Kapitel 3:

Webserver-Frameworks: Django

Limitierungen der Low-Level-Entwicklung (1)

Die Low-Level-Entwicklung mit PHP und SQL ist bzgl. ***einer effizienten Softwareentwicklung*** *limitiert …*

- **Problem 1: Funktionalität wird durch primitive Operationen realisiert**
  - z.B. Formularverarbeitung: Jedes DB-Attribut ...
    - ... muss bei der Formularausgabe erzeugt und ggf. vorbelegt werden
    - … muss bei der Formulardatenverarbeitung geprüft und einem Datensatz-Attribut zugewiesen werden
  - Eine Erweiterung des Schemas führt zu vielen Anpassungen

- **Problem 2: Mischung von Darstellungs- und Kontroll-Ebene führt zu unübersichtlichen Strukturen**
  - Es ist oft schwer zu erkennen, was die wesentlichen Kontrollabläufe sind und ob alle Fälle vollständig behandelt sind

Limitierungen der Low-Level-Entwicklung (2)

*Die Low-Level-Entwicklung mit PHP und SQL ist bzgl. **der Sicherung der Datenintegrität** limitiert …*

- **Problem 3: Datenintegrität wird an vielen Punkten verteilt bzw. repliziert gesichert**
  - z.B. Feldlänge / Typ / Wertebereich eines Attributs eines editierbaren DB-Datensatzes ...
    - … ist im DB-Schema (SQL) festgelegt
    - … muss an HTML-Formular übergeben werden
    - ... muss bei POST-Daten geprüft werden
    - Eingabefehler sollten per Postback im Formular angezeigt werden
  - Änderung des Schemas führt dadurch zu vielen Anpassungen

Limitierungen der Low-Level-Entwicklung (3)

Die Low-Level-Entwicklung mit PHP und SQL ist bzgl. ***Schutz vor Sicherheitslücken*** ***limitiert …***

- **Problem 4: Sicherheitsprobleme müssen oft manuell und überall im Code verteilt gelöst werden**
  - vollständige Abschottung aller potentiellen Angriffsvektoren erforderlich
  - dies erfolgt auf sehr geringem Abstraktionsniveau, z.B.
    - SQL-Injections vermeiden bei textuellen Aufbau von SQL-Queries
    - XSS-Attacken vermeiden bei textuellem Aufbau von HTML-Ausgaben

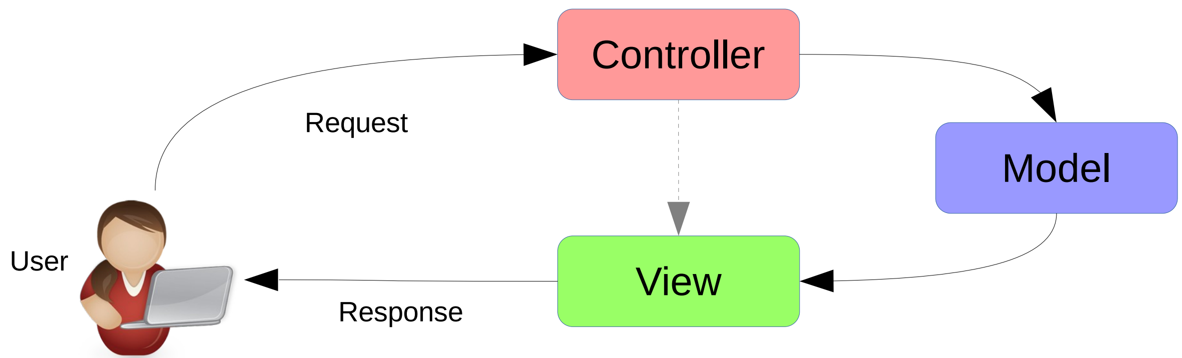
- **Problem 5: Mechanismen sind oft „Unsafe by default"**
  - Sicherheit bekommt man nur, wenn man ***alles richtig*** macht

