Django

Ein Python-basiertes Web-Entwicklungsframework
“for *perfectionists with deadlines*”

Warum Django?

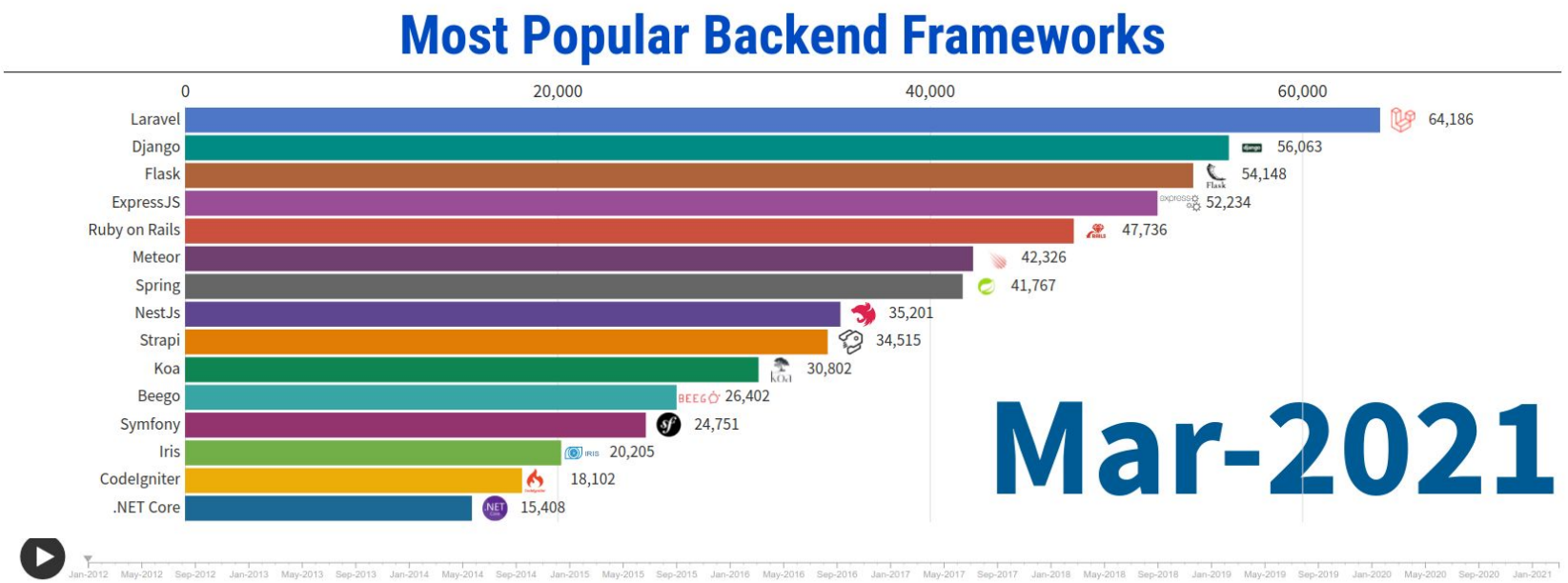
- Python-basiert
- open source
- datenbankgestützte Webanwendungen schnell & einfach entwickeln
- kein JavaScript / AJAX / JSON / php notwendig
- kaum Boilerplate, hoher Grad an Abstraktion
- klarer, intuitiver Syntax
- “batteries included” - alles dabei, was man braucht
- viele Erweiterungen
- hervorragende Dokumentation
- aktive und hilfsbereite Community

Batteries included

- leistungsstarker ORM
  - unterstützt alle gängigen Datenbanken (PostgreSQL, MariaDB, MySQL, Oracle, SQLite)
  - ORM abstrahiert Datenbank - Code unabhängig vom verwendeter DB
  - Lernen einer zusätzlichen Abfragesprache entfällt
- integrierte Template-sprache
- integriertes Usermanagement
- integrierte Admin-Konsole
- breite Palette von Sicherheits-features
  - XSS & CSRF Schutz
  - Schutz vor SQL injections
  - Host Header Validierung
  - uvm.
- umfangreiche Lokalisierungs-features
- integrierte Testfunktionen
- uvm…

in den nächsten 1 ½ Stunden zu erwarten

- Vorstellung des Frameworks
- “Gusto” machen
- interaktive Inhalte
- kurze Demonstrationen
- Zusatz (wenn Zeit)): Interaktive Websites & SPAs mit Django & HTMX

