Inhalt

[1 Vorbereitungen 1](#_Toc114422133)

[1.1 Projektanlage 1](#_Toc114422134)

[1.2 Flask Installation 2](#_Toc114422135)

[2 Webseite für Kommentare 4](#_Toc114422136)

[2.1 Projektstruktur 4](#_Toc114422137)

[2.2 Erzeugen der App 4](#_Toc114422138)

[2.3 Hauptprogramm main.py 5](#_Toc114422139)

[2.4 Startseite – Das Template index.html 6](#_Toc114422140)

[2.5 Kommentarseite – comments.html 7](#_Toc114422141)

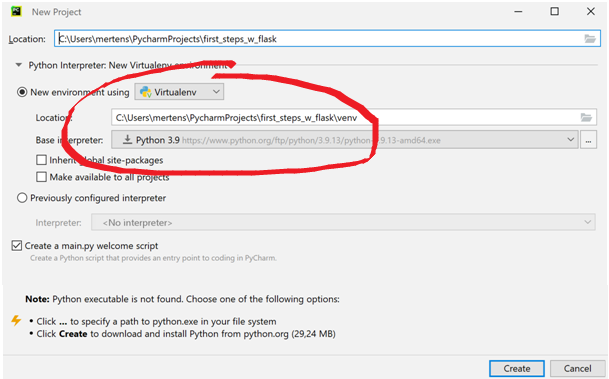
[2.6 Anzeigen der Kommentare - Methode GET 8](#_Toc114422142)

[2.7 Verfassen eines neuen Kommentars – Methode POST 9](#_Toc114422143)

# Vorbereitungen

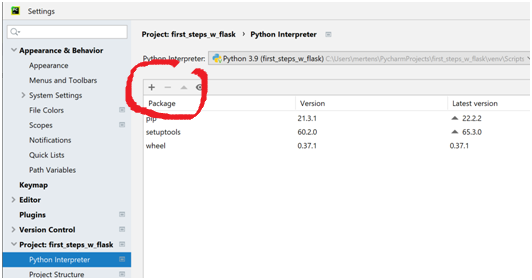
## Projektanlage

Wir legen zunächst ein neues Projekt in PyCharm an. Bitte darauf achten, dass das Projekt in einem virtuellen Environment angelegt wird.

