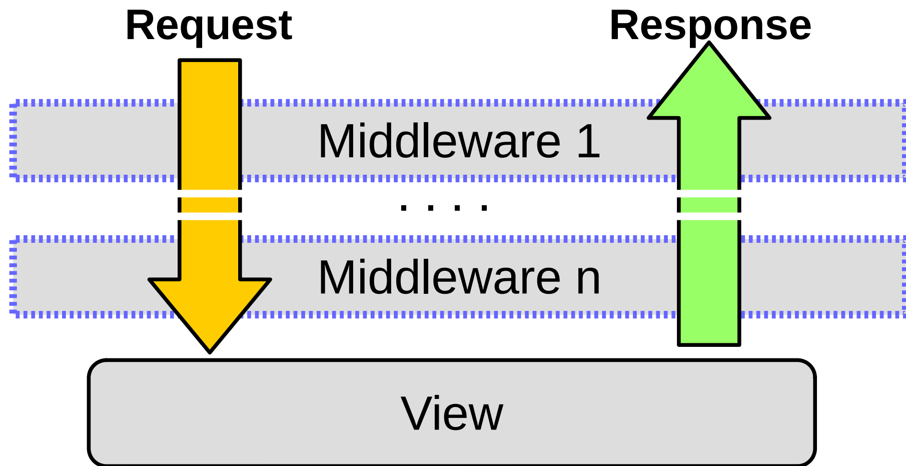
```
from django.db import models
class Vorlesung(models.Model):
    vorlnr = models.IntegerField('Vorl.Nr.', unique=True)
    titel  = models.CharField(   'Titel',    max_length=128)
    dozent = models.ForeignKey(  Professor,  null=True)
```

Django: Formulare

- **Allgemeine Formulare und Modell-Formulare haben viel gemeinsam**
  - Die Klasse django.forms.**Form** ist die Basisklasse für django.forms.**ModelForm**
  - Entsprechend teilen Sie sich viele Eigenschaften
    - Sie benutzen weitgehend die selben clean...()-Methoden
    - Nachdem die Methode **is_valid()** aufgerufen wurde und True liefert, enthält das Attribut **cleaned_data** die Formulardaten
  - Sie haben aber auch **Unterschiede**
    - Modell-Formulare können hier einfach save() aufrufen
    - Allgemeine Formulare müssen die Daten in cleaned_data explizit verarbeiten
  - Man kann sogar hybride Formulare erzeugen
    - Also Modell-Formulare, die zusätzlichen Felder enthalten
      - Beispiel: Doppelte Eingabe der Email-Adresse bei der Registrierung auf Webseite
        - Es wird nur geprüft, ob die Eingaben übereinstimmen, danach wird das zweite Email-Feld nicht mehr gebraucht (im Modell wird die Email-Adresse nur einmal gespeichert).

Django: Middlewares

- **Django beinhaltet einen Middleware-Mechanismus**
  - Im System ist eine Menge von **aktivierten** Middlewares eingestellt
    - Paramter MIDDLEWARE_CLASSES in settings.py
  - Middlewares bearbeiten eingehende Requests und ausgehende Responses
    - Man kann so z.B.
      - beim Response die Kommentare aus dem ausgehenden HTML-Text entfernen
      - Im Request zusätzliche Template-Variablen hinzufügen
      - Responses Cachen und bei erneuten Anfragen aus dem Cache beantworten
      - usw.



Django: Middlewares

- **Standardmäßig aktive Middlewares sind u.a.**
  - **SessionMiddleware**
    - Verwaltet **Sessions** und stellt request.session bereit
  - **CsrfViewMiddleware**
    - Verhindert **Cross-Site-Request-Forgeries** (**CSRFs**)
  - **AuthenticationMiddleware, SessionAuthenticationMiddleware**
    - Verwaltet **Sessions** und stellt request.user bereit
  - **MessageMiddleware**
    - Verwaltet einmalig angezeigte Nachrichten an den Benutzer, die in der Webseite angezeigt werden.
      - z.B. „Der Datensatz wurde gespeichert.“

