

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:

Template-
rendering

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

```
<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.
 - Der Typ („*Converter*“) definieren zulässige Zeichenketten (z.B. „`int`“: Zahlen)
 - Der Name (z.B. „`year`“, „`id`“) dient zur Übergabe der Werte an die View

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:

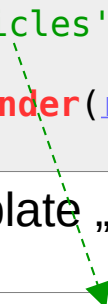
```
news.views.month_archive(request, year=2022, month=12)
```

- 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:

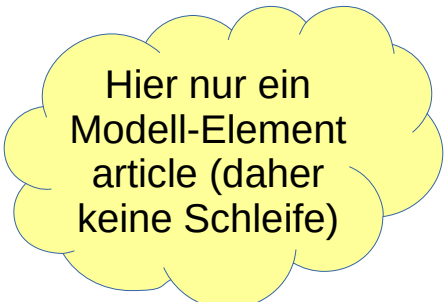
```
news.views.article_detail(request, id=123)
```

- 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.
  - Evtl. gibt es auch nur ein Template, dem z.B. `ext` als Parameter übergeben wird.

# 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:

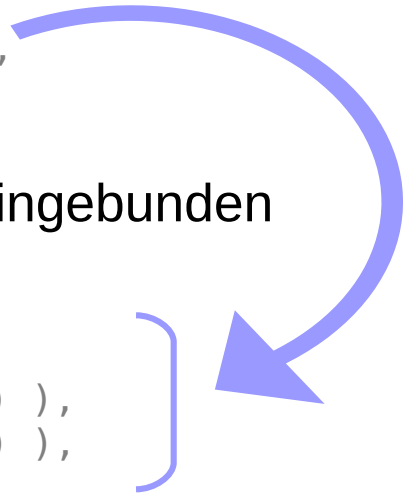
```
from django.conf.urls import path, include
from django.contrib import admin

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 }} ' >{{ 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;">
 </div>
 <div style="background: #afa;">
 {% block menu %}
 [Home]
 {% 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 }}
 [Liste aller Vorl.]
{% 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>
 {{ vl.dozent }}
 </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>...
 <td>{{ vl.vorlnr }}
 <td>...
 {% 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>

 {% for vl in prof.vorlesung_set.all %}
 {{ vl }}
 {% empty %}
 Keine Vorlesungen
 {% endfor %}

```

# 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 # Modellklasse
 fields = ('titel', 'dozent',) # editierbare Attribute
 # exclude = () # oder: ausgeschlossene Attribute
```

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

# Django: Formulare

- **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 resultierende 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">
 This field is required.

 <input id="id_titel" type="text" name="titel"
 maxlength="128" />
 <tr><th><label for="id_dozent">Dozent:</label>
 . . .
```

## Vorlesung 'ET' [5003]

Titel: • This field is required.

Dozent: • This field is required.

Speichern

# 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-Adresse angegeben')
 return email
```

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

# Django: Formulare

---

- **Der Formular-Validierungsmechanismus**

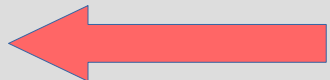
- `form.is_valid()` löst verschiedene `clean...()`-Methodenaufrufe aus
  - zuerst in den Feldern, dann im Formular selbst
- In diese kann man sich einklinken (s.o.), z.B.
  - `field.clean()` in den einzelnen Feldern des Formulars
  - `form.clean_FIELDNAME()` im Formular für jedes Feld *FIELDNAME*
  - `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 diesem 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 SpecialVorlesungsForm(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 VorlesungsForm(ModelForm):
 class Meta:
 model = Vorlesung
 fields = ('vorlnr', 'titel',)
```

- Das zugrunde liegende Modell:

```
from django.db import models
class Vorlesung(models.Model):
 vorlnr = models.IntegerField(label='Vorl.Nr.', unique=True)
 titel = models.CharField(label='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).