Do-it-yourself: Statischer HTTP-Server

Wir verwenden wieder einmal die Programmiersprache Python. Die *Socket-Programmierschnittstelle* haben wir ja schon hier kennengelernt. Deswegen steigen wir direkt in den Code ein:

```
import socket


# Allgemeine Defiitionen:
SERVER_HOST = '0.0.0.0' # d.h. alle Netzwerkschnittstellen des Rechners
SERVER_PORT = 8000 # nur im Beispiel,
                   # per Konvention geht http über Port 80

# Wir binden uns an einen Socket
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:

    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server_socket.bind((SERVER_HOST, SERVER_PORT))
    server_socket.listen(1)
    print('Lausche auf Port %s ...' % SERVER_PORT)

    # Abarbeiten aller Client-Anfragen in einer Endlosschleife:
    while True:
        # hier warten wir auf den ersten Client
        client_connection, client_address = server_socket.accept()

        # da hat einer angebissen, jetzt die Daten des Requests:
        request = client_connection.recv(1024).decode("utf-8")
        print(request)

        # Und unser HTTP Response dazu:
        response = 'HTTP/1.0 200 OK\n\nHallo Welt\n\n'
        #
        #          \n ist das Sonderzeichen für Zeilenende
        #
        client_connection.sendall(response.encode("utf-8"))
        client_connection.close()

# Wenn die Endlosschleife am Ende ist...
server_socket.close()
```

Das ist der allereinfachste statische Webserver. Es wird immer (egal welche Ressource angefordert wurde) Das (Pseudo-)Dokument „**Hallo Welt**“ zurückgeliefert.

Den Kommunikationsablauf schauen wir uns trotzdem mal in der Konsole (wieder mit Rechtsklick+*Neues Fenster*) an:

Server starten mit

```
cd Schulung
python3 hello_http.py
```

und in einer zweiten Konsole:

```
telnet 127.0.0.1 8000
...
GET /irgendwas HTTP/1.0

...
```

Und dabei nach dem GET nicht die leere Zeile vergessen!

So, das war nur zu Demonstrationszwecken. Dieser Code prüft nicht einmal auf korrekte Syntax des Requests (warum auch ).

Ein korrekter Web-Server würde die Request-Zeile („GET /…“) auf den geforderten Aufbau hin überprüfen und interpretieren.

Die eigentliche Aufgabe besteht aber darin, je nach Request-Kommando und der angefragten Ressource den Inhalt nach der Leerzeile des Response zusammenzutragen.

Wir nehmen hier immer die einfachste Form des Request, das GET. Der Vollständigkeit halber ein paar Worte zu den anderen Request-Typen:

- **HEAD**: Spezialform des GET, es soll kein Inhalt übertragen werden. Der Browser will hier nur nachschauen, ob sich zwischenzeitlich an der Ressource etwas geändert hat oder ob ein erneutes GET gespart werden kann.
- **POST**: Das ist die Form eines Requests, wenn der Browser dem Server auch noch Daten schicken will. Die können aus Formulareingabefeldern aus der Webseite stammen oder beispielsweise ein Datei zum Hochladen sein. Offensichtlich hat der Server dann etwas mehr zu tun. Die Daten werden auch hier nach der Leerzeile hinter der Requestzeile angehängt. Details dazu spare ich mir hier.
- In späteren HTTP-Versionen kommen weitere Request-Typen dazu: PUT, DELETE, … Die spielen aber beim Web-Surfen auch keine bedeutende Rolle.