Django: Middlewares

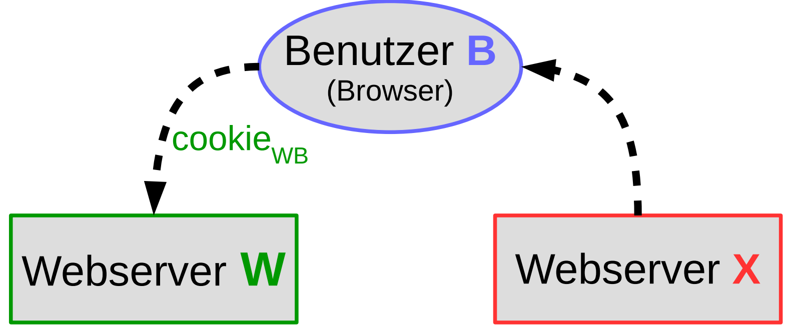
- **Manche Middlewares brauchen Zugriff auf den Request beim Rendern der Antwort**
  - Deshalb haben wir request an render() übergeben
    - render(request, template_name, parameter_dict, …) erhält als ersten Parameter immer das **Request-Objekt**

```
def edit_vl(request, id):
    # ...
    return render(request, 'vl_edit.html', dict(vl=vl, form=form))
```

    - Mit Hilfe des Request-Kontext hat man z.B. in den Templates dann ...
      - Zugriff auf {% csrf_token %} bei der CsrfViewMiddleware
      - Zugriff auf {{ user }} bei der AuthenticationMiddleware
      - Zugriff auf {{ request }}

Django: Middlewares

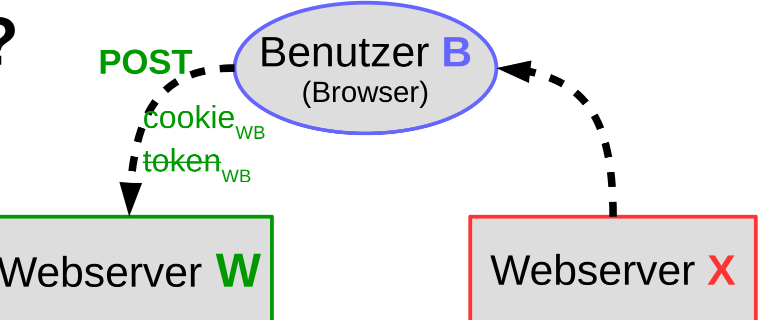
- **Was sind CSRFs?**
  - Betrachten wir folgendes Szenario:



    - Benutzer **B** ist auf einer Webseite **W** **eingeloggt** (gültiges Session-Cookie)
      - Bei jedem Zugriff auf diese Webseite W wird das Session-Cookie übertragen
      - Damit erkennt der Webserver W, dass die Zugriffe vom Benutzer B stammen
    - Anschließend / Parallel ruft Benutzer B die Webseite **X** auf
      - Auf Webseite X befindet sich ein Link auf eine URL in W
        - z.B. ein IMG-Tag mit entsprechendem SRC-Parameter oder über Javascript
    - Dadurch entsteht vom Browser von B ein Request auf Webserver W
      - GET-Request oder ein POST-Request
      - Dieser enthält das gültige Session-Cookie von B
    - Der Webserver W kann nicht erkennen, dass der (von B authentifizierte) Request nicht von B explizit gewünscht war, sondern von X provoziert wurde
      - X könnte so im Namen von B auf Webserver W agieren
        - z.B. Daten manipieren, Bestellungen auslösen, indirekte Angriffe vorbereiten
      - Dabei sind (normalerweise) **GET-Requests** unkritisch
        - Warum?

