Webserver mit Flask

Basics

Flask: Minimales Beispiel

```python
from flask import Flask

# Konfiguration des Servers
app = Flask(__name__)


# Definition einer 'Route'
@app.route('/')
def route_index():
    return 'Hello, world!'


# Starten des Servers
if __name__ == "__main__":
    app.run()
```

Anlegen eines neuen Flask-Projekts mit PyCharm über

- `File` → `New Project`
- Template `Flask` auswählen

Dadurch werden sinnvolle Default-Einstellung (zum Beispiel Ausführung als Flaskserver) ausgewählt.

Routen

Eine Route ordnet über einen Decorator (`@app.route(`*`url`*`)`) eine URL einer Python-Methode zu.

Beim Aufruf der URL wird dann die Methode aufgerufen und sollte die Antwort auf den Request und einen Statuscode (Standard: 200 [OK]) zurückliefern.

Als Rückgabetypen auf einen Request sind verschiedene Optionen möglich:

- Text
- Html
- Xml, Json
- ...

```python
@app.route('/')
def route_index():
    return 'Hello, world!'
```

Der Browser kann den Rückgabetyp meist korrekt interpretieren und darstellen.

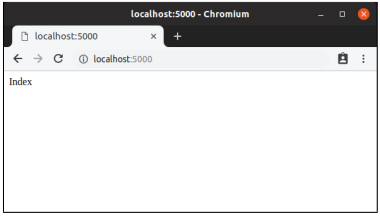
Beispiel: Routen in Flask

```python
from flask import Flask, render_template

app = Flask(__name__)


@app.route('/')
def route_index():
    return 'Index'


@app.route('/test')
def route_test():
    return render_template('test.html')
```



Dynamische Anteile in Routen

Um nicht mehrere Routen für ähnliche Ressources (zum Beispiel Waren in einem Shop) schreiben zu müssen, können Routen Parameter enthalten:

```
@app.route('/products/<int:id>')
def route_product_by_id(id: int):
    # find product
    product: Product = find_product_by_id(id)
    # return representation of product
    # function render_product has to be implemented!
    return render_product(product)
```

Diese werten an Hand des angegebenen Datentyps geparst und an die Funktion weitergegeben.

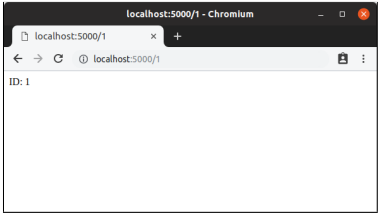
Beispiel: Dynamische Routen in Flask

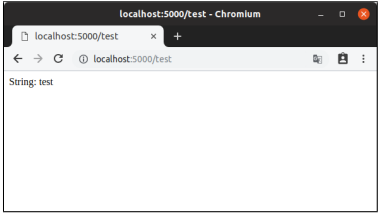
```python
from flask import Flask

app = Flask(__name__)


@app.route('/<int:id>')
def route_with_int(id: int):
    return "ID: {}".format(id)


@app.route('/<string:my_str>')
def route_with_string(my_str: str):
    return "String: {}".format(my_str)
```





Aufgabe: Flask Basics

Erstellen Sie einen Flask-Server, der den Nutzer grüßt. Diesem soll in der Url `'/'` ein String übergeben werden.

Achten Sie hierbei darauf, dass ein Leerzeichen in einer URL mit `%20` escaped werden muss.

Für „Max Mustermann“ muss die URL also zum Beispiel `http://localhost:5000/Max%20Mustermann` lauten.

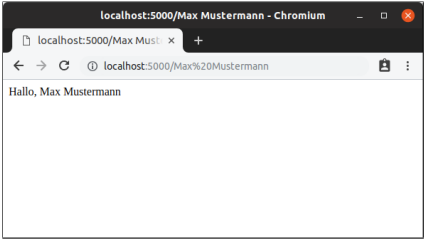
Lösung: Flask Basics

```python
from flask import Flask

app = Flask(__name__)


@app.route('/<string:who>')
def route_greet(who: str):
    return "Hallo, {}".format(who)


if __name__ == "__main__":
    app.run()
```



Jinja - Templatesystem

Rückgabe von HTML

Bisher geben wir immer nur reinen Text zurück.

