

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()
```

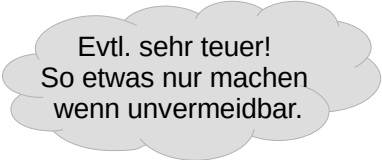
- Vorsicht bei solchen Massen-Updates! Was tut folgendes?
`Student.objects.all().update(name='Bob')`

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 max. ein Treffer möglich (Übungsfrage: 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()
```



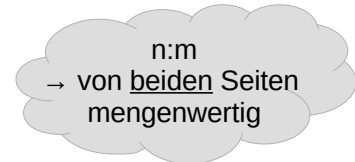
Evtl. sehr teuer!
So etwas nur machen
wenn unvermeidbar.

Django: QuerySet-API

• Mengenartige Operationen auf Relationen

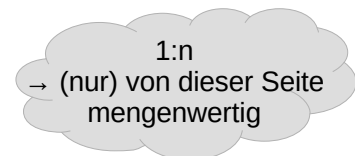
– Ändern einer **ManyToMany**-Relation

- `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(...)`

- Weitergehend 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: HttpResponseRedirect-Objekt-Varianten**

- Es gibt einige Spezialisierungen der HttpResponseRedirect-Klasse, die in erster Linie den 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**)
- Es gibt weitere HttpResponseRedirect-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.

