

# 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 Initialwert 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

---

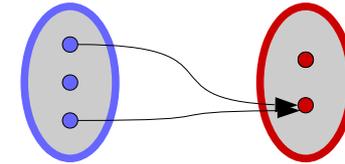
- **Feld-Typen** (komplexe Typen)
  - **DateField, TimeField, DateTimeField**
    - Datum- oder Zeitstempel-Felder
  - **EmailField, URLField**
    - Zeichenketten, die Emails oder URLs (ggf. mit existierendem Ziel) enthalten
  - **AutoField**
    - Spezielles IntegerField mit dem **AUTO\_INCREMENT**-Verhalten 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)**

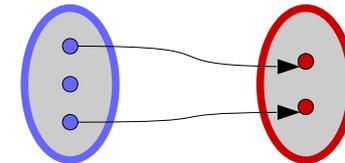
- **ForeignKey(*othermodel*)**

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



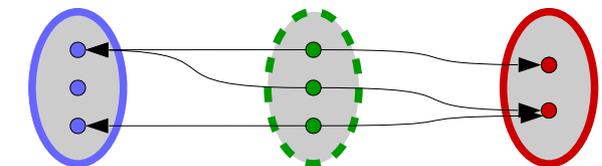
- **OneToOneField(*othermodel*)**

- **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]>]

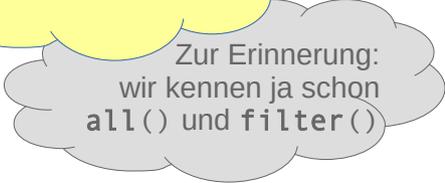
– Zur n:1-Relation `Vorlesung.dozent` → `Professor` ist *implizit* die Rückrichtung `Professor.vorlesung_set` → `Vorlesung` als 1:n-Relation (liefert *Menge*, also *Query-Set*) zugreifbar.

– **Übungsfrage:** Wie kann man das selbe Ergebnis direkt erhalten?

- `Vorlesung.objects.filter(dozent=d)`



QuerySet.get(...) liefert 1 Objekt



Zur Erinnerung: wir kennen ja schon all() und filter()

# 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 *implizit* 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 Parametervergleich
  - `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('vorgesezte__gehalt'))`
        - Liefert Alle, die mehr verdienen als ihre jeweiligen Vorgesetzten

- **Q-Ausdrücke:** Logische Verknüpfung von Bedingungen

- Mit `models.Q(...)` kann man aus den übergebenen Filter-Kriterien **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, das 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 ein Objekt im QuerySet existiert
    - Entspricht also `count() > 0`, ist aber effizienter