Häufiger will man jedoch HTML zurückgeben, um dem Nutzer eine interessantere Erfahrung zu bieten.

Dafür gibt es in Flask die Funktion `render_template`, mit der - nach dem Import - ein HTML-Dokument (eigentlich: -Template) angezeigt werden kann.

Die Templates werden standardmäßig im Ordner `templates/` gesucht.

Die Funktion setzt außerdem verschiedene Metadaten in der Antwort, so dass der Client (Browser) weiß, dass es sich um HTML handelt.

Rückgabe von HTML

main.py

```python
from flask import Flask, render_template

app = Flask(__name__)


@app.route('/')
def route_index():
    return render_template('index.html')
```

templates/index.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<p>Dies ist ein Test</p>
</body>
</html>
```

Dynamisches HTML, serverseitig

Meist wird ein Webserver an eine Datenbank oder eine andere Datenhaltungsmöglichkeit angebunden, da sich die Daten einer Webanwendung ändern können.

Da ein festes HTML-Dokument meist sehr schnell nicht mehr den aktuellen Stand repräsentiert, wird meist für jede Anfrage (nach dynamischen Daten) ein neues HTML-Dokument generiert.

In diesem werden die (zum Beispiel aus der Datenbank ausgelesenen) aktuellen Daten in HTML repräsentiert.

In Flask gibt es dafür verschiedene Template-Systeme. Normalerweise wird `Jinja` benutzt, das auch bei Django zum Einsatz kommt.

Beispiel: Jinja-Template

```
<!doctype html>
<html lang="de">
<head>
    <title>Jinja-Biespiel</title>
</head>
<body>
{# Kommentar: product_names ist eine List[str] #}
{% if product_names %}
    {% for product_name in product_names %}
        <p>{{ product_name }}</p>
    {% endfor %}
{% else %}
    <p>Es konnten keine Produkte gefunden werden!</p>
{% endif %}
</body>
</html>
```

Beispiel: Jinja, Aufruf

```python
from typing import List
from flask import Flask, render_template

app = Flask(__name__)


@app.route('/products')
def route_products():
    products: List[str] = ['Apfel', 'Birne', 'Ei']
    # Übergebe products an template unter dem name product_names
    return render_template('jinja_example.html', product_names=products)
```

Jinja: Befehle und Kontrollstrukturen

Befehle werden in Jinja mit geschweiften Klammern eingeleitet und beendet:

- `{# Kommentare … #}` werden vom Templateprozessor ignoriert
- `{% Befehl %}`: Statements dienen vor allem zur Ausführung von Kontrollstrukturen wie `if - else` oder `for - in`, die mit `endif` beziehungsweise `endfor` abgeschlossen werden
- `{{ Ausdruck }}`: Ausdrücke werden „in das fertige Dokument geschreiben“

(Dynamische) URLs: Motivation

In reinem HTML werden URL *hartkodiert* in das Dokument geschrieben.

```
<p><a href="/products/1">1. Banane</a></p>
<p><a href="/products/2">2. Apfel</a></p>
<p><a href="/products/3">3. Orange</a></p>
```

Ändern sich diese, müssen alle Dokumente angepasst werden. Außerdem muss jeder Link einzeln von Hand eingegeben werden.

Daher können URL in Flask (inklusive möglicher dynamischer Teile) mit der Funktion `url_for` dynamisch generiert werden. Diese bekommt den Namen der Funktion übergeben, der die zu generierende URL zugeordnet ist. URL-Parameter werden als benannte Parameter übergeben.

Python: Dynamische URLs

```python
@app.route('/products/<int:product_id>')
def route_product_by_id(product_id: int):
    product = find_product_by_id(product_id)
    return render_product(product)
```