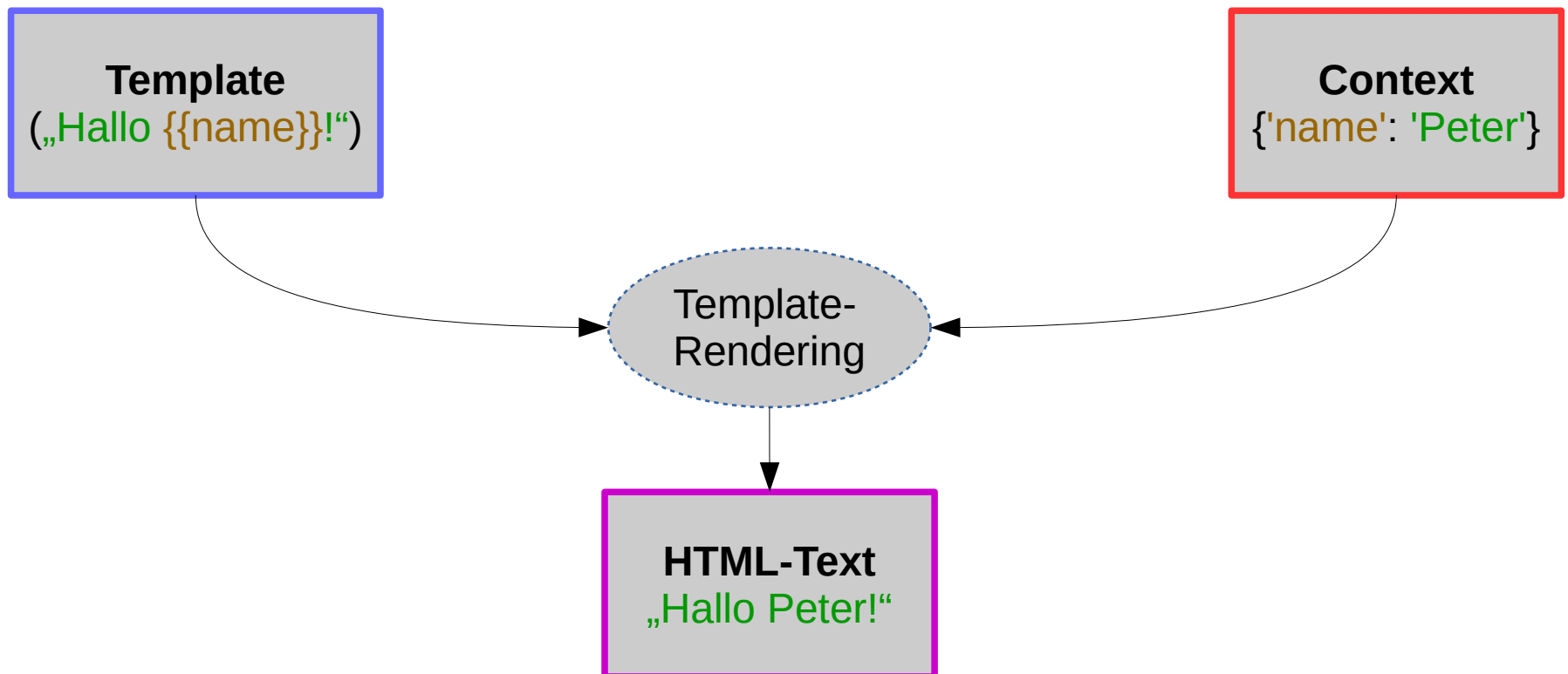
# Django: Request-Handling

---

- **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) voneinander 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.

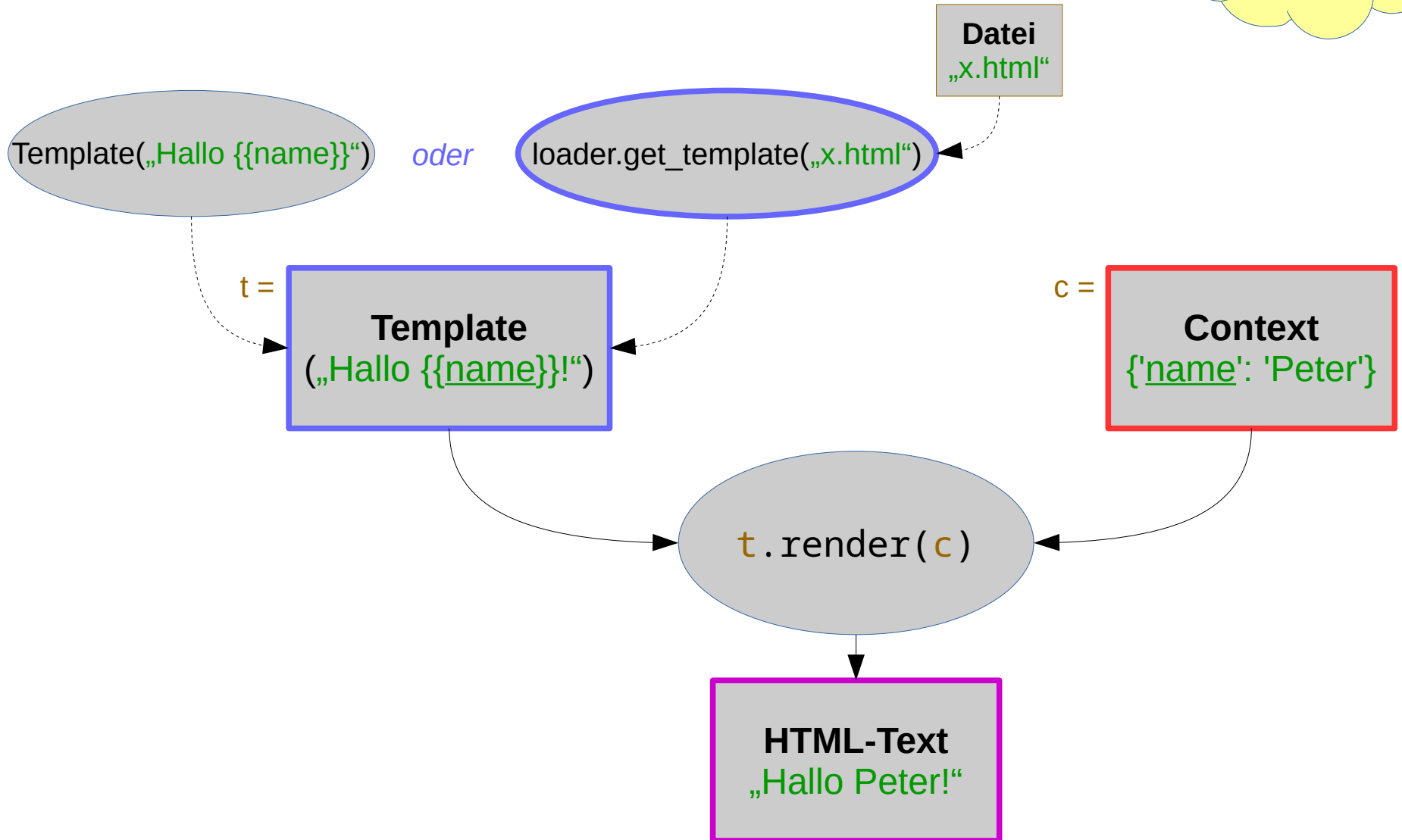
- **Das Ergebnis des Renderings ist wie erwartet**

- Das Template ist aber viel übersichtlicher als zuvor.

Django: Template-Engine

- **Datenfluss der Template-Verarbeitung**

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



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
```

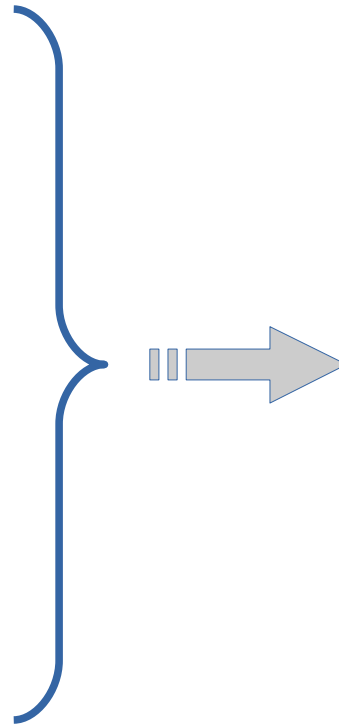
```
def show_dozenten_anzahl(request):  
    d = {'prof_count': Professor.objects.count(), }  
    context = Context(d)  
    template = loader.get_template('dozenten_anzahl.html')  
    return HttpResponse(template.render(context, request))
```

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** realisierbar

# Django: Template-Engine

---

- **Tags** können Klammern bilden oder auch nicht:
- **Manche Tags stehen alleine für sich**
  - sie bilden **keine Klammern**
  - z.B. `{% url 'impresum' %}`, 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 Stringlitterale (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 ausgegeben (*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 %}
```

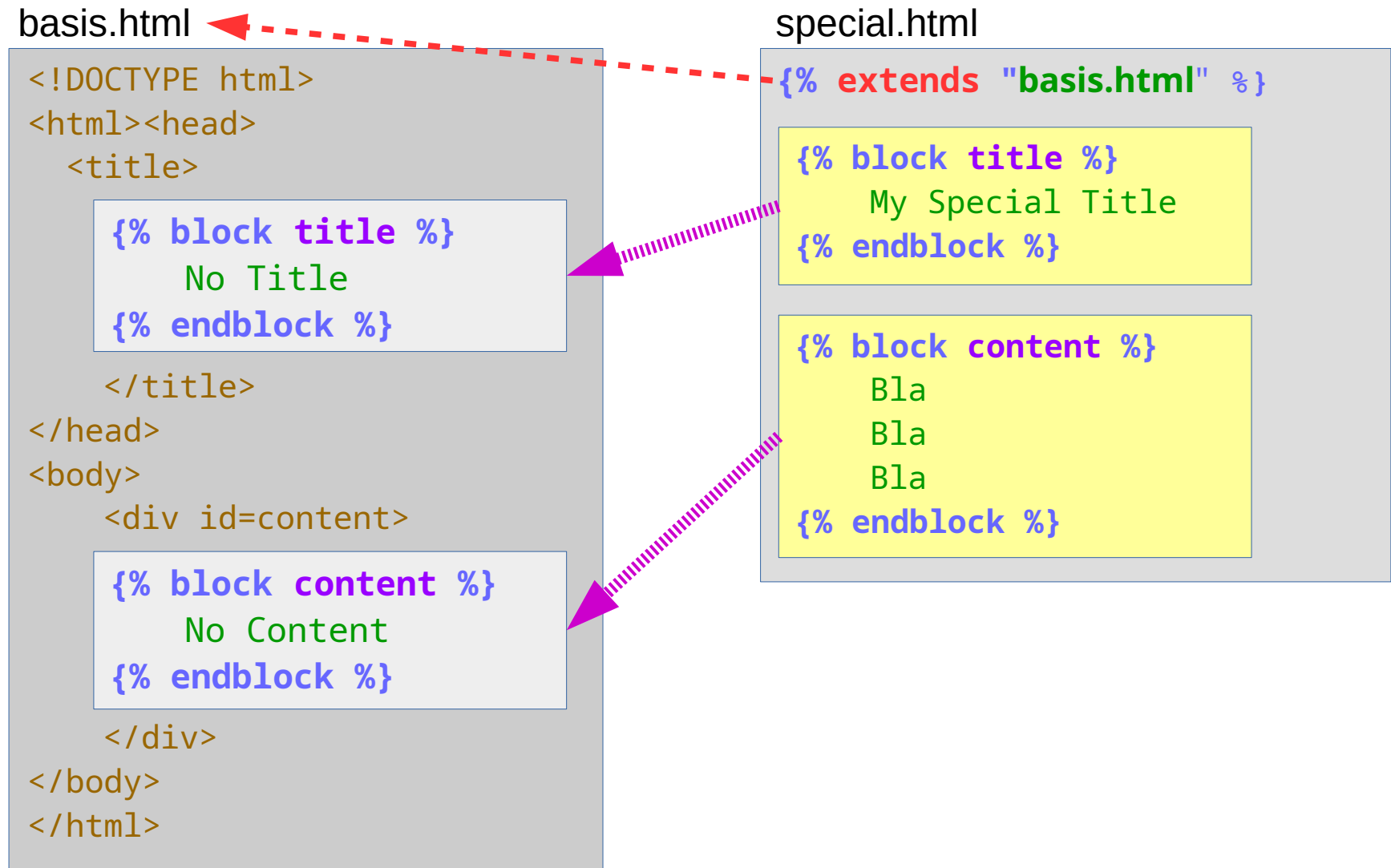
# Django: Template-Engine

---

- **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 spezielleren Templates beerben ein Basis-Template („**extends**“-Tag) und modifizieren die Blöcke
    - Sie können dabei auch auf 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



# 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“

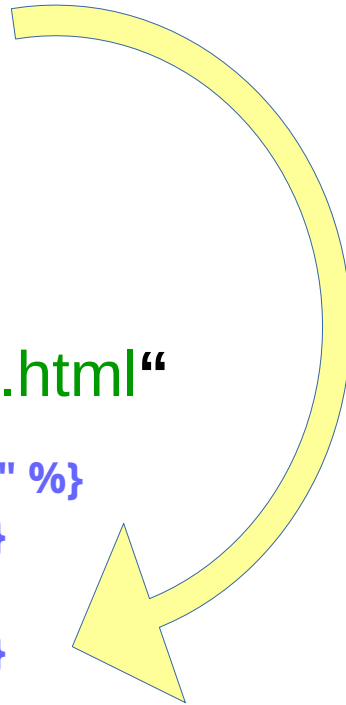
```
<!-- ... -->
<menu>

 {% block menuitems %}
 Home
 About
 {% endblock %}

</menu>
<!-- ... -->
```

- Template „special.html“

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



- Template „special.html“ entspricht:

```
<!-- ... -->
<menu>

 {% block menuitems %}
 First
 Home
 About
 Last
 {% endblock %}

</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 solte `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' %}">
```

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

Template-  
rendering

# 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.

*Tip:* 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/>