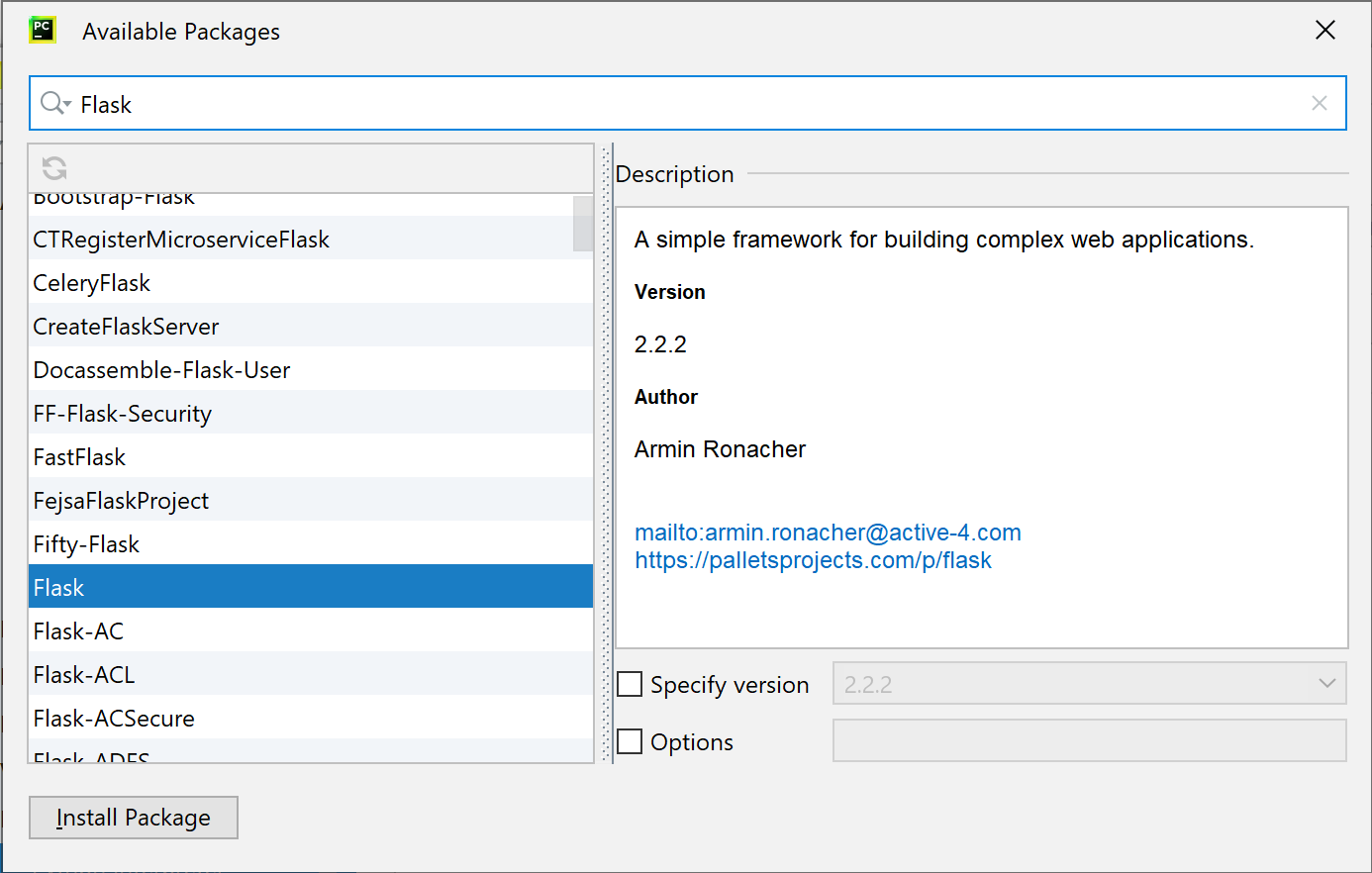


## Flask Installation

Für das Tutorial wird das Package Flask benötigt. Dies wird nun im virtuellen Environment installiert. Dazu das Menü unter File-Settings-Project-Python Interpreter aufrufen.



Mit dem „Plus“ die verfügbaren Packages anzeigen lassen, nach Flask suchen und „Install Package“ anklicken.



Funktionstest

Wir legen folgende main.py an:

from flask import Flask

app = Flask(\_\_name\_\_)

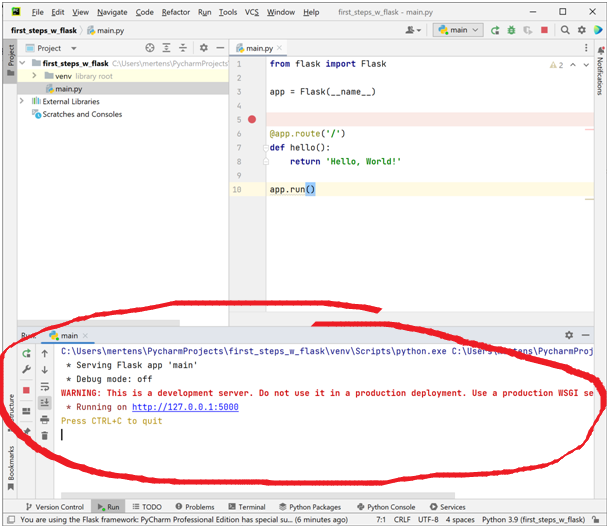
@app.route('/')

