Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Schweizer Armee
Kommando Cyber

EINFÜHRUNG IN DJANGO

Lehrgang ICT-Systemspezialist
Junior / 01.2023

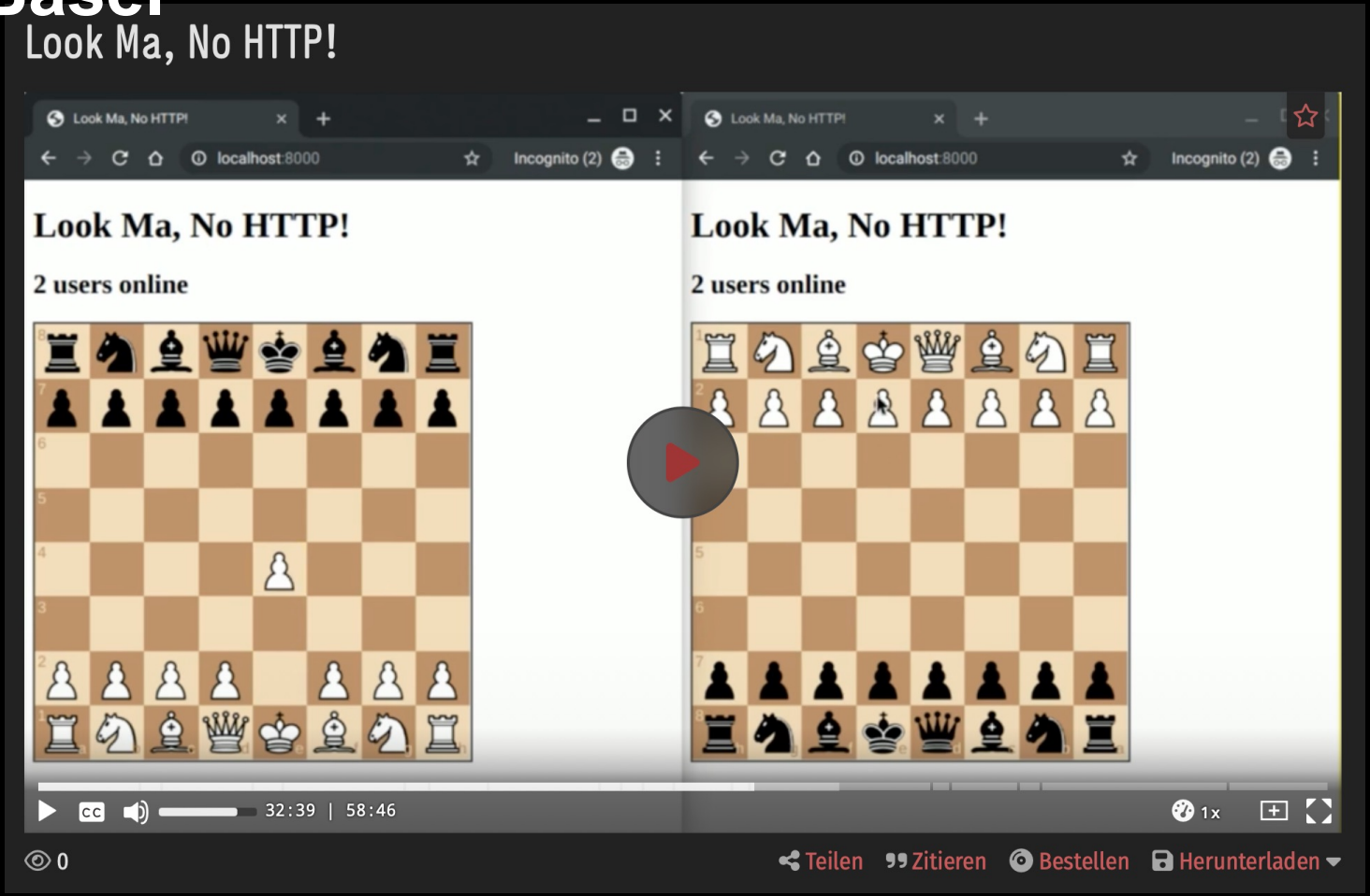
Dr. Thomas Staub, Leiter ICT Warrior Academy

Repetition

Kahoot Spiel

Nice to know

Live coding session by Miguel Grinberg at Europython 2019 in Basel



Einführung

Lernziele

- Kennenlernen des Python Web Frameworks Django
- VirtualEnv verstehen
- Django ORM kennenlernen, eigene Modelle erstellen und mittels ORM anwenden
- Selbst eine erste Django WebApp erstellen