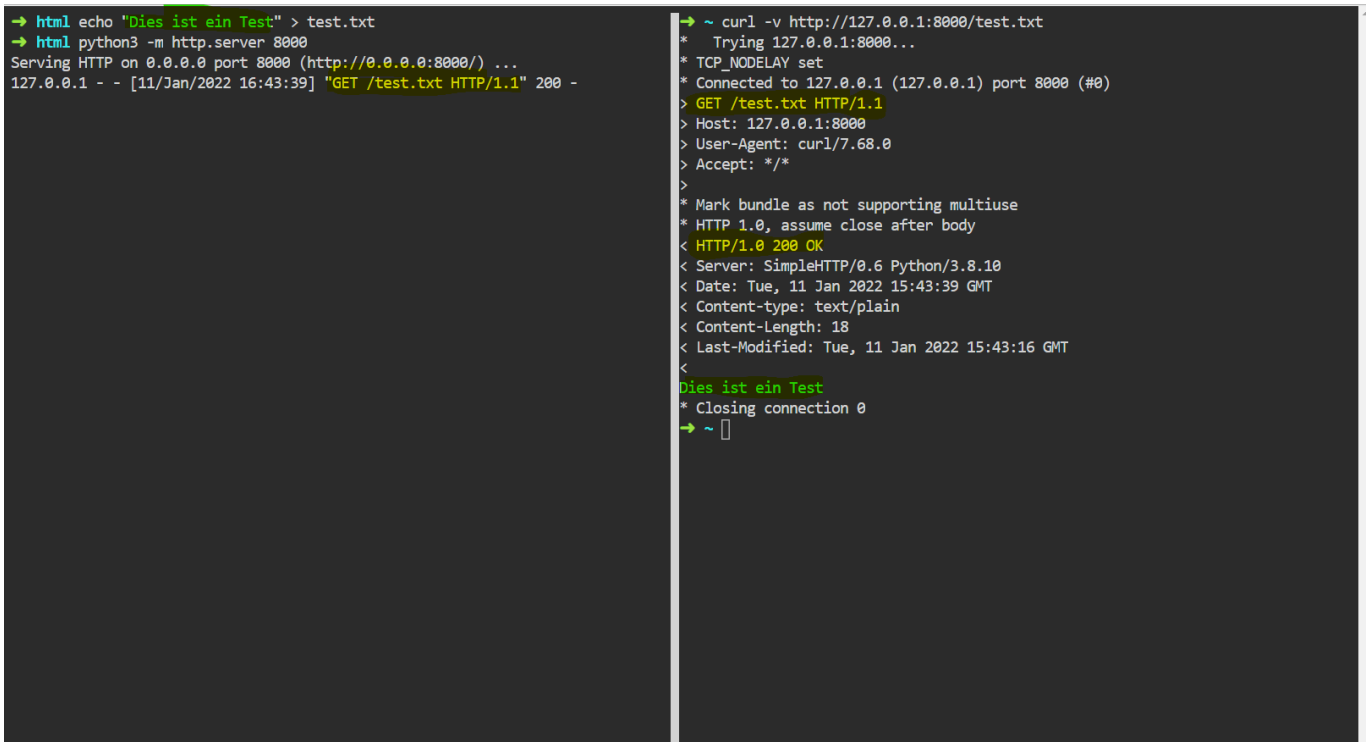
Zu Übungszwecken ersetzen wir den etwas zu einfachen HTTP Server von eben mit dem nächst einfachen Modell, das jede Ressource in der URL als Dateinamen im aktuellen Verzeichnis interpretiert:

```
cd ~/Schulung/html
echo "Dies ist ein Test." > test.txt
python3 -m http.server 8000
```

und auf der *Client* Seite verwenden wir jetzt auch mal ein etwas leistungsstärkeres Erprobungstool: **curl** verhält sich wie ein richtiger Browser, gibt aber alle Request-Inhalte direkt als Text im Terminal aus. Beachte das **-v**! Damit wird die eigentliche http-Kommunikation mit rausgeschrieben, einschließlich aller Header-Zeilen:

```
curl -v http://localhost:8000/test.txt
```

Das Ergebnis sieht dann in Summe etwa so aus:



Aber schon wenn man ein echtes (wenn auch recht einfaches) HTML Dokument auf diesem Weg anzeigen will, dann sieht man nicht viel. Beispiel für so eine Datei:

```
<!doctype html>

<html lang=de>

<head>
  <meta charset=utf-8>
  <title>Schulungsseite</title>
</head>

<body>
  <h1>Willkommen</h1>
  <p>Diese Seite dient nur zu Schulungszwecken.</p>

  <img src="nachbarn.png"></img>


  <li> <a href="gollum.jpg">Dies ist ein Link auf ein weiteres Bild...</a>
</li>
  <li> <a href="existiert.nicht">und dieser Link provoziert eine 404
Fehlermeldung.</a></li>

</body>

</html>
```

Deshalb schauen wir uns das Ergebnis lieber in einem richtigen Browser an.

Im wirklichen Leben macht sich außerdem ein richtiger Web-Server etwas mehr Arbeit bei seinem Response...