Anforderungen

- **Wir brauchen ...**
  - **Trennung** von Modell, Präsentation und Steuerung (→ MVC)
    - Klarere Struktur der Applikation
    - Keine Durchmischung von Applikationslogik und Darstellung
  - **Abstrakte** Schema-Definition (Modell)
    - Schema-Eigenschaften werden nicht wiederholt sondern zentral definiert und gesichert (und überall benutzt)
  - Implementierung von Routine-Abläufen auf **hoher Ebene**
    - Automatisch Änderungsformulare zum Schema erzeugen, Antworten prüfen und speichern
  - **Sicherer** Umgang mit unsicheren Daten (u.a. Präsentation)
    - **Escape-by-Default** für alle unsicheren Inhalte (**Safe-by-Default**)
  - Mechanismen zur **zuverlässigen** Fehler- und Ausnahmebehandlung
    - z.B. keine unkontrollierten Ausgaben an den Nutzer (mitten in der Webseite) im Fehlerfall
  - ...

Hintergrund: MVC

- **Entwurfsmuster „Model – View – Controller“ (MVC)**
  - Idee: Klare **Trennung** von Modell, Präsentation und Steuerung
    - Das **Modell** („**model**“) hält und pflegt die **Daten**
      - Unabhängig von View und Controller
    - Die **Präsentation** („**view**“) stellt die Daten dem Benutzer dar
      - Daten werden aus dem Modell gelesen und dargestellt
    - Die **Steuerung** („**controller**“) verwaltet Modell und Präsentation
      - Sie verarbeitet und kontrolliert Änderungen und steuert auch z.B. die Einhaltung von Benutzerrechten (wer darf was sehen oder ändern)



Hintergrund: MVC

- **Entwurfsmuster „Model – View – Controller“ (MVC)**
  - Es gibt **viele Varianten** und offene Punkte dieses Modells
    - Siehe https://de.wikipedia.org/wiki/Model_View_Controller
  - *Zur Vertiefung*:
    - Wie sind die Abläufe in einem LAMP-Server hier zuzuordnen?
      - Was davon haben wir bisher davon explizit architekturell abgegrenzt?
      - Siehe Abschnitt „Serverseitige Webanwendungen“
    - Wie sind die Abläufe des Systems Webserver + Webclient hier einzuordnen?
      - Siehe Abschnitt „Zusammenspiel von Server und Browser bei Webanwendungen“
  - Viele Web-Frameworks setzen das Konzept um
    - Allerdings oft mit sehr unterschiedlichen Sichtweisen
      - Django (s.u.) hat z.B. eine eher unorthodoxe Sicht auf den Begriff „**View**“:
        - https://docs.djangoproject.com/en/4.2/faq/general/
        - https://django-book.readthedocs.io/en/latest/chapter01.html#the-mvc-design-pattern

Lösungsansätze (1)

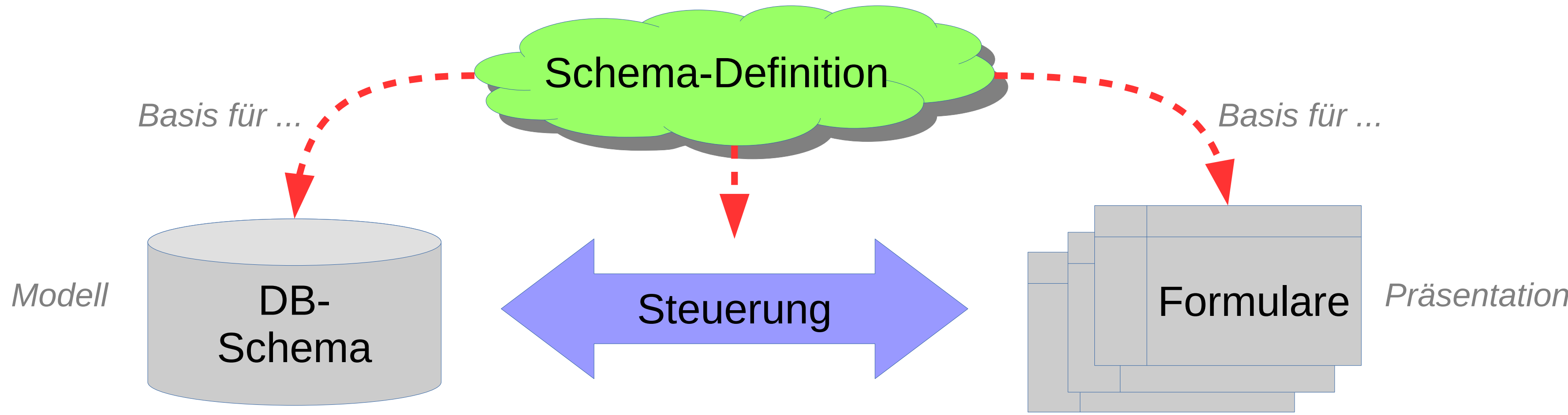
- **Trennung von Modell, Präsentation und Steuerung**
  - Separate Kontrollogik / Steuerung
    → Klare und verständliche Strukturen bei der Anfrageverarbeitung