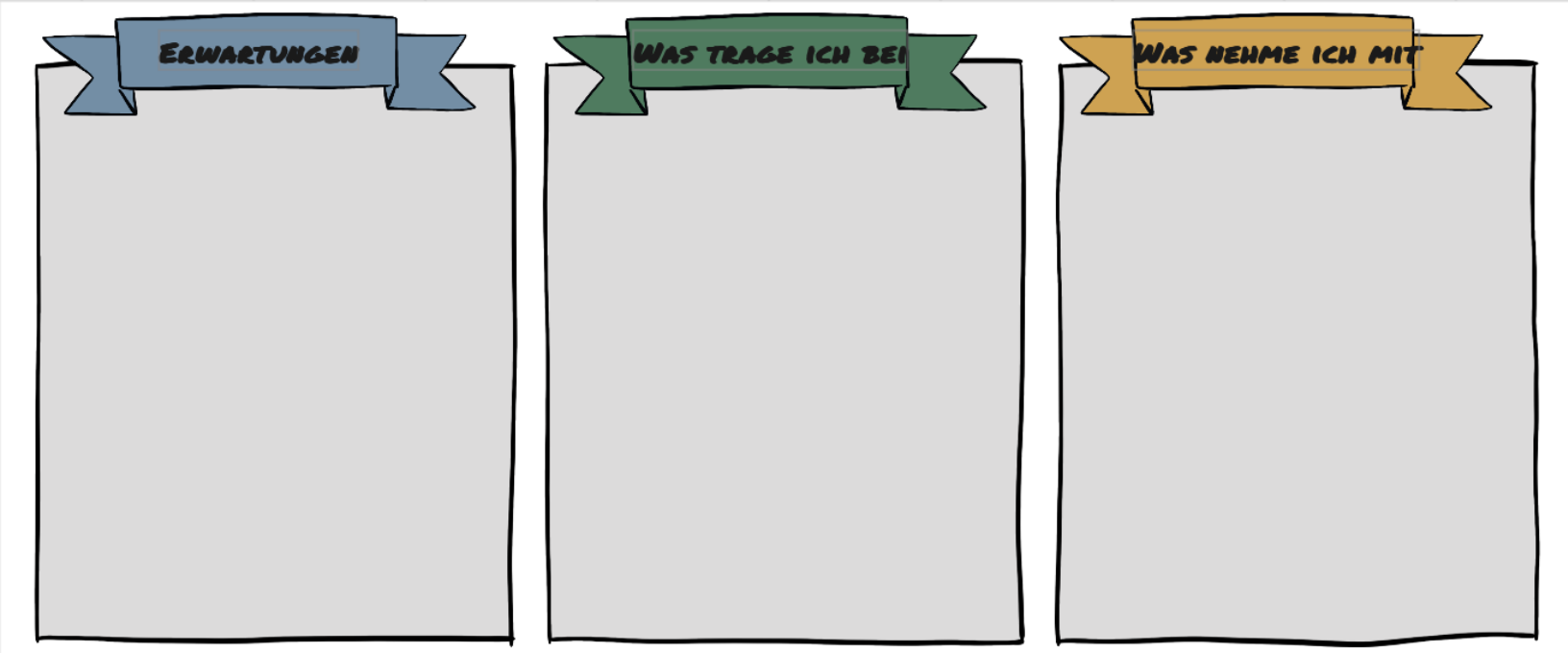
Repetition Python Zen
Django
Vorbereiten der Projektumgebung
- Virtual Environment
- Visual Studio Code
- Installieren von Django

Minimale Django App
Debugger Launch Profile
Templates
Static Files
Data Models + Migrations
Django API → python manage.py shell
Django Admin
Forms

Tag 5 / 6

Erwartungen / Eigener Beitrag

- Miro Board (Passwort: Python2024): https://miro.com/app/board/uXjVN87Q-Tc=/?share_link_id=210521214832



Repetition

Python Zen

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
Now is better than never.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.

