JUST DO IT
HÖHERE ABTEILUNG FÜR INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

A07: Python Webframeworks
CherryPy - CRUD

SEW
5BHIT 2017/18

Marvin Ertl & Lukas Zuba

Version 1.0
Note:
Begonnen am 22. Dezember 2017
Betreuer:
Beendet am 11. Januar 2018

Inhaltsverzeichnis

1 Aufgabenstellung

Suche dir mit einem/einer Partner/in ein Python Webframework aus und präsentiere deine Lösung!

Grundanforderungen (70%):

- Installiere und konfiguriere eines der präsentierten Frameworks
- Überlege dir einen sinnvollen Anwendungsfall für das Framework und erstelle eine passende Datenbank dazu
- Erstelle eine simple Seite, welche Datensätze aus einer Datenbank anzeigt
- Protokolliere deine Vorgehensweise, aufgetretene Probleme etc.

Erweiterungen (30%):

- Verwende ein ansprechendes Design
- Die Seite erlaubt zusätzlich:
  - Das Bearbeiten von Datensätzen
  - Das Löschen von Datensätzen
  - Das Erstellen von neuen Datensätzen

Abgabe: Protokoll inkl. Arbeitsaufwand, Screenshots und Beschreibungen

2 Ergebnis

Folgende Punkte wurden bearbeitet:

- Allgemein
- CherryPy und andere angewendeten Technologien
- Umsetzung
- Aufgetretene Probleme
- Ergebnis

2.1 Allgemein

Wir haben uns für CherryPy entschieden und für einen Anwendungsfall, der schnell zu implementieren und eine niedrige Komplexität aufweist. Das ist laut CherryPy einer der häufigsten Anwendungsfälle für CherryPy. Ein weiterer wichtiger Vorteil ist, dass CherryPy auf OOP ausgelegt ist und lässt sich dadurch zusätzlich leichter schreiben und verstehen.

Wir haben eine Webseite mit CRUD-Funktionen erstellt mit denen man Benutzer verwalten kann. Dazu haben wir in der Datenbank eine Tabelle angelegt die wie folgt ausschaut:

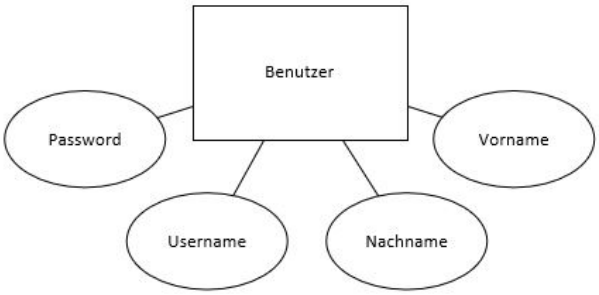

Abbildung 1: Aufbau von Benutzer

Das Einsteigen in CherryPy ist mit Abstand einer der größten Vorteil, den mit wenigen Zeilen lässt sich eine Webseite erstellen und bereitstellen.

```
1 import cherrypy

3 class HelloWorld(object):
      @cherrypy.expose
5     def index(self):
          return "Hello World!"
7
  cherrypy.quickstart(HelloWorld())
```

Mit diesen Zeilen ist eine Webseite erstellt und bereit erweitert zu werden.

Insgesamt beläuft sich der Arbeitsaufwand auf ungefähr 0 Stunden.

2.2 CherryPy und andere angewendeten Technologien

CherryPy ist ein Non-Full-Stack-Framework und bietet mit einer sehr einfachen Verfahren einen Webserver an, der mit wenigen Befehlen gestartet und mit Inhalt gefüllt werden kann.

Weiters haben wir für unsere Persistierung die eingebaute Schnittstelle von Python zu SQLite verwendet. Dieses Schnittstelle ermöglicht es mit wenigen Befehlen eine simple Datenbank zu betreiben. Allerdings hatten wir auch einigen Probleme mit der Datenbank auf die wir später noch genauer eingehen.

Um die Webseite mit der CherryPy-Rückgaben zu verbinden, haben wir jQuery verwendet.

Zusätzlich haben wir, um der Webseite ein sprechenden Design zu geben, CSS und Bootstrap verwendet.