Django: Middlewares

- **Was kann man gegen CSRFs tun?**



  - Kritisch sind v.a. POST-Requests
    - also Formulare mit method POST
  - Es ist kein Geheimhaltungsproblem
    - Niemand außer B und W kennen das Session-Cookie cookie$_{WB}$
  - Trotzdem ist ein fremder POST-Request mit cookie$_{WB}$ versehen
    - Weil X im Browser von B ein POST-Request an W auslösen kann.
  - Man möchte also erkennen können, **woher** der POST-Request ursprünglich stammt, nicht wer ihn **verschickt** hat *(in beiden Fällen B)*
    - **Idee**: W integriert in sein Formular an B eine geheime Information (**token$_{WB}$**)
      - Diese ist spezifisch für die Paarung (W, B) und nur in dem Formular enthalten
      - Diese Information wird später Teil des POST-Requests
    - Wenn B das „*gute*“ W-Formular an W POSTet, ist das Geheimnis enthalten
    - X kennt diese geheime Information nicht
      - Also kann X keinen POST-Request mit token$_{WB}$ erzeugen

Django: Middlewares

- **Wie funktioniert die CSRF-Protection von Django?**
  - In jedes Formular wird das Geheimnis $token_{WB}$ eingebettet
    - Dazu hatten wir oben im Formular das Tag „`{% csrf_token %}`“ eingefügt
    - Dieses erzeugt ein hidden-Input-Field mit $token_{WB}$

```
<div style='display:none'>
     <input    type='hidden'
               name='csrfmiddlewaretoken'
               value='q392uasopdhjqp92asdkqp23rfasd' />
</div>
```

  - Bei einem eingehenden Request …
    - wird von der **CsrfViewMiddleware** geprüft, ob der entsprechende Paramter mit dem korrekten Wert enthalten ist
    - Ist er nicht enthalten wird eine 403-Response zurück geschickt.
    - Ist er enthalten wird der entfernt und der Request wird normal weiter verarbeitet

Django: Transaktionen

- **Wir kennen aus SQL bereits Transaktionen**
  - Transaktionen definieren einen **semantisch atomaren Zustandsübergang** der Datenbank
    - von einem konsistenten Zustand zu einem anderen
    - inkosistente Zustände sollen für andere Transaktionen nicht sichtbar werden
  - Die Modell-Definition in Django definiert Konsistenzbedingungen
    - Attributtypen, unique, unique_together, …
- **Wir wollen in Django auch DB-Transaktionen steuern können**
  - Beginn / Ende einer Transaktion, Abbruch einer Transaktion, …
  - Standard-Transaktions-Verhalten:
    - **Auto-Commit** jeder Django-DB-Operation
      - z.B. bei jedem Aufruf von model.save() oder model.delete()

Django: Transaktionen

- **Atomarität von Views**
  - Views verarbeiten einen Request und erzeugen einen Response
  - Aus **Nutzersicht** haben diese Transaktionseigenschaften
    - Löse ich eine Funktion „Warenkorb bestellen“ aus, so soll z.B.
      - ***geprüft*** *werden, ob die Waren im Warenkorb gerade verfügbar sind,*
      - *die Waren als* ***gekauft*** *markiert aus dem verfügbaren Bestand entfernt werden,*
      - *die* ***Zahlung*** *veranlasst werden (Kundenkonto belasten, Gutscheine, etc.),*
      - *die Waren zum* ***Versand*** *vorgesehen werden*
    - Alle Operationen sollen **atomar**, also alle oder keine davon erfolgen
    - Auch für den Benutzer ist „ganz oder gar nicht“ auf Request-Ebene leicht zu verstehen.
  - Aus **Systemsicht** sollen diese Operationen **ACID** erfolgen
    - Das Autocommit-Modell der Einzel-Operationen ist oft nicht sinnvoll.
  - Wir möchten also ein Auto-Commit eines kompletten View-Aufrufs
    - also **eines Requests**