Django

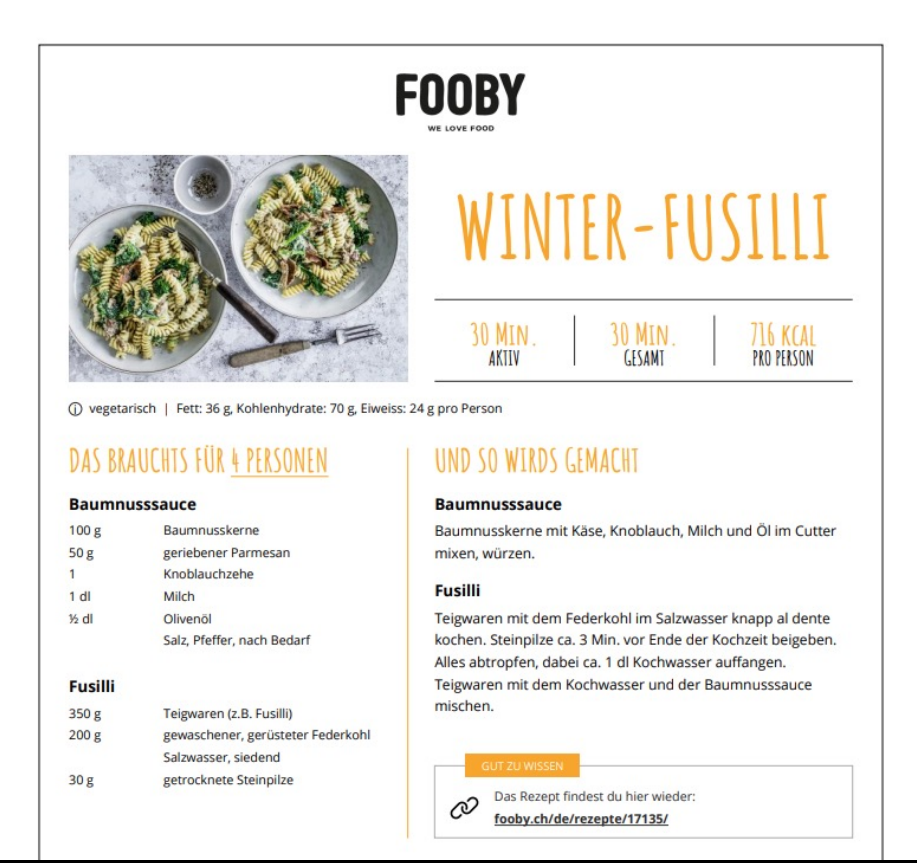
- https://www.djangoproject.com

- Gut geeignet für Rapid Prototyping
- Viele Extras für die meisten Web Tasks
- Sicherheit von Grund auf
- Skalierbarkeit
- Sehr vielfältig

- ➔ vorallem einfach zu benutzen ☺

Projekt

- Digitales Rezeptbuch



FOOBY
WE LOVE FOOD

WINTER-FUSILLI

30 MIN. AKTIV | 30 MIN. GESAMT | 716 KCAL PRO PERSON

vegetarisch | Fett: 36 g, Kohlenhydrate: 70 g, Eiweiss: 24 g pro Person

DAS BRAUCHTS FÜR 4 PERSONEN

Baumnusssauce

100 g	Baumnusskerne
50 g	geriebener Parmesan
1	Knoblauchzehe
1 dl	Milch
½ dl	Olivenöl
	Salz, Pfeffer, nach Bedarf

Fusilli

350 g	Teigwaren (z.B. Fusilli)
200 g	gewaschener, gerüsteter Federkohl
	Salzwasser, siedend
30 g	getrocknete Steinpilze

UND SO WIRDS GEMACHT

Baumnusssauce

Baumnusskerne mit Käse, Knoblauch, Milch und Öl im Cutter mixen, würzen.

Fusilli

Teigwaren mit dem Federkohl im Salzwasser knapp al dente kochen. Steinpilze ca. 3 Min. vor Ende der Kochzeit beigeben. Alles abtropfen, dabei ca. 1 dl Kochwasser auffangen. Teigwaren mit dem Kochwasser und der Baumnusssauce mischen.

GUT ZU WISSEN

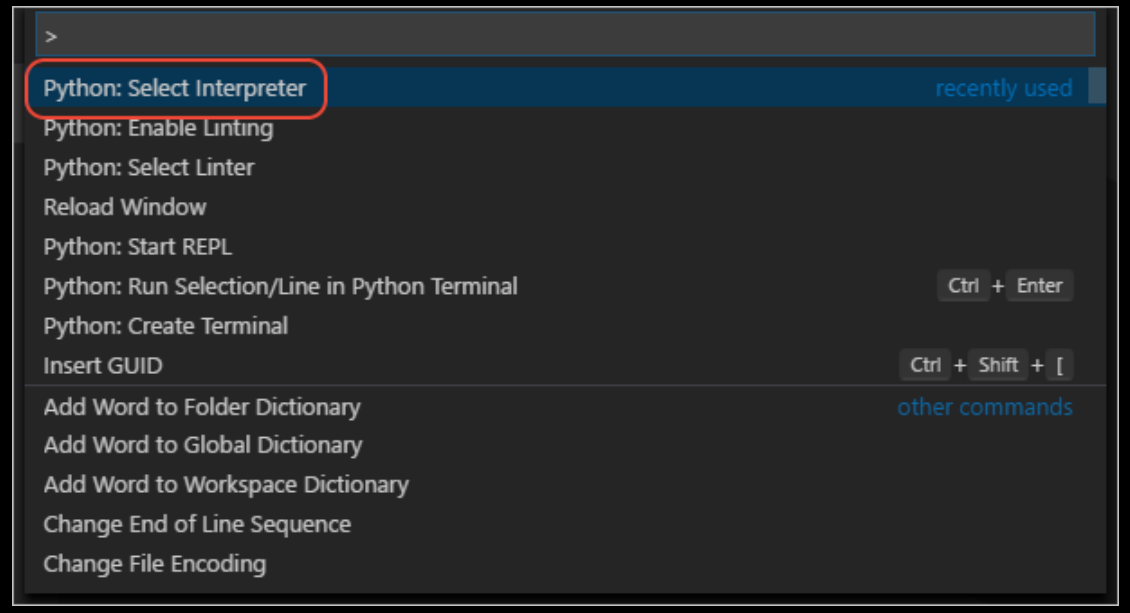
Das Rezept findest du hier wieder:
fooby.ch/de/rezepte/17135/