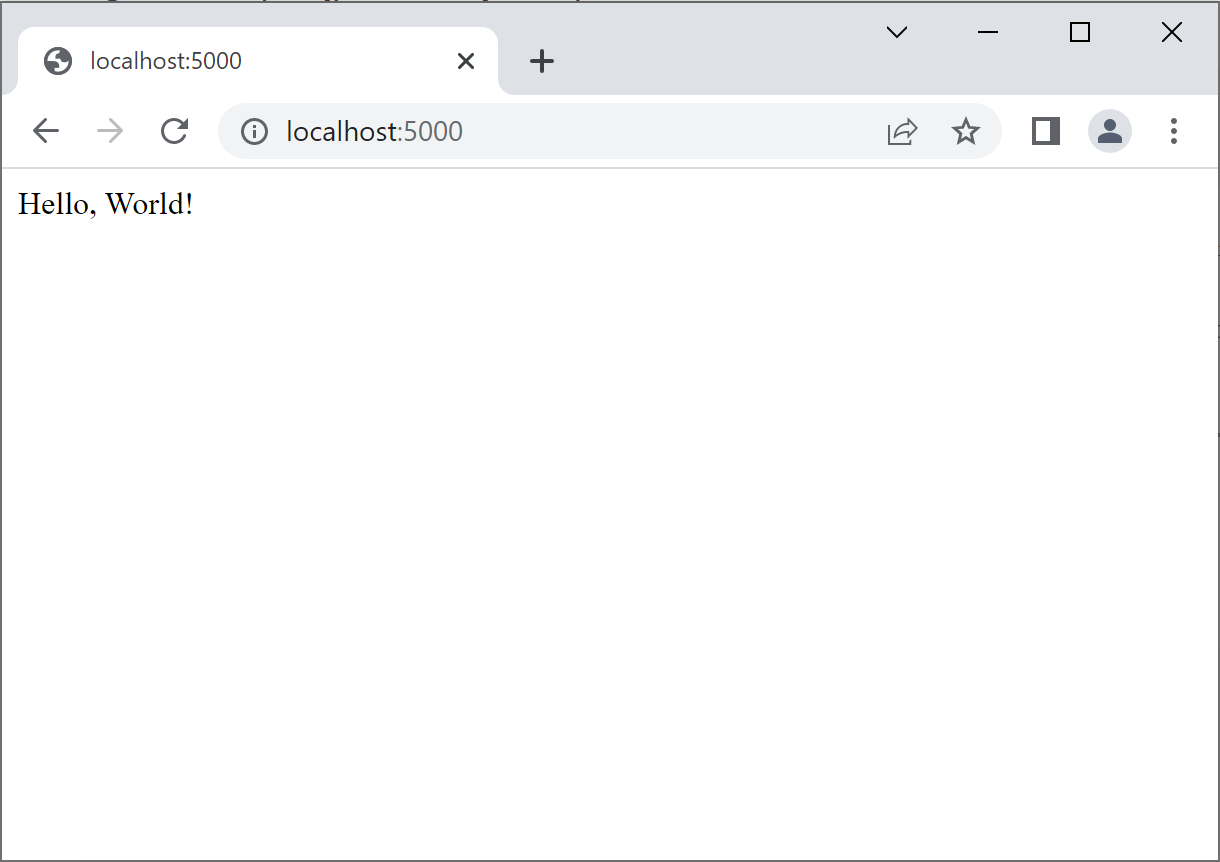
def hello():

return 'Hello, World!'

app.run()

Wir starten das Programm (eventuell vorher eine Run-Konfiguration anlegen) und erhalten untenstehende Nachricht.



Mit dem Starten der app mit app.run() wurde ein Webserver gestartet. Der Webserver wartet auf Port 5000 auf Anfragen.

Mit dem Decorator @app.route(“/“) wird definiert, dass bei Aufruf von „/“die Funktion hello() aufgerufen wird. Hello() liefert den String „Hello world“ zurück und der Webserver liefert das entsprechend aus.

