

---

# Laborprotokoll

## Cloud-Datenmanagement

---

Systemtechnik Labor  
5BHIT 2017/18

Martin Wölfer

Note:  
Betreuer: Michael Borko

Version 2.0  
Begonnen am 5. Oktober 2017  
Beendet am 22. November 2017

# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>1</b>
1.1	Ziele . . . . .	1
1.2	Voraussetzungen . . . . .	1
1.3	Aufgabenstellung . . . . .	1
<b>2</b>	<b>First try</b>	<b>2</b>
2.1	Quickstart Tutorial . . . . .	2
2.1.1	Projekt aufsetzen . . . . .	2
2.1.2	Serializer definieren . . . . .	3
2.1.3	View definieren . . . . .	3
2.1.4	URLs definieren . . . . .	3
2.1.5	Einstellungen anpassen . . . . .	4
2.1.6	Beispiel testen . . . . .	4
2.2	Tutorial mit User Authentifizierung . . . . .	7
2.2.1	Projekt erstellen . . . . .	7
<b>3</b>	<b>Second try</b>	<b>8</b>
3.1	Dependencies definieren . . . . .	8
3.2	Entities definieren . . . . .	9

# 1 Einführung

Diese Übung zeigt die Anwendung von mobilen Diensten.

## 1.1 Ziele

Das Ziel dieser Übung ist eine Webanbindung zur Benutzeranmeldung umzusetzen. Dabei soll sich ein Benutzer registrieren und am System anmelden können.

Die Kommunikation zwischen Client und Service soll mit Hilfe einer REST Schnittstelle umgesetzt werden.

## 1.2 Voraussetzungen

- Grundlagen einer höheren Programmiersprache
- Verständnis über relationale Datenbanken und dessen Anbindung mittels ODBC oder ORM-Frameworks
- Verständnis von Restful Webservices

## 1.3 Aufgabenstellung

Es ist ein Webservice zu implementieren, welches eine einfache Benutzerverwaltung implementiert. Dabei soll die Webapplikation mit den Endpunkten `/register` und `/login` erreichbar sein.

### *Registrierung*

Diese soll mit einem Namen, einer eMail-Adresse als BenutzerID und einem Passwort erfolgen. Dabei soll noch auf keine besonderen Sicherheitsmerkmale Wert gelegt werden. Bei einer erfolgreichen Registrierung (alle Elemente entsprechend eingegeben) wird der Benutzer in eine Datenbanktabelle abgelegt.

### *Login*

Der Benutzer soll sich mit seiner ID und seinem Passwort entsprechend authentifizieren können. Bei einem erfolgreichen Login soll eine einfache Willkommensnachricht angezeigt werden.

Die erfolgreiche Implementierung soll mit entsprechenden Testfällen (Acceptance-Tests bez. aller funktionaler Anforderungen mittels Unit-Tests) dokumentiert werden. Verwenden Sie auf jeden Fall ein gängiges Build-Management-Tool (z.B. Maven oder Gradle). Dabei ist zu beachten, dass ein einfaches Deployment möglich ist (auch Datenbank mit z.B. file-based DBMS).

## 2 First try

Ich hab mich für das **Django REST Framework** entschieden, da ich schon mittelmäßige Erfahrung mit Django besitze und ich persönlich gerne mit Python arbeite. Es wurde auch Eve in Erwägung gezogen aber da ich schon mit Django gearbeitet habe, habe ich mich für diese Variante entschieden.

### 2.1 Quickstart Tutorial

#### 2.1.1 Projekt aufsetzen

Der erste Schritt war es ein Django Projekt zu erstellen und alle nötigen Packages mit zu installieren.

**Virtual Environment erzeugen** Zuerst wurde ein sogenanntes **Virtual Environment** erzeugt. Dieses erzeugt eine virtuelle Umgebung mit einer unabhängigen Installation von Python und allen anderen nötigen Zusatzpaketen.

Es gibt 2 Hauptgründe warum man ein Virtual Environment erzeugen sollte:

1. Die Umgebung ist unabhängig vom System, d.h. es ist sichergestellt das jeder der das Projekt nun öffnet die gleichen Dependencies und Packages besitzt wie auf dem System auf dem das Projekt erstellt wurde
2. Die lokale Installation auf dem System kann nicht verändert, beschädigt oder gelöscht werden

Um die Umgebung aufzusetzen und diese zu verwenden, werden folgende Kommandos verwendet:

```
1 virtualenv env
   env\Scripts\activate
```

Nun werden alle packages welche installiert werden, in dieser virtuellen Umgebung installiert und sind somit isoliert.

**Packages installieren** Die 2 nötigen Packages für das Beispiel sind `django` und `djangorestframework`, diese würden über `pip` installiert mit:

```
2 pip install django
   pip install djangorestframework
```

**Django Projekt erstellen** Das tatsächliche Projekt wird über das `django-admin.py` erstellt:

```
2 django-admin.py startproject application .
   cd tutorial
   django-admin.py startapp quickstart
```

Nun wird noch die Datenbank synchronisiert:

```
1 manage migrate
```

**Admin-User erstellen** Um die Datenbank zu verwalten wird ein **super user** benötigt. Dieser kann auch mit `manage.py` erstellt werden:

```
1 manage createsuperuser
```

Nach dem Betätigen der Enter-Taste wird veranlasst einen Namen, eine E-Mail Adresse und ein Passwort anzugeben. Für das Beispiel wurde **admin** mit dem Passwort **schueler** gewählt.

### 2.1.2 Serializer definieren

Es wird im Verzeichnis `application/quickstart` ein File namens `serializers.py` erstellt.

In diesem File wird eine `UserSerializer` Klasse erstellt, welche von `serializers.HyperlinkedModelSerializer` erbt. Dies beschreibt einfach nur wie die Relationen zueinander stehen, es können auch klassisch Primary Key und andere Beziehungen verwendet werden, aber HyperLinked Beziehungen sind gutes **RESTful** design.

In dieser Klasse wird beschrieben welche Felder der User besitzt, d.h. wenn ein neuer User erstellt welche Felder alle ausgefüllt werden müssen. Es wurden folgende Attribute entschieden:

- URL (Wird automatisch erstellt)
- Vorname
- Nachname
- E-Mail
- Passwort (Funktioniert derweil nicht als richtiges Passwort, nur als Feld angelegt)

```
1 class UserSerializer(serializers.HyperlinkedModelSerializer):  
    class Meta:  
3         model = User  
         fields = ('url', 'first_name', 'last_name', 'email', 'password')
```

### 2.1.3 View definieren

Im Verzeichnis `application/quickstart` wird das File `views.py` geöffnet. In diesem wird die Klasse `UserViewSet` erstellt. Diese Klasse definiert wie die Query für die User aussieht, wenn diese angezeigt werden sollen. In dem Beispiel schaut die Query folgendermaßen aus:

`User.objects.all().order_by('-date_joined')`. Zusätzlich wird noch der Serializer angegeben:

```
1 class UserViewSet(viewsets.ModelViewSet):  
2     queryset = User.objects.all().order_by('-date_joined')  
     serializer_class = UserSerializer
```

### 2.1.4 URLs definieren

Der nächste Schritt besteht darin die nötigen URLs zu definieren.

Da in dem Beispiel nicht einzelnen Views verwendet werden sondern **ViewSets**, kann ein **Router** verwendet werden für die URLs. Praktisch funktioniert es so, dass zuerst ein router definiert wird, in diesem Router dann das ViewSet der User **registriert** wird, und dieser Router anschließend als URL **included** wird.

Zusätzlich werden noch URLs vom `rest_framework` definiert um Login- und LogoutViews anzuzeigen:

```
1 router = routers.DefaultRouter()
  router.register(r'users', views.UserViewSet)
3
  urlpatterns = [
5     url(r'^$', include(router.urls)),
    url(r'^api-auth/', include('rest_framework.urls', namespace='rest_framework'))
7 ]
```

### 2.1.5 Einstellungen anpassen

Bevor man die ersten Ergebnisse sehen kann, müssen im File `settings.py` noch einige Einstellungen angepasst werden.

Der erste Schritt ist es in der `INSTALLED_APPS` Liste unser Framework hinzuzufügen:

```
1 INSTALLED_APPS = [
    'django.contrib.admin',
3    'django.contrib.auth',
    'django.contrib.contenttypes',
5    'django.contrib.sessions',
    'django.contrib.messages',
7    'django.contrib.staticfiles',
    'rest_framework'
9 ]
```

Zusätzlich wird noch eine Einschränkung hinzugefügt, dass nur der Admin User alles anzeigen lassen kann und es wird noch eine Seitennummerierung eingefügt:

```
REST_FRAMEWORK = {
2     'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAdminUser',
4     ],
    'PAGE_SIZE': 10
6 }
```

### 2.1.6 Beispiel testen

#### Webservice starten

Der webservice muss zuerst in der Konsole gestartet werden

```
manage runserver
```

**Zugriff durch Browser** Es kann nun auf den service via Browser zugegriffen werden über die Adresse `localhost:8000`.

Im Root Verzeichnis bekommt man lediglich eine JSON Response mit folgendem Inhalt:

```
1 HTTP 403 Forbidden
  Allow: GET, HEAD, OPTIONS
3 Content-Type: application/json
  Vary: Accept
5
  {
7     "detail": "Authentication credentials were not provided."
  }
```