Vorbereiten der Projektumgebung

- Projektordner “recipe_book” erstellen
  - 🚀 `$ mkdir recipe_book`
- virtualenv
  - Tool um isolierte Python Umgebungen zu erstellen
  - Nur die notwendigen Bibliotheken in der für die Applikation richtigen Version installieren resp. zur Verfügung stellen
- Installation (Mac / Linux):
  - Linux: $ sudo apt-get install python3-dev
  - 🚀 `$ python3 -m venv recipes-env`
- Tipp: virtualenvwrapper → einfacher
- 🚀 `$ mkvirtualenv recipe-env`

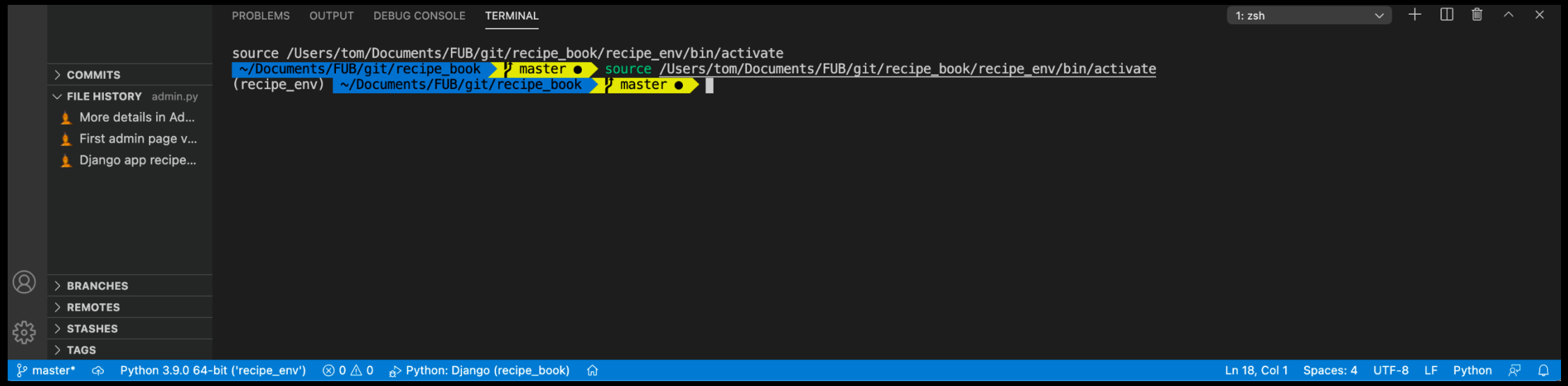
Integration in Visual Studio Code

- 🚀 `$ code .`
- Öffnen der Command Palette (Crtl + Shift + P resp. Command + Shift + P) und Selektieren des Python Interpreters (… venv), Python Pfad sollte mit ./recipe-env starten