Wenn der Browser den Text ausgibt, ist die Installation korrekt.

# Webseite für Kommentare

Die Ausgabe eines Strings wäre sicherlich einfacher als mit Flask zu realisieren gewesen. Daher bauen wir das Programm zu einer Webseite aus, auf der man Kommentare hinterlassen kann. Dabei werden folgende Inhalte gezeigt:

* Modulares Programmieren
* Wiedergabe von Html-Seiten
* Erzeugen dynamischer Inhalte
* Verlinkung und Redirects
* Behandlung von Get und Post Requests

Die Webseite soll aus einer Startseite bestehen, über die man eine Kommentarseite findet. Dort werden automatisch alle Kommentare aufgelistet und man hat die Möglichkeit einen eigenen Kommentar anzufügen.

## Projektstruktur

Unser Projekt erhält folgende Struktur:

|--templates/

| |--base.html

| |--comments.html

| |--index.html

|--static/

| |--css/

| |--comments.css

|--main.py

|--comments.py

|--\_\_init\_\_.py

Flask erwartet HTML Seiten unter templates. Es handelt sich ja auch tatsächlich um Vorlagen, die von der Anwendung gefüllt werden, und nicht um fertige Webseiten.

Bilder stehen im Verzeichnis static und im Unterverzeichnis dazu css-Dateien.

Die Python Programmdateien lassen wir alle im Projekthauptverzeichnis stehen.

## Erzeugen der App

Wir lagern das Erzeugen der app in die Datei \_\_init\_\_.py aus. (Stichwort: Factory Design Pattern)

Unser Hauptprogramm ruft die Funktion create\_app() auf und erhält ein Objekt mit der Applikation.

*Achtung: Vorsicht bei copy&paste aus diesem Dokument. Word nutzt zum Teil andere Anführungszeichen. Das kann dazu führen, dass das Programm richtig aussieht aber dennoch nicht funktioniert.*

|  |  |
| --- | --- |
| **\_\_init\_\_.py Programmcode** | **Erläuterung** |
| from flask import Flask  def create\_app():  app = Flask(\_\_name\_\_)   app.config['SECRET\_KEY'] = 'xcde234'   from main import main as main\_blueprint  app.register\_blueprint(main\_blueprint)   #from comments import comments as comments\_blueprint  #app.register\_blueprint(comments\_blueprint)   return app | Flask wird importiert, wie im Setup-Beispiel.  Flask wird erzeugt und app zugewiesen.  Anschließend wird die Applikation noch konfiguriert.  Ein Blueprint ist eine Art Vorlage mit der die Anwendung in Module aufgeteilt werden kann. Diese Module müssen bei der app registriert werden, damit sie benutzt werden können.  Wir teilen das Programm in 2 Module ein:   * main für index.html * comments für comments.html   comments bleibt zunächst auskommentiert, damit wir schrittweise testen können. |

## Hauptprogramm main.py

|  |  |
| --- | --- |
| **main.py Programmcode** | **Erläuterung** |
| from flask import Blueprint, render\_template from \_\_init\_\_ import create\_app  main = Blueprint('main', \_\_name\_\_)  @main.route('/') def index():  return render\_template('index.html')  if \_\_name\_\_ == '\_\_main\_\_':  create\_app().run() | Imports  Hier wird der Blueprint main angelegt, den wir in \_\_init\_\_.py registrieren.  Decorator, der die Funktion index() aufruft, wenn / im Browser aufgerufen wird.  Die Anweisung create\_app().run wird nur aufgerufen, wenn main.py als Programm gestartet wurde. Bei einer Integration in einen Webserver erfolgt der Start anders. Auf diese Weise kann das Programm unverändert zwischen Entwicklungs- und Produktivsystem bleiben. |

Im Gegensatz zum ersten Beispiel wird hier kein einfacher String von index() zurückgegeben sondern ein komplettes html-Dokument. Die Funktion render\_template bereitet das das Template index.html vor und übergibt eine String-Repräsentation der Seite. Das Rendern erfolgt mot dem Package jinja, das automatisch mit Flask installiert wurde.