```jinja
{% for product in products %}
    <p>
        <a href="{{ url_for('route_product_by_id', product_id=product.id) }}">
            {{ product.id }}. {{ product.name }}
        </a>
    </p>
{% endfor %}
```

Aufgabe: Dynamische Webseiten

Gegeben sei eine Klasse `User` mit den Attributen `id`, `username`, `name` und `birthday`. Erstellen Sie einen Webserver (Routen + Templates) nach folgender Spezifikation:

- Die Url `/users` soll eine Übersicht über alle Nutzer (als Liste) anzeigen. Für einen Nutzer soll hier nur die ID und der Nutzername als Link auf die Nutzer (siehe unten) ausgegeben werden.
- Die Urls `/users/<id>` soll die Daten eines einzelnen Nutzer (mit der entsprechenden ID) anzeigen.

Speichern Sie die Nutzer als Variable (zum Beispiel einer Liste oder einem Dict), um in den verschiedenen Methoden die selben Daten zur Verfügung zu haben.

Lösung: Dynamische Webseiten, Klasse `User` und statische Variable

```
class User:
    def __init__(self, id: int, username: str, real_name: str, age: int
        self.id: int = id
        self.username: str = username
        self.real_name: str = real_name
        self.age: int = age


all_users: Dict[int, User] = {
    1: User(1, 'j.bond', 'James Bond', 51),
    2: User(2, 'j.bourne', 'Jason Bourne', 56),
    3: User(3, 'j.bauer', 'Jack Bauer', 48)
}
```

Lösung: Dynamische Webseiten, Nutzerliste (Ausschnitt)

```python
@app.route('/users')
def route_users():
    return render_template('users.html', users=all_users)
```

```jinja
{% if users %}
  <ul>
    {% for id, user in users.items() %}
      <li>
        <a href="{{ url_for('route_user_by_id', id=id) }}">{{ user.username }}<
      </li>
    {% endfor %}
  </ul>
{% else %}
  <p>Es konnten keine Nutzer gefunden werden!</p>
{% endif %}
```

Lösung: Dynamische Webseiten, Nutzerübersicht (Ausschnitt)

```python
@app.route('/users/<int:id>')
def route_user_by_id(id: int):
    user = all_users.get(id)
    if user is None:
        return "TODO: no such user..."

    return render_template('user.html', user=user)
```

```jinja
<p>ID: {{ user.id }}</p>
<p>Nutzername: {{ user.username }}</p>
<p>Realer Name: {{ user.real_name }}</p>
<p>Alter: {{ user.age }}</p>
```

Statische Dateien

Ein Server liefert nicht nur HTML-Dateien aus, sondern auch statische Dateien, die sich nicht ändern. Darunter fallen zum Beispiel:

- CSS-Dateien
- Javascript-Dateien
- Bilder

Diese werden in Flask üblicherweise im Ordner `static` gespeichert und können in den Templates über die Funktion

```
url_for('static', filename='remaining_path')
```

aufgelöst werden.

Statische Dateien

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Statische Dateien</title>

    <link href="{{ url_for('static', filename='css/style.css') }}">
    <script src="{{ url_for('static', filename='js/script.js') }}"></sc
</head>
<body>
<img src="{{ url_for('static', filename='imgs/image.png') }}">
</body>
</html>
```

Erben von Templates

Bei Webanwendungen gibt es Teile, die in jeder einzelnen Seite vorhanden sind.

Darunter falles zum Beispiel das Basismarkup (Dokumententyp, Head, Body) als auch statische Dateien wie CSS und eventuell Javascript.

Um dies nicht für jede einzelne neu schreiben zu müssen und es vor allem nicht immer auf jeder einzelnen Seite ändern zu müssen, können Template-Dateien voneinander erben.

Dabei werden bei einem Template Bereiche („Blöcke“) angegeben, die überschreiben werden können (zum Beispiel Titel, Hauptinhalt, …).

Ein anderes Template gibt dann an, von diesem Template zu erben und kann diese Bereiche überschreiben.