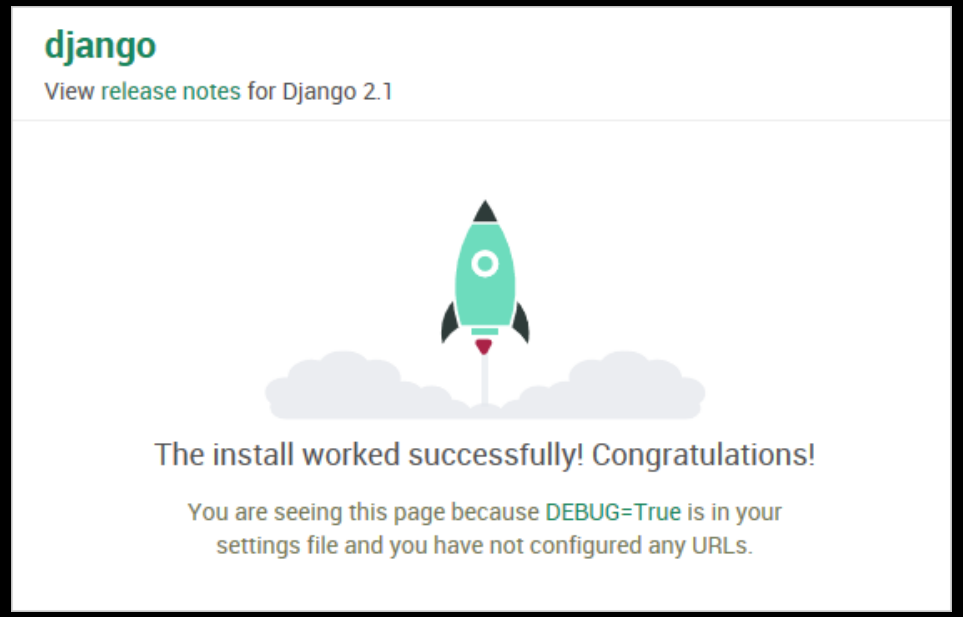
Integration in Visual Studio Code

- Starten des VSC “Terminal: Create New Terminal in Active Terminal” (Crtl + Shift + `)
- Terminal mit entsprechendem Interpreter sollte starten



Install Django / Erstellen des ersten Projektes

- $ python -m pip install Django
- $ django-admin startproject **recipe_book_web** .
  - ACHTUNG: “.” am Schluss!
- Starten des Debug Server
  $ python manage.py runserver



Erstellen der ersten Django App

- Mittels “startapp” die Grundstruktur erstellen
  $ python manage.py startapp **recipes**
- Anpassen des Views (recipes/views.py)

```
from django.shortcuts import render
from django.http import HttpResponse

# Create your views here.
def home(request):
    return HttpResponse("Hello, here my recipe book will be located")
```

Erstellen der ersten Django App

- Erstellen der Datei recipes/urls.py

```
from django.urls import path
from recipes import views

urlpatterns = [
    path("", views.home, name="home"),
]
```

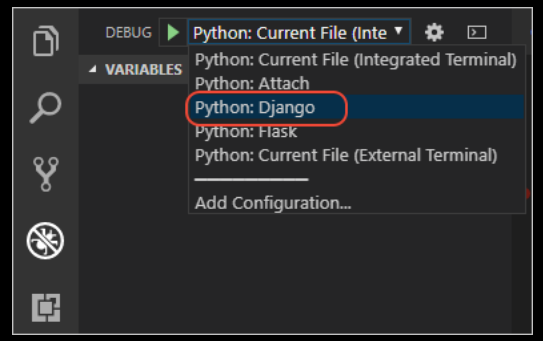
- Anpassen der Datei recipe_book_web/urls.py

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path("", include("recipes.urls")),
]
```

Debugger Launch Profile

- “Gear / Debug” Icon auswählen, launch.json erstellen und nach der Zeile mit “program” folgende Zeile hinzufügen: ”console": "integratedTerminal",



- Speichern
- Aus Dropdown Liste “Python: Django” auswählen
- Jetzt lässt sich Django “manage.py runserver” aus dem Debugger ausrufen

git

- Git Cheat Sheet: https://about.gitlab.com/images/press/git-cheat-sheet.pdf

git → .gitignore Datei

```
__pycache__/
*.py[cod]
*$py.class

#virtualenv
recipe_env/


# Django stuff:
*.log
local_settings.py

# Jupyter Notebook
.ipynb_checkpoints

# pyenv
.python-version

.vscode

.DS_Store
*.sqlite3
media/
*.pyc
*.db
*.pid
```

git

Initialisieren des Repositories sowie Linken des Remote

- 🚀 recipe_book $ git init
- 🚀 recipe_book $ git remote add origin https://gitlab.com/thomas.st/recipe_book.git

- 🚀 recipe_book $ git add .gitignore
- 🚀 recipe_book $ git commit -m"Added .gitignore"

- 🚀 recipe_book $ git push --set-upstream origin master

Erste Django App

- <server:port>/hello/<name>
- Anpassen recipes/urls.py (→ Route hinzufügen)

```
path("hello/<name>", views.hello, name="hello"),
```

Erste Django App

- Anpassen des Views (recipes/views.py):

```
import re
from datetime import datetime
from django.shortcuts import render
from django.http import HttpResponse

# Create your views here.
def home(request):
    return HttpResponse("Hello, here my recipe book will be located")

def hello(request, name):
    now = datetime.now()
    formatted_now = now.strftime("%A, %d %B, %Y at %X")