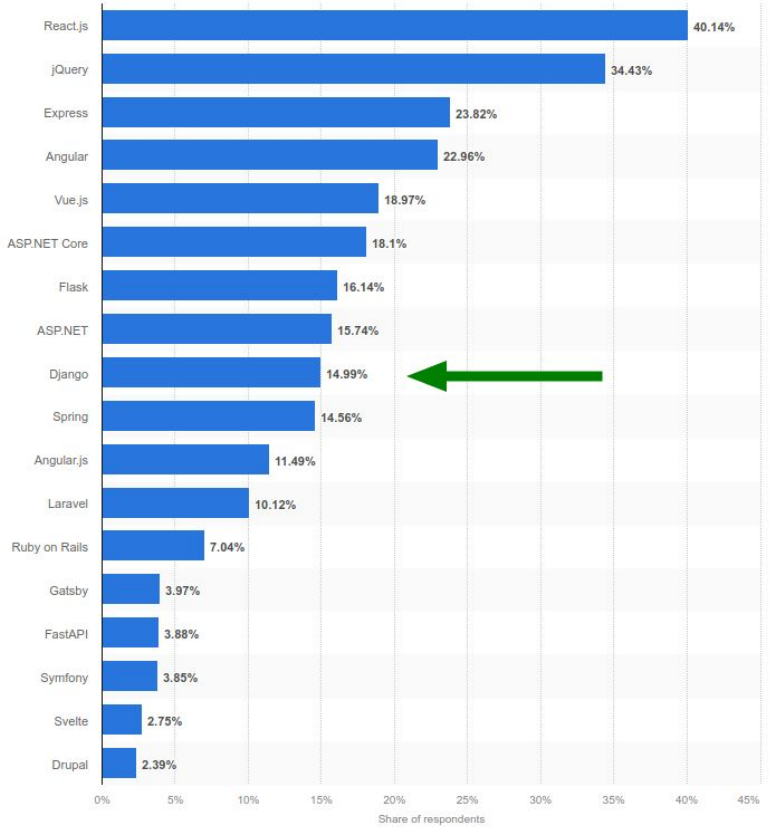


Die beliebtesten Backend-Frameworks von 2012-2021 (Quelle: https://**statisticsanddata.org**/data/most-popular-backend-frameworks-2012-2021/)

am häufigsten verwendete Web-Frameworks in 2021

In der Grafik sind auch auch reine Frontend-Frameworks wie jQuery inkludiert.