Beispiel: Erben von Templates, Basistemplate **base.html**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    {# Block für Titel, damit dieser geändert werden kann  #}
    <title>{% block title %}Standardtitel{% endblock %}</title>
    {# Binde CSS auf *allen* Seiten ein, die von dieser "erben" #}
    <link href="{{ url_for('static', filename='css/style.css') }}">
</head>
<body>
{# Block für den eigentlichen Inhalt der Seite #}
{% block content %}{% endblock %}
</body>
</html>
```

Beispiel: Erben von Templates, `customers.html`, `customers.py`

```python
@app.route('/customers')
def route_customers():
    customers: List[str] = ['Jack', 'James', 'Jason', 'John']
    return render_template('customers.html', customers=customers)
```

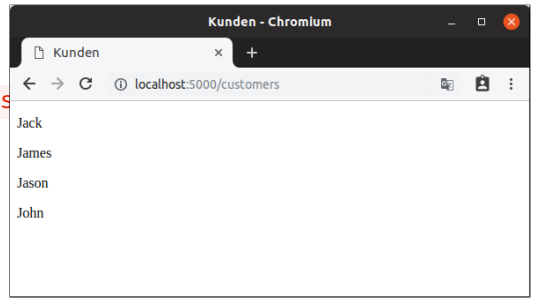
```jinja
{% extends "base.html" %}

{% block title %}Kunden{% endblock %}

{% block content %}
    {% for customer in customers %}
        <p>{{ customer }}</p>
    {% endfor %}
{% endblock %}
```

Beispiel: Erben von Templates, Resultat

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Kunden</title>
    <link href="/static/css/style.css
</head>
<body>
    <p>Jack</p>
    <p>James</p>
    <p>Jason</p>
    <p>John</p>
</body>
</html>
```



Aufgabe: Erben von Templates

Erstellen Sie ein Template `base.html`, das einen Titel- und einen Inhaltsblock definiert.

Erweitern Sie Ihre Nutzerübersicht, so dass beide Seiten von diesem Template erben.

Es soll auch auf beiden Seiten ein Link zur Nutzerübersicht vorhanden sein.

Lösung: Erben von Templates, `base.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
<p><a href="{{ url_for('route_users') }}">Nutzerübersicht</a></p>
{% block content %}{% endblock %}
</body>
</html>
```

Lösung: Erben von Templates, `users.html`

```
{% extends "base.html" %}
{% block title %}Nutzerübersicht{% endblock %}

{% block content %}
  {% if users %}
    <ul>
      {% for id, user in users.items() %}
        <li>
          <a href="{{ url_for('route_user_by_id', id=id) }}">{{ user.username }
        </li>
      {% endfor %}
    </ul>
  {% else %}
    <p>Es konnten keine Nutzer gefunden werden!</p>
  {% endif %}
{% endblock %}
```

Lösung: Erben von Templates, `user.html`

```
{% extends "base.html" %}

{% block title %}Nutzer {{ user.username }}{% endblock %}

{% block content %}
    <p>ID: {{ user.id }}</p>
    <p>Nutzername: {{ user.username }}</p>
    <p>Realer Name: {{ user.real_name }}</p>
    <p>Alter: {{ user.age }}</p>
{% endblock %}
```

Andere Antworttypen

Rückgabe von JSON

```python
from flask import Flask, jsonify

app = Flask(__name__)


@app.route('/api/customers')
def route_api_customers():
    customers = [
        {'first_name': 'Jason', 'family_name': 'Bourne'},
        {'first_name': 'James', 'family_name': 'Bond'},
        {'first_name': 'Jack', 'family_name': 'Bauer'}
    ]
    # Wandelt Argument in JSON um, teilt Browser Datentyp 'application/json' mi
    return jsonify(customers)
```

Rückgabe von XML

```python
from flask import Flask

app = Flask(__name__)


@app.route('/api/products')
def route_api_customers():
    xml_str: str = """\
<products>
    <product>Banane</product
    <product>Apple</product>
</customers>"""

    # Durch Setzen von 'Content-Type' erkennt Browser Datentyp
    return xml_str, {'Content-Type': 'application/xml'}
```