2.3 Umsetzung

Die Übung ist in 2 Bereiche aufzuteilen, den Server Teil, dieser kümmert sich um das Speichern und Lesen aus der Datenbank, diese Aufgabe wurde mittels den Python WebFramework CherryPy umgesetzt. Der Client Teil kümmert sich hingegen um die Darstellung sowie um die Eingaben, dies wurde mittels HTML, CSS und JavaScript umgesetzt.

2.3.1 CherryPy

Als erstes wurde eine Klasse CRUD erstellt, diese wird aufgerufen sobald man auf localhost mit der entsprechenden Port Nummer zugreift. Die Klasse öffnet die index.html Datei und sendet den Inhalt an den Browser zurück.

```
class CRUD(object):
    @cherrypy.expose
    def index(self):
        return open('index.html')
```

Listing 1: Klasse zur Darstellung der Index.html

Nun wurde eine Klasse erstellt, welche auf die HTML Befehle reagiert, dies sind GET, POST, PUT, ... In unserem Fall haben wir nur POST verwendet mit 2 Parameter. Der erste Parameter param gibt an welche Aktion wir durchführen wollen, der zweite Parameter input gibt liefert zusätzliche Informationen, wie zum Beispiel eingeben Werte.

```
@cherrypy.expose
class CRUDWebService(object):

    @cherrypy.tools.accept(media='text/plain')
    def POST(self, param, input):
```

Listing 2: Klasse zur verwaltung der HTML Befehle

Wird ein read als Parameter gesendet, erwartet sich der Client alle Benutzer. Aus diesem Grund werden alle Benutzer aus der Datenbank ausgelesen und eine HTML Tabelle erstellt. Diese wird dann als HTML Response zum Client zurückgesendet und dargestellt.

```
if param == "read":
    with sqlite3.connect(DB_STRING) as c:
```

```
        r = c.execute("SELECT * FROM benutzer")
        response = "<table width='100%' class='table table-striped table-bordered'" \
        "cellspacing='0'><tr><td>Nr</td><td>Vorname</td><td>Nachname</td>" \
        "<td>Username</td></tr>"
        while True:
            row = r.fetchone()
            if row is None:
                break
            response += "<tr><td>" + str(row[0]) + "</td><td>" + row[1] + "</td><td>" + row[2] +
                "</td><td>" + row[3] + "</td></tr>"
        response += "</table>"
    return response
```

Listing 3: Auslesen aller Benutzer aus der Datenbank

Auf den Parameter update reagiert das Programm sehr ähnlich, einziger Unterschied ist, dass die Benutzer in einer Tabelle zurückgeliefert werden, die bei jedem Benutzer einen Button zum Bearbeiten hat. Außerdem müsste hier ein kurzes sleep eingebaut werden, da update gleich nach einer Datenbank ändern die neuen Benutzer ausliest, dadurch konnte es passieren das beim Auslesen noch nicht die neuen Werte dabei waren.

```
if param == "update":
    sleep(0.05)
    with sqlite3.connect(DB_STRING) as c:
        r = c.execute("SELECT * FROM benutzer")
        response = "<table width='100%' class='table table-striped table-bordered'" \
        "cellspacing='0'><tr><td>Nr</td><td>Vorname</td><td>Nachname</td>" \
        "<td>Username</td><td>Aendern</td></tr>"
        while True:
            row = r.fetchone()
            if row is None:
                break
            response += "<tr><td>" + str(row[0]) + "</td><td>" + row[1] + "</td><td>" + row[2] + "</td><
                td>" + row[3] + "</td><td> <button onClick='updateBenutzer(" + str(row[0]) + ")'>Aendern
                </button></td></tr>"
        response += '</table>'
    return response
```

Listing 4: Auslesen aller Benutzer aus der Datenbank und bearbeitbar zurückliefern

Wenn der Benutzer eine Person bearbeiten will, müssen die aktuellen Informationen von dieser Person abgerufen werden, hierfür gibt es den Parameter getBenutzer. Dieser hat zusätzlich als input die Nummer, welche der Primary Key der Entität ist. Die zurückgelieferten Informationen werden, dann in Input Felder angezeigt um diese zu bearbeiten.

```
if param == "getBenutzer":
    with sqlite3.connect(DB_STRING) as c:
        r = c.execute("SELECT * FROM benutzer WHERE nr=" + input)
        while True:
            row = r.fetchone()
            if row is None:
                break
            response = str(row[0]) + "#" + row[1] + "#" + row[2] + "#" + row[3] + "#" + row[4];
    return response
```

Listing 5: Auslesen eines Benutzer aus der Datenbank

Wenn der Benutzer die Informationen in den Input Feldern geändert hat und auf Speichern klickt, werden die neuen Informationen über den Parameter updateBenutzer und die Informationen als input an den Server gesendet. Anschließend wird der Benutzer in der Datenbank aktualisiert.

```
if param == "updateBenutzer":
    with sqlite3.connect(DB_STRING) as c:
        liste = input.split('#')
```

```
        c.execute("UPDATE benutzer SET vorname='" + liste[1] + "',nachname='"
            + liste[2] + "',username='" + liste[3] + "',password='" + liste[4] + "' WHERE nr=" + liste
                [0])
    return "Benutzer erfolgreich geaendert."
```

Listing 6: Updaten der Benutzerinformationen

Wird ein neuer Benutzer erstellt, wird über den Parameter create und den Benutzerinformationen als input an den Server gesendet. Dieser speichert die Informationen in der Datenbank.

```
if param == "create":
    with sqlite3.connect(DB_STRING) as c:
        liste = input.split('#')
        c.execute("INSERT INTO benutzer(vorname, nachname, username, password) VALUES (" + liste[0] + ",
            "
        + liste[1] + ", " + liste[2] + ", " + liste[3] + ")")
    return "Benutzer erfolgreich gespeichert."
```

Listing 7: Updaten der Benutzerinformationen

Der delete Branch ist sehr ähnlich zum update Branch, der einzige Unterschied ist, dass anstele eines Update Button wird ein Löschen Button erstellt.

```
if param == "delete":
    with sqlite3.connect(DB_STRING) as c:
        r = c.execute("SELECT * FROM benutzer")
        response = "<table width='100%' class='table table-striped table-bordered'" \
                   "cellspacing='0'><tr><td>Nr</td><td>Vorname</td><td>Nachname</td>" \
                   "<td>Username</td><td>Loeschen</td></tr>"
        while True:
            row = r.fetchone()
            if row is None:
                break
            response += "<tr><td>" + str(row[0]) + "</td><td>" + row[1] + "</td><td>" + row[2] + "</td>
                <td>" \
            + row[3] + "</td><td> <button onClick='deleteBenutzer(" + str(row[0]) + ")'>Loeschen" \
            "</button></td></tr>"
        response += '</table>'
    return response
```

Listing 8: Auslesen aller Benutzer mit einen Löschen Button

Abschließend können Benutzer aus der Datenbank gelöscht werden, hierfür wird als Parameter deleteBenutzer angegeben und als input die Nummer des Benutzers. Falls ein ungültiger Parameter gesendet wird, wird ein error zurückgeliefert.

```
if param == "deleteBenutzer":
    with sqlite3.connect(DB_STRING) as c:
        c.execute("DELETE FROM benutzer WHERE nr=" + input)
    return self.POST("delete", "")

return "Ungueltige Anfrage"
```

Listing 9: Löschen eines bestimmten Benutzers

Weiters gibt es die Methode setup_database diese wird immer beim Starten des Servers ausgeführt und soll sicherstellen, dass die Datenbank sowie Tabelle vorhanden ist. Außerdem werden in diesem Fall noch 2 Benutzer als Testdaten standardmäßig importiert.

```
def setup_database():
    with sqlite3.connect(DB_STRING) as con:
        con.execute("DROP TABLE IF EXISTS benutzer")
        con.execute("CREATE TABLE IF NOT EXISTS benutzer(nr INTEGER PRIMARY KEY AUTOINCREMENT, vorname
            VARCHAR, "
        "nachname VARCHAR, username VARCHAR, password VARCHAR)")
```

```
6           con.execute("INSERT INTO benutzer VALUES(null,'Marvin', 'Ertl', 'mertl', 'password')")
            con.execute("INSERT INTO benutzer VALUES(null,'Lukas', 'Zuba', 'lzuba', 'password')")
```

Listing 10: Erstellen der Tabelle Benutzer beim Start des Servers

Ebenso gibt es eine Methode namens cleanup_database, diese löscht die Tabelle beim Beenden des Server.

```
1 def cleanup_database():
      with sqlite3.connect(DB_STRING) as con:
3         con.execute("DROP TABLE IF EXISTS benutzer")
```

Listing 11: Löschen der Tabelle benutzer beim Beenden des Servers

Abschließend wird in einer IF-Anweisung die Konfiguration hineingeschrieben um den Server zu starten. Ebenfalls wird hier auch angegeben welche Methoden beim Starten sowie beim Beenden des Servers ausgeführt werden sollen. Außerdem wird die auto reload ausgeschaltet, dies führt ansonsten dazu, dass wenn das Python Programm abstürzt der Server weiterhin im Hintergrund den Port belegt. Schlussendlich wird angeben welche Klasse beim Aufrufen reagiert, sowie welche Klasse auf die HTML Befehle reagiert.

```
1 if __name__ == '__main__':
      conf = {
3         '/': {
              'tools.sessions.on': True,
5             'tools.staticdir.root': os.path.abspath(os.getcwd())
          },
7         '/generator': {
              'request.dispatch': cherrypy.dispatch.MethodDispatcher(),
9             'tools.response_headers.on': True,
              'tools.response_headers.headers': [('Content-Type', 'text/plain')],
11        },
          '/static': {
13            'tools.staticdir.on': True,
              'tools.staticdir.dir': './public'
15        }
      }
17    cherrypy.engine.subscribe('start', setup_database)
      cherrypy.engine.subscribe('stop', cleanup_database)
19    cherrypy.config.update({'server.socket_port': 8080})
      cherrypy.config.update({'engine.autoreload_on': False})
21
      webapp = CRUD()
23    webapp.generator = CRUDWebService()
      cherrypy.quickstart(webapp, '/', conf)
```

Listing 12: Konfiguration des Servers

2.4 Tabelle

2.5 Aufzählung

- **Lorem ipsum:** dolor sit amet, consetetur sadipscing elitr
- sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat
- ut labore et dolore magna aliquyam erat, sed diam voluptua

Header	Kopf
Lorem	Ipsum dolor sit amet, consetetur sadipscing elitr
Ipsum	At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus
Dolor	Consetetur sadipscing elitr, sed diam nonumy

Tabelle 1: Lorem ipsum dolor sit amet [1]

2.6 Aufgetretene Probleme

2.6.1 Tabelle nicht gefunden

Die häufigste Fehlermeldung dieser Art war: *no such table : benutzer*

Diesen Fehler konnten wir auf keine genaue Stelle eingrenzen, da dieser Fehler zufällig beim Starten des Server hin und wieder aufgetretten ist.

Gelöst haben wir diesen Fehler dann durch das Sicherstellen, dass die Tabelle noch existiert und wenn nicht die Tabelle neu erstellt und mit Standarddatensätzen wieder befüllt wird.

2.6.2 Datensatz nicht richtig abgespeichert

Ein Fehler, der leicht zu lösen war, war: *no such attribute : zuba*

Hierbei hat es sich um eine INSERT Anweisung gehandelt, die eine syntaktisch falschen Aufbau hatte, den wir allerdings erst sehr spät bemerkt haben.

Gelöst haben wird dies, indem wir die Zusammensetzung der Anweisung neu geschrieben haben und explizit die Reihenfolge der Attribute angegeben haben.

2.6.3 JS Funktionen nicht aufrufbar

function showuser not found

2.6.4 Aufrufprobleme

Delete konnte Daten nicht anzeigen, wenn davor Anzeigen aufgerufen wurde

2.7 Ergebnis

Literatur

[1] A.S. Tanenbaum and M. Van Steen. *Verteilte Systeme: Prinzipien und Paradigmen.* Pearson Studium. Addison Wesley Verlag, 2007.

Tabellenverzeichnis

Listings

Abbildungsverzeichnis