    MVC: Controller
  - Mechanismen zur sicheren Fehlerbehandlung
    - z.B. keine unkontrollierten Ausgaben an den Nutzer im Fehlerfall
  - Nutzung von **Template-Engines** zur Präsentation

    MVC: View
    - Idee: Daten + **Template** → HTML-Response
    - Darstellungsaspekte (u.a. Design) werden im Template behandelt
    - Template- und Applikationsdesigner können getrennt / parallel arbeiten
    - Variable Designs durch umschaltbare Template-Sätze
  - Escape-by-Default für alle unsicheren Inhalte
    - Tagging (interne Markierung / Klassifizierung) von Strings
      - unsicher / sicher / bereits escaped
    - vermeidet Injection-Probleme (XSS, HTML-, SQL-Injection)

Lösungsansätze (2)

- **Nur eine Schema-Definitionen für Datenablage (Modell) und Formularbehandlung**
  - Beschreibung des Schemas auf hoher Abstraktionsebene



  - Generierung von DB-Schema und Formularen aus dem selben normativen Schema (Das DB-Schema könnte auch als normatives Schema dienen.)
  - Automatische Prüfung von Formulardaten auf Schema-Konformität
  - Bietet konsistenten / effizienten Mechanismus zu Verabeitungskette DB → Formular → Bearbeitung → Formular → DB

Lösungsansätze (3)

- **Applikationsstruktur, die Routineabläufe automatisiert**
  - „**Konvention vor Konfiguration**“
    - Softwaredesign-Paradigma („*Don´t Repeat Yourself*“)
    - http://de.wikipedia.org/wiki/Konvention_vor_Konfiguration
  - Hohes Abstraktionsniveau wo immer möglich, z.B.
    - Abstraktes Schema (einmal für Alles)
    - Abstrakte Filterung von Daten (keine komplizierten SQL-Joins)
    - Robuste URL-Analyse und -Synthese
    - Hierarchische HTML-Templates
  - Robustheit und Pflegbarkeit
    - Logging / Benachrichtigung des Administrators bei Fehlern
    - automatische Schema- und Datenmigration bei Upgrades
    - KISS („Keep it small and simple“)
  - Skalierbarkeit (Funktionen, Schema, Performanz)

Lösung: Web-Application-Frameworks

- **Web-Application-Frameworks**
  - Software, die die effiziente Entwicklung von Web-Applikationen unterstützt, vereinfacht und sicherer macht
    - Konzept: Die Mechansimen werden nur noch an den entscheidenden Punkten (Schemaentwurf, Applikationslogik, HTML-Design) angepasst
  - Realisieren die o.g. Lösungsansätze (mehr oder weniger)
    - Datenbankabstraktion, Schemamanagement, Template-Engine
    - **Scaffolding** (typische Abläufe sind sehr leicht realisierbar)
  - Es gibt eine große Zahl von ihnen, z.B.
    - ASP.NET — C# / Visual Basic (Microsoft)
    - AngularJS, NodeJS, Express — Javascript / Typescript (u.a. Google)
    - JSF — Java / JavaBeans (SUN/Oracle, IBM)
    - Zend Framework, Laravel, Symfony — PHP
    - **Django**, Flask — Python
    - Ruby on Rails — Rails
    - **...**
      - siehe http://de.wikipedia.org/wiki/Liste_von_Webframeworks oder z.B. in einem MDN Artikel

Web-Application-Framework: Django

- **Django ist ein auf der Sprache Python basierendes Web-Application-Framework**
  - Integrierter Objektrelationaler Mapper (ORM)
    - Datenbank-Backend ist austauschbar (MySQL, Postgres, Oracle, SQLite, ...)
    - Schema-Definition und DB-Zugriff erfolgen High-Level, ohne Bedarf für SQL
  - Generiert zum Schema ein komplettes Administrations-Interface
    - Basierend auf Benutzerverwaltung mit fein steuerbaren Rechten
  - Mächtige Template-Sprache
    - Trennung von Datenaufbereitung und Darstellung (mit Vererbung zwischen Templates)
  - Flexibles URL-Management
    - Mustergesteuerte URL-Zerlegung und URL-Synthese
  - Sicher vor SQL-Injections, CSRF, Schutz vor XSS
  - **...**

Programmiersprache Python

- **Python** Grundkonzepte
  - universelle Multiparadigmen-Sprache
    - imperativ, objektorientiert, funktional
  - dynamisch getypt (aber auch strikt getypt)
    - Typen sind an Daten / Objekte gebunden, nicht an Variablen
    - Ducktyping (→ Wikipedia)
  - Haupt-Ziele:
    - **Hohe Ausdrucksfähigkeit** (Probleme sind sehr kompakt lösbar)
    - **Sehr gute Lesbarkeit**
    - vielfältige Nutzbarkeit durch vielfältige Schnittstellen (Bibliotheken)
    - Erweiterbar durch Module
  - Programme müssen nicht explizit übersetzt werden
  - interaktiv nutzbar (Kommandos eingeben und direkt ausführen)
    - Python-Shell oder Erweiterung „ipython“

Programmiersprache Python

- **Beispiele**

Grundoperationen

```
[~] python3

>>> 1+2*3
7
>>> s = 'a' + "b"
>>> s
'ab'
>>> s == 'ab'
True
```

Strukturierte Anweisungen

```
>>> x = 3
>>> print(x)
3
>>> if x == 3:
...     print('x ist drei.')
... else:
...     print('x ist nicht drei.')
...
x ist drei.
```

Funktionen

```
>>> def max(a,b):
...     if a>b:
...         return a
...     else:
...         return b
...

>>> max(4,6)
6
```

Klassen, Methoden

```
>>> class C():
...      def f(self):
...          pass   // do nothing
... class D(C):
...     def f(self):
...         print("Ich bin D.")
...
>>> x = D()
>>> x.f()
Ich bin D.
```

Programmiersprache Python

- **Beispiele für Datentypen und Ausdrücke**

Tupel

```
>>> x = (0,1,2,3,4,5)
>>> x[3]
3
>>> x[-1]
5
>>> x[3:5]
(3, 4)
>>> x[3:4]
(3,)
>>> (3)
3
```

(3,) ist 1-Tupel
(3) ist Zahl

Listen

```
>>> x = [0,1,2,3,4,5]
>>> x[3]
3
>>> x.append('Y')
>>> x[3:]
[3, 4, 5, 'Y']
>>> x[:3]
[0, 1, 2]
>>> x[3:4]
[3]
```

Warum ist hier [3,] unnötig?

Dictionaries

```
>>> s = dict(a=1, b=2)
>>> s
{'a':1, 'b':2}
>>> s == {'a':1, 'b':2}
True
>>> s['a']
1
>>> s.get('c', 'unknown')
'unknown'
>>> s.items()
[('a', 1), ('b', 2)]
```

```
>>> '%s + %s = %s' % (1,2,3)
'1 + 2 = 3'

>>> a, b   =  '', 'sonst'
>>> a or b, bool(a), bool(b)
('sonst', False, True)

>>> x = 'ja' if a else 'nein'
>>> print(x)
'nein'
```

```
>>> [ x*x for x in [1,2,3] ]
[1, 4, 9]

>>> p = [ (x, x*x) for x in [1,2,3] ]
>>> p
[(1, 1), (2, 4), (3, 9)]

>>> dict(p)
{1: 1, 2: 4, 3: 9}
```