Django: Transaktionen

- **Views in einer DB-Transaktion abwickeln**
  - Um dies zum Default zu machen dient der settings-Parameter ATOMIC_REQUESTS ¶
    - Default ist False → in settings.py auf True setzen
    - Genaueres: https://docs.djangoproject.com/en/4.2/topics/db/transactions/#tying...
  - Ablauf:
    - **Request-Behandlung** (also **View-Aufruf**) ≙ **Transaktion**
    - Endet der View-Aufruf normal, erfolgt automatisch ein **Commit**
    - Endet sie mit einer unbehandelten Exception, so wird erfolgt **Rollback**
  - Vorteil:
    - Man muss keine unvollständigen Zwischenzustände im Fehlerfall behandeln
  - Nachteil:
    - Transaktionen dauern so lange wie die gesamte Request-Behandlung
    - Vorsicht beim **Logging** in die Datenbank (wird evtl. mit zurück gesetzt)

Django: Transaktionen

- **Man kann die Transaktionsbehandlung auch individuell steuern**
  - z.B. durch einen Decorator der View-Funktion
    - **@atomic**
      - Nur bestimmte Views atomic ausführen

```
from django.db import transaction

@transaction.atomic
def viewfunc(request):
    # ....
```

  - z.B. nur einen Code-Abschnitt atomar ausführen

```
from django.db import transaction

with transaction.atomic():
    a.save()
     b.save()
```

  - Die Transaktionssteuerung ist sehr präzise möglich *(Savepoints, ...)*
    - Siehe https://docs.djangoproject.com/en/4.2/topics/db/transactions/

Django: Benutzerverwaltung

- **Die Benutzerverwaltung in Django**
  - wird u.a. im Admin-Interface benutzt
  - Hier kann man auch Benutzer, Benutzergruppen und Rechte (für Benutzern oder für Gruppen) zuordnen
    - Rechte sind u.a. für jede Modell-Klassen ...
      - das Recht ein Objekt anzulegen
      - das Recht ein Objekt zu ändern
      - das Recht ein Objekt zu löschen
    - Darüber hinaus hat jeder Benutzer die Bool-Attribute
      - Active (wenn False kann der Benutzer nicht authentifiziert werden)
      - Staff-Status (darf man sich im Admin-Interface anmelden)
      - Superuser-Status (hat man alle Rechte implizit)
  - Sie basiert auf der AuthenticationMiddleware
    - Ist diese aktiv (default), so hat jeder Request die Komponente **request.user**, die auf ein User-Objekt verweist
      - ggf. auf den AnonymousUser, wenn niemand authentifiziert ist

Django: Benutzerverwaltung

- **Methoden der User-Objekte**
  - Um einen echt angemeldeten User vom AnonymousUser, zu unterscheiden, dient das Attribut is_authenticated
    - Möchte man die Funktion einer View (z.B. Änderungs-Formular) nur angemeldeten Nutzern bereitstellen, so kann man das in der View auch folgendermaßen prüfen:

```
def viewfunc(request):
     if request.user.is_authenticated:
          # nur authentifizierte User
```

    - Eleganter ist dieser **Dekorator**:

```
@login_required
def viewfunc(request):
          # nur authentifizierte User
```

      - Ist der Benutzer nicht angemeldet wird man auf eine Login-Seite umgeleitet
    - In Templates gibt es die Variable „**user**“

```
{% if user.is_authenticated %}
     {# nur authentifizierte User #}
{% endif %}
```

      - Weitere Einzelheiten dazu: https://docs.djangoproject.com/en/4.2/topics/auth/

Django: Benutzerverwaltung

- **Tipp: Eine Login-Seite für Prototyp-Applikationen**
  - Der login_required-Dekorator benötigt ggf. eine Login-Seite ...
    - auf die er den (noch nicht eingeloggten) Nutzer umleiten kann
    - Standardmäßig nutzt er „`/accounts/login/`“
  - Wenn man kein Login-Template oder keine View dafür anlegen will, kann man folgende URL-Mapper-Regel benutzen

```
path( 'accounts/login/',
      'django.contrib.auth.views.login',
      dict(template_name='admin/login.html')
),
```

  - Sie nutzt
    - eine vom System bereitgestellte view-Funktion und
    - das Login-Seiten-Template des Admin-Interfaces
      - Letzteres kann natürlich leicht ersetzt werden um eine eigene Login-Seite passend zum Applikations-Design anzulegen