    # Filter the name argument to letters only using regular expressions. URL arguments
    # can contain arbitrary text, so we restrict to safe characters only.
    match_object = re.match("[a-zA-Z]+", name)

    if match_object:
        clean_name = match_object.group(0)
    else:
        clean_name = "Friend"

    content = "Hello " + clean_name + "! It's " + formatted_now
    return HttpResponse(content)
```

Erste Django App - Templates

- Hinzufügen von recipes zu INSTALLED_APPS in der Datei recipe_book_web/settings.py
- $ mkdir –p recipes/templates/recipes
- hello.html

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>Recipe Book</title>
   </head>
   <body>
      <strong>Hello {{ name }}!</strong> It's {{ date | date:"l, d F, Y" }} at {{ date | time:"H:i:s" }}
   </body>
</html>
```

Erste Django App - Templates

- views.py

```
def home(request):
    return HttpResponse("Hello, here my recipe book will be located")

def hello(request, name):
    return render(
        request,
        'recipes/hello.html',
        {
            'name': name,
            'date': datetime.now()
        }
    )
```

Static files

- Unveränderte Dateien, welche so wie sie sind vom Webserver ausgeliefert werden sollen (z.B. CSS)
- recipe_book_web/urls.py
  - Add import

    from django.contrib.staticfiles.urls import staticfiles_urlpatterns
  - Add at the end

    urlpatterns += staticfiles_urlpatterns()

Static files

- Im Template auf Static Files verweisen
  - $ mkdir –p recipes/static/recipes
  - Datei site.css im neuen Verzeichnis erstellen

```
.message {
    font-weight: 600;
    color: blue;
}
```

  - Bei templates/recipes/hello.html nach <title> hinzufügen:

```
{% load static %}
<link rel="stylesheet" type="text/css" href="{% static 'recipes/site.css' %}" />
…
<strong class='message'>
```

Static files

- Prod: alle statischen Dateien im gleichen Verzeichnis (dezidierter Server)
  1. recipes/settings.py ergänzen mit
     STATIC_ROOT = os.path.join(BASE_DIR, 'static_collected’)
  2. $ python manage.py collectstatic

- Collectstatic sollte bei jeder Änderung der statischen Inhalte resp. mindestens vor den Deployment auf Prod ausgeführt werden

Mehrere Templates

- Gemeinsame Teile auslagern
- Block Tag
  {% block <name> %} {% endblock %}
- Erstellen des layout.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8"/>
    <title>{% block title %}{% endblock %}</title>
    {% load static %}
    <link rel="stylesheet" type="text/css" href="{% static 'recipes/site.css' %}"/>
</head>

<body>
<div class="navbar">
    <a href="{% url 'home' %}" class="navbar-brand">Home</a>
    <a href="{% url 'about' %}" class="navbar-item">About</a>
    <a href="{% url 'contact' %}" class="navbar-item">Contact</a>
</div>

<div class="body-content">
    {% block content %}
    {% endblock %}
    <hr/>
    <footer>
        <p>© ICT Warrior Academy 2020</p>
    </footer>
</div>
</body>
</html>
```

Mehrere Templates

- site.css

```
.navbar {
    background-color: lightslategray;
    font-size: 1em;
    font-family: 'Trebuchet MS', 'Lucida Sans Unicode', 'Lucida Grande', 'Lucida Sans', Arial, sans-serif;
    color: white;
    padding: 8px 5px 8px 5px;
}

.navbar a {
    text-decoration: none;
    color: inherit;
}

.navbar-brand {
    font-size: 1.2em;
    font-weight: 600;
}

.navbar-item {
    font-variant: small-caps;
    margin-left: 30px;
}

.body-content {
    padding: 5px;
    font-family:'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
}
```

Templates

- about.html

```
{% extends "recipes/layout.html" %}
{% block title %}
About us
{% endblock %}
{% block content %}
<p>ICT Warrior Academy</p>
{% endblock %}
```

Mehrere Templates

- Erstellen der folgenden Dateien basierend auf layout.html:
  - home.html
  - about.html
  - contact.html
- Routen zu urls.py hinzufügen

```
path("about/", views.about, name="about"),
path("contact/", views.contact, name="contact"),
```

- Ergänzen von views.py

```
def about(request):
    return render(request, "recipes/about.html")
…
```

Data Models + Migrations

- Daten in DB gespeichert
- Wie können Daten aus der DB representiert werden?
- Django model := von django.db.models.Model abgeleitete Python Klasse
- **models.py**

- Welche Modelle brauchen wir für unser Rezeptbuch?

Data Models + Migrations

- “models that you define in code”
- Ablauf
  1. models.py anpassen
  2. Migrations erstellen
     $ python manage.py makemigrations
  3. Migrations anwenden
     $ python manage.py migrate

- Unterstützte DBs: SQLite, PostgreSQL, MySQL, SQLServer, …

Data Models + Migrations

- Typen von Feldern:
  - CharField, TextField
  - EmailField
  - URLField
  - IntegerField, DecimalField, FloatField
  - BooleanField
  - DateTimeField
  - ForeignKey, ManyToMany
- Attribute: max_length, blank=True (field is optional), null=True (value is optional), choices

Data Models + Migrations

- Aufgabe:
  - Erstellt mal die Modelle / Klassen für unsere Rezeptbuch App

Data Models + Migrations (models.py)

```
class Recipe(models.Model):
    name = models.CharField(verbose_name="Name des Rezepts", max_length=20, help_text="Name des Rezepts")
    created = models.DateTimeField(verbose_name="Erstellungsdatum",
                                   default=timezone.now, help_text="Das Rezeot wurde zu dieser Zeit erstellt.")

    EASY = "easy"
    INTERMEDIATE = ”medium"
    HARD = ”hard"


    DIFFICULTY_CHOICES = (
        (EASY, "Einfach"),
        (INTERMEDIATE, "Mittel"),
        (HARD, "Schwierig"),
    )
    difficulty = models.CharField(max_length=10, verbose_name="Schwierigkeitsgrad", choices=DIFFICULTY_CHOICES,
                                  help_text="Schwierigkeitsgrad des Rezepts")

    instructions = models.TextField(verbose_name="Anleitung", help_text="So wird es gemacht")
    hints = models.TextField(verbose_name="Hinweise", help_text="Das muss man beachten")
```

Data Models + Migrations (models.py)

```
class Ingredient(models.Model):
    name = models.CharField(verbose_name="Name des Zutat", max_length=20,
                            help_text="Name der Zutat")
    quantity = models.DecimalField(verbose_name="Menge", max_digits=6,
                            decimal_places=2,
                            blank=True, null=True, help_text='Mengenangabe')
    recipe = models.ForeignKey(Recipe, default=None, blank=True, null=True,
                            on_delete=models.SET_NULL)
```

Data Models + Migrations

- $ python manage.py makemigrations
- $ python manage.py migrate

- __str__() Methode zu den Modellen / Klassen hinzufügen

Data Models + Migrations

- $ python manage.py shell

from recipes.models import Recipe, Ingredient

```
>>> from recipes.models import Recipe, Ingredient
>>> Recipe.objects.all()
<QuerySet []>
>>> r = Recipe(name="Tom's Spezial", difficulty=Recipe.HARD, instructions="bla", hints="bla")
>>> r.save()
>>> Recipe.objects.all()<QuerySet [<Recipe: Tom's Spezial>]>
>>> Ingredient.objects.all()
<QuerySet []>
>>> i = Ingredient(name="Safran", quantity=1.5, recipe=r)
>>> i.save()
>>> i
<Ingredient: Safran>
>>> i.recipe
<Recipe: Tom's Spezial>
>>> i2 = Ingredient(name="Kardamon", quantity=2, recipe=r)
>>> i2.save()
>>> for ingredient in r.ingredient_set.all():
...     print(ingredient)
...
Safran
Kardamon
>>> r.ingredient_set.all().count()
2
```

Django Admin - Philosophy

- Generating admin sites for your staff or clients to add, change, and delete content is tedious work that doesn’t require much creativity. For that reason, Django entirely automates creation of admin interfaces for models.
- Django was written in a newsroom environment, with a very clear separation between “content publishers” and the “public” site. Site managers use the system to add news stories, events, sports scores, etc., and that content is displayed on the public site. Django solves the problem of creating a unified interface for site administrators to edit content.
- The admin isn’t intended to be used by site visitors. It’s for site managers.

Django Admin

- Erstellen eines Admin Users

```
$ python manage.py createsuperuser
Username (leave blank to use 'tom'): tom
Email address: thomas.staub@vtg.admin.ch
Password:
Password (again):
Superuser created successfully.
```

- http://127.0.0.1:8000/admin

Django Admin

- Registrieren von Recipe / Ingredient in admin.py

```
admin.site.register(Recipe)
admin.site.register(Ingredient)
```

- http://127.0.0.1:8000/admin

Django Admin (→ Inline Interface)

- Verbessertes Admin Interface (admin.py)

```
from django.contrib import admin

from .models import Recipe, Ingredient

# Register your models here.
admin.site.register(Ingredient)

class IngredientInlineAdmin(admin.StackedInline):
    model = Ingredient
    show_change_link = True
    extra=0
    fields = (('name', 'quantity’),)