„List-Comprehension“

Programmiersprache Python

- **Python-Doku**
  - Zentrale Python-Weseite: https://www.python.org
    - Enthält auch eine interaktive Shell: https://www.python.org/shell/
    - Doku: https://docs.python.org/
    - Interakt. Tutorials: https://docs.python.org/3/tutorial/
      https://www.learnpython.org/
    - Weitere Tutorials: https://wiki.python.org/moin/BeginnersGuide/Programmers
  - Wikipedia-Seite (Überblick):

    http://de.wikipedia.org/wiki/Python_(Programmiersprache)
  - Für frühere Python-Version (Python 2.x):
    - Python 2 Quick-Reference (PQR): http://rgruet.free.fr
      - **Sehr nützliches Nachschlagewerk**, leider bisher kein Python 3 Update
      - Aktuell PQR 2.7: http://rgruet.free.fr/PQR27/PQR2.7.html

Django Quickstart

- **Django-Installation in Debian-Linux**

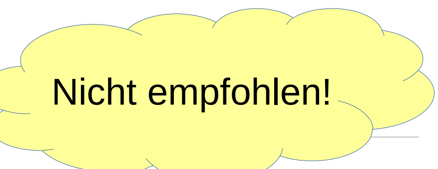
1. Grundlagen-Pakete installieren
   - Auf den Übungs-Servern (scilab-*) bereits erledigt
   - Als User root …

```
aptitude install ipython3 python3-pip gettext
    # Unterstützung für MySQL, WSGI, Sprachunterstützung
aptitude install python3-psycopg2 python-sqlite
    # Optional: Unterstützung für weitere DBMS (Postgres, SQLite)
aptitude install ipython3
    # Optional: Die Python-Shell iPython
```

2. Django installieren (2 Möglichkeiten)
   - Methode A: Als **Debian-Paket**
   - Methode B: Mit **PIP** → wählbar, z.B. Django **4.2**

– Wir benutzen hier Methode B (PIP)

Django Quickstart



- Alternative: **Django-Installation als Debian-Paket**
  - Als User root …

```
aptitude show python-django python3-django
    # Paket-Infos zu "python-django" und "python3-django" anschauen

aptitude install python3-django
    # "python-django" installieren (dauert ca. eine Minute)
```

  - Es gibt noch diverse weitere Pakete

```
aptitude search python3-django
    # ca. 50 Erweiterungs-Pakete zu Django werden aufgelistet
```

  - Nachteil: Debian-Pakete hier zum Teil nicht sehr aktuell

Django Quickstart

- Hintergrund: **Python Packet-Installer PIP**
  - PIP ist ein Python-Werkzeug um Python-Pakete zu installieren
    - Funktioniert Unabhängig von den Debian-Paketen
    - Bietet sehr große Zahl von Paketen, meist sehr aktuelle Versionen
  - ggf. zunächst **PIP** installieren: Als User root …

```
aptitude install python3-pip
    # Werkzeug pip installieren
```

    - Auf den Übungs-Servern (scilab-*) bereits erledigt

Django Quickstart

- Hintergrund: **Python Packet-Installer PIP**
  - PIP-Pakete können an 2 Orten installiert werden …
    - ins **System** (als User root)
    - in ein **Benutzer-Verzeichnis** (als regulärer Benutzer)

Ab jetzt am Besten als regulärer User

  - Bei Bedarf zunächst **PIP** Paket-Liste aktualisieren

```
pip3 install --upgrade pip --user
    # als User (oder als root ohne --user ins System installieren)
```

Immer `pip3` benutzen (nicht `pip`)

  - Paket-Index von PIP (Suche nach Paketen)
    - https://pypi.org/search/
    - z.B. zu Django:
      https://pypi.org/search/?q=Django
  - Dokumentation zu PIP:
    - https://pip.pypa.io/en/stable/user_guide/