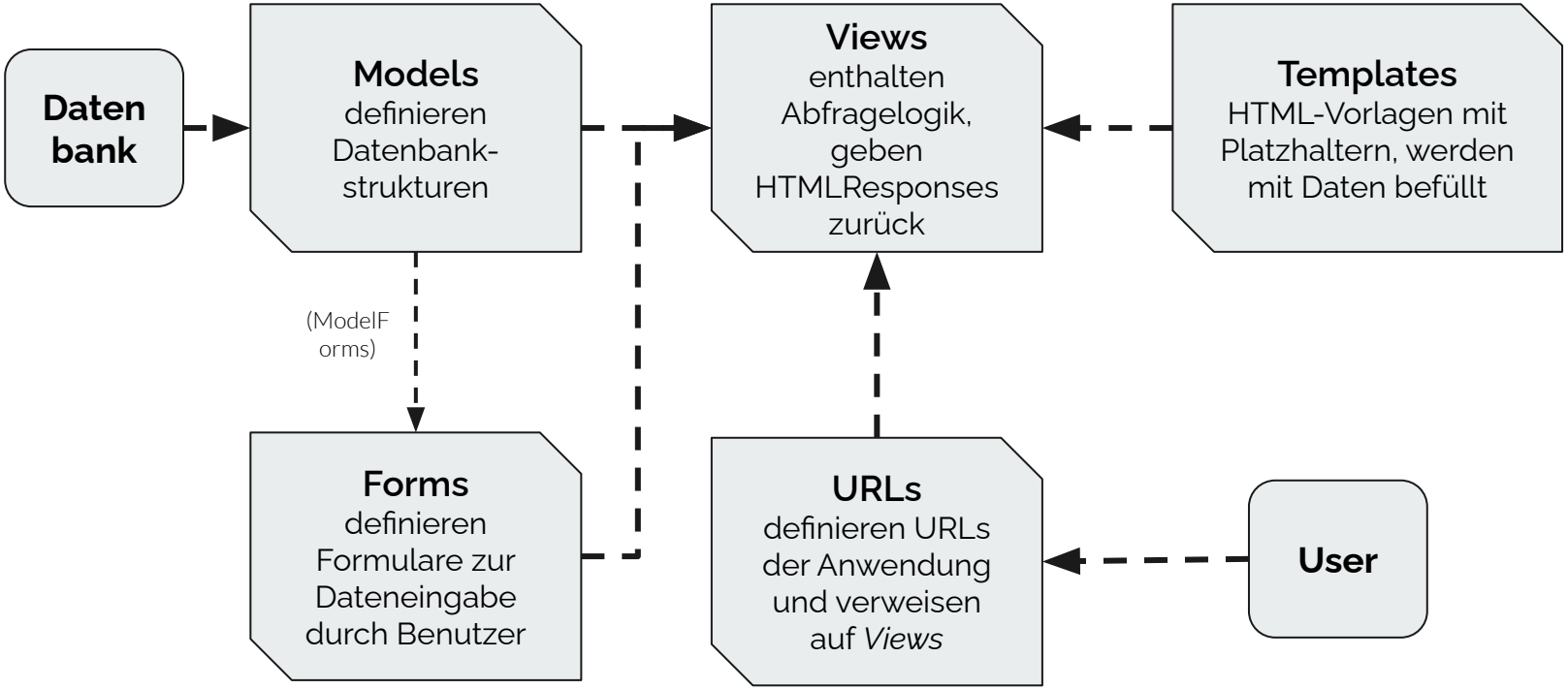
(Quelle: https://**www.statista.com**/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/)