@admin.register(Recipe)
class RecipeAdmin(admin.ModelAdmin):
    list_display = ('name', 'created', 'difficulty', 'instructions', 'hints')
    inlines = (IngredientInlineAdmin,)
```

- http://127.0.0.1:8000/admin

Django Admin

- Passt das Admin Interface so an, dass ihr Eure Datenmodelle mit Daten befüllen könnt?
- Könnt ihr dies auch in «manage.py shell»?

Benutzen der DB durch Models

- home.html

```
…
{% block content %}
<h2>List of recipes</h2>
{% if recipe_list %}
<table class="recipe_list">
<thead>
<tr>
…
</tr>
</thead>
<tbody>
{% for recipe in recipe_list %}
<tr>
            <td>{{ recipe.created | date:'d M Y' }}</td>
            <td>{{ recipe.created | time:'H:i:s' }}</td>
            <td>{{ recipe.name }}</td>
            <td>{{ recipe.difficulty }}</td>
            <td>{{ recipe.instructions }}</td>
            <td>{{ recipe.hints }}</td>
</tr>
{% endfor %}
</tbody>
</table>
{% else %}
            <p>No data found.</p>
{% endif %}
{% endblock %}
```

Benutzen der DB durch Models

- HomeListView → views.py

```
from recipes.models import Ingredient, Recipe
from django.views.generic import ListView


# Create your views here.
class HomeListView(ListView):
        """Renders the home page, with a list of all recipes."""
        model = Recipe

        def get_context_data(self, **kwargs):
                context = super(HomeListView,
                        self).get_context_data(**kwargs)
                return context
```

Benutzen der DB durch Models

- HomeListView → urls.py

```
from recipes.models import Recipe

home_list_view = views.HomeListView.as_view(
      queryset = Recipe.objects.all()[:5], # :5   limits the
results to the five most recent
      context_object_name="recipe_list",
      template_name="recipes/home.html",
)

urlpatterns = [
      path("", home_list_view, name="home"),
      …
```

Benutzen der DB durch Models

- forms.py

```
from django import forms
from recipes.models import Ingredient

class IngredientForm(forms.ModelForm):
    class Meta:
        model = Ingredient
        fields = ("name", "quantity",)
```

Benutzen der DB durch Models

- site.css

```
.recipe_list th,td {
    text-align: left;
    padding-right: 25px;
}
```

Benutzen der DB durch Models

- Erstellen eines neuen Templates enter_ingredient.html

```
{% extends "recipes/layout.html" %}
{% block title %}
    Enter an ingredient
{% endblock %}
{% block content %}
    <form method="POST" class="ingredient-form">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit" class="save btn btn-
default">Enter</button>
    </form>
{% endblock %}
```

Benutzen der DB durch Models

- Hinzufügen eines Views “enter_ingredient”
- Anpassen urls.py
- Anpassen templates/recipes/layout.html (→ In NavBar hinzufügen)

Benutzen der DB durch Models

- enter_ingredient → views.py
- ```
  from django.shortcuts import render, redirect
  from recipes.forms import IngredientForm
  ```
- ```
  ...
  def enter_ingredient(request):
      form = IngredientForm(request.POST or None)

      if request.method == "POST":
          if form.is_valid():
              ingredient = form.save(commit=False)
              ingredient.save()
              return redirect("home")
      else:
          return render(request,"recipes/enter_ingredient.html",
                        {"form": form})
  ```

Benutzen der DB durch Models

- urls.py

```
…
path("enter_ingredient/", views.enter_ingredient,
name="enter_ingredient")
…
```

Benutzen der DB durch Models

- Hinzufügen eines Bildes zu Recipe Model (→ models.py)

```
image = models.ImageField(upload_to='images',
verbose_name="Bild", blank=True, null=True,
help_text="illustrierendes Bild")
```

- Installieren der notwendigen Library
  $ python -m pip install Pillow

Benutzen der DB durch Models

- Media Root (→ settings.py)

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'
```

Benutzen der DB durch Models

- home.html

```
{% if recipe.image %}
<td><img src={{ recipe.image.url }}></td>
{% else %
<td></td>
{% endif %}
```

requirements.txt

- Abhängigkeiten des Python Environments speichern

  $ pip freeze > requirements.txt

- Abhängigkeiten des Python Environments laden

  $ pip install -r requirements.txt

Aufgabe

- Erstellen eines ersten Prototypen der Rezepteverwaltung
  - Ergänzen der notwendigen Felder zu Recipe + Ingredient
  - Darstellung der Rezepte als Liste (ListView)
  - Darstellung eines einzelnen Rezepts (Recipe)
  - Erstellen, Bearbeiten und Löschen von Recipe / Ingredient

Links

- Django Documentation

Tag 5 / 6

Review

- Miro Board (Passwort: Python2024): https://miro.com/app/board/uXjVN87Q-Tc=/?share_link_id=210521214832