Django: Benutzerverwaltung

- **Feingranulare Benutzerrechte**
  - Durch die feingranularen Benutzerrechte kann man weitaus präziser Zugriffe steuern.
    - Beispiel:

```
def viewfunc(request):
    if request.user.has_permission('pruefungsamt.change_vorlesung'):
        # nur wenn man Vorlesungen ändern darf
```

    - Oder wiederum als Dekorator:

```
@permission_required('pruefungsamt.change_vorlesung')
def viewfunc(request):
        # nur wenn man Vorlesungen ändern darf
```

    - In Templates gibt es dazu die Kontext-Variable „**perms**“:

```
{% if perms.pruefungsamt.change_vorlesung %}
    {# nur wenn man Vorlesungen ändern darf #}
{% endif %}
```

  - Man kann auch beliebig neue Rechte anlegen und darauf testen
    - Auch hier gilt dann: Superuser haben immer alle Rechte
    - Mehr dazu: https://docs.djangoproject.com/en/4.2/topics/auth/default/

Django: Migrationen

- **Schema-Migration**
  - ist die **Anpassung der DB-Strukturen** an ein geändertes Daten-Modell
    - Beispiel: Die Professoren erhalten ein neues Attribut „Vorname“
  - Schema-Migrationen treten durch die Weiterentwicklung der Applikation regelmäßig auf
    - Die Anpassung soll bei der Aktivierung einer neuen Software-Version weitgehend oder ganz automatisch erfolgen
- **Daten-Migration**
  - ist die **Anpassung der Daten in der DB**
    - Beispiel: Alle Namen sollen ab jetzt mit Großbuchstaben anfangen
      - Neue Eingabe erfüllen das bereits, die Altdaten müssen angepasst werden
  - Daten-Migrationen sind oft die Folge von Schema-Migrationen
    - Beispiel: Nach Einführung des Vornamen-Attributs sollen die Namensfelder am enthaltenen Komma in Name und Vorname aufgeteilt werden

Django: Migrationen

- **Migrationen sollen …**
  - zuverlässig,
  - weitgehend automatisch und
  - umkehrbar ablaufen
- **Wunschvorstellung**
  - Bei der Aktivierung einer neuen Software-Version finden automatisch auch alle nötigen Migrationen statt
    - Danach ist das System ohne Nacharbeiten sofort wieder betriebsbereit.
    - Auch die **Rückmigration** nach einem Downgrade auf eine ältere Software-Version soll automatisch stattfinden.
      - selbst wenn zwischenzeitlich auf dem neuen System Daten verändert wurden
  - Die Erstellung der Schema-Migrations-Scripte erfolgt weitgehend automatisch
    - Man kann aber eingreifen

Django: Migrationen

- **Django legt zu jeder Migration ein Script an**
  - Für App „pruefungsamt“ z.B. in „pruefungsamt**/migrations**“
  - Die Scripte sind aufsteigend von 0001 durchnummeriert, z.B.
    - 0001_initial.py
    - 0002_auto__add_field_professor_vorname.py
  - In jedem der Scripte wird definiert, welche Änderungen erfolgen müssen
    - Damit ist es möglich, den Migrationsschritt von der nächst kleineren Stufe zu der Stufe der Scriptnummer hin machen oder diesen Schritt umzukehren
  - Zusätzlich speichert Django in der Datenbank die Nummer der aktuellen Migrationsstufe
    - Damit ist klar, in welcher Stufe das DB-Schema sich befindet ...
    - … und was zu tun ist um jeweils eine Stufe auf- oder abzusteigen
  - Siehe https://docs.djangoproject.com/en/4.2/topics/migrations/

Django: Migrationen

- **Beispiel:**
  - Wir legen in der App „pruefungsamt“ im Modell „Professor“ ein neues Attribut „vorname“ an

```
class Professor(models.Model):
    persnr = models.IntegerField(max_length=10, unique=True)
    name   = models.CharField(max_length=64)
    vorname= models.CharField(max_length=64, blank=True)
```

  - Wir lassen manage.py das zugehörige Migrations-Script erzeugen

```
./manage.py makemigrations pruefungsamt --auto
```

→ 0002_auto__add_field_professor_vorname.py

```
class Migration(migrations.Migration):
  # ...
  operations = [
    migrations.AddField(
            model_name='Professor',
            name='vorname',
            field=models.CharField(blank=True, max_length=64),
        ),
   ]
```

Django: Migrationen

- **Wie generiert man Schemamigrations-Scripte?**
  - Folgender Aufruf erzeugt automatisch ein Migrationsscript:

```
./manage.py makemigrations pruefungsamt --auto
```

    - „--auto“ bewirkt dabei, dass der Script-Name automatisch erzeugt wird
      - z.B. „0002_auto__add_field_professor_vorname.py“ im obigen Beispiel
  - Hier wird nichts an dem DB-Schema geändert, nur das Script erzeugt
- **Wie aktualisiert man das DB-Schema?**
  - Folgender Aufruf bringt das DB-Schema auf Stufe 0002

```
./manage.py migrate pruefungsamt 0002
```

  - Um alle Apps auf die jeweils neueste Stufe zu bringen:

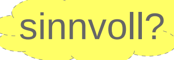
```
./manage.py migrate
```

  - Um zu sehen, welche Migrationen noch durchzuführen sind:

```
./manage.py showmigrations
```

Django: Migrationen

- **Datenmigration**
  - Neben Schema-Anpassungen müssen bei Migrationen gelegentlich auch Daten angepasst werden.
  - Das erfolgt in einer **Datenmigration** mit den gleichen Mechanismen wie oben in der **Schemamigration**, es gibt also auch ein Migrationsscript
  - Die Daten-Migrationen müssen allerdings manuell erstellt werden
    - Das System kann ja nicht wissen was wir an den Daten ändern wollen
  - Beispiel: Umwandlung *Personalnummer* (Integer) in *Personalkennung* (Struktur Abteilungs-Kennung „-“ Personalnummer, z.B. „FBINF-3587“)



1) Schemamigration: Attribut Personalkennung (Charfield) hinzufügen (Null=True)

2) Datenmigration: Für jeden Mitarbeiter: Personalkennung berechnen und setzen

3) Schemamigration: Null=True aus Attribut Personalkennung entfernen

4) Schemamigration: Altes Attribut Personalnummer löschen

→ Bei Aufruf von „`./manage.py migrate`“ werden dann alle Schritte ausgeführt.

  - Mehr dazu: https://docs.djangoproject.com/en/4.2/topics/migrations/#data-migrations

Vertiefung zu Kapitel 3

Reguläre Ausdrücke (RE) und ihre Verwendung in Django

Einschub: Reguläre Ausdrücke

- **Reguläre Ausdrücke** (Regular Expressions, RE)
  - Sind **Muster** (engl. Pattern), zu denen Strings passen (können)
    - Ähnlich zu einer Suchfunktion in einer Textverarbeitung „suchen“ wir nach einem **Match** des Regulären Ausdrucks auf dem String
    - Beispiel: RE 'ge' matcht 'Wege' und 'gehen', da 'ge' in beiden vorkommt
    - Beispiel: RE 'ge' matcht nicht 'Straßen', da 'ge' nicht darin vorkommt
  - Diese Muster können Steuerzeichen enthalten
    - So bedeutet '^', dass es nur am Anfang des Strings passen kann
    - Beispiel: RE '^ge' matcht 'gehen', da 'ge' am Anfang steht steht
    - Beispiel: RE '^ge' matcht nicht 'Wege' und 'Straßen', da 'ge' nicht am Anfang steht
  - In Python schreiben wir Reguläre Ausdrücke in String-Literalen meist mit Präfix 'r' (z.B. r'^ge')
    - Dadurch werden Sonderzeichen („\“) im String nicht von Python interpretiert

Einschub: Reguläre Ausdrücke

- **Reguläre Ausdrücke**
  - Es gibt viele RE-Steuerzeichen. Hier eine Auswahl.
    - Siehe https://docs.python.org/3/library/re.html
  - *Grundregel: Jedes nicht-Steuerzeichen* matcht sich selbst (z.B. „a“)
  - '**.**' Punkt = Beliebiges Zeichen
  - '**^**', '**$**' Anfang bzw. Ende des Strings
  - '**?**', '*****', '**+**' Das vorangehende Muster darf 0...1 mal ('?'), >=0 mal ('*') bzw. >=1 mal ('+') auftreten
    - Beispiel: r'ab*c' bedeutet „einmal a, beliebig oft b, einmal c“
    - Der String 'xabbcy' wird also gematcht von r'ab*c' und r'ab+c', nicht aber von r'ab?c'
  - '**{**n,m**}**', '**{**n**}**' Das vorangehende Zeichen oder Muster muss n bis m mal bzw. genau n mal auftreten
    - Beispiel: Die RE r'ab{3,3}c', r'ab{3}c' und r'abbbc' sind äquivalent

Einschub: Reguläre Ausdrücke

- **Reguläre Ausdrücke**
  - '(…)' Der RE in der Klammer gehört zusammen, das Ergebnis des enthalten RE wird namenlos im Ergebnis des Matchvorgangs gespeichert
    - Beispiel RE r'^(ab)+$' matcht 'ab', 'abab', 'ababab' usw.
  - '[…]' Eines der Zeichen in der Klammer
    - Beispiel: RE r'^[ab]+$' matcht 'a', 'b', 'aa', 'ab', 'ba', 'bb' usw.
    - In der eckigen Klammer haben „.“, „$“, „?“ etc. keine besondere Bedeutung mehr
  - '[…-...]' Eines der Zeichen im Bereich von … bis
    - z.B. '[1-4]' entspricht '[1234]'
  - '[^...]' Keines der Zeichen aus …
    - z.B. '[^0-9.]' matcht ein beliebiges Zeichen, das keine Ziffer und kein „.“ ist
  - '\d' Dezimalziffern (entspricht '[0-9]')
  - '\w' Buchstaben (entspricht '[a-zA-Z0-9_]')
  - '(?P<name>...)' Das Ergebnis des RE '...' wird auch unter der Bezeichnung name im Ergebnis des Matchvorgangs gespeichert

Einschub: Reguläre Ausdrücke

- **Reguläre Ausdrücke verwenden**
  - Unix-Kommandozeile: z.B. mit **grep**
    - **grep**: ein Tool, das aus Dateien alle Zeilen ausgibt, die zum Pattern passen
    - z.B. liste alle Zeilen aus x.txt auf, die mit einer Ziffer anfangen:

```
grep '^[0-9]' x.txt
```

    - z.B. liste alle Zeilen aus x.txt auf, die mindestens eine Ziffer enthalten:

```
grep '[0-9]' x.txt
```

    - z.B. liste alle Zeilen aus x.txt auf, die nur aus Ziffer bestehen:

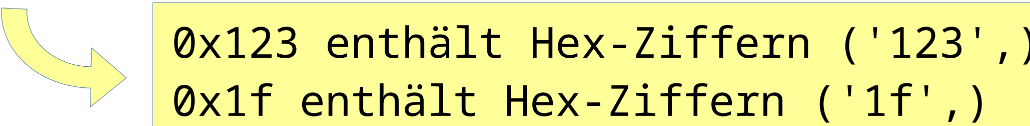
```
grep '^[0-9][0-9]*$' x.txt
```

    Fragen:
      - Was ist der Unterschied zwischen „`[0-9][0-9]*`“ und „`[0-9]+`“?
      - Warum dann „`[0-9][0-9]*`“ statt „`[0-9]+`“?
        - Es gibt verschieden Sprachumfänge für REs
        - grep kennt u.a. „`+`“ nicht (es ist in grep's RE ein normales Zeichen)
        - Wie sieht dann eine Zeile aus, die in grep zu „`^[0-9]+$`“ passt?

Einschub: Reguläre Ausdrücke

- **Reguläre Ausdrücke verwenden**
  - Python: Bibliothek **re** (siehe https://docs.python.org/3/library/re.html)
    - Die Funktion re.**match**(pattern, string) liefert ein **Match-Objekt** oder None
    - Die Match-Object-Methode **groups**() liefert die Matches als Tupel:

```
import re
for s in ('0x123', '0x1f', '0x1g', '01d'):
    m = re.match(r'^0x([0-9a-f]+)$', s)
    if m is not None:
        print('%s enthält Hex-Ziffern %s' % (s, m.groups()))
```



- Die Match-Object-Methode **groupdict**() liefert benannte Matches als Dictionary:

```
m = re.match(r'(?P<name>\w+)=(?P<wert>\d+)', 'counter=17')
print(m.groups())
print(m.groupdict())
```

```
('counter', '17')
{'name': 'counter', 'wert': '17'}
```

Django: Reguläre Ausdrücke

- Anwendung von REs: **Queryset-Filter**
  - `Professor.objects.filter(name__regex=r'^W.*[hr]$')`
    - Liefert alle Professoren, deren Name mir „W“ beginnt und mit „h“ oder „r“ endet.
- Anwendung von REs: **Modelfield-Validatoren**
  - `validators = [ RegexValidator(r'^#[0-9a-f]{6}$') ]`
    - `models.CharField` mit diesem validators-Parameter erlaubt nur 6-stellige CSS-Hex-Farbangaben z.B. `#ffaa37`
- Anwendung von REs: **Custom Path Converter**

```
class FourDigitYearConverter:
    regex = '[0-9]{4}'
    def to_python(self, value):
        return int(value)
    def to_url(self, value):
        return '%04d' % value
```

```
register_converter(
    FourDigitYearConverter, 'yyyy'
)
urlpatterns = [
    path('articles/<yyyy:year>/', ...),
]
```

  - https://docs.djangoproject.com/en/4.2/topics/http/urls/#registering-custom-path-converters

Django: Reguläre Ausdrücke

- Anwendung von REs: `re_path` statt `path` in **urls.py** **(1)**
  - https://docs.djangoproject.com/en/4.2/topics/http/urls/#using-regular-expressions

Positions-URL-Argumente

- from django.conf.urls imprt url, include
  import news.views

```
urlpatterns = [
    re_path(r'^articles/(\d{4})/$',                    news.views.year_archive ),
    re_path(r'^articles/(\d{4})/(\d{2})/$',            news.views.month_archive ),
    re_path(r'^articles/(\d{4})/(\d{2})/(\d+)/$',      news.views.article_detail ),
]
```

- Beim Zugriff der URL /articles/2015/12/ resultiert folgender Aufruf:
  - `news.views.month_archive(request, '2015', '12')`
  - Die Reihenfolge der Parameter entspricht der im RE

Verständnisfrage: *Was passiert, wenn man oben das „$“ weglässt? Kann man das Problem vermeiden?*

Django: Reguläre Ausdrücke

- Anwendung von REs: `re_path` statt `path` in **urls.py** **(2)**

  Benannte-URL-Argumente

  – Man kann die Argumente auch **benennen**
    - siehe RE-Pattern: `'(?P<name>...)'`

  – z.B.
    - `re_path(r'^articles/(?P<year>\d{4})/(?P<month>\d{2})/$', ...)`
    - Der RE matcht genau wie `r'^articles/\d{4}/\d{2}/$'`
    - Hier werden die Ergebnisse den benannten Parametern year und month zugewiesen
    - Beim Aufruf der URL '/articles/2015/12/' erfolgt folgender Funktionsaufruf:
      – `news.views.month_archive(request, year='2015', month='12')`
    - Analog zu `path('articles/<int:year>/<int:month>/$'`, ...)
      – Aber bei re_path mehr Kontrolle über Format (hier: Anzahl der Ziffern).