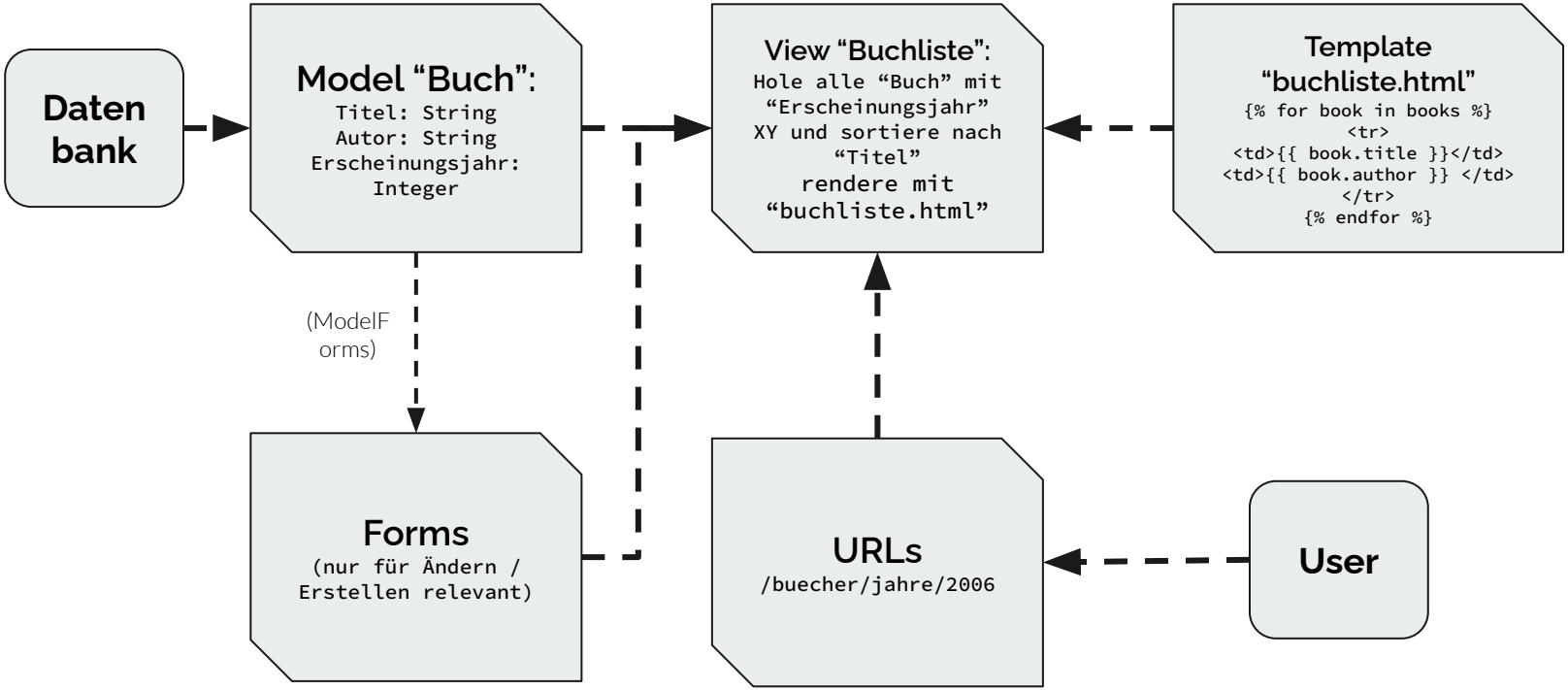
Django / Webentwicklung in der Schule

- Python ist schnell zu lernen
- hoher Abstraktionsgrad ermöglicht Konzentration auf das Wesentliche
- Webentwicklung deckt breites Spektrum an Themen und Kompetenzen ab:
  - objektorientierte Programmierung
  - relationale Datenbanksysteme
  - HTML, CSS
  - HTTP Request-Response-Zyklus
- Entwicklung von Software zur Digitalisierung div. organisatorischer Abläufe in der Schule

(Bildquelle: https://www.piqsels.com/)



wesentlichste Elemente einer Django-Applikation



wesentlichste Elemente an konkretem Beispiel

Django Syntax - ausgewählte Beispiele

“Guter Code braucht keine Kommentare”

fbr.io/tkyth

Beispiel 1: View

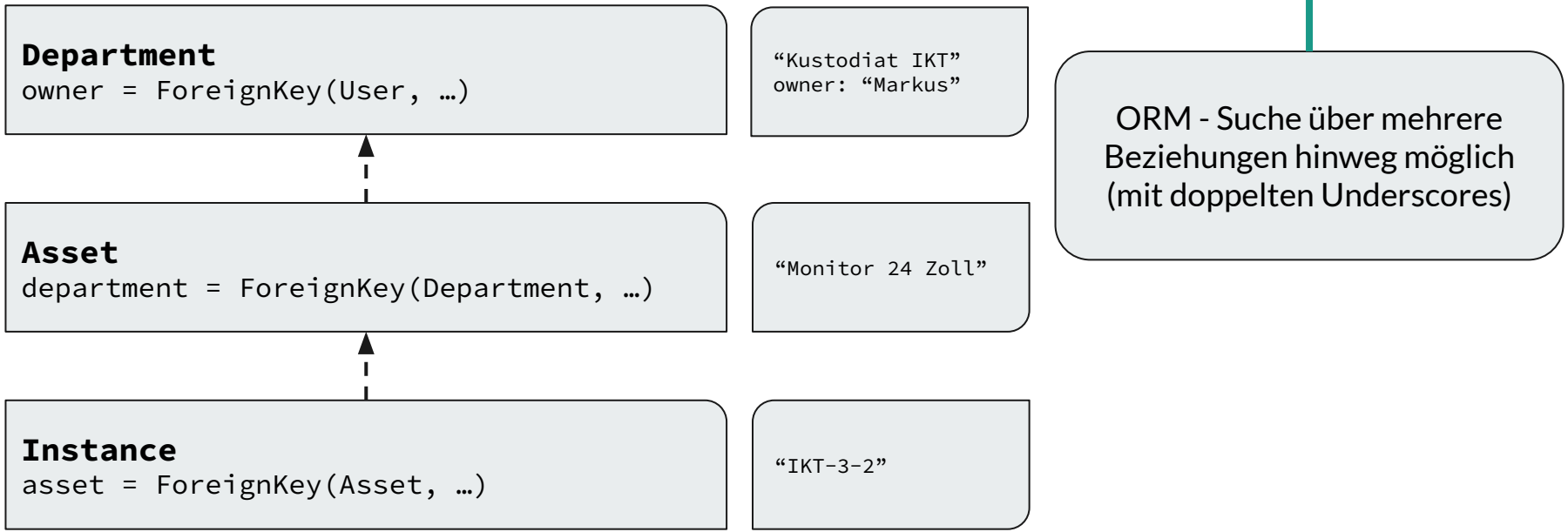
```
def asset_details(request, asset_id):
```

