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“)

    sinnvoll?

    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