Django Quickstart

- **Django-Installation mit PIP**
  - **Django** durch PIP installieren: Als regulärer User …

```
pip3 install Django==4.2 --user
    # Django Version 4.2.x in ~/.local/ installieren
```

    - PIP installiert so die Pakete im Benutzer-Verzeichnis `~/.local/`
  - Damit die Programme gefunden werden, muss `~/.local/bin` in den Programm-Suchpfad aufgenommen werden
    - Abhängig von der Shell
    - Für csh / tcsh: Eintrag `set path=( ~/.local/bin $path )`

      → systemweit in `/etc/csh.cshrc` (systemweit, als root)

      → oder benutzerlokal in `~/.cshrc`

      evtl. direkt nach der Installation 'rehash' um das neue Programm zu finden
    - Auf den Übungs-Servern (scilab-*) systemweit bereits erledigt
  - Danach kann man als User 'django-admin' aufrufen

Web-Application-Framework: Django

- **Django: Es gibt sehr Ausführliche Dokumentation**
  - Homepage: https://www.djangoproject.com/
    - Online-Doku Django 4.2: https://docs.djangoproject.com/en/4.2/
  - The Django Book: https://django-book.readthedocs.io/
    - Freies Online-eBook, Tutorial
  - Aktive Community
    - Viele (zentral und dezentral gepflegete) Module
    - z.B. Django-Snippets (Tricks & Erweiterungen)
      - https://djangosnippets.org/
    - z.B. Django im **Python Package Index** (https://pypi.org/)
      - https://pypi.org/search/?q=Django

Django Quickstart

- **Neues Projekt anlegen (als User)**
  - `mkdir django ; cd django`
    - # unser Verzeichnis für unsere Django-Projekte
  - `django-admin startproject test1`
    - # Wir legen ein neues Projekt "test1" an
  - `cd test1 ; dir -R`

```
# Schauen wir uns an …
test1                                     <-- wir sind hier (aktuelles Verzeichnis)
       manage.py                          <-- unser Management-Programm
       test1                              <-- das Config-Verzeichnis des Projekts
              __init__.py
             settings.py                  <-- hier macht man Einstellungen
             urls.py
             wsgi.py
```

  - `python3 ./manage.py`
    - chmod u+x ./manage.py   # Script vorher ggf. noch ausführbar machen
    - Evtl. in der ersten Zeile „python“ gegen „python3“ austauschen
    - # ab jetzt rufen wir dieses Script nur noch mit „`./manage.py`“ auf

Django Quickstart (falls SQLite)

- **Django ist vorkonfiguriert für SQLite als Datenbank**
  - **SQLite** benötigt keinen eigenen Datenbank-Server
    - Die Daten werden in lokalen Dateien abgelegt
    - Siehe https://www.sqlite.org/docs.html
  - In einfachen Szenarien genügt SQLite zum Betrieb
    - Geringe Last, geringe Anforderungen an Zuverlässigkeit, ...
  - # Auszug aus **`test1/settings.py`**

```
DATABASES = { 'default': {
    'ENGINE': 'django.db.backends.sqlite3',
    'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
} }
```

Name der SQLite-Datei

  - `./manage.py migrate`
    - Aktualisiert / Erzeugt Datenbank-Schema
  - `./manage.py createsuperuser`
    - # Legt Admin-Account an (Name, Mailadresse und neues Passwort angeben)

Django Quickstart (falls SQLite)

- **Wie sieht die SQLite-Datenbank jetzt aus?**
  - ./manage.py dbshell
    - # nur interessehalber mal anschauen …
      .tables

```
auth_group
auth_group_permissions
auth_permission
auth_user
auth_user_groups
auth_user_user_permissions
django_admin_log
django_content_type
django_migrations
django_session
```

    - # Inhalt der Benutzerdatenbank ...
      .schema **auth_user**
      SELECT * FROM auth_user;
    - Tipp: evtl. muss man als User root die sqlite3-Tools installieren

```
aptitude install sqlite3
    # Kommando-Frontend für sqlite3 installieren
```

DB-Shell, ruft Shell zum konfigurierten DB-Backend auf (Aufruf unabhängig vom DB-Backend)

Die Kommandos **in** der DB-Shell hängen aber natürlich vom DBMS ab (hier: **SQLite**)

Tipp: Mit den Kommandos „.headers on“ und „.mode columns“ wird die Ausgabe lesbarer. (Kann man auch in ~/.sqliterc schreiben.)