Funktion, die als Argument einen Web-Request (und evtl. weitere Argumente) erwartet und einen Web-Response zurückgibt.

In der Praxis üblicherweise: Aufruf über URL, Rückgabe einer HTTPResponse

```python

@login_required
def asset_details(request, asset_slug):
    asset = get_object_or_404(Asset, asset_slug=asset_slug)

    if asset.owner != request.user:
        raise PermissionDenied

    instances = asset.instances.all()

    context = {
        'asset': asset,
        'instances': instances,
    }

    return render(request, 'assets/asset_single_view.html', context)


```

```python

@login_required
def instances_per_room_list(request, room_id):
    room = get_object_or_404(Room, pk=room_id)
    instances = room.instances.filter(asset__department__owner=request.user).order_by("asset", "status")

```



```python
from django.contrib import messages
from django.contrib.auth import authenticate, login, logout

def sign_in(request):
    form = MyLoginForm(request)

    if request.method == "POST":
        username = request.POST['username']
        password = request.POST['password']
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user=user)

            if "next" in request.GET:
                return redirect(request.GET['next'])
            else:
                return render(request, 'landing_page/login_success.html')
        else:
            messages.add_message(request, messages.ERROR, "Benutzername oder Passwort falsch. Bitte versuche es noch einmal!")

    return render(request, 'landing_page/login.html', context={'form': form, })

```

login(), authenticate()
erledigen komplexe Arbeit

Beispiel: Einloggen eines Users (Passworthash überprüfen, Session-Cookie setzen etc.)

Beispiel 2: Model

```
class Asset(models.Model):
```

Definition von Datenstrukturen
(“Modellen”)

Ein *Model* entspricht i.A. einer Tabelle in der Datenbank.

Felder (=Spalten) werden einfach über Variablendefinitionen definiert.

```python
class Asset(models.Model):
    description = models.CharField(max_length=40)
    make = models.CharField(max_length=30, blank=True)
    model = models.CharField(max_length=30, blank=True)
    purchase_price = models.DecimalField(max_digits=8, decimal_places=2, default=0)
    asset_uid = models.PositiveIntegerField()
    comments = models.CharField(max_length=400, blank=True)
    notes = models.CharField(max_length=400, blank=True)
    department = models.ForeignKey(Department, on_delete=models.CASCADE, related_name="assets")
    asset_slug = models.SlugField(max_length=10, unique=True)

    def get_full_name(self):
        full_name = f"{self.description} ({self.make} {self.model})"
        return full_name

```

Beziehungen über ForeignKey / ManyToManyField erstellbar
Verhalten bei Löschen über on_delete

related_name macht Code verständlicher:
`department.assets.all()`

Funktionen in Modellen stehen in jeder Instanz des Modells zur Verfügung

```python
class Instance(models.Model):
    instance_uid = models.PositiveIntegerField()
    asset = models.ForeignKey(Asset, on_delete=models.CASCADE, related_name="instances")
    amount = models.PositiveSmallIntegerField(default=1)
    room = models.ForeignKey(Room, on_delete=models.SET_NULL, blank=True, null=True, related_name="instances")
    purchase_date = models.DateField(blank=True)
    decommission_date = models.DateField(blank=True, null=True)
    comments = models.CharField(max_length=400, blank=True)

    DECOMMISSION_REASON_CHOICES = [
        (1, _('Defective')),
        (2, _('Lost')),
        (3, _('Outdated')),
        (99, _('other')),
        (0, 'n/a'),
    ]

    decommission_reason = models.IntegerField(choices=DECOMMISSION_REASON_CHOICES, default=0)

    class Meta:
        constraints = [
            models.UniqueConstraint(fields=['instance_uid', 'asset'],
                                    name="unique UID per asset"),
        ]
        ordering = ["number"]
```

unterschiedliches Verhalten bei Löschen des übergeordneten Eintrags

choices vereinfacht Optionsfelder

constraints bietet mächtige Werkzeuge zum Setzen und Erzwingen von Regeln

Beispiel 3: Templates

`assets/asset_list.html`

HTML-Seite wird auf Basis einer *Template* erstellt. (“gerendert”)

Views übermitteln die darzustellenden Daten mit dem *context*.

Django Templates sind HTML Dateien mit speziellen Template-tags.

`{{ objekt.wert }}` Felder und sonstige mit dem *context* übermittelte Variablen

Spezielle Tags:

`{% if %} {% endif %}`

```
{% extends "assets/base.html" %}
{% block content %}

<h1>{{ department.name }}</h1>

<table class="table table-hover table-sm">
    <thead class="bg-secondary">
        <tr>
            <th scope="col">ID</th>
            <th scope="col">Asset</th>
            <th scope="col"></th>
        </tr>
    </thead>
    <tbody>
        {% for asset in assets %}
            <tr class="table-light">
                <th scope="row">{{ asset.asset_number }}</th>
                <td>{{ asset.description }}</td>
                <td>
                    <a href="{% url 'assets:asset_details' asset.id %}"
                       class="btn btn-secondary btn-sm">
                        Details
                    </a>
                </td>
            </tr>
        {% endfor %}
    </tbody>
</table>
{% endblock %}

```

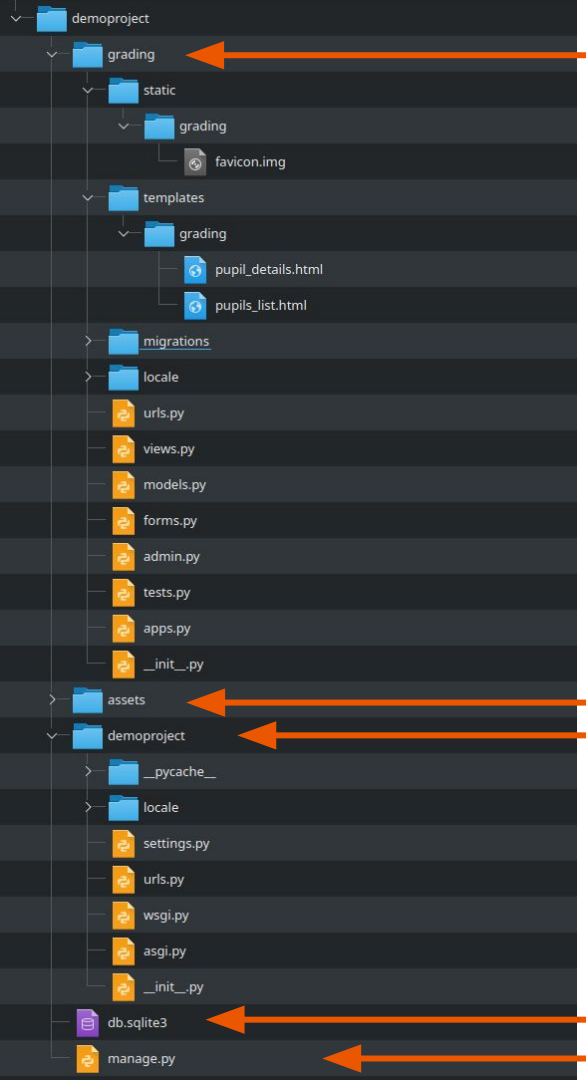
`{% url %}` unterstützt beim Generieren von URLs nach dem zentralen URL-Schema