## Startseite – Das Template index.html

Die Seite index.html soll im Header ein Logo zeigen und mit einem einfachen Link auf die Seite comments.html verweisen.

Da der Header üblicherweise auf allen Html Seiten gleich ist, teilen wir die Datei index.html in base.html und index.html auf.

base.html enthält die Elemente, die auf allen Seiten gleich sind. Also in unserem Fall den Header mit Logo und die allgemeinen Html-Definitionen wie z.B. den Verweis auf ein Style-Sheet.

Beim Rendern wird nach den Kennzeichnung „{{ }}“ oder „{% %}“. An diesen Stellen wird Inhalt dynamisch generiert.

|  |  |
| --- | --- |
| **base.html** | **Erläuterungen** |
| <!DOCTYPE html> <html>  <head>  <meta charset="utf-8">  <meta http-equiv="X-UA-Compatible" content="IE=edge">  <meta name="viewport" content="width=device-width, initial-scale=1">  <title>HSS Flask Tutorial</title>  <link rel="stylesheet" href="static/css/commments.css" > </head>  <body>  <header class="header">  <img src="static/HSS Logo.png" class="logo" height='50px' >  </header>   <div class="hss-body">  <div class="container has-text-centered">  **{% block content %}  {% endblock %}**  </div>  </div> </body>  </html> | Standard Html Definitionen  An diese Stelle wird der Inhalt von index.html eingefügt. |

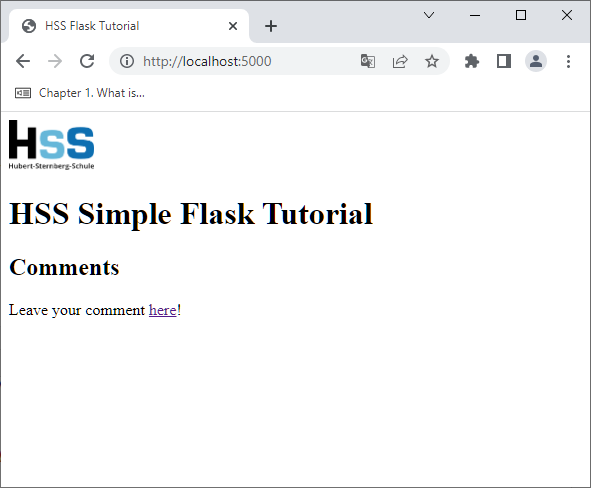
Im Hauptprogramm wird der Funktion render\_template() der Name des Templates index.html übergeben. Daher muss index.html als erstes auf base.html verweisen.

|  |  |
| --- | --- |
| **index.html** | **Erläuterungen** |
| **{% extends "base.html" %}**  **{% block content %}** <h1 class="title">  HSS Simple Flask Tutorial </h1> <h2 class="subtitle">  Comments </h2>  Leave your comment <a href="{{ url\_for('comments.comment') }}">here</a>!  **{% endblock %}** | Hier ist der Verweis auf base.html.  Hier wird der Beginn des Content-Blocks gekennzeichnet.  Der Inhalt zwischen den Content Markierungen wird in base.html eingefügt.  Der Ausdruck  {{ url\_for('comments.comment') }}  Ermittelt die URL, die man benutzen muss, um die Funktion comment() im Modul comments.py aufzurufen.  Hier wird das Ende des Content-Blocks gekennzeichnet. |

Nach dem Start (Versuch) erhalten wir die Fehlermeldung, dass comments.comment nicht gefunden wurde. Das ist richtig, da wir das Modul comments.py noch nicht geschrieben haben.