Django Quickstart (falls mySQL)

- **Alternative: z.B. mysql-Datenbank**
  - EDIT **test1/settings.py**
    - # Editiere folgende Zeilen um die Datenbank anzubinden:

```
DATABASES = { 'default': {
    'ENGINE':   'django.db.backends.mysql',
    'NAME':     'test1',
    'USER':     'lamp',
    'PASSWORD': 'xxx_MEIN_PASSWORT_xxx',   } }
```

  - mysql
    - CREATE DATABASE test1;
      # wir legen noch die o.g. DB an
      # ab jetzt benutzen wir statt "mysql" das Kommando "./manage.py dbshell"
  - ./manage.py migrate
    - Aktualisiert / Erzeugt Datenbank-Schema
    - Die SQL-Operationen kann man vorher mit „./manage.py sqlmigrate“ sehen
  - ./manage.py createsuperuser
    - # Legt Admin-Account an (Name, Mailadresse und neues Passwort angeben)

Django Quickstart (falls mySQL)

- **Wie sieht die mysql-Datenbank jetzt aus?**
  - `./manage.py dbshell`
    - # nur interessehalber mal anschauen …
      SHOW tables;

```
+----------------------------+
| Tables_in_test1            |
+----------------------------+
| auth_group                 |
| auth_group_permissions     |
| auth_permission            |
| auth_user                  |
| auth_user_groups           |
| auth_user_user_permissions |
| django_admin_log           |
| django_content_type        |
| django_migrations          |
| django_session             |
+----------------------------+
```

    - # Inhalt der Benutzerdatenbank ...
      DESCRIBE **auth_user**;
      SELECT * FROM auth_user \G

DB-Shell, ruft Shell zum konfigurierten DB-Backend auf (Aufruf unabhängig vom DB-Backend)

Die Kommandos **in** der DB-Shell hängen aber natürlich vom DBMS ab (hier: **mySQL**)

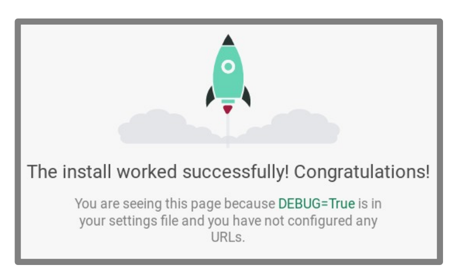
Django Quickstart

- **Start des Test-Servers auf Port 8000**
  - `./manage.py runserver 0.0.0.0:8000`
    - # Wir starten die Web-Applikation
      # Und können auf der Adresse des Servers mit einem Webbrowser
      # darauf zugreifen, z.B.
- **Web-Client-Zugriff auf den Test-Server**
  - z.B. http://scilab-0100.cs.uni-kl.de:8000/

    → Fehlermeldung
  - EDIT **test1/settings.py**
    - # Editiere folgende Zeilen um den Server-Hostnamen freizugeben, z.B.:

      ALLOWED_HOSTS = ['scilab-0100.cs.uni-kl.de']
  - Nochmal http://scilab-0100.cs.uni-kl.de:8000/

    → „*The install worked successfully!*“



Django Quickstart

- **Das Admin-Interface ist ein Django-App**
  - Man kann eigene Apps schreiben oder existierende nutzen
  - Auszug aus `test/settings.py`
    - ```
      INSTALLED_APPS = [
          'django.contrib.admin',
          'django.contrib.auth',
          'django.contrib.contenttypes',
          'django.contrib.sessions',
          'django.contrib.messages',
          'django.contrib.staticfiles',
      ]
      ```
  - Auszug aus `test1/urls.py`
    - ```
      urlpatterns = [
          path('admin/', admin.site.urls),
      ]
      ```
  - Zugriff auf Admin-Interface
    - z.B. http://scilab-0100.cs.uni-kl.de:8000/admin/