Projekt-Struktur

project, app, manage.py…

Begriffe & Konzepte

- Project
  - Das gesamte Projekt, z.B. eine Webseite, eine Webapp mit unterschiedlichen Funktionalitäten
  - *Apps* eines *Projects* teilen sich die Datenbank, zentrale Einstellungen uvm.
- App
  - *Projects* bestehen aus einer oder mehreren *Apps*
  - bilden unterschiedliche Funktionalitäten ab, z.B. Direct-messaging, Kommentare, Inventarverwaltung…
  - “plugable” - Austausch- und wiederverwendbar - ein und dieselbe *App* kann in mehreren Projekten eingesetzt werden
  - Apps können untereinander kommunizieren / gegenseitig auf ihre Daten zugreifen



Apps- Ordner

Projekt-Ordner (zentrale Einstellungen etc.)
(Trägt selben Namen wie Projekt)

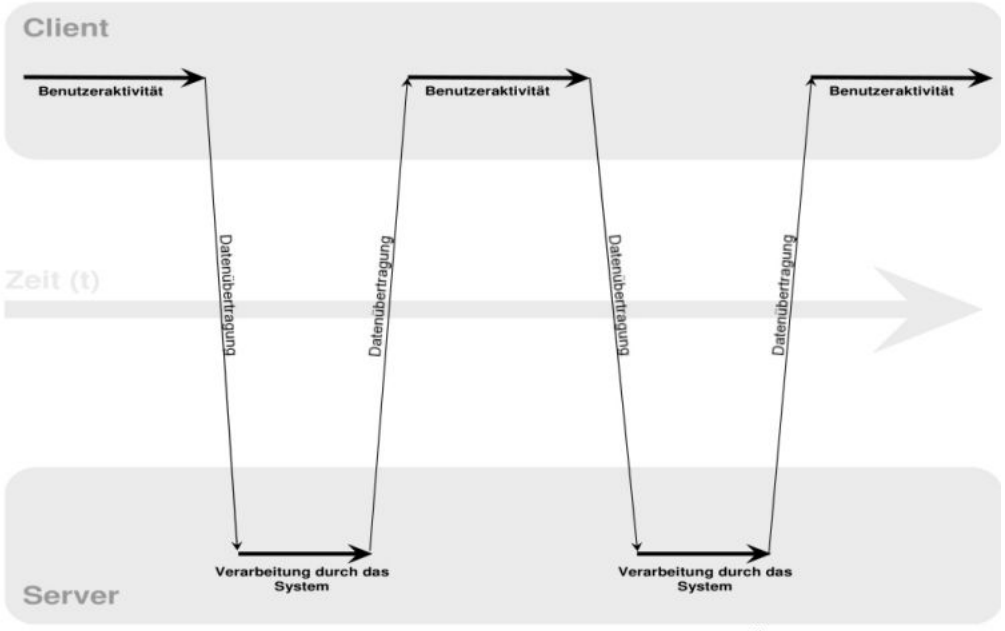
Datenbank (nur bei SQLite)

manage.py
essentielles Pythonskript für viele administrative Aufgaben

Eine wesentliche Einschränkung…

Synchrone vs asynchrone Kommunikation

Entscheidender Nachteil / Einschränkung bei Django: synchrone Client / Server Kommunikation



Bildquelle: Jesse James Garett, Ajax: A New Approach to Web Applications, adaptive path publications, 2005.

→ Django häufig ins Backend “verbannt”

- Frontend mit anderen Technologien (z.B. React, Vue)
- Django liefert Daten für Frontend (API)
- häufig auch auf unterschiedlichen Servern
- Verständlich: Django ORM ist sehr mächtig, Entwicklung sehr schnell möglich, aber synchrone Webapps nicht mehr zeitgemäß

Alternative: Django & HTMX

- immer häufiger verwendete Kombination
- problemloses Zusammenspiel mit HTML-Templates
- ermöglicht interaktive Webapps ohne JavaScript und XML
- sogar SPAs möglich

HTMX “high power tools for HTML”

- kleine, leichtgewichtige Library
- ermöglicht diversen HTML Elementen das Senden von GET oder POST requests…
- …und das Austauschen von Elementen im DOM mit der Serverantwort (als HTML)
- kein / kaum neuer Syntax, in HTML integriert

```
<!-- Load from unpkg -->
<script src="https://unpkg.com/htmx.org@1.7.0"></script>
<!-- have a button POST a click via AJAX -->
<button hx-post="/clicked" hx-swap="outerHTML">
  Click Me
</button>
```

Quelle: htmx.org