Zum Test ersetzen wir ‚comments.comment‘ durch ‚main.index‘

Nach dem Start ergibt sich folgendes Bild. Der Link here verweist auf „localhost:5000/“



Für die nächsten Schritte machen wir die Ersetzung von ‚comments.comment‘ wieder rückgängig.

## Kommentarseite – comments.html

Auf der Kommentarseite erstellen wir im oberen Bereich ein Formular, in dem der Benutzer seinen Text eingeben und submitten kann.

Direkt unterhalb zeigen wir alle Kommentar an.

Der Anfang ist einfach

|  |  |
| --- | --- |
| **comments.html Teil 1** | **Erläuterungen** |
|  |  |
| **{% extends "base.html" %}**  **{% block content %}**  <h1>Your comment</h1> <form method="post" action="/comments">  <input name="txt" type="text">  <button id="save" type="Submit">Send</button> </form> | Wieder der Verweis auf base.html.  Hier wird der Beginn des Content-Blocks gekennzeichnet.  Einfaches Textfeld mit dem Button Send. Beim Submit wird die Post-Methode verwendet.  Die Seite, die aufgerufen wird ist /comments. Bei einem größeren Projekt sollte man den Ausdruck {{url\_for(…)}} verwenden |

Da wir die Kommentare noch nicht kennen, muss der HTML-Code dynamisch erzeugt werden. Dazu kann man der Methode render\_template() Variablen mit Daten übergeben.

Nehmen wir an, wir bekommen in einer Variable comments alle Kommentare übergeben. Dann können wir in einer Schleife den entsprechende HTML Code erzeugen lassen

|  |  |
| --- | --- |
| **comments.html Teil 2** | **Erläuterungen** |
| <h1>Previous comments</h1>  {% for comment in comments %}  {{comment}} <br> {% endfor %}  {% endblock %} | Iteration über alle Kommentare  Ausgabe Kommentar  Ende-Markierung des Blocks für base.html |

## Anzeigen der Kommentare - Methode GET

|  |  |
| --- | --- |
| **comments.pyl Teil 1** | **Erläuterungen** |
| from flask import Blueprint, render\_template, current\_app, redirect, url\_for,request  comments = Blueprint('comments', \_\_name\_\_)  @comments.route('/comments') def comment():   comments=""  try:  f = open('comments.txt', 'r')  f.seek(0)  comments = f.readlines()  f.close()  except:  f = open('comments.txt', 'x')  f.close()  return render\_template('comments.html', comments=comments) | Imports  Funktion, die mit GET bei /comments aufgerufen wird.  Datei (mit den Kommentaren) öffnen und Kommentare lesen.  Datei anlegen, falls noch nicht existent  Funktion render\_template() aufrufen und alle Kommentare übergeben (für die for-Schleife in comments.html) |

## Verfassen eines neuen Kommentars – Methode POST

Wurde ein Kommentar eingegeben wird wieder die Seite /comments aufgerufen, aber mit der Methode POST und den Formulardaten. Dafür erstellen wir eine eigene Funktion comments\_post()

|  |  |
| --- | --- |
| **comments.pyl Teil 2** | **Erläuterungen** |
| @comments.route('/comments', methods=['POST']) def comment\_post():  comment = "\n" + request.form.get('txt')  f = open('comments.txt', 'a')  f.writelines(comment)  f.close()  return redirect(url\_for('comments.comment')) | Die Formulardaten werden in der Variable request bereitgestellt.  Im Html wurde die Textbos mit dem Namen txt angelegt (<input name="txt"…)  Abfrage des Kommentars mit form.get(key)  Datei öffnen (Modus append)  Schreiben und Datei schließen  Jetzt machen ein redirect auf /comments (das ist dann wieder die GET Methode), damit der neue Kommentar mit angezeigt wird. |

Bevor wir testen können, müssen wir noch den Blueprint für comments aktivieren – Kommentare in \_\_init\_\_.py entfernen.

Das Ergebnis sieht dann so aus:

