

Webapplikationen mit Flask

Jochen Reinholdt

Version 03.05.2022

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie;
detaillierte bibliografische Daten sind im Internet über <http://dnb.dnb.de> abrufbar.

© 2022 Jochen Reinholdt (i.d.R. Sie bzw. Ihr Pseudonym)

Lektorat: Vorname Name oder Institution

Korrektorat: Vorname Name oder Institution

weitere Mitwirkende: Vorname Name oder Institution

Herstellung und Verlag: BoD – Books on Demand, Norderstedt

ISBN: 978-3-XXXX-XXXX-X

1	Basistechnologien.....	9
1.1	Lernziele.....	9
1.2	Protokolle und Technologien	9
1.3	http.....	10
1.4	HTML und CSS	14
1.5	Lernziele.....	14
1.6	Aufgaben zu HTML	18
1.7	CSS	19
1.8	Zusammenfassung	20
2	Installation	21
2.1	Lernziele.....	21
2.2	Architektur.....	21
2.3	Virtuelles Environment.....	22
2.4	Tutorial 1.....	22
2.5	Hintergrund	28
2.6	Zusammenfassung	31
3	Templates.....	32
3.1	Lernziele.....	32
3.2	Templates mit Jinja2	32
3.3	Tutorial Microblog 1.....	38
3.4	Musterlösungen zu Microblog	41
3.5	Zusammenfassung	42
4	Forms.....	43
4.1	Lernziele.....	43
4.2	Vorbereitungen	43
4.3	Das config.py File	44
4.4	Login Form	45
4.5	Form Template.....	47
4.6	Form Views.....	48
4.7	Erster Test der Form.....	48
4.8	Daten aus der Form akzeptieren.....	50
4.9	Hintergrund	53
4.10	Zusammenfassung	56
5	Datenbanken	57
5.1	Lernziele.....	57
5.2	Objekte und Tabellen	57
5.3	Verbindung zu einem Datenbanksystem.....	58
5.4	Datenmodell	60
5.5	Datenbank-Migrationen vorbereiten	62
5.6	Migration der Klasse User.....	62

5.7	Posts	64
5.8	Test der Datenbank in der REPL	66
5.9	Die Flask-Shell.....	67
5.10	Hintergrund.....	69
5.11	Zusammenfassung.....	77
6	Benutzer-Login	78
6.1	Lernziele	78
6.2	Passwörter.....	78
6.3	Die Erweiterung flask-login.....	80
6.4	Login in der View-Funktion	82
6.5	Logout.....	83
6.6	Seiten schützen	83
6.7	Anzeige des Benutzers in Templates	84
6.8	Erster Test	85
6.9	User registrieren.....	86
6.10	Musterlösungen	88
6.11	Test.....	89
6.12	Hintergrund.....	90
6.13	Zusammenfassung.....	93
7	Benutzerprofile	95
7.1	Lernziele	95
7.2	Die Profilseite	95
7.3	Avatare.....	97
7.4	Templates inkludieren	99
7.5	Zusätzliche Informationen zu User-Accounts	99
7.6	Der Profil-Editor.....	101
7.7	Musterlösungen	103
7.8	Zusammenfassung.....	104
8	Fehlerbehandlung	105
8.1	Lernziele	105
8.2	Fehler in Flask	105
8.3	Debug-Modus.....	107
8.4	Eigene Fehler-Seiten	108
8.5	Logging	111
8.6	FehlerbeHebung	115
8.7	Musterlösungen	116
8.8	Zusammenfassung.....	116
9	Follower.....	117
9.1	Lernziele	117
9.2	Beziehungen im Datenmodell	117

9.3	Follower hinzufügen und entfernen.....	119
9.4	Posts von Followern.....	120
9.5	Unit Tests	121
9.6	Integration in die App.....	125
9.7	Zusammenfassung	129
10	Blog Posts.....	130
10.1	Lernziele.....	130
10.2	Form, View-Funktion und Template	130
10.3	Andere User finden.....	132
10.4	Paginierung.....	134
10.5	Seiten-Navigation	136
10.6	Paginierung in der Profilseite.....	138
10.7	Zusammenfassung	139
11	E-Mail.....	140
11.1	Lernziele.....	140
11.2	Flask-Mail und pyjwt.....	140
11.3	Test von Flask-Mail.....	141
11.4	Eine Mail-Funktion	141
11.5	Passwort vergessen?	142
11.6	Der Passwort Reset Token	144
11.7	Das Passwort-Reset Email	147
11.8	Passwort-Reset durchführen.....	148
11.9	Asynchrone Emails	151
11.10	Zusammenfassung	153
12	Bootstrap.....	155
12.1	Lernziele.....	155
12.2	CSS-Frameworks	155
12.3	Vorher und Nacher	156
12.4	Flask-Bootstrap	158
12.5	Formulare mit Bootstrap	162
12.6	Liste der Posts mit Bootstrap.....	163
12.7	Zusammenfassung	165
13	Deployment	167
13.1	Lernziele.....	167
13.2	Deployment-Optionen.....	167
13.3	Deployment auf Linux.....	168
13.4	Versionierung und Deployment mit Git.....	174
13.5	Zusammenfassung	180
14	Ein RESTful API.....	181
14.1	Lernziele.....	181

14.2	REST	181
14.3	Das Datenformat JSON	183
14.4	Das API	184
14.5	Zwei einfache API-Endpunkte.....	185
14.6	Weitere Endpunkte für GET	190
14.7	Fehlerbehandlung	192
14.8	Authentifizierung	200
14.9	API-RouteN mit POST oder PUT.....	195
15	Anhang: Weitere Feldtypen in Formularen.....	202
15.1	Einfaches Auswahlfeld.....	202
15.2	Auswahlfeld mit Datenbank-Query	203

1 BASISTECHNOLOGIEN

Für das Verständnis der folgenden Kapitel benötigen Sie einige Grundkenntnisse des http-Protokolls, der Auszeichnungssprache HTML und Sie müssen wissen, was der Sinn und Zweck von CSS ist.

1.1 LERNZIELE

Nach der Bearbeitung dieses Kapitels können Sie

- Die Bedeutung des http-Protokolls für die Kommunikation im Internet beschreiben
- Die Methoden von http beschreiben und einsetzen
- Die Grundstruktur von HTML-Dokumenten beschreiben

1.2 PROTOKOLLE UND TECHNOLOGIEN

Das Hypertext Transfer Protokoll (http) ist die Grundlage der Kommunikation im Internet. Es basiert auf dem TCP-Protokoll. Webseiten benutzen die Auszeichnungssprache HTML (Hypertext Markup Language), dessen Erweiterung CSS (Cascading Style Sheets) und JavaScript, einer Programmiersprache, die im Browser ausgeführt wird und sie sehr häufig benutzt wird, um Webseiten interaktiv zu gestalten.

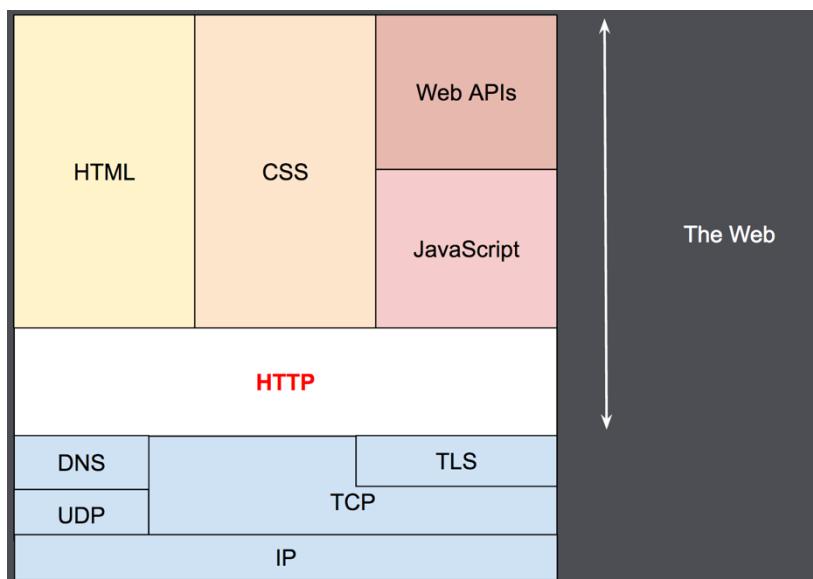


Abbildung 1-1 Basistechnologien. Quelle: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview>

Für die Web-Entwicklung auf der Client-Seite und auf der Serverseite existieren verschiedene Frameworks, die die Entwicklung von Websites unterstützen.

Beispiele für Frameworks für die Programmierung der Client-Seite sind

- React – Ein Framework für JavaScript
- Angular – Ein Framework für JavaScript

Beispiele für Frameworks auf der Server-Seite sind

- Flask – Ein 'Micro-Framework' in Python
- Symfony – Ein Framework für PHP
- Express.js - Ein Framework für JavaScript

1.3 HTTP

Das http-Protokoll wird für das Senden von Anfragen (Requests) von Clients an Server und für Antworten (Responses) der Server an Clients verwendet.

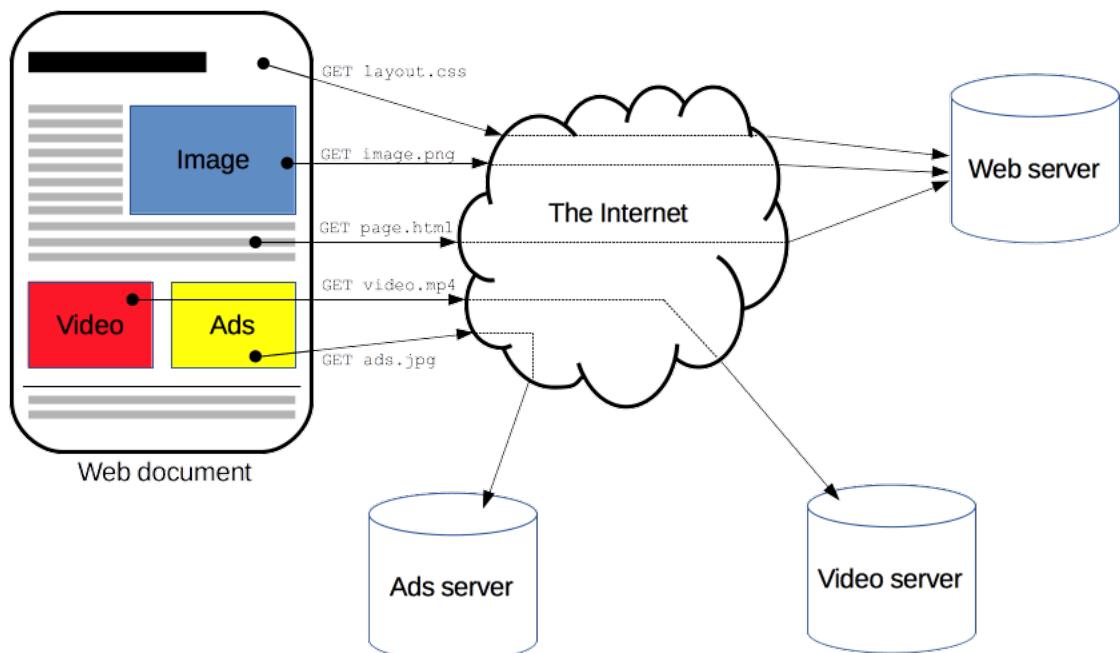


Abbildung 1-2 HTTP, Quelle: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview>

1.3.1 CLIENT, SERVER UND PROXIES

In der http-Kommunikation können verschiedene Stationen durchlaufen werden:

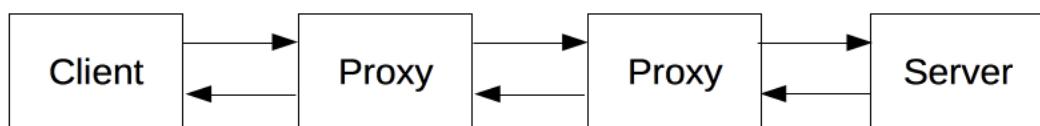


Abbildung 1-3 Stationen in der http-Kommunikation. Quelle:

- Web Browser (Client)
 - Initiiert immer die Kommunikation mit einem oder mehreren Servern
 - Zeigt die Seite auf Basis von HTML, CSS und JavaScript an
 - Führt clientseitige Logik aus, z.B. mit JavaScript
- Web Server
 - Erreichbar über URL `https://server.domain`
 - Erscheint aus Sicht der Clients als einzelne Maschine. Es kann sich dabei aber auch um eine zusammengehörige Menge von Servern handeln (Load Balancer, Cache-System, Webserver, Applikationsserver, Datenbankserver ...)

- Proxies sind Stellvertreter für einen anderen Server. Sie kontrollieren oft den Zugriff auf Server und Dienste oder bieten zusätzliche Funktionen an, wie
 - Caching (Erhöht Zugriffsgeschwindigkeit und Verfügbarkeit von Diensten)
 - Filterung (Malware, Kontrolle der verfügbaren Webinhalte)
 - Authentifizierung (Kontrolle des Zugriffs auf sensible Ressourcen)
 - Protokollierung (Zugriffe sollen historisch festgehalten werden)

Wichtige Eigenschaften des http-Protokolls sind:

- Es ist einfach. HTTP kommt mit sehr wenigen Methoden ("Verben") aus.
- Es ist für Maschinen wie auch für Menschen lesbar (Letzteres jedenfalls bis HTTP 1.1)
- Es ist erweiterbar. Über HTTP Header können neue Arten von Inhalt vereinbart werden
- Die Kommunikation ist zustandslos. Jeder Request mit der dazu gehörenden Antwort wird separat behandelt. Der Server "erinnert" sich nicht an vergangene Requests des Clients und seine Antworten (Mittels Cookies können aber Informationen über vergangene Request/Antwort Paare auf dem Client gespeichert werden und somit eine Art Session verwaltet werden).
- http ist überall verbreitet und standardisiert.

Die Kommunikation zwischen Client und Server besteht aus einzelnen Request / Response Paaren:

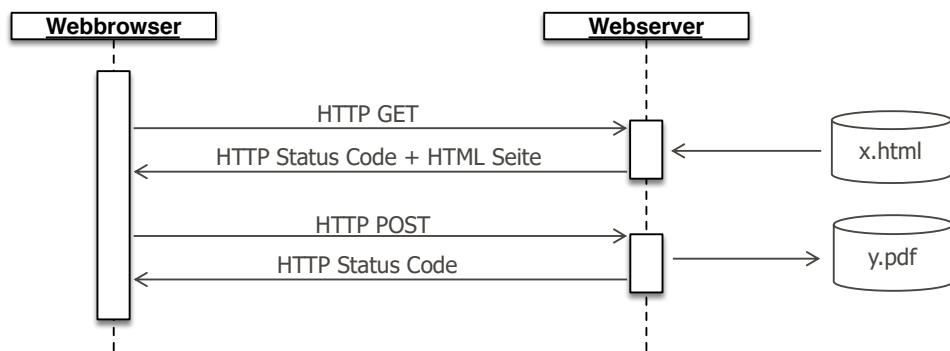


Abbildung 1-4 Beispiel Response / Request

1.3.2 METHODEN

http definiert Methoden. Wichtige Methoden sind:

- GET – Fordert etwas vom Server an
- POST – Übermittelt etwas zum Server
- OPTIONS - Informationen über verfügbare Kommunikationsoptionen abrufen
- HEAD - Fordert nur den Header eines Dokuments oder Quelle an
- PUT – Modifiziert Ressourcen auf dem Server oder legt dort neue Daten an
- DELETE – Löscht Ressourcen auf dem Server

1.3.3 STRUKTUR DER NACHRICHTEN

Eine Nachricht enthält die Elemente Header (Bei Requests und Responses) und Body (Bei Responses).

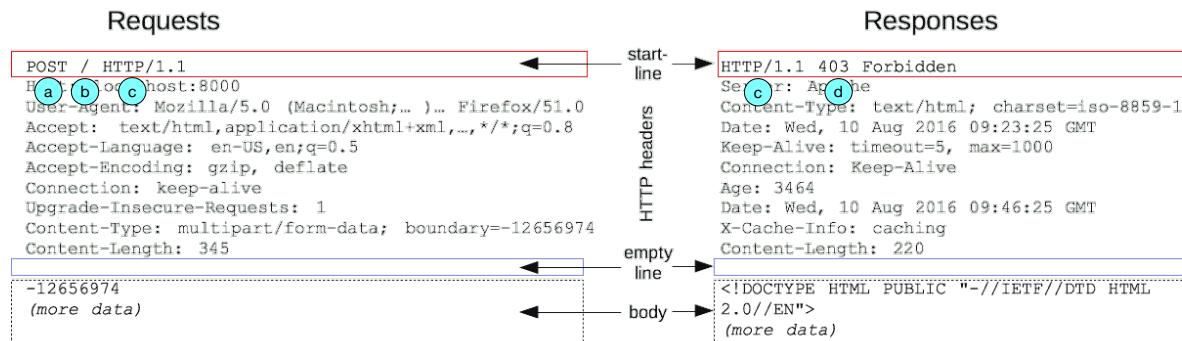


Abbildung 1-5 Quelle: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages>

1.3.4 HEADER

Die Startzeile besteht aus:

- Methode – Entweder 'Verb' (GET, PUT, POST, DELETE) oder 'Nomen' (HEAD, OPTIONS) bei Requests
- Ziel – URL / Pfad bei Requests
- HTTP-Version bei Requests und Responses
- Status-Code bei Responses

Der restliche Header hat je nach Art der Nachricht unterschiedliche Bereiche:

- Request-Header liefern Informationen wie Host, Browser und Betriebssystem, akzeptierte Formate, akzeptierte Sprachen und akzeptierte Kodierungen.
- General Headers liefern allgemeine Informationen wie keep-alive.
- Representation Headers beschreiben die Art des Inhalts und die Grösse der eigentlichen Nachricht.

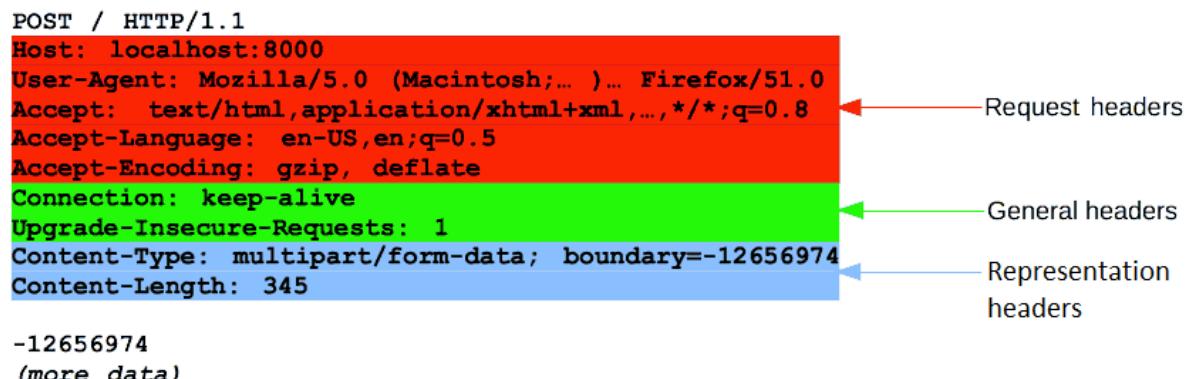


Abbildung 1-6 Quelle: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages>

1.3.5 BODY

Der Body enthält den eigentlichen Inhalt der Nachricht. Dabei gilt:

- Requests der Arten GET, HEAD, DELETE und OPTIONS brauchen keinen Body
- Requests der Arten PUT und POST schicken Daten und haben einen Body
- Responses des Servers liefern im Body Informationen. Bei einem Webserver ist das typischerweise ein HTML-Dokument.

1.3.6 HTTP STATUS-CODES

Eine Response enthält im Header immer einen dreistelligen Statuscode. Es gibt folgende Arten von Statuscodes:

- 1xx Codes für Informationen
- 2xx Codes für Erfolgreiche Operationen
- 3xx Codes für Umleitungen
- 4xx Codes für Client-Fehler
- 5xx Codes für Server-Fehler

Eine genaue Beschreibung der Status-Codes finden Sie hier:

<https://de.wikipedia.org/w/index.php?title=HTTP-Statuscode&oldid=218657268>

Mit Status Codes kommt der Benutzer von Webseiten meist nur bei Fehlern in Berührung:



Abbildung 1-7 Status Code 404, wenn eine URL nicht gefunden wurde

1.4 HTML UND CSS

In diesem Kapitel finden Sie eine sehr kurze Einführung HTML und CSS. In diesem Kurs ist es nicht notwendig, dass Sie vertiefte HTML- und CSS-Kenntnisse haben, aber Sie sollten einige wenige Grundlagen beherrschen, die für das Verständnis der folgenden Kapitel hilfreich sind.

1.5 LERNZIELE

Nach der Bearbeitung dieses Kapitels können Sie

- Die Rolle von HTML bei der Erstellung von Webseiten erklären
- Die Grundstruktur von HTML-Dokumenten beschreiben
- Einige Wichtige Elemente von HTML anwenden
- Die Rolle von CSS für Webseiten erklären

1.5.1 HTML

Die Hypertext Markup Language ist eine Sprache zur Strukturierung von Webseiten. Browser können HTML-Dokumente anzeigen. Die Dokumente enthalten Texte mit Hyperlinks, Bilder und andere Inhalte. Neben den vom Browser angezeigten Inhalten können HTML-Dateien zusätzliche Angaben enthalten, z. B. über die im Text verwendeten Zeichensatz-Kodierungen, die verwendeten Sprachen oder den Autor.

Dem Dokument wird durch Auszeichnungen (englisch *markup*) eine Struktur gegeben.

1.5.2 BEISPIEL 1



Abbildung 1-8 Beispiel 1 im Browser

HTML-Dokument dazu

```
<html>
  <head>
    <title>Startseite</title>
  </head>
  <body>
    <h1>DBWE Test Nummer 1!</h1>
    <p>Willkommen beim NGINX index file.</p>
  </body>
</html>
```

Auszeichnungen werden von tags eingerahmt. Im Beispiel beginnen mit `<tag>` und enden mit `</tag>`.

- Das gesamte Dokument wird eingerahmt von `<html>` am Anfang und `</html>` am Ende.
- Es gibt dazwischen zwei Bereiche: Von `<head>` bis `</head>` und von `<body>` bis `</body>`.
- Im Head gibt es eine Zeile, die mit `<title>` beginnt und mit `</title>` endet. Der Text 'Startseite', der dazwischen steht, wird im Browsetab als Titel angezeigt.
- Im Body gibt es zwei Textzeilen. Die Zeile mit `<h1>` ist eine Überschrift.
- Die zweite Zeile mit `<p>` ist ein Absatz.

Im nächsten Abschnitt finden sie ein weiteres Beispiel, das deutlich mehr Elemente und neue Tags enthält.

1.5.3 BEISPIEL 2



Willkommen im body des index.html File!

Es gibt 6 verschiedene Überschriftgrößen, <h1> bis <h6>. Diese hier hat die Grösse <h2>

Die Überschrift wurde mit <h2> eingeleitet und mit </h2> abgeschlossen

Dies ist ein Paragraph. Er wird mit <p> eingeleitet. Ein Paragraph kann beliebig lang sein und HTML kümmert sich nicht um Zeilenumbrüche, es sei denn, ein Zeilenumbruch wird mit
 eingefügt:
Dann geht es in der nächsten Zeile weiter.

Hier gibt es interne Links zu weiteren Beispielen

- [Textformatierung](#)
- [Listen](#)
- [Tabellen](#)
- [Mehr Style mit CSS](#)

Externe Links zu Tutorials

- [HTML Tutorial W3 Schools](#)
- [CSS Tutorial W3 Schools](#)

Abbildung 1-9 Beispiel 2 im Browser

Das HTML-Dokument dazu:

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <title>Startseite</title>
  </head>
  <body>
    <h1><center>DBWE – Demo statische Website</h1>
    <p><center>Willkommen im body des index.html File!</p>
    <hr></center>
    <h2>Es gibt 6 verschiedene Überschriftgrößen, &lt;h1&gt; bis &lt;h6&gt;.
      Diese hier hat die Grösse &lt;h2&gt;</h2>
    <p>
      Die Überschrift wurde mit &lt;h2&gt; eingeleitet und
      mit &lt;/h2&gt; abgeschlossen
    </p>
    <p>
      Dies ist ein Paragraph. Er wird mit &lt;p&gt; eingeleitet.
      Ein Paragraph kann beliebig lang sein und HTML kümmert
      sich nicht um Zeilenumbrüche, es sei denn, ein Zeilenumbruch
      wird mit &lt;br&gt; eingefügt: <br>
      Dann geht es in der nächsten Zeile weiter.
    </p>
    <h2>Hier gibt es interne Links zu weiteren Beispielen</h2>
    <ul>
      <li><a href="textformate.html">Textformatierung</a></li>
      <li><a href="listen.html">Listen</a></li>
      <li><a href="tabellen.html">Tabellen</a></li>
      <li><a href="style-css.html">Mehr Style mit CSS</a></li>
    </ul>
    <h2>Externe Links zu Tutorials</h2>
    <ul>
      <li><a href="https://www.w3schools.com/html/default.asp">HTML Tutorial</a></li>
      <li><a href="https://www.w3schools.com/css/default.asp">CSS Tutorial</a></li>
    </ul>
  </body>
</html>
```

Diese HTML-Elemente werden im nächsten Abschnitt erklärt.

1.5.4 TYPISCHE HTML-ELEMENTE

<head> ... </head>

Im Abschnitt `<head>`, Unterabschnitt `<meta>` befindet sich typischerweise eine Angabe `charset=`. Wenn Sie Umlaute und Sonderzeichen ausserhalb des ASCII-Zeichensatzes darstellen wollen, setzen Sie hier `charset=utf-8`:

```
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8">
  <title>Startseite</title>
</head>
```

`<title>` setzt einen Titel, der im Browser-Tab angezeigt wird

<body> ... </body>

Dies ist der eigentliche Inhalt der Seite. Er enthält diverse andere Elemente, die wieder aus Tags und Text bestehen.

<h1> bis <h6>

Überschriften in verschiedenen Textgrössen. `<h1>` ist die grösste, `<h6>` die kleinste Überschrift

<p> ... </p>

Ein neuer Absatz (Paragraf)

**
**

Ein Zeilenumbruch innerhalb eines Paragrafen

** ... **

Eine Auflistung mit bullet points. Darin verschachtelt werden die einzelnen List-Items mit `Text` angegeben.

** ... **

Eine nummerierte Aufzählung. Darin verschachtelt werden die einzelnen List-Items mit `Text` angegeben

Linktext

Ein Link auf eine andere HTML-Datei der gleichen Website. Statt des Dateinamens wird der Linktext angezeigt. Der Linktext ist optional.

Linktext

Ein Link auf eine andere Website. Statt des URL wird der Linktext angezeigt. Der Linktext ist optional.

1.5.5 WEITERE INFORMATIONEN ZU HTML

Dies ist nur eine kleine Auswahl der Möglichkeiten, die sie bei der Gestaltung von HTML-Dokumenten zur Verfügung haben. Falls Sie mehr über HTML lernen wollen, empfiehlt sich das folgende Tutorial
<https://www.w3schools.com/html/default.asp>

Sie können die in diesem Abschnitt beschriebenen Elemente anhand der Übungen im folgenden Abschnitt Aufgaben studieren und anwenden.

Im weiteren Verlauf des Kurses werden Sie HTML benutzen, aber die wesentlichen Elemente werden Ihnen als fertige Dateien geliefert.

1.6 AUFGABEN ZU HTML

Sie benötigen die Dateien aus dem GitHub Repository unter Flask/kap-1:

index.html	listen.html	style-css.html	tabellen-css.html
tabellen.html	textformate.html		

- 1) Falls Sie git bereits installiert und für den Zugriff auf das Repository verwendet haben, verwenden Sie dieses Kommando in Ihrem bestehenden Git-Verzeichnis:

```
$ git pull https://github.com/dozent2018/IFA\_DBWE
```

Falls Sie git noch nicht installiert oder verwendet haben, tun sie das bitte jetzt: <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git> und verwenden sie anschliessend in einem lokalen Verzeichnis Ihrer Wahl:

```
$ git clone https://github.com/dozent2018/IFA\_DBWE
```

- 2) Geben Sie in einem Browser folgendes ein (Eventuell müssen Sie den Pfad anpassen):

file:///meinverzeichnis/IFA_DBWE/Flask/kap-1/index.html

Sie sollten nun die Seite aus dem Beispiel 2 sehen.

- 3) Im Browser können Sie die Anzeige und den Quelltext gemeinsam ansehen. Für den Chrome-Browser machen Sie dafür Folgendes:

- Windows: [F12] oder die Tastenkombination [Strg]+[Umschalt]+[J].
- Mac: Tastenkombination [⌘]+[Optionstaste]+[J].

Oder sie wählen aus dem Menu Anzeigen > Entwickler > Quelltext anzeigen.

Über den Reiter "Elements" kommen Sie in die Ansicht des Seiten-Quelltextes

- 4) Probieren Sie die Webseiten aus. Lesen Sie die Angaben im Browserfenster und vergleichen Sie diese mit dem jeweiligen Seitenquelltext. Folgen Sie den verschiedenen Links.
- 5) Machen Sie sich eine Skizze mit den 6 Seiten, die zeigt, wie diese durch Links verbunden sind.
- 6) Experimentieren Sie mit den bestehenden Seiten, fügen Sie etwas hinzu oder ändern Sie etwas
- 7) Fügen Sie eine neue eigene HTML-Datei im gleichen Verzeichnis hinzu, probieren Sie selbst die verschiedenen Möglichkeiten aus und verlinken Sie Ihre neue Seite über einen neuen Link in index.html

1.7 CSS

CSS (Cascading Style Sheets) erlauben es, die Darstellung der HTML-Seiten zu steuern. Damit können Sie unter anderem die folgenden Eigenschaften steuern:

- Textfarben
- Hintergrundfarben
- Fonts
- Schriftgrößen
- Layouts

CSS kann einem HTML-Dokument auf 3 verschiedene Arten hinzugefügt werden:

- Inline, durch das hinzufügen des style-Attributs in einem HTML-Element
- Intern, durch ein `<style>`-Element innerhalb von `<head>`
- Extern, durch ein `<link>`-Element, das auf eine separate CSS-Datei verweist

1.7.1 BEISPIELE INLINE CSS

Die folgenden style-Angaben stehen im `<body>` direkt beim jeweiligen Element. Sie beeinflussen nur das jeweilige Element

```
<h1 style="color:blue;">Eine blaue Überschrift</h1>
<p style="color:red;">Ein roter Paragraf.</p>
```

1.7.2 BEISPIELE INTERNES CSS

Internes CSS steht im jeweiligen Dokument in einem Bereich `<style>` innerhalb von `<head>`. Die folgenden Angaben beeinflussen den Hintergrund des Body, alle Überschriften der Grösse h1 und alle Paragrafen, es sei denn, es wird inline etwas anderes vorgeschrieben.

```
<head>
<style>
    body {background-color: powderblue;}
    h1   {color: blue;}
    p    {color: red;}
</style>
</head>
```

1.7.3 BEISPIELE EXTERNES CSS

Externes CSS steht in einer separaten Datei. Styles, die in einer solchen Datei definiert werden können in mehreren HTML-Dokumenten über einen Link eingebunden werden.

```
<head>
    <link rel="stylesheet" href="styles.css">
</head>
```

Diese Methode ist die beste, wenn für alle Seiten ein einheitliches Design eingerichtet werden soll.

**1.7.4 <DIV> UND **

Die zwei HTML-Tags `<div>` und `` sind im Zusammenhang mit CSS nützlich. `<div>` steht für einen Bereich im `body` und ist definiert einen abgegrenzten Block in der vollen Breite. Das Element `` definiert einen Bereich innerhalb eines Elements.

1.7.5 WEITERE INFORMATIONEN ZU CSS

Die Beispiele in diesem Lehrmittel zeigen nur einen sehr kleinen Auschnitt der Möglichkeiten von CSS. Wenn Sie mehr darüber erfahren wollen, sehen Sie sich dieses Tutorial an:

<https://www.w3schools.com/css/default.asp>

Im weiteren Verlauf des Kurses werden Sie HTML und CSS benutzen, aber die wesentlichen Elemente werden Ihnen als fertige Dateien geliefert.

1.8 ZUSAMMENFASSUNG

- Die Basis der Kommunikation im Internet ist das http-Protokoll, das standardisiert ist.
- Es basiert auf Requests von Clients und Responses von Servern
- Es definiert Methoden wie GET, POST, PUT
- Eine Kommunikation ist auf je ein Request / Response Paar beschränkt
- Es definiert Status-Codes, die vom Server in den Responses verschickt werden
- HTML ist die Auszeichnungssprache (markup language) für Webseiten, die durch Browser verarbeitet und dargestellt werden können.
- HTML-Dokumente haben eine hierarchische Struktur, die durch Tags erzeugt wird
- Es gibt mindestens die Elemente `<head>` und `<body>`
- Der `<head>` enthält wichtige Metainformationen und Stil-Angaben und allenfalls einen Titel für den Client
- Der `<body>` enthält die eigentlichen Inhalte der Webseite. Hier können viele verschiedene Tags für Textformate, Strukturen und Darstellungsoptionen stehen.
- Links im Body können auf andere Seiten der gleichen Website verweisen oder auf andere Websites.
- CSS ermöglicht eine strukturierte Steuerung des Aussehens einer Website

2 INSTALLATION

In diesem Kapitel installieren Sie das Web-Framework Flask, mit dem in allen folgenden Kapiteln gearbeitet wird. Nachdem Sie im ersten Kapitel mit statischen Webseiten gearbeitet haben, ist dies die Grundlage für den Aufbau dynamischer Webseiten.

2.1 LERNZIELE

Nach der Bearbeitung dieses Kapitels können Sie:

- Die Rolle von Flask in einer Webapplikation beschreiben
- Flask auf Ihrem Computer installieren und benutzen

2.2 ARCHITEKTUR

Der Ablauf bei einem Request des Browsers an eine Flask-App kann so dargestellt werden:

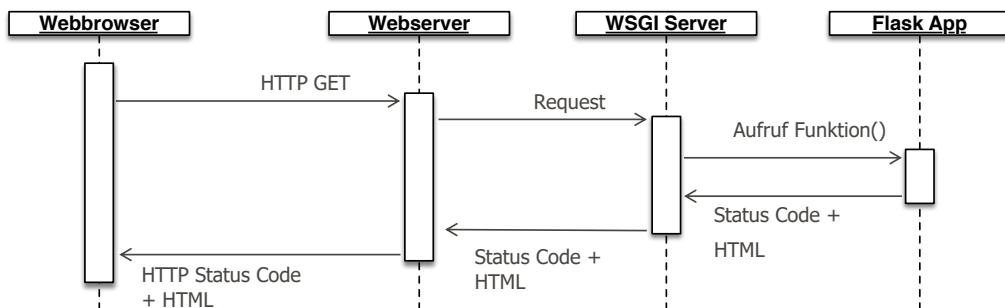


Abbildung 2-1 Ablauf Request / Response

Das **Web Server Gateway Interface (WSGI)** ist eine Schnittstellen-Spezifikation für Python, die die Art der Kommunikation zwischen Webservern und Webframeworks bzw. Web Application Servern festlegt.

Der **WSGI-Server** stellt der App einige Objekte zur Verfügung, die damit über Requests des Webservern benachrichtigt wird.

Die **App** muss eine Funktion liefern, die auf den Request reagiert.

Der **Webserver** agiert in Abbildung 2-1 als Proxy-Server für den WSGI-Server. Er erhält das von der App erzeugte HTML-Dokument und schickt es seinerseits an den Webbrowser.

Um die Entwicklung zu vereinfachen, enthält Flask einen kleinen eingebauten Webserver, der WSGI beherrscht:

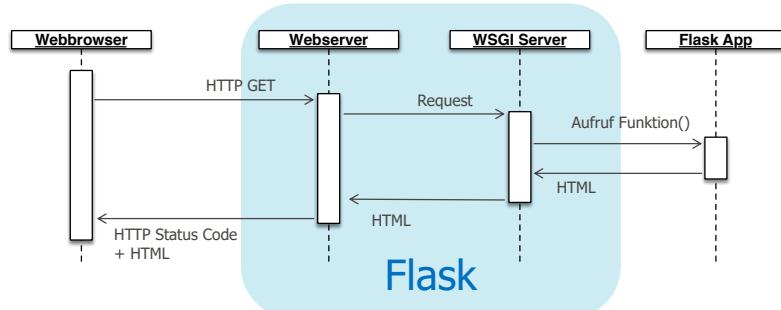


Abbildung 2-2 Flask enthält einen kleinen Webserver

Sie werden den einbauten Webserver von Flask in diesem und den nächsten Kapiteln benutzen. In einen späteren Kapitel werden Sie Ihn durch einen anderen Server ersetzen, der auch für den Produktionseinsatz geeignet ist.

2.3 VIRTUELLES ENVIRONMENT

In Ihren Projekten brauchen Sie eventuell verschiedene Versionen von Python, von Modulen und Paketen wie Flask und seinen Erweiterungen. Um sicher zu sein, die richtigen Versionen zur Verfügung zu haben, richten Sie für jedes Projekt eine separate Umgebung ein:

```
Globale Installation:  
$ which python3  
/usr/local/bin/python3  
  
$ python3 --version  
Python 3.9.0
```

Virtuelles Environment für Projekt 'flaskintro':
\$ which python3
/home/jochen/Flask/flaskintro/venv/bin/python3
\$ python3 --version
Python 3.10.0

Virtuelles Environment für Projekt 'projekt42':
\$ which python3
/home/jochen/Flask/projekt42/venv/bin/python3
\$ python3 --version
Python 3.7.0

Abbildung 2-3 virtuelle Environments

2.4 TUTORIAL 1

Die folgenden Unterkapitel enthalten Kommandos und Code-Beispiele. Führen Sie diese als praktische Übung durch:

- Sie richten ein virtuelles Environment ein
- Sie installieren darin Flask
- Sie erzeugen eine kleine 'Hello, World' Applikation, um Flask zu testen

2.4.1 FÜR MICROSOFT WINDOWS

Benutzen Sie **cmd** im Terminal! Sie können in einem beliebigen Verzeichnis arbeiten. Bei den folgenden Beispielen wird angenommen, dass Ihr Verzeichnis C:\Flask\kap-2 heisst.

1. Richten Sie in Ihrem Arbeitsverzeichnis ein virtuelles Environment ein

```
$ cd kap-2  
  
$ python -m venv myenv  
  
$ dir myenv\Scripts  
Activate.ps1  activate.csh  pip      pip3.9      python3  
activate   activate.fish  pip3      python     python3.9  
  
$ dir myenv\lib  
python3.9
```

2. Aktivieren Sie das virtuelle Environment. Am Anfang Ihres Prompts sollte danach (myenv) erscheinen.

```
$ myenv\Scripts\activate  
(myenv) $ where python  
C:\Users\jochen\Flask\kap-2\myenv\bin\python
```

3. Installieren Sie Flask

```
(myenv) $ pip install flask  
# Sie sehen diverse Ausgaben ...  
  
(myenv) $ pip freeze  
# Sie sehen diverse Ausgaben ...
```

4. Legen Sie ein neues Verzeichnis mit dem Namen 'app' an und wechseln Sie in das Verzeichnis.

```
(myenv) $ mkdir app  
(myenv) $ cd app
```

5. Legen Sie in kap-2\app die Dateien __init__.py und routes.py an

```
# kap-2/app/__init__.py  
from flask import Flask  
app = Flask(__name__)  
from app import routes  
  
# kap-2/app/routes.py  
from app import app  
@app.route('/')  
@app.route('/index')  
def index():  
    return "<h1>Hello, Flask!</h1>"
```

6. Legen sie im Directory darüber die Datei hello_flask an

```
# kap-2/hello_flask.py  
from app import app
```

7. Überprüfen Sie, ob Ihre Verzeichnisstruktur so aussieht:

```
kap-2  
└── app  
    ├── __init__.py  
    └── routes.py  
└── hello_flask.py
```

8. Setzen Sie die Variable FLASK_APP

```
(myenv) $ set FLASK_APP=hello_flask.py
```

2.4.2 FÜR MAC OS UND LINUX

Benutzen Sie ein Terminal. Sie können in einem beliebigen Verzeichnis arbeiten. Bei den folgenden Beispielen wird angenommen, dass das Verzeichnis kap-2 heisst.

1. Richten Sie in Ihrem Arbeitsverzeichnis ein virtuelles Environment ein

```
$ cd kap-2  
$ pip install virtualenv  
$ python3 -m venv myenv  
  
$ ls myenv/bin  
Activate.ps1  activate.csh  pip      pip3.9      python3  
activate     activate.fish  pip3      python    python3.9  
  
$ ls myenv/lib  
python3.9
```

2. Aktivieren Sie das virtuelle Environment. Am Anfang Ihres Prompts erscheint (myenv)

```
$ source myenv/bin/activate  
(myenv) $ which python  
..../IFA_DBWE/Flask/kap-2/myenv/bin/python
```

3. Installieren Sie Flask

```
(myenv) $ pip install flask  
# Sie sehen diverse Ausgaben ...  
  
(myenv) $ pip freeze  
# Sie sehen eine Liste der installierten Pakete mit Versionen ...
```

4. Legen Sie ein neues Verzeichnis mit dem Namen 'app' an und wechseln Sie in das Verzeichnis

```
(myenv) $ mkdir app  
(myenv) $ cd app
```

5. Legen Sie die Dateien __init__.py und routes.py in app an

```
# kap-2/app/__init__.py  
from flask import Flask  
app = Flask(__name__)  
from app import routes  
  
# kap-2/app/routes.py  
from app import app  
@app.route('/')  
@app.route('/index')  
def index():  
    return "<h1>Hello, Flask!</h1>"
```

6. Legen sie im Directory darüber die Datei hello_flask an

```
# kap-2/hello_flask.py
from app import app
```

7. Überprüfen Sie, ob Ihre Verzeichnisstruktur so aussieht:

```
kap-2
├── app
│   ├── __init__.py
│   └── routes.py
└── hello_flask.py
```

8. Setzen Sie die Variable FLASK_APP

```
(myenv) $ export FLASK_APP=hello_flask.py
```

2.4.3 FÜR ALLE BETRIEBSSYSTEME

9. Starten Sie Flask:

```
(myenv) $ flask run
* Serving Flask app 'hello_flask.py' (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production
  deployment.
    Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [27/Feb/2022 18:29:53] "GET / HTTP/1.1" 200 -
```

(Der Server läuft im Terminal-Fenster, bis Sie ihn mit CTRL-C beenden)

10. Öffnen Sie einen Browser-Tab und geben Sie folgenden URL ein:

<http://localhost:5000/>

Das Ergebnis sollte so aussehen:



Abbildung 2-4 Anzeige von <http://localhost:5000/>

11. Fügen Sie in routes.py eine weitere Funktion hinzu:

```
@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, {}!</h1>'.format(name)
```

12. Beenden Sie den Server und starten Sie ihn neu

```
...  
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)  
127.0.0.1 - - [27/Feb/2022 18:29:53] "GET / HTTP/1.1" 200 -
```

(Ihre Eingabe: CTRL-C)

```
$ flask run
```

13. Probieren Sie im Browser die URL <http://localhost:5000/user/jochen> (Verwenden Sie statt 'jochen' Ihren eigenen Namen)

14. Beenden Sie den Server und das virtuelle Environment

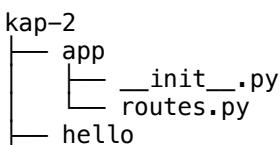
```
...  
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)  
127.0.0.1 - - [27/Feb/2022 18:29:53] "GET / HTTP/1.1" 200 -
```

(Ihre Eingabe: CTRL-C)

```
$ deactivate
```

2.4.4 ERKLÄRUNG DES BEISPIELS

Hier ist noch einmal die Verzeichnis-Struktur:



Sie haben Sie diesen Code für `__init__.py` geschrieben:

```
from flask import Flask
app = Flask(__name__)
from app import routes
```

Es enthält zwei Importe und die Erzeugung eines Flask-Objekts

- Zeile 1: Die Klasse Flask wird aus dem Modul flask importiert.
- Zeile 2: Der Inhalt von `__name__` wird dem Konstruktor der Klasse Flask() als Argument übergeben. Das erzeugt ein neues Flask-Objekt. Die Variable `__name__` ist eine vordefinierte Python-Variable, die sie bereits aus dem Python-Teil kennen. Wird Sie in einer Datei mit dem ebenfalls vordefinierten Namen `__init__.py` benutzt, enthält Sie den Namen des Verzeichnisses, in dem es steht, in diesem Falle `app`. Dadurch wird `app` zum Namen des Moduls oder Package.
- Zeile 3: Der Inhalt von `routes.py` im gleichen Verzeichnis wird importiert.

Es gibt zweimal die Bezeichnung **app**:

- Das **Package app** wird durch den Verzeichnisnamen `app` definiert und in der import-Anweisung in Zeile 3 benutzt.

- Der **Variablenname app** wird als Name des neuen Flask-Objekts benutzt. Die Variable app wird daher ebenfalls zum Bestandteil des Package 'app'

Warum steht der zweite Import am Ende und nicht am Anfang? Das erklärt sich, wenn man den Import in routes.py im gleichen Package 'app' betrachtet:

```
from app import app
```

Würde in `__init__.py` die Zeile `from app import routes` nicht am Ende, sondern am Anfang stehen, käme es zu einem zirkulären Import: app wird in routes.py aus dem Package app importiert und in `__init__.py` wird wiederum routes importiert, die Variable app, die importiert werden soll, ist aber dann noch gar nicht existent

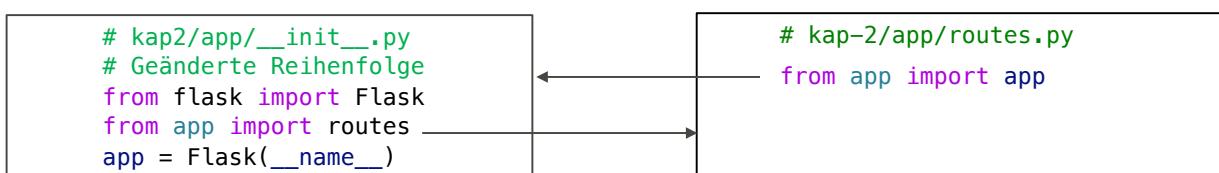


Abbildung 2-5 Problematischer Import

```
Error: While importing 'app', an ImportError was raised:
Traceback (most recent call last):
  File ... /app/__init__.py", line 4, in <module>
    from app import routes
  File ... /app/routes.py", line 4, in <module>
    from app import app
ImportError: cannot import name 'app' from partially initialized module 'app'
(most likely due to a circular import) (.../app/__init__.py)
```

Dass der Import von routes am Ende steht, vermeidet man einen solchen Zirkel.

Zentral für das Funktionieren des Beispiels im Tutorial 1 ist der folgende Code in routes.py, der sogenannten View-Funktion:

```
3 @app.route('/')
4 @app.route('/index')
5 def index():
6     return "<h1>Hello, Flask!</h1>"
```

In den Zeilen 3 und 4 wird ein Decorator `@app.route` benutzt. Er wird durch das Flask Framework definiert und bewirkt, dass die Funktion `index()` aufgerufen wird, sobald ein Request für die relative Route '/' (der letzte Teil des URL <http://localhost:5000/>) oder '/index' (der letzte Teil des URL <http://localhost:5000/index>) eintrifft.

In Zeile 6 gibt `index()` das HTML-Dokument an den WSGI-Server zurück, der es an den Webserver schickt. Der Webserver schickt es zusammen mit einem Statuscode zurück an den Client.

2.5 HINTERGRUND

Dieses Kapitel enthält weitere Informationen zu den Abläufen in Flask bei der Beantwortung eines Requests. Die Lektüre ist keine Voraussetzung für das Verständnis der folgenden Kapitel. Sie können darauf bei Bedarf auch später wieder zurückkommen.

2.5.1 REQUESTS

Wie kommt die App zum Request des Clients und wie wird die angefragten Route übermittelt? Für jeden http-Request passiert folgendes:

1. Der WSGI-Server (im Tutorial der in Flask eingebaute Server) erhält einen Request
2. Flask erzeugt ein Request Context-Objekt, das den http-Request enthält sowie ein Application Context-Objekt und 'pusht' sie. (Siehe die Diskussion von Context-Objekten in den Kapiteln zu Dateien und zu Datenbanken im Python-Teil).
3. Die App kann danach auf beide Objekte zugreifen. Die Variablen in den Context-Objekten können in der App wie globale Variablen verwendet werden.
4. Das Request-Objekt enthält die Route (z.B. '/'). Die dafür definierte Funktion ist in einer URL-Map gespeichert.
5. Die App reagiert auf den Request mit dem Aufruf der in der URL-Map gefundenen Funktion und stellt als Antwort ein Response-Objekt zur Verfügung.
6. Die Request und Application Context-Objekte werden wieder entfernt

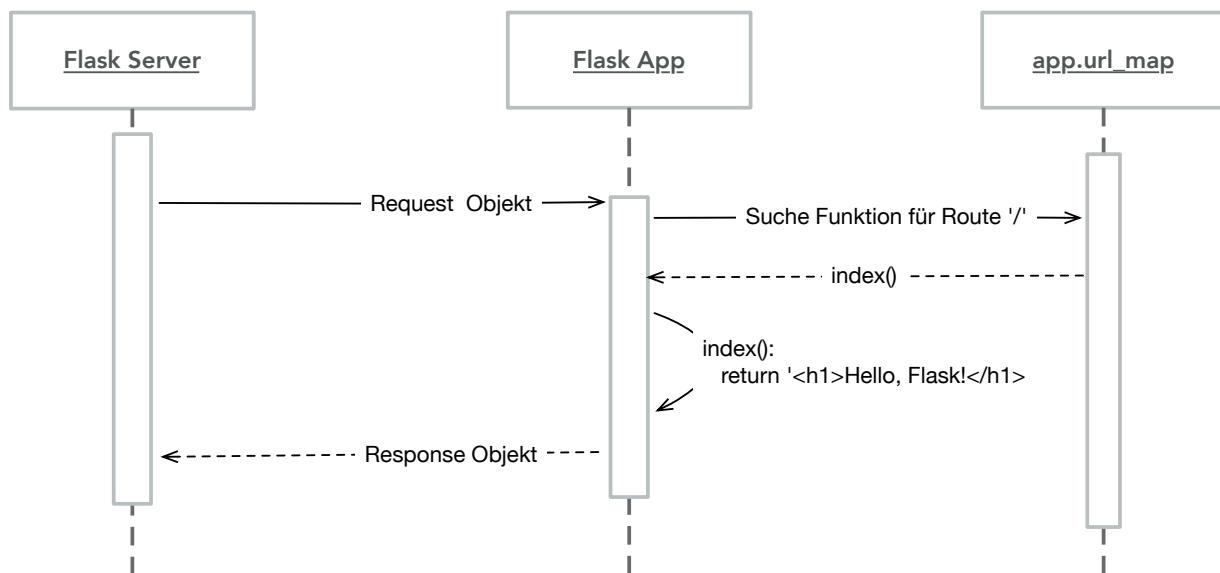


Abbildung 2-6 Ablauf Request / Response im Detail

Die Variablen der Application- und Request-Objekte

Variable	Context	Beschreibung
current_app	Application	Die Applikations-Instanz für die aktive App
g		Dient der Speicherung von Informationen, die während der Verarbeitung eines Requests erhalten bleiben sollen. Wird beim nächsten Request gelöscht.
request	Request	Enthält Informationen zum HTTP-Request des Clients
session		Dient der Speicherung von Informationen, die zwischen zwei Requests erhalten bleiben sollen.

Einige Attribute von request

Attribut	Inhalt
request.headers	Ein Dictionary mit allen http-Header-Informationen
request.method	Die http-Methode (z.B. GET oder POST)
request.host	Der Host im Request, inklusive Port
request.url	Die komplette URL im Request
request.args	Falls im URL Argumente angegeben sind, stehen sie in diesem Dictionary in der Form {name : wert} zur Verfügung.
request.path	Die Pfad-Komponente im URL
request.cookies	Ein Dictionary mit allen Cookies im Request

Weitere Attribute und Methoden finden sie hier:

<https://tedboy.github.io/flask/generated/generated/flask.Request.html>

Die Variablen verwenden

Die Variablen des Application Context und des Request Context kann man in der App verwenden.

Fügen Sie testweise die folgenden Zeilen an die View-Funktion routes.py an:

```
from flask import request
@app.route('/useragent')
def uagent():
    user_agent = request.headers.get('User-Agent')
    return '<p> Ihr Browser: {}</p>'.format(user_agent)

from flask import current_app
@app.route('/appname')
def appname():
    name = current_app.name
    return '<p> Ihre App heisst: {}</p>'.format(name)
```

Aktivieren Sie das virtuelle Environment und starten Sie den Server erneut.

```
$ source myenv/bin/activate      # Windows: myenv\Scripts\activate  
$ export FLASK_APP=hello_flask.py  
$ flask run
```

Im Browser können Sie nun folgende URLs ausprobieren:

<http://localhost/useragent>

<http://localhost/appname>

Die URL-Map ansehen

Flask führt für jede Applikation eine URL Map, in der für jede Route die passende Funktion gespeichert ist. Die Einträge in der URL Map werden durch die Verwendung des Dekorierers @app.route(route) erzeugt. Sie können sich den Inhalt in der REPL ansehen:

```
$ source myenv/bin/activate      # Windows: myenv\Scripts\activate  
$ (myenv) python3  
Python 3.9.5 (v3.9.5:0a7dcbdb13, May  3 2021, 13:17:02)  
[Clang 6.0 (clang-600.0.57)] on darwin  
Type "help", "copyright", "credits" or "license" for more information.  
  
>>> from hello_flask import app  
>>> app.url_map  
Map([<Rule '/useragent' (OPTIONS, HEAD, GET) -> uagent>,  
     <Rule '/appname' (OPTIONS, HEAD, GET) -> appname>,  
     <Rule '/index' (OPTIONS, HEAD, GET) -> index>,  
     <Rule '/' (OPTIONS, HEAD, GET) -> index>,  
     <Rule '/static/<filename>' (OPTIONS, HEAD, GET) -> static>,  
     <Rule '/user/<name>' (OPTIONS, HEAD, GET) -> user>])
```

Das Response-Objekt

Neben einem HTML-String muss in der Regel noch ein Status-Code zurückgegeben werden. Wird dieser bei return in der View-Funktion nicht angegeben, ist der Default Code 200. Es kann aber auch ein anderer Code angegeben werden.

```
@app.route('/unsinn')  
def unsinn:  
    return '<h1>Bad Request</h1>', 400
```

Der zweite Wert, 400 wird im Response-Objekt gespeichert. Einige Attribute und Methoden eines Response-Objekts sind:

Attribut	Inhalt
Response.headers	Ein Dictionary mit allen http-Header-Informationen, die geschickt werden sollen
Response.status_code	Der Status-Code, der geschickt werden soll
Response.content_length	Länge des Response-Body

Response.content_length	Der Medien-Typ des Response-Body
Response.set_data	Setzt den Inhalt des Body als String oder Bytes
Response.set_cookie()	Fügt der Response einen Cookie hinzu
Response.delete_cookie()	Entfernt einen Cookie

2.6 ZUSAMMENFASSUNG

- Das Web Server Gateway Interface (WSGI) ist eine Spezifikation für die Kommunikation zwischen Webservern und Python-Programmen.
- Eine Python-Webapplikation liegt 'hinter' einem Webserver und einem WSGI-Server. Es gibt Webserver, die WSGI-Fähig sind, dann fällt der separate WSGI-Server weg.
- Flask enthält einen kleinen Webserver, der WSGI beherrscht. Dieser wird mit flask run gestartet und ist für die Entwicklung von Apps ausreichend.
- Der Server gibt Requests des Clients an die Applikation weiter
- In einer App definieren Sie Routen mittels des Decorators @app.route. Damit legen Sie die Funktionen fest, die für bestimmte URLs aufgerufen werden sollen. Solche Funktionen geben in den Beispielen dieses Kapitels einen String mit einem HTML-Dokument zurück.
- Die Applikation antwortet den Server mit dem HTML-Dokument und einem Status-Code
- Der Server schickt das HTML-Dokument und den Status-Code an den Browser

3 TEMPLATES

In den Beispielen des vorhergehenden Kapitels wurde als Antwort der App auf einen Request in der View-Funktion einfach ein HTML-String zurückgegeben. Um die Grundlagen der Kommunikation mit Flask zu verstehen war das ausreichend, aber in einer professionellen App sollte man folgende Augaben nicht vermischen:

- Präsentationslogik (Anzeige im Browser)
- Geschäftslogik (Abläufe, Regeln, Berechnungen, usw.)
- Datenspeicherung (optional)

Mit Templates können die Präsentationsaufgaben von der Geschäftslogik getrennt werden.

3.1 LERNZIELE

- Nach der Bearbeitung dieses Kapitels können Sie:
- Templates mit Jinja2 erstellen
- Mit einem Basis-Template und davon abgeleiteten Untertemplates arbeiten
- Ihrer Webapplikation mit Bootstrap ein besseres Aussehen geben

3.2 TEMPLATES MIT JINJA2

Bei der Installation von Flask im Kapitel 2 wurde die Template Engine Jinja2 mit installiert. Damit können Sie in einer einfachen Template-Sprache und etwas HTML die Darstellung Ihrer Seiten in separaten Template-Dateien definieren. Flask sucht diese Dateien in einem Unterverzeichnis templates im Hauptverzeichnis Ihrer Applikation.

3.2.1 VARIABLEN

Im Tutorial 1 haben Sie direkt in der View-Funktion einen HTML-String zurückgegeben:

```
return '<h1>Hello, {}!</h1>'.format(name)
```

Mit einer Template-Datei hello.html würde das anders aussehen. Das folgende Template enthält im HTML-Code zwei Platzhalter-Variablen:

{{ title }} in Zeile 6 und

{{ name }}.in Zeile 9

```
1  <!-- Kap 3.2.1 app/templates/hello_user.html -->
2  <!doctype html>
3  <html>
4      <head>
5          <meta http-equiv="content-type" content="text/html; charset=utf-8">
6          <title>{{ title }}</title>
7      </head>
8      <body>
9          <h1>Hello, {{ name }} ! Willkommen im Kapitel 3.</h1>
10     </body>
11 </html>
```

Abbildung 3-1 Beispiel Template mit Platzhaltern

Die View-Funktion kann nun so geschrieben werden:

```
1 # kap-3.2.1 /app/routes.py
2 from flask import render_template
3 from app import app
4 @app.route('/')
5 @app.route('/index')
6 def index():
7     user = 'Jochen'
8     return render_template('hello_user.html', title='Home', name=user)
```

Abbildung 3-2 Beispiel View-Funktion mit render_template()

In Zeile 2 wird render_template importiert.

In Zeile 8 wird nun nicht mehr direkt ein HTML-String zurückgegeben, sondern das, was render_template() erzeugt. Diese Funktion bekommt den Dateinamen des Template, den Titel und den Namen übergeben. Das Ergebnis:

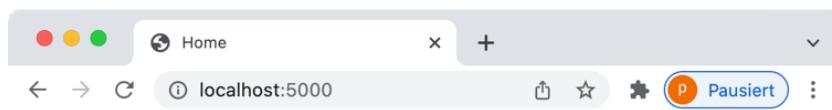


Abbildung 3-3 Anzeige im Browser mit Template

Im Template wird 'Home' anstelle des Platzhalters {{ title }} eingesetzt und 'Jochen' anstelle von {{ name }}. Es gibt nun kein HTML mehr in der View-Funktion.

Alles, was mit der Darstellung der Seite zu tun hat, ist nun in die Template-Datei ausgelagert.

Die Darstellung der Webseite kann nun in der Template-Datei verändert werden, ohne dass die View-Funktion dafür angepasst werden müsste.

3.2.2 IF IN TEMPLATES

Mit Jinja2 können bedingte Anweisungen verwendet werden. Das folgende Beispiel prüft mit { % if title %}, ob die Variable title oder die Variable name leer ist. Falls ja, wird dort ein Ersatz-String produziert. Anderfalls kommt der { % else %} Zweig zur Anwendung. Die Bedingung muss mit { % endif %} abgeschlossen werden.

```
1  <!-- Kap 3.2.2 app/templates/hello_if.html -->
2  <!doctype html>
3  <html>
4      <head>
5          {% if title %}
6              <title>{{ title }}</title>
7          {% else %}
8              <title>Startseite</title>
9          {% endif %}
10     </head>
11     <body>
12         {% if name %}
13             <h1>Hello, {{ name }}!</h1>
14         {% else %}
15             <h1>Hello, stranger!</h1>
16         {% endif %}
17     </body>
18 </html>
```

Abbildung 3-4 Ein Template mit Bedingungen

Ergebnis bei Aufruf von `render_template('hello_if.html', name = 'Jochen')` also ohne title:

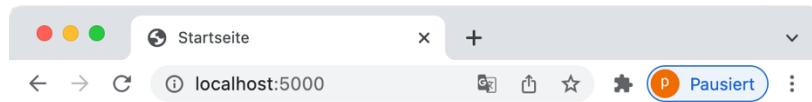


Abbildung 3-5 Ersatzwert für title wird angezeigt

Ergebnis bei `render_template('hello_if.html', title = 'Home')` also ohne name:

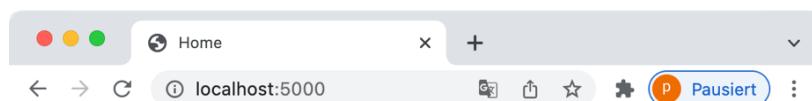


Abbildung 3-6 Ersatzwert für name wird angezeigt

3.2.3 SCHLEIFEN IN TEMPLATES

Eine weitere Kontrollstruktur beginnt mit { % for variable in iterierbares-Objekt % } und endet mit { % endfor % }. Sie verhält sich wie eine for-Schleife.

In der View-Funktion wird das Template benutzt. In der View-Funktion routes.py wird dafür eine neue Route /looptest eingefügt:

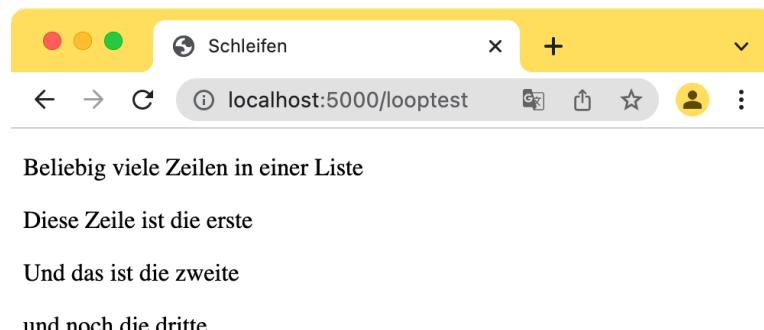
```
1  <!-- Kap 3.2.3 app/templates/hello_for.html -->
2  <!doctype html>
3  <html>
4      <head>
5          {% if title %}
6              <title>{{ title }}</title>
7          {% else %}
8              <title>Startseite</title>
9          {% endif %}
10     </head>
11     <body>
12         {% for line in lines %}
13             <div>
14                 <p>{{ line }} </p>
15             </div>
16         {% endfor %}
17     </body>
18 </html>
```

Abbildung 3-7 Template mit Schleife

```
13 @app.route('/looptest')
14 def looptest():
15     user = 'Jochen'
16     zeilen = ['Beliebig viele Zeilen in einer Liste',
17               'Diese Zeile ist die erste',
18               'Und das ist die zweite', 'und noch die dritte']
19     return render_template('hello_for.html', name=user,
20                           title='Schleifen', lines=zeilen)
```

Abbildung 3-8 Iterierbares Objekt wird an render_template übergeben

Das Ergebnis sieht so aus:



3.2.4 MAKROS

In Jinja2 ist das Äquivalent zu einer Funktion ein macro. In der Datei templates/macros.html wird ein macro definiert:

```
1 <!-- Kap 3.2.4 app/templates/macros.html -->
2 {% macro render_line(line) %}
3 |   <li>{{line}}</li>
4 {% endmacro %}
```

Abbildung 3-9 Definition eines macro

In der Datei templates/hello_macros.html wird die Datei mit dem macro in Zeile 4 mit *import* importiert und die Macros aus der Datei können anschliessend in Zeile 11 benutzt werden. Der Aufruf ist ähnlich zu einer Funktion.

```
1 <!-- Kap 3.2.3 app/templates/hello_macros.html -->
2 <!doctype html>
3 <html>
4   {% import 'macros.html' as macros %}
5   <head>
6     <title>{{ title }}</title>
7   </head>
8   <body>
9     <ul>
10       {% for line in lines %}
11         {{ macros.render_line(line) }}
12       {% endfor %}
13     </ul>
14   </body>
15 </html>
```

Abbildung 3-10 Benutzung eines macro

Für alle Zeilen in lines wird nun der HTML-Code aus dem macro render_line() verwendet.

Teile von Templates, die mehrfach gebraucht werden, können grundsätzlich in separate Dateien ausgelagert werden und mit *include* in andere Dateien importiert und verwendet werden:

```
{% include 'gemeinsam.html' %}
```

Import und include verhalten sich leicht unterschiedlich. Bei *import* verhält es sich ähnlich wie bei Modulen in Python, sie werden in einem Cache zwischengespeichert. Macros, die importiert wurden, haben keinen direkten Zugriff auf die Variablen im Kontext des Template, das den import gemacht hat. Variablen müssen wie bei einer Funktion als Argumente übergeben werden.

Makros, die mit *include* importiert wurden, werden nicht im Cache gespeichert. Sie können auf die Variablen im Kontext des Template, das sie inkludiert hat, direkt zugreifen.

3.2.5 TEMPLATES ERWEITERN

Ein anderer Mechanismus zur Wiederverwendung von Template-Code ist die Erweiterung eines Basis-Templates. Sie kennen einen sehr ähnlichen Mechanismus aus Python, bei dem eine Oberklasse durch Unterklassen erweitert werden kann und bei dem die Unterklassen Attribute und Methoden der Basisklasse erbt.

Im folgenden Beispiel werden zwei Templates, base.html und hello_derived.html gezeigt.

```
1 <!-- Kap 3.2.5 app/templates/base.html -->
2 <!doctype html>
3 <html>
4   <head>
5     {% block head %}
6       <meta http-equiv="content-type" content="text/html; charset=utf-8">
7       <title>{% block title %}{% endblock %} - Meine App</title>
8     {% endblock %}
9   </head>
10  <body>
11    {% block body %}
12    {% endblock %}
13  </body>
14 </html>
```

Abbildung 3-11 Das Basis-Template

Mit {% block name %} und {% endblock %} werden Bereiche auf der Seite festgelegt, die Inhalte aufnehmen können. In base.html werden blocks definiert, die in davon abgeleiteten Templates so übernommen oder überschrieben werden können.

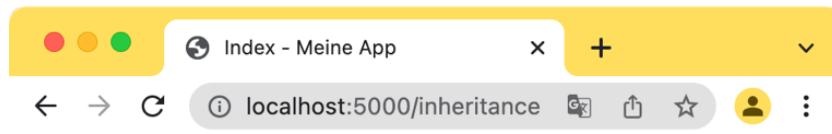
```
1 <!-- Kap 3.2.5 app/templates/hello_derived.html -->
2 {% extends "base.html" %}
3 {% block title %}Index{% endblock %}
4 {% block head %}
5   {{ super() }}
6   <style>
7   </style>
8 {% endblock %}
9 {% block body %}
10 <h1> Hello, World! </h1>
11 {% endblock %}
```

Abbildung 3-12 Das von base.htm abgeleitete Template

- Das Template hello_derived.html erweitert base.html. Dies geschieht mit *extends* in Zeile 2
- Darauf folgen drei Blöcke, die in base.html definiert werden und die an der jeweiligen Stelle in base.html eingesetzt werden.
- Wenn einer der Blöcke in beiden Templates HTML-Inhalt hat, hat der Inhalt von hello_derived.html Vorrang vor den Inhalt aus base.html

- Im Block head in hello_derived.html wird in Zeile 5 `super()` aufgerufen. Dadurch wird der Inhalt des Blocks head in base.html referenziert.

In der folgenden Abbildung sehen Sie die Wirkung:



Hello, World!

Abbildung 3-13 Wirkung bei abgeleitetem Template

- Der Inhalt 'Index' (aus Zeile 3 von hello_derived.html) wird mit dem Inhalt '- Meine App' (In Zeile 7 von base.html zusammengefügt im Titel des Browser Tabs angezeigt).
- Der body-Block in base.html hat keinen Inhalt. Dort wird der Inhalt 'Hello, World' aus dem body-Block von derived.html eingefügt.

Mit der Erweiterung per `extends` und mit `super()` können Sie Elemente, die auf allen Seiten erscheinen sollen, in einem Basistemplate einheitlich definieren und gestalten. Ändern sich die Einstellungen im Basis-Block werden die Änderungen auch in allen Seiten mit davon abgeleiteten Templates wirksam.

3.3 TUTORIAL MICROBLOG 1

Das folgende Tutorial und die folgenden Tutorials folgt dem 'Flask Megatutorial' von Miguel Grinberg. Darin wird schrittweise eine kleine Blog-App erstellt. Sie können damit:

- Sich selbst als User registrieren
- Sich danach als User anmelden und ein kleines Profil erstellen
- Kurze Beiträge erfassen, die auf der Startseite angezeigt werden
- Follower eines anderen Users werden
- Anderen Usern private Nachrichten senden

Dazu gibt es auch einen Online-Kurs¹ und ein E-book von Miguel Grinberg (Kosten für den Kurs und das Buch z.Zt. USD 39). Einige Kapitel sind auch als Serie von Blogbeiträgen frei verfügbar². Miguel stellt den Quellcode zum Tutorial auf GitHub frei zur Verfügung. Sie können sein Repository clonen und dort die Musterlösungen verfolgen und mit Ihrem eigenen Code vergleichen. Gehen Sie wie folgt vor:

¹ <https://courses.miguelgrinberg.com/p/flask-mega-tutorial>

² <https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world>

1. Erzeugen Sie ein Verzeichnis für Ihren eigenen Code. In den Beispielen verwenden wir dafür den Namen 'tutorial'. Wechseln Sie in das Verzeichnis und legen Sie dort ein neues virtuelles Environment an und installieren Sie dort Flask:

```
$ mkdir tutorial
$ cd tutorial
$ python3 -m venv tutorial           # python3 muss in Ihrem Pfad sein
$ tutorial\Scripts\activate          # In der cmd Shell auf Windows
# oder
$ source tutorial/bin/activate      # In der bash Shell auf Mac OS und Linux

(tutorial) $ pip install flask
```

2. Legen Sie folgende Verzeichnisstruktur an:

```
tutorial
└── app
    └── templates
```

3. Wir empfehlen Ihnen, dass Sie im neuen Verzeichnis tutorial den Code selbst Schritt für Schritt eingeben, nachvollziehen und allenfalls auch anpassen (Sie können natürlich auch einfach den Code von Miguel Grinberg ausführen, aber Ihr Lerneffekt wird geringer sein. So banal es klingt: Programmieren lernen geht durch die Finger!)

4. Initialisieren Sie Ihr Verzeichnis mit dem folgenden Kommando (im Ihrem neuen Verzeichnis):

```
$ git init
```

5. Es sollte nun ein Unterverzeichnis .git entstanden sein (Überprüfen Sie das)

6. Schreiben Sie zwei Template-Files im Verzeichnis app/templates

```
<!-- app/templates/base.html -->
<!doctype html>
<html>
    <head>
        {% if title %}
            <title>{{ title }} - Microblog</title>
        {% else %}
            <title>Welcome to Microblog</title>
        {% endif %}
    </head>
    <body>
        <div>Microblog: <a href="/index">Home</a></div>
        <hr>
        {% block content %}{% endblock %}
    </body>
</html>

<!-- app/templates/index.html -->
{% extends "base.html" %}
{% block content %}
    <h1>Hi, {{ user.username }}!</h1>
    {% for post in posts %}
        <div><p>{{ post.author.username }} says: <b>{{ post.body }}</b></p></div>
    {% endfor %}
{% endblock %}
```

7. Schreiben Sie die Datei app/__init__.py

```
# app/__init__.py
from flask import Flask
app = Flask(__name__)
from app import routes
```

8. Schreiben Sie eine Datei app/routes.py. Da Sie an diesem Punkt noch keine Datenbank mit echten Usern und Blog-Posts haben, enthält sie ein Dictionary mit Fake-Einträgen, damit Sie Ihre Templates testen können:

```
# app/routes.py
from flask import render_template
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Jochen'}
    posts = [
        {
            'author': {'username': 'Paul'},
            'body': 'Schöner Abend hier in Zürich!'
        },
        {
            'author': {'username': 'Susanne'},
            'body': 'Der Unterricht war heute mal gut!'
        }
    ]
    return render_template('index.html', title='Home', user=user, posts=posts)
```

9. Schreiben Sie tutorial/microblog.py

```
# tutorial/microblog.py
from app import app
```

10. Ihre Verzeichnisstruktur sollte nun so aussehen;

```
tutorial
└── app
    ├── __init__.py
    ├── routes.py
    └── templates
        ├── base.html
        └── index.html
└── microblog.py
└── tutorial      # Das venv-Verzeichnis mit den installierten Paketen
```

11. Setzen Sie die Variable FLASK_APP=microblog.py

12. Starten Sie den Flask Server mit **flask run**

13. Sehen Sie sich das Ergebnis mit <http://localhost:5000/index> im Browser an

3.4 MUSTERLÖSUNGEN ZU MICROBLOG

1. Für die Musterlösungen clonen Sie das Repository microblog in einem neuen, separaten Verzeichnis mit:

```
$ git clone https://github.com/miguelgrinberg/microblog
```

Dort, wo sie das Kommando ausgeführt haben, wurde ein Verzeichnis 'microblog' angelegt. Dieses Verzeichnis ist ein git-Repository, dass pro Kapitel des Tutorials einen commit enthält. Führen Sie das folgende Kommando in microblog aus:

```
$ git log
```

Wichtig: Benutzen Sie für Ihren Code nicht dieses Verzeichnis, sondern ein anderes! Das Verzeichnis microblog sollte nur lesend zum Vergleich mit den Musterlösungen benutzt werden.

2. Sie sehen eine lange Liste mit allen commits für microblog.

```
...
commit 3172e0955c59c8fbeaca6caa8889e1480213ea51 (tag: v0.3)
Author: Miguel Grinberg <miguel.grinberg@gmail.com>
Date:   Tue Sep 5 00:04:56 2017 -0700
```

Chapter 3: Web Forms (v0.3)

```
commit 5d4559227a7f215251ed1dc15f884443109e153 (tag: v0.2)
Author: Miguel Grinberg <miguel.grinberg@gmail.com>
Date:   Mon Sep 4 23:28:11 2017 -0700
```

Chapter 2: Templates (v0.2)

```
commit 23b3fc85d407f8c68c95efd8a29dd053467708c5 (tag: v0.1)
Author: Miguel Grinberg <miguel.grinberg@gmail.com>
Date:   Mon Sep 4 23:23:07 2017 -0700
```

Chapter 1: Hello, World! (v0.1)

```
commit e308d0d969fcf10ed9f86d5cba99063bb6b384d8 (tag: v0.0)
Author: Miguel Grinberg <miguel.grinberg@gmail.com>
Date:   Mon Sep 4 23:22:25 2017 -0700
```

Initial commit (v0.0)

Jeder Commit hat eine lange ID und einen Tag. Die ältesten Commits sind unten in der Liste

3. Benutzen Sie eines der folgenden Kommandos, um den zu diesem Kapitel passenden Stand auszuchecken:

```
$ git checkout 5d4559227a7f215251ed1dc15f884443109e153
```

oder

```
$ git checkout v0.2
```

4. Das Verzeichnis microblog hat jetzt einen anderen Inhalt als vorher, nämlich den zum Stand des Kapitels im Buch 'The Flask Megatutorial'
5. Wenn Sie zum ursprünglichen Stand des Verzeichnisinhalts zurückkehren wollen, benutzen Sie das Kommando

```
$ git switch -
```

3.5 ZUSAMMENFASSUNG

- Templates ermöglichen die Trennung von Geschäftslogik und Präsentation. Der HTML-Code kann mit Ihnen aus den View-Funktionen in separate .html Dateien ausgelagert werden.
- Jinja2 ist eine Template-Engine für Flask/Python
- Mit Jinja2-Templates können Platzhalter in den HTML-Code eingesetzt werden
- Mit Jinja2-Templates können Bedingungen dafür, was angezeigt wird formuliert werden
- Mit Jinja2-Templates können for-Schleifen erzeugt werden, mit denen Auflistungen erzeugt werden können
- Templates können Macros verwenden, die Python-Funktionen ähnlich sind
- Mit import und include können andere Template-Dateien eingefügt werden
- Blocks können Platzhalter für ganze Bereiche sein
- Mit extends können Sie ein bestehendes Template erweitern. Das erlaubt die einheitliche Gestaltung von mehreren Seiten mit einem zentralen Basis-Template.

4 FORMS

In diesem Kapitel erstellen Sie ein Webformular, das so aussehen soll:

The screenshot shows a web browser window with a yellow header bar. The title bar says 'Sign In - Microblog'. The address bar shows 'localhost:5000/login'. Below the header is a navigation bar with back, forward, and search icons. The main content area has a heading 'Microblog: [Home](#) [Login](#)'. Below this is a 'Sign In' section with the following fields:

- Username: An input field.
- Password: An input field.
- Remember Me: A checkbox.
-

Abbildung 4-1 Vorschau des Login-Formulars

4.1 LERNZIELE

Nach der Bearbeitung dieses Kapitel können Sie

- Die Erweiterung flask-wtf installieren
- Eingabefelder mit Templates und einer View-Funktion erzeugen
- Daten aus einer Form in der App verarbeiten
- Eingaben in eine Form validieren
- Links einfacher in die Weboberfläche einbetten

Das ganze Kapitel ist ein praktisches Tutorial mit eingebetteten Erklärungen. Es basiert auf dem tutorial im Kapitel 3.3. Die Beispiele in den nächsten Unterkapiteln sollten Sie also Schritt für Schritt ausführen.

4.2 VORBEREITUNGEN

WTForms ist ein Python-Package für die Erzeugung von Formularen sowie die Validierung von Eingaben in Formularen. Die Flask-Erweiterung flask-wtf ermöglicht es, WTForms auf einfache Weise in eine Flask-Applikation einzubinden. Sie wird im virtuellen Environment installiert (Es wird angenommen, dass Sie in einem Unterverzeichnis mit dem Namen 'tutorial' arbeiten und dort bereits im vorhergehenden Kapitel ein virtuelles Environment eingerichtet haben, das ebenfalls 'tutorial' heißt. Falls nicht, holen Sie das bitte nach.

- Ihre Verzeichnisstruktur sollte zu Anfang so aussehen (Falls nicht, gehen Sie bitte zu Kapitel 3.3 zurück):

```

tutorial
└── app
    ├── __init__.py
    ├── routes.py
    └── templates
        ├── base.html
        └── index.html
└── microblog.py
└── tutorial      # Das venv-Verzeichnis mit den installierten Paketen

```

- Gehen Sie in das (obere) Verzeichnis tutorial

```
$ cd tutorial
```

- Aktivieren sie das virtuelle Environment tutorial:

```
$ tutorial/Scripts/activate      # In der cmd Shell auf Windows
```

oder

```
$ source tutorial/bin/activate      # In der bash Shell auf Mac OS und Linux
```

```
(tutorial) $
```

- Hier installieren sie nun flask-wtf:

```
(tutorial) $ pip install flask-wtf
```

4.3 DAS CONFIG.PY FILE

Sobald eine Flask-App etwas umfangreicher wird ist es sinnvoll, Konfigurations-Details in einer Datei zu speichern.

Legen Sie ein Modul (eine neue Datei) config.py an:

```

tutorial
└── app
    ├── __init__.py
    ├── routes.py
    └── templates
        ├── base.html
        └── index.html
└── config.py
└── microblog.py

```

Erfassen Sie den folgenden Inhalt :

```

1 # tutorial/config.py
2 import os
3
4 class Config(object):
5     SECRET_KEY = os.environ.get('SECRET_KEY') or 'erraeatst-Du-nie'

```

- Zeile 2: Das Modul `os` wird importiert. Es ermöglicht Zugriffe auf das Betriebssystem.
- Zeile 4: Zum Speichern von Konfigurationsdetails wird eine Klasse `Config` verwendet.
- Zeile 5: Falls im Betriebssystem die Variable `SECRET_KEY` gesetzt ist, kann Ihr Inhalt mit der Methode `os.environ.get(Name-der-Variablen)` aus dem `os`-Modul abgefragt werden. Fals sie

gesetzt ist, wird Python-Variable SECRET_KEY der gleiche Wert zugewiesen. Falls nicht, wird ersatzweise der String 'erraetst-Du-nie' zugewiesen.

Die Variable SECRET_KEY wird in flask-wtf benutzt, um einen Angriff namens Cross Site Request Forgery (CSRF) zu verhindern. Bei einem CSRF-Angriff wird dem Browser des Opfers ohne dessen Wissen eine http-Anfrage untergeschoben³.

Ändern Sie app/__init__.py wie folgt:

```
1 # app/init.py
2 from flask import Flask
3 from config import Config
4
5 app = Flask(__name__)
6 app.config.from_object(Config)
7
8 from app import routes
```

- Zeile 3: Import der Klasse Config aus dem Modul config.py
- Zeile 6: dem app-Objekt wird die Konfiguration mit der eingebauten Methode app.config.from_object() hinzugefügt.

4.4 LOGIN FORM

Legen Sie ein neues Modul mit dem Namen forms.py im Paket app an:

```
tutorial
└── app
    ├── __init__.py
    ├── forms.py      <- Die neue Datei
    ├── routes.py
    └── templates
        └── base.html
            └── index.html
└── config.py
└── microblog.py
```

Erfassen Sie den folgenden Inhalt:

```
1 # app/forms.py
2 from flask_wtf import FlaskForm
3 from wtforms import StringField, PasswordField, BooleanField, SubmitField
4 from wtforms.validators import DataRequired
5
6 class LoginForm(FlaskForm):
7     username = StringField('Username', validators=[DataRequired()])
8     password = PasswordField('Password', validators=[DataRequired()])
9     remember_me = BooleanField('Remember Me')
10    submit = SubmitField('Sign In')
```

³ <https://de.wikipedia.org/wiki/Cross-Site-Request-Forgery>.

In flask_wtf werden Klassen benutzt, um Web-Formulare zu definieren.

- Zeile 2: Die Klasse FlaskForm wird aus dem paket flask_wtf importiert Zeile 3: Weitere Klassen für verschiedene Feldtypen werden aus wtforms importiert.
- Zeile 4: Die Klasse Die Klasse DataRequired wird aus wtforms.validators importiert
- Zeile 6: Eine eigene Klasse LoginForm wird von der vordefinierten Klasse FlaskForm abgeleitet. Sie bekommt die Klassen-Variablen username, password, remember_me und submit.
- Zeilen 7 bis 10: Das Paket wtforms bietet Klassen für verschiedene Feldtypen an. Jedes Feld-Objekt wird mit dem Konstruktor der jeweiligen Klasse erzeugt (StringField(), PasswordField(), BooleanField() und SubmitField()).
- Zeilen 7 und 8: Für das Felde username hat die Methode StringField() das Argument validators=[DataRequired]. Beim Feld password git dasselbe für die Funktion BooleanField(). DataRequired ist eine andere Klasse aus wtforms. Sie prüft, ob das Feld ausgefüllt wurde.

4.5 FORM TEMPLATE

Der nächste Schritt ist die Erstellung eines HTML-Template unter app/templates. Es bekommt den Namen login.html:



Die Felder der Klasse LoginForm aus dem letzten Abschnitt können bereits HTML anzeigen, so dass für das Rendering der Login-Seite nur folgender Inhalt in login.html stehen muss:

```
1  <!-- # app/templates/login.html -->
2  {% extends "base.html" %}
3
4  {% block content %}
5      <h1>Sign In</h1>
6      <form action="" method="post" novalidate>
7          {{ form.hidden_tag() }}
8          <p>
9              {{ form.username.label }}<br>
10             {{ form.username(size=32) }}<br>
11             {% for error in form.username.errors %}
12                 <span style="color: red;">{{ error }}</span>
13             {% endfor %}
14         </p>
15         <p>
16             {{ form.password.label }}<br>
17             {{ form.password(size=32) }}<br>
18             {% for error in form.password.errors %}
19                 <span style="color: red;">{{ error }}</span>
20             {% endfor %}
21         </p>
22         <p>{{ form.remember_me() }} {{ form.remember_me.label }}</p>
23         <p>{{ form.submit() }}</p>
24     </form>
25  {% endblock %}
```

- Zeile 2: Das Template erweitert base.html und übernimmt damit die Inhalte und Formatierungen, die dort schon definiert wurden.

- Zeilen 6-24: Der Abschnitt <form> bis </form> ist ein HTML-Formular. Die diversen Jinja2-Platzhalter mit 'form.' im Namen erwarten, dass es ein Objekt der Klasse LoginForm gibt, dass dem Template mit render_template() übergeben wurde.
- Zeile 6: Die Attribute von <form> bedeuten:
 - action="" : Es wird hier keine Aktion für den Browser angegeben
 - method="post": Die http-Methode POST soll verwendet werden, wenn der Benutzer den Inhalt des Formulars abschickt.
 - novalidate : Der Browser soll keine Validierung der Feldinhalte ausführen (Das soll die Flask-App selbst tun)
- Zeile 7: {{ form.hidden_tag() }} fügt ein verborgenes Feld ein, dass den SECRET_KEY enthalten soll. Damit wird das ganze Formular gegen eine CSRF-Attacke geschützt.
- Zeilen 8-21: Hier werden die Felder des LoginForm-Objekts gerendert
- Zeilen 11 - 13: Ein Abschnitt für Fehlermeldungen zum Username wird bereitgestellt
- Zeilen 18 – 20: Ein Abschnitt für Fehlermeldungen zum Passwort wird bereitgestellt
- Zeile 22: Das Feld für die 'Remember Me' Checkbox
- Zeile 23: Das Feld für den 'Sign in' Button

4.6 FORM VIEWS

Zur Seite /login müssen Sie nun noch eine View-Funktion schreiben, die das Template benutzt. Das geschieht in app/routes.py. Fügen Sie dort Folgendes hinzu:

```
33 @app.route('/login')
34 def login():
35     form = LoginForm()
36     return render_template('login.html', title='Sign In', form=form)
```

Um den URL localhost:5000/login zum Testen nicht manuell eingeben zu müssen, fügen sie der Datei app/templates/base.html aus dem Kapitel 3.3 noch zwei Links hinzu und zwar am Anfang des <body>:

```
12     <div>
13         Microblog:
14             <a href="{{ url_for('index') }}>Home</a>
15             <a href="{{ url_for('login') }}>Login</a>
16     </div>
```

Die Funktion url_for() liefert zu einer Funktion (hier 'index' oder 'login') den URL zurück. Das ist praktisch, denn wenn Sie die URLs ändern sollten, aber nicht den Namen der View-Funktion, funktioniert das Template weiterhin.

4.7 ERSTER TEST DER FORM

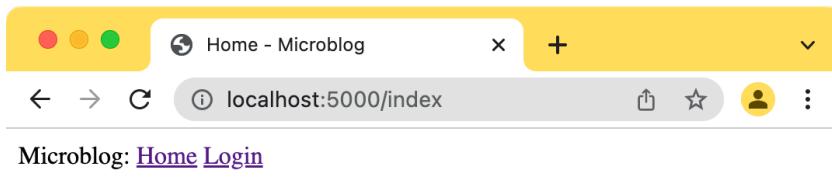
Sie können nun testen, ob das Formular richtig angezeigt wird und ob der Link auf die richtige Seite führt. Die Form ist noch nicht fertig, hier geht es zunächst nur um die Anzeige:

```
(tutorial) $ set FLASK_APP=microblog.py          # cmd / Windows  
oder  
(tutorial) $ export FLASK_APP=microblog.py        # bash / Mac OS und Linux
```

Starten Sie den Server:

```
(tutorial) $ flask run
```

Die App sollte nun so aussehen:



Paul says: **Schöner Abend hier in Zürich!**

Susanne says: **Der Unterricht war heute mal gut!**

Abbildung 4-2 Die Startseite mit Links

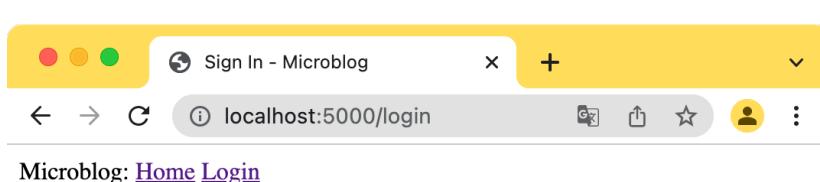


Abbildung 4-3 Die Login-Seite

Wenn Sie nun etwas eingeben und auf 'Sign in' klicken, bekommen Sie die Fehlermeldung 'Method not allowed'. Das liegt daran, dass Ihre View-Funktion noch nicht mit der Methode 'POST' umgehen kann, die hier zur Anwendung kommt.

4.8 DATEN AUS DER FORM AKZEPTIEREN

Falls die Forms nun korrekt angezeigt wird, muss nun noch festgelegt werden, was passieren soll, wenn etwas eingegeben und mit dem Button 'Sign in' an den Server geschickt wurde. Ändern Sie die Datei app/routes.py ab, so dass sie jetzt so aussieht:

```
1 # app/routes.py
2 from flask import render_template, flash, redirect, url_for
3 from app import app
4 from app.forms import LoginForm
5
6 @app.route('/')
7 @app.route('/index')
8 def index():
9     user = {'username': 'Jochen'}
10    posts = [
11        {
12            'author': {'username': 'Paul'},
13            'body': 'Schöner Abend hier in Zürich!'
14        },
15        {
16            'author': {'username': 'Susanne'},
17            'body': 'Der Unterricht war heute mal gut!'
18        }
19    ]
20    return render_template('index.html', title='Home',
21                           user=user, posts=posts)
22
23 @app.route('/login', methods=['GET', 'POST'])
24 def login():
25     form = LoginForm()
26     if form.validate_on_submit():
27         flash('Login requested for user {}, remember_me={}'.format(
28             form.username.data, form.remember_me.data))
29         return redirect(url_for('index'))
30     return render_template('login.html', title='Sign In', form=form)
```

- Zeile 2: neben render_template werden jetzt *flash*, *redirect* und *url_for* importiert.
- Zeile 23: Dem app.route Decorator wurde ein neues Argument 'methods=['GET','POST']' hinzugefügt. Ohne diese Angabe konnte die Funktion login() nur mit der Methode GET umgehen (Das ist der Default-Wert für den Parameter methods).
- Zeile 26: form.validate_on_submit() erledigt alle Feldvalidierungen, wenn der Browser den Form-Inhalt mit POST abschickt ('Log in'-Button wurde gedrückt). Sie sammelt die Daten aller Felder ein und prüft die dafür festgelegten Regeln (Siehe: Validator in app/routes.py). Sie gibt True zurück, wenn alle Prüfungen bestanden wurden.

- Zeile 27: flash() produziert eine Nachricht, die aber nicht automatisch angezeigt wird. Dazu muss ein entsprechendes Element im Template eingerichtet sein.
- Zeile 29: redirect(url_for('index')) schickt den Benutzer zurück auf die Startseite.

Im Moment haben Sie noch keine Möglichkeit, auf echte User und Passworte zuzugreifen, denn die App hat noch keine Möglichkeit, diese zu speichern (Das werden Sie im nächsten Kapitel, Datenbanken einrichten). Als Provisorium wird deshalb der eingegebene Inhalt mit der flash()-Nachricht auf der Startseite angezeigt. In app/templates/base.html wird in Zeile 18 ein Element eingerichtet, das flash-Nachrichten anzeigen kann. Damit steht es auch in allen von base.html abgeleiteten Templates zur Verfügung:

```

1  <!-- # app/templates/base.html -->
2  <!doctype html>
3  <html>
4      <head>
5          {% if title %}
6              <title>{{ title }} - Microblog</title>
7          {% else %}
8              <title>Welcome to Microblog</title>
9          {% endif %}
10     </head>
11     <body>
12         <div>
13             Microblog:
14             <a href="{{ url_for('index') }}>Home</a>
15             <a href="{{ url_for('login') }}>Login</a>
16         </div>
17         <hr>
18         {% with messages = get_flashed_messages() %}
19             {% if messages %}
20                 <ul>
21                     {% for message in messages %}
22                         <li>{{ message }}</li>
23                     {% endfor %}
24                 </ul>
25             {% endif %}
26             {% endwith %}
27             {% block content %}{% endblock %}
28         </body>
29     </html>
```

Die Fehlermeldungen, die angezeigt werden, falls das User- oder das Passwort leer sind, werden durch die for-Schleife in Zeile 18-20 von app/templates/login.html angezeigt:

```

18     {% for error in form.password.errors %}
19         <span style="color: red;">[{{ error }}]</span>
20     {% endfor %}
```

Hier sehen Sie die provisorische Ausgabe der Benutzereingaben:

The screenshot shows a web browser window with a yellow title bar. The title is "Sign In - Microblog". The address bar shows "localhost:5000/login". Below the title bar, there are standard browser controls: back, forward, refresh, and a search bar with a magnifying glass icon. To the right of the search bar are icons for bookmarking, a user profile, and more. The main content area has a header "Microblog: [Home](#) [Login](#)". Below this is a large bold heading "Sign In". There are two input fields: one for "Username" containing "jochen" and another for "Password" containing a series of dots. A checkbox labeled "Remember Me" is followed by a checked state indicator. At the bottom is a "Sign In" button.

Abbildung 4-5 Eingabe in Login-Form

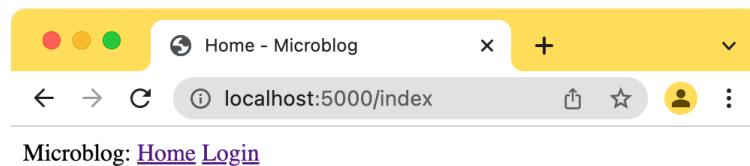


Abbildung 4-4 Anzeige der Eingabe als flash-Message

4.8.1 MUSTERLÖSUNGEN

Die Musterlösungen zu diesem Kapitel sind Bestandteil des Repository

<https://github.com/miguelgrinberg/microblog>

Wenn Sie es wie im Kapitel beschrieben gclont haben, geben Sie im Verzeichnis microblog das folgende Kommando ein:

```
$ git checkout v0.3
```

4.9 HINTERGRUND

Dieses Kapitel enthält Zusatzinformationen, die zum tieferen Verständnis des praktischen Beispiels beitragen können. Für die weiteren Kapitel dieses Kurses ist die Lektüre aber nicht zwingend nötig.

Die im Tutorial verwendeten Klassen stammen aus dem Paketen wtforms und flask-wtf macht sie in Flask verfügbar:



Abbildung 4-6 Die Erweiterung flask-wtf

Am Beispiel dieser Klassen sollen an dieser Stelle noch einmal die Konzepte Oberklasse, Unterklasse, Vererbung und Klassenhierarchie erklärt werden.

4.9.1 DIE FORM KLASSE

Die Klasse LoginForm aus app/forms.py wurde von der Klasse FlaskForm abgeleitet:

```
from flask_wtf import FlaskForm
class LoginForm(FlaskForm):
    # Code der Klasse ...
```

FlaskForm stammt aus flask-wtf und ist selbst wieder von der Klasse Form aus dem Paket wtforms abgeleitet:

```
class FlaskForm(Form):
    """Flask-specific subclass of WTForms :class:`wtforms.form.Form` ..."""
    # Code der Klasse ...
```

Die Klasse Form ist wiederum von einer anderen Klasse in wtforms abgeleitet. Diese heisst BaseForm. Sie sehen im Diagramm eine Hierarchie von Klassen, bei der BaseForm die oberste Ebene darstellt.

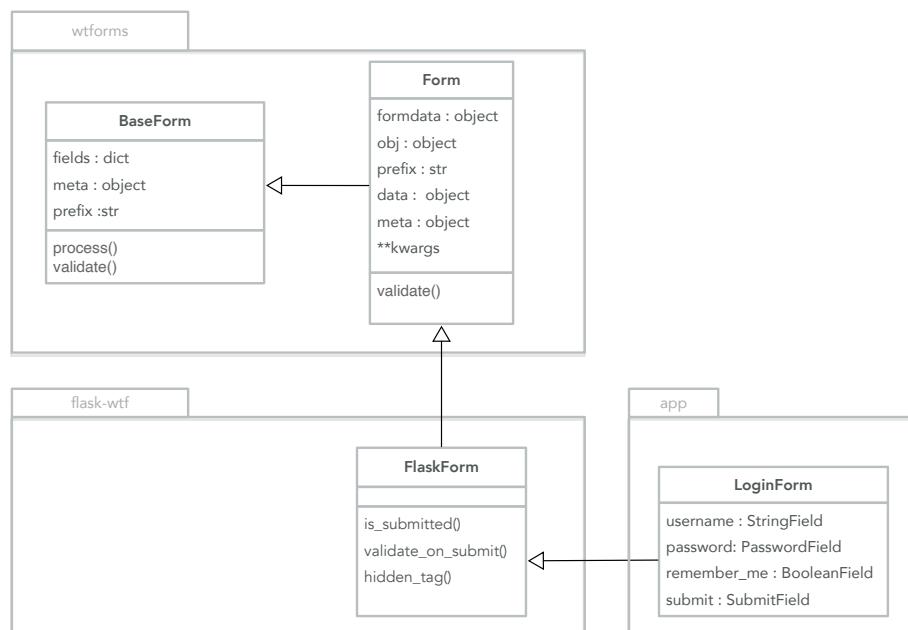


Abbildung 4-7 Klassenhierarchie für Forms

Die Beziehung im Diagramm bedeutet 'Ist eine'. Also: LoginForm ist eine FlaskForm, FlaskForm ist eine Form und Form ist eine BaseForm. Bei dieser Art der Beziehung gilt:

- Die abgeleitete Klasse (die *Unterklasse*) *erbt* alle Attribute und Methoden der Klasse, von der sie abgeleitet wurde (die *Oberklasse*). Objekte der Unterklasse haben und können alles, was die Oberklasse hat und kann.
- Die Unterklasse kann der Oberklasse weitere Attribute und Methoden hinzufügen.
- Die Unterklasse kann Methoden der Oberklasse überschreiben und damit diesen Methoden spezielle Fähigkeiten verleihen, die für Objekte der Unterklasse nützlich sind.

Für diese Art der Beziehung wird häufig der Begriff *Vererbung* benutzt.

Für das konkrete Beispiel bedeutet das:

LoginForm hat neben den vier dort definierten Feldern noch die Attribute:

- formdata aus Form
- data aus Form
- die Attribute aus BaseForm

LoginForm hat zusätzlich die Methoden:

- issubmitted() aus FlaskForm
- validate_on_submit aus FlaskForm
- hidden_tag () aus FlaskForm
- validate() aus Form
- process() aus BaseForm

Es ist für Sie als Entwickler nicht notwendig, die Details der Klassen Form und BaseForm zu kennen, wenn sie lediglich FlaskForm benutzen wollen. Sie brauchen dazu nur den import von flask-wtf die Dokumentation der Klasse FlaskForm⁴.

Das Beispiel ist aber nützlich, um Klassenhierarchien mit Vererbung zu erklären. Vererbung ist eine Art, Code wiederzuverwenden und möglicherweise wollen Sie in Ihrem eigenen Code diese Möglichkeit verwenden. Es gibt aber noch andere Arten der Wiederverwendung, die in manchen Fällen besser geeignet ist

⁴ <https://flask-wtf.readthedocs.io/en/1.0.x/>

4.9.2 DIE FIELD-KLASSEN

Neben FlaskForm wurden in diesem Kapitel noch Klassen aus dem Paket wtforms importiert und benutzt:

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import ValidationError, DataRequired, Email, EqualTo
from app.models import User

class LoginForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    password = PasswordField('Password', validators=[DataRequired()])
    remember_me = BooleanField('Remember Me')
    submit = SubmitField('Sign In')
```

Die Feld-Objekte im Paket app sind Instanzen von Klassen aus dem Paket wtforms. Sie bilden wie die Form-Klassen eine Hierarchie.

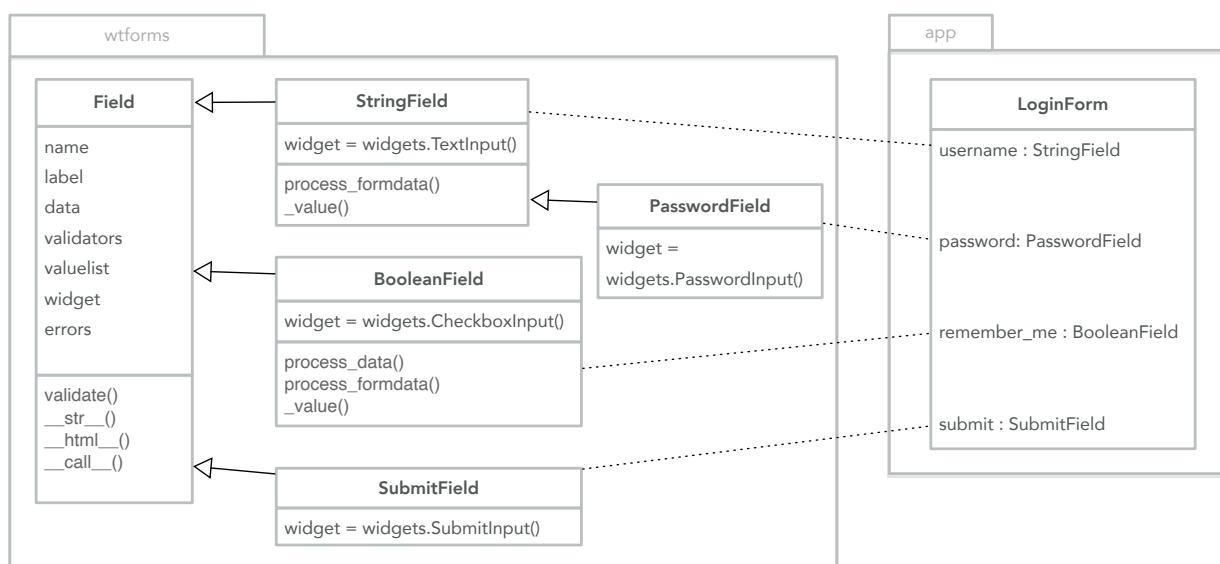


Abbildung 4-8 Klassenhierarchie im Paket wtforms

Sie haben aber von Alldem nur die folgenden Klassen benötigt:

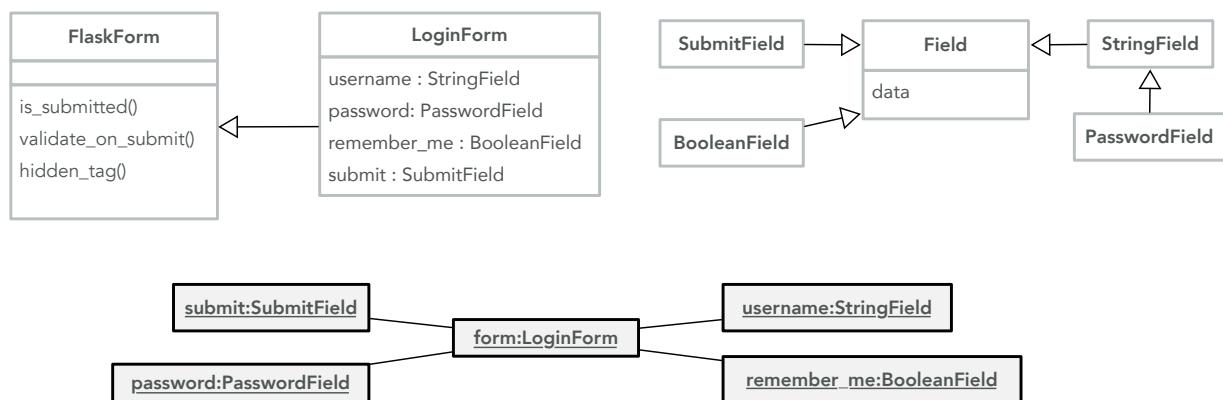


Abbildung 4-9 Klassen und Objekte aus dem Beispiel

Generell gilt: Wenn Sie nur vordefinierte Klassen verwenden wollen, brauchen Sie nicht zu wissen, wie diese implementiert wurden. Sie müssen nur wissen, welche Klassen Sie brauchen und welche Importe Sie machen müssen. Diese Informationen finden Sie bei Bedarf in der Dokumentation von wtforms⁵.

4.10 ZUSAMMENFASSUNG

- HTML-Forms können mit wtforms und dem Package flask-forms in Flask Apps integriert werden.
- Forms sind Klassen, die von der Klasse FlaskForm aus flask_wtf abgeleitet werden
- Für verschiedene Arten von Feldern gibt es vordefinierte Klassen, die wtforms zur Verfügung stellt. Sie können mit vordefinierten validator-Klassen verknüpft sein, die Benutzereingaben prüfen.
- Für eine Form muss ein eigenes Template geschrieben werden, das die Form-Klasse und Ihre Felder referenziert.
- Die Reaktion auf einen POST (Benutzer schickt die Form-Daten im Browser ab) wird in der View-Funktion in routes.py festgelegt.
- Der Decorator @app.route kann über einen Parameter auch mit POST umgehen (Default ist nur GET)
- Die Methode validate_on_submit() erledigt alle Feldvalidierungen
- Die Funktion url_for(Funktions-Name) liefert die URL zu einer View-Funktion. Das Ergebnis kann in für die dynamische Verwendung von Links in einem Template benutzt werden.
- Mit flash() können Messages erzeugt werden, die in einem geeigneten Bereich eines Template angezeigt werden kann. Den haben Sie in base.html eingebaut und damit 'erben' die anderen Templates diesen Bereich.

⁵ <https://wtforms.readthedocs.io/en/3.0.x/>

5 DATENBANKEN

Im vorhergehenden Kapitel konnten Sie noch nicht auf echte User und ihre Passwörter zugreifen, da diese noch nirgends gespeichert sind und Sie haben einen Workaround benutzt um die Eingaben eines Benutzers anzuzeigen.

Für die Startseite benutzen Sie seit dem vorletzten Kapitel 'Fake'-Daten für die Blog Posts:

```
9     user = {'username': 'Jochen'}
10    posts = [
11        {
12            'author': {'username': 'Paul'},
13            'body': 'Schöner Abend hier in Zürich!'
14        },
15        {
16            'author': {'username': 'Susanne'},
17            'body': 'Der Unterricht war heute mal gut!'
18        }
19    ]
```

Das soll sich ab diesem Kapitel ändern.

5.1 LERNZIELE

Nach der Bearbeitung des Kapitels können Sie:

- Den Begriff 'Object Relational Mapping' (ORM) erklären
- Flask mit einem Datenbanksystem verbinden
- Eine Datenbank für User und Blog Posts erzeugen
- Mit der Erweiterung Flask-SQLAlchemy Datenbanken anlegen und Objekte in einem relationalen Datenbanksystem speichern, ändern und lesen.
- Änderungen an der Datenstruktur in Flask mit der Erweiterung flask-migrate auf die DB synchronisieren

5.2 OBJEKTE UND TABELLEN

Die folgende Abbildung zeigt, was bedacht sein muss, wenn eine Flask App die Daten ihrer Objekte in einer relationalen Datenbank speichern soll:

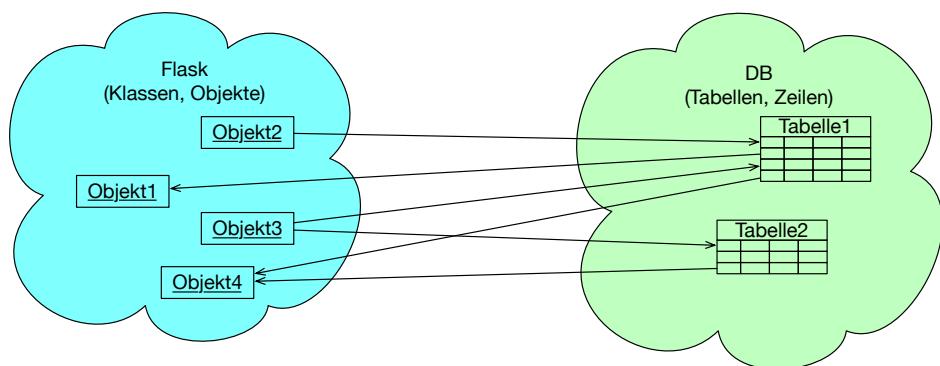


Abbildung 5-1 Objekt-Welt und Tabellen-Welt

Objekte haben Attribut-Werte, also Daten. Tabellen haben Spalten und Zeilen. Manchmal ist es möglich, alle Daten eines Objekts in einer Zeile einer Tabelle zu speichern oder aus den Daten in einer Zeile ein Objekt zu erzeugen.

Es gibt aber auch komplexe Objekte, deren Daten nicht in einer einzigen Tabelle gespeichert werden können oder die man nicht aus einer einzigen Zeile einer einzelnen Tabelle erzeugen kann.

Die Beziehungen zwischen Objekten müssen so in der Datenbank gespeichert werden, dass sie auch nach einem Neustart der App wieder so sind, wie sie beim Beenden waren.

5.3 VERBINDUNG ZU EINEM DATENBANKSYSTEM

Im Python-Teil haben Sie gesehen, wie sie zu einem bestimmten Datenbanksystem Verbindung aufnehmen können. Dieses Verfahren könnten Sie auch in einer Flask-App verwenden, aber:

- Es hat die Installation eines Treiber-Moduls erfordert. Dieser Treiber war nur für ein spezielles DBMS wie z.B. MySQL geeignet. Wenn Sie beispielsweise von MySQL auf PostgreSQL wechseln wollen, müssen Sie mindestens die Konfiguration, meistens aber auch den Code der App anpassen.
- Der Umgang mit eingebetteten SQL-Anweisungen ist relativ umständlich.
- In Flask werden Klassen und Objekte verwendet und Klassen folgen einem anderen Denkansatz als Datenbank-Tabellen: Objekte werden nicht mit Schlüsseln verknüpft, gegenseitige Beziehungen sind im Tabellen-Modell nicht sinnvoll und die Beziehung 'is a' bei der Vererbung kann in SQL nicht gut umgesetzt werden.
- Wenn sich das Datenmodell in der Flask-App ändert, muss es manuell auch in der Datenbank geändert werden.

Flask mit einer Datenbank zu verbinden kann also eine ziemlich komplexe Angelegenheit sein.

Glücklicherweise gibt es eine einfachere Alternative. Es gibt diverse Produkte, die eine Brücke zwischen der Welt der Objekte und der Welt der Tabellen bauen. Man spricht dabei von 'Object Relational Mapping', kurz ORM.

Das ORM-Produkt, dass in diesem Kapitel vorgestellt wird, heisst SQLAlchemy.

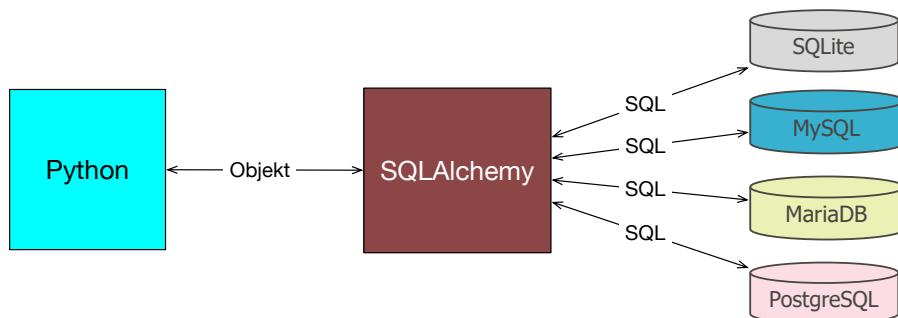


Abbildung 5-2 SQLAlchemy vermittelt zwischen der Objekt- und der Tabellen-Welt

SQLAlchemy ist freie Software und kann mit der Erweiterung flask-sqlalchemy installiert werden. Stellen Sie sicher, dass Ihr virtuelles Environment für Ihr eigenes Tutorial-Projekt aktiviert ist und führen Sie darin das folgende Kommando aus:

```
(tutorial) $ pip install flask-sqlalchemy
```

Die zweite Erweiterung ist flask-migrate. Sie stellt die Funktionen des Tools Alembic für Ihre Flask-App zur Verfügung. Damit können Sie später Änderungen an Ihrem Datenmodell aus Flask in die Datenbank übertragen. Installieren sie flask-migrate in Ihrem virtuellen Environment mit:

```
(tutorial) $ pip install flask-migrate
```

Für die weitere Entwicklung Ihrer App haben Sie zwei Möglichkeiten:

1. Sie benutzen SQLite, ein kleines DBMS, das mit Python bereits vorinstalliert ist. Für die lokale Entwicklung reicht das aus, aber für den Produktionsbetrieb nicht.
2. Sie benutzen MySQL, MariaDB oder ein anderes relationales DBMS Ihrer Wahl. Eine Kurzanleitung für MySQL finden Sie im Abschnitt 5.10.3 Hintergrund weiter unten.

Hier wird zunächst die Benutzung von SQLite gezeigt. Der erste Schritt ist, Flask mit dem DBMS bekannt zu machen. Erweitern Sie die Klasse Config in tutorial/config.py um zwei neue Variablen:

```
1 # tutorial/config.py
2 import os
3 basedir = os.path.abspath(os.path.dirname(__file__))
4
5 class Config(object):
6     SECRET_KEY = os.environ.get('SECRET_KEY') or 'erraegst-Du-nie'
7     SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
8     |   |   |   | 'sqlite:///+' + os.path.join(basedir, 'app.db')
9     SQLALCHEMY_TRACK_MODIFICATIONS = False
```

- Zeile 2 und 3: Das Modul os aus der Standardbibliothek wird importiert. Die Methode os.path.abspath() liefert einen Dateipfad. Ihr Argument ist das, was os.path.dirname(__file__) zurückgibt. Das ist das Verzeichnis, in dem die Datei config.py steht
- Zeile 5+6: Die Variable SQLALCHEMY_DATABASE_URI wird auf den Wert in der Umgebungsvariable DATABASE_URL gesetzt, falls diese existiert. Falls nicht, wird sie auf den URI gesetzt, der auf die lokale SQLite Datenbank im Projekt verweist.
- SQLALCHEMY_TRACK_MODIFICATIONS wird auf False gesetzt. Wäre der Wert True, würde SQLAlchemy jedes Mal, wenn sich die Datenbank ändert ein Signal an die Flask-App senden. Das wird in diesem Kapitel nicht gebraucht.

Als nächstes müssen Sie neue Einträge in app/__init__.py machen:

```
1 from flask import Flask
2 from flask_sqlalchemy import SQLAlchemy
3 from flask_migrate import Migrate
4 from config import Config
5
6 app = Flask(__name__)
7 app.config.from_object(Config)
8 db = SQLAlchemy(app)
9 migrate = Migrate(app, db)
10
11 from app import routes, models
```

- Zeile 2 : Aus dem Paket flask_sqlalchemy wird die Klasse SQLAlchemy importiert
- Zeile 3 : Aus dem Paket flask_migrate wird die Klasse Migrate importiert
- Zeile 4: Aus dem soeben von Ihnen erstellten Modul config.py wird Config importiert
- Zeile 7 : Die Configuration wird aus der Config-Klasse eingelesen
- Zeile 8 : Ein Objekt der Klasse SQLAlchemy wird erzeugt. Die Variable db repräsentiert jetzt die Datenbank-Engine
- Zeile 9 : Ein Objekt der Klasse Migrate wird erzeugt. Die Variable migrate repräsentiert jetzt die Migrations-Engine
- Zeile 11: Aus app wird zusätzlich models importiert

5.4 DATENMODELL

Das Datenmodell für diese Version der App Kapitel soll so aussehen:

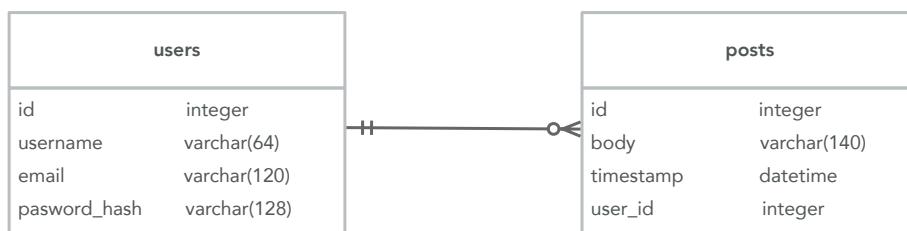


Abbildung 5-3 Das Datenmodell

Sie wissen bereits aus dem SQL-Teil, wie Sie dieses kleine Datenbankschema mit SQL erzeugen können. Hier sollen Sie es aber mit flask-sqlalchemy und flask-migrate tun.

Der grundsätzliche Ablauf der nächsten Schritte ist:

1. Erzeugung einer Model-Klasse in app/models.py
2. Vorbereitung einer sogenannten Migration. Damit ist die Erzeugung einer Datenbank-Tabelle auf Basis der Model-Klasse gemeint. Als erstes muss die Datenbank mit `flask init db` initialisiert werden. Dabei passiert folgendes:
 3. Neue Verzeichnisse unter dem Projektverzeichnis/migrations werden angelegt

4. Einige Konfigurationsdateien und Scripts werden angelegt
5. Anschliessend wird die erste Migration gestartet, bei der die Tabelle users in der Datenbank angelegt.

Legen Sie zunächst ein neues Modul unter app an. Es soll models.py heissen:

```

1 # app/models.py
2 ~ from datetime import datetime
3   from app import db
4
5 ~ class User(db.Model):
6     id = db.Column(db.Integer, primary_key=True)
7     username = db.Column(db.String(64), index=True, unique=True)
8     email = db.Column(db.String(120), index=True, unique=True)
9     password_hash = db.Column(db.String(128))
10
11 ~ def __repr__(self):
12     return '<User {}>'.format(self.username)

```

- Zeile 3: Aus dem Paket app wird db importiert (db wurde in __init__.py erzeugt)
- Zeile 5: Die Klasse User wird von db.Model abgeleitet
- Zeilen 6-9: Die Klassenvariablen von User entsprechen den Spalten der Tabelle users aus dem Datenmodell. Sie werden als Objekte der Klasse db.Column() erzeugt.
- Die id, der username und die email müssen eindeutig sein. Die id soll der Primärschlüssel in der Datenbanktabelle sein
- Zeile 11: Die Methode __repr__ ist nützlich für die Ausgabe von User-Objekten in der REPL

5.5 DATENBANK-MIGRATIONEN VORBEREITEN

Für die Klasse User aus dem Modul models.py soll nun eine Datenbanktabelle angelegt werden.

Zunächst wird eine SQLite Datenbank angelegt. Dabei sind Sie schon im virtuellen Environment (tutorial) und die Umgebungsvariable FLASK_APP hat den Inhalt 'microblog.py'

```
(tutorial) $ flask db init
  Creating directory ....tutorial/migrations/versions ... done
  Generating ....tutorial/migrations/script.py.mako ... done
  Generating ....tutorial/migrations/env.py ... done
  Generating ....tutorial/migrations/README ... done
  Generating ....tutorial/migrations/alembic.ini ... done
  Please edit configuration/connection/logging settings in
'....tutorial/migrations/alembic.ini' before proceeding.
```

Es entsteht ein neues Unterverzeichnis *migrations*, das teilweise schon Inhalte hat. Dort gibt es noch ein Unterverzeichnis *versions*, das zunächst leer ist.

```
tutorial
├── app
├── config.py
└── microblog.py
└── migrations
    ├── README
    ├── alembic.ini
    ├── env.py
    └── script.py.mako
└── versions
```

5.6 MIGRATION DER KLASSE USER

Im nächsten Schritt wird automatisch ein Migrations-Script auf Basis des Inhalts von models.py und config.py erzeugt:

```
(tutorial) $ flask db migrate -m "users table"
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'user'
INFO  [alembic.autogenerate.compare] Detected added index 'ix_user_email' on
  '['email']'
INFO  [alembic.autogenerate.compare] Detected added index 'ix_user_username' on
  '['username']'
  Generating .../migrations/versions/91726de3c32f_users_table.py ... done
```

- Das Kommando hat noch keine Datenbank und keine Tabelle angelegt, sondern nur das Script unter migrations/versions.
- Die ID 91726de3c32f im Dateinamen wird automatisch erzeugt und identifiziert diese Migration eindeutig (Bei Ihnen wird es eine andere ID sein).
- Sehen Sie Sich das neu erzeugte Script an. Es enthält zwei Funktionen, *upgrade()* und *downgrade()*.
- Mit *upgrade()* können Sie eine Änderung an der Datenbank durchzuführen
- Mit *downgrade()* können Sie die Änderung wieder zurücknehmen

Mit diesen Funktionen kann das ORM-Tool Alembic jeden beliebigen Zustand der Datenbank im Laufe der Zeit erzeugen oder wiederherstellen.

Die Funktion upgrade wird nun mit dem folgenden Kommando ausgeführt:

```
(tutorial) $ flask db upgrade
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade  -> 91726de3c32f, users table
```

Das legt die Datenbank an. Sie haben in config.py definiert, dass SQLite benutzt werden soll (Wenn in der Umgebungsvariablen SQLALCHEMY_DATABASE_URI nichts anderes steht). Es wird eine neue Datei app.db im Projektverzeichnis angelegt. Dort liegen die Tabellen von SQLite. Ihr Projektverzeichnis sollte nun folgende Struktur haben:

```
tutorial
├── app
│   ├── __init__.py
│   ├── forms.py
│   ├── models.py
│   ├── routes.py
│   └── templates (Unterverzeichnisse ausgeblendet)
├── app.db
├── config.py
├── microblog.py
└── migrations
    ├── alembic.ini
    ├── env.py
    ├── script.py.mako
    └── versions
        └── 91726de3c32f_users_table.py
```

Sie können sich testweise bei SQLite anmelden und sich mit dem Kommando `.schema` überzeugen, dass die Tabelle angelegt wurde (Unter Windows müssen Sie dafür sqlite.exe⁶ benutzen):

```
(tutorial) $ sqlite3 app.db
sqlite> .schema
CREATE TABLE alembic_version (
    version_num VARCHAR(32) NOT NULL,
    CONSTRAINT alembic_version_pk PRIMARY KEY (version_num)
);
CREATE TABLE user (
    id INTEGER NOT NULL,
    username VARCHAR(64),
    email VARCHAR(120),
    password_hash VARCHAR(128),
    PRIMARY KEY (id)
);
CREATE UNIQUE INDEX ix_user_email ON user (email);
CREATE UNIQUE INDEX ix_user_username ON user (username);
sqlite>
```

Mit CTRL-D können Sie sqlite3 wieder verlassen.

⁶ Download: <https://www.sqlite.org/download.html> : sqlite-tools-win32-x86-3380100.zip

5.7 POSTS

Nun muss noch die Datenbank für Blog-Posts konfiguriert werden. Dafür wird in models.py eine neue Klasse definiert:

```
1 # app/models.py
2 from datetime import datetime
3 from app import db
4 class User(db.Model):
5     id = db.Column(db.Integer, primary_key=True)
6     username = db.Column(db.String(64), index=True, unique=True)
7     email = db.Column(db.String(120), index=True, unique=True)
8     password_hash = db.Column(db.String(128))
9     posts = db.relationship('Post', backref='author', lazy='dynamic')
10
11    def __repr__(self):
12        return '<User {}>'.format(self.username)
13
14 class Post(db.Model):
15    id = db.Column(db.Integer, primary_key=True)
16    body = db.Column(db.String(140))
17    timestamp = db.Column(db.DateTime, index=True, default=datetime.utcnow)
18    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))
19
20    def __repr__(self):
21        return '<Post {}>'.format(self.body)
```

- Die Zeilen 1 bis 8 und 11 bis 12 hatten Sie schon im bisherigen Verlauf angelegt.
- In den Zeilen 14 bis 21 implementieren Sie die Klasse Post. Das ist die Grundlage für eine Tabelle posts.
- Zeile 18: Hier wird die Beziehung zwischen Post und User definiert. Aus dem SQL-Teil wissen sie bereits, dass dies in einer Datenbank durch einen Fremdschlüssel in der Tabelle posts geschieht. Das wird in Zeile 18 festgelegt.
- Die Zeile 9 in der Klasse User ist neu. Die Methode db.relationship() richtet die Möglichkeit ein, alle Posts eines Users zu suchen. In einer relationalen Datenbank müssten Sie dazu per SELECT die Zeilen der Tabelle posts durchsuchen und die Ergebnisse mit JOIN mit der Tabelle users verbinden, die als Fremdschlüssel die ID eines bestimmten Users enthalten.
- In der Objekt-Welt können Sie auch umgekehrt vorgehen: Über den Einstiegspunkt User alle Posts finden, die dieser gemacht hat. Dies wird über die Angabe backref ermöglicht.

Nun kommt die nächste Migration. Zunächst wird das Migrations-Script erzeugt:

```
(tutorial) $ flask db migrate
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'post'
INFO  [alembic.autogenerate.compare] Detected added index 'ix_post_timestamp' on
  '['timestamp']'
  Generating .../tutorial/migrations/versions/9e5e15cdc72e_posts_table.py ...
done
```

Danach wird die Migration ausgeführt:

```
(tutorial) $ flask db upgrade
INFO [alembic.runtime.migration] Context impl SQLiteImpl.
INFO [alembic.runtime.migration] Will assume non-transactional DDL.
INFO [alembic.runtime.migration] Running upgrade 91726de3c32f -> 9e5e15cdc72e,
posts table
```

In SQLite sollte jetzt die neue Tabelle posts erscheinen:

```
(tutorial) $ sqlite3 app.db

sqlite> .schema
CREATE TABLE alembic_version (
    version_num VARCHAR(32) NOT NULL,
    CONSTRAINT alembic_version_pkc PRIMARY KEY (version_num)
);
CREATE TABLE user (
    id INTEGER NOT NULL,
    username VARCHAR(64),
    email VARCHAR(120),
    password_hash VARCHAR(128),
    PRIMARY KEY (id)
);
CREATE UNIQUE INDEX ix_user_email ON user (email);
CREATE UNIQUE INDEX ix_user_username ON user (username);
CREATE TABLE post (
    id INTEGER NOT NULL,
    body VARCHAR(140),
    timestamp DATETIME,
    user_id INTEGER,
    PRIMARY KEY (id),
    FOREIGN KEY(user_id) REFERENCES user (id)
);
CREATE INDEX ix_post_timestamp ON post (timestamp);
```

Verlassen Sie SQLite mit CTRL-D

5.8 TEST DER DATENBANK IN DER REPL

Die Datenbank ist angelegt, aber Ihre App ist noch nicht so eingerichtet, dass sie die Datenbank nutzen kann. Einen ersten Test, ob Flask mit der Datenbank zusammenarbeitet, können Sie aber schon in der REPL machen. Stellen Sie sicher, das ihr virtuelles Environment aktiv ist und starten Sie die REPL. Zunächst importieren Sie die nötigen Definitionen:

```
>>> from app import db  
>>> from app.models import User, Post
```

Mit diesen Anweisungen legen Sie das erste User-Objekt an und speichern es in der Datenbank:

```
>>> u = User(username='jochen', email='jochen122@example.com')  
>>> db.session.add(u)  
>>> db.session.commit()  
>>> users = User.query.all()
```

Alle Operationen mit der Datenbank werden mit einem db.session-Objekt ausgeführt. Hinter db.session.add(Objekt) steht ein INSERT in die Datenbank. Alle Änderungen müssen zusätzlich mit db.session.commit() bestätigt werden, ansonsten würde der Eintrag nicht gespeichert werden.

Fügen Sie ein weiteres User-Objekt ein:

```
>>> u = User(username='paula', email='paula90@example.com')  
>>> db.session.add(u)  
>>> db.session.commit()
```

Eine Abfrage aller User-Objekte können Sie mit der Methode User.query.all() ausführen. Sie gibt eine Liste der gefundenen Objekte zurück:

```
>>> users = User.query.all()  
>>> users  
[<User jochen>, <User paula>]  
>>> for u in users:  
...     print(u.id, u.username)  
...  
1 jochen  
2 paula
```

Einen einzelnen User können Sie auch mit seiner id abfragen:

```
>>> u = User.query.get(1)  
>>> u  
<User jochen>  
>>>
```

Blog-Posts können Sie auch der Datenbank speichern. Für die Beziehung zum User bzw. Autor benutzen sie dabei das vorher aus der Datenbank gelesene User-Objekt:

```
>>> u = User.query.get(1)  
  
>>> p = Post(body='my first post!', author=u)  
>>> p = Post(body='Das scheint tatsächlich zu funktionieren!', author=u)  
>>> db.session.add(p)  
>>> db.session.commit()  
  
>>> p = Post(body='Das ist mein zweiter Post', author=u)  
>>> db.session.add(p)  
>>> db.session.commit()
```

Die Posts lassen sich wie folgt abfragen:

```
>>> u = User.query.get(1)
>>> u
<User jochen>

>>> posts = u.posts.all()
>>> posts
[<Post Das scheint tatsächlich zu funktionieren!>,
 <Post Das ist mein zweiter Post>]
```

Erfassen Sie einen Post für den zweiten User Account:

```
>>> u = User.query.get(2)
>>> u
<User paula>

>>> p = Post(body='Dann mache ich auch mal einen Post!', author=u)
>>> db.session.add(p)
>>> db.session.commit()

>>> posts = u.posts.all()
>>> posts
[<'Dann mache ich auch mal einen Post!>]
```

Die folgende Abfrage liefert alle Posts für alle User:

```
>>> posts = Post.query.all()
>>> for p in posts:
...     print(p.id, p.author.username, p.body)
...
1 jochen Das scheint tatsächlich zu funktionieren!
2 jochen Das ist mein zweiter Post
3 paula Dann mache ich auch mal einen Post!
```

Das Löschen von Objekten erfolgt mit db.session.delete(Objekt):

```
>>> users = User.query.all()
>>> for u in users:
...     db.session.delete(u)
...
>>> posts = Post.query.all()
>>> for p in posts:
...     db.session.delete(p)
...
>>> db.session.commit()
```

5.9 DIE FLASK-SHELL

Im weiteren Verlauf des Kurses werden Sie wahrscheinlich häufiger Tests in der REPL durchführen. Um das zu vereinfachen, können Sie die Flask-Shell benutzen. Damit können Sie die Klassen Ihrer App benutzen, ohne alles einzeln importieren zu müssen. Dafür benötigen Sie einen sogenannten Shell Context, den Sie in der Datei tutorial/microblog.py einrichten:

```
1 # tutorial/microblog.py
2 from app import app, db
3 from app.models import User, Post
4
5 @app.shell_context_processor
6 def make_shell_context():
7     return {'db': db, 'User': User, 'Post': Post}
```

- Zeilen 2 und 3: Die nötigen imports für die Klassen db, User und Post werden gemacht.
- Zeile 5: Mit dem vordefinierten Decorator @app.shell_context_processor vor der Funktion make_shell_context() wird die Funktion als 'Context Processor' in Flask registriert.
- Zeile 7: Rückgabewert der dekorierten Funktion ist ein Dictionary, dass die Klassen db, User und Post als Werte enthält.

Um die Flask Shell benutzen zu können, muss Ihr virtuelles Environment aktiv sein und die Variable FLASK_APP muss in Ihrer Betriebssystem-Shell gesetzt sein:

```
(tutorial) $ flask shell
Python 3.9.5 (v3.9.5:0a7dcbdb13, May 3 2021, 13:17:02)
[Clang 6.0 (clang-600.0.57)] on darwin
App: app [production]
Instance: /Users/jochenreinholdt/tutorial/instance

>>> db
<SQLAlchemy engine=sqlite:///Users/jochenreinholdt/tutorial/app.db>

>>> User
<class 'app.models.User'>

>>> Post
<class 'app.models.Post'>
>>>
```

Alle benötigten Definitionen sind jetzt importiert und Sie können sofort Objekte Ihrer App interaktiv anlegen und testen!

Tipp:

Wenn Sie in Zukunft nicht immer die Variable FLASK_APP und andere Variablen manuell setzen wollen, können Sie sie in eine Datei .flaskenv in Ihrem Projektverzeichnis eintragen:

```
1 # tutorial/.flaskenv
2 FLASK_APP=microblog.py
```

5.9.1 MUSTERLÖSUNGEN

Die Musterlösungen zu diesem Kapitel sind Bestandteil des Repository

<https://github.com/miguelgrinberg/microblog>

Wenn Sie es wie im Kapitel 3.4 beschrieben geclont haben, geben Sie im Verzeichnis microblog das folgende Kommando ein:

```
$ git checkout v0.4
```

5.10 HINTERGRUND

Dieses Kapitel enthält Zusatzinformationen, die zum tieferen Verständnis des praktischen Beispiels beitragen können. Für die Bearbeitung der weiteren Kapitel dieses Kurses ist die Lektüre aber nicht zwingend nötig.

Zu den praktischen Beispielen in diesem Kapitel finden Sie hier eine Diskussion der benutzten Pakete, Klassen und Objekte

5.10.1 SQLALCHEMY UND FLASK-SQLALCHEMY

Die Erweiterung flask-sqlalchemy macht Klassen aus SQLAlchemy in Flask verfügbar:



Abbildung 5-4 Die Erweiterung flask-sqlalchemy

SQLAlchemy hat die folgenden Komponenten:

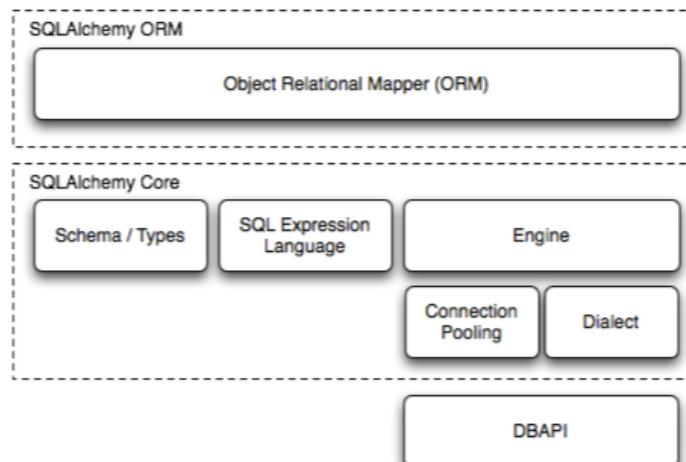


Abbildung 5-5 Architektur von SQLAlchemy. Quelle:
<https://docs.sqlalchemy.org>

Der **SQLAlchemy Core** basiert auf der Schnittstellenbeschreibung **DBAPI** von Python und enthält:

- **Schema / Types:** Verwaltet Datenbank-Schemaobjekte wie Tabellen und Spaltentypen
- **SQL Expression Language:** Erzeugt SQL-Anweisungen, die von einem DBMS wie MySQL aufgeführt werden können und die eine Ergebnismenge erzeugen.
- Die **Engine** ist dafür zuständig, Connections zu einem Datenbankserver aufzubauen und zu verwalten. Sie benutzt dabei **Connection Pooling** (Ein Mechanismus, mit dem die meist begrenzten Anzahl von Datenbank-Connections umzugehen) und einen **Dialect** (Der die für Produkte wie MySQL, MariaDB, PostgreSQL usw. spezifischen Eigenheiten verwaltet)

SQLAlchemy ORM baut auf dem Core auf. Diese Komponente hat die folgenden Aufgaben:

- Datenbank-Tabellen aus Python-Klassen erzeugen
- Fremdschlüssel-Beziehungen in der Datenbank erzeugen
- Klassenhierarchien mit Vererbung in der Datenbank abbilden
- Python-Objekte in Tabellen-Zeilen speichern
- Ergebnis-Zeilen von SELECT lesen und daraus Python-Objekte erzeugen
- Dafür zu sorgen, dass die Beziehungen zwischen verschiedenen Objekten korrekt in den Datenbanktabellen erhalten bleiben.

Die Klasse SQLAlchemy und das db-Objekt

In **app/__init__.py** haben Sie anschliessend folgende Zeilen eingegeben:

```
# Import der Klasse:  
from flask_sqlalchemy import SQLAlchemy  
  
# Erzeugung eines SQLAlchemy-Objekts:  
db = SQLAlchemy(app)
```

Was bedeutet das?

- Sie haben Ihr app-Objekt dem Konstruktor SQLAlchemy() übergeben. Das Ergebnis ist ein SQLAlchemy-Objekt, dem Sie den Namen *db* gegeben haben.
- Dieses Objekt repräsentiert nun (über den ORM-Mechanismus) die Datenbank. Es ermöglicht die Benutzung der Funktionen und Klassen aus den Modulen sqlalchemy und sqlalchemy.orm. Ihre App ist dem Objekt db ebenfalls bekannt, da Sie diese beim Erzeugen des Objekts übergeben haben.
- Das Objekt mit dem Namen db ist jetzt Bestandteil des Package app, weil es in `__init__.py` erzeugt wird. Es kann jetzt in anderen Modulen aus app importiert und genutzt werden

In **app/models.py** geschieht genau das:

```
from app import db  
class User(db.Model):  
    id = db.Column(db.Integer, primary_key=True)  
    username = db.Column(db.String(64), index=True, unique=True)  
    # usw ...
```

Was steht hinter db.Model, db.Column, db.String und db.Integer ?

- db.Model ist eine Basis-Klasse aus SQLAlchemy, die die Deklaration einer eigenen Klasse (und einer Tabelle) des Datenmodells ermöglicht. Die Klasse User wird hier als Unterklasse von db.Model erzeugt. Sie erbt damit die Attribute und Methoden von db.Model.
- db.Column ist eine Klasse aus SQLAlchemy, mit der Attribute / Tabellenspalten deklariert werden können. Durch Aufruf der Konstruktormethode db.Column() werden Objekte mit den Namen id und username erzeugt, die zwei Tabellenspalten repräsentieren.

- db.String und db.Integer sind Klassen aus SQLAlchemy für Spaltentypen in Tabellen. Durch den Aufruf der Konstruktormethode db.String() wird die Spalte username in der Datenbank als Datentyp VARCHAR mit der Eigenschaft UNIQUE angelegt. Durch den Aufruf der Konstruktormethode db.Integer() wird die Spalte id in der Datenbank mit der Eigenschaft NOT NULL erzeugt. Außerdem 'weiss' SQLAlchemy nun, dass es sich um den Primärschlüssel der Tabelle handeln soll und nur eindeutige Werte eingefügt werden dürfen.

Die Klasse User erbt von db.Model die Methode query.all() und query.get(id) mit der Abfragen aus der Tabelle user in der Datenbank möglich sind.

Das SQLAlchemy-Objekt db hat:

- Die Methode create_all(), mit der alle Models als Tabellen angelegt werden können
- Die Methode drop_all(), mit der alle Tabellen entfernt werden können.
- Ein Attribut *session*. Das ist ein sogenannter Kontext, mit dem verschiedene Aktionen in der App ausgeführt werden können.

Eine session baut eine Verbindung mit dem Datenbanksystem auf. Sie können Model-Objekte zu einer session hinzufügen und dann in die Datenbank schreiben.

```
>>> p = User(username='paula', email='paula90@example.com')
>>> db.session.add(p)
```

Die session verwaltet die Beziehungen zwischen Objekten und Tabellenzeilen in einer sogenannten 'Identity Map', einer Datenstruktur, die Kopien einzelner eindeutiger Objekte enthält. Jedes dieser Objekte bekommt von SQLAlchemy einen eindeutigen Primärschlüssel-Wert zugewiesen.

Sie benutzt eine Engine, mit der für Objekte der App SELECT-, INSERT- und UPDATE-Anweisungen auf der Datenbank ausgeführt werden können.

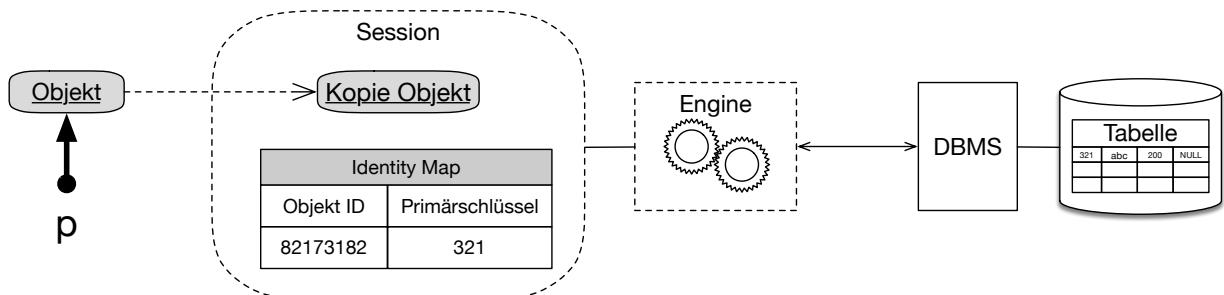


Abbildung 5-6 Session mit Objekt und Tabellen-Zeile

Zeilen werden der Tabelle hinzugefügt, sobald sie zu einer session mit session.add() hinzugefügt wurden. Permanent gespeichert sind Sie aber erst nach dieser Anweisung:

```
>>> db.session.commit()
```

Das Beispiel führt zu einem INSERT in die Tabelle user. Mehrere Änderungen können einer Session hinzugefügt werden und dann gemeinsam mit session.commit() bestätigt werden. Dies entspricht einer Datenbank-Transaktion mit BEGIN ... COMMIT. Eine Transaktion fasst mehrere Änderungen zu einer

Einheit zusammen. Entweder werden alle Änderungen bestätigt (COMMIT) oder alle werden verworfen (ROLLBACK).

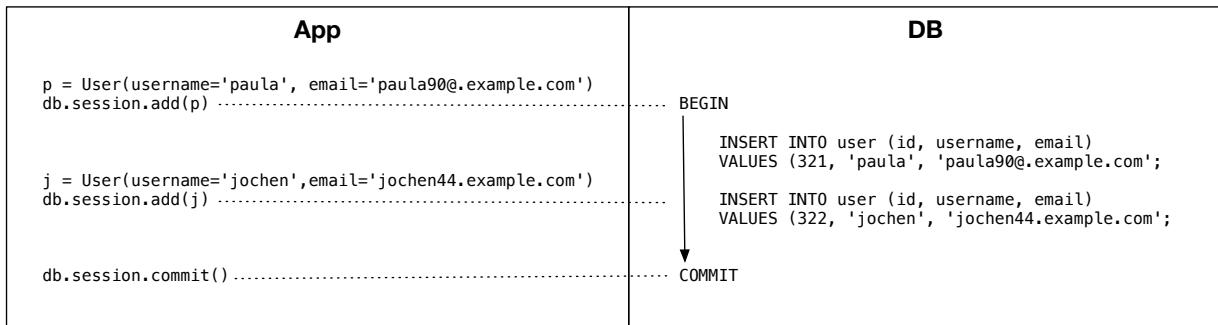


Abbildung 5-7 Datenbank-Transaktion

Mehrere Änderungen, die einer session hinzugefügt werden, bevor db.session.commit() ausgeführt wird, bilden also eine Einheit.

SQLAlchemy Query Methoden

Objekt.query.all()	Entspricht SELECT * FROM tabelle ohne WHERE. Liefert alle Datensätze aus der Tabelle
Objekt.query.first()	Entspricht SELECT * FROM tabelle ohne WHERE. Liefert nur den ersten Datensatz.
Objekt.query.get(Schlüsselwert)	Entspricht SELECT * FROM tabelle mit WHERE id = Schlüsselwert. Liefert genau einen Datensatz mit der id (Primärschlüssel)
Objekt.query.count()	Entspricht COUNT(*) FROM tabelle

SQLAlchemy Filter

Objekt.query.filter_by(Filterbedingung).all()	Entspricht SELECT * mit WHERE. Liefert alle Datensätze aus der Tabelle, auf die die Filterbedingung Attribut = Wert zutrifft.
Objekt.query.filter_by(Filterbedingung).first()	Liefert nur die ersten Anzahl/Datensätze, auf die die Filterbedingung Attribut = Wert zutrifft.
Objekt.query.order_by(Attribut).all()	Sortiert die Ergebnissezeilen nach Attribut
Objekt.query.group_by(Attribut).all()	Gruppert die Ergebnissezeilen nach SELECT * FROM tabelle

Beispiele für Queries ohne Filter

```
l = User.query.all()                      # Alle User, als Liste
u = User.query.first()                    # Nur den 1. User
u = User.query.get(1)                     # User mit der id 1
```

Beispiele für Queries mit Filter

Filter werden vor all() und first angegeben:

```
u = User.query.filter_by(username='peter').first()      # einzelnes Objekt
l = User.query.filter(User.email.endswith('.com')).all()    # Als Liste
l = User.query.order_by(User.username).all()            # liefert Liste
l = User.query.limit(1).all()                          # liefert Liste
```

UPDATE

Ein UPDATE erfolgt einfach durch eine Änderung des Objekts das erneute Hinzufügen zu einer session und db.session.commit:

```
>>> u = User.query.get(1)
>>> u.email = 'neu.example.ch'
>>> db.session.add(u)
>>> db.session.commit()
```

DELETE

Um einen Datensatz zu löschen, wird db.session benutzt

```
>>> u = User.query.get(1)
>>> db.session.delete(u)
>>> db.session.commit()
```

Weitere Details und fortgeschrittene Möglichkeiten finden Sie in der Dokumentation von flask-sqlalchemy⁷ und SQLAlchemy⁸

⁷ <https://flask-sqlalchemy.palletsprojects.com/en/2.x/>

⁸ <https://www.sqlalchemy.org>

5.10.2 ALEMBIC UND FLASK-MIGRATE

Sie hätten die Tabellen User und Post auch schon nur mit SQLAlchemy durch db.create_all() anlegen können. Stattdessen haben Sie das indirekt mit zwei Kommandos aus Alembic gemacht:

```
$ alembic revision          # erzeugt ein Migrationsscript mit Versionsnummer  
$ alembic upgrade head      # Führt das script mit der neuesten Version aus
```

Der Vorteil von mit Alembic verwalteten Migrationen ist, dass Sie damit eine automatische Versionierung aller Änderungen am Datenbankschema erhalten und Änderungen auch zurücknehmen können, falls Sie später feststellen, dass die Änderung nicht ganz Ihren Erwartungen entsprochen hat. Sie können damit den Zustand des Datenbankschemas zu jedem Zeitpunkt in der Vergangenheit wiederherstellen.

Die Erweiterung flask-migrate macht Klassen aus Alembic in Flask verfügbar



Abbildung 5-8 Die Erweiterung flask-migrate

Nach der Installation mit pip konnten Sie daher die folgenden Kommandos benutzen:

```
$ flask db migrate -m 'User v0.1'  # wie alembic revision  
$ flask db upgrade                 # wie alembic upgrade head
```

Der Vorteil von flask-migrate gegenüber der direkten Anwendung von Alembic ist die vereinfachte Handhabung von Migrationen und die Bereitstellung von flask db downgrade, womit sie auf den vorherigen Stand zurückkehren können.

Die Versionen stehen an dieser Stelle des Projektverzeichnisses:

```
└── app  
    ├── __init__.py  
    ├── forms.py  
    ├── models.py  
    ├── routes.py  
    └── templates  
        ├── base.html  
        ├── index.html  
        └── login.html  
└── app.db  
└── config.py  
└── microblog.py  
└── alembic.ini  
└── env.py  
└── script.py.mako  
└── migrations  
    └── versions  
        ├── 91726de3c32f_users_table.py  <- Die zweite Migration  
        └── 9e5e15cdc72e_posts_table.py  <- Die erste Migration
```

Ein Blick in das mit **flask db migrate** erzeugte Script kann sich lohnen:

```
1  """users table
2  Revision ID: 91726de3c32f
3  Revises:
4  Create Date: 2022-03-09 09:33:05.074094
5  """
6  from alembic import op
7  import sqlalchemy as sa
8
9  # revision identifiers, used by Alembic.
10 revision = '91726de3c32f'
11 down_revision = None
12 branch_labels = None
13 depends_on = None
14
15 def upgrade():
16     # ### commands auto generated by Alembic - please adjust! ###
17     op.create_table('user',
18         sa.Column('id', sa.Integer(), nullable=False),
19         sa.Column('username', sa.String(length=64), nullable=True),
20         sa.Column('email', sa.String(length=120), nullable=True),
21         sa.Column('password_hash', sa.String(length=128), nullable=True),
22         sa.PrimaryKeyConstraint('id')
23     )
24     op.create_index(op.f('ix_user_email'), 'user', ['email'], unique=True)
25     op.create_index(op.f('ix_user_username'), 'user', ['username'], unique=True)
26     # ### end Alembic commands ###
27
28 def downgrade():
29     # ### commands auto generated by Alembic - please adjust! ###
30     op.drop_index(op.f('ix_user_username'), table_name='user')
31     op.drop_index(op.f('ix_user_email'), table_name='user')
32     op.drop_table('user')
33     # ### end Alembic commands ###
```

- Zeile 7: sqlalchemy wird importiert. Alembic und flask-migrate benutzen also SQLAlchemy.
- Zeilen 10 -13: Dies ist das Script für die initiale Migration. Es gibt eine revision id für diese Version. Die down_revision ist hier gleich None. Für spätere Versionen würde hier die id der vorherigen Version stehen.

```
12 # revision identifiers, used by Alembic.
13 revision = '9e5e15cdc72e'
14 down_revision = '91726de3c32f'
```

- Zeilen 15 – 26: Die Funktion upgrade enthält die nötigen Anweisungen zum anlegen des Tabellschemas. Hier könnte Sie eingreifen. Falls Sie zum Beispiel den Tabellennamen 'users' statt 'user' bevorzugen, editieren Sie das Script einfach, bevor Sie es mit flask db upgrade ausführen!
- Zeilen 28 – 33: Ein Downgrade von dieser 1. Version würde die Tabelle user und die Indizes löschen, da dies der Zustand der leeren Datenbank wäre.

Weitere Details und fortgeschrittene Möglichkeiten finden Sie in der Dokumentation von flask-migrate⁹ und Alembic¹⁰

5.10.3 DURCHFÜHRUNG DES BEISPIELS MIT MYSQL

Statt SQLite möchten Sie vielleicht MySQL als Datenbanksystem nutzen. Das ist nicht weiter schwierig, aber es sind weitere Schritte nötig, die bei SQLite wegfallen, da es bereits in Python eingebaut ist.

1. Sie müssen mysql als root aufrufen

```
$ mysql -u root -p
```

2. Sie müssen eine neue Datenbank 'microblog' anlegen

```
mysql> CREATE DATABASE microblog;
```

3. Sie sollten einen neuen DB-Benutzer mit Passwort anlegen

```
mysql> CREATE USER 'jochen'@'localhost' IDENTIFIED BY 'geheim';
```

4. Diesem Benutzer geben Sie alle Rechte in der Datenbank 'microblog'

```
mysql> GRANT ALL PRIVILEGES ON microblog.* TO 'jochen'@'localhost';
```

5. Verlassen Sie danach mysql und Aktivieren Sie Ihr virtuelles Environment

6. Installieren Sie das Paket pymysql, einen Connector zwischen Python und MySQL

```
(tutorial) $ pip install pymysql
```

7. Installieren Sie das Paket cryptography, das SQLAlchemy und Alembic für das Login bei MySQL gebraucht wird

```
(tutorial) $ pip install cryptography
```

8. Sie müssen die Umgebungsvariable DATABASE_URL setzen. Zunächst noch einmal kurz zur Datei config.py und speziell zu der folgenden Zeile:

```
SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
    'sqlite:/// + os.path.join(basedir, 'app.db')
```

Falls DATABASE_URL vorhanden ist, wird der Inhalt benutzt. Falls nicht, wird stattdessen die SQLite verwendet. Sie können nun die Umgebungsvariable erzeugen:

```
### Linux und Mac OS
(tutorial) $ export \
DATABASE_URL='mysql+pymysql://jochen:geheim@localhost/microblog'
```

```
### Windows:
(tutorial) $ set \
DATABASE_URL='mysql+pymysql://jochen:geheim@localhost/microblog'
```

9. Stellen Sie sicher, dass FLASK_APP gesetzt ist

10. Führen Sie die folgenden Kommandos im virtuellen Environment aus

```
(tutorial) $ flask db init
(tutorial) $ flask db migrate -m "users table"
(tutorial) $ flask db upgrade
```

⁹ <https://flask-migrate.readthedocs.io/en/latest/>

¹⁰ <https://alembic.sqlalchemy.org/en/latest/>

5.11 ZUSAMMENFASSUNG

- In Flask benutzen Sie Objekte, in der Datenbank Tabellen. Die Daten der Objekte müssen in Datenbank-Tabellen abgespeichert und daraus wieder gelesen werden. Mechanismen dafür könnten Sie selbst programmieren, aber der Aufwand wäre sehr hoch.
- Das Datenmodell einer App wird in Form von Model-Klassen in `models.py` definiert. Für jede Art von Objekt in Ihrer App brauchen Sie eine solche Klasse.
- Mit den Erweiterungen `flask-sqlalchemy` und `flask-migrate` können Sie einen einfachen Zugriff Ihrer Flask-App auf eine relationale Datenbank einrichten, ohne dass Sie dafür jede Menge Code schreiben müssen.
- Durch `pip install flask-sqlalchemy` wird das Paket SQLAlchemy in Flask integriert. SQLAlchemy erlaubt das 'Object-Relational Mapping' (ORM), also eine Übersetzung von Objekten zu Tabellen und umgekehrt.
- Welche Datenbank für die App verwendet werden soll, wird in `config.py` festgelegt. Die Beispiele in diesem Kapitel wurden mit dem kleinen DBMS SQLite durchgeführt, aber natürlich kann für den Produktionseinsatz auch MySQL, MariaDB oder PostgreSQL verwendet werden.
- Durch `flask-sqlalchemy` werden Datenbankoperationen wie `db.session.add(Objekt)` und `db.session.commit()` in der App verfügbar. Die Klassen aus `models.py` erhalten durch `flask-sqlalchemy` zusätzliche Methoden wie `Klasse.query.all()` und `Klasse.query.get()`, mit denen Daten abgefragt werden.
- Durch `pip install flask-migrate` wird das Paket Alembic in Flask integriert. Alembic erlaubt es, Änderungen an der Datenbank kontrolliert durchzuführen. Das Kommando `flask db migrate` erzeugt aus den Model-Klassen ein Migrations-Script im Unterverzeichnis `migrations`, das mit `flask db upgrade` ausgeführt werden kann. Damit werden die nötigen Änderungen am Datenbankschema durchgeführt: Anlegen der DB, Erzeugen von Tabellen, Änderungen an der Tabellenstruktur.
- Alle Änderungen an der Datenbank werden als Versionen im Verzeichnis `migrations/versions` gespeichert.
- Neben `upgrade` gibt es auch noch das `downgrade`, mit dem zu einer älteren Version des Datenbank-Schemas zurückgekehrt werden kann, falls sich eine Änderung als Irrweg erweisen sollte.

6 BENUTZER-LOGIN

Nachdem im letzten Kapitel die Datenbank für die Aufnahme von Benutzer-Accounts vorbereitet wurde, soll das Login-Formular aus Kapitel 4 auch wirklich benutzt werden. In diesem Kapitel wird das als praktische Übung gezeigt, die Sie bitte Schritt für Schritt ausführen.

6.1 LERNZIELE

Nach der Bearbeitung dieses Kapitels können Sie:

- Beschreiben, was eine Hash-Funktion leistet
- Methoden zur Erzeugung und Überprüfung eines Hash-Wertes anwenden
- Die Erweiterung flask-login installieren und anwenden
- Die An- und Abmeldung von Benutzern verwalten
- Seiten in Ihrer App gegen unbefugte Benutzer schützen
- Die Registrierung als neuer Benutzer ermöglichen

6.2 PASSWÖRTER

Im User-Model in models.py ist in der Zeile 9 bereits ein Feld password_hash für das Passwort vorgesehen.

```
5 class User(db.Model):  
6     id = db.Column(db.Integer, primary_key=True)  
7     username = db.Column(db.String(64), index=True, unique=True)  
8     email = db.Column(db.String(120), index=True, unique=True)  
9     password_hash = db.Column(db.String(128))  
10    posts = db.relationship('Post', backref='author', lazy='dynamic')
```

Abbildung 6-1 Die bestehende Klasse User

Dabei gibt es noch eine kleine Hürde: Passwörter sollten auf keinen Fall unverschlüsselt gespeichert werden und das deutet der name password_has bereits an.

Ein Hash ist ein Wert, der mit einer Hash-Funktion aus einem String berechnet wurde. Eine Hash-Funktion ähnelt ein wenig einer 'Falltür' die sich nur in einer Richtung öffnet:

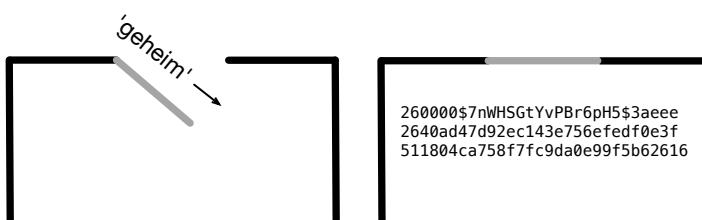


Abbildung 6-2 Falltür:-Analogie für eine Hash-Funktion

Eine Hash-Funktion hat die folgenden Eigenschaften:

- Wird die hash-Funktion wiederholt auf den gleichen String verwendet, liefert sie immer das gleiche Ergebnis

- Aus dem Ergebnis kann man den String nicht 'zurückberechnen', jedenfalls nicht mit einem vernünftigen Aufwand an Zeit und Rechenleistung.

In Flask stellt die Komponente *werkzeug* geeignete Hash-Funktionen zur Verfügung:

```
>>> from werkzeug.security import generate_password_hash

>>> hash = generate_password_hash('geheim')
>>> hash
'pbkdf2:sha256:260000$7nWHSgtYvPBr6pH5$3aeee2640ad47d92ec143e756efedf0e3f511804c
a758f7fc9da0e99f5b62616'

>>> from werkzeug.security import check_password_hash

>>> check_password_hash(hash, 'geheim')
True

>>> check_password_hash(hash, 'gemein')
False
```

In der Datei models.py fügen Sie der Klasse User zwei Importe und zwei Methoden hinzu:

```
from werkzeug.security import generate_password_hash, check_password_hash

class User(db.Model):
    # Zeilen ausgelassen
    # Neue Methoden:

    def set_password(self, password):
        self.password_hash = generate_password_hash(password)

    def check_password(self, password):
        return check_password_hash(self.password_hash, password)
```

Mit diesen beiden Methoden können Sie nun in der Flask Shell experimentieren. Stellen Sie zunächst sicher, dass Ihr virtuelles Environment aktiv ist und FLASK_APP=microblog.py gesetzt ist (Manuell oder in tutorial/.flaskenv):

```
(tutorial) $ flask shell
Python 3.9.5 (v3.9.5:0a7dcdbb13, May 3 2021, 13:17:02)
[Clang 6.0 (clang-600.0.57)] on darwin
App: app [production]
Instance: /Users/jochenreinholdt/00_Arbeit/00_Unterricht/tutorial/instance

>>> u = User(username='paul', email='paul@example.com')
>>> u.set_password('strengGeheim')

>>> u.check_password('IchVersuchsMal')
False

>>> u.check_password('strengGeheim')
True
```

6.3 DIE ERWEITERUNG FLASK-LOGIN

Neben dem Mechanismus für die verschlüsselten Passwörter brauchen Sie noch etwas, was eine Benutzer-Sitzung (session) verwaltet, einen sogenannten LoginManager. Dieser wird durch die Erweiterung flask-login geliefert.



Installieren Sie sie in Ihrem virtuellen Environment:

```
(tutorial) $ pip install flask-login
```

Flask-Login stellt die session-Verwaltung für Benutzer Ihrer App zur Verfügung. Es ermöglicht Anmeldung und Abmeldung und speichert die session-Informationen. Es ermöglicht Ihnen insbesondere:

- Die user-id des aktiven Benutzers zu speichern und jederzeit verfügbar zu halten
- Den Zugriff auf Seiten Ihrer App zu kontrollieren
- Die 'Remember Me' Funktionalität zu nutzen
- Benutzersitzungen ohne Cookies zu kontrollieren
- Die Integration mit anderen Erweiterungen für die Authorisierung einzurichten und den Zugriff auch über andere Mechanismen als Username/Passwort zu regeln

In der Datei app/__init__.py brauchen Sie neue Einträge (Zeilen 5 und 13)

```
1 from flask import Flask
2 from flask_sqlalchemy import SQLAlchemy
3 from flask_migrate import Migrate
4 # LoginManager importieren
5 from flask_login import LoginManager
6 from config import Config
7
8 app = Flask(__name__)
9 app.config.from_object(Config)
10 db = SQLAlchemy(app)
11 migrate = Migrate(app, db)
12 # Hier wird die Erweiterung initialisiert
13 login = LoginManager(app)
14 login.login_view = 'login'
15
16 from app import routes, models
```

Das Objekt login ist eine Instanz der Klasse LoginManager aus flask-login. Da Sie es in __init__.py initialisieren, ist es nun Bestandteil Ihres Package app.

Nun müssen Sie die Klasse User in app/models.py noch anpassen, damit sie mit dem LoginManager zusammenarbeiten kann. Nötig sind:

- Ein Attribut vom Typ bool mit den Namen **is_authenticated**. Es hat den Wert True, wenn der Benutzer erfolgreich angemeldet ist und False, wenn nicht.
- Ein Attribut vom Typ bool mit den Namen **is_active**. Es hat den Wert True, wenn der Benutzer aktiv ist und False, wenn er gesperrt wurde.
- Ein Attribut vom Typ bool mit den Namen **is_anonymous**. Es hat den Wert True, wenn der Benutzer nicht angemeldet ist und False, wenn er angemeldet ist.
- Eine Methode **get_id()**, die die User-id umgewandelt in einen String zurückgibt.

Sie können diese vier Attribute in die Klasse User einbauen, aber Sie können sie auch automatisch bekommen, wenn Sie die Klasse nicht nur von der Basisklasse db.Model, sondern zusätzlich von der Klasse UserMixin ableiten. Erweitern Sie deshalb app/models.py folgendermassen:

```
from app import db, login          #neu: login
from flask_login import UserMixin   #neu
from werkzeug.security import generate_password_hash, check_password_hash

class User(UserMixin, db.Model):      # User erbt nun von 2 Klassen
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), index=True, unique=True)
    # weitere Zeilen ...
```

Was eine Mixin-Klasse ist wird weiter unten im Abschnitt 'Hintergrund' am Ende dieses Kapitels erklärt.

Flask-Login 'folgt' dem angemeldeten Benutzer bei seiner Navigation durch die verschiedenen Seiten der App. Dazu braucht es die user-id, die Flask für jeden User speichert. Jedesmal, wenn der Benutzer auf eine andere Seite navigiert, liest Flask die user-id aus der user session und lädt sie in den Hauptspeicher.

Für Flask-Login muss id app/models.py noch eine Funktion hinzugefügt werden, die den User aus der Datenbank liest. Sie wird durch den Decorator @login.user_loader registriert. Fügen Sie folgendes zur Datei models.py hinzu (nicht innerhalb der Klasse User, sondern auf der gleichen Ebene):

```
@login.user_loader
def load_user(id):
    return User.query.get(int(id))
```

6.4 LOGIN IN DER VIEW-FUNKTION

Die Importe und die `login()` Funktion in `routes.py` müssen geändert bzw. ergänzt werden:

```
from flask import render_template, flash, redirect, url_for
from app import app, db
from app.forms import LoginForm, RegistrationForm
from app.models import User #neu
from flask_login import login_user, current_user #neu

@app.route('/login', methods=['GET', 'POST'])
def login():
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    form = LoginForm()
    if form.validate_on_submit():
        # geänderte Zeilen:
        user = User.query.filter_by(username=form.username.data).first()
        if user is None or not user.check_password(form.password.data):
            flash('Invalid username or password')
            return redirect(url_for('login'))
        login_user(user, remember=form.remember_me.data)
        return redirect(url_for('index'))
    return render_template('login.html', title='Sign In', form=form)
```

Die ersten beiden Zeilen in `login()` sind erklärungsbedürftig. Sie behandeln den Fall, dass ein angemeldeter Benutzer in der Hauptnavigation auf den Link `login` klickt oder den URL-Pfad `/login` eingibt.

Die Variable `current_user` ist ein Objekt aus der session von Flask-Login. Die Variable `current_user.is_authenticated` ist eines der vier Attribute, dass Sie der Klasse `User` über den zusätzlichen Eintrag `UserMixin` als 2. Basisklasse im letzten Abschnitt hinzugefügt haben.

Der Benutzer kann nun angemeldet werden. Der erste Schritt ist, den Benutzer aus der Datenbank zu laden:

```
user = User.query.filter_by(username=form.username.data).first()
```

Falls der `username` nicht in der Datenbank existiert oder die Überprüfung des Passworts fehlschlägt, gibt es eine Fehlermeldung und der Benutzer wird mit `redirect()` erneut zur Login-Form geschickt:

```
if user is None or not user.check_password(form.password.data):
    flash('Invalid username or password')
    return redirect(url_for('login'))
```

Wenn alles gut läuft, wird die Anmeldung vollzogen und der Benutzer auf die Startseite weitergeleitet:

```
login_user(user, remember=form.remember_me.data)
return redirect(url_for('index'))
```

6.5 LOGOUT

Dem Benutzer soll nun auch die Möglichkeit gegeben werden sich abzumelden. Dazu braucht es eine neue View-Funktion in app/routes.py. Zunächst muss ein Import gemacht werden, der die Funktion logout_user() verfügbar macht:

```
# ...
from flask_login import logout_user      #neu

# ... Weitere Zeilen ...

@app.route('/logout')                      #neu
def logout():
    logout_user()
    return redirect(url_for('index'))
```

Um einen Logout-Link im Navigationsbereich auf jeder Seite anzuzeigen, wird er am Anfang von <body> in app/templates/base.html hinzugefügt:

```
<div>
    Microblog:
        <a href="{{ url_for('index') }}>Home</a>
        {% if current_user.is_anonymous %}
            <a href="{{ url_for('login') }}>Login</a>
        {% else %}
            <a href="{{ url_for('logout') }}>Logout</a>
        {% endif %}
</div>
```

Es wird geprüft, ob der Benutzer überhaupt angemeldet ist. Wenn nicht, hat current_user.is_anonymous den Wert True. Dann wird der login-Link angezeigt. Andernfalls wird der logout-Link angezeigt.

6.6 SEITEN SCHÜTZEN

Ob der Benutzer angemeldet ist, soll für bestimmte Seiten der App geprüft werden. Falls ein User nicht angemeldet ist und er eine Seite aufruft, für die ein Login erforderlich ist, soll er auf die Login-Seite weitergeleitet werden. In app/__init__.py wird folgender Eintrag hinzugefügt:

```
# ...neu:
login = LoginManager(app)
login.login_view = 'login'
# ...
```

Damit wird in der gesamten App der LoginManager und die View-Funktion login() bekannt gemacht.

In app/routes.py wird der index() Funktion der Decorator @login_required hinzugefügt. Zusätzlich muss ein weiterer Import gemacht werden:

```
from flask_login import login_required    #neu

# ... usw. ...

@app.route('/')
@app.route('/index')
@login_required                         #neu
def index():
    # ... usw. ...
```

Im Moment haben Sie nur die Seiten /index und /login. In späteren Versionen wird der Benutzer aber von irgendeiner Seite auf die /login Seite geschickt. Nachdem der Benutzer sich erfolgreich angemeldet hat, soll er auf die ursprünglich Seite zurückgeschickt werden.

Der Decorator @login_required fügt dem URL eine Information hinzu, von welcher Seite aus er den Benutzer zu /login weitergeleitet hat. Das steht in einem Attribut next. Kommt der Benutzer beispielsweise von /index auf /login, steht dort: **/login?next=/index**

Die App kann diesen Pfad in der View-Funktion benutzen, um den redirect zu machen. In app/routes.py sind für diese Funktionalität zwei weitere Importe für request und url_parse nötig:

```
from flask import render_template, flash, redirect, url_for, request
from werkzeug.urls import url_parse

@app.route('/login', methods=['GET', 'POST'])
def login():
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(username=form.username.data).first()
        if user is None or not user.check_password(form.password.data):
            flash('Invalid username or password')
            return redirect(url_for('login'))
        login_user(user, remember=form.remember_me.data)
        next_page = request.args.get('next') # Rückkehr-Pfad
        if not next_page or url_parse(next_page).netloc != '':
            next_page = url_for('index')
        return redirect(next_page)
    return render_template('login.html', title='Sign In', form=form)
```

Hier wird Das request.args-Objekt von Flask benutzt, um die Argumente im URL auszuwerten. Es gibt drei Fälle:

1. Es gibt kein next-Argument im URL. Dann wird der Benutzer zu /index weitergeleitet.
2. Es gibt ein next-Argument. Dann wird der Benutzer dorthin weitergeleitet.
3. Es gibt ein next-Argument, aber es kann sein, dass sein Inhalt böswillig untergeschoben wurde und eine Weiterleitung auf eine fremde Website enthält. Falls next eine Domäne enthält, wird der Benutzer einfach an /index weitergeleitet.

Ob next_page eine Domäne enthält, kann mit url_parse(next_page).netloc geprüft werden. Wenn darin kein leerer String steht, wurde eine Domäne 'eingebaut'.

6.7 ANZEIGE DES BENUTZERS IN TEMPLATES

Nun kann der tatsächlich angemeldete Benutzer auch in Templates angezeigt werden. Dafür wird current_user.username genutzt. Machen Sie folgende Änderung in app/templates/index.html:

```
<!-- app/templates/index.html -->
{% extends "base.html" %}

{% block content %}
    <h1>Hi, {{ current_user.username }}!</h1>
    {% for post in posts %}
        <div><p>{{ post.author.username }} says: <b>{{ post.body }}</b></p></div>
    {% endfor %}
{% endblock %}
```

In app/routes.py wird nun die Variable user nicht mehr an das Template übergeben werden. Die Zeile:

```
return render_template('index.html', title='Home', user=user, posts=posts)
```

Ändern Sie bitte in:

```
return render_template('index.html', title='Home', posts=posts)
```

6.8 ERSTER TEST

Ein erster Test von /login kann nun gemacht werden. Allerdings hat die App noch keine Registrierung von neuen Benutzern. Sie können aber jederzeit User in der Python REPL anlegen:

```
(tutorial) $ flask shell
Python 3.9.5 (v3.9.5:0a7dcbdb13, May 3 2021, 13:17:02)
[Clang 6.0 (clang-600.0.57)] on darwin
App: app [production]
Instance: /Users/jochenreinholdt/00_Arbeit/00_Unterricht/tutorial/instance

>>> u = User(username='jochen', email='jochen@example.com')
>>> u.set_password('geheim')
>>> db.session.add(u)
>>> db.session.commit()
```

6.9 USER REGISTRIEREN

Für die Benutzerregistrierung brauchen Sie jetzt noch drei Dinge:

- Eine form-Klasse
- Ein Template
- Eine View-Funktion

6.9.1 DIE FORM FÜR DIE REGISTRIERUNG

Die Form-Klasse RegistrationForm wird in app/forms.py definiert

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import ValidationError, DataRequired, Email, EqualTo
from app.models import User

# ... weitere Zeilen

class RegistrationForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    email = StringField('Email', validators=[DataRequired(), Email()])
    password = PasswordField('Password', validators=[DataRequired()])
    password2 = PasswordField(
        'Repeat Password', validators=[DataRequired(), EqualTo('password')])
    submit = SubmitField('Register')
```

Die Feldtypen und die Validatoren sind Standard. Es wird automatisch geprüft, ob die Eingabe von Email dem Format user@domain entspricht und ob die zweite Eingabe des Passworts der ersten entspricht. Sie müssen dafür ein Modul in Ihrem virtuellen Environment installieren

```
(tutorial) $ pip install email-validator
```

Die neuen Validatoren müssen auch noch am Anfang von app/forms.py importiert werden:

```
from wtforms.validators import ValidationError, DataRequired, Email, EqualTo
```

Darüber hinaus sollen noch zwei Regeln gelten:

- Usernamen sollen eindeutig sein. Es muss also geprüft werden, ob der Username schon existiert.
- E-Mail Adressen sollen eindeutig sein. Es muss geprüft werden, ob eine Adresse schon existiert

Was Sie deshalb selbst schreiben müssen sind zwei zusätzliche Methoden für RegistrationForm:

```
def validate_username(self, username):
    user = User.query.filter_by(username=username.data).first()
    if user is not None:
        raise ValidationError('Please use a different username.')

def validate_email(self, email):
    user = User.query.filter_by(email=email.data).first()
    if user is not None:
        raise ValidationError('Please use a different email address.)
```

Wenn in einer FlaskForm-Klasse Methoden mit einem Namen, der dem Muster validate_Feldname vorhanden sind, werden diese automatisch als zusätzliche Validatoren für die Felder hinzugefügt und bei validate_on_submit() ausgeführt.

Die Methode validate_username(`self, username`) sucht in der Datenbank nach dem Usernamen und wenn Sie fündig wird (`user is not None`), wird `raise` benutzt. Mit `raise` können Sie selbst eine Exception auslösen, hier ist es eine Exception vom Typ ValidationError.

Die Methode validate_email(`self, email`) funktioniert nach demselben Prinzip.

Beachten Sie, das auch ValidationError importieren müssen. Die Exception ValidationError wird von FlaskForm in validate_on_submit() verarbeitet, wenn sie auftritt.

6.9.2 DAS TEMPLATE FÜR DIE REGISTRIERUNG

Das Template wird als neue Datei unter app/templates angelegt und hat den Inhalt:

```
<!-- app/templates/register.html -->
{% extends "base.html" %}

{% block content %}
    <h1>Register</h1>
    <form action="" method="post">
        {{ form.hidden_tag() }}
        <p>
            {{ form.username.label }}<br>
            {{ form.username(size=32) }}<br>
            {% for error in form.username.errors %}
                <span style="color: red;">{{ error }}</span>
            {% endfor %}
        </p>
        <p>
            {{ form.email.label }}<br>
            {{ form.email(size=64) }}<br>
            {% for error in form.email.errors %}
                <span style="color: red;">{{ error }}</span>
            {% endfor %}
        </p>
        <p>
            {{ form.password.label }}<br>
            {{ form.password(size=32) }}<br>
            {% for error in form.password.errors %}
                <span style="color: red;">{{ error }}</span>
            {% endfor %}
        </p>
        <p>
            {{ form.password2.label }}<br>
            {{ form.password2(size=32) }}<br>
            {% for error in form.password2.errors %}
                <span style="color: red;">{{ error }}</span>
            {% endfor %}
        </p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

Die Elemente unterscheiden sich nicht wesentlich vom Template, das sie für die Login-Form geschrieben haben.

Nun brauchen wir noch einen Link auf der Login-Seite, der auf die Registrierung verweist. Fügen Sie den in app/templates/login.html am Ende vor `{% endblock %}` ein:

```
</form>
<p>New User? <a href="{{ url_for('register') }}>Click to Register!</a></p>
{% endblock %}
```

6.9.3 DIE VIEW-FUNKTION FÜR DIE REGISTRIERUNG

Als letzte Pendenz schreiben Sie die View-Funktion in app/routes.py.

```
from flask import render_template, flash, redirect, url_for, request
from flask_login import login_user, logout_user, current_user, login_required
from werkzeug.urls import url_parse
from app import app, db
from app.forms import LoginForm, RegistrationForm      #neu
from app.models import User

@app.route('/register', methods=['GET', 'POST'])
def register():
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    form = RegistrationForm()
    if form.validate_on_submit():
        user = User(username=form.username.data, email=form.email.data)
        user.set_password(form.password.data)
        db.session.add(user)
        db.session.commit()
        flash('Congratulations, you are now a registered user!')
        return redirect(url_for('login'))
    return render_template('register.html', title='Register', form=form)
```

Der User wird nun erzeugt und das Passwort verschlüsselt. Anschliessend wird der neue User der Datenbank hinzugefügt, eine Meldung ausgegeben und der Benutzer wird auf die Login-Seite geschickt.

6.10 MUSTERLÖSUNGEN

Die Musterlösungen zu diesem Kapitel sind Bestandteil des Repository

<https://github.com/miguelgrinberg/microblog>

Wenn Sie es wie im Kapitel 3.4 beschrieben geklont haben, geben Sie im Verzeichnis microblog das folgende Kommando ein:

```
$ git checkout v0.5
```

6.11 TEST

Es wurden eine Vielzahl von Funktionen eingebaut und das Login und die Registrierung sollen einigen Regeln folgen. Was müssen Sie alles testen, damit Sie allfällige Fehler finden?

Aufgabe

Stellen Sie einen Testplan auf. Er enthält für jede Funktion bzw. Fehlermöglichkeit mindestens zwei Testfälle:

- Ein Testfall, der Fehler bei richtiger Benutzung finden kann. Beispiele:
 - Ein registrierter Benutzer kann sich einloggen und anschliessend auf die Seite /index zugreifen
 - Ein neuer Benutzer kann sich registrieren und ist anschliessend in der Datenbank gespeichert
- Ein Testfall, der Fehler bei falscher Benutzung finden kann. Beispiele:
 - Ein registrierter Benutzer gibt das falsche Passwort ein und kann nicht auf die Seite /index zugreifen
 - Ein Benutzer gibt den falschen Usernamen an und wird erneut zum Login aufgefordert

Beschreiben Sie für jeden Testfall, was das erwartete Ergebnis ist

Führen Sie die Tests durch und dokumentieren Sie die Ergebnisse.

6.12 HINTERGRUND

Dieses Kapitel enthält Zusatzinformationen, die zum tieferen Verständnis des praktischen Beispiels beitragen können. Für die Bearbeitung der weiteren Kapitel dieses Kurses ist die Lektüre aber nicht zwingend nötig.

6.12.1 MEHRFACH-VERERBUNG

In Python ist es möglich, dass eine Klasse von mehreren anderen Klassen abgeleitet wird. Dies wird im Code des Tutorials bei der Klasse User angewendet:

```
class User(UserMixin, db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), index=True, unique=True)
    email = db.Column(db.String(120), index=True, unique=True)
    password_hash = db.Column(db.String(128))
    posts = db.relationship('Post', backref='author', lazy='dynamic')
```

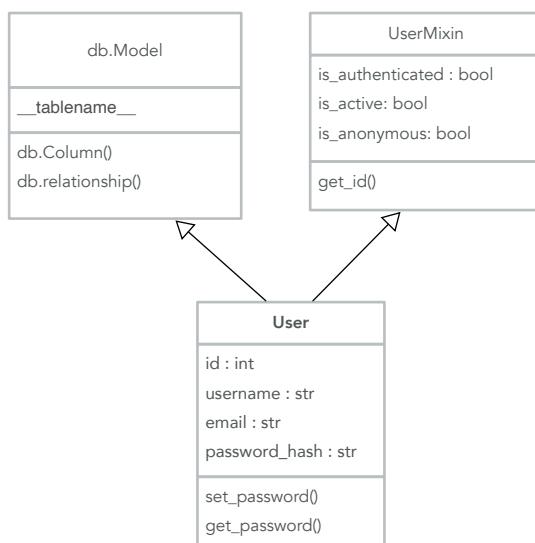


Abbildung 6-3 Mehrfache Vererbung

Die Klasse erbt damit die Eigenschaften und Methoden sowohl aus UserMixin als auch aus db.Model. Das folgende Beispiel zeigt, dass das Attribut `__tablename__` aus db.Model verfügbar ist, und dass das Attribut `is_authenticated` sowie die Methode `get_id()` aus UserMixin in einem Objekt der Klasse User verfügbar sind:

```
>>> u = User(username='paula', email='paula@example.com')
>>> db.session.add(u)
>>> db.session.commit()
>>> u.__tablename__
'user'

>>> u.is_authenticated
True
>>> u.get_id()
'1'
```

Mehrfach-Vererbung kann in anderen Programmiersprachen zu Fehlern führen, die schwer zu finden sind, wenn eine Vererbung von Methoden aus Klassen erfolgt, die wiederum von derselben Basisklasse

abgeleitet sind. Im folgenden Beispiel definiert die Klasse A eine Methode m(). Die Klassen B und C erweitern A. Die Klasse D wird sowohl von B als auch von C abgeleitet.

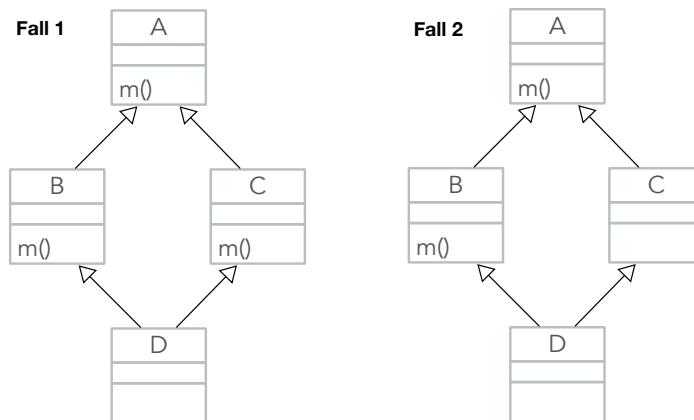


Abbildung 6-4 'Diamant-Problem' bei Mehrfachvererbung.

In Fall 1 überschreiben sowohl A als auch B die Methode m() mit einer eigenen Implementierung.

In Fall 2 überschreibt nur B die Methode m()

Welche Methode m() gilt nun in folgendem Fall?

```
>>> obj = D(B,C)
>>> obj.m()
```

In Python wurde dieses Problem wie folgt geregelt

Fall 1: Die Reihenfolge bei der Definition von D ist ausschlaggebend. Bei `obj = D(B,C)` wird `B.m()` aufgerufen. Bei `obj = D(C,B)` wird `B.m()` aufgerufen.

Fall 2: Python sucht der Hierarchie das oberste Vorkommen von m() und findet A. Python prüft dann, absteigend von A, ob in den abgeleiteten Klassen die Methode m() überschrieben wurde. Das letzte Glied in der Kette vor der Klasse D, das die Methode m() überschreibt, ist das ausschlaggebende. Bei `obj = D(C, B)` wird `B.m()` aufgerufen.

Überprüfen Sie das selbst mit einem kleinen Beispiel:

```
class Fahrzeug:
    def fahren(self):
        print('fahren() aus Fahrzeug')

class Motorfahrzeug(Fahrzeug):
    def fahren(self):
        print('fahren() aus Motorfahrzeug')

class Velo(Fahrzeug):
    #def fahren(self):
    #    print('fahren() aus Velo')
    pass

class Pedelec(Motorfahrzeug,Velo):
    pass

p = Pedelec()
p.fahren()
```

6.12.2 MIXIN-KLASSEN

Vererbung wird in der objektorientierten Programmierung aus zwei Gründen verwendet:

1. Zwei neue Typen (Unterklassen) Motorfahrzeug und Velo sollen auf Basis einer bestehenden Klasse (Oberklasse) Fahrzeug erzeugt werden. Die Vererbungs-Beziehung kann dann so formuliert werden: 'Ein Motorfahrzeug *ist ein* Fahrzeug' und 'Ein Velo *ist ein* Fahrzeug'. Motorfahrzeug erbt alle Attribute und Methoden von Fahrzeug, fügt eigene hinzu und überschreibt allenfalls Methoden von Fahrzeug. Mit Objekten der Klasse Motorfahrzeug kann man all das machen, was mit Fahrzeug möglich ist. Für Objekte der Klasse Velo gilt das gleiche. Motorfahrzeug und Velo sind neue Typen.
2. Man möchte lediglich das mehrfache Schreiben von identischem Code vermeiden und Methoden von einer bestehenden Klasse oder von mehreren bestehenden Klassen in eine neue Klasse übernehmen.

Für den Fall 2 wird mit Mixin-Klassen gearbeitet. Eine MixinKlasse definiert keinen neuen Typ. Sie ist eine Sammlung von Methoden, die anderswo nützlich sein können. Python bietet keine Möglichkeit, eine spezielle Art von Klasse zu definieren. Daher wird in der Praxis nach einer Konvention verfahren:

- Man fügt dem Klassennamen 'Mixin' hinzu, um deutlich zu machen, dass diese Klasse nur dazu dient, Methoden zur Verfügung zu stellen (Dieser Namenszusatz ist natürlich technisch nicht erforderlich und es gibt auch Fälle in offiziellen Python-Packages, bei denen sich die Programmierer nicht an diese Namenskonvention gehalten haben).
- Man erzeugt keine Objekte aus einer Mixin-Klasse.
- Man leitet eine Klasse niemals nur von einer Mixin-Klasse ab, sondern benutzt diese immer nur zusätzlich zur eigentlichen Oberklasse.
- Eine Mixin-Klasse sollte einen klar begrenzen und für einen speziellen Anwendungsbereich nützlichen Vorrat an Methoden bieten (Eine Klasse 'AllzweckMixin' mit diversen Methoden, die keinen echten Zusammenhang haben zu schreiben, ist technisch möglich. Es wäre aber schlechter Stil).

6.12.3 REQUESTS

Das Request-Objekt und die Informationen, die es zum http-Request liefert wurden bereits in Kapitel 2.5.1, Requests erklärt. Da das schon viele Seiten zurückliegt, wollen Sie es vielleicht noch einmal lesen.

6.12.4 MODEL-VIEW-CONTROLLER

Für Applikationen mit einer Benutzeroberfläche benutzt man häufig ein Architekturmuster mit dem Namen Model-View-Controller (MVC). Dabei geht es darum, die folgende Aufgaben zu trennen:

- Daten und Geschäftslogik der Applikation ('Model')
- Darstellung der Benutzeroberfläche ('View')
- Reaktion auf Benutzeraktionen ('Controller')

Diese drei Elemente und Ihre Zusammenarbeit werden häufig so dargestellt:

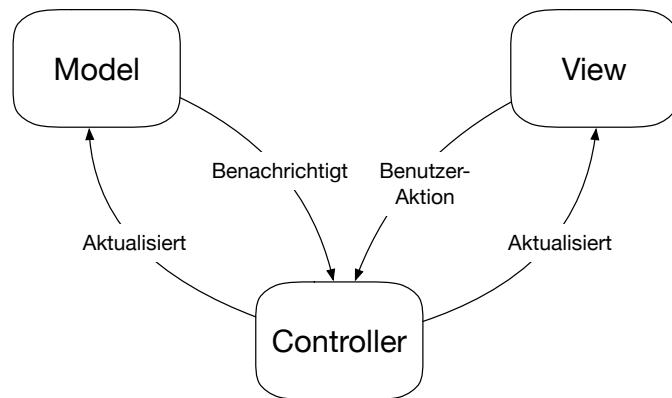


Abbildung 6-5 Muster Model-View-Controller

Es gibt viele verschiedene Spielarten des MVC-Musters und bei Ihrer Flask-App ist es eine dieser Spielarten. Die drei Elemente müssen nicht zwingend in separaten Modulen implementiert sein. Wie kann man Ihren Code den drei Elementen zuordnen?

Model:

- Die Model-Klassen User und Post in app/models.py
- Die Repräsentation in der Datenbank

View:

- Das Template für eine Seite unter app/templates
- Die Aufrufe von render_template() in den View-Funktionen
- Der Code in app/forms.py, der Felder definiert

Controller:

- Die Initialisierung der FlaskApp in __init__.py
- Die @app.route() Decorators in app/routes.py
- Der Code in den View-Funktionen in app/routes.py, der auf Benutzeraktionen reagiert, z.B. validate_on_submit() und die Zugriffe auf das Model set_password(), db.session.add() und db.session.commit().
- Der Code in app/forms.py, der auf das Model zugreift (Validierung username und password_hash)

6.13 ZUSAMMENFASSUNG

- Passwörter können Sie mit den Funktionen generate_password_hash() verschlüsseln und mit check_password_hash() überprüfen.
- Die Erweiterung Flask-Login ermöglicht es Ihnen, eine Benutzer-Session zu verwalten. Sie können damit ein login-Objekt erzeugen. Die Aufgaben Login, Logout und 'Remember me' (Eingeloggt bleiben) können Sie an dieses Objekt delegieren. Das gilt für die gesamte Zeitspanne, in der ein Benutzer angemeldet ist.

- Flask-Login stellt der App die Methoden `login_user()` und `logout_user()` zur Verfügung.
- Flask-Login benötigt eine von Ihnen definierte Funktion, die die Datens eines Users liefert. Diese Funktion wird Flask-Login mit dem Decorator `@login.user_loader` bekannt gemacht. In unserem Beispiel ist das die Funktion `load_user()`, die den User-Eintrag aus der Datenbank liest. In Ihrer App stellt Flask-Login diese Informationen im Objekt `current_user` zur Verfügung.
- Die App muss einige Attribute (`is_authenticated`, `is_active`, `is_anonymous`) und die Methode `get_id()` bereitstellen, damit Flask-Login genutzt werden kann. Das login-Objekt setzt diese Attribute automatisch auf True oder False und Sie können mit dem Decorator `@login_required` bestimmen, welche Seiten nur von angemeldeten Benutzern betreten werden können.
- Mit `redirect()` wird der Benutzer zu einer anderen Seite geschickt, nachdem er sich eingeloggt hat. Der Decorator `@login_required` fügt dem URL eine Information hinzu, von welcher Seite aus der Benutzer zu `/login` weitergeleitet wurde.
- Diese Seite steht in einem Attribut `next` im Request-Objekt von Flask, das alle Informationen des letzten http-Requests enthält (Wird im Code mit `request.args.get('next')` abgefragt. Da ein http-Request manipuliert sein kann, muss überprüft werden, ob darin wirklich eine Seite der App steht, oder eine externe Seite. Nur Redirects auf interne Seiten sollten zugelassen werden.

7 BENUTZERPROFILE

In diesem Kapitel wird die Erstellung und Darstellung eines Benutzerprofils behandelt.

7.1 LERNZIELE

Nach der Bearbeitung des Kapitels können Sie:

- Profilinformationen zu den Benutzern der App auf Profilseiten darstellen
- Einen kleinen Editor für Profilinformationen erstellen
- Den Webdienst Gravatar nutzen, um Benutzerprofilen Avatare hinzuzufügen

7.2 DIE PROFILSEITE

In app/routes.py wird eine neue View-Funktion erstellt. Der Decorator @login_required bewirkt, dass nur erfolgreich angemeldete Benutzer diese Seite betreten können. Die Route im Decorator @app.route enthält den Benutzernamen <username>. Dies ist eine dynamische Komponente in der URL.

```
73     @app.route('/user/<username>')
74     @login_required
75     def user(username):
76         user = User.query.filter_by(username=username).first_or_404()
77         posts = [
78             {'author': user, 'body': 'Test post #1'},
79             {'author': user, 'body': 'Test post #2'}
80         ]
81         return render_template('user.html', user=user, posts=posts)
```

Abbildung 7-1 - Die View-Funktion für einen User

- Zeile 75: Wenn eine Route wie in diesem Beispiel eine dynamische Komponente enthält, akzeptiert Flask an dieser Stelle einen beliebigen Text und wird den Text als Inhalt der Variablen username an die View-Funktion übergeben.
- Zeile 76: Eine Variante von User.query.first() wird zusammen mit einer Filterbedingung angewendet, um den User mit dem Usernamen aus der URL in der Datenbank zu finden. Die Methode first_or_404() liefert das User-Objekt, falls der User existiert. Falls der User nicht in der Datenbank gefunden wird, schickt Sie eine Response mit den Status-Code 404 (Not Found) an den Browser. Es ist also nicht nötig zu prüfen, ob die query None liefert, da in diesem Fall die Seite ohnehin verlassen wird.
- Zeilen 77-80: Hier werden provisorisch zwei Fake-Posts definiert (Die Erfassung von Posts wird erst in Kap. 10 implementiert. Sie könnten alternativ aber auch schon Posts mit der Flask-Shell erfassen).
- Zeile 81: Die Funktion render_template benutzt das Template user.html, dass im nächsten Schritt erstellt wird.

```

1  <!-- app/templates/user.html, 1. Version -->
2  {% extends "base.html" %}
3  {% block content %}
4      <h1>User: {{ user.username }}</h1>
5      <hr>
6      {% for post in posts %}
7          <p>
8              {{ post.author.username }} says: <b>{{ post.body }}</b>
9          </p>
10     {% endfor %}
11 {% endblock %}

```

Abbildung 7-2 - Vorläufiges Template für die Profilseite

- Zeile 2: Auch dieses Template erbt die Elemente aus base.html
- Zeile 4: Ein Platzhalter für den user.username aus der View-Funktion wird verwendet
- Zeilen 6-10: Die for-Schleife zeigt die Elemente des Dictionary posts aus der View-Funktion an, das die provisorischen Fake-Posts enthält.

Die Profilseite ist nun in einer 1. Version fertig. Allerdings gibt es noch keinen Link, der auf die Seite verweist. Daher wird ein Link mit dem Titel 'Profile' in die Liste der Links eingefügt:

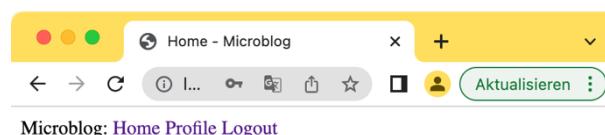
```

11   <div>
12       Microblog:
13       <a href="{{ url_for('index') }}">Home</a>
14       {% if current_user.is_anonymous %}
15           <a href="{{ url_for('login') }}">Login</a>
16       {% else %}
17           <a href="{{ url_for('user', username=current_user.username) }}">Profile</a>
18           <a href="{{ url_for('logout') }}">Logout</a>
19       {% endif %}
20   </div>

```

Abbildung 7-3 - Änderung der Links in base.html

Zeile 17: Hier wird der username aus dem current_user Objekt von Flask-Login (Siehe Kap. 6.3) in der Funktion url_for verwendet, um einen Link auf die Profilseite des angemeldeten Benutzers zu generieren. Die Anzeige sieht nun so aus:



Hi, jochen!

John says: Beautiful day in Portland!

Susan says: The Avengers movie was so cool!

Abbildung 7-4 Version 1 der Profilseite

7.3 AVATARE

Im Microblog-Tutorial werden Usern Profilbilder zugeordnet. Anstatt direkt Bilder hochzuladen, wird der Dienst Gravatar benutzt. Auf Gravatar¹¹ können Sie ein Bild hochladen oder ein generiertes geometrisches Muster erzeugen. Das Bild kann mit <https://www.gravatar.com/avatar/<hash>> abgerufen werden, wobei <hash> aus dem MD5-Hash der Email-Adresse des Benutzers besteht.

```
>>> from hashlib import md5  
>>> 'https://www.gravatar.com/avatar/' + md5(b'john@example.com').hexdigest()  
'https://www.gravatar.com/avatar/d4c74594d841139328695756648b6bd6'
```

Das Bild ist 80 x 80 Pixel gross, aber Sie können auch der URL mit s=Grösse einen Query-String mit einer anderen Grösse mitgeben:

<https://www.gravatar.com/avatar/729e26a2a2c7ff24a71958d4aa4e5f35?s=128>

In diesem Fall gibt gravatar.com ein Bild mit 128 x 128 zurück. Für ein anonymes, aber für jede Email-Adresse eindeutiges Muster können Sie im URL einen weiteren Query-String d=Muster angeben, z.B. d=identicon, der ein Muster in dieser Art erzeugt:

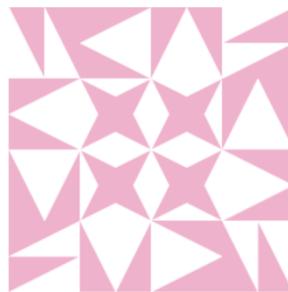


Abbildung 7-5 - 'Identicon'-Muster

Die Avatare gehören zu Usern, also wird in app/models.py md5 importiert und der Klasse User eine Methode zum Holen des Avatars hinzugefügt:

```
5  from hashlib import md5  
  
15  class User(UserMixin, db.Model):  
16  # Zeilen ausgelassen ...  
  
23  def avatar(self, size):  
24      digest = md5(self.email.lower().encode('utf-8')).hexdigest()  
25      return 'https://www.gravatar.com/avatar/{}?d=identicon&s={}'.format(  
26          digest, size)
```

Abbildung 7-6 Methode avatar() in der Klasse User

Die Methode gibt einen URL mit der gewünschten Grösse in Pixeln zurück. Für User, die kein eigenes Bild bei Gravatar hinterlegt haben, wird mit d=identicon ein Identicon zurückgegeben. Bei der Erzeugung des MD5-Hash gibt es die Besonderheit, dass dafür in Python der Datentyp byte verwendet

¹¹ <https://gravatar.com>

werden muss. In Zeile 24 wird die Email-Adresse zunächst in Kleinbuchstaben umgewandelt und dann von str in byte.

Nun wird die Anzeige der Avatare in das Template app/templates/user.html eingebaut:

```
1  <!-- app/template/user.html -->
2  {% extends "base.html" %}
3  {% block content %}
4      <table>
5          <tr valign="top">
6              <td></td>
7              <td><h1>User: {{ user.username }}</h1></td>
8      </tr>
9  </table>
10 <hr>
11 {% for post in posts %}
12     <table>
13         <tr valign="top">
14             <td></td>
15             <td>{{ post.author.username }} says:<br>{{ post.body }}</td>
16         </tr>
17     </table>
18 {% endfor %}
19 {% endblock %}
```

Abbildung 7-7 Das user-Template mit Avataren

Zeile 4-9: Es wird ein <table>-Element benutzt, um das grosse Avatar-Bild (128 x 128) und den Usernamen nebeneinander anzuzeigen.

Zeile 11-18: Es wird eine weitere Tabelle verwendet, um die Blog Posts dieses Users jeweils mit dem kleinen Avatar-Bild (36 x 36) anzuzeigen.

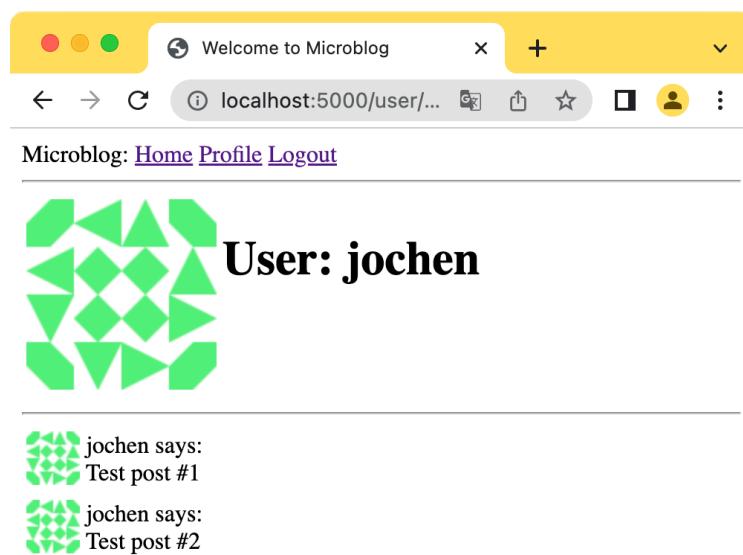


Abbildung 7-8 Zwischenergebnis User-Profilseite

7.4 TEMPLATES INKLUDIEREN

Nun sollen Blog-Posts auf der /index Seite in der gleichen Art wie in der Profilseite angezeigt werden. Anstatt den Template-Code zu kopieren, können Sie die Anzeige der Posts auch in ein separates Unter-Template _posts.html auslagern und es auf der Profilseite und der Indexseite mit include einfügen.

```
1 <!-- app/templates/_posts.html -->
2 <table>
3   <tr valign="top">
4     <td></td>
5     <td>{{ post.author.username }} says:<br>{{ post.body }}</td>
6   </tr>
7 </table>
```

Abbildung 7-10 Untertemplate _posts.html

```
1 <!-- app/template/user.html -->
2 {% extends "base.html" %}
3 {% block content %}
4   <table>
5     <tr valign="top">
6       <td></td>
7       <td><h1>User: {{ user.username }}</h1></td>
8     </tr>
9   </table>
10  <hr>
11  {% for post in posts %}
12    {% include '_posts.html' %}
13  {% endfor %}
14  {% endblock %}
```

Abbildung 7-9 user.html mit include für das Untertemplate

7.5 ZUSÄTZLICHE INFORMATIONEN ZU USER-ACCOUNTS

Zu den Benutzern sollen zwei zusätzliche Informationen angezeigt werden:

1. Ein kurzer Text, mit dem der User sich vorstellt
2. Die Zeit des letzten Besuchs auf der Website

Dafür erweitern Sie die Klasse User um zwei neue Variablen: about_me und last_seen:

```
9 class User(UserMixin, db.Model):
10   id = db.Column(db.Integer, primary_key=True)
11   username = db.Column(db.String(64), index=True, unique=True)
12   email = db.Column(db.String(120), index=True, unique=True)
13   password_hash = db.Column(db.String(128))
14   posts = db.relationship('Post', backref='author', lazy='dynamic')
15   about_me = db.Column(db.String(140))
16   last_seen = db.Column(db.DateTime, default=datetime.utcnow)
```

Abbildung 7-11 Zwei neue Variablen in der Klasse User

Nun muss noch die Datenbanktabelle um die neuen Spalten `about_me` und `last_seen` erweitert werden. Das erfolgt wie in Kapitel 6 in zwei Schritten, der Erzeugung des Migrationsscripts und seine Ausführung:

```
(tutorial) $ flask db migrate -m "neue spalten im user model"
(tutorial) $ flask db upgrade
```

Wenn die Datenbank-Migration gut gelaufen ist, können die beiden Felder zum Template der Profilseite hinzugefügt werden:

```
1  <!-- app/template/user.html -->
2  {% extends "base.html" %}
3  {% block content %}
4      <table>
5          <tr valign="top">
6              <td></td>
7              <td><h1>User: {{ user.username }}</h1>
8              {% if user.about_me %}<p>{{ user.about_me }}</p>{% endif %}
9              {% if user.last_seen %}<p>Last seen on: {{ user.last_seen }}</p>{% endif %}
10             </td>
11         </tr>
12     </table>
13     <hr>
14     {% for post in posts %}
15         {% include '_posts.html' %}
16     {% endfor %}
17 {% endblock %}
```

Abbildung 7-12 Die beiden neuen Felder im Template

In den Zeilen 8 und 9 werden die Feldinhalte `about_me` und `last_seen` nur dann angezeigt, wenn sie auch gesetzt sind. Solange hierfür noch keine Daten erfasst sind werden Sie dort keine Ausgaben sehen.

Zunächst soll das Feld `last_seen` mit der aktuellen Zeit gefüllt werden, und zwar jedes Mal, wenn ein angemeldeter Benutzer einen Request an die App schickt. Hierfür wird der Decorator `@app.before_request` benutzt, der bewirkt, dass die folgende Funktion bei jedem Request ausgeführt wird und zwar bevor die eigentliche View-Funktion aufgerufen wird. Fügen Sie Folgendes zu `app/routes.py` hinzu:

```
7 | from datetime import datetime
8 |
9 | @app.before_request
10| def before_request():
11|     if current_user.is_authenticated:
12|         current_user.last_seen = datetime.utcnow()
13|         db.session.commit()
```

Abbildung 7-13 @app.before_request in app/routes.py

Das `current_user` Objekt wurde beim Login mit der `load_user()` Funktion bereits aus der Datenbank geladen, so dass es hier in Zeile 12 nur noch um den Zeitstempel ergänzt und dann mit

`db.session.commit()` in der Datenbank gespeichert werden muss. Selbstverständlich geschieht dies nur, wenn der Benutzer angemeldet ist, was über `current_user.is_authenticated` abgefragt wird.

Die gewählte Zeitangabe ist hier UTC (Universal Time Coordinated). Würde stattdessen die Ortszeit des Servers genommen, würde das bei internationaler Benutzung zu falschen Anzeigen führen. Die korrekte Anzeige der Ortszeit wird in einem späteren Kapitel behandelt. Testen Sie das, indem Sie flask neu starten und auf die Profilseite gehen:

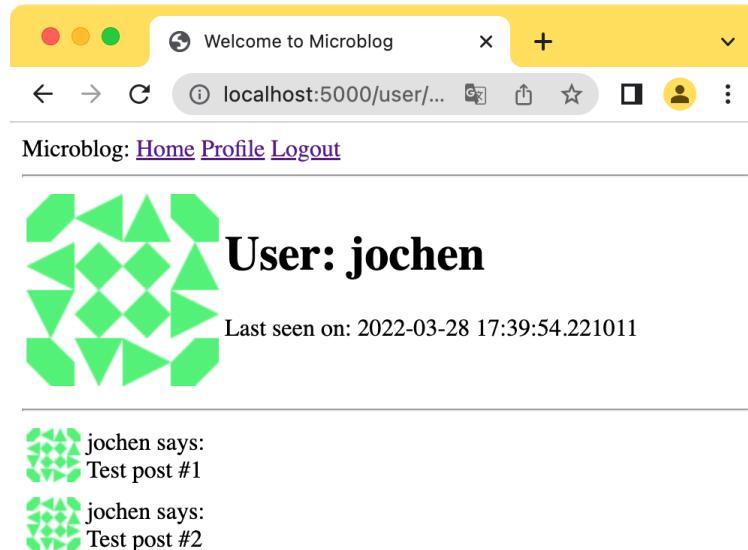


Abbildung 7-14 Anzeige 'Last seen on' im Profil

7.6 DER PROFIL-EDITOR

Um das zweite Feld 'about_me' erfassen zu können wird eine weitere Form-Klasse, ein Template und eine View-Funktion benötigt. Die Form-Klasse soll `EditProfileForm` heißen und wie folgt aussehen:

```
1 | # app/forms.py
2 | from flask_wtf import FlaskForm
3 | from wtforms import StringField, PasswordField, BooleanField, SubmitField, TextAreaField
4 | from wtforms.validators import ValidationError, DataRequired, Email, EqualTo, Length
5 | from app.models import User
6 |
7 | class EditProfileForm(FlaskForm):
8 |     username = StringField('Username', validators=[DataRequired()])
9 |     about_me = TextAreaField('About me', validators=[Length(min=0, max=140)])
10|    submit = SubmitField('Submit')
```

Abbildung 7-15 Die Form-Klasse in `app/forms.py`

Beachten Sie die Importe von `TextAreaField` in Zeile 3 und `Length` in Zeile 4. Ein `TextAreaField` ist für die Eingabe mehrzeiliger Texte geeignet. Das Datenbankfeld `about_me` wurde auf 140 Zeichen beschränkt und diese maximale Textlänge wird mit dem Validator `Length` erzwungen.

Das dazu passende Template ist app/templates/edit_profile.html:

```
1 <!-- app/templates/edit_profile.html -->
2 {% extends "base.html" %}
3 {% block content %}
4     <h1>Edit Profile</h1>
5     <form action="" method="post">
6         {{ form.hidden_tag() }}
7         <p>
8             {{ form.username.label }}<br>
9             {{ form.username(size=32) }}<br>
10            {% for error in form.username.errors %}
11                <span style="color: red;">{{ error }}</span>
12            {% endfor %}
13        </p>
14        <p>
15            {{ form.about_me.label }}<br>
16            {{ form.about_me(cols=50, rows=4) }}<br>
17            {% for error in form.about_me.errors %}
18                <span style="color: red;">{{ error }}</span>
19            {% endfor %}
20        </p>
21        <p>{{ form.submit() }}</p>
22    </form>
23 {% endblock %}
```

Abbildung 7-16 Das Template für den Profil-Editor

Und schliesslich wird noch eine View-Funktion mit dem Namen edit_profile() benötigt:

```
9 # In app/routes.py hinzufügen
10 from app.forms import EditProfileForm
11
12 @app.route('/edit_profile', methods=['GET', 'POST'])
13 @login_required
14 def edit_profile():
15     form = EditProfileForm()
16     if form.validate_on_submit():
17         current_user.username = form.username.data
18         current_user.about_me = form.about_me.data
19         db.session.commit()
20         flash('Your changes have been saved.')
21         return redirect(url_for('edit_profile'))
22     elif request.method == 'GET':
23         form.username.data = current_user.username
24         form.about_me.data = current_user.about_me
25     return render_template('edit_profile.html', title='Edit Profile',
26                           form=form)
```

Abbildung 7-17 Die View-Funktion für den Profil-Editor

Falls validate_on_submit True zurückgibt, werden die Felder username und about_me im current_user Objekt mit den Eingaben aus der Form gefüllt und es wird db.session.commit() aufgerufen um das geänderte Objekt in der Datenbank zu sichern und die Form erneut aufzurufen (Zeilen 16 – 21).

Falls validate_on_submit ()False zurückgibt, kann das an zwei Ursachen liegen:

1. Der Benutzer hat noch nichts gemacht, sondern nur die Route /edit_profile (mit GET) aufgerufen. Dann soll natürlich die Form angezeigt werden, aber der Benutzername und der bestehende Inhalt sollen bereits angezeigt werden (Zeilen 22 - 24)
 2. Der Benutzer hat den Submit-Button angeklickt (POST), aber eine der Validierungen ist fehlgeschlagen. Dann soll die Form erneut geladen werden (Zeile 25).

In base.html wird jetzt noch ein Link hinzugefügt:

```
20     {% if user == current_user %}  
21         <a href="{{ url_for('edit_profile') }}>Edit your profile</a>  
22     {% endif %}
```

Abbildung 7-18 - Link, Falls der User der Agemeldete User ist

Die Anzeigen sehen nun so aus:

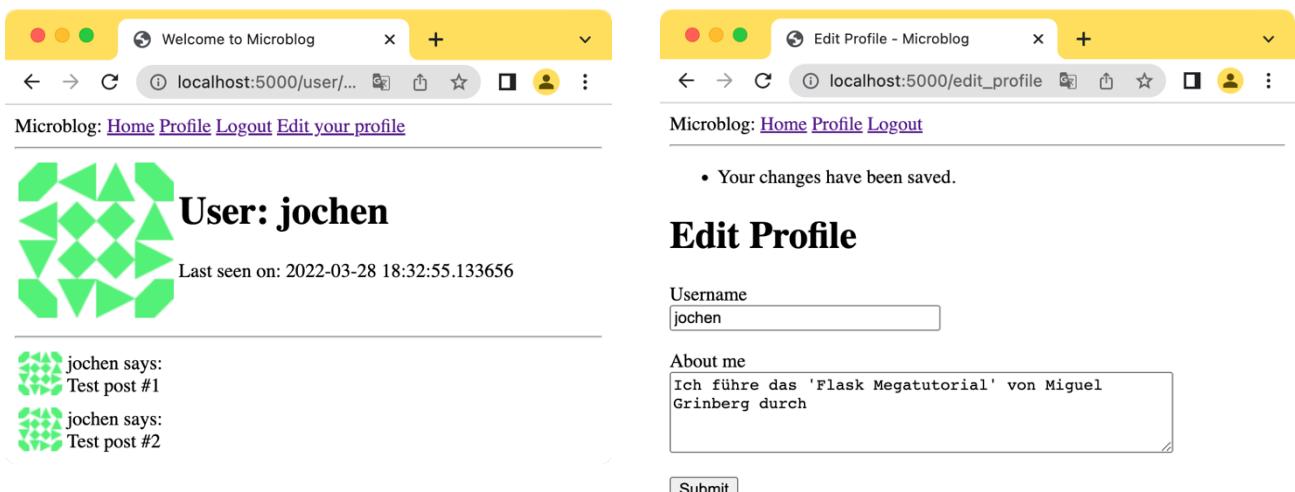


Abbildung 7-19 Die Anzeigen der Profilseite und des Editors

7.7 MUSTERLÖSUNGEN

Die Musterlösungen zu diesem Kapitel sind Bestandteil des Repository

<https://github.com/miquelgrinberg/microblog>

Wenn Sie es wie im Kapitel 3.4 beschrieben geclont haben, geben Sie im Verzeichnis microblog das folgende Kommando ein:

```
$ git checkout v0.6
```

7.8 ZUSAMMENFASSUNG

- Für bestimmte Funktionalitäten können Webservices eine Alternative zur Eigenprogrammierung sein. Am Beispiel Gravatar wurde gezeigt, dass dafür in der eigene App nur an geeigneter Stelle in einem Template ein URL eingebaut werden muss (Ein anderer denkbarer Fall wäre die Anzeige einer Karte für einen Standort mit Google Maps).
- Solche URLs können Query-Strings enthalten, die beeinflussen, welches Ergebnis zurückgeliefert werden. Im Beispiel Gravatar wurde dadurch die Art des Bildes und die Grösse bestimmt.
- In den Beispielen wurde abermals das Objekt `current_user` benutzt, das mit der Funktion `load_user()` und dem Decorator `@login.user_loader` aus der Datenbank gelesen und durch die Erweiterung Flask-Login im Verlauf der User-Session verwaltet wird.
- Die Prüfung, ob ein Benutzer zum Betreten einer Seite berechtigt ist, erfolgt auch hier über das Attribut `current_user.is_authenticated`
- Die Erstellung einer Seite folgt auch bei der Profil- und der Profil-Editor Seite dem Zusammenspiel von Model (hier User), View (Template) und Controller (View-Funktion).
- Zeitangaben sollten bei einer Webapplikation immer als UTC (Universal Time Coordinated) gespeichert werden, da die App potentiell von Benutzern in verschiedenen Zeitzonen genutzt werden kann.

8 FEHLERBEHANDLUNG

Während der Entwicklung und bei der Nutzung einer App treten Fehler auf. Das ist praktisch unvermeidlich. Fehlermeldungen sollten je nach Benutzer (Entwickler oder End-User) und Situation (Entwicklung oder produktiver Betrieb) einen unterschiedlichen Informationsgehalt haben. Den Entwicklern sollten Sie möglichst präzise Informationen über die Fehlerursache liefern, für End-User sollten Sie vor allem 'freundlich' sein.

8.1 LERNZIELE

Nach der Bearbeitung dieses Kapitels können Sie:

- Die Fehlermeldungen der Server-Konsole nutzen, um die Fehlerursache zu finden und zu beheben
- Den Flask-Debugger im Development-Modus nutzen um Fehler auch im Browser analysieren zu können.
- Fehler und andere Informationen in einer Log-Datei auf dem Server speichern
- Fehlerinformationen per E-Mail an den Administrator senden

8.2 FEHLER IN FLASK

Sehr wahrscheinlich haben Sie im Verlauf des Kurses schon mehrmals Fehlermeldungen beim Start des Development-Servers mit *flask run* oder *flask shell* gesehen. Sie erscheinen in der Server-Konsole.

```
$ flask run
...
... Viele Zeilen ausgeblendet ...
File "/Users/jochenreinholdt/microblog/tutorial/lib/python3.9/site-
packages/flask/cli.py", line 260, in locate_app
    __import__(module_name)
File "/Users/jochenreinholdt/microblog/app/__init__.py", line 14, in <module>
    from app import routes, models
File "/Users/jochenreinholdt/microblog/app/routes.py", line 6, in <module>
    from app.forms import LoginForm, RegistrationForm, EditProfileForm
File "/Users/jochenreinholdt/microblog/app/forms.py", line 36, in <module>
    class EditProfileForm(FlaskForm):
File "/Users/jochenreinholdt/microblog/app/forms.py", line 38, in
EditProfileForm
    about_me = TextAreaField('About me', validators=[Length(min=0, max=140)])
NameError: name 'TextAreaField' is not defined
```

Diese Fehlermeldung ist hilfreich. 'TextAreaField' ist nicht bekannt werden und Sie bekommen auch eine genaue Angabe, wo der Fehler zu suchen ist, nämlich in Zeile 38 von *app/forms.py*. In diesem Fall wurde in *app/forms.py* ein Import vergessen, nämlich '*from wtforms import TextAreaField*'.

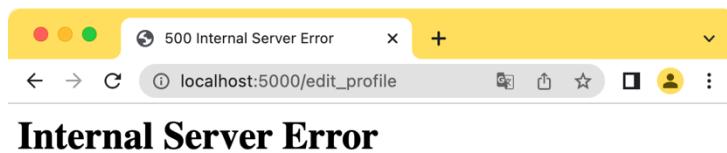
Die Art der Fehlerausgabe ist ein sogenannter *stack trace*. Er kann auch viel mehr Zeilen enthalten. Die wichtigste ist die letzte, denn sie gibt Ihnen die Information, wo der Fehler in Ihrem Code ausgelöst wurde. Die Zeilen darüber enthalten häufig Informationen darüber, was ausgehend von der eigentlichen Fehlerstelle in den vielen anderen Dateien der benutzten Pakete ausgelöst wurde. Also sind diese Meldungen Folgefehler der eigentlichen Fehler. Da das aber nicht Ihr Code ist, tragen diese Informationen aber in der Regel nichts dazu bei, den Fehler zu finden und zu korrigieren.

Merke: Einen *stack trace* lesen Sie am besten von unten nach oben.

Wenn Sie sich bisher an den Code im Tutorial gehalten haben, können Sie einen darin versteckten Fehler reproduzieren. Führen Sie dazu in der App die folgende Schritte aus:

1. Registrieren Sie zwei Benutzer 'jochen' und 'paula'.
2. Melden Sie sich als 'jochen' an und navigieren Sie zu Profile / Edit Profile
3. Ändern Sie den Benutzernamen 'jochen' in 'paula'
4. Klicken Sie den Submit-Button

Das Resultat:



Im Konsolen-Fenster, in dem Sie den Server mit flask run aufgerufen haben wird der stack trace angezeigt:

Traceback (most recent call last):

... Viele Zeilen ausgeblendet ...

```
File "/Users/jochenreinholdt/microblog/tutorial/lib/python3.9/site-packages/sqlalchemy/engine/base.py", line 1808, in _execute_context
    self.dialect.do_execute(
File "/Users/jochenreinholdt/microblog/tutorial/lib/python3.9/site-packages/sqlalchemy/engine/default.py", line 732, in do_execute
    cursor.execute(statement, parameters)
sqlalchemy.exc.IntegrityError: (sqlite3.IntegrityError) UNIQUE constraint failed: user.username
[SQL: UPDATE user SET username=?, about_me=? WHERE user.id = ?]
[parameters: ('paula', '', 1)]
(Background on this error at: https://sqlalche.me/e/14/gkpj)
127.0.0.1 -- [30/Mar/2022 12:17:06] "POST /edit_profile HTTP/1.1" 500 -
```

Die untersten Zeilen liefern den entscheidenden Hinweis: 'UNIQUE constraint failed: user.username ...'. In der Definition des Usernamens im Model User wurde für das Feld username festgelegt, dass dieser eindeutig sein muss und Flask-Migrate hat dementsprechend für die Spalte username in der Tabelle user einen UNIQUE-Constraint angelegt. Der Profil-Editor in der jetzigen Version erlaubt aber, einen beliebigen Usernamen einzutragen. Es wird dabei nicht geprüft, ob der Name schon vergeben ist und dieser Fehler schlägt bis in die Datenbank durch.

Die Fehlermeldung im Browser sagt nichts darüber aus. Es wäre auch nicht gut, wenn ein normaler Benutzer die Details aus dem stack trace sehen würde. Allerdings ist die Meldung nicht sehr benutzerfreundlich und die Anzeige im Browser weicht von den anderen Seiten ab. Zu mindestens ein Link 'Zurück' oder ähnlich wäre hier angebracht.

Während der Entwicklung wäre es aber nicht schlecht, wenn die Fehlerdetails für den Entwickler auch im Browser sichtbar wären.

8.3 DEBUG-MODUS

Machen Sie bitte Folgendes:

1. Stoppen Sie den Server

2. Setzen Sie die folgende Variable:

```
(tutorial) $ export FLASK_ENV=development # Windows: set FLASK_ENV=development
```

3. Starten Sie den Server neu. Die Ausgabe von flask run sieht jetzt anders aus:

```
(tutorial) $ flask run
 * Serving Flask app 'microblog.py' (lazy loading)
 * Environment: development
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 685-266-988
```

4. Wiederholen Sie die Schritte aus dem letzten Abschnitt, die zum Fehler geführt haben.

Die folgende Seite wird im Browser angezeigt:

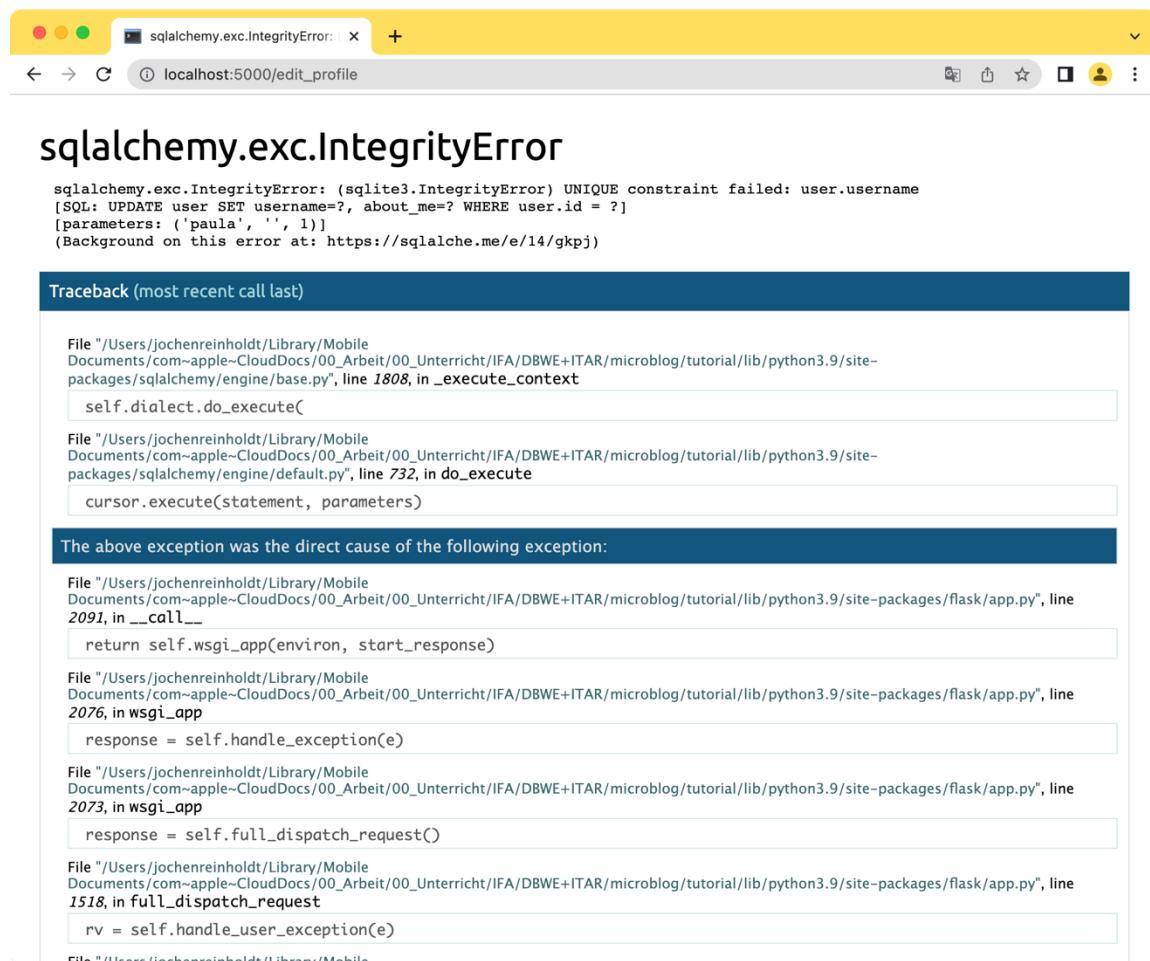


Abbildung 8-1 Anzeige des Flask Debugger im Browser

Sehen Sie sich die einzelnen Bereiche dieser Seite an. Die eingerahmten Felder können sie mit einem Klick erweitern und sich den Code im Detail ansehen.

In der Code-Anzeige können Sie auf das Terminal-Symbol klicken und können dann die PIN eingeben, die in der Ausgabe von `flask run` angezeigt wurde:

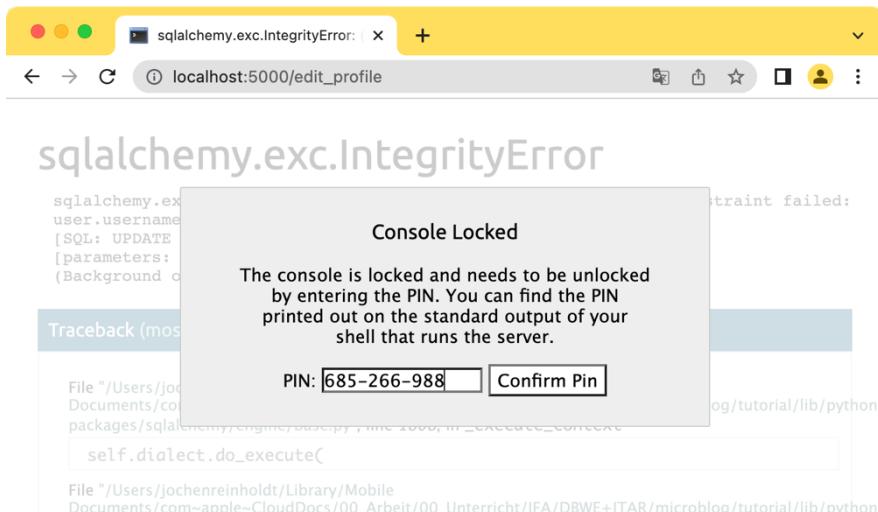


Abbildung 8-2 Eingabe der PIN aus der Ausgabe von `flask run` im Debug-Modus

Nach Eingabe der korrekten PIN bekommen Sie die Möglichkeit, interaktiv beliebigen Python-Code wie in der REPL einzugeben:

```
[console ready]
>>>
```

Abbildung 8-3 Beliebiger Code kann am Prompt eingegeben werden

Diese Möglichkeiten sind natürlich nur für Entwickler sinnvoll. Eine in der Produktion bereitgestellte App sollte niemals im Debug-Modus gestartet werden, denn die Informationen sind für Endbenutzer sinnlos und für potentielle Angreifer im Web enthalten Sie unter Umständen wertvolle Hinweise!

8.4 EIGENE FEHLER-SEITEN

Für die Standard-Fehlercodes 404 (Seite nicht gefunden) und 500 (Interner Server-Fehler) sollen nun eigene Fehler-Seiten erzeugt werden. Das hat den Vorteil, dass die Fehlermeldung speziell auf die App zugeschnitten werden kann, dass das 'Look and Feel' der Fehlerseiten an den Rest der App angepasst werden kann und dass auf den Fehlerseiten ein Link zurück zur App angezeigt werden kann.

Ausserdem wird beim 500-er Fehler im Beispiel für den doppelten Benutzernamen weiter oben ein Datenbank-Fehler ausgelöst. Für Datenbank-Fehler sollte vorsichtshalber grundsätzlich ein ROLLBACK der letzten Änderung erfolgen.

Dafür machen Sie folgende Schritte:

- Sie erzeugen ein neues Modul app/errors.py mit zwei View-Funktionen für die Fehlercodes 404 und den 500.
- Sie erzeugen zwei neue Templates für diese Routen in app/templates
- Sie importieren und initialisieren einen Error-Handler in app/__init__.py

1. Erzeugen Sie die beiden neuen Template-Dateien:

```

1  <!-- app/templates/404.html -->
2  {% extends "base.html" %}
3  {% block content %}
4      <h1>File Not Found</h1>
5      <p><a href="{{ url_for('index') }}>Back</a></p>
6  {% endblock %}

```

Abbildung 8-4 Template für die 404-Seite

```

1  <!-- app/templates/500.html-->
2  {% extends "base.html" %}
3  {% block content %}
4      <h1>An unexpected error has occurred</h1>
5      <p>The administrator has been notified. Sorry for the inconvenience!</p>
6      <p><a href="{{ url_for('index') }}>Back</a></p>
7  {% endblock %}

```

Abbildung 8-5 Template für die 505-Seite

Beide erweitern base.html, so dass alle Elemente von dort übernommen werden und die Seiten die gleiche Gestaltung haben wie die übrigen Seiten. Ein Link mit der Beschriftung 'Back' wird unter der Fehlermeldung angezeigt.

2. Erzeugen Sie die Datei app/errors.py und schreiben Sie zwei View-Funktionen:

```

1  # app/errors.py
2  from flask import render_template
3  from app import app, db
4
5
6  @app.errorhandler(404)
7  def not_found_error(error):
8      return render_template('404.html'), 404
9
10
11 @app.errorhandler(500)
12 def internal_error(error):
13     db.session.rollback()
14     return render_template('500.html'), 500

```

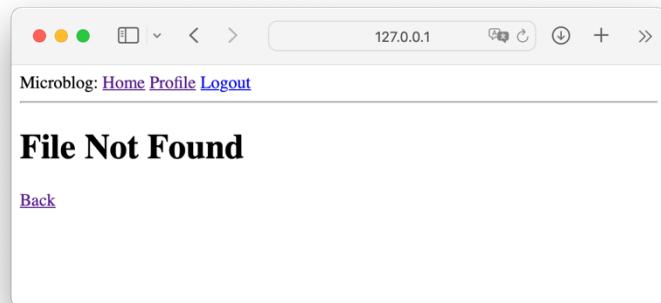
Abbildung 8-6 Die neuen View-Funktionen

Neu ist der Decorator `app.errorhandler(Fehlercode)` in den Zeilen 6 und 11. Er sorgt dafür, dass die folgende Funktion aufgerufen wird, falls der Fehlercode auftritt. In Zeile 13 werden vorsichtshalber die offenen Datenbankänderungen mit `db.session.rollback()` verworfen.

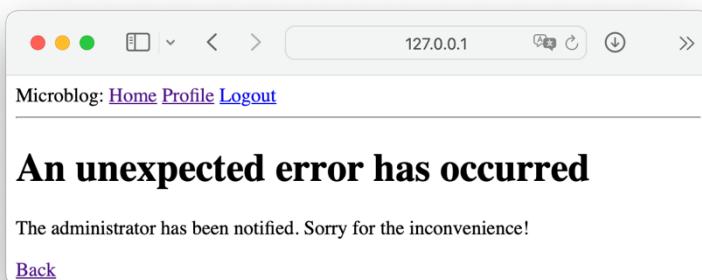
3. Importieren Sie am Ende von `app/__init__.py` zusätzlich das neue errors-Modul:

```
14 |     from app import routes, models, errors
```

4. Um die beiden Fehlerseiten zu testen, muss der Debug-Modus abgeschaltet werden. Setzen Sie in Ihrer Shell `FLASK_ENV=production`, bevor Sie das tun.
5. Testen Sie die 404-Seite, indem Sie die nicht existierende URL-Route `localhost:5000/gibtsnicht` eingeben



6. Testen Sie die 500-Seite, indem Sie die Schritte zur Reproduktion des duplizierten Usernamens wiederholen (Profil bearbeiten, anderen schon existierenden Usernamen angeben)



8.5 LOGGING

Als Administrator einer Website schauen Sie weder ständig die Konsolen-Ausgaben auf Ihrem Server an und die Fehlermeldungen, die Sie für Ihre Endbenutzer ausgeben sind mit Absicht nicht sehr detailliert. Deshalb nutzt Flask das logging-Package von Python¹², mit dem Sie Meldungen aller Art in Logdateien schreiben oder per E-Mail versenden können.

8.5.1 FEHLER IN LOGFILE SCHREIBEN

In app/__init__.py fügen Sie folgende Inhalte hinzu:

```
6 import logging
7 from logging.handlers import RotatingFileHandler
8 import os

18 if not app.debug:
19     if not os.path.exists('logs'):
20         os.mkdir('logs')
21     file_handler = RotatingFileHandler('logs/microblog.log', maxBytes=10240,
22                                         backupCount=10)
23     file_handler.setFormatter(logging.Formatter(
24         '%(asctime)s %(levelname)s: %(message)s [in %(pathname)s:%(lineno)d]'))
25     file_handler.setLevel(logging.INFO)
26     app.logger.addHandler(file_handler)
27     app.logger.setLevel(logging.INFO)
28     app.logger.info('Microblog startup')
29
30 from app import routes, models, errors
```

Abbildung 8-7 Logfile-Konfiguration in app/__init__.py

- Zeilen 6 und 7: Das Package *logging* aus der Standardbibliothek von Python wird importiert. Dort gibt es verschiedene *handlers*. Logfiles sollen nicht beliebig wachsen und *RotatingFileHandler* rotiert. Logfiles, nachdem eine bestimmte Grösse erreicht wurde. Die Grösse kann von Ihnen festgelegt werden und die Anzahl der auf dem Server aufbewahrten Kopien älterer Logfiles ebenso.
- Zeile 8: Das Modul os aus der Standardbibliothek wird benötigt, da auf Verzeichnisse und Dateien im Betriebssystem zugegriffen werden soll.
- Zeile 18: Die folgenden Zeilen werden nur ausgeführt, falls die Umgebungsvariable FLASK_ENV nicht auf development gesetzt ist. Das wird über das Attribut app.debug abgefragt.
- Zeilen 19 - 20: Falls im Projektverzeichnis noch kein Unterverzeichnis logs existiert, wird es angelegt.
- Zeilen 21 - 22: Die maximale Grösse für das Logfile wird auf 10Kbyte festgelegt und es werden nach einer Log-Rotation nur die neuesten 10 Kopien aufbewahrt.
- Zeilen 23 - 24: Inhalt und Format eines Eintrags im Logfile wird festgelegt.

¹² <https://docs.python.org/3/howto/logging.html>

- Zeilen 25 und 27: Im Python-Logging gibt es verschiedene Level, die steuern, wann ein Logeintrag erzeugt wird. Hier wird das Level INFO für file_handler und app.logging gesetzt.

Level	Numeric value
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

- Zeile 26: Das in Zeile 21 erzeugte file_handler-Objekt wird dem einbauten logging hinzugefügt.
- Zeile 28: Eine erste Meldung wird ins Logfile geschrieben

Stellen Sie sicher, dass die Umgebungsvariable FLASK_ENV=production oder leer ist, und provozieren Sie den Fehler mit dem doppelten Usernamen bitte erneut. Im Logfile logs/microblog.log sollten nun einige Zeilen stehen. Die Meldung 'Microblog startup' aus Zeile 28 in app/__init__.py und die Fehlermeldung haben einen Zeitstempel, die Angabe des Levels und den Meldungstext. Anschliessend wurde der gesamte stack trace, den Sie auch auf der Konsole sehen, in das Logfile geschrieben:

```
2022-04-03 11:02:17,682 INFO: Microblog startup [in
/Users/microblog/app/__init__.py:28]
2022-04-03 11:02:52,452 ERROR: Exception on /edit_profile [POST] [in
/Users/microblog/tutorial/lib/python3.9/site-packages/flask/app.py:1457]

Traceback (most recent call last):
sqlite3.IntegrityError: UNIQUE constraint failed: user.username
The above exception was the direct cause of the following exception:

Traceback (most recent call last):
[... diverse Zeilen weggelassen ...]

sqlalchemy.exc.IntegrityError: (sqlite3.IntegrityError) UNIQUE constraint
failed: user.username
[SQL: UPDATE user SET username=? , about_me=? WHERE user.id = ?]
[parameters: ('paula', '', 1)]
(Background on this error at: https://sqlalche.me/e/14/gkpj)
```

8.5.2 FEHLERMELDUNGEN ALS E-MAIL

Als Administrator Ihrer Website möchten Sie vielleicht Fehlermeldungen per E-Mail erhalten. Mit der Erweiterung Flask-Mail können Sie Emails verschicken, aber das Python logging Package hat bereits einen SMTPHandler, den Sie Nutzen können.

Die Konfiguration erfolgt in config.py

```
1 import os
2 basedir = os.path.abspath(os.path.dirname(__file__))
3
4
5 class Config(object):
6     SECRET_KEY = os.environ.get('SECRET_KEY') or 'you-will-never-guess'
7     SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
8         'sqlite:///{} + os.path.join(basedir, 'app.db')
9     SQLALCHEMY_TRACK_MODIFICATIONS = False
10    MAIL_SERVER = os.environ.get('MAIL_SERVER')
11    MAIL_PORT = int(os.environ.get('MAIL_PORT') or 25)
12    MAIL_USE_TLS = os.environ.get('MAIL_USE_TLS') is not None
13    MAIL_USERNAME = os.environ.get('MAIL_USERNAME')
14    MAIL_PASSWORD = os.environ.get('MAIL_PASSWORD')
15    MAIL_DEFAULT_SENDER = os.environ.get('MAIL_DEFAULT_SENDER')
16    ADMINS = ['jochen.reinholdt@dozent.ipso.ch']
```

Abbildung 8-8 Mail-Konfiguration in config.py

Da die meisten Mail-Server keine Emails von unbekannten und nicht als sicher angesehenen Clients entgegennehmen und auch Google ab Mai 2022 die Möglichkeit 'allow less secure clients' für einen Gmail-Account nicht mehr anbietet, müssen Sie auf einen externen E-Mail Dienst wie SendGrid zugreifen, falls Sie die folgenden Schritte ausprobieren möchten. Gehen Sie bei Interesse zu:

<https://sendgrid.com> und registrieren Sie sich dort für einen kostenlosen limitierten Test-Account. Eine Beschreibung, wie sie aus einer Flask App mit SendGrid emails verschicken, finden Sie hier:
<https://sendgrid.com/blog/sending-emails-from-python-flask-applications-with-twilio-sendgrid/>

In meinem persönlichen Versuch mit SendGrid habe ich zunächst einige Einträge in config.py gemacht (Zeilen 10 - 17):

Da hier auf diverse Umgebungsvariablen zugegriffen wird, sind diese in meiner Shell gesetzt:

```
export MAIL_SERVER=smtp.sendgrid.net
export MAIL_PORT=587
export MAIL_USE_TLS=1
export MAIL_USERNAME=apikey
export SENDGRID_API_KEY=<Mein Sendgrid API-Key>
export MAIL_PASSWORD==<Mein Sendgrid API-Key>
export MAIL_DEFAULT_SENDER=jochen.reinholdt@dozent.ipso.ch
```

Nun muss die Klasse SMTPHandler in __init__.py importiert und eine Instanz davon zum Logger der App hinzugefügt werden

```
8 | from logging.handlers import SMTPHandler
```

- Genau wie beim Logging in das Logfile sollen Emails nur verschickt werden, wenn FLASK_ENV nicht auf 'development' gesetzt ist:

```
19 | if not app.debug:
```

- Wenn die Variable MAIL_SERVER überhaupt gesetzt ist (Zeile 31), können MAIL_USERNAME und MAIL_PASSWORD verwendet werden

```
31 |     if app.config['MAIL_SERVER']:
32 |         auth = None
33 |         if app.config['MAIL_USERNAME'] or app.config['MAIL_PASSWORD']:
34 |             auth = (app.config['MAIL_USERNAME'], app.config['MAIL_PASSWORD'])
35 |         secure = None
36 |         if app.config['MAIL_USE_TLS']:
37 |             secure = ()
38 |             mail_handler = SMTPHandler(
39 |                 mailhost=(app.config['MAIL_SERVER'], app.config['MAIL_PORT']),
40 |                 fromaddr=app.config['MAIL_DEFAULT_SENDER'],
41 |                 toaddrs=app.config['ADMINS'], subject='Microblog Failure',
42 |                 credentials=auth, secure=secure)
43 |             mail_handler.setLevel(logging.ERROR)
44 |             app.logger.addHandler(mail_handler)
```

- Zeile 36: Wenn die Variable MAIL_USE_TLS gesetzt ist, kann die Konfiguration vervollständigt und der SMTPHandler mithilfe der anderen Variablen erzeugt werden.
- Zeile 43: Nur bei Ereignissen ab der Schwere ERROR sollen E-Mails verschickt werden.
- Zeile 44: Der mail_handler wird zu app.logger hinzugefügt.

8.6 FEHLERBEHEBUNG

Nachdem der Bug mit dem duplizierten Usernamen sehr nützlich war, um diverse Möglichkeiten zur Fehlerdarstellung und -Protokollierung auszuprobieren, soll er nun behoben werden.

Er kann auftreten, weil bei einer Änderung des Usernamens im Profil-Editor keine Validierung stattfindet, die prüft, ob der Name schon in der Datenbank vergeben ist. In der Registrierung gibt es eine ähnliche Validierung, aber hier besteht ein kleiner Unterschied: Falls der Benutzer den bestehenden Namen nicht ändert, braucht nichts geprüft werden. Nur wenn er ihn ändert, muss die Datenbank abgefragt werden.

```
35  class EditProfileForm(FlaskForm):
36      username = StringField('Username', validators=[DataRequired()])
37      about_me = TextAreaField('About me', validators=[Length(min=0, max=140)])
38      submit = SubmitField('Submit')
39
40      def __init__(self, original_username, *args, **kwargs):
41          super(EditProfileForm, self).__init__(*args, **kwargs)
42          self.original_username = original_username
43
44      def validate_username(self, username):
45          if username.data != self.original_username:
46              user = User.query.filter_by(username=self.username.data).first()
47              if user is not None:
48                  raise ValidationError('Please use a different username.')
Abbildung 8-9 Validierung des Usernamens in der Form
```

Der bestehenden EditProfileForm (Zeilen 35 bis 38) werden zwei Methoden hinzugefügt.

- Zeile 40 - 42: Die Methode `__init__()` der Basisklasse `FlaskForm` wird überladen. Bei dieser Version wird als zusätzliches Argument der Username hinzugefügt. In Zeile 41 wird der Original-Konstruktor der Basisklasse mit `super().__init__()` aufgerufen und dann in einer Variable `original_username` der bestehende Username gespeichert.
- Zeilen 44 - 48: Ein eigener Validator wird implementiert, der prüft ob der username überhaupt geändert wurde. Nur dann muss in der Datenbank nachgesehen werden, ob der Name schon verwendet wurde. Falls ja, wird mit `raise` ein `ValidationError` erzeugt, der im Formular zur Anzeige einer Fehlermeldung führt.

Damit nun der geänderte Konstruktor aufgerufen wird, wird in `app/routes.py` in der View-Funktion `edit_profile()` die Zeile 87 geändert:

```
84  @app.route('/edit_profile', methods=['GET', 'POST'])
85  @login_required
86  def edit_profile():
87      form = EditProfileForm(current_user.username)
Abbildung 8-10 Anpassung der View-Funktion
```

Damit ist der Bug behoben.

8.7 MUSTERLÖSUNGEN

Die Musterlösungen zu diesem Kapitel sind Bestandteil des Repository

<https://github.com/miguelgrinberg/microblog>

Wenn Sie es wie im Kapitel 3.4 beschrieben geklont haben, geben Sie im Verzeichnis microblog das folgende Kommando ein:

```
$ git checkout v0.7
```

8.8 ZUSAMMENFASSUNG

- Für http-Responses mit Fehlercode können eigene Fehlerseiten erstellt werden. View-Funktionen für Fehlercodes können mit dem Decorator `@app.errorhandler(Fehlercode)` registriert werden. Falls eine Funktion für einen bestimmten Fehler registriert wurde, wird diese aufgerufen und sie kann dann das entsprechende Template rendern. Falls nicht, gibt der Server die Standardseite für diesen Fehler zurück.
- Neben dem Wunsch nach benutzerfreundlichen Meldungen und einer einheitlichen Gestaltung der Seiten auch bei Fehlern ist es bei Fehlercodes der 500-Serie immer möglich, dass Änderungen der Datenbank offen geblieben sind. Deshalb sollte in solchen Fällen vorsichtshalber ein ROLLBACK erfolgen.
- Während der Entwicklung kann die Umgebungsvariable `FLASK_ENV=development` gesetzt werden. Dies aktiviert die Anzeige des Flask Debuggers im Browser. Zuätzlich ist es in diesem Modus möglich, dass Code-Änderungen sofort geladen werden, ohne dass der Development-Server neu gestartet werden muss.
- Um detaillierte Angaben zu Fehlern zu bekommen, stehen neben den Ausgaben auf der Konsole und dem Debugger das Modul `logging` aus der Standardbibliothek zur Verfügung. Es ermöglicht, verschiedene *logging handlers* zu registrieren. In diesem Kapitel wurden die beiden Handler `RotatingFileHandler` und `SMTPHandler` vorgestellt.
- Für solche Fehler-Handler können sechs verschiedene Schweregrade festgelegt werden.

9 FOLLOWER

User können die Beiträge anderer User abonnieren. Bei Blogs hat sich dafür der Ausdruck Follower etabliert. In diesem Kapitel wird der Entwurf und die Implementierung der Follower gezeigt.

9.1 LERNZIELE

Nach der Bearbeitung des Kapitels können Sie

- Die Beziehung 'follows' in der Datenbank entwerfen
- Die entsprechenden Model-Klassen implementieren
- Die neue Datenbank-Struktur mit Flask-Migrate erzeugen
- Join-Queries in Flask implementieren
- Das Paket unittest verwenden, um automatisierte Tests zu programmieren

9.2 BEZIEHUNGEN IM DATENMODELL

Das Modell der Beziehungen 'follows' und 'followed by' ist eine Viele-zu-Viele Beziehung zwischen Usern:

- Ein User kann 'Follower' von beliebig vielen Usern sein
- Ein User kann beliebig viele Follower haben

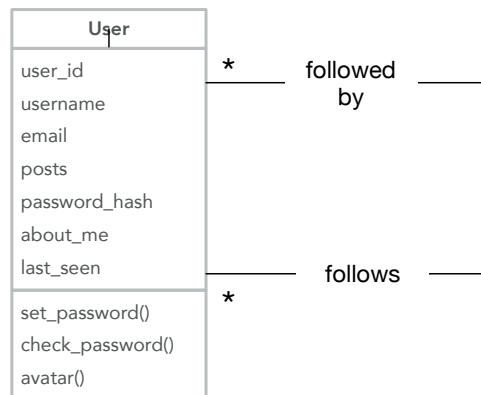


Abbildung 9-1 Follower-Beziehungen (Klassenmodell)

In einer relationalen Datenbank werden solche Viele-zu-Viele Beziehungen mithilfe einer zusätzlichen Beziehungs-Tabelle modelliert.

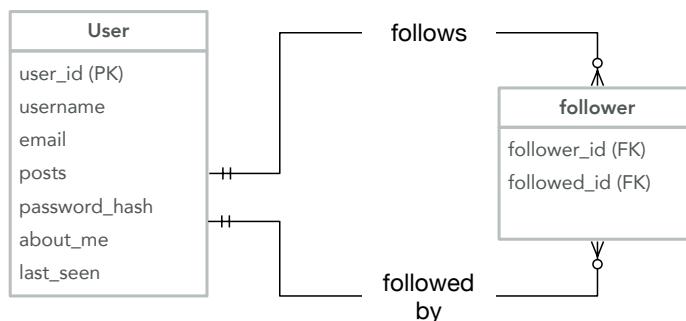


Abbildung 9-2 Follower im Entity-Relationship Diagramm

Wegen der durch das relationale Modell bedingten Einschränkung, dass Datenbankspalten nur einfache Werte und keine Listen enthalten können, werden die Fremdschlüssel-Spalten ('FK' im Diagramm) in die zusätzliche Tabelle follower verschoben.

Da Follower keine eigenständige Entität im Datenmodell ist und die Tabelle follower nur zur Abbildung der Beziehung in der Datenbank benötigt wird, wird keine Klasse 'Follower' angelegt und Flask-Migrate kommt in diesem Fall nicht zum Einsatz. Die Tabelle wird stattdessen direkt mit SQLAlchemy erzeugt. In app/models.py kommen folgende Einträge hinzu:

```

8   followers = db.Table(
9     'followers',
10    db.Column('follower_id', db.Integer, db.ForeignKey('user.id')),
11    db.Column('followed_id', db.Integer, db.ForeignKey('user.id'))
12 )

```

Abbildung 9-3 followers-Tabelle in app/models.py

Der Klasse User wird ein Attribut vom Typ db.relationship hinzugefügt:

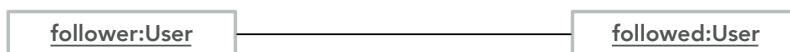
```

15 class User(UserMixin, db.Model):
16     # ... Bestehende Attribute ...
17
18     followed = db.relationship(
19         'User', secondary=followers,
20         primaryjoin=(followers.c.follower_id == id),
21         secondaryjoin=(followers.c.followed_id == id),
22         backref=db.backref('followers', lazy='dynamic'), lazy='dynamic')
23
24
25
26
27

```

Abbildung 9-4 Erweiterung der Klasse User

- Zeile 23-24: Bei einer Abfrage der Follower eines Users erwartet man, eine Liste von User-Objekten zu bekommen. Mit db.relationship() wird die Beziehung spezifiziert. Dabei geht man von jeweils zwei User-Objekten aus. Der User 'follower' (links) folgt einem anderen User, 'followed' (rechts):



Die Frage '**Welchen Usern folge ich?**' aus der Perspektive des linken Users (follower) kann mithilfe der Tabelle followers beantwortet werden. Das wird durch secondary=followers festgelegt.

- Zeile 25: Mit primaryjoin () wird festgelegt, über welche Fremdschlüssel die Beziehung von links nach rechts geregelt werden soll. Das ist die follower_id.
- Zeile 26: Mit secondaryjoin() wird festgelegt, über welche Fremdschlüssel die Beziehung von rechts nach links geregelt werden soll. Das ist die followed_id.
- Zeile 27: Mit backref() wird festgelegt, wie aus der Perspektive des rechten Users (followed) her die Frage '**Welche User folgen mir?**' beantwortet werden kann, nämlich ebenfalls über followers. Die Angabe lazy=dynamic beeinflusst, wann die Beziehung in der Datenbank abgefragt

wird. Bei 'dynamic' wird eine Datenbankabfrage erst dann ausgeführt, wenn das Ergebnis gebraucht wird. Das wird für beide Abfrage-Richtungen spezifiziert.

Nachdem Sie diese Änderungen in app/models.py gemacht haben, führen Sie eine weitere Datenbank-Migration aus:

```
(tutorial) $ flask db migrate  
(tutorial) $ flask db upgrade
```

9.3 FOLLOWER HINZUFÜGEN UND ENTFERNEN

Das ORM über SQLAlchemy vereinfacht den Zugriff auf Follower sehr stark. Das Attribut *followed* eines User-Objekts aus dem vorhergehenden Abschnitt ist eine Liste von User-Objekten. Anstatt mit SQL die Follower aus der Datenbank zu selektieren, in einer Liste zu speichern und darauf zugreifen können Sie einfach mit den Standard-Methoden *followed.append()* hinzugefügt und mit *followed.remove()* aus der Liste und der Datenbank entfernt werden. Diese Operationen werden in den neuen Methoden *User.follow()* und *User.unfollow()* in der Klasse User implementiert.

```
44     def follow(self, user):  
45         if not self.is_following(user):  
46             self.followed.append(user)  
47  
48     def unfollow(self, user):  
49         if self.is_following(user):  
50             self.followed.remove(user)  
51  
52     def is_following(self, user):  
53         return self.followed.filter(  
54             followers.c.followed_id == user.id).count() > 0
```

Abbildung 9-5 Methoden für Follower in der Klasse User

Die Methode *is_following()* führt eine Datenbankabfrage durch, um zu ermitteln, ob zwischen zwei User-Objekten bereits eine Verbindung existiert. Sie haben in Kapitel bereits die *filter_by()* methode von SQLAlchemy kennengelernt. Sie funktionierte ähnlich wie eine WHERE-Bedingung mit Attribut = Wert in einer SELECT-Anweisung.

Die Methode *filter()* in Zeile 53 - 54 kann beliebige Bedingungen auf eine Datenbankabfrage anwenden. Mit *tabelle.c.spaltenname* kann eine bestimmte Spalte angegeben werden. Die Abfrage sucht also nach Einträgen in der tabelle follower, bei denen die Werte in der Spalte *is_followed* gleich der *user_id* des User-Objekts sind, das der Methode als Argument übergeben wurde. Abgeschlossen wird die Abfrage mit *count()*, zählt also die Zeilen, auf die die Filterbedingung zutrifft. Wenn die Abfrage ein Ergebnis grösser 0 liefert, wird True zurückgegeben.

9.4 POSTS VON FOLLOWERN

Was in der Klasse User noch fehlt, ist eine Abfrage der Blog-Posts und zwar der eigenen Blog-Posts und der Posts von anderen Usern, denen ein User folgt. Aus Datenbank-Sicht erfordert das einen JOIN über die Tabellen user, follower und post.

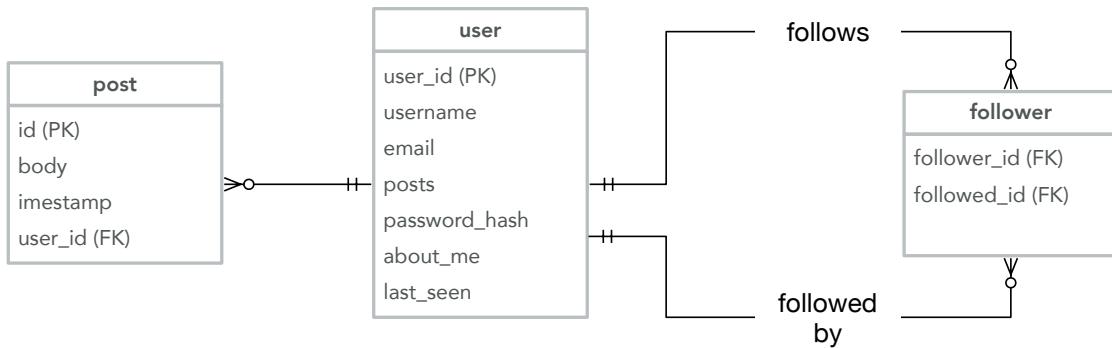


Abbildung 9-6 Das komplette Datenmodell

In SQL würden Sie für die eigenen Posts folgende Abfrage schreiben:

```

SELECT p.body, p.timestamp
WHERE p.user_id = <Die eigene user_id>
  
```

Für die Posts der User, denen Sie folgen, könnte die Abfrage lauten:

```

SELECT p.body, p.timestamp
FROM posts p JOIN follower f ON p.user_id = f.followed_id
WHERE f.follower_id = <Die eigene user_id>
  
```

Und für beide zusammen, sortiert nach dem Timestamp:

```

SELECT p.body, p.timestamp
FROM posts p JOIN user u ON p.user_id = u.user_id
WHERE u.user_id = <Die eigene user_id>
UNION
SELECT p.body, p.timestamp
WHERE p.user_id = <Die eigene user_id>
ORDER BY p.timestamp
  
```

Die Operationen JOIN, UNION und ORDER BY stehen auch über SQLAlchemy zur Verfügung. Die Abfrage aller eigenen und der verfolgten Posts kann in der Klasse User folgt geschrieben werden:

```

56  def followed_posts(self):
57      followed = Post.query.join(
58          followers, (followers.c.followed_id == Post.user_id)).filter(
59              followers.c.follower_id == self.id)
60      own = Post.query.filter_by(user_id=self.id)
61      return followed.union(own).order_by(Post.timestamp.desc())
  
```

Abbildung 9-7 Klasse User: JOIN, UNION und ORDER BY mit SQLAlchemy

- Zeile 57-59: Die Methode Objekt.query.join verbindet zwei Tabellen (hier: followers und post). Dabei werden als Schlüssel die followed_id aus followers und die user_id aus posts verwendet (entspricht einer ON-Klausel für den JOIN in SQL). Mit `filter(followers.c.follower_id == self.id)` wird

eine WHERE-Bedingung hinzugefügt, denn es sind ja nur die Posts von solchen Usern von Interesse, denen der eigene User folgt. Die Ergebnisliste bekommt den Namen followed.

- Zeile 60: Die eigenen posts werden selektiert. Die Ergebnisliste bekommt den Namen own.
- Zeile 61: Mit `followed.union(own)` werden die beiden Ergebnislisten zusammengefügt und mit `order_by()` nach der Spalte timestamp sortiert. Die Ergebnisliste wird mit return zurückgegeben.

Fügen Sie bitte die Methode `followed_posts(self)` in die Klasse User ein.

9.5 UNIT TESTS

Bisher haben Sie Ihre App manuell getestet und es gibt allein in der Klasse User genügend Dinge, die man testen müsste. Sie hat nun einen gewissen Komplexitätsgrad erreicht und mit jedem neuen Attribut und mit jeder neuen Methode steigt die Wahrscheinlichkeit, dass sich Fehler einschleichen.

Mit Unit-Tests kann man Tests automatisieren. Ein Unit-Test ist ein Programm, das ein Modul oder einen Teil davon systematisch testet und die Testergebnisse anzeigt. Es besteht aus mehreren Teilen:

- **Das Setup.** In diesem Teil werden die Voraussetzungen für einen wiederholbaren und vergleichbaren Test geschaffen. Welche Voraussetzungen das sind, hängt vom Testgegenstand ab. Beispiele sind: Variablen, die auf bestimmte Werte gesetzt sind, bestimmte Datenbankinhalte, Dateien und Verzeichnisse, die vorhanden sein müssen und die einen bestimmten Inhalt haben müssen, Software-Versionen, die installiert sein müssen und Software, die gestartet sein muss. Die Idee dabei ist, dass jeder einzelne Testlauf die gleiche Ausgangsbasis hat.
- **Die Test Cases.** Sie führen die zu testende Softwarekomponente aus und vergleichen das tatsächliche Resultat mit dem erwarteten Resultat. Das Ergebnis jedes einzelnen Test Cases wird protokolliert.
- **Der Teardown.** Nach einem Testlauf sollen die durch Setup und die Test Cases erfolgten Änderungen am System wieder zurückgesetzt werden, damit für den nächsten Test wieder alles in einem neutralen Ausgangszustand ist.

Es gibt für Python die beiden Pakete unittest und pytest, die Sie benutzen können, um automatisierte Tests zu schreiben. In diesem Kapitel wird unittest aus der Standardbibliothek verwendet.

Das Paket stellt die Klasse `unittest.TestCase` zur Verfügung, für die Sie die Methoden `setUp()` und `tearDown()` entsprechend Ihrem speziellen Testvorhaben implementieren müssen. Diese Methoden werden bei der Ausführung automatisch zu Beginn und am Ende eines Testlaufs aufgerufen.

Ausserdem erhalten Sie eine Sammlung von assert-Methoden, die die Überprüfung und Protokollierung von Ergebnissen vereinfachen. Die wichtigsten davon sind:

Methode	Überprüft die Bedingung
<code>.assertEqual(a, b)</code>	<code>a == b</code>
<code>.assertTrue(x)</code>	<code>bool(x) is True</code>
<code>.assertFalse(x)</code>	<code>bool(x) is False</code>
<code>.assertIs(a, b)</code>	<code>a is b # a und b verweisen auf dasselbe Objekt</code>

.assertIsNone(x)	x is None
.assertIn(a, b)	a in b
.assertIsInstance(a, b)	isinstance(a, b) # a ist eine Instanz der Klasse b
.assertEqual(a, b)	a == b

Für die User-Klasse schreiben Sie eine Test-Klasse UserModelCase in einer neuen Datei tests.py im Projektverzeichnis. Dort sind zunächst einige Importe erforderlich:

```
# tests.py
from datetime import datetime, timedelta
import unittest
from app import app, db
from app.models import User, Post
```

Die Klasse UserModel Case wird von der Klasse unittest.TestCase abgeleitet. Sie bekommt eine setUp-Methode, die eine Testdatenbank erzeugt. Die Test-DB wird mit der Methode tearDown() wieder entfernt.

```
class UserModelCase(unittest.TestCase):
    def setUp(self):
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite://'
        db.create_all()

    def tearDown(self):
        db.session.remove()
        db.drop_all()
```

Der erste Testfall prüft, ob die Verschlüsselung und die Überprüfung des Passworts funktionieren:

```
def test_password_hashing(self):
    u = User(username='susan')
    u.set_password('cat')
    self.assertFalse(u.check_password('dog'))
    self.assertTrue(u.check_password('cat'))
```

Hier wird mit .assertFalse() überprüft, ob es nicht möglich ist, sich mit einem falschen Passwort anzumelden und mit .assertTrue(), ob es mit dem richtigen Passwort funktioniert.

Der nächste Fall ist die Überprüfung der Avatar-Methode:

```
def test_avatar(self):
    u = User(username='john', email='john@example.com')
    self.assertEqual(u.avatar(128), ('https://www.gravatar.com/avatar/'
                                    'd4c74594d841139328695756648b6bd6'
                                    '?d=identicon&s=128'))
```

Mit .assertEqual wird hier geprüft, ob mit der Methode avatar() der erwartete MD5-Hash für einen User erzeugt wird.

Um die Follower-Funktionalität zu testen, müssen zuerst zwei User-Accounts angelegt und in der Datenbank gespeichert werden.

```
def test_follow(self):
    u1 = User(username='john', email='john@example.com')
    u2 = User(username='susan', email='susan@example.com')
    db.session.add(u1)
    db.session.add(u2)
    db.session.commit()
    self.assertEqual(u1.followed.all(), [])
    self.assertEqual(u1.followers.all(), [])
```

Die Datenbank-Abfragen u1.followed.all() und u1.followers.all() sollten jeweils eine leer Liste als Ergebnis haben.

Danach wird die follow()-Methode geprüft

```
u1.follow(u2)
db.session.commit()
self.assertTrue(u1.is_following(u2))
self.assertEqual(u1.followed.count(), 1)
self.assertEqual(u1.followed.first().username, 'susan')
self.assertEqual(u2.followers.count(), 1)
self.assertEqual(u2.followers.first().username, 'john')
```

Hier werden 5 verschiedene erwartete Ergebnisse geprüft: Folgt u1 u2? Hat u2 genau einen Follower? Stimmt der Username für u2? Usw.

Anschliessend geht es an die unfollow()-Methode:

```
u1.unfollow(u2)
db.session.commit()
self.assertFalse(u1.is_following(u2))
self.assertEqual(u1.followed.count(), 0)
self.assertEqual(u2.followers.count(), 0)
```

Für die Tests der eigenen Posts und der Posts von Usern, denen gefolgt wird, werden Posts in der Datenbank und einige weitere User-Accounts gebraucht:

```
def test_follow_posts(self):
    # create four users
    u1 = User(username='john', email='john@example.com')
    u2 = User(username='susan', email='susan@example.com')
    u3 = User(username='mary', email='mary@example.com')
    u4 = User(username='david', email='david@example.com')
    db.session.add_all([u1, u2, u3, u4])

    # create four posts
    now = datetime.utcnow()
    p1 = Post(body="post from john", author=u1,
```

```

        timestamp=now + timedelta(seconds=1))
p2 = Post(body="post from susan", author=u2,
           timestamp=now + timedelta(seconds=4))
p3 = Post(body="post from mary", author=u3,
           timestamp=now + timedelta(seconds=3))
p4 = Post(body="post from david", author=u4,
           timestamp=now + timedelta(seconds=2))
db.session.add_all([p1, p2, p3, p4])
db.session.commit()

# setup the followers
u1.follow(u2) # john follows susan
u1.follow(u4) # john follows david
u2.follow(u3) # susan follows mary
u3.follow(u4) # mary follows david
db.session.commit()

# check the followed posts of each user
f1 = u1.followed_posts().all()
f2 = u2.followed_posts().all()
f3 = u3.followed_posts().all()
f4 = u4.followed_posts().all()
self.assertEqual(f1, [p2, p4, p1])
self.assertEqual(f2, [p2, p3])
self.assertEqual(f3, [p3, p4])
self.assertEqual(f4, [p4])

```

Am Ende des Moduls

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

Wenn tests.py mit python test.py aufgerufen wird, wird unittest.main() aufgerufen. Das führt den test runner von unittest aus, der alle Klassen in dieser Datei findet, die von unittest.TestCase abgeleitet wurden.

Die Ausgabe des Unit-Tests im Beispiel sieht so aus:

```
(tutorial) $ python3 tests.py
[2022-04-05 17:53:25,789] INFO in __init__: Microblog startup
test_avatar (__main__.UserModelCase) ... ok
test_follow (__main__.UserModelCase) ... ok
test_follow_posts (__main__.UserModelCase) ... ok
test_password_hashing (__main__.UserModelCase) ... ok
```

Ran 4 tests in 0.159s

OK

Beachten Sie, dass häufig nicht nur getestet werden muss, ob die Methode das erwartete Ergebnis hat sondern auch, ob sie nicht auch ein unerwartetes Ergebnis hat.

Die Vielzahl der getesteten Fälle hat Sie hoffentlich überzeugt, dass sich der Aufwand, diesen Unit-Test zu schreiben lohnt, weil er wesentlich geringer ist, als all diese Überprüfungen auch nur zehnmal manuell durchzuführen!

Von jetzt an können Sie bei jeder Änderung der Klasse User automatisch überprüfen, ob die bisherigen Testfälle noch erfolgreich sind. Für alle Erweiterungen und die anderen Klassen fügen Sie natürlich Testfälle und weitere Unit-Tests hinzu!

9.6 INTEGRATION IN DIE APP

Nachdem Follower nun in der Datenbank gespeichert sind und im User-Model die Methoden follow() und unfollow() getestet sind, kann das Ganze nun in die Oberfläche der App integriert werden.

Grundsätzlich könnten die Aktionen 'Follow' und 'Unfollow' als http GET- oder POST-Request an den Server geschickt werden, da keine Daten an die App übermittelt werden müssen.

GET wäre einfacher, aber für die Übermittlung aus einem Formular mit POST kennen Sie bereits einen Mechanismus, der die APP gegen eine CSRF-Attacke absichert (Siehe Kap. 4). Daher wird in app/forms.py ein Formular hinzugefügt, das nur ein einzelnes SubmitField enthält:

```
51     class EmptyForm(FlaskForm):  
52         submit = SubmitField('Submit')
```

In app/routes.py wird EmptyForm importiert und es werden zwei neue Funktionen für die Routen /follow/<username> und /unfollow/<username> implementiert. Zunächst ist ein Import von EmptyForm erforderlich:

```
6      from app.forms import LoginForm, RegistrationForm, EditProfileForm, EmptyForm  
  
102     @app.route('/follow/<username>', methods=['POST'])  
103     @login_required  
104     def follow(username):  
105         form = EmptyForm()  
106         if form.validate_on_submit():  
107             user = User.query.filter_by(username=username).first()  
108             if user is None:  
109                 flash('User {} not found.'.format(username))  
110                 return redirect(url_for('index'))  
111             if user == current_user:  
112                 flash('You cannot follow yourself!')  
113                 return redirect(url_for('user', username=username))  
114             current_user.follow(user)  
115             db.session.commit()  
116             flash('You are following {}!'.format(username))  
117             return redirect(url_for('user', username=username))  
118         else:  
119             return redirect(url_for('index'))
```

Abbildung 9-8 Funktion für 'Follow' in app/routes.py

```

122     @app.route('/unfollow/<username>', methods=['POST'])
123     @login_required
124     def unfollow(username):
125         form = EmptyForm()
126         if form.validate_on_submit():
127             user = User.query.filter_by(username=username).first()
128             if user is None:
129                 flash('User {} not found.'.format(username))
130                 return redirect(url_for('index'))
131             if user == current_user:
132                 flash('You cannot unfollow yourself!')
133                 return redirect(url_for('user', username=username))
134             current_user.unfollow(user)
135             db.session.commit()
136             flash('You are not following {}'.format(username))
137             return redirect(url_for('user', username=username))
138         else:
139             return redirect(url_for('index'))

```

Abbildung 9-9 Funktion für 'Unfollow' in app/routes.py

Beiden Funktionen gemeinsam ist:

- Es gibt keine eigenen Seiten für diese Funktionen. Sie sollen später über die user()-Route gerendert werden.
- Validate_on_submit() würde False zurückgeben, wenn der CSRF-Token aus form.hidden_tag() nicht korrekt wäre.
- Es gibt keine Eingaben, die validiert werden müssten, aber die Routen könnten im Browser manuell eingegeben werden. Die Validierung prüft daher vorsichtshalber mit User.query.filter_by(username=username).first(), ob
 - der beim Aufruf der Funktion angegebene User überhaupt existiert und
 - der User versucht, sich selbst zu folgen, bzw. nicht mehr zu folgen (user == current_user)
- Es gibt bei Erfolg eine Weiterleitung auf die User-Seite
- Es gibt bei Misserfolg eine Weiterleitung auf die Index-Seite

Die einzigen Unterschiede liegen in den Meldungen, die ausgegeben werden.

Die Buttons für 'Follow' und 'Unfollow' werden in der user()-Funktion gerendert und im Template für das User-Profil eingebaut (Zeile 82):

```

73     @app.route('/user/<username>')
74     @login_required
75     def user(username):
76         user = User.query.filter_by(username=username).first_or_404()
77         posts = [
78             {'author': user, 'body': 'Test post #1'},
79             {'author': user, 'body': 'Test post #2'}
80         ]
81         # Für 'Follow' / 'Unfollow'
82         form = EmptyForm()
83         return render_template('user.html', user=user, posts=posts, form=form)

```

Abbildung 9-11 Änderung von user() in app/routes.py

```

1  {% extends "base.html" %}
2
3  {% block content %}
4      <table>
5          <tr valign="top">
6              <td></td>
7              <td>
8                  <h1>User: {{ user.username }}</h1>
9                  {% if user.about_me %}<p>{{ user.about_me }}</p>{% endif %}
10                 {% if user.last_seen %}<p>Last seen on: {{ user.last_seen }}</p>{% endif %}
11                 <p>{{ user.followers.count() }} followers, {{ user.followed.count() }} following.</p>
12                 {% if user == current_user %}
13                     <p><a href="{{ url_for('edit_profile') }}>Edit your profile</a></p>
14                     {% elif not current_user.is_following(user) %}
15                         <p>
16                             <form action="{{ url_for('follow', username=user.username) }}" method="post">
17                                 {{ form.hidden_tag() }}
18                                 {{ form.submit(value='Follow') }}
19                             </form>
20                         </p>
21                     {% else %}
22                         <p>
23                             <form action="{{ url_for('unfollow', username=user.username) }}" method="post">
24                                 {{ form.hidden_tag() }}
25                                 {{ form.submit(value='Unfollow') }}
26                             </form>
27                         </p>
28                     {% endif %}
29                 </td>
30             </tr>
31         </table>
32         <hr>
33         {% for post in posts %}
34             {% include '_post.html' %}
35         {% endfor %}
36     {% endblock %}

```

Abbildung 9-10 Änderungen in app/templates/user.html

- Zeile 11: Es wird angezeigt, wie viele Follower es gibt und wie vielen anderen Usern dieser User folgt.

- Zeilen 12 - 13: Es wird geprüft, ob der User auf seiner eigenen Profilseite ist. Dann wird der 'Edit your Profile' Link angezeigt.
- Zeilen 14 - 20: Falls es nicht das eigene Profil ist und der aktuelle User diesem User noch nicht folgt, wird die EmptyForm mit der Button-Beschriftung 'Follow' angezeigt (Für einen Submit-Button kann mit dem Argument 'value=Text' die Beschriftung festgelegt werden).
- Zeilen 21 - 27: Falls es nicht das eigene Profil ist und der aktuelle User diesem User bereits folgt, wird die EmptyForm mit der Button-Beschriftung 'Unfollow' angezeigt.

Es gibt in der App noch keine Auswahlmöglichkeit für die User, aber Sie können bereits einen kleinen Test machen, indem Sie einige User registrieren und die Profilseite direkt über den URL aufrufen. Im folgenden Beispiel gibt es die User 'jochen' und 'paula'. Der User 'jochen' ist angemeldet:

Microblog: Home Profile Logout

User: jochen

Last seen on: 2022-04-06 08:25:47.791825

0 followers, 0 following.

[Edit your profile](#)

Jochen gibt den URL localhost:5000/user/paula ein. Er sieht in Ihrem Profil einen 'Follow'-Button:

Microblog: Home Profile Logout

User: paula

Last seen on: 2022-04-06 08:32:37.610339

0 followers, 0 following.

[Follow](#)

Nach dem Klick auf 'Follow', ändert sich der Button in 'Unfollow':

Microblog: Home Profile Logout

- You are following paula!

User: paula

Last seen on: 2022-04-06 08:32:37.610339

1 followers, 0 following.

[Unfollow](#)

9.7 ZUSAMMENFASSUNG

Zum Datenmodell:

- Ein User kann beliebig viele Posts haben
- Ein User kann beliebig vielen anderen Usern folgen und er kann selbst beliebig viele Follower haben.
- Eine Viele-zu-Viele Beziehung wird in einer relationalen Datenbank mithilfe einer zusätzlichen Beziehungstabelle (Hier: `followers`) umgesetzt. Sie enthält keine eigenen Daten, sondern nur die Fremdschlüssel aus zwei anderen Tabellen.
- Sie könnten theoretisch die Daten aus den Tabellen `user`, `followers` und `post` separat abfragen, in Python-Datenstrukturen einfügen und deren Elemente mit Python-Code miteinander verknüpfen und sortieren. Für solche Aufgaben sind aber die Operationen einer Datenbank (`JOIN`, `WHERE`, `UNION`, `ORDER BY`) viel besser geeignet.
- Mit dem ORM von SQLAlchemy können Sie Fremdschlüssel-Beziehungen deklarieren und mit Flask-Migrate in der Datenbank bekannt gemacht werden. Dabei ist es möglich, die Beziehungen in beiden Richtungen (`follows` / `followed by`) anzufragen.
- SQLAlchemy ermöglicht es Ihnen, Objekte zu denen eine Beziehung besteht als einfache Liste zu bearbeiten.
- Die Query- und Filter-Methoden von SQLAlchemy erlauben Ihnen die Formulierung von `JOINS`, `WHERE`-Bedingungen, `UNION` und `ORDER BY`, ohne dass direkt SQL-Anweisungen in den Code eingebettet werden müssen.

Zum Thema Unit-Test:

- Auch wenn die App noch relativ einfach ist, gibt es eine stetig wachsende Anzahl von Funktionalitäten und damit auch stetig wachsende Möglichkeiten für Fehler im Code. Diese manuell zu testen ist viel zu zeitaufwändig und Tests sollten daher automatisiert werden.
- Das Paket `unittest` aus der Standardbibliothek vereinfacht die Programmierung von automatischen Tests. Es bietet die Standard-Operationen `setUp()` zum Vorbereiten der Testumgebung, `tearDown()` zum Zurücksetzen der Test-Umgebung und viele `assert()`-Funktionen zum überprüfen und protokollieren der detaillierten Testfälle.

Zum Thema Sicherheit:

- Auch für Seitenelemente, bei denen nur ein Button benötigt wird ist es sinnvoll, eine Form und die Methode `POST` zu verwenden, da diese mit einem CSRF-Token abgesichert werden kann.

10 BLOG POSTS

Bisher haben Sie mit Fake-Posts gearbeitet, damit Sie die Anzeige von Posts testen konnten. In diesem Kapitel werden nun richtige Posts in die App eingebaut.

10.1 LERNZIELE

Nach der Bearbeitung dieses Kapitels können Sie:

- Die Erfassung von Blog-Posts implementieren
- Längere Listen seitenweise anzeigen

10.2 FORM, VIEW-FUNKTION UND TEMPLATE

Die Benutzer sollten nach dem Login in der Lage sein, einen kurzen Text als Blog-Post zu erfassen.

Erzeugen Sie dafür ein kleines Eingabeformular in app/forms.py:

```
55 class PostForm(FlaskForm):  
56     post = TextAreaField('Say something', validators=[DataRequired()])  
57     submit = SubmitField('Submit')
```

Abbildung 10-1 Das Formular für einen Post

Die Erfassung soll auf der Seite /index eingebaut werden. Ändern Sie app/templates/index.html wie folgt:

```
1  {% extends "base.html" %}  
2  
3  {% block content %}  
4      <h1>Hi, {{ current_user.username }}!</h1>  
5      <form action="" method="post">  
6          {{ form.hidden_tag() }}  
7          <p>  
8              {{ form.post.label }}<br>  
9              {{ form.post(cols=32, rows=4) }}<br>  
10             {% for error in form.post.errors %}  
11                 <span style="color: red;">{{ error }}</span>  
12             {% endfor %}  
13             </p>  
14             <p>{{ form.submit() }}</p>  
15         </form>  
16         {% for post in posts %}  
17             <p>  
18                 {{ post.author.username }} says: <b>{{ post.body }}</b>  
19             </p>  
20         {% endfor %}  
21     {% endblock %}
```

Zeile 5 - 15: Hier wird die Form angezeigt (Mitsamt hidden_tag Feld). Das TextArea-Feld wird im Format 4 Zeilen zu je 32 Zeichen formatiert. Allfällige Validierungsfehler werden wie bei den anderen Formularen mittels einer for-Schleife darunter ausgegeben. Darunter wird der Submit-Button platziert.

Auch die View-Funktion index() in app/routes.py muss nun angepasst werden. Zunächst müssen zwei Importe gemacht werden:

```
8  from app.forms import PostForm
9  from app.models import Post

18 @app.route('/', methods=['GET', 'POST'])
19 @app.route('/index', methods=['GET', 'POST'])
20 @login_required
21 def index():
22     form = PostForm()
23     if form.validate_on_submit():
24         post = Post(body=form.post.data, author=current_user)
25         db.session.add(post)
26         db.session.commit()
27         flash('Your post is now live!')
28         return redirect(url_for('index'))
29     posts = current_user.followed_posts().all()
30     return render_template("index.html", title='Home Page', form=form,
31                           posts=posts)
```

- Zeilen 18-19: Da in der Seite eine Form eingebaut werden soll, muss die Methode POST beim Decorator @app.route angegeben werden.
- Zeile 22: ein neues Objekt der Klasse PostForm wird erzeugt
- Zeilen 23 - 28: Wenn Die Validierung der Eingabe erfolgreich war, wird ein neues Post-Objekt erzeugt, zur db.session hinzugefügt und mit session.commit() gespeichert. Der Benutzer erhält ein Feedback über eine flash-Meldung.

Abbildung 10-2 Änderungen an der View-Funktion index() in routes.py

- Zeile 29 Die Fake-Posts werden durch echte Posts ersetzt. Dafür wird die Methode followed_posts() verwendet, die sie im vorhergehenden Kapitel der Klasse User hinzugefügt haben. Sie gibt ein SQLAlchemy Query-Objekt zurück und mit .all() wird die Query ausgeführt. Das Ergebnis ist eine Liste mit Ergebnissen, die die gleiche Struktur hat wie die Fake-Liste zuvor. Das Template muss daher nicht geändert werden.
- Zeilen 30 - 31: Bei render_template() wird zusätzlich das Objekt form übergeben.

Warum wird der Redirect in Zeile 28 gemacht?

Es ist gute Praxis, auf einen POST mit einem erneuten Laden der Seite zu reagieren. Wenn der Benutzer nämlich im Browser die Seite neu laden würde, würde der letzte Request erneut gesendet werden. Das wäre aber ein erneuter POST des Formularinhalts. Der Browser würde dann den Benutzer fragen, ob das Formular erneut senden soll, was verwirrend wäre. Wenn er bestätigt, würde der gleiche Blog-Post erneut an den Server geschickt.

Mit dem Redirect wird stattdessen ein GET-Request geschickt und bei einem erneuten Laden der Seite gibt es keine Verwirrung und keine unabsichtlichen Duplikate von Blog-Posts.

Testen Sie das auf der Index-Seite, erfassen Sie einige Beiträge



10.3 ANDERE USER FINDEN

Bisher sehen Sie in der App noch keine anderen Benutzer, sie werden nirgends angezeigt. Dafür soll nun eine zusätzliche Seite mit dem Namen 'Explore' erzeugt werden, die alle Posts von allen Usern anzeigt und dabei jeweils einen Link zum Profil des Autors anbietet.

Die View-Funktion `explore()` erzeugen Sie in `app/routes.py`:

```
141 |     @app.route('/explore')
142 |     @login_required
143 |     def explore():
144 |         posts = Post.query.order_by(Post.timestamp.desc()).all()
145 |         return render_template('index.html', title='Explore', posts=posts)
```

Abbildung 10-3 Die neue View-Funktion in `app/routes.py`

- Zeile 144: Alle Blog-Posts werden mit SQLAlchemy aus der Datenbank gelesen und nach Zeitstempel sortiert.
- Zeile 145: Es wird hier kein neues Template verwendet, sondern `index.html`!

Bevor Sie das testen, sollten Sie noch Änderungen an `app/templates/index.html` und `app/templates/base.html` machen. Fügen Sie in `base.html` einen Link 'Explore' hinzu:

```
<a href="{{ url_for('explore') }}>Explore</a>
```

In `index.html` muss nun noch berücksichtigt werden, dass Sie das Template einmal mit der Form für einen Blog-Post (über die Funktion `index()`) und einmal ohne (über die Funktion `explore()`) verwenden. In `explore()` gibt es keine Form. Deshalb muss in `index.html` eine Bedingung eingebaut werden, da `render_template()` sonst einen Fehler verursachen würde:

```

5      {% if form %}
6      <form action="" method="post">
7          {{ form.hidden_tag() }}
8          <p>
9              {{ form.post.label }}<br>
10             {{ form.post(cols=32, rows=4) }}<br>
11             {% for error in form.post.errors %}
12                 <span style="color: red;">[{{ error }}]</span>
13             {% endfor %}
14         </p>
15         <p>{{ form.submit() }}</p>
16     </form>
17     {% endif %}

```

- Zeile 5: Nur wenn beim Aufruf von render_template() ein Argument form angegeben wurde, wird die Form angezeigt.
- Zeile 17: Vergessen Sie nicht das {% endif }!

Die Links zu den Autoren werden nun in app/templates/_posts.html eingebaut:

```

1  <table>
2      <tr valign="top">
3          <td></td>
4          <td>
5              <a href="{{ url_for('user', username=post.author.username) }}">
6                  {{ post.author.username }}
7              </a>
8              says:<br>{{ post.body }}
9          </td>
10     </tr>
11 </table>

```

Abbildung 10-4 Erweiterung von app/templates/_posts.html um Autoren-Links

Dieses Subtemplate setzen Sie nun in anstelle von

`{{ post.author.username }} says: {{ post.body }}`
in der for-Schleife ein:

```

18      {% for post in posts %}
19          {% include '_post.html' %}
20      {% endfor %}

```

Abbildung 10-5 Subtemplate für Posts in app/templates/index.html

Testen Sie nun, ob die Anzeige auf der Index-Seite Ihren Erwartungen entspricht:

Wird das Subtemplate _posts.html verwendet?

Werden Avatare angezeigt?

Werden Links zu den Benutzerprofilen angezeigt und führen diese auf das richtige Profil?

Ist der Explore-Link auf allen Seiten zu sehen?

Werden auf der Explore-Seite alle Posts mit Avatar und Link zum User angezeigt?

The screenshot shows a web browser window with the URL 127.0.0.1. The title bar says "Microblog: Home Explore Profile Logout". The main content area displays a message "Hi, kurt!" in large bold letters. Below it is a text input field with placeholder "Say something" and a "Submit" button. Two comments are visible:

- A comment from "kurt" saying: "Ich habe festgestellt, dass die Anzeige der auf der Index-Seite funktioniert, aber auf meiner Profil-Seite immer noch Fake-Posts stehen!"
- A comment from "kurt" saying: "Das ist mein erster ECHTER Blog-Post. Er wird in der Datenbank gesichert und auf der Index-Seite angezeigt!"

The screenshot shows a web browser window with the URL 127.0.0.1:5000/explore. The title bar says "Microblog: Home Explore Profile Logout". The main content area displays a message "Hi, kurt!" in large bold letters. Below it is a list of posts:

- A post from "kurt" saying: "Ich habe festgestellt, dass die Anzeige der auf der Index-Seite funktioniert, aber auf meiner Profil-Seite immer noch Fake-Posts stehen!"
- A post from "kurt" saying: "Das ist mein erster ECHTER Blog-Post. Er wird in der Datenbank gesichert und auf der Index-Seite angezeigt!"
- A post from "paula" saying: "Mein zweiter echter Beitrag. Er wird jetzt in der Datenbank gespeichert und auf der Index- und auf der Profil-Seite angezeigt!"
- A post from "paula" saying: "Mein erster ECHTER Blog Post"
- A post from "jochen" saying: "Mein zweiter erster Blog-Post"
- A post from "jochen" saying: "Mein erster echter Blog Post"

10.4 PAGINIERUNG

Würden im Laufe der Zeit sehr viele, vielleicht tausende Blog-Beiträge veröffentlicht, könnte die in diesem Kapitel bisher verwendete Art der Anzeige problematisch werden. Neben einer schlechten Benutzbarkeit sehr langer Listen könnten die Datenbankzugriffe langsam werden, und die Listen der Suchergebnisse könnten den Hauptspeicher des Servers schnell erschöpfen.

Aus diesem Grund zeigt praktisch jede Webanwendung lange Listen nur seitenweise an. Von einer Liste werden nur die ersten 10 oder 25 Einträge dargestellt und der Benutzer kann dann über einen 'Vorwärts' und einen 'Rückwärts' Link weitere Einträge anfordern und zwischen den Ergebnisseiten vor

und zurück navigieren. Häufig wird aber schon unter den ersten Ergebnissen fündig, insbesondere wenn Sie nach Aktualität sortiert sind.

Die Seitenweise Ausgabe wird Paginierung genannt und Flask-SQLAlchemy unterstützt das Lesen einer begrenzten Anzahl von Datensätzen mit der Methode paginate().

Sie können die Query

```
user.followed_posts().all()
```

ersetzen durch:

```
user.followed_posts().paginate(1, 20, False).items
```

Die drei Argumente von .paginate() sind: Die Startseite (1), die Anzahl von angezeigten Elementen pro Seite (20) und ein Flag, das bestimmt, was passiert, wenn eine Seite angefragt wird, die über das Ende der tatsächlich gefundenen Seiten hinausgeht. Wenn es den Wert False hat, wird in diesem Fall eine leere Liste zurückgegeben.

Die Anzahl der Elemente in den verschiedenen Seiten der App sollte zentral in config.py gesetzt werden. Fügen Sie dort der Config-Klasse eine weitere Variable hinzu:

```
class Config(object): # ...
    POSTS_PER_PAGE = 3
```

Wie kann die Seitennummer in URLs dargestellt werden? Eine Möglichkeit ist ein Query-String im URL:

```
http://localhost:5000/index?page=1 oder
http://localhost:5000/index?page=5
```

Den Inhalt des Arguments page bekommen Sie mit dem request.args Objekt (siehe Kapitel 2.5.1 und 6.6). Editieren Sie in app/routes.py die Funktionen index.py und explore.py

- Fragen sie die page mit request.args.get() ab
- Ändern Sie nun die Queries der Posts und
- Benutzen Sie in render_template posts.items als Wert für das Argument posts

```
29     page = request.args.get('page', 1, type=int)
30     # posts = current_user.followed_posts().all()
31     posts = current_user.followed_posts().paginate(
32         page, app.config['POSTS_PER_PAGE'], False)
33     return render_template("index.html", title='Home Page', form=form,
34                           posts=posts.items)
```

Abbildung 10-6 Paginierte Query in index()

```
147     page = request.args.get('page', 1, type=int)
148     # posts = Post.query.order_by(Post.timestamp.desc()).all()
149     posts = Post.query.order_by(Post.timestamp.desc()).paginate(
150         page, app.config['POSTS_PER_PAGE'], False)
151     return render_template('index.html', title='Explore', posts=posts.items)
```

Abbildung 10-7 Paginiert Query in explore()

Jetzt ermitteln die beiden Funktionen die anzugezeigende Seite entweder aus dem page-Argument des Query-Strings in der URL oder setzen sie = 1, wenn page= nicht angegeben wurde. Die Anzahl der Seiten wird aus der Anzahl der Ergebnisse und dem Config-Eintrag POSTS_PER_PAGE ermittelt.

Probieren Sie diese Änderung aus (Stellen Sie sicher, dass Sie mehr als 3 Blog-Posts erfasst haben).

Für die 2. Seite geben Sie die folgende URL mit Query-String ein:

<http://localhost:5000/explore?page=2>

10.5 SEITEN-NAVIGATION

Die nächste Änderung fügt Links zum Vorwärts- und Rückwärtsblättern unterhalb der Liste der Posts ein. Im letzten Abschnitt wurde ein Pagination-Objekt posts verwendet, genauer gesagt dessen Attribut .items . Es enthält eine Liste der Ergebnisse, die auf eine Seite passt. Ein Pagination-Objekt hat aber noch weitere Attribute, die für eine Seitennavigation nützlich sind:

- **has_next** ist True, wenn es noch weitere Seiten gibt
- **has_prev** ist True, wenn es Vorgänger-Seiten gibt
- **next_num** enthält die Nummer der nächsten Seite
- **prev_num** enthält die Nummer der Vorgänger-Seite

Damit können die Vorwärts- und Rückwärts-Links erzeugt und an das Template übergeben werden. In routes.py müssen Sie dafür index() und explore() abermals anpassen:

```
29     page = request.args.get('page', 1, type=int)
30     posts = current_user.followed_posts().paginate(
31         page, app.config['POSTS_PER_PAGE'], False)
32     next_url = url_for('index', page=posts.next_num) \
33         if posts.has_next else None
34     prev_url = url_for('index', page=posts.prev_num) \
35         if posts.has_prev else None
36     return render_template('index.html', title='Home', form=form,
37                           posts=posts.items, next_url=next_url,
38                           prev_url=prev_url)
```

Abbildung 10-9 Anpassungen in index()

```
151     page = request.args.get('page', 1, type=int)
152     posts = Post.query.order_by(Post.timestamp.desc()).paginate(
153         page, app.config['POSTS_PER_PAGE'], False)
154     next_url = url_for('explore', page=posts.next_num) \
155         if posts.has_next else None
156     prev_url = url_for('explore', page=posts.prev_num) \
157         if posts.has_prev else None
158     return render_template("index.html", title='Explore', posts=posts.items,
159                           next_url=next_url, prev_url=prev_url)
```

Abbildung 10-8 Anpassungen in explore()

Der Funktion url_for() können Sie beliebige zusätzliche Schlüsselwort-Argumente übergeben. Hier wird das mit page=Wert gemacht. Wenn page=Wert noch nicht Teil des URL ist, fügt Flask das zum URL hinzu, der von url_for() gefunden wird.

- Zeile 154: Die next_url wird mithilfe von url_for(funktion, page=posts.next_num) ermittelt. Falls die aktuelle Seite bereits die letzte Ergebnis-Seite ist, ist das Attribut posts.has_next == False und next_url wird auf den leeren Wert None gesetzt.
- Zeile 156: Die prev_url wird mithilfe von url_for(funktion, page=posts.prev_num) ermittelt. Falls die aktuelle Seite die erste Ergebnis-Seite ist, ist das Attribut posts.has_prev == False und prev_url wird auf den leeren Wert None gesetzt.
- Zeile 159: Die Schlüsselwort-Argumente next_url und prev_url werden der Funktion render_template zusätzlich übergeben.

next_url und prev_url können nun im Template app/templates/index.html als Links eingesetzt werden:

```
18     {% for post in posts %} 
19       |   {% include '_post.html' %} 
20     |   {% endfor %} 
21     |   {% if prev_url %} 
22       <a href="{{ prev_url }}>Newer posts</a> 
23     |   {% endif %} 
24     |   {% if next_url %} 
25       <a href="{{ next_url }}>Older posts</a> 
26     |   {% endif %} 
27   |   {% endblock %} 
--
```

Falls next_url oder prev_url nicht None sind, erscheinen sie als Link unterhalb der Blog-Posts:

The screenshot shows a web browser window with the URL `127.0.0.1:5000/explore`. The page content includes a header "Hi, kurt!", followed by several blog posts from user "kurt". Each post has a timestamp and a message. Below the posts is a link labeled "Older posts".

Microblog: Home Explore Profile Logout

Hi, kurt!

[kurt](#) says:
Ich habe festgestellt, dass die Anzeige der auf der Index-Seite funktioniert, aber auf meiner Profil-Seite immer noch Fake-Posts stehen!

[kurt](#) says:
Das ist mein erster ECHTER Blog-Post. Er wird in der Datenbank gesichert und auf der Index-Seite angezeigt!

[paula](#) says:
Mein zweiter echter Beitrag. Er wird jetzt in der Datenbank gespeichert und auf der Index- und auf der Profil-Seite angezeigt!

[Older posts](#)

The screenshot shows a web browser window with the URL `127.0.0.1:5000/explore?page=2`. The page content includes a header "Hi, kurt!", followed by several blog posts from users "paula" and "jochen". Each post has a timestamp and a message. Below the posts is a link labeled "Newer posts".

Microblog: Home Explore Profile Logout

Hi, kurt!

[paula](#) says:
Mein erster ECHTER Blog Post

[jochen](#) says:
Mein zweiter erster Blog-Post

[jochen](#) says:
Mein erster echter Blog-Post

[Newer posts](#)

10.6 PAGINIERUNG IN DER PROFILSEITE

Ersetzen Sie jetzt die noch verbliebenen Fake-Posts auf der Profilseite durch die echten Posts des Users und die der User, denen er folgt und wenden Sie die gleichen Massnahmen zur Paginierung an, die Sie schon bei index() und explore() benutzt haben. Ändern Sie die Funktion user(username) in app/routes.py entsprechend der folgenden Abbildung:

```
80  def user(username):
81      user = User.query.filter_by(username=username).first_or_404()
82      page = request.args.get('page', 1, type=int)
83      posts = user.posts.order_by(Post.timestamp.desc()).paginate(
84          page, app.config['POSTS_PER_PAGE'], False)
85      next_url = url_for('user', username=user.username, page=posts.next_num) \
86          if posts.has_next else None
87      prev_url = url_for('user', username=user.username, page=posts.prev_num) \
88          if posts.has_prev else None
89      form = EmptyForm()
90      return render_template('user.html', user=user, posts=posts.items,
91          next_url=next_url, prev_url=prev_url, form=form)
```

Abbildung 10-10 Echte Posts und Paginierung für die Profilseite

In Zeile 83 wird der Umstand ausgenutzt, dass im User-Model bereits eine Beziehung 'posts' zu den Posts definiert wurde, die bei einer Abfrage der Datenbank mit einem JOIN benutzt wird. In user.posts liegt nach der Abfrage eine Liste der Posts vor, die dieser User geschrieben hat, inklusive Zeitstempel. Die Query wird ausgeführt, die Ergebnisse werden nach Zeitstempel sortiert und ein Pagination-Objekt posts wird erstellt.

Die Übrigen Zeilen unterscheiden sich nicht vom Vorgehen bei index() und explore()

Machen Sie in user.html die gleichen Anpassungen wie in index.html:

```
33      {% for post in posts %}
34          {% include '_post.html' %}
35      {% endfor %}
36      {% if prev_url %}
37          <a href="{{ prev_url }}>Newer posts</a>
38      {% endif %}
39      {% if next_url %}
40          <a href="{{ next_url }}>Older posts</a>
41      {% endif %}
42  {% endblock %}
```

Abbildung 10-11 Änderungen in app/templates/user.html

In den Zeilen 33 bis 35 wird das mit render_template() übergebene Objekt posts durchlaufen. Dabei wird automatisch angehalten, wenn die maximale Anzahl Posts pro Seite erreicht ist.

In den Zeilen 36 bis 41 befindet sich der Code für die Links der Seitennavigation, die Sie schon bei index.html in den vorhergehenden Abschnitten umgesetzt haben.

Setzen Sie den Wert von POSTS_PER_PAGE in config.py auf eine grössere Anzahl, z.B. 10 oder 25 und probieren Sie das aus!

10.7 ZUSAMMENFASSUNG

- Sie haben in diesem Kapitel erneut die Muster View-Funktion/Template und View-Funktion/Form/Template angewendet, das Ihnen inzwischen vertraut sein sollte.
- In Ihrer App können jetzt Blog Posts erfasst und gespeichert werden. Dafür haben Sie die in den vorhergehenden Kapiteln aufgebauten Datenstrukturen genutzt.
- Sie haben die Anzeige der provisorischen Fake-Blog-Posts durch echte Blog-Posts ersetzt
- Sie haben die Paginierungs-Möglichkeiten mit SQLAlchemy kennengelernt und verschiedene eingebaute Attribute des Pagination-Objekts wie .items, .has_next, has_prev, next_num und prev_num genutzt, um eine seitenweise Anzeige von Listen mit 'Vorwärts' und 'Rückwärts' Links zu implementieren.
- Sie haben das Untertemplate _posts.html in mehreren anderen Templates wieder verwendet.

11 E-MAIL

Es gibt verschiedene Gründe, warum man von einer Website aus E-Mails verschicken möchte. Dazu gehören Newsletter und Bestätigungs-E-mails für Anmeldungen und Käufe. Einer der häufigsten Gründe ist aber sicher, einem Benutzer zu helfen, sein Passwort zurückzusetzen. Diese Funktionalität soll in diesem Kapitel hinzugefügt werden.

11.1 LERNZIELE

- Nach der Bearbeitung dieses Kapitels können Sie
- Die Erweiterung Flask-Mail konfigurieren und benutzen
- Sichere Token für Einmal-Logins mit JSON Web Tokens und pyjwt einsetzen
- Die Vorteile einer asynchronen Ausführung von Funktionen benennen und beim Email-Versand Threads einsetzen

11.2 FLASK-MAIL UND PYJWT

Für die Aufgaben in diesem Kapitel brauchen Sie zwei neue Erweiterungen. Flask-Mail liefert Ihnen die Klassen Mail() und Message(), mit denen Sie in Ihrer App das SMTP (Simple Mail Transfer Protokoll) aufsetzen können und E-Mails versenden können. Installieren Sie es mit:

```
(tutorial) $ pip install flask-mail
```

Um Benutzern eine Möglichkeit zu geben, ihr verlorenes Passwort zurückzusetzen, ist ein Login über einen Link nötig, der nur einmal benutzt werden kann. Dieser Link muss ein sicheres Token enthalten, das mit JSON Web Tokens (JWT) erzeugt werden kann. JWT ist ein offener Standard ([RFC 7519](#)) für die sichere Übermittlung von Informationen im JSON-Format (JavaScript Object Notation). Diese Informationen können verifiziert werden und sind vertrauenswürdig, weil sie signiert sind. Das Paket pyjwt implementiert JWT für Python. Installieren Sie es mit:

```
(tutorial) $ pip install pyjwt
```

Flask-Mail wird in app/__init__.py importiert und initialisiert. Tragen Sie dort folgende Zeilen ein:

```
# ... Zeilen ausgelassen
from flask_mail import Mail #NEU

app = Flask(__name__)
app.config.from_object(Config)
db = SQLAlchemy(app)
migrate = Migrate(app, db)
login = LoginManager(app)
login.login_view = 'login'

mail = Mail(app)          #NEU
# ... weitere Zeilen ausgelassen
```

Damit Sie Mail an einen SMTP-Server versenden können, müssen folgende Variablen in der Klasse Config() in config.py gesetzt sein:

```
class Config(object):
    # Zeilen ausgelassen, neu hinzu kommt:
    MAIL_SERVER = os.environ.get('MAIL_SERVER')
    MAIL_PORT = int(os.environ.get('MAIL_PORT') or 587)
    MAIL_USE_TLS = os.environ.get('MAIL_USE_TLS') is not None
    MAIL_USERNAME = os.environ.get('MAIL_USERNAME')
    MAIL_PASSWORD = os.environ.get('MAIL_PASSWORD')
    ADMINS = ['your-email@example.com']
```

Auch hier wird wieder das Prinzip verfolgt, keine sensiblen Informationen direkt in config.py zu schreiben, sondern die Werte für diese Variablen aus Umgebungsvariablen des Betriebssystems zu laden.

Setzen Sie die entsprechenden Umgebungsvariablen in Ihrer Betriebssystem-Shell. Die Werte hängen von Ihrem Mailserver oder -Provider ab. Falls Sie keinen Zugriff auf einen eigenen Mail-Server haben, der Mails von Ihrer App akzeptiert, empfiehlt es sich, den Service SendGrid zu nutzen, wie es bereits in Kapitel 8.5.2 beschrieben wurde. Für SendGrid müssen Sie folgendes setzen (Unter Windows benutzen Sie *set* anstelle von *export*):

```
(tutorial) $ export MAIL_SERVER=smtp.sendgrid.net
(tutorial) $ export MAIL_PORT=587
(tutorial) $ export MAIL_USE_TLS=1
(tutorial) $ export MAIL_USERNAME=apikey
(tutorial) $ export SENDGRID_API_KEY=<Mein Sendgrid API-Key>
(tutorial) $ export MAIL_PASSWORD==<Mein Sendgrid API-Key>
```

11.3 TEST VON FLASK-MAIL

Nachdem Sie die nötigen Angaben als Umgebungsvariablen konfiguriert haben, testen Sie bitte das Versenden einer E-Mail in der flask shell:

```
(tutorial) $ flask shell
>>> from flask_mail import Message
>>> from app import mail
>>> msg = Message('Test Betreff', sender=app.config['ADMINS'][0],
...     recipients=['ihr-email-benutzer@ihre-email-domain'])
>>> msg.body = 'Test E-Mail aus der REPL'
>>> msg.html = '<h1>Test E-Mail aus der REPL</h1>'
>>> mail.send(msg)
```

Wenn das funktioniert hat, können Sie diese Schritte in einer Funktion in Ihrer App verwenden.

11.4 EINE MAIL-FUNKTION

Erzeugen Sie eine neue Datei app/email.py. Sie soll eine Funktion enthalten, die im Wesentlichen die Elemente aus dem vorhergehenden interaktiven Test in der App verfügbar macht:

```
from flask_mail import Message
from app import mail
def send_email(subject, sender, recipients, text_body, html_body):
    msg = Message(subject, sender=sender, recipients=recipients)
    msg.body = text_body
    msg.html = html_body
    mail.send(msg)
```

Hier werden nicht alle Möglichkeiten von Flask-Mail genutzt. Für eine Übersicht, was es sonst noch für Möglichkeiten gibt, studieren Sie bitte die Dokumentation¹³.

11.5 PASSWORT VERGESSEN?

Das Zurücksetzen des Passworts erfolgt in drei Schritten:

1. Der Benutzer fragt einen Passwort-Reset an
2. Die App verschickt einen Link, der das ermöglicht
3. Wenn der Benutzer den Link benutzt, wird Ihm ein Formular präsentiert, mit dem er sein neues Passwort eingeben kann

Im das Login-Formular app/templates/login.html wird unten auf der Seite ein neuer Link eingefügt:

```
23   </form>
24   <p>New User? <a href="{{ url_for('register') }}>Click to Register!</a></p>
25   <p>
26   |   Forgot Your Password?
27   |   <a href="{{ url_for('reset_password_request') }}>Click to Reset It</a>
28   </p>
29  {% endblock %}
```

Abbildung 11-1 Neuer Link in der Login-Seite (app/templates/login.html)

Wenn dieser Link angeklickt wird, soll ein Formular angezeigt werden. Fügen Sie folgende Klasse zu app/forms.py hinzu:

```
59  class ResetPasswordRequestForm(FlaskForm):
60      email = StringField('Email', validators=[DataRequired(), Email()])
61      submit = SubmitField('Request Password Reset')
```

Abbildung 11-2 Neues Formular in app/forms.py

¹³ <https://pythonhosted.org/Flask-Mail/>

Das HTML-Template für das Formular erstellen Sie als neue Datei
app/templates/reset_password_request.html:

```
1  {% extends "base.html" %}          1  {% extends "base.html" %}          1
2  {% block content %}              2  {% block content %}              2
3      <h1>Reset Password</h1>        3      <h1>Reset Password</h1>
4      <form action="" method="post">    4      <form action="" method="post">
5          {{ form.hidden_tag() }}       5          {{ form.hidden_tag() }}
6          <p>                      6          <p>
7              {{ form.email.label }}<br> 7              {{ form.email.label }}<br>
8              {{ form.email(size=64) }}<br> 8              {{ form.email(size=64) }}<br>
9              {% for error in form.email.errors %} 9              {% for error in form.email.errors %}
10                 <span style="color: red;">[{{ error }}]</span> 10                 <span style="color: red;">[{{ error }}]</span>
11                 {% endfor %}           11                 {% endfor %}
12             </p>                      12             </p>
13             <p>{{ form.submit() }}</p> 13             <p>{{ form.submit() }}</p>
14         </form>                    14         </form>
15     {% endblock %}                15     {% endblock %}
```

Abbildung 11-3 Neues Template app/templates/reset_password_request.html

Fügen Sie in app/routes.py zwei Importe und die Funktion reset_password_request() hinzu:

```
9   from app.forms import ResetPasswordRequestForm
10  from app.email import send_password_reset_email
169 @app.route('/reset_password_request', methods=['GET', 'POST'])
170 def reset_password_request():
171     if current_user.is_authenticated:
172         return redirect(url_for('index'))
173     form = ResetPasswordRequestForm()
174     if form.validate_on_submit():
175         user = User.query.filter_by(email=form.email.data).first()
176         if user:
177             send_password_reset_email(user)
178             flash('Check your email for the instructions to reset your password')
179             return redirect(url_for('login'))
180     return render_template('reset_password_request.html',
181                           title='Reset Password', form=form)
```

- Zeilen 9 - 10: Zunächst importieren Sie die Form und die neue email-Funktion aus app/email
- Zeile 169: Da es sich um eine Seite mit Formular handelt, müssen beim Decorator @app.route die Methoden GET und POST angegeben werden.
- Zeilen 171 - 172: Der Passwort-Reset soll nur nicht angemeldeten Benutzern angeboten werden. Das prüfen Sie mithilfe von current_user.is_authenticated. Wenn das True liefert, wird der Benutzer auf die Hauptseite umgeleitet.
- Zeile 173: Ansonsten wird ein Objekt form aus der Klasse ResetPasswordRequestForm() erzeugt.
- Der Rest entspricht den anderen View-Funktionen, die Sie bisher geschrieben haben: Wenn der Benutzer die Seite mit GET betritt, wird einfach das Formular gerendert (Zeile 180 am Ende der

Funktion). Falls er die Angaben korrekt ausgefüllt hat, wird in der Datenbank nach dem User gesucht (Zeile 175) und wenn der gefunden wurde, wird in Zeile 177 die Funktion send_password_reset_email(user) aufgerufen, die sie noch schreiben müssen. Anschliessend wird die Meldung 'Check your email ...' ausgegeben und der Benutzer wird auf die Login-Seite zurückgeschickt.

11.6 DER PASSWORT RESET TOKEN

Da E-Mail an den User soll einen Link enthalten, der einen sicheren Token für ein Einmal-Login enthält. Wenn der Benutzer diesen Link anklickt, soll er auf die soeben angelegte Seite kommen, auf der er ein neues Passwort eingeben kann.

Der Token im Link wird mit JWT (Jason Web Token) erzeugt. Diese Art Token braucht keine weiteren Elemente zur Verifizierung, der Link allein reicht. Testen Sie die Generierung und anschiesende Entschlüsselung eines Tokens in der REPL mithilfe des Package jwt:

```
>>> import jwt
>>> token = jwt.encode({'a': 'b'}, 'mein-geheimer-string', algorithm='HS256')
>>> token
'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhIjoiYiJ9.C0U8_N0r6Dy0_zX8-
8UdElAF3MV6svmGJJV8IQ6FoCU'
>>> jwt.decode(token, 'mein-geheimer-string', algorithms=['HS256'])
{'a': 'b'}
```

Beachten Sie, dass bei encode() ein String-Argument algorithm und bei decode() eine Liste algorithms (Mehrzahl) übergeben wird!

Das Dictionary {'a' : 'b'} ist im Beispiel der eigentliche Inhalt des Tokens. Um den Token sicher zu machen, wird ein Schlüssel gebraucht ('mein-geheimer-string'). Das Argument algorithm wird hier auf 'HS256' gesetzt, einem weit verbreiteten Algorithmus für eine symmetrische Kodierung ohne den vorherigen Austausch eines öffentlichen Schlüssels.

Der Token ist nicht verschlüsselt und kann relativ einfach von einer dritten Partei in Klartext umgewandelt werden. Was ihn aber trotzdem sicher macht, ist der Umstand, dass er signiert ist. Falls jemand das Email abfangen und den Inhalt des Dictionary verändern würde, wäre die Signatur ungültig.

Die Sicherheit des Verfahrens hängt vom Schlüssel ab und in einer öffentlichen Website muss er natürlich geheim und schwer zu erraten sein. Falls sie einen guten und lange Schlüssel für die Variable SECRET_KEY in config.py gesetzt haben (Sie haben den bereits für das Feld form.hidden_key in Templates verwendet, um CSRF-Attacken zu verhindern), ist das ein guter Kandidat!

In app/models.py importieren sie jwt, time und app.

```
6 | import jwt
7 | from time import time
8 | from app import app
```

Die Klasse User bekommt zwei neue Methoden:

```
18 class User(UserMixin, db.Model):
19     # ... Weitere Zeilen
20
21     def get_reset_password_token(self, expires_in=600):
22         return jwt.encode(
23             {'reset_password': self.id, 'exp': time() + expires_in},
24             app.config['SECRET_KEY'], algorithm='HS256')
25
26     @staticmethod
27     def verify_reset_password_token(token):
28         try:
29             id = jwt.decode(token, app.config['SECRET_KEY'],
30                             algorithms=['HS256'])['reset_password']
31         except:
32             return
33         return User.query.get(id)
```

Abbildung 11-4 Token-Handling in der Klasse User (app/models.py)

- Zeile 64: Die Methode hat ein Argument expires_in, mit dem festgelegt wird, wie lange der Token gültig sein soll. Wird nichts anderes angegeben, ist der Default 600 Sekunden.
- Zeilen 65 - 67: Die Funktion encode() aus dem Package jwt bekommt als Inhalt ein Dictionary mit den folgenden Schlüssel/Wert Paaren: 'reset_password' hat als Wert die user_id des User-Objekts. 'exp' enthält die Anzahl Sekunden, die der Token gültig sein soll. Das nächste Argument ist der Wert von SECRET_KEY aus config.py und der Algorithmus ist 'HS256'. Der Rückgabewert der Methode ist der aus diesen Angaben generierte Token.
- Zeile 69: @staticmethod ist ein neuer Decorator, der bisher noch nicht erklärt wurde. Er bewirkt, dass die Methode verify_reset_password_token() direkt über die Klasse User aufgerufen werden kann: User.verify_reset_password_token(). Das heisst, die Methode steht zur Verfügung ohne dass vorher ein User-Objekt erzeugt wurde.
- Zeilen 72 - 73: Die Funktion decode() ist das Gegenstück zu encode(). Sie liefert die den Inhalt des Tokens als Dictionary zurück. 'reset_password' ist der Schlüssel zum Eintrag, die user_id ist der Wert dazu. Er wird der Variablen id zugewiesen. Beachten Sie, dass bei encode() ein String-Argument 'algorithm' und bei decode() eine Liste 'algorithms' übergeben wird! Falls der Token nicht korrekt oder abgelaufen ist, würde eine Exception auftreten, die mit try und except abgefangen wird. Die Anweisung return ohne weitere Angabe gibt dann None zurück.

- Zeile 76 wird erreicht, wenn der Token erfolgreich dekodiert wurde und es keine Exception gegeben hat. Dann wird das User-Objekt mit der id aus dem Token aus der Datenbank geladen und von der Methode zurückgegeben.

11.7 DAS PASSWORT-RESET EMAIL

In app/email.py importieren Sie nun render_template und app und schreiben eine neue Funktion mit dem Namen send_password_reset_email(). Ihr Argument ist ein User-Objekt.

```
1  # app/email.py
2  from flask_mail import Message
3  from app import mail
4  from flask import render_template
5  from app import app
6
7  def send_email(subject, sender, recipients, text_body, html_body):
8      msg = Message(subject, sender=sender, recipients=recipients)
9      msg.body = text_body
10     msg.html = html_body
11     mail.send(msg)
12
13 def send_password_reset_email(user):
14     token = user.get_reset_password_token()
15     send_email('[Microblog] Reset Your Password',
16                 sender=app.config['ADMINS'][0],
17                 recipients=[user.email],
18                 text_body=render_template('email/reset_password.txt',
19                                           user=user, token=token),
20                 html_body=render_template('email/reset_password.html',
21                                           user=user, token=token))
```

Abbildung 11-5 Änderungen an app/email.py

Zeilen 4 - 5: Die Importe von render_template und app

Zeile 14: In send_password_reset_email() wird die Methode get_reset_password_token() des User-Objekts aufgerufen, die den Token zurückgibt.

Zeile 15: Die Funktion send_email() aus derselben Datei wird mit einem Betreff, einem Absender (Aus der Liste ADMINS) und zwei Dokumenten aufgerufen, die mit render_template() erzeugt werden. Das erste basiert auf einer Textdatei für einfache Text-E-mails, dem User-Objekt und dem Token. Das zweite basiert auf einer Datei für HTML-Mails., dem User-Objekt und dem Token.

Unter app/templates/email werden die Templates für diese Dokumente angelegt:

```
<!-- app/templates/email/reset_password.txt -->
Dear {{ user.username }},

To reset your password click on the following link:
{{ url_for('reset_password', token=token, _external=True) }}

If you have not requested a password reset simply ignore this message.

Sincerely,
The Microblog Team

<!-- app/templates/email/reset_password.html -->
<p>Dear {{ user.username }},</p>
```

```

<p>
    To reset your password
    <a href="{{ url_for('reset_password', token=token, _external=True) }}">
        click here
    </a>.
</p>
<p>Alternatively, you can paste the following link in your browser's address
bar:</p>
<p>{{ url_for('reset_password', token=token, _external=True) }}</p>
<p>If you have not requested a password reset simply ignore this message.</p>
<p>Sincerely,</p>
<p>The Microblog Team</p>

```

Die Route `reset_password`, die `url_for` in diesen Templates übergeben wird, existiert noch nicht, sie wird erst im nächsten Abschnitt implementiert.

Das Argument `_external=True` beim Aufruf von `url_for()` bewirkt, dass eine Komplette URL erzeugt wird. Ansonsten würde `url_for` nur eine interne Route innerhalb der App-Seitenstruktur liefern.

11.8 PASSWORT-RESET DURCHFÜHREN

In `app/routes.py` wird die neue View-Funktion für das Zurücksetzen des Passworts benötigt:

```

184     from app.forms import ResetPasswordForm
185
186     @app.route('/reset_password/<token>', methods=['GET', 'POST'])
187     def reset_password(token):
188         if current_user.is_authenticated:
189             return redirect(url_for('index'))
190         user = User.verify_reset_password_token(token)
191         if not user:
192             return redirect(url_for('index'))
193         form = ResetPasswordForm()
194         if form.validate_on_submit():
195             user.set_password(form.password.data)
196             db.session.commit()
197             flash('Your password has been reset.')
198             return redirect(url_for('login'))
199         return render_template('reset_password.html', form=form)

```

- Zeile 184: Import der neuen Form aus `app/forms.py` (Sie können den Import hier oder am Anfang der Datei machen)
- Zeile 186: Bei Seiten mit einem Formular ist immer die Angabe `methods=['GET', 'POST']` beim Decorator `@app.route` nötig
- Zeilen 188 - 189: Wenn der Benutzer schon angemeldet ist, ist ein Passwort-Reset nicht sinnvoll. Deshalb wird er in diesem Fall auf die Hauptseite weitergeleitet.
- Zeilen 190 - 192: Ansonsten wird der User auf Basis des Token aus der Datenbank geladen (Die in `app/models.py` definierte Methode macht die Datenbank-Query. Falls der User nicht gefunden wurde, liefert sie `None` zurück und in diesem Fall wird der User auch auf die Hauptseite weitergeleitet.

- Zeile 193: Wenn User.verify_reset_password_token(token) einen passenden User gefunden hat, wird das Formular erzeugt (muss noch im nächsten Schritt implementiert werden)
- Zeile 194 - 198: Falls validate_on_submit() True zurückgibt, wird die methode set_password() aus der Klasse User ausgeführt und die Änderung mit db.session.commit() in der Datenbank gespeichert. Anschliessend wird der User auf die Login-Seite weitergeleitet.
- Zeile 199: Falls der User die Seite mit GET betreten hat oder bei validate_on_submit() die Validierung fehlgeschlagen ist, wird die Seite geladen.

Das Formular dazu wird in app/forms.py definiert:

```

63  class ResetPasswordForm(FlaskForm):
64      password = PasswordField('Password', validators=[DataRequired()])
65      password2 = PasswordField(
66          'Repeat Password', validators=[DataRequired(), EqualTo('password')])
67      submit = SubmitField('Request Password Reset')

```

Abbildung 11-6 Formular zum Durchführen des Passwort-Reset

Es enthält die üblichen Felder für solche Fälle: Passwort, Wiederholung des Passworts und einen Submit-Button. Beide Felder müssen ausgefüllt sein und einen identischen Inhalt haben.

Als letzter Schritt wird das Template als neue Datei app/templates/reset_password.html erzeugt:

```

1  {% extends "base.html" %} 
2  {% block content %} 
3      <h1>Reset Your Password</h1>
4      <form action="" method="post">
5          {{ form.hidden_tag() }} 
6          <p>
7              {{ form.password.label }}<br>
8              {{ form.password(size=32) }}<br>
9              {% for error in form.password.errors %}
10                 <span style="color: red;">{{ error }}</span>
11             {% endfor %}
12             </p>
13             <p>
14                 {{ form.password2.label }}<br>
15                 {{ form.password2(size=32) }}<br>
16                 {% for error in form.password2.errors %}
17                     <span style="color: red;">{{ error }}</span>
18                 {% endfor %}
19                 </p>
20                 <p>{{ form.submit() }}</p>
21         </form>
22     {% endblock %}

```

Abbildung 11-7 Template für den Reset

In diesem Kapitel wurden sehr viele Änderungen gemacht und einige neue Dateien und Verzeichnisse erzeugt. Prüfen Sie das noch einmal anhand der folgenden Übersicht nach:

```

microblog
└── app
    ├── __init__.py          # Import Mail, mail=Mail(app)
    ├── email.py              # Importe Message aus flask-mail, mail aus app,
                                # render_template aus flask, app aus app.
                                # Funktionen send_email() und
                                # send_password_reset_email()
    ├── forms.py              # Klassen ResetPasswordRequestForm und
                                # ResetPasswordForm
    ├── models.py              # Importe time aus time, jwt, app aus app .
                                # Neue Methoden get_reset_password_token(),
                                # verify_reset_password_token() in class User
    ├── routes.py              # Importe von ResetPasswordForm und
                                # ResetPasswordRequestForm aus app.forms,
                                # send_password_reset_email aus app.email.
                                # Funktionen reset_password_request() und
                                # reset_password()
    └── templates
        └── email
            ├── reset_password.html      # Vorlage HTML-Mail
            └── reset_password.txt       # Vorlage Text-Mail
        ├── login.html                # Neuer Link
        ├── reset_password.html       # Neu
        └── reset_password_request.html # Neu

```

Einstellungen für E-Mail

Bitte testen Sie nun beide Teile des kompletten Ablaufs:

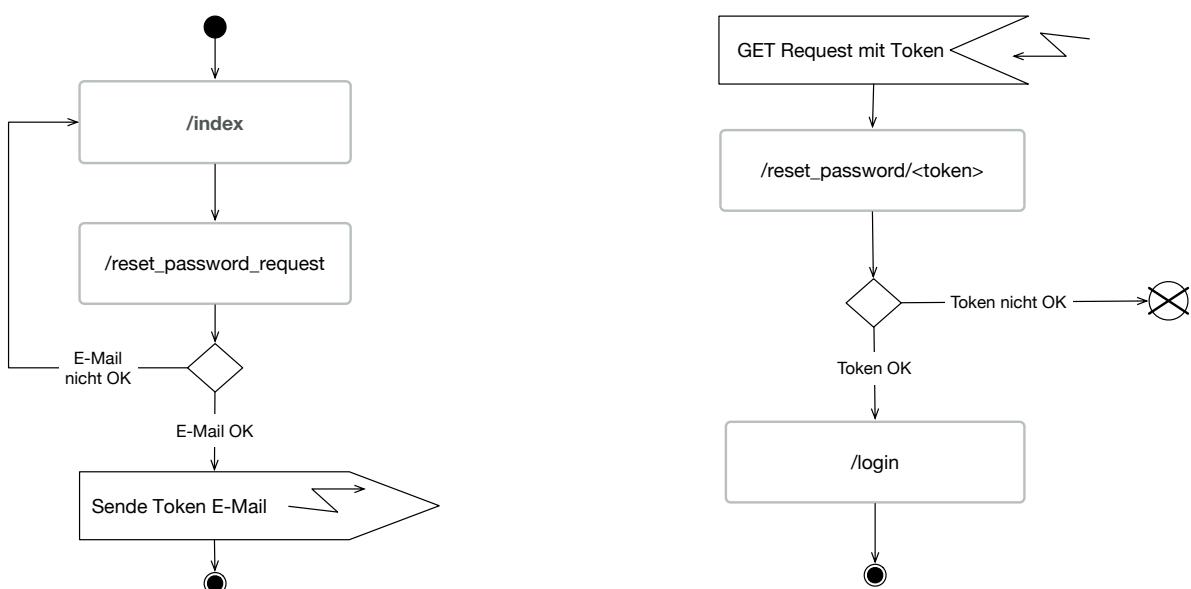


Abbildung 11-8 Ablauf Passwort Reset

11.9 ASYNCHRONE EMAILS

Die Art und Weise, wie das Versenden des Tokens im vorhergehenden Kapitel implementiert wurde, enthält einen Zugriff auf ein anderes Subsystem, (In unserem Fall greift Flask-Mail auf die smtplib¹⁴ aus der Standardbibliothek zurück, welche eine Client-Session mit dem SMTP-Server herstellt). Es vergeht dabei Zeit, bis die Nachricht erfolgreich verschickt wurde.

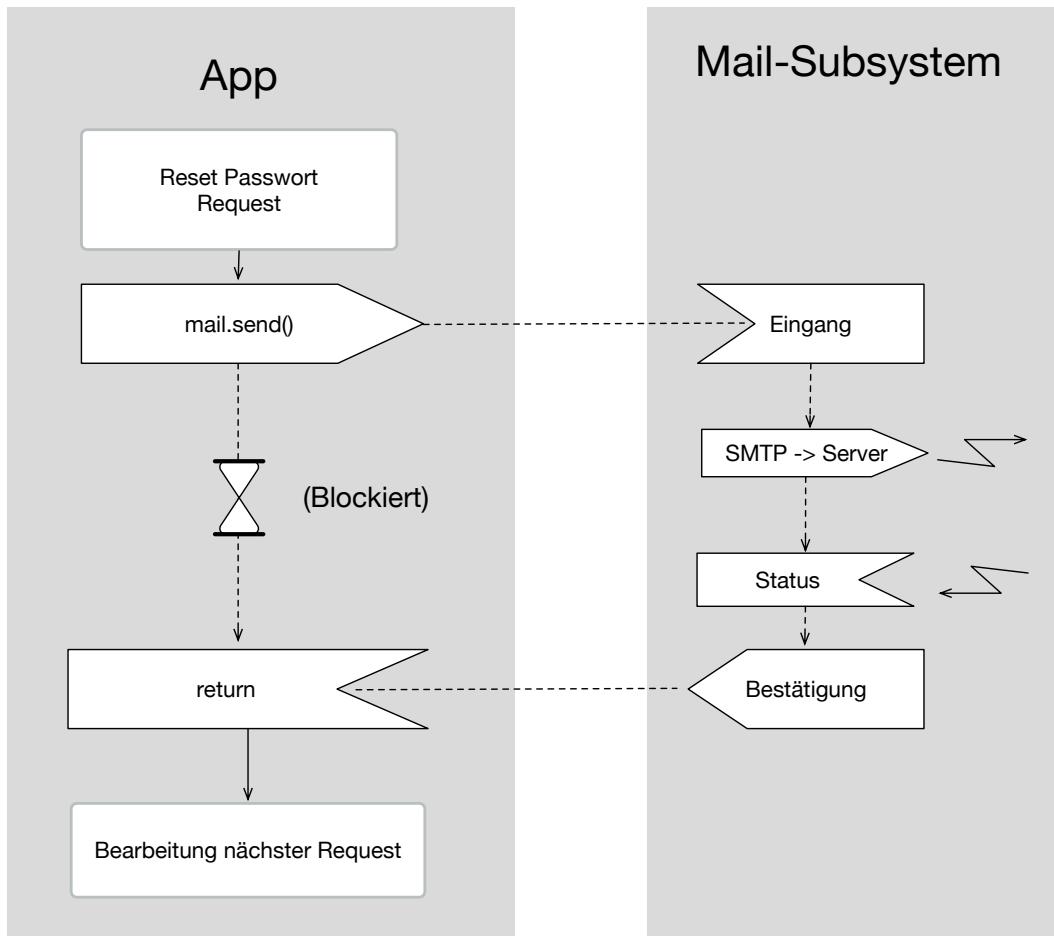


Abbildung 11-9 Ablauf beim Senden der E-Mail

Während dieser Zeit ist die App blockiert. Flask-Mail wartet auf die Bestätigung, dass die Mail verschickt wurde und die App wartet auf die Bestätigung von Flask-Mail. Weitere Requests des Clients müssen warten.

Ist eine Komponente in einer Client/Server Architektur blockiert, spricht man von einer synchronen Kommunikation. Dabei können im produktiven Betrieb verschiedene Probleme auftreten:

- Der Email-Server antwortet erst nach einer längeren Zeit
- Der Email-Server antwortet gar nicht
- Mehrere User machen zur gleichen Zeit eine Anfrage für einen Passwort-Reset und blockieren sich gegenseitig

¹⁴ <https://docs.python.org/3/library/smtplib.html>

Bei einer synchronen Kommunikation werden diese Probleme meist mittels eines Timeout behandelt. Ist nach einer festgelegten Zeit keine Antwort da, wird der Versuch abgebrochen. Eine feste Zeit für den Timeout kann aber problematisch sein:

- Die Zeit ist zu lang und die App ist für den Benutzer spürbar blockiert
- Die Zeit ist zu kurz und es gibt bei schlechten Verbindungen unnötige Abbrüche

Eine bessere Lösung ist, den Sendevorgang parallel durchzuführen. Dafür wird ein neuer Thread gestartet und die App kann sofort den nächsten Request des Clients behandeln.

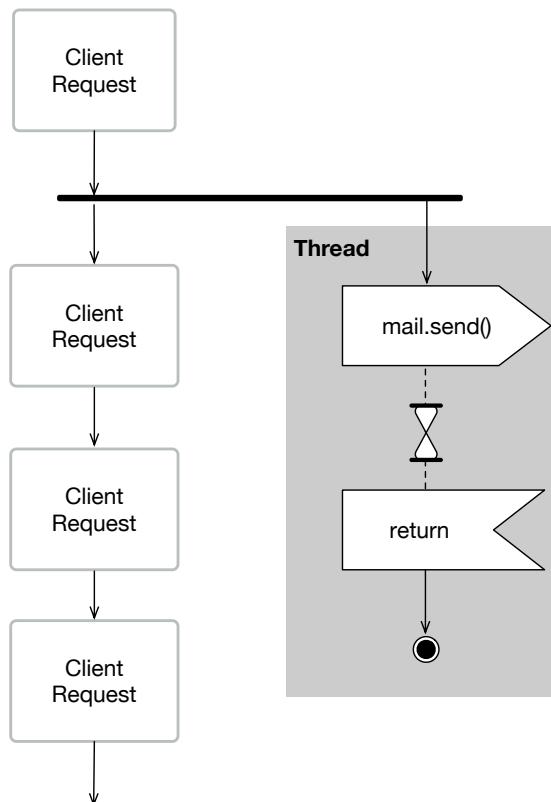


Abbildung 11-10 Ein Thread für E-Mail

Versenden Sie E-Mails nun *asynchron*. Dazu wird das Package `threading` aus der Standardbibliothek importiert. Implementieren Sie dort neue Funktion:

```

from threading import Thread
# ...

def send_async_email(app, msg):
    with app.app_context():
        mail.send(msg)

```

Die Funktion `send_email()` verändern Sie wie folgt:

```

def send_email(subject, sender, recipients, text_body, html_body):
    msg = Message(subject, sender=sender, recipients=recipients)
    msg.body = text_body
    msg.html = html_body
    Thread(target=send_async_email, args=(app, msg)).start()

```

Immer wenn Sie mit einer externen Ressource (hier: einem Mail-System) arbeiten, ist es empfehlenswert mit einem Kontext in einer with-Anweisung zu arbeiten. Der Application Context stellt Objekte von Flask zur Verfügung, auf die ähnlich wie auf globale Variablen zugegriffen werden kann (Siehe request- und application-Context in Kap. 2.5). Es ist dann nicht mehr nötig, diese Objekte an Methoden und Funktionen zu übergeben.

Das mail-Objekt wird in `__init__.py` auf Basis der Klasse Mail erzeugt und die Einstellungen für das Versenden sind im Config-Objekt (siehe config.py) gespeichert. Das ist hier der Grund, weshalb das app-Objekt der Funktion übergeben wird.

```
send_async_email(app, msg):
```

In vielen Fällen ist ein Context ohne explizite Übergabe an eine Funktion in dieser Funktion verfügbar, aber falls ein separater Thread gestartet wird, muss für diesen Thread eine neue Kopie erzeugt werden. In der Zeile

```
with app.app_context():
```

wird ein eigener app_context für den neuen Thread erzeugt. Der Thread selbst wird in `send_email()` mit der Methode `.start()` erzeugt. Dabei werden die Funktion, die im Thread ausgeführt werden soll, das app-Objekt und die Nachricht übergeben:

```
Thread(target=send_async_email, args=(app, msg)).start()
```

11.10 ZUSAMMENFASSUNG

- Die Python-Standardbibliothek enthält das Modul smtplib, mit deren Hilfe ein Python-Programm E-Mails zu einem Server schicken kann, der die Protokolle SMTP (Simple Mail Transfer Protocol) oder ESMTP (SMTP Service Extensions) beherrscht.
- Durch die Erweiterung Flask-Mail kann die Funktionalität von smtplib in eine Flask-App eingebunden werden. Sie stellt die Klassen Mail() und Message() zur Verfügung.
- Initialisiert wird die Funktionalität von Flask-Mail durch den Aufruf `mail = Mail(app)` in `app/__init__.py`, mit dem das Objekt mail erzeugt wird.
- Die Konfiguration für den Mail-Server und den Port, sowie für die Authentifizierung erfolgt durch Einträge in config.py.
- Eine E-Mail Nachricht wird durch die Erzeugung und Konfiguration eines Message() Objekts erstellt und der Versand der Nachricht kann anschliessend durch die `.send()` Methode von Mail() angestoßen werden.
- In der Client/Server Kommunikation wird zwischen *synchroner* und *asynchroner* Kommunikation unterschieden. Bei der synchronen Kommunikation wartet der Client auf die Antwort des Servers und ist blockiert, bis diese eintrifft oder bis eine feste Zeit (Timeout) abgelaufen ist. Beim Versenden von E-Mail an einen entfernten Mail-Server ist die synchrone Kommunikation nicht vorteilhaft, da die Zeitspanne, innerhalb derer der Server mit einer Bestätigung antwortet, nicht absehbar ist.

- Um die App nicht zu blockieren, kann der Vorgang des Versands asynchron gemacht werden. Die App startet dazu einen zusätzlichen Thread, der diese Aufgabe separat vom Haupt-Thread der App erledigt.
- Prozesse, die auf einem Computer ausgeführt werden, können andere Prozesse starten und jeder Prozess kann aus mehreren Threads bestehen. Beide können unabhängig voneinander ausgeführt werden. Neue Prozesse sind aufwändiger zu erzeugen als Threads, die zum selben Prozess gehören. Python unterstützt Threads und mehrere Prozesse durch die Module `threading`¹⁵ und `multiprocessing`¹⁶ aus der Standardbibliothek.
- Eine Funktion (Im Beispiel ist das `send_async_email()`) kann einem neuen Thread mithilfe der Klasse `Thread()` aus `threading` zugeordnet werden und mit der Methode `.start()` wird die Funktion als separater Thread ausgeführt.
- Das Zurücksetzen eines verlorenen Passworts durch den User erfordert ein einmaliges Login. Dieses Login wird über einen Link zur Verfügung gestellt. Der Link enthält einen von der App signierten Token, der für eine gewisse Zeit gültig ist. Das im Beispiel verwendete Format ist JSON Web Token (JWT), dass mittels des Moduls PyJWT verfügbar gemacht wird.
- Der Token-Inhalt nicht verschlüsselt, aber mit dem Schlüssel `SECRET_KEY` aus `config.py` siegiert. Es wird ausserdem festgelegt, dass der Token nur für eine kurze Zeitspanne. Solange der Schlüssel nur der App bekannt ist, ist diese Methode ausreichend gegen Manipulationen gesichert.

¹⁵ <https://docs.python.org/3/library/threading.html>

¹⁶ <https://docs.python.org/3/library/multiprocessing.html>

12 BOOTSTRAP

Die Gestaltung der App ist bisher nicht sehr ansehnlich. Das soll sich mit diesem Kapitel ändern. Aus Kapitel 1.7 wissen Sie, dass die Gestaltung von Webseiten mit Cascading Style Sheets (CSS) erfolgt. Webdesigner benutzen CSS intensiv, doch als Entwickler einer Server-App mussten Sie sich schon mit sehr vielen Details der Programmierung mit Flask beschäftigen. Die Erstellung von professionellem CSS zu erlernen wäre ein enormer zusätzlicher Aufwand. Diesen Aufwand können Sie weitgehend vermeiden, indem Sie ein CSS-Framework einsetzen. In diesem Kapitel wird das Framework Bootstrap verwendet.

12.1 LERNZIELE

Nach der Bearbeitung dieses Kapitels können Sie

- Das CSS-Framework Bootstrap mit der Erweiterung Flask-Bootstrap in Ihrer App verfügbar machen
- Das Konzept der Böcke im Basis-Template von Flask-Bootstrap erklären
- Ihr Basis-Template von einem Basis-Template für Bootstrap ableiten und
- Die Komponenten von Bootstrap in Templates benutzen
- Formulare sehr viel einfacher darstellen
- Die Paginierung-Links in Listen mit Bootstrap benutzerfreundlicher gestalten

12.2 CSS-FRAMEWORKS

Eine Klasse ist in CSS - eine Gruppierung von Eigenschaften, die einmal in einem Stylesheet (Datei mit der Endung .css) definiert wird und die anschliessend über ihren Namen in mehreren HTML-Dokumenten benutzt werden können, um die Darstellung eines HTML-Elements anzupassen. Im folgenden Beispiel wird eine Klasse in der Datei beispiel.css definiert

```
/*! Stylesheet beispiel.css */
.hinweis{
  font-style: italic;
}
```

In der Datei demo.html wird sie über einen Link referenziert und auf ein <p>-Element angewendet:

```
<head>
[...]
<link rel="STYLESHEET" type="text/CSS" href="styles.CSS">
[...]
</head>

<p class="hinweis"> Absatz, der mit formatiert ist.</p>
```

Ein CSS-Framework umfasst vorgefertigte Stylesheets, die Sie als Entwickler benutzen können, um Farben, Layouts, Schriften und Standard-Elemente wie Kopf- und Fusszeilen, Buttons, Slider, Menus und Navigation Bars auf den Webseiten Ihrer App darzustellen. Sie müssen die Klassen des Frameworks nur noch in ihren HTML-Templates einbauen, um diese zu nutzen.

Bootstrap ist das zurzeit am meisten verbreitete CSS-Framework. Es wurde 2011 von Twitter als Open Source Software verfügbar gemacht und die aktuelle Version ist 5.x. Im Microblog-Tutorial wird

allerdings noch Bootstrap 3 verwendet, dass aber bereits alles bietet was für die Beispiel-App gebraucht wird und dass über die Erweiterung Flask-Bootstrap sehr einfach zu nutzen ist.

Bootstrap besteht aus vielen CSS-Dateien, die Sie lokal auf Ihren Computer herunterladen könnten. Da man aber annehmen kann, dass ein Server mit einer Webapplikation sicher über eine Internet-Verbindung verfügt, können die Dateien auch über das Bootstrap Content Delivery Network (CDN) dynamisch geladen werden.

```
<link href="{{bootstrap_find_resource('css/bootstrap.css', cdn='bootstrap')}}" rel="stylesheet">
```

Dieser Ansatz wird für die Beispiele in diesem Kapitel genutzt.

12.3 VORHER UND NACHER

Die folgenden Beispiele auf der rechten Seite zeigen, wie die Seiten Ihrer App nach dem Einsatz von Flask-Bootstrap in diesem Kapitel aussehen werden:

The image shows two side-by-side screenshots of a web browser window titled "Sign In - Microblog". Both screenshots show the same "Sign In" form with fields for "Username" and "Password", a "Remember Me" checkbox, and a "Sign In" button. Below the form are links for "New User? Click to Register!" and "Forgot Your Password? Click to Reset It".

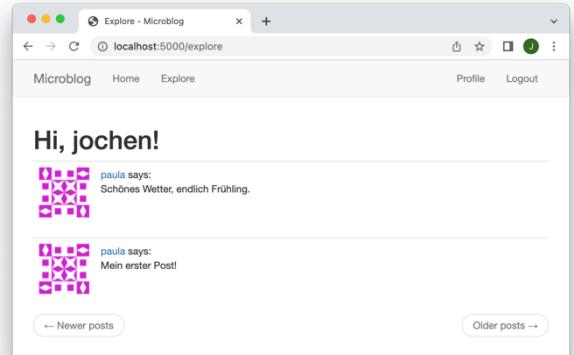
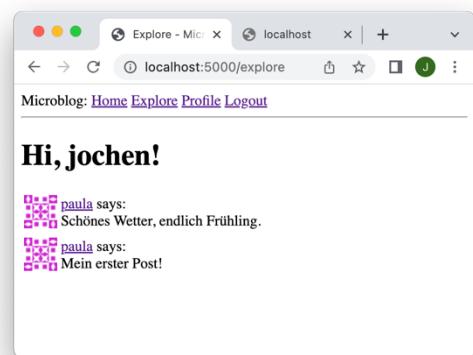
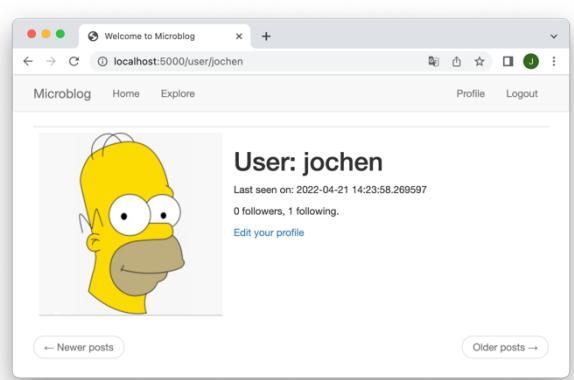
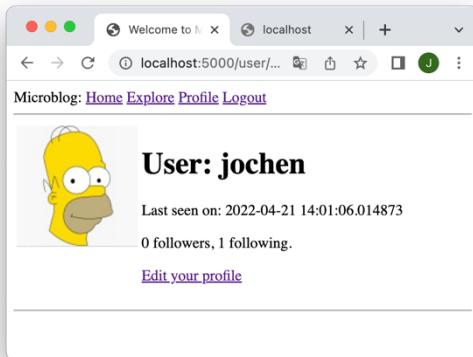
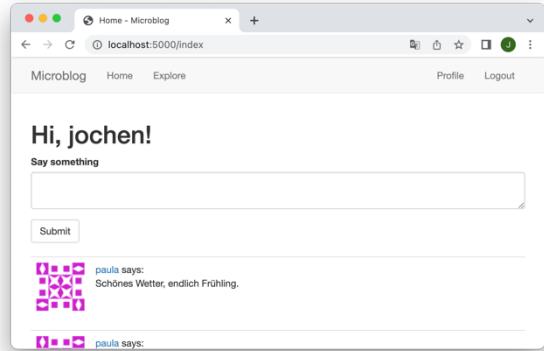
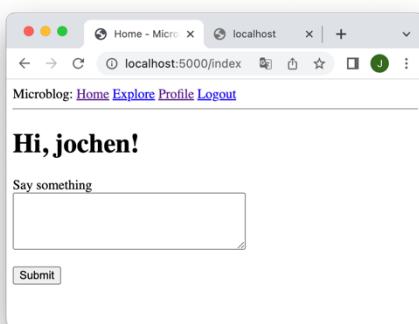
The left screenshot represents the application "before" using Flask-Bootstrap. The entire page has a plain, light gray background. The form fields are simple rectangular inputs.

The right screenshot represents the application "after" using Flask-Bootstrap. The page features a clean, modern design with a white background and blue accents. The "Sign In" button is highlighted with a blue border. The overall aesthetic is more professional and visually appealing.

The image shows two side-by-side screenshots of a web browser window titled "Register - Microblog". Both screenshots show the same "Register" form with fields for "Username", "Email", "Password", and "Repeat Password", along with a "Register" button.

The left screenshot represents the application "before" using Flask-Bootstrap. The page has a plain, light gray background. The form fields are simple rectangular inputs.

The right screenshot represents the application "after" using Flask-Bootstrap. The page features a clean, modern design with a white background and blue accents. The "Register" button is highlighted with a blue border. The overall aesthetic is more professional and visually appealing.



12.4 FLASK-BOOTSTRAP

Die Erweiterung Flask-Bootstrap stellt ein Basis-Template zur Verfügung, das die Bestandteile von Bootstrap 3 enthält. Sie installieren sie mit pip:

```
(tutorial) $ pip install flask-bootstrap
```

Im virtuellen Environment findet sich nach der Installation die Datei base.html unter

```
./tutorial/lib/python3.9/site-packages/flask_bootstrap/templates/bootstrap/
```

Der Dateipfad kann bei Ihrer Installation abweichen. Sie brauchen die Datei aber auch garnicht direkt öffnen, denn die Erweiterung wird in app/__init__.py importiert und initialisiert:

```
# ... weitere Zeilen ...
from flask_bootstrap import Bootstrap
app = Flask(__name__)
# ... weitere Zeilen ...

bootstrap = Bootstrap(app)

# ... weitere Zeilen ...
# letzte Zeile:
from app import routes, models, errors
```

Danach steht bootstrap/base.html in den Jinja2-Templates zur Verfügung. Bisher haben Sie die Templates Ihrer Seiten mit *extends* von Ihrem eigenen base.html abgeleitet. Um die Inhalte von bootstrap.html zur Verfügung zu haben leiten Sie nun neu app/templates/base.html von bootstrap/base.html ab. In base.html fügen sie ganz am Anfang die folgende Zeile ein:

```
{% extends 'bootstrap/base.html' %}
```

Dadurch vermeiden Sie, dass sie alle bestehenden Templates bearbeiten müssen, um Bootstrap zu benutzen. Alle bisherigen Templates für Ihre Seiten erben von app/templates/base.html. Dadurch, dass app/templates/base.html selbst von bootstrap/base.html erbt, erben auch alle anderen Templates die Elemente von bootstrap/base.html.

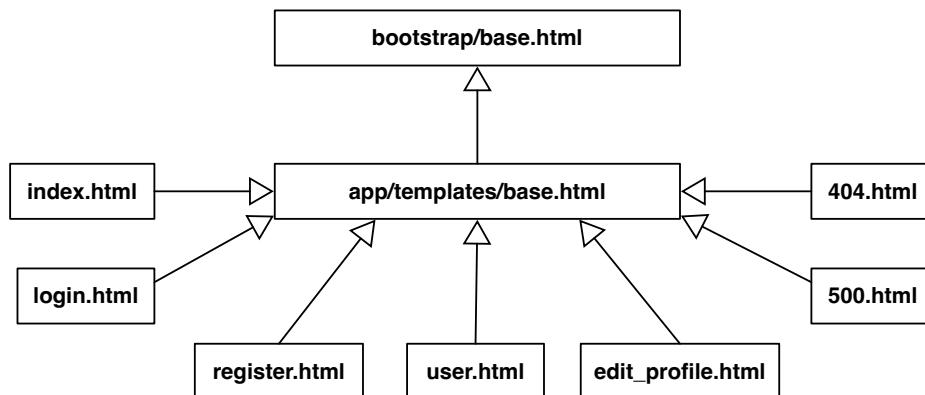


Abbildung 12-1 Alle Templates erben von bootstrap/base.html

bootstrap/base.html ist in einer Hierarchie von Blöcken strukturiert. Der äusserste Block ist doc. Er enthält den Block html_attribs für Attribute wie language=en im <html>-Tag und den Block html, der

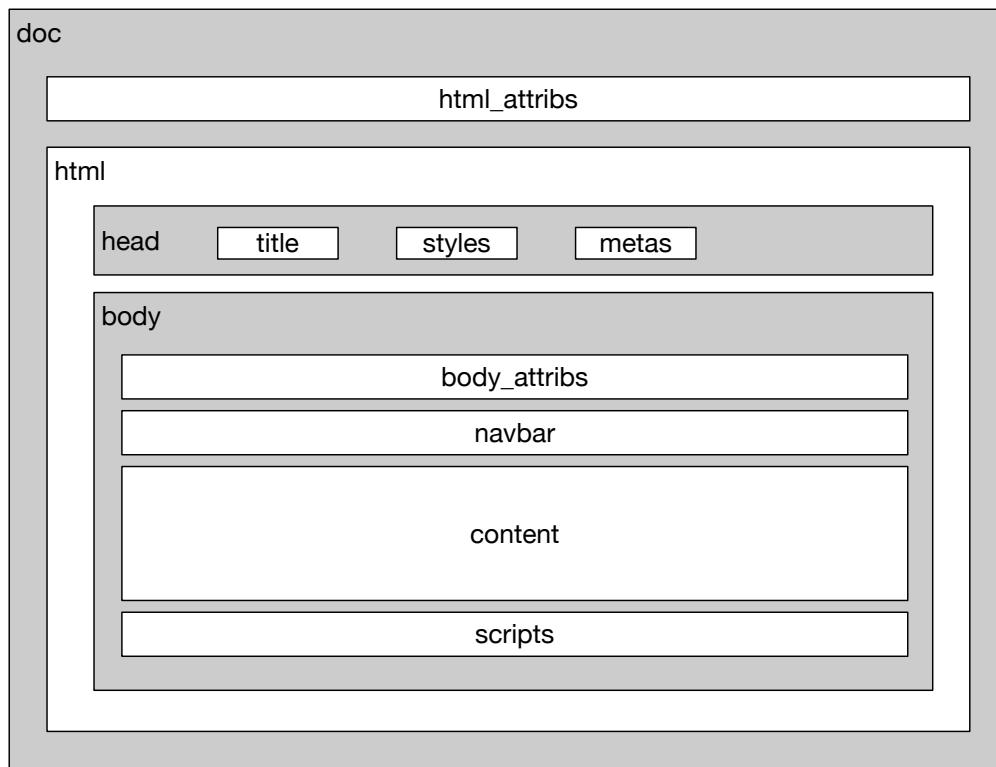


Abbildung 12-2 Blockstruktur von flask-bootstrap

dem gesamten <html> Element einer Seite entspricht.

Im Block **html** befinden sich head, was dem <head> Element einer Seite entspricht, und **body**, was dem <body>-Element einer Seite entspricht.

Im Block **head** gibt es die Unter-Blöcke

- title für den Inhalt des <title> Elements
- styles enthält Links zu CSS-Styles für das <head> Element
- metas enthält das, was im <meta> Element vorkommt, wie z.B. die Zeichensatzkodierung

Im Block **body**, der dem <body> Element entspricht, finden sich die Unter-Blöcke:

- body_attribs für die Angabe von Attributen beim <body>-Tag
- navbar ist ein leerer Block oben auf der Seite
- content ist für den eigentlichen Inhalt der Seite gedacht
- scripts kann alle <script>-Tags für JavaScript aufnehmen

Diese Blöcke müssen in app/templates/base.html implementiert werden:

```
{% extends 'bootstrap/base.html' %}

{% block title %}
    {% if title %}{{ title }} - Microblog{% else %}Welcome to Microblog{% endif %}
{% endblock %}

{% block navbar %}
<nav class="navbar navbar-default">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle collapsed"
                data-toggle="collapse"
                data-target="#bs-example-navbar-collapse-1"
                aria-expanded="false">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="{{ url_for('index') }}>Microblog</a>
        </div>
        <div class="collapse navbar-collapse"
            id="bs-example-navbar-collapse-1">
            <ul class="nav navbar-nav">
                <li><a href="{{ url_for('index') }}>Home</a></li>
                <li><a href="{{ url_for('explore') }}>Explore</a></li>
            </ul>
            <ul class="nav navbar-nav navbar-right">
                {% if current_user.is_anonymous %}
                    <li><a href="{{ url_for('login') }}>Login</a></li>
                {% else %}
                    <li><a href="{{ url_for('user',
                        username=current_user.username) }}>Profile</a></li>
                    <li><a href="{{ url_for('logout') }}>Logout</a></li>
                {% endif %}
            </ul>
        </div>
    </div>
</nav>
{% endblock %}

{% block content %}
    <div class="container">
        {% with messages = get_flashed_messages() %}
            {% if messages %}
                {% for message in messages %}
                    <div class="alert alert-info" role="alert">{{ message }}</div>
                {% endfor %}
            {% endif %}
        {% endwith %}

        {% block app_content %}{% endblock %}
    </div>
{% endblock %}
```

Es werden vordefinierte Elemente (<nav>, <button>) und CSS-Klassen ("navbar navbar-default", "container", "navbar-header", usw.) aus Bootstrap verwendet.

Details zu CSS erklären, würde den Rahmen dieses Kurses sprengen und die im Kapitel 1 vermittelten Kenntnisse zu CSS reichen zum vollen Verständnis nicht aus. Übernehmen Sie daher am besten app/templates/base.html am aus den Musterlösungen. Hier gibt es aber ein paar Anmerkungen dazu:

In der 1. Zeile wird festgelegt, dass das template vom Bootstrap-Basitemplate erbt.

Im **{ block title }** wird die gleiche Logik wie im alten base.html verwendet: Wenn die Variable title existiert wird der Inhalt sowie ' - Microblog' angezeigt, ansonsten wird der feste Text 'Welcome to Microblog' angezeigt.

Im Abschnitt **{% block navbar %}** werden CSS-Klassen aus Bootstrap verwendet, um die Navigationslinks oben auf der Seite darzustellen. Der Code wurde aus der Bootstrap-Dokumentation für den default-navbar-Block¹⁷ übernommen und leicht abgeändert. Darin werden der Text 'Microblog' und die Links auf die weiteren Seiten wie folgt platziert:

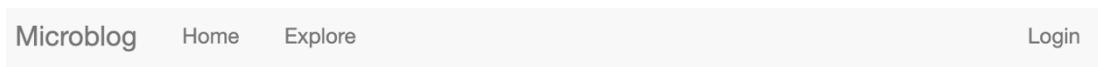


Abbildung 12-3 Layout Navbar

Das Layout mit Bootstrap ist *responsive*. Das heisst, es reagiert auf die verfügbare Bildschirmbreite. Sie sehen das im folgenden Beispiel. Die Links oben auf den Seiten werden nur bei genügender Breite nebeneinander dargestellt. Wenn die Seite auf einem Handy dargestellt oder wenn Sie das Browser-Fenster verkleinern, klappen die Links ein und können über ein 'Hamburger' Menu erreicht werden:

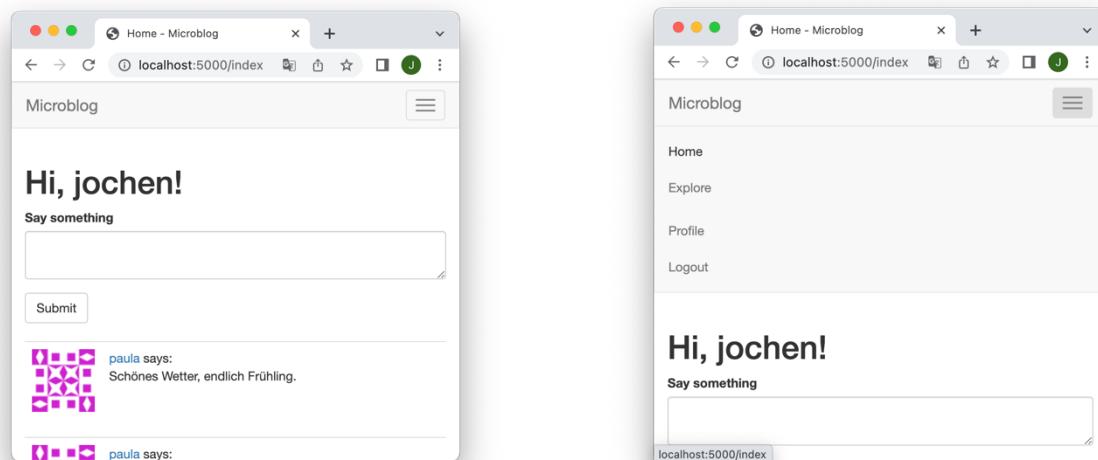


Abbildung 12-4 Der Bootstrap Navbar ist responsive

Der Abschnitt **{% block content %}** ist ein Container, in dem ebenfalls CSS-Klassen aus Bootstrap verwendet werden. Er enthält zwei Unterabschnitte:

- Einen Bereich für die Anzeige von Meldungen. Diese werden nun als Bootstrap-Alerts angezeigt (ein weiteres Element, das Bootstrap zur Verfügung stellt).

¹⁷ <https://bootstrapdocs.com/v3.0.3/docs/components/#navbar>

- Den `{% block app_content %}`. Dessen Inhalt muss von den jeweiligen Templates der verschiedenen Seiten zur Verfügung gestellt werden.

Den Namen `{% block content %}` haben Sie allerdings bereits in den verschiedenen Templates der letzten Kapitel verwendet und er ist nun durch Flask-Bootstrap bereits vergeben. Daher müssen alle Vorkommen von `{% block content %}` in den verschiedenen Seiten-Templates durch `{% block app_content %}` ersetzt werden. Machen Sie das bitte in den folgenden Dateien:

```
app/templates/404.html
app/templates/500.html
app/templates/edit_profile.html
app/templates/index.html
app/templates/login.html
app/templates/register.html
app/templates/reset_password.html
app/templates/reset_password_request.html
app/templates/user.html
```

12.5 FORMULARE MIT BOOTSTRAP

Ein Bereich, in dem Bootstrap die Arbeit stark vereinfacht, ist die Darstellung von Formularen. In der bisherigen Version der Templates für Formulare musste das Styling jedes einzelnen Feldes separat erfolgen. Hier ein Auschnitt aus `app/templates/register.html`:

```
<h1>Register</h1>
<form action="" method="post">
    {{ form.hidden_tag() }}
    <p>
        {{ form.username.label }}<br>
        {{ form.username(size=32) }}<br>
        {% for error in form.username.errors %}
            <span style="color: red;">[{{ error }}]</span>
        {% endfor %}
    </p>
    <p>
        {{ form.email.label }}<br>
        {{ form.email(size=64) }}<br>
        {% for error in form.email.errors %}
            <span style="color: red;">[{{ error }}]</span>
        {% endfor %}
    </p>
    <p>
        {{ form.password.label }}<br>
        {{ form.password(size=32) }}<br>
        {% for error in form.password.errors %}
            <span style="color: red;">[{{ error }}]</span>
        {% endfor %}
    </p>
    <p>
        {{ form.password2.label }}<br>
        {{ form.password2(size=32) }}<br>
        {% for error in form.password2.errors %}
            <span style="color: red;">[{{ error }}]</span>
        {% endfor %}
    </p>
    <p>{{ form.submit() }}</p>
</form>
```

Mit Flask-Bootstrap steht ein Makro `quick-form()` zur Verfügung, mit dem ein komplettes Formular gerendert werden kann. Sie können `app/templates/register.html` nun stark vereinfachen:

```

{% extends "base.html" %}
{% import 'bootstrap/wtf.html' as wtf %}

{% block app_content %}
    <h1>Register</h1>
    <div class="row">
        <div class="col-md-4">
            {{ wtf.quick_form(form) }}
        </div>
    </div>
{% endblock %}

```

Das Template ist nun auf lediglich 10 Zeilen geschrumpft. Sämtliche Validierungen und Fehlermeldungen werden jetzt durch das Makro quick_form() verwaltet und dargestellt.

Diese Vereinfachung sollte auch in den anderen Templates genutzt werden. Den Code finden Sie in den angepassten Dateien aus der Musterlösung:

```

app/templates/edit_profile.html
app/templates/index.html
app/templates/login.html
app/templates/reset_password.html
app/templates/reset_password_request.html

```

12.6 LISTE DER POSTS MIT BOOTSTRAP

Die Darstellung der Liste der Blog-Posts haben Sie in Kapitel 7 in das Untertemplate _posts.html ausgelagert und es auf verschiedenen Seiten mit include wieder verwendet. Es muss noch angepasst werden, damit es mit Bootstrap gut aussieht. Es soll nun so aussehen:

```

<<!-- app/templates/_posts.html -->





```

In Kapitel 10 haben Sie eine Seitennavigation für die Blog-Posts eingerichtet, um die Navigation in langen Listen benutzerfreundlicher zu machen und um zu vermeiden, dass die Performance der Seiten bei sehr langen Listen schlecht ist.

Die Paginierungs-Links können Sie ebenfalls mit Bootstrap verbessern. In der bisherigen Version haben Sie den Link mit dem Text 'Newer Posts' nur dann angezeigt, falls es Vorgänger in der Liste gibt und den Link 'Older Posts' nur, falls es Nachfolger gibt.

```

18  {% for post in posts %}
19    |   {% include '_post.html' %}
20    |   {% endfor %}
21    |   {% if prev_url %}
22    |     <a href="{{ prev_url }}>Newer posts</a>
23    |   {% endif %}
24    |   {% if next_url %}
25    |     <a href="{{ next_url }}>Older posts</a>
26    |   {% endif %}
27   {% endblock %}
--
```

Abbildung 12-5 Logik der Paginierung (alt)

Mit Bootstrap können Sie die Vorgänger- und Nachfolger-Links/Buttons einfach deaktivieren statt sie verschwinden zu lassen. Sie werden dann als ausgegraute Buttons angezeigt, die inaktiv sind.



Abbildung 12-6 Deaktivierte Buttons

Hier wird das am Beispiel von app/templates/index.html gezeigt (Der Code wurde aus der Dokumentation von Bootstrap 3¹⁸ übernommen und angepasst):

```

{% block app_content %}
  <h1>Hi, {{ current_user.username }}!</h1>
  {% if form %}
    {{ wtf.quick_form(form) }}
    <br>
  {% endif %}
  {% for post in posts %}
    {% include '_post.html' %}
  {% endfor %}
  <nav aria-label="...">
    <ul class="pager">
      <li class="previous{% if not prev_url %} disabled{% endif %}">
        <a href="{{ prev_url or '#' }}>
          <span aria-hidden="true">&larr;</span> Newer posts
        </a>
      </li>
      <li class="next{% if not next_url %} disabled{% endif %}">
        <a href="{{ next_url or '#' }}>
          Older posts <span aria-hidden="true">&rarr;</span>
        </a>
      </li>
    </ul>
  </nav>
  {% endblock %}
```

Der Code ist jetzt etwas umfangreicher geworden, aber die Seite sieht wesentlich besser aus.

¹⁸ <https://getbootstrap.com/docs/3.3/components/#optional-disabled-state>

Die zweite Seite, auf der Sie dieses Vorgehen anwenden können, ist die Profilseite app/templates/user.html. Ersetzen Sie dort die Zeilen nach `{% endfor %}`:

```
# ... Zeilen ausgelassen ...
{% for post in posts %}
    {% include '_post.html' %}
{% endfor %}
# Neu:
<nav aria-label="...">
    <ul class="pager">
        <li class="previous{% if not prev_url %} disabled{% endif %}">
            <a href="{{ prev_url or '#' }}">
                <span aria-hidden="true">&larr;</span> Newer posts
            </a>
        </li>
        <li class="next{% if not next_url %} disabled{% endif %}">
            <a href="{{ next_url or '#' }}">
                Older posts <span aria-hidden="true">&rarr;</span>
            </a>
        </li>
    </ul>
</nav>
{% endblock %}
```

12.7 ZUSAMMENFASSUNG

Mit der Anwendung der Bootstrap-Elemente haben Sie das Aussehen der App professioneller gemacht. Dabei wurden lediglich die HTML-Templates angepasst. Alle anderen Teile der App sind unverändert geblieben. Das entspricht dem Architektur-Prinzip 'Separation of Concerns' (Trennung der Zuständigkeiten). Das Ziel ist dabei, die die Geschäftslogik unabhängig von der Darstellung der Seiten zu machen:

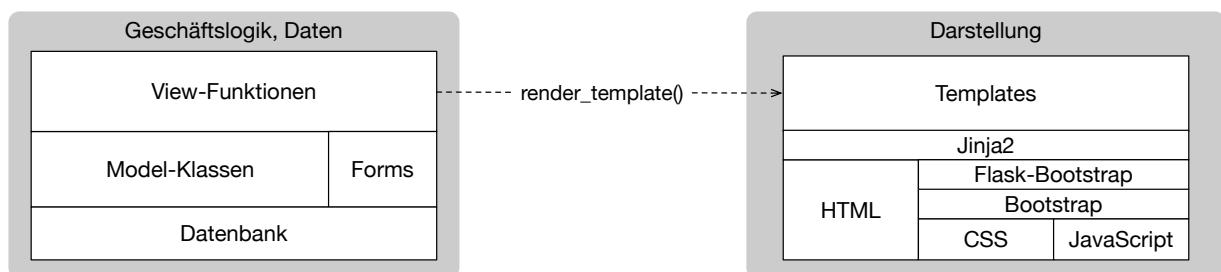


Abbildung 12-7 Trennung von Geschäftslogik und Darstellung im Client

- Das Design einer Webapplikation wird über CSS gesteuert.
- CSS-Frameworks wie Bootstrap stellen eine Sammlung von Layouts, Styles und Elementen der Benutzeroberfläche zur Verfügung. Sie ermöglichen auch auf einfache Art, Seiten responsive zu gestalten. Das heisst, die Darstellung passt sich an die Fenstergrösse an (Browser auf dem PC, Handy, Tablet).
- Bootstrap 3.x wird über die Erweiterung Flask-Bootstrap eingebunden

- Um das Framework zu nutzen ist es nicht nötig, detaillierte Kenntnisse von CSS zu haben. Die Beispiele in diesem Kapitel wurden mit Hilfe der Codebeispiele aus der Dokumentation von Bootstrap 3 erstellt.
- Das Basis-Template von Flask-Bootstrap teilt das Seitenlayout in vordefinierte Blöcke auf. Innerhalb dieser Blöcke kann die App eigene Blöcke platzieren.
- Die Grundstruktur wird über den extends-Mechanismus von Jinja2 in die Templates der einzelnen Seiten übernommen.

13 DEPLOYMENT

Bisher haben Sie die App stetig weiterentwickelt und auf Ihrem lokalen Rechner getestet. Sobald Sie mit dem Entwicklungsstand zufrieden sind, möchten Sie sie sicher im Internet öffentlich zugänglich machen.

13.1 LERNZIELE

Nach der Bearbeitung des Kapitels können Sie:

- Verschiedene für den produktiven Einsatz geeignete Basis-Komponenten (DBMS, WSGI-Server, Web-Server, Mail-System und Management-Software) auf einem Server installieren und konfigurieren.
- Den Code der App auf einen Server kopieren und sie dort für einen weitgehend automatischen Betrieb einrichten.
- Die Komponenten Ihrer App mit git versionieren und für das Server-Deployment auf GitHub bereitstellen.

13.2 DEPLOYMENT-OPTIONEN

Grundsätzlich haben Sie verschiedene Optionen für die produktive Bereitstellung Ihrer App im Internet:

1. Traditionelles Deployment auf einem physischen Server oder in einer virtuellen Maschine in einer öffentlichen Cloud-Infrastruktur
2. Deployment mit Containern
3. Deployment mittels 'Serverless' oder 'Platform as a Service' (PaaS) Dienstleister

In diesem Kapitel wird die Variante 1 behandelt. Einen Überblick über die beteiligten Komponenten finden Sie in der folgenden Abbildung:

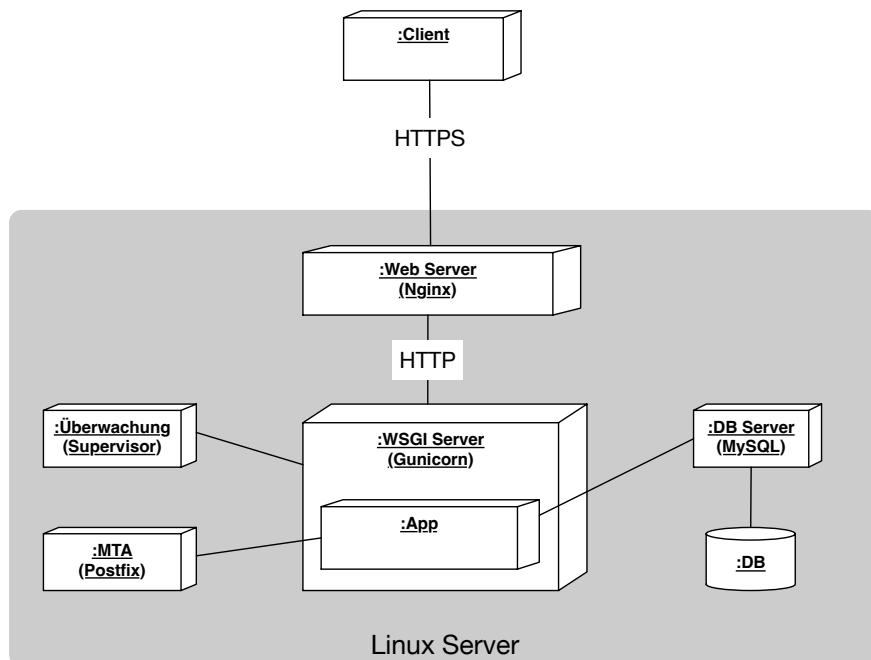


Abbildung 13-1 Deployment auf Linux

13.3 DEPLOYMENT AUF LINUX

In diesem Kapitel erfahren Sie, wie sie Ihre App auf einer virtuellen Maschine mit dem Betriebssystem Ubuntu 20.04 LTS bereitstellen können.

Die Schritte dafür sind:

- Login auf der VM per SSH
- Installation der benötigten Software
- Deployment Ihres Codes
- Konfiguration des Webservers
- Konfiguration des WSGI-Servers
- Konfiguration des Datenbanksystems

Login per ssh

Melden Sie sich aus einem Terminal per ssh bei der vorbereiteten Linux-VM an:

```
$ ssh student@<IP-ADRESSE>
```

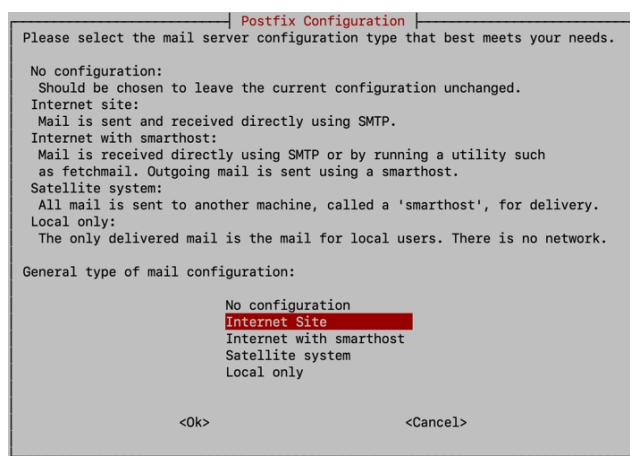
Die IP-Adresse und das Passwort erhalten Sie von Ihrem Dozenten.

Softwareinstallation

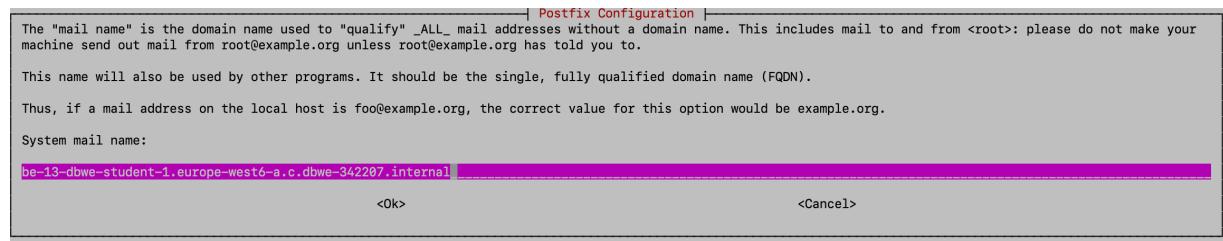
Sie benötigen Python, MySQL, den Mail Transfer Agent Postfix, den Webserver Nginx und git. Führen Sie folgende Kommandos aus:

```
$ sudo apt update  
$ sudo apt install python3 python3-venv python3-dev  
$ sudo apt install mysql-server postfix mailutils supervisor nginx git  
$ sudo apt upgrade
```

Während der Installation von Postfix werden Sie gefragt, wie postfix konfiguriert werden soll. Wählen Sie hier 'Internet Site'.



In der nächsten Abfrage übernehmen Sie die vorgeschlagene Domain



Falls hier kein Vorschlag erscheint, nehmen Sie den Output des Kommandos `dnsdomainname`.

Die restliche Konfiguration von Postfix und der anderen Produkte wird in den folgenden Unterkapiteln separat beschrieben.

13.3.1 DEPLOYMENT DER APPLIKATION

Ihre App entwickeln und testen Sie bisher lokal auf Ihrem Rechner. Gehen Sie dort ins Projektverzeichnis, aktivieren Sie Ihr virtuelles Environment und erzeugen Sie eine Datei, die alle benötigten Pakete auflistet:

```
(tutorial) $ pip freeze > requirements.txt
```

Eine einfache Methode, Ihren Code vom Laptop auf den Server zu bekommen ist, das gesamte Projektverzeichnis von Ihrem Laptop mit dem scp-Kommando in das Home-Directory von student auf dem Server zu kopieren.:

```
$ scp -r microblog student@<IP-ADRESSE>:~/microblog
```

Melden Sie sich anschliessend erneut auf dem Server an, gehen sie ins Verzeichnis /home/student/microblog und löschen Sie einige Unterverzeichnisse und Dateien, die nicht gebraucht werden:

- Das Unterverzeichnis für das virtuelle Environment (Das Environment wird auf dem Server neu erzeugt). Im Beispiel hat das Environment den Namen 'venv':

```
student@vm:~$ cd microblog
student@vm:~/microblog$ rm -rf tutorial
```
- Alle Unterverzeichnisse mit dem Namen __pycache__ im Projektverzeichnis:

```
student@vm:~/microblog$ find . -name __pycache__ -exec rm -rf {} \;
```
- Die Datei app.db (SQLite wird auf dem Server nicht benutzt):

```
student@vm:~/microblog$ rm -f app.db
```

Anschliessend bauen Sie das Virtuelle Environment mithilfe von requirements.txt wieder neu auf. Das ist sinnvoll, weil Sie Versionen für die dort installierten Pakete für das Betriebssystem Linux brauchen:

```
student@vm:~/microblog$ python3 -m venv venv
student@vm:~/microblog$ source venv/bin/activate
```

```
(venv) student@vm:~/microblog$ pip3 install -r requirements.txt
```

Installieren Sie im Environment nun noch Gunicorn, einen WSGI-Server für den produktiven Einsatz, den Python-Connector für MySQL und cryptography, das für die Authentisierung bei MySQL gebraucht wird:

```
(venv) student@vm:~/microblog$ pip3 install gunicorn pymysql cryptography
```

Damit ist Ihre Kopie des Applikations-Code vorerst bereit. Es müssen in den nächsten Abschnitten müssen Sie noch einige kleinere Ergänzungen in config.py für die Datenbank-Verbindung für und das Versenden von E-Mails machen.

13.3.2 KONFIGURATION VON MYSQL

In MySQL müssen Sie eine Datenbank mit dem Namen microblog anlegen, einen User-Account mit dem Namen microblog für die App erzeugen und Ihm alle Berechtigungen für die Datenbank geben:

```
student@vm:~/microblog$ sudo mysql -u root
```

```
mysql> create database microblog character set utf8 collate utf8_bin;
mysql> create user 'microblog'@'localhost' identified by 'Ihr Passwort';
mysql> grant all privileges on microblog.* to 'microblog'@'localhost';
mysql> flush privileges;
mysql> exit;
```

Das von Ihnen gewählte Passwort an der Stelle *Ihr Passwort* tragen Sie in der .env-Datei ein.

Nun müssen die Tabellen noch mit Flask-Migrate erzeugt werden:

```
student@vm:~/microblog$ source venv/bin/activate
(venv) student@vm:~/microblog$ vi
```

Prüfen Sie allenfalls, ob User und Passwort funktionieren und ob die Tabellen korrekt angelegt wurden:

```
(venv) student@vm:~/microblog$ sudo mysql -u microblog -p
```

```
mysql> use microblog;
mysql> show tables;
mysql> exit;
```

Sie haben auf Ihrem Laptop wahrscheinlich SQLite als Datenbank-Server verwendet. Damit die App mit MySQL zusammenarbeitet, müssen Sie in config.py diesen Eintrag ändern:

```
# Alt:
SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
    'sqlite:///` + os.path.join(basedir, 'app.db')
# Neu:
SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')
```

Da es auf einem Produktions-Server nicht in Frage kommt, SQLite zu benutzen, ist der Teil nach or jetzt überflüssig. SQLALCHEMY_DATABASE_URI wird nun immer der Inhalt der Umgebungsvariable DATABASE_URL zugewiesen.). Der Grund dafür ist, dass SQLALCHEMY_DATABASE_URI für MySQL ein Passwort enthalten muss. Vertrauliche Informationen wie diese sollten nie in einer Quellcode-Datei stehen! Wie Sie Umgebungsvariablen auf dem Server automatisch beim Start der App setzen, wird in Kap 0 beschrieben.

13.3.3 KONFIGURATION VON POSTFIX

Die App verschickt E-Mail in zwei Fällen:

1. Ein Fehler soll an den Administrator gemeldet werden
2. Ein Token für den Passwort Reset soll an einen Benutzer geschickt werden

Emails werden über den Mail Transfer Agent (MTA) postfix an den eigentlichen Mail-Server weitergeleitet.

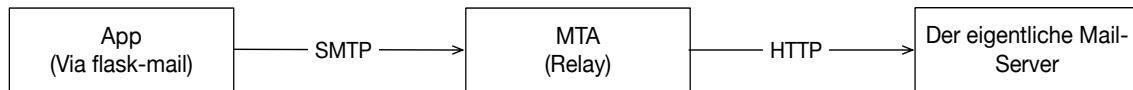


Abbildung 13-2 Ein MTA als Relay-Server

Postfix (der MTA in diesem Beispiel) wird dabei als Relay-Server konfiguriert. E-Mail soll ausschliesslich gesendet, aber nicht empfangen werden.

Die weitere Konfiguration erfolgt über die Datei /etc/postfix/main.cf

```
$ sudo vi /etc/postfix/main.cf      # Oder mit einem anderen Editor wie nano
```

Fügen Sie dort die folgenden Einträge hinzu oder passen Sie die folgenden Einträge an, falls sie schon existieren:

```
inet_interfaces = loopback-only
mydestination = $myhostname, localhost.<HOSTNAME>.<DOMAIN>, \
                <HOSTNAME>.<DOMAIN>, localhost
#Für <HOSTNAME> benutzen Sie die Ausgabe von hostname
#Für <DOMAIN> benutzen Sie die Ausgabe von dnsdomainname
relayhost = [<ZIEL-MAILSERVER>]:587
# Für <ZIEL-MAILSERVER> benutzen Sie den Mail-Server, an
# den Sie E-Mail weiterleiten wollen, z.B.:
# ipso-ch.mail.protection.outlook.com

# SASL Authentifizierung aktivieren
smtp_sasl_auth_enable = yes
# Keine anonyme Authentifizierung erlauben
smtp_sasl_security_options = noanonymous
# Ort für Datei sasl_passwd
smtp_sasl_password_maps = hash:/etc/postfix/sasl_passwd
# STARTTLS Verschlüsselung aktivieren
smtp_use_tls = yes
# Wo können Zertifikate gefunden werden
smtp_tls_CAfile = /etc/ssl/certs/ca-certificates.crt
```

Der Username und das Passwort wird anschliessend konfiguriert:

Zuletzt starten Sie den Dienst Postfix neu und testen das Versenden einer Email

13.3.4 UMGEBUNGSVARIABLEN SETZEN

Für die Umgebungsvariablen, die in config.py gelesen werden, legen Sie im Projektverzeichnis eine Datei .env an. Sie hat den Inhalt:

```
SECRET_KEY=49347611a88144e5a7d964c7d8f75506 # Ein zufälliger String
```

```

MAIL_SERVER=localhost
MAIL_PORT=587
DATABASE_URL=mysql+pymysql://microblog:<MYSQL-PASSWORT>@localhost:3306/microblog
# Das Passwort haben Sie im vorherigen Abschnitt in mysql gesetzt
MAIL_SERVER=<MAILSERVER>      # siehe Postfix-Konfiguration
MAIL_PORT=587
MAIL_USE_TLS=1
MAIL_USERNAME=<MAIL-ACCOUNT> # z.B. vorname.nachname@student.ipso.ch
MAIL_PASSWORD=<PASSWORD>      # Ihr Passwort für den Account

```

Damit diese Datei beim Start der App gelesen wird, fügen Sie folgende Zeilen in config.py hinzu:

```

from dotenv import load_dotenv
basedir = os.path.abspath(os.path.dirname(__file__))
load_dotenv(os.path.join(basedir, '.env'))
# Weitere Zeilen ...

```

Eine solche .env-Datei und die Funktion load_dotenv() in config.py auch auf Ihrem Laptop zu benutzen ist eine gute Idee, aber bedenken Sie bitte, dass sie dort allenfalls andere Einträge benötigen, falls Sie SQLite für die Entwicklung benutzen.

Die Variable FLASK_APP wird nicht in .env, sondern in ihrem .profile im Home-Verzeichnis gesetzt, da sie schon beim Start der App bekannt sein muss. Führen Sie dafür folgendes Kommando aus:

```
student@vm:~/microblog$ echo "export FLASK_APP=microblog.py" >> ~/.profile
```

Oder tragen Sie die Zeile 'export FLASK_APP=microblog.py' mit einem Editor in ~/.profile ein.

13.3.5 KONFIGURATION VON GUNICORN UND SUPERVISOR

Gunicorn wird in der Produktionsumgebung statt dem eingebauten WSGI-Webserver von Flask benutzt. Statt mit *flask run* kann die App wie folgt gestartet werden:

```
(venv) student@vm:~/microblog$ gunicorn -b localhost:8000 -w 4 microblog:app
```

Die Option -b gibt an, von welchem Host und auf welchem Port Gunicorn http-Requests empfangen soll. Die Option -w gibt die Anzahl *worker-Prozesse* an, die http-Requests verarbeiten können. Mit vier *workers* können gleichzeitig vier Requests verarbeitet werden. Das Argument microblog:app gibt den Namen des Moduls (microblog.py) und den Namen des Pakets app an

Brechen Sie den gunicorn-Server mit CTRL-C wieder ab, denn ein manueller Start ist bei einem produktiven Server unpraktisch. Stattdessen soll das Tool Supervisor eingesetzt werden, mit dem der Gunicorn beim Booten automatisch gestartet, laufend überwacht und nach Unterbrüchen wieder neu gestartet werden kann.

Supervisor wird über Dateien im Verzeichnis */etc/supervisor/conf.d* gesteuert. Legen Sie dort eine Datei *microblog.conf* an. Sie soll den folgenden Inhalt haben:

```

[program:microblog]
command=/home/student/microblog/venv/bin/gunicorn -b \
          localhost:8000 -w 4 microblog:app
directory=/home/student/microblog
user=student
autostart=true
autorestart=true
stopasgroup=true

```

```
killasgroup=true
```

- Die Einträge *command*, *directory* und *user* beschreiben, wie die App gestartet werden kann.
- Der Eintrag *autostart=true* sorgt dafür, dass die App bei einem Reboot des Servers nicht manuell gestartet werden muss.
- Der Eintrag *autorestart=true* bewirkt, dass *command* erneut ausgeführt werden soll, falls Supervisor bemerkt, dass gunicorn nicht mehr läuft
- Mit *stopasgroup=true* wird festgelegt, dass bei einem Stop auch alle Unterprozesse von gunicorn (Die worker-Prozesse) gestoppt werden.
- Mit *killasgroup=true* wird festgelegt, dass bei einem gewollten Abbruch auch alle Unterprozesse von gunicorn (Die worker-Prozesse) abgebrochen werden.

13.3.6 KONFIGURATION VON NGINX

Nginx und Apache sind die Webserver, die am häufigsten verwendet werden, um Anfragen (Requests) zu empfangen und Webseiten (Responses) an den Browser eines Benutzers zu übermitteln. Diese Kommunikation soll verschlüsselt über das Protokoll HTTPS erfolgen.

In diesem Kapitel wird Nginx als *Proxy-Server* vor dem WSGI-Server Gunicorn eingesetzt. Das heisst, der Client schickt seine Requests verschlüsselt per SSL / HTTPS (Port 443) oder unverschlüsselt mit HTTP (Port 80) an den Prox-Server und erhält Responses verschlüsselt über HTTPS.

Benutzt ein Client unverschlüsseltes HTTP, sollte der Webserver die Requests immer auf HTTPS umleiten. Der Proxy-Server *terminiert* die HTTPS-Kommunikation. Intern leitet er die Requests zum WSGI-Server Gunicorn weiter, der die App bereitstellt. Gunicorn reagiert auf Requests auf Port 8000 (HTTP).

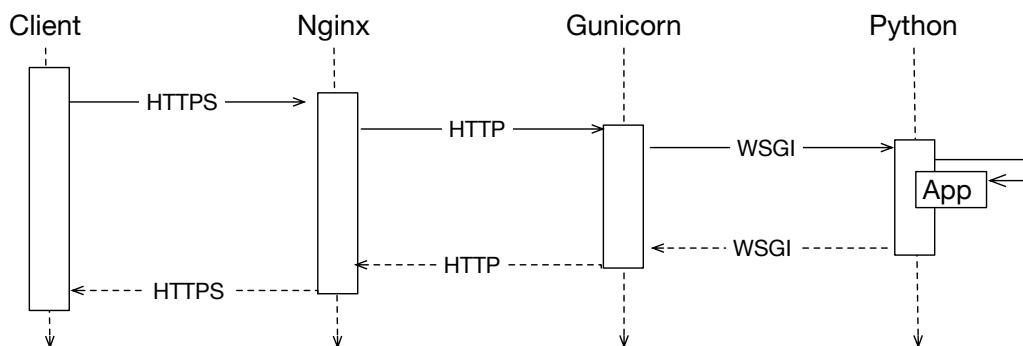


Abbildung 13-3 Ablauf Request/Response

Erzeugen Sie nun eine Konfigurationsdatei für Nginx

```
student@vm:~$ sudo vi /etc/nginx/sites-enabled/microblog
```

Den Inhalt der Datei können Sie hier übernehmen:

```
server {  
    # Für Requests auf Port 80 (http)  
    listen 80;  
    server_name _;
```

```

        location / {
            # Redirect von http-Requests an die gleiche URL,
            # aber per https
            return 301 https://$host$request_uri;
        }
    }
server {
    # Für Requests auf Port 443 (https)
    listen 443 ssl;
    server_name _;
    # Speicherort des SSL-Zertifikats
    ssl_certificate /home/student/microblog/certs/cert.pem;
    ssl_certificate_key /home/student/microblog/certs/key.pem;
    # Zugriffs- und Fehlerlogs sollen in /var/log geschrieben werden
    access_log /var/log/microblog_access.log;
    error_log /var/log/microblog_error.log;
    location / {
        # Weitergabe von Requests an Gunicorn
        proxy_pass http://localhost:8000;
        proxy_redirect off;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
    location /static {
        # Statische Files (z.B. CSS-Dateien, Bilder ...) soll
        # Nginx direkt bedienen und nicht an die App weiterleiten
        alias /home/student/microblog/app/static;
        expires 30d;
    }
}

```

13.4 VERSIONIERUNG UND DEPLOYMENT MIT GIT

Falls Sie den Code lokal weiterentwickeln, wird es irgendwann schwierig, die Übersicht zu behalten, welche Version auf dem Server vorliegt. Daher empfiehlt es sich, den Code mit Git zu versionieren und zu ein im Internet erreichbares Repository zu verwenden, von dem aus Ihr aktueller Code auf den Webserver synchronisiert werden kann.

Sie sollten zunächst einen kostenlosen privaten Account auf GitHub, Bitbucket oder GitLab einrichten, falls Sie noch keinen besitzen. Starten Sie hier für GitHub.

13.4.1 GITHUB ACCOUNT ANLEGEN

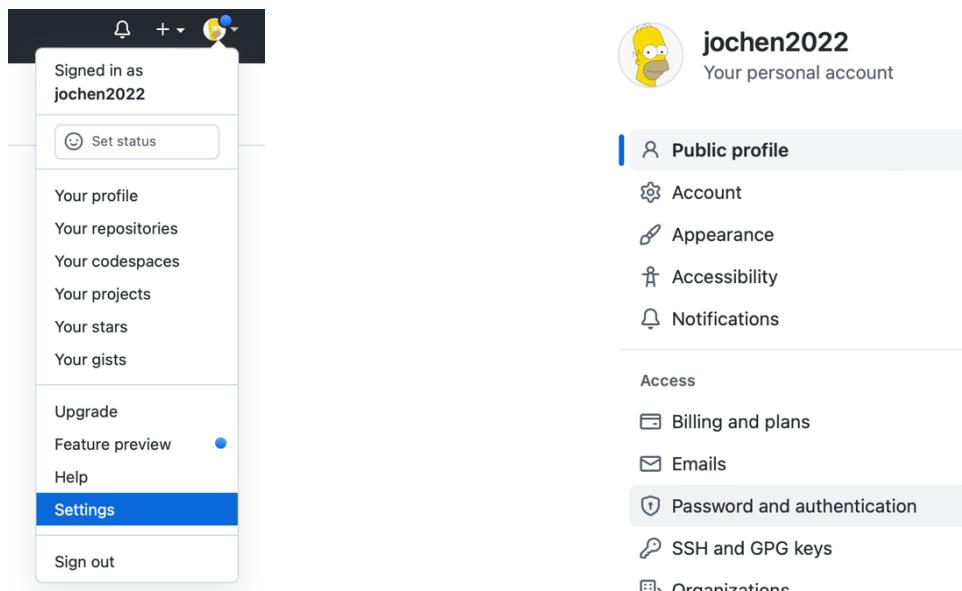
<https://github.com/join>

- Sie müssen einen Usernamen, eine E-Mail Adresse und ein Passwort auswählen und eine CAPTCHA-Aufgabe lösen
- Sie bekommen eine Email mit einer PIN, die sie bei GitHub eingeben
- Sie beantworten einige Fragen
- Sie wählen 'Continue for free', um einen kostenlosen Account anzulegen

13.4.2 2-FACTOR AUTHENTISIERUNG EINRICHTEN

GitHub empfiehlt, eine 2-Factor Authentisierung mit dem Passwort und einen Code, den Sie mit Apps wie 1Password, Authy, Microsoft Authenticator oder SMS empfangen können. Falls Sie das wünschen, fahren Sie mit den Schritten in diesem Abschnitt fort.

Im Menu unter dem Account-Symbol wählen Sie 'Settings'. In den Settings finden Sie in der Seiteleiste den Link 'Password and Authentication':



Dort finden Sie 'Enable two-factor authentication'. Wählen Sie das aus und folgen Sie den Anweisungen

Two-factor authentication



Two factor authentication is not enabled yet.

Two-factor authentication adds an additional layer of security to your account by requiring more than just a password to sign in.

[Enable two-factor authentication](#)

[Learn more](#)

Entscheiden Sie Sich, ob Sie Zugangscodes über eine App oder SMS erhalten wollen.

The screenshot shows the first step of a two-step wizard for setting up two-factor authentication. A blue circular progress indicator labeled '1' is at the top left. The main area has a light blue header bar with a shield icon containing a keyhole. Below it, the section title 'Two-factor authentication' is displayed. A descriptive paragraph explains that 2FA is an extra layer of security used when logging into websites or apps, with a link to 'Learn more'. Two options are presented: 'Set up using an app' (selected, indicated by a blue dot) and 'Set up using SMS' (indicated by a white dot). Below each option is a brief description and a note about delivery limitations. At the bottom right of the wizard are 'Cancel' and 'Continue' buttons.

1

Two-factor authentication

Two-factor authentication (2FA) is an extra layer of security used when logging into websites or apps. [Learn more](#)

Set up using an app

Use an application on your phone to get two-factor authentication codes when prompted. We recommend using cloud-based TOTP apps such as: [1Password](#), [Authy](#), [LastPass Authenticator](#), or [Microsoft Authenticator](#).

Set up using SMS

GitHub will send you an SMS with a two-factor authentication code when prompted. SMS cannot be delivered in all countries. Check that [your country is supported](#) before you select this option.

[Cancel](#) [Continue](#)

13.4.3 NEUES REPOSITORY ERSTELLEN

Wählen Sie oben rechts das +-Zeichen und 'New Repository' Geben Sie Ihrem Repository einen Namen

The screenshot shows the GitHub interface with a context menu open. The 'New repository' option is highlighted. Below it, the 'Create a new repository' form is displayed. It includes fields for 'Owner' (set to 'jochen2022'), 'Repository name' (set to 'microblog'), and a 'Description' field which is empty. There are two radio button options for visibility: 'Public' (unchecked) and 'Private' (checked). Under 'Initialize this repository with:', there is a checkbox for 'Add a README file' which is unchecked. Below that, there's a section for '.gitignore' with a dropdown set to 'None'. A 'Choose a license' section shows 'License: None'. At the bottom, a note says '(1) You are creating a private repository in your personal account.' A green 'Create repository' button is at the very bottom.

eingeladene Benutzer können Ihr Repository sehen).

Klicken Sie auf 'Create Repository'. Nun sollten Sie die Folgende Seite sehen:

This screenshot shows the 'Quick setup' page for the newly created 'microblog' repository. It provides instructions for setting up the repository on the command line or via a web interface. The top part shows a URL for cloning the repository: `https://github.com/jochen2022/microblog.git`. Below that, there are three sections: 1) '...or create a new repository on the command line' with a code block:

```
echo "# microblog" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/jochen2022/microblog.git
git push -u origin main
```

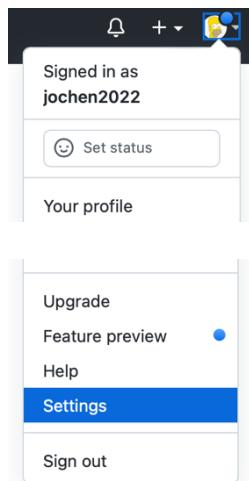
 2) '...or push an existing repository from the command line' with a code block:

```
git remote add origin https://github.com/jochen2022/microblog.git
git branch -M main
git push -u origin main
```

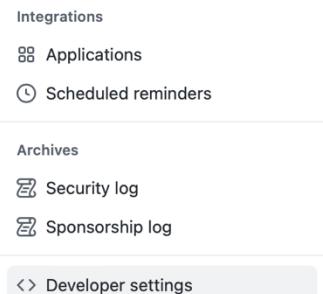
 3) '...or import code from another repository' with a note: 'You can initialize this repository with code from a Subversion, Mercurial, or TFS project.' and a 'Import code' button.

13.4.4 ACCESS-TOKEN ERSTELLEN

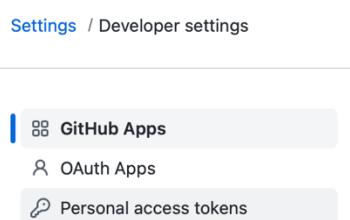
Für die nächsten Schritte brauchen Sie einen Access-Token von GitHub. Wählen Sie zunächst im Menu unter Ihrem Account- Symbol 'Settings'



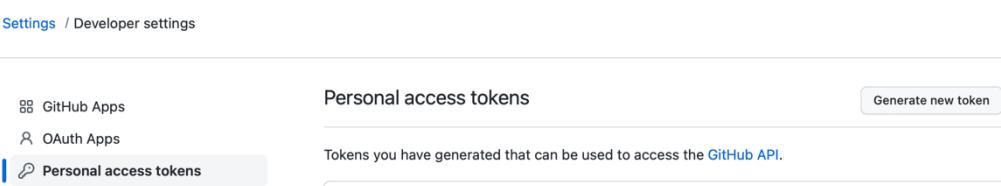
In der linken Seitenleiste unten finden Sie den Link 'Developer Settings':



In den Developer Settings wählen Sie 'Personal Access Tokens'



Klicken Sie auf den Button 'Generate New Token'



Den erzeugten Token kopieren Sie sich und bewahren ihn sicher auf.

13.4.5 REPOSITORY VORBEREITEN UND SYNCHRONISEREN

Aktivieren Sie im lokalen Projektverzeichnis Ihr virtuelles Environment und erzeugen Sie die Datei requirements.txt:

```
$ pip freeze > requirements.txt
```

Sie sollten nicht alle Inhalte Ihres Projektverzeichnisses auf Github synchronisieren, weil erstens einige Dateien und Verzeichnisse nur auf Ihrem Laptop einen Sinn ergeben und zweitens, weil der Inhalt einiger Dateien vertraulich ist.

Legen Sie daher eine Datei .gitignore in Ihrem lokalen Projektverzeichnis an. Dort wird alles aufgelistet, was nicht synchronisiert werden soll. Der Inhalt sollte so aussehen:

```
**/__pycache__
app.db
logs
.env
venv # Ihr Unterverzeichnis für das virtuelle Environment
```

Alle Unterverzeichnisse mit dem Namen __pycache__ enthalten kompilierten Code, der nur auf Ihrem Entwicklungssystem Sinn ergibt, die SQLite-Datenbank benutzen Sie nur zum Testen und die Logs sind auch nicht von Interesse. Wenn Sie eine .env Datei haben, stehen darin möglicherweise vertrauliche Informationen wie SECRET_KEY oder Ihr Mail-Passwort. Weitere Dateien, die vertrauliche Informationen beinhalten, können Sie ebenfalls hinzufügen.

Je nachdem, ob Sie Ihr lokales Projektverzeichnis schon als Git-Repository eingerichtet führen Sie dort verschiedene Kommandos aus:

1. Ihr Projektverzeichnis bereits ein Git-Repository:

```
$ git remote add origin https://github.com/<USER>/<REPOSITORY>.git
$ git branch -M main
$ git push https://<TOKEN>@github.com/<USER>/<REPOSITORY>.git
```

Ersetzen Sie <USER> mit Ihrem Usernamen auf GitHub, <REPOSITORY> mit dem Namen des neuen Repository und <TOKEN> mit Ihrem Access-Token aus dem letzten Schritt.

2. Ihr Projektverzeichnis ist noch nicht als Git-Repository eingerichtet

```
$ echo "# microblog" >> README.md
$ git init
$ git add *
$ git commit -m "first commit"
$ git branch -M main
$ git remote add origin https://github.com/<USER>/<REPOSITORY>.git
$ git push https://<TOKEN>@github.com/<USER>/<REPOSITORY>.git
```

Ersetzen Sie <USER> mit Ihrem Usernamen auf GitHub, <REPOSITORY> mit dem Namen des neuen Repository und <TOKEN> mit Ihrem Access-Token für GitHub.

Eine Erklärung dieser Schritte finden Sie in der öffentlichen Dokumentation von GitHub¹⁹.

¹⁹ <https://docs.github.com/en/get-started/getting-started-with-git/about-remote-repositories>

13.4.6 REPOSITORY AUF DEM SERVER CLONEN

Nun können Sie auf dem Server das GitHub-Repository clonen, anstatt Ihr Projektverzeichnis manuell hochzuladen. Melden Sie Sich auf dem Server an. Wenn Sie in Ihrem App-Verzeichnis (z.B. ~/microblog) Dateien wie .env haben, die Sie für die Konfiguration der App benötigen, sichern Sie diese an einem anderen Ort!

Löschen Sie das Verzeichnis Ihrer App unter Ihrem Home-Verzeichnis.

Anschliessend geben Sie im Home-Verzeichnis folgendes Kommando ein:

```
$ git clone https://github.com/<USER>/<REPOSITORY>.git
```

13.5 ZUSAMMENFASSUNG

14 EIN RESTFUL API

Alle Funktionalität, die Sie bisher in die App eingebaut haben, war auf eine einzige Art von Client ausgerichtet: Den Web-Browser. Es gibt aber viele verschiedene andere Arten von Clients, die einen Zugriff auf die Daten und Funktionen einer App benötigen könnten:

- Mobil-Apps auf dem Handy
- Eine andere Web-Applikation
- Eine auf einem PC installierte Applikation
- Eine Server-Applikation
- Ein Kommandozeilen-Programm

In diesem Kapitel wird ein Weg gezeigt, wie ein solcher Zugriff über HTTP über ein API (Application Programming Interface) aufgebaut werden kann.

14.1 LERNZIELE

Nach der Bearbeitung dieses Kapitels können Sie

- Die Prinzipien für ein RESTful API beschreiben.
- Datenstrukturen im Format JSON erzeugen
- Datenstrukturen im Format JSON lesen
- Eine Beschreibung eines API für Ihre App erstellen
- In Ihrer App API-Routen erstellen
- Fehler bei API-Aufrufen behandeln
- Das API gegen unbefugte Zugriffe schützen

14.2 REST

Die Abkürzung REST steht für Representational State Transfer. Der Begriff wurde durch Dr. Roy Fielding in seiner Dissertation geprägt. Fielding war am CERN an der Entwicklung des http-Protokolls beteiligt und hat darin das Grundprinzip der Kommunikation zwischen Client und Server im Web beschrieben.

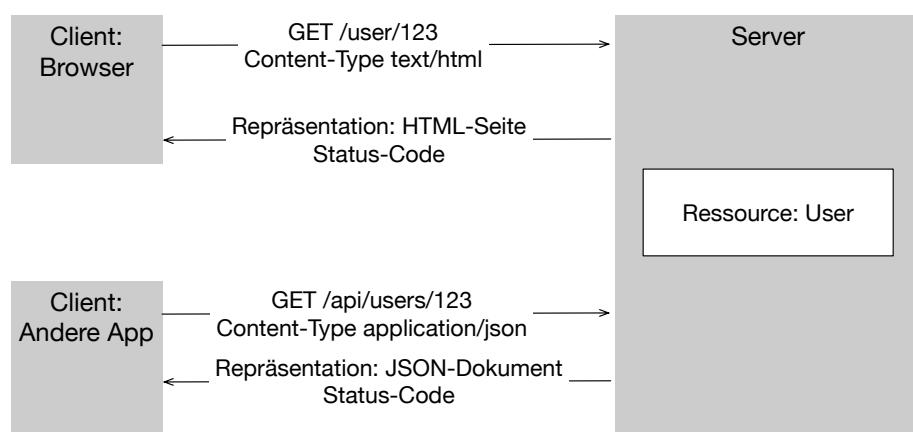


Abbildung 14-1 Server-Ressource User mit verschiedenen Repräsentationen

In der Abbildung sehen Sie die Begriffe *Ressource* und *Repräsentation*. Die Ressource ist etwas, was der Server anbietet. Bei REST kann eine Ressource alles Mögliche sein, z.B. Eine Information, die auf dem Server liegt oder eine Operation die der Server ausführen kann. Die Repräsentation ist nicht die Ressource selbst, sondern eine Darstellungsform der Ressource. Ein User könnte mit einer HTML-Seite dargestellt werden sein oder mit einer maschinenlesbaren Datenstruktur (In diesem Fall JSON-Dokument).

Eine Kommunikation entspricht dem REST-Konzept (Sie ist 'RESTful'), wenn sie sechs Prinzipien folgt, die hier vereinfacht aufgelistet werden:

1. Die Rollen Client und Server sind klar abgegrenzt. Sie sind separate Prozesse, die Nachrichten über einen beliebigen Transport-Mechanismus austauschen (in der Praxis: http-Protokoll über TCP). Ob die Prozesse auf verschiedenen Rechnern oder dem gleichen Rechner laufen, ist nicht von Bedeutung.
2. Für die Kommunikation spielt es keine Rolle, ob Client und Server Nachrichten direkt austauschen oder ob es zwischen ihnen weitere Stationen gibt, die die Nachricht weiterleiten oder vermitteln. Für den Server kommt die Anfrage immer vom Client, für den Client kommt eine Antwort immer vom Server.
3. Die Nachrichten können auf dem Weg zwischen Client und Server in Caches zwischengespeichert werden, um die Performance und die Verfügbarkeit zu erhöhen.
4. (Optional) Der Server darf ausführbaren Code an den Client senden. Es muss aber vereinbart sein, welche Art von Code der Client ausführen kann. Ein Webbrowser kann beispielsweise JavaScript ausführen.
5. Die Kommunikation ist zustandslos. Das bedeutet, dass auf Server keinerlei Informationen über vergangene Requests eines Clients gespeichert werden. In einem zustandslosen API muss jeder einzelne Request die nötigen Informationen enthalten, die der Server benötigt, um festzustellen, welche Inhalte der Client erhalten will oder welche Operation der Server für den Client ausführen soll. Bei jedem einzelnen Request muss sich der Client auch neu authentifizieren, damit der Server die Berechtigungen prüfen kann. Es gibt also keine längere Session, die über mehrere Requests/Responses hinweg besteht.
6. Die Kommunikation erfolgt über immer gleiche Schnittstelle ('Uniform Interface'), die allen Clients bekannt sein muss. Eine Schnittstelle ist 'uniform', wenn folgendes gegeben ist:
 - Jede Ressource verfügt über einen eindeutigen URI (Uniform Ressource Identifier). Im Fall von microblog wäre das beispielsweise eine URL für einen bestimmten User, wie /api/users/<id>.
 - Nicht die Ressource selbst (also das User-Objekt), sondern eine Repräsentation davon wird übermittelt. Die gleiche Ressource (z.B. ein User) könnte beispielsweise als HTML-, XML- oder JSON-Repräsentation übermittelt werden. Client und Server müssen sich über das erwartete und das gelieferte Format einigen. In Web-APIs erfolgt das über die sogenannte content negotiation des http-Protokolls. Im http-Header steht dann ein Content Type wie text/html oder application/json

- Nachrichten müssen selbstbeschreibend sein. Das bedeutet, dass eine Nachricht alle Informationen enthält, die ein Client oder Server benötigt, um die Nachricht zu verarbeiten. In einem Web-API enthält die http-Nachricht immer eine Methode wie GET (Client möchte Informationen zu einer Ressource abrufen), POST (Client möchte eine neue Ressource anlegen), PUT oder PATCH (Client möchte eine Ressource ändern) oder DELETE. Der Body der Nachricht enthält darüber hinaus meistens Daten in einem 'selbstbeschreibenden' Format (typischerweise JSON oder XML).
- Der Client kann über Hypermedia in den angebotenen Ressourcen navigieren. Wenn es Beziehungen zwischen den verschiedenen Ressourcen gibt, können diese über Links erreicht werden. Der Client hat typischerweise eine URL als Einstiegspunkt in das API. Die Response mit dem Einstiegspunkt enthält dann Links, mit denen der Client weitere verfügbare Ressourcen 'entdecken' kann. Ein Programm, dass das API benutzt kann diesen Links folgen, so wie ein Benutzer im Browser über Links auf andere Seiten wechseln kann.

14.3 DAS DATENFORMAT JSON

Das heute verbreitetste selbstbeschreibende Datenformat ist JSON (JavaScript Object Notation). Es ist maschinenlesbar wie XML, aber weitaus besser für Menschen lesbar.

Eine JSON-Repräsentation eines Users aus dem microblog-Modell könnte so aussehen:

```
{
  "id": 1,
  "username": "jochen"
  "email" : "jochen@example.com"
  "about_me": null,
  "post_count": 1,
  "followed_count": 1,
  "follower_count": 0,
  "last_seen": "2022-04-26T07:24:32.429951Z",
  "_links": {
    "avatar": "https://www.gravatar.com/avatar/ca76fe9azb?d=identicon&s=128",
    "followed": "/api/users/1/followed",
    "followers": "/api/users/1/followers",
    "self": "/api/users/1"
  }
}
```

Die Syntax des Beispiels ist einfach zu verstehen:

- Die Elemente in einem JSON-Objekt werden von geschweiften Klammern eingerahmt und geschweifte Klammern werden auch benutzt, um ein Objekt weiter zu untergliedern.
- Jedes Element besteht aus einem Schlüssel (z.B. "username") und einem Wert (z.B. "jochen"). Schlüssel und Wert sind durch einen Doppelpunkt (der *Separator*) getrennt.
- Aufeinanderfolgende Elemente sind durch Kommas getrennt. Auch nach dem letzten Element kann ein Komma gesetzt werden.
- Ein Wert kann wieder Schlüssel/Wert-Paare enthalten (Im Beispiel ist dies beim Schlüssel "_links" der Fall, das in geschweiften Klammern die Schlüssel "avatar", "followed", "followers" und "self" enthält, die als Werte verschiedene Links haben).

Mehrere Objekte werden als Liste dargestellt. Dafür werden Objekte in eckigen Klammern nacheinander aufgelistet. Das folgende Beispiel zeigt eine solche Liste mit zwei Posts , wie sie von /api/users/1/posts geliefert würde:

```
{
  "items": [
    {
      "id": 3,
      "author": "/api/users/1",
      "body": "Ich bearbeite das Flask Megatutorial von Miguel Grinberg\r\n",
      "timestamp": "Fri, 22 Apr 2022 07:17:10 GMT",
      "url": "http://localhost:5000/api/posts/3"
    },
    {
      "id": 4,
      "author": "/api/users/1",
      "body": "Mein zweiter Blogbeitrag",
      "timestamp": "Sat, 30 Apr 2022 16:16:25 GMT",
      "url": " http://localhost:5000/api/posts/4"
    }
  ]
}
```

Ihnen fällt sicher die Ähnlichkeit von JSON zu Python-Directories auf, und tatsächlich sind Python-Directories kompatibel zur JSON-Notation und lassen sich leicht in JSON kodieren.

14.4 DAS API

Die folgende Tabelle beschreibt, wie ein API für microblog aussehen könnte:

HTTP Methode	Ressourcen-URL (Endpunkt)	Beschreibung
GET	/api/users/<id>	Liefert einen bestimmten User
GET	/api/users/	Liefert alle User
GET	/api/users/<id>/followers	Liefert alle Follower eines bestimmten Users
GET	/api/users/<id>/followed	Liefert alle User, denen ein User folgt
GET	/api/users/<id>/posts	Liefert einen bestimmten User
POST	/api/users/<id>	Registrierung eines neuen User-Accounts
PUT	/api/users/<id>	Modifikation eines bestehenden User-Accounts
POST	/api/users/<id>/posts	Erzeugung eines neuen Posts
PUT	/api/users/<id>/posts/<id>	Modifikation eines Posts

In folgenden Abschnitt erstellen Sie zunächst zwei einfache Endpunkte für Ihr API, und zwar die ersten zwei aus der Tabelle.

14.5 ZWEI EINFACHE API-ENDPUNKTE

Als ersten, einfachen Schritt implementieren Sie die Endpunkte für

GET <http://localhost:5000/api/users/<id>> für die Daten eines Users und

GET <http://localhost/api/users/> für die Liste aller User-Account.

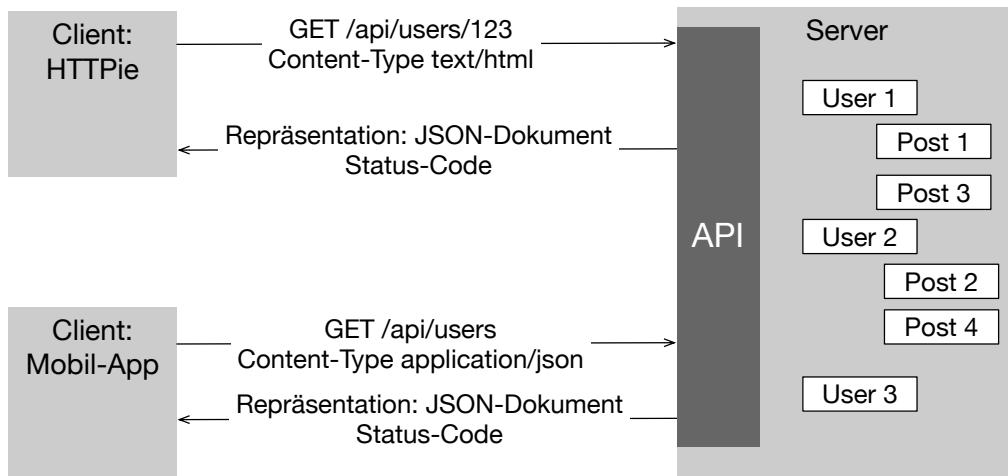


Abbildung 14-2 Aufruf zweier Endpunkte des API

14.5.1 ANPASSUNG DES MODEL

Ergänzen Sie zunächst die Klasse User in `app/models.py` wie folgt:

```
from flask import url_for

# ...
# In class User:
    def to_dict(self, include_email=False):
        data = {
            'id': self.id,
            'username': self.username,
            'last_seen': self.last_seen.isoformat() + 'Z',
            'about_me': self.about_me,
            'post_count': self.posts.count(),
            'follower_count': self.followers.count(),
            'followed_count': self.followed.count(),
            '_links': {
                'self': url_for('get_user', id=self.id),
                'followers': url_for('get_followers', id=self.id),
                'followed': url_for('get_followed', id=self.id),
                'avatar': self.avatar(128)
            }
        }
        if include_email:
            data['email'] = self.email
        return data
```

- Die Funktion `to_dict()` erzeugt ein Dictionary `data` mit den Werten des aktuellen Users, der Anzahl Posts, der Anzahl Posts, der Anzahl und einigen weiterführenden Links. Die Funktion gibt am Ende dieses Dictionary an den Aufrufer zurück. Ob auch die E-Mail-Adresse geliefert werden soll, kann über den Parameter `include_email` geteuert werden

- Das Element mit dem Schlüssel '_links' enthält als Wert wieder ein Dictionary. Es enthält die Links zu sich selbst, den Followern, den Usern, denen dieser User folgt und zum Avatar-Bild.
- Die Links werden mit der Funktion url_for() erzeugt, die aus der internen url_map die passenden internen Routen für get_user(), get_followers() und get_followed() heraussucht.

Die zweite Methode, to_collection() fragt alle User aus der Datenbank ab und benutzt eine List Comprehension, um jeden gefundenen Datensatz mit der Methode to_dict() in eine Liste mit dem Namen data einzutragen.

```
# In class User:
@staticmethod
def to_collection():
    users = User.query.all()
    data = {'items': [item.to_dict() for item in users]}
    return(data)
```

14.5.2 API-ROUTEN

Erzeugen Sie nun eine neue Datei app/api.py. Tragen Sie dort zunächst folgendes ein

```
# app/api.py
from app import app
from app.models import User, Post
from flask import jsonify
```

Flask stellt eine Funktion jsonify() zur Verfügung, der Sie ein Dictionary (oder eine Reihe von Schlüsselwort-Argumenten in der Art key=value) übergeben können. Daraus erzeugt jsonify() ein http Response-Objekt mit dem Content-Type application/json.

Tragen Sie die folgenden Funktionen in app/api.py ein:

```
@app.route('/api/users/<int:id>', methods=['GET'])
def get_user(id):
    data = User.query.get_or_404(id).to_dict()
    return jsonify(data)

@app.route('/api/users', methods=['GET'])
def get_users():
    data = User.to_collection()
    return jsonify(data)

@app.route('/api/users/<int:id>/followers', methods=['GET'])
def get_followers(id):
    pass

@app.route('/api/users/<int:id>/followed', methods=['GET'])
def get_followed(id):
    pass

@app.route('/api/users/<int:id>/posts', methods=['GET'])
def get_posts(id):
    pass
```

Den Funktionen wird wie bei den View-Funktionen in app/routes der Decorator @app.route vorangestellt. Dadurch werden Sie als Endpunkt für die jeweilige Route in die interne url_map

einetragen und somit ausgeführt, wenn der Server einen GET-Request mit einer URL erhält, die mit dieser Route endet. Es wird aber kein gerendertes HTML an den Server zurückgegeben, sondern das Response-Objekt, das von jsonify() erzeugt wurde.

Die letzten drei Funktionen wurden nur als Platzhalter mit pass eingefügt. Sie müssen für die ersten Tests schon vorhanden sein und werden später noch vervollständigt.

Nun müssen Sie noch das Modul api.py in app/__init__.py importieren. Ergänzen Sie dort die letzte Zeile:

```
# Am Ende von app/__init__.py
from app import routes, models, errors, api
```

Die Funktion url_for() funktioniert nur, wenn es einen Client-Request an die App gibt. Damit Sie die Methoden in models.py in der Flask Shell auch ohne Request testen können, muss die Config-Variable Tragen Sie daher folgendes provisorisch in config.py ein:

```
class Config(object):
    # Provisorisch
    SERVER_NAME = 'localhost:5000'
    # SERVER_NAME kann später wieder entfernt werden
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'you-will-never-guess'
```

14.5.3 TEST DER METHODEN

Stellen Sie sicher, das Sie mindestens zwei User-Accounts haben, wovon der eine dem anderen folgt.

Rufen Sie anschliessend die Flask Shell auf:

```
(venv) jochen@host microblog % flask shell
>>> users = User.query.all()
>>> users
[<User jochen>, <User paula>]
>>> user1=users[0]
>>> user2=users[1]
>>>
```

Nun haben Sie eine Liste users und zwei User-Objekte. Rufen Sie nun die beiden neuen Methoden von User auf:

```
>>> user1_data = user1.to_dict()
>>> user2_data = user2.to_dict()
>>> all_user_data = User.to_collection()
>>>
```

Um Dictionaries 'schön' darzustellen, können Sie die Funktion pprint aus der Standardbibliothek importieren und anschliessend benutzen:

```
>>> from pprint import pprint
>>> pprint(user1_data)
{'_links': {'avatar': 'https://www.gravatar.com/avatar/...', 'followed': 'http://localhost:5000/api/users/1/followed', 'followers': 'http://localhost:5000/api/users/1/followers', 'self': 'http://localhost:5000/api/users/1'}, 'about_me': None, 'followed_count': 1, 'follower_count': 0, 'id': 1, 'last_seen': '2022-04-30T16:16:25.354258Z', 'post_count': 2, 'username': 'jochen'}

>>> pprint(all_user_data)
{'items': [
    {'_links': {'avatar': 'https://www.gravatar.com/avatar/...', 'followed': 'http://localhost:5000/api/users/1/followed', 'followers': 'http://localhost:5000/api/users/1/followers', 'self': 'http://localhost:5000/api/users/1'}, 'about_me': None, 'followed_count': 1, 'follower_count': 0, 'id': 1, 'last_seen': '2022-04-30T16:16:25.354258Z', 'post_count': 2, 'username': 'jochen'},
    {'_links': {'avatar': 'https://www.gravatar.com/avatar/...', 'followed': 'http://localhost:5000/api/users/2/followed', 'followers': 'http://localhost:5000/api/users/2/followers', 'self': 'http://localhost:5000/api/users/2'}, 'about_me': None, 'followed_count': 0, 'follower_count': 1, 'id': 2, 'last_seen': '2022-04-21T13:59:14.595669Z', 'post_count': 2, 'username': 'paula'}]}
```

(Die Ausgabe von pprint(all_user_data) wurde aus Platzgründen leicht angepasst)

14.5.4 TEST-CLIENT

Als Client für den Test API-Aufrufe benutzen Sie HTTPie. Installieren Sie es gemäss der Beschreibung auf <https://httpie.io/docs/cli>, als Benutzer mit Administrator-Rechten:

```
$ python -m pip install --upgrade pip wheel
$ python -m pip install httpie
```

Danach steht Ihnen das Kommando 'http' auf der Kommandozeile zur Verfügung.

Nachdem Sie Ihre App mit flask run gestartet haben, testen Sie den API-Zugriff wie folgt:

```
$ http --json GET http://localhost:5000/api/users/1 accept:application/json
HTTP/1.0 200 OK
Content-Length: 409
Content-Type: application/json
Date: Sun, 01 May 2022 07:49:18 GMT
Server: Werkzeug/2.0.3 Python/3.9.12

{
    "_links": {
        "avatar": "https://www.gravatar.com/avatar/...",
        "followed": "/api/users/1/followed",
        "followers": "/api/users/1/followers",
        "self": "/api/users/1"
    },
    "about_me": null,
    "followed_count": 1,
    "follower_count": 0,
    "id": 1,
    "last_seen": "2022-04-30T16:16:25.354258Z",
    "post_count": 2,
    "username": "jochen"
}
```

Testen Sie auch die zweite API-Route wie folgt:

```
$ http --json GET http://localhost:5000/api/users accept:application/json
```

Die Ausgabe sollte eine Liste aller User-Objekte im JSON-Format sein. Die Angabe accept:application/json setzt das Feld Accept im HTTP-Header. API-Clients geben damit an, in welchem Format sie die Antwort erwarten.

14.6 WEITERE ENDPUNKTE FÜR GET

Wenn Sie die beiden einfachen Fälle erfolgreich implementiert und getestet haben, können Sie die restlichen Endpunkte für GET komplettieren. Die dafür vorgesehenen Route-Funktionen in app/api.py haben Sie bereits eingetragen, allerdings nur provisorisch mit pass:

/api/users/<id>/followers - Alle Follower zu einem User

In app/models.py fügen Sie der Klasse User die Methode followers_to_collection() hinzu:

```
def followers_to_collection(self):
    data = {'items': [item.to_dict() for item in self.followers]}
    return data
```

Ändern Sie in app/api.py die Funktion get_followers() wie folgt:

```
@app.route('/api/users/<int:id>/followers', methods=['GET'])
def get_followers(id):
    user = User.query.get_or_404(id)
    data = user.followers_to_collection()
    return jsonify(data)
```

Testen Sie das mit einem User, der mehrere Follower hat:

```
$ http http://localhost:5000/api/users/2/followers accept:application/json
```

/api/users/<id>/followed - Alle User, denen ein User folgt

In app/models.py fügen Sie der Klasse User die Methode followed_to_collection() hinzu:

```
def followed_to_collection(self):
    data = {'items': [item.to_dict() for item in self.followed]}
    return data
```

Ändern Sie in app/api.py die Funktion get_followed() wie folgt:

```
@app.route('/api/users/<int:id>/followed', methods=['GET'])
def get_followed(id):
    user = User.query.get_or_404(id)
    data = user.followed_to_collection()
    return jsonify(data)
```

Testen Sie das mit einem User, der mehreren anderen Usern folgt:

```
$ http http://localhost:5000/api/users/2/followed accept:application/json
```

/api/users/<id>/posts - Alle Posts eines Users

In app/models.py fügen Sie der Klasse Posts folgende Methoden hinzu:

```
def to_dict(self):
    data = {
        'id': self.id,
        'url': url_for('get_posts', id=self.id, _external=True),
        'body': self.body,
        'timestamp': self.timestamp,
        'author': url_for('get_user', id=self.user_id, _external=True)
    }
    return data
```

```
@staticmethod  
def to_collection():  
    resources = Post.query.all()  
    data = {'items': [item.to_dict() for item in resources]}  
    return(data)
```

In app/models.py fügen Sie der Klasse User die Methode posts_to_collection() hinzu:

```
def posts_to_collection(self):  
    data = {'items': [item.to_dict() for item in self.posts]}  
    return data
```

Ändern Sie in app/api.py die Funktion get_posts () wie folgt:

```
@app.route('/api/users/<int:id>/posts', methods=['GET'])  
def get_posts(id):  
    user = User.query.get_or_404(id)  
    data = user.posts_to_collection()  
    return jsonify(data)
```

Testen Sie das mit einem User, der mindestens 2 Posts hat:

```
$ http http://localhost:5000/api/users/2/posts accept:application/json
```

14.7 FEHLERBEHANDLUNG

Im Falle eines Zugriffsfehlers auf eine Website mit dem Browser wird in der Response des Servers ein Status-Code aus dem Bereich 400 bis 499 und eine HTML-Fehlerseite zurückgesendet. Mit der HTML-Seite kann ein API-Client aber nichts anfangen, er erwartet zusätzlich zum Status-Code eine Fehlermeldung mit dem Content-Type application/json.

So wie das API bis jetzt implementiert wurde, werden alle Fehler mit dem Decorator @app.errorhandler behandelt. Dort müsste geprüft werden, was der Content-Type des Requests ist und die passende Übermittlungs-Methode für die Fehlermeldung ausgewählt werden.

Die folgende Liste enthält eine Auswahl von Status-Codes, die für ein API wichtig sind:

Code	Name	Bedeutung
200	OK	Der Request wurde erfolgreich ausgeführt
201	Created	Eine neue Ressource wurde angelegt
202	Accepted	Der Request wurde akzeptiert, ist aber noch in (asynchroner) Bearbeitung
204	No Content	Der Request wurde erfolgreich ausgeführt, aber es gibt keinen Inhalt für die Response
400	Bad Request	Der Request ist ungültig
401	Unauthorized	Der Client konnte nicht authentifiziert werden
403	Forbidden	Die Berechtigungen für den Client reichen für den Request nicht aus
404	Not Found	Die angefragte Ressource wurde nicht gefunden
405	Method Not Allowed	Die Methode im Request wird für die Ressource nicht unterstützt
500	Internal Server Error	Bei der Bearbeitung des Requests ist ein unerwarteter Fehler aufgetreten

Die Fehlerbehandlung für die Status-Codes 404 und 500 haben Sie im Kapitel 8.4 implementiert. Dort gibt es schon zwei Funktionen:

```
@app.errorhandler(404)
def not_found_error(error):
    return render_template('404.html'), 404

@app.errorhandler(500)
def internal_error(error):
    db.session.rollback()
    return render_template('500.html'), 500
```

Sie rufen je eine Fehler-Seite auf. Ein API-Client erwartet aber bei Fehlern keine HTML-Seite, sondern ein JSON-Objekt.

Im Request-Objekt von Flask finden Sie alle Informationen zum aktuellen http-Request.

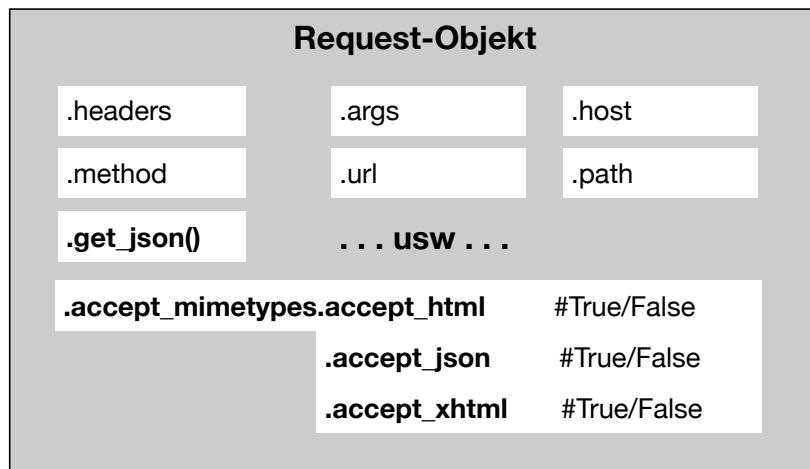


Abbildung 14-3 Attribute des Request-Objekts

Wichtig für das API sind die Methode `get_json()` und `request.accept_mimetypes`. Die vom Client akzeptierten Content-Typen sind dort in vermerkt. Von Interesse sind zunächst:

- `request.accept_mimetypes.accept_html` (True, wenn HTML akzeptiert wird)
- `request.accept_mimetypes.accept_json` (True, wenn JSON akzeptiert wird)

In Errorhandler-Funktionen können diese Attribute abgefragt werden, so dass zwischen einem Browser- und einem API-Request unterschieden werden kann. Falls der Client-Request im Header bei `Accept: application/json` enthält, wird statt der HTML-Seite ein JSON-Objekt zurückgegeben.

Ändern Sie `app/errors.py` wie folgt:

```
# app/errors.py
# Zusätzliche Importe: request-Objekt und jsonify()
from flask import render_template, request, jsonify
from app import app, db

@app.errorhandler(404)
def not_found_error(error):
    if request.accept_mimetypes.accept_json and \
       not request.accept_mimetypes.accept_html:
        response = jsonify( {'error': 'Not Found'} )
        response.status_code = 404
        return response
    else:
        return render_template('404.html'), 404

@app.errorhandler(500)
def internal_error(error):
    db.session.rollback()
    if request.accept_mimetypes.accept_json and \
       not request.accept_mimetypes.accept_html:
        response = jsonify( {'error': 'Internal Server Error'} )
        response.status_code = 500
        return response
    return render_template('500.html'), 500
```

Testen Sie den Fall des 404-Fehlers, indem Sie

1. Im Browser Die URL <http://localhost:5000/gibtsnicht> eingeben. Dann sollte weiterhin die eigene Fehlerseite angezeigt werden.
2. Auf der Kommandozeile HTTPie explizit mit Accept:'application/json' aufrufen. Die Response sollte jetzt so aussehen:

```
$ http http://localhost:5000/api/users/99999 accept:application/json
HTTP/1.0 404 NOT FOUND
Content-Length: 27
Content-Type: application/json
Date: Sun, 01 May 2022 14:53:54 GMT
Server: Werkzeug/2.0.3 Python/3.9.12

{
    "error": "Not Found"
}
```

Für viele Fehlersituationen brauchen Sie die Fälle HTML/JSON nicht unterscheiden, weil sie nur bei API-Requests vorkommen. Nehmen Sie die folgende Funktionen in app/errors.py auf, Sie werden sie später brauchen:

```
from werkzeug.http import HTTP_STATUS_CODES

def error_response(status_code, message=None):
    payload = {'error': HTTP_STATUS_CODES.get(status_code, 'Unknown error')}
    if message:
        payload['message'] = message
    response = jsonify(payload)
    response.status_code = status_code
    return response

def bad_request(message):
    return error_response(400, message)

# Beispiel: bad_request('Diese Email-Adresse ist schon vergeben')
```

Die Funktion error_response() erzeugt für beliebige Fehler eine Response. Ihr können Sie einen Status-Code und eine Fehlermeldung übergeben.

- Das Dictionary HTTP_STATUS_CODES wird aus der Flask-Komponente Werkzeug importiert. Es enthält als Schlüssel alle definierten Status-Codes und als Werte die Namen dazu.
- Der Funktion können Sie optional noch den String message übergeben.

Ein damit erzeugtes Response-Objekt hat dann einen Status-Code, den Namen dazu und Ihre Fehlermeldungs-message.

14.8 API-ENDPUNKTE MIT POST UND PUT

Die App soll nun noch ermöglichen, einen neuen User über das API zu registrieren und einen bestehenden User zu ändern. Dafür benötigen Sie eine zusätzliche Methode in der User-Klasse (In app/models.py):

```
def from_dict(self, data, new_user=False):
    for field in ['username', 'email', 'about_me']:
        if field in data:
            setattr(self, field, data[field])
    if new_user and 'password' in data:
        self.set_password(data['password'])
```

Die Methode from_dict() erwartet als Argument ein Dictionary data mit Angaben zum User und die Angabe new_user, über die unterschieden werden kann, ob ein neuer bestehender User geändert oder ob ein User angelegt werden soll.

- Im ersten Fall werden die Daten des bestehenden Users mit *setattr(Objekt, attributname, wert)* geändert. Die Funktion stammt aus der Standardbibliothek.
- Im zweiten Fall (new_user == True and password in data) wird auch das Passwort gesetzt. Dabei wird die bereits existierende Methode User.set_password() benutzt.

Endpunkt /api/users mit POST - Registrierung eines neuen Users

Nun können Sie in app/api.py eine Funktion einfügen, mit der ein neuer User angelegt werden kann:

```
# app/api.py – Neue Importe hervorgehoben
from app import app, db
from app.errors import bad_request
from app.models import User, Post
from flask import jsonify, request, url_for

# Endpunkt für User-Registrierung
@app.route('/api/users', methods=['POST'])
def create_user():
    data = request.get_json() or {}
    if 'username' not in data or 'email' not in data or 'password' not in data:
        return bad_request('must include username, email and password fields')
    if User.query.filter_by(username=data['username']).first():
        return bad_request('please use a different username')
    if User.query.filter_by(email=data['email']).first():
        return bad_request('please use a different email address')
    user = User()
    user.from_dict(data, new_user=True)
    db.session.add(user)
    db.session.commit()
    response = jsonify(user.to_dict())
    response.status_code = 201
    response.headers['Location'] = url_for('get_user', id=user.id)
    return response
```

- Mit der Methode get_json() aus dem Request-Objekt wird der Inhalt der Nachricht als JSON kodiert und der Variablen data zugewiesen.

- Anschliessend wird geprüft, ob die Pflichtfelder username, email und password in data Werte haben. Falls nicht wird die Funktion bad_request() aus app/errors.py aufgerufen, um dem Client eine Fehlermeldung zu schicken.
- Usernamen und Email-Adressen müssen in microblog eindeutig sein. Falls das nicht der Fall ist, werden ebenfalls Fehlernachrichten ausgelöst.
- Wenn alle benötigten Angaben korrekt sind, wird ein neues User-Objekt erzeugt und in der Datenbank gespeichert.
- Nun muss dem Client noch eine Antwort geschickt werden. Hierzu wird mit jsonify() ein Response-Objekt mit den User-Daten erzeugt. Das Response-Objekt hat diverse Attribute, unter anderem den Status und das Dictionary headers:

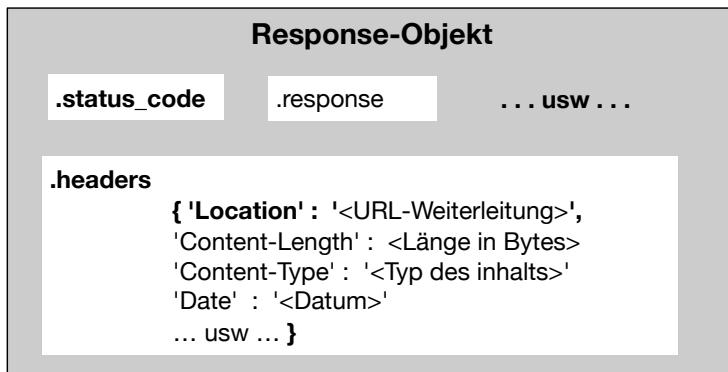


Abbildung 14-4 Attribute des Response-Objekts

- Der Status-Code wird auf 201 (Created) gesetzt.
- Mit response.headers['Location'] = url_for('get_user', id=user.id) wird eine Weiterleitung gesetzt, die dem API-Client die Information zum neuen User gibt.
- Die Response wird mit return an den Server übergeben, der die Nachricht an den Client ausliefert.

Tests des Endpunkts

a) Korrekter Fall:

```

$ http POST http://localhost:5000/api/users username=anna \
  password=geheim \
  email=anna@example.com \
  about_me="Hallo, ich bins"

HTTP/1.0 201 CREATED
Content-Length: 420
Content-Type: application/json
Date: Tue, 03 May 2022 09:33:30 GMT
Location: http://localhost:5000/api/users/5
Server: Werkzeug/2.0.3 Python/3.9.12
  
```

```
{
  "_links": {
    "avatar": "https://www.gravatar.com/avatar/6b56d06?d=identicon&s=128",
    "followed": "/api/users/5/followed",
    "followers": "/api/users/5/followers",
    "self": "/api/users/5"
  },
  "about_me": "Hallo, ich bins",
  "followed_count": 0,
  "follower_count": 0,
  "id": 5,
  "last_seen": "2022-05-03T09:33:30.314553Z",
  "post_count": 0,
  "username": "anna"
}
```

- b) Unvollständige Angaben

```
$ http POST http://localhost:5000/api/users password=geheim \
  email=peter@example.com about_me="Ich bin Peter"
```

```
HTTP/1.0 400 BAD REQUEST
Content-Length: 95
Content-Type: application/json
Date: Tue, 03 May 2022 09:41:46 GMT
Server: Werkzeug/2.0.3 Python/3.9.12
```

```
{
  "error": "Bad Request",
  "message": "must include username, email and password fields"
}
```

- c) Username existiert schon

```
$ http POST http://localhost:5000/api/users username=jochen password=geheim \
  email=xyz@example.com about_me="Ich bin der neue Jochen"
```

```
HTTP/1.0 400 BAD REQUEST
Content-Length: 78
Content-Type: application/json
Date: Tue, 03 May 2022 09:43:47 GMT
Server: Werkzeug/2.0.3 Python/3.9.12
```

```
{
  "error": "Bad Request",
  "message": "please use a different username"
}
```

- d) Email existiert schon

```
$ http POST http://localhost:5000/api/users username=petra password=geheim \
  email=xyz@example.com
```

```
HTTP/1.0 400 BAD REQUEST
Content-Length: 83
Content-Type: application/json
Date: Tue, 03 May 2022 09:46:30 GMT
Server: Werkzeug/2.0.3 Python/3.9.12
```

```
{
  "error": "Bad Request",
  "message": "please use a different email address"
}
```

Endpunkt /api/users mit PUT - Änderung von Userdaten

In app/api.py richten Sie diese neue Funktion ein:

```
@app.route('/api/users/<int:id>', methods=['PUT'])
def update_user(id):
    user = User.query.get_or_404(id)
    data = request.get_json() or {}
    if 'username' in data and data['username'] != user.username and \
        User.query.filter_by(username=data['username']).first():
        return bad_request('please use a different username')
    if 'email' in data and data['email'] != user.email and \
        User.query.filter_by(email=data['email']).first():
        return bad_request('please use a different email address')
    user.from_dict(data, new_user=False)
    db.session.commit()
    return jsonify(user.to_dict())
```

- Zunächst wird der gewünschte User aus der Datenbank geladen. Falls kein User mit der angegebenen id existiert, wird das ganze mit einem Status 404 (Not Found) abgebrochen.
- Wieder wird mit request.get_json() der Inhalt der Nachricht abgerufen und mit der Variablen data referenziert. Anschliessend wird geprüft, ob der Username geändert wurde und wenn ja, ob schon ein solcher Username in der Datenbank ist. In diesem Fall wird bad_request() eine Fehlernachricht geschickt. Für die E-Mail-Adresse gilt das gleiche.
- Wenn alles in Ordnung ist, wird user.from_dict(data, new_user=False) aus models.py benutzt, um die Änderungen im user-Objekt zu setzen. Mit db.session.commit() wird der geänderte User gespeichert
- Die Response bei PUT hat keinen speziellen Status-Code, wenn nichts anderes gesetzt ist, ist er 200 (OK). Auch hier wird eine Repräsentation der angepassten Ressource geschickt.

Tests des Endpunkts

a) Korrekter Fall:

```
$ http PUT http://localhost:5000/api/users/6 username=joerg123 \
accept:application/json
HTTP/1.0 200 OK
Content-Length: 411
Content-Type: application/json
Date: Tue, 03 May 2022 10:13:07 GMT
Server: Werkzeug/2.0.3 Python/3.9.12

{
    "_links": {
        "avatar": "https://www.gravatar.com/avatar/65ab25?d=identicon&s=128",
        "followed": "/api/users/6/followed",
        "followers": "/api/users/6/followers",
        "self": "/api/users/6"
    },
    "about_me": null,
    "followed_count": 0,
    "follower_count": 0,
    "id": 6,
    "last_seen": "2022-05-03T09:45:52.972417Z",
    "post_count": 0,
    "username": "joerg123"
}
```

- b) User existiert nicht

```
$ http --json PUT http://localhost:5000/api/users/99 \
    username=joerg123 accept:application/json
HTTP/1.0 404 NOT FOUND
Content-Length: 22
Content-Type: application/json
Date: Tue, 03 May 2022 12:58:16 GMT
Server: Werkzeug/2.0.3 Python/3.9.12

{
    "error": "Not Found"
}
```

- c) Neuer Username existiert schon bei einem anderen User

```
$ http PUT http://localhost:5000/api/users/6 username=jochen \
    accept:application/json
HTTP/1.0 400 BAD REQUEST
Content-Length: 68
Content-Type: application/json
Date: Tue, 03 May 2022 13:00:04 GMT
Server: Werkzeug/2.0.3 Python/3.9.12

{
    "error": "Bad Request",
    "message": "please use a different username"
}
```

- d) Neue Email existiert schon bei einem anderen User

```
$ http PUT http://localhost:5000/api/users/6 email=jochen@example.com \
    accept:application/json
HTTP/1.0 400 BAD REQUEST
Content-Length: 73
Content-Type: application/json
Date: Tue, 03 May 2022 14:20:40 GMT
Server: Werkzeug/2.0.3 Python/3.9.12

{
    "error": "Bad Request",
    "message": "please use a different email address"
}
```

14.9 AUTHENTIFIZIERUNG

Die API-Endpunkte sind noch für jedermann ohne weiteres zugänglich. Als letzter Schritt sollen sie nun gegen unberechtigte Nutzung geschützt werden.

- Ein API-Client kann mit Username und Passwort einen Token anfordern
- Das API übermittelt dem Client einen Token, der für eine begrenzte Zeit gültig ist
- Jede weitere Anfrage muss sich der Client mit einem Token beim API authentifizieren

14.9.1 ÄNDERUNGEN AM USER-MODEL

Fügen Sie diese Importe in app/models.py hinzu:

```
import base64
from datetime import timedelta
import os
```

Fügen Sie der Klasse User in models.py zwei zusätzliche Instanz-Attribute hinzu:

```
# In class User:
# ...
    # Der aktuelle API-Token in der Datenbank
    token = db.Column(db.String(32), index=True, unique=True)
    # Das Ablaufdatum des Token in der Datenbank
    token_expiration = db.Column(db.DateTime)
```

Fügen Sie der Klasse User neue Methoden hinzu:

```
# Token erzeugen, speichern und zurückgeben
def get_token(self, expires_in=3600):
    now = datetime.utcnow()
    if self.token and self.token_expiration > now + timedelta(seconds=60):
        return self.token
    self.token = base64.b64encode(os.urandom(24)).decode('utf-8')
    self.token_expiration = now + timedelta(seconds=expires_in)
    db.session.add(self)
    return self.token

# Token ungültig machen
def revoke_token(self):
    # Ablaufdatum auf aktuelle Zeit - 1 sek. setzen
    self.token_expiration = datetime.utcnow() - timedelta(seconds=1)

# Token prüfen
@staticmethod
def check_token(token):
    user = User.query.filter_by(token=token).first()
    if user is None or user.token_expiration < datetime.utcnow():
        return None # Token nicht gefunden oder abgelaufen
    return user # Token ist gültig
```

Die Änderung am User-Model muss noch in das DB-Schema eingebaut werden:

```
(venv) $ flask db migrate -m "user tokens"
(venv) $ flask db upgrade
```

14.9.2 TOKEN ANFORDERN

Für die Anforderung eines Token muss sich der API-Client einmalig mit Usernamen und Passwort anmelden. Das wird mithilfe der Erweiterung Flask-HTTPAuth ermöglicht:

```
(venv) $ pip install flask-httpauth
```

Flask-HTTPAuth stellt verschiedene Arten der Authentisierung zur Verfügung. Zunächst verwenden Sie die Methode http Basic Authentication, bei der der Client Usernamen und Passwort in einem Authorization-Header übermittelt.

Um Flask-HTTPAuth zu nutzen, muss die App zwei Funktionen definieren:

- Eine Funktion, die Passwort und Usernamen überprüft
- Eine Funktion, die im Fall einer erfolglosen Anmeldung eine Fehler-Response zurückgibt.

Diese Funktionen werden Flask-HTTPAuth über Dekorierer bekannt gemacht. Sie werden dann im Laufe der Anmeldung aufgerufen.

15 ANHANG: WEITERE FELDTYPEN IN FORMULAREN

15.1 EINFACHES AUSWAHLFELD

Ein SelectField von WTForms baut einen HTML <select> Tag mit einer Werteauswahl in das Formular ein.

Ein einfaches Select-Field mit festen Werten braucht eine Liste mit Auswahloptionen. Jedes Element der Liste besteht aus einem String-Wert und einer Beschriftung für den Wert. z.B:

Wert	Beschriftung
'1'	'Sonntag'
'2'	'Montag'
'3'	'Dienstag'
'4'	'Mittwoch'
...	...



Form in app/forms.py:

```
from wtforms import SelectField

class SelectDemoForm(FlaskForm):

    tage = SelectField(label = 'Wochentag', validators=[DataRequired()],
                       choices = [('1', 'Sonntag'), ('2', 'Montag'),
                                  ('3', 'Dienstag'), ('4', 'Mittwoch'), ('5', 'Donnerstag'),
                                  ('6', 'Freitag'), ('7', 'Samstag')])
    submit = SubmitField('Submit')
```

Template app/templates/select_demo.html:

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

{% block content %}
    <h1>Auswahlfelder</h1>
    <form action="" method="post">
        {{ form.hidden_tag() }}
        <p>
            {{ form.tage.label }}<br>
            {{ form.tage }}<br>
            {% for error in form.tage.errors %}
                <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

View-Funktion in app/routes.py:

```
from app.forms import SelectDemoForm

@app.route('/select_demo', methods=['GET', 'POST'])
def select_demo():
    form = SelectDemoForm()
    if form.validate_on_submit() == True :
        nummer_tag = int(form.tage.data)
        return '<html><h3>Auswahl: Tag Nr. {}</h3></html>'.format(nummer_tag)
    else:
        return render_template('select_demo.html', title='Select', form=form)
```

Der Wert in feld.data ist bei SelectField immer vom Typ str. Im Beispiel wird er noch in int umgewandelt, falls das benötigt wird.

15.2 AUSWAHLFELD MIT DATENBANK-QUERY

Statt einer Festen Liste von Werten soll eine dynamische Auswahl auf Basis einer Tabelle präsentiert werden. Dafür kann der Feldtyp QuerySelectField aus dem Paket WTForms-SQLAlchemy verwendet werden²⁰²¹ (<https://pypi.org/project/WTForms-SQLAlchemy/>):

```
(tutorial) $ pip install WTForms-SQLAlchemy
```

Ein QuerySelectField generiert eine Auswahlliste auf Basis einer SQLAlchemy-Query, deren Ergebnis als Liste von Model-Objekten vorliegt. Im folgenden Beispiel wird dies mit Einträgen aus der Tabelle user demonstriert.

Formular in app/forms.py:

```
from wtforms_sqlalchemy.fields import QuerySelectField

class QuerySelectDemoForm(FlaskForm):
    users = QuerySelectField(allow_blank = True, get_label = 'username',
                           validators = [DataRequired()])
    submit = SubmitField('Submit')
```

- Mit dem Argument allow_blank kann gesteuert werden, ob im Auswahlfeld ein leerer Eintrag an erster Stelle steht. Würde hier allow_blank = False stehen, wäre der erste Datensatz das erste User-Objekt.
- Mit get_label wird angegeben, welches Attribut von User als Beschriftung angezeigt wird.
- Die eigentliche Query wird im Beispiel nicht in app/forms.py angegeben, sondern in app/routes.py (Siehe unten)

²⁰ <https://pypi.org/project/WTForms-SQLAlchemy/>

²¹ https://wtforms-sqlalchemy.readthedocs.io/en/latest/wtforms_sqlalchemy/

- WTForms-SQLAlchemy entdeckt den Primärschlüssel in einfachen Fällen automatisch. Ansonsten kann er mit get_pk=primärschlüsselspalte bei der Erzeugung des Feldes angegeben werden.

Template app/templates/query_select_demo.html:

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

{% block content %}
    <h1>Auswahlfelder</h1>
    <form action="" method="post">
        {{ form.hidden_tag() }}
        <p>
            {{ form.users.label }}<br>
            {{ form.users }}<br>
            {% for error in form.users.errors %}
                <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

View-Funktion in app/routes.py:

```
from app.forms import QuerySelectDemoForm

@app.route('/query_select_demo', methods=['GET', 'POST'])
def query_select_demo():
    form = QuerySelectDemoForm()
    form.users.query = db.session.query(User)
    if form.validate_on_submit() == True :
        user_id = int(form.users.data.id)
        return '<html><h3>Auswahl: User {}</h3></html>'.format(user_id)
    else:
        return render_template('query_select_demo.html', title='Select DB',
                               form=form)
```

- Die Query db.session.query(User) liefert alle User-Einträge als Liste von User-Objekten zurück. Sie wird in diesem Beispiel in der View-Funktion erstellt und das Ergebnis an das Feld form.users.query zugewiesen.
- Ein QuerySelectField funktioniert mit Tabellen, deren Primärschlüssel einen Datentyp hat, der problemlos in einen String umgewandelt werden kann, da in einem HTML <select> Element nur Strings möglich sind. Bei der id in User ist das der Fall, int kann von Python automatisch in str umgewandelt werden.
- Die Auswahl wird als User-Objekt in form.users.data verfügbar gemacht. Die User-ID ist hier schon vom Typ int.