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

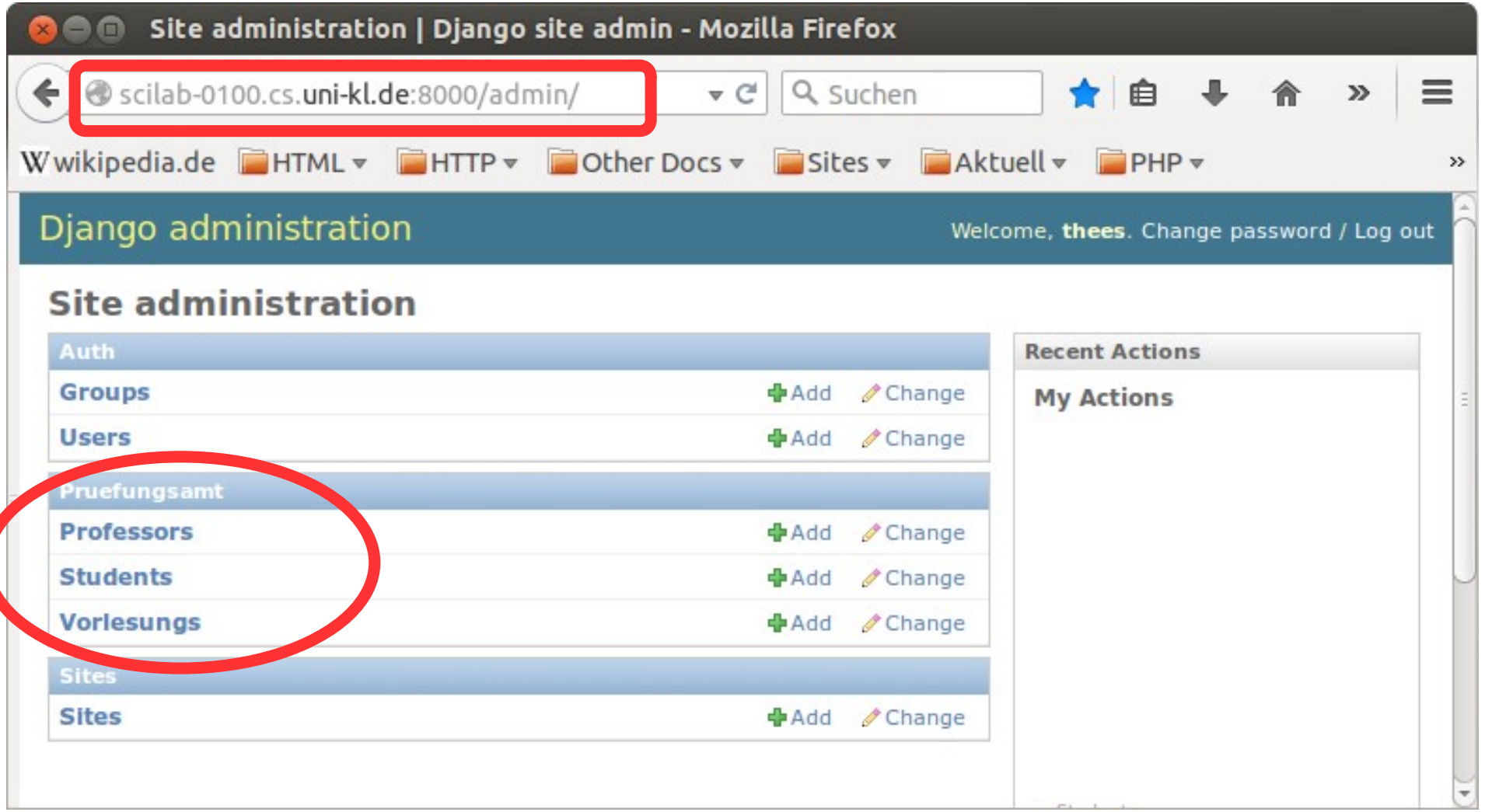
Django Quickstart

- **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

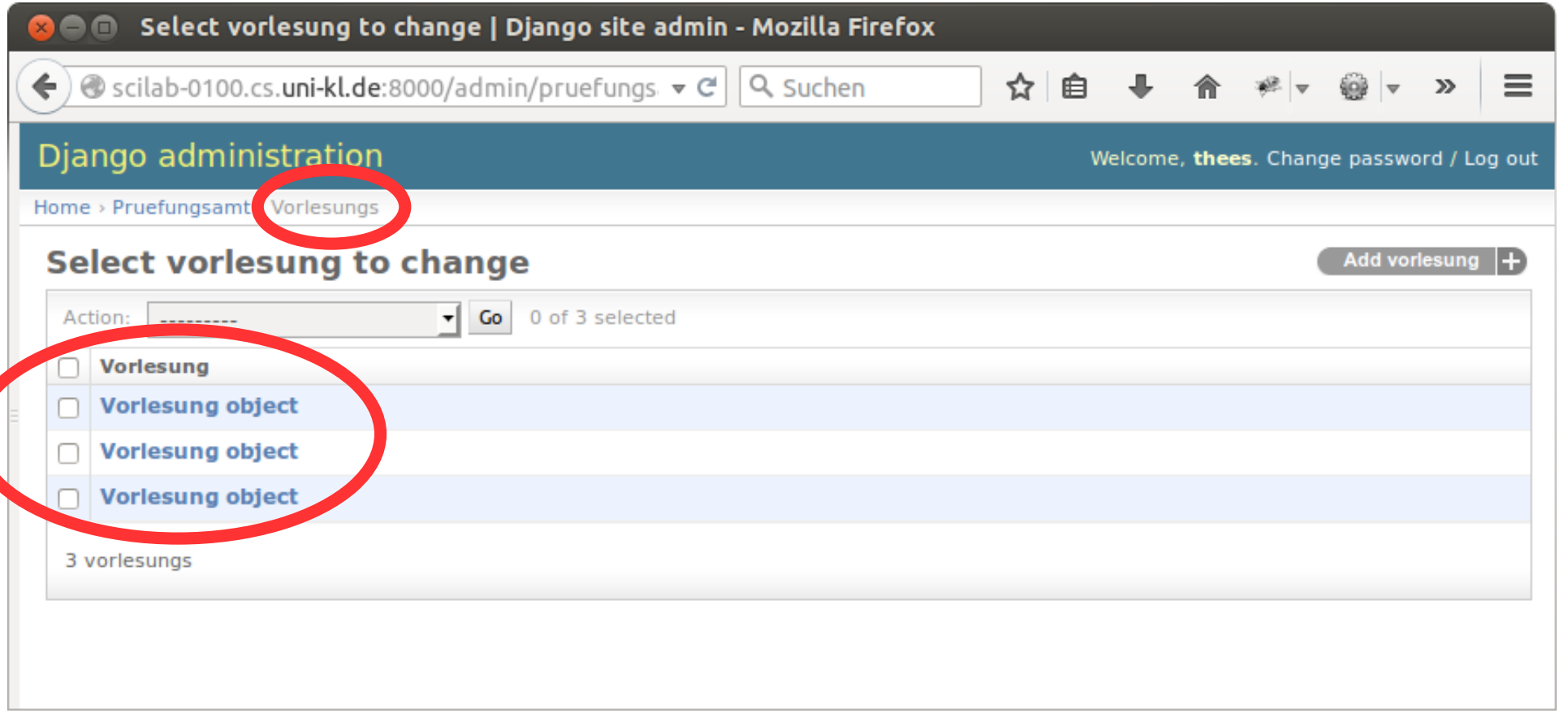
Django Quickstart

- **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



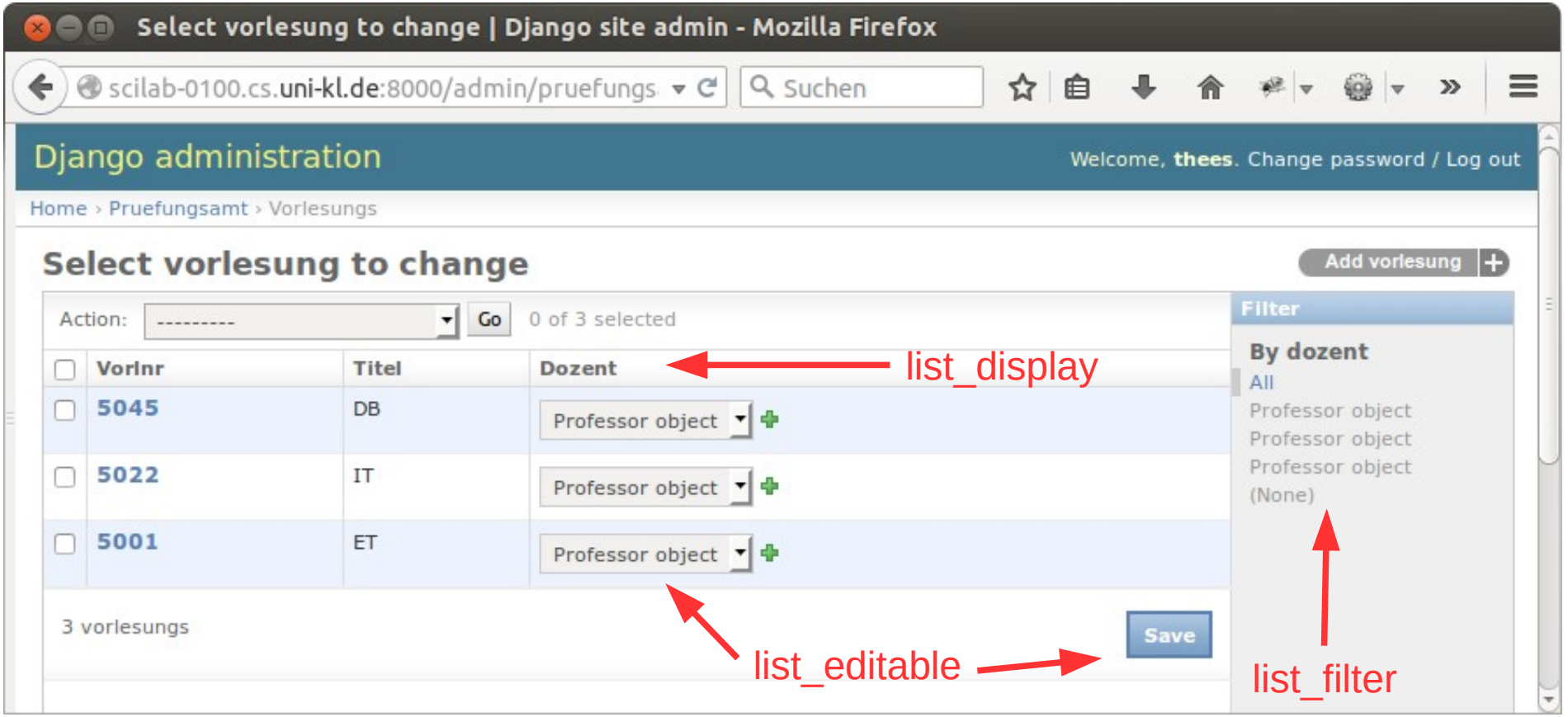
Django Quickstart

- **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

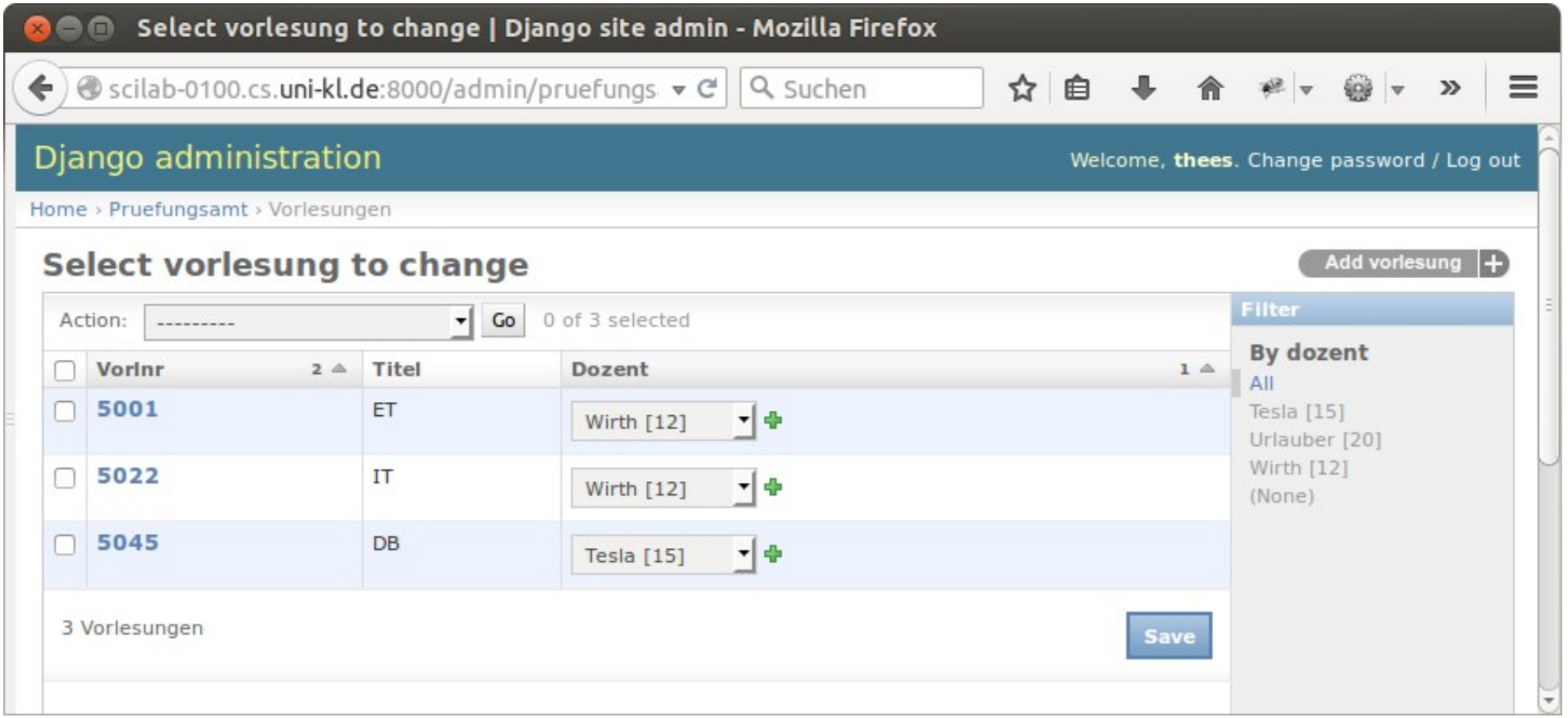
- **Wir ergänzen die Schema-Objekte:**
  - um eine Methode `__str__`, die einen lesbaren Text liefert
  - um eine Meta-Klasse, die Zusatzinformationen liefert
    - z.B. Name der Klasse in der Darstellung (singular / plural)
    - z.B. Standard-Reihenfolge bei Queries

- **`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',)
```

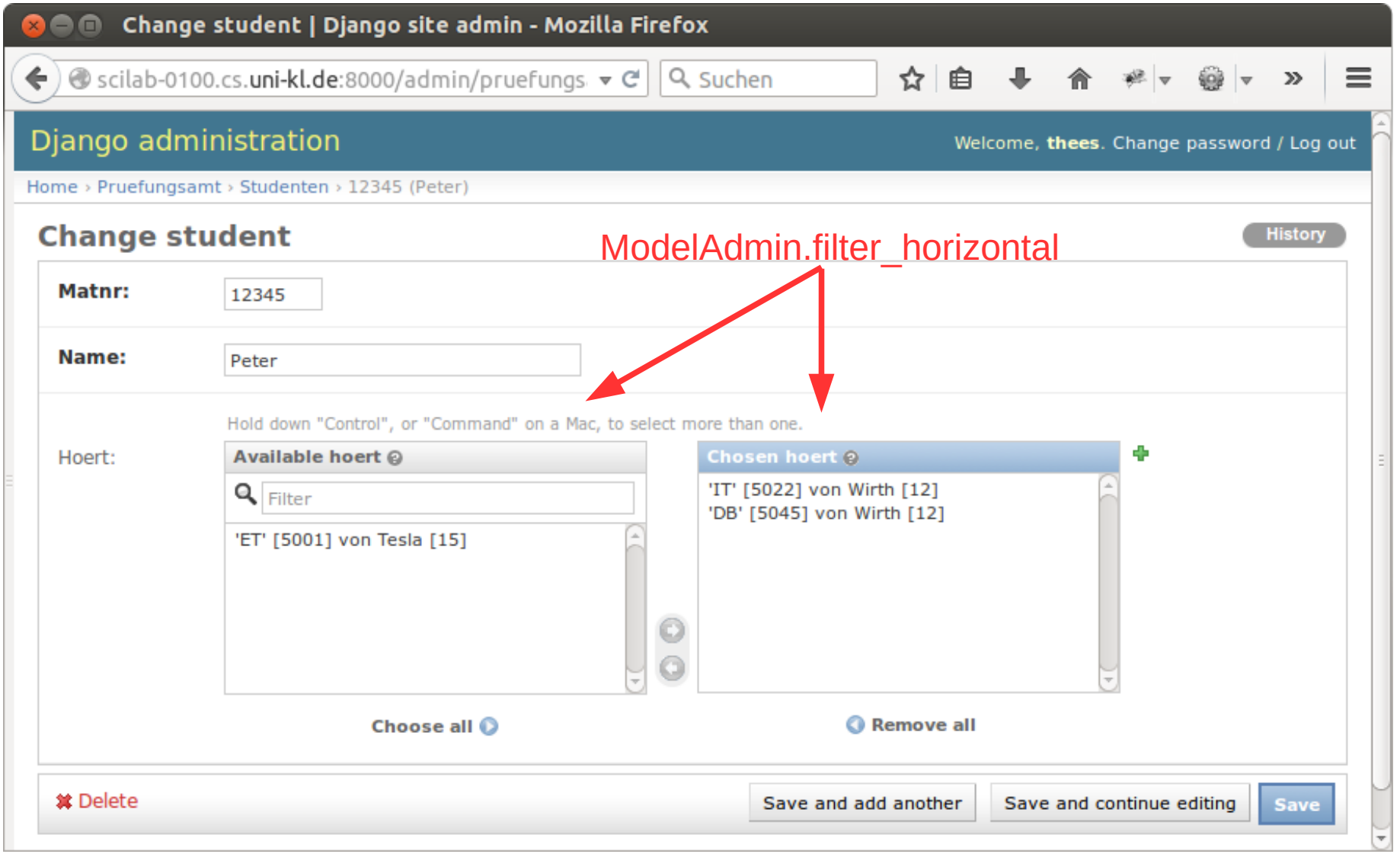
Django Quickstart

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



Django Quickstart

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



Django Quickstart

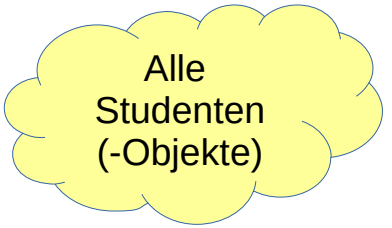
- **Schema-Zugriff mit der Python-Shell (1)**
  - Man kann die Schema-Objekte in eigenen Programmen oder gar interaktiv in einer Python-Shell zugreifen
    - Zu letzterem rufen wir das Django-Kommando „shell“ auf
    - Wenn `ipython` installiert ist, wird dieses als Shell benutzt (mehr Komfort)
  - ./manage.py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

  - 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?

```
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