**Login als Admin** Um die User sehen zu können muss sich zuerst einloggen als Admin:

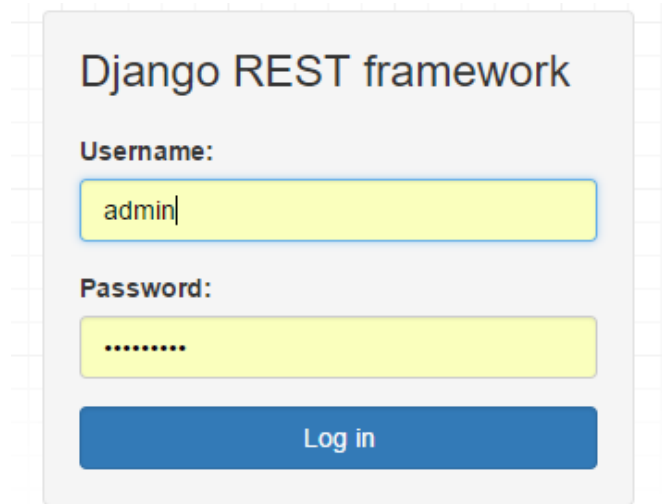


Abbildung 1: Login als Admin

Nun erhält man folgende Antwort:

```
HTTP 200 OK
2 Allow: GET, HEAD, OPTIONS
  Content-Type: application/json
4 Vary: Accept
6 {
8   "users": "http://localhost:8000/users/"
}
```

### User einsehen und hinzufügen

Um nun die CRUD-Funktionalität nutzen zu können muss auf die URL `http://localhost:8000/users` gewechselt werden.

Man kann nun alle User einsehen oder neue hinzufügen:

```
[
  {
    "url": "http://localhost:8000/users/3/",
    "first_name": "",
    "last_name": "",
    "email": "test2@hotmail.com",
    "password": ""
  },
  {
    "url": "http://localhost:8000/users/2/",
    "first_name": "",
    "last_name": "",
    "email": "test@hotmail.com",
    "password": ""
  },
  {
    "url": "http://localhost:8000/users/1/",
    "first_name": "",
    "last_name": "",
    "email": "mwolfer01@student.tgm.ac.at",
    "password": "pbkdf2_sha256$36000$gK16U9IomouK$V+xxmyx6M9S4e0Q8DoZGMRo7fI2FccehbDcWEnHHC0="
  }
]
```

Raw data HTML form

First name

Last name

Email address

Password

POST

Abbildung 2: User einsehen oder hinzufügen



**User verändern oder löschen** Da jeder User eine eigene URL besitzt, kann man diese auch einzeln einsehen und anschließend verändern bzw. löschen:

**User Instance** [DELETE] [OPTIONS] [GET ▼]

GET /users/1/

```

HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
  "url": "http://localhost:8000/users/1/",
  "first_name": "",
  "last_name": "",
  "email": "mwoelfer01@student.tgm.ac.at",
  "password": "pbkdf2_sha256$36000$qK16U9iomouK$V+xxmyx6M9S4e0Q8DoZGMRo7fI2FccehbDcWrEnhHC0="
}
  
```

[Raw data] [HTML form]

First name

Last name

Email address

Password

[PUT]

Abbildung 3: User kann verändert oder gelöscht werden

## 2.2 Tutorial mit User Authentifizierung

Nun da die Grundlagen verstanden wurden durch das Quickstart Tutorial, kann ein Projekt erstellt werden welches genau in die Thematik eintaucht. Auf der Django-REST Framework Seite gibt es ein Tutorial welches verschiedene Abschnitte unterteilt ist, welches nun Schritt für Schritt abgearbeitet und verstanden wird.

### 2.2.1 Projekt erstellen

#### Virtuelle Umgebung

Da schon für das erste Tutorial eine virtuelle Umgebung erstellt wurde, muss keine neue definiert werden aber es muss ein Paket hinzugefügt werden:

```
pip install pygments
```

Dieses Paket dient für Code-Highlighting

#### Django Projekt erstellen

Wie bereits gewohnt wird das Projekt erstellt mit:

```
1 django-admin startproject application_authentication
```

## App erstellen

Wie auch im ersten Tutorial wird nun eine App erstellt:

```
1 manage startapp snippets
```

Diese muss natürlich in `application_authentication/settings.py`, sowie das `rest_framework`, zu den `INSTALLED_APPS` hinzugefügt werden:

```
1 INSTALLED_APPS = (  
2     ...  
3     'rest_framework',  
4     'snippets.apps.SnippetsConfig',  
5 )
```

## 3 Second try

Da der erste Versuch nicht funktionierte, wurde auf eine Realisierung in JavaEE gesetzt mit Unterstützung von Spring Boot, Spring Security, Spring Data JPA und der Datenbank HSQL. Es wurde mit folgendem Tutorial gearbeitet:

<https://hellokoding.com/registration-and-login-example-with-spring-security-spring-boot-spring-data-jpa-hsql-jsp/>

Das Endergebnis ist ein RESTful Webservice bei welchem man sich einloggen und registrieren kann und mit einer Willkommensnachricht begrüßt wird.

### 3.1 Dependencies definieren

Die dependencies, d.h. welche Framework verwendet wird. Normalerweise würde das bedeuten die jeweiligen `.jar` Files herunterzuladen und dem Build-Path hinzuzufügen, aber dank Maven kann man dies ganz einfach im `pom.xml` File definieren:

```
1 <dependencies>  
3     <dependency>  
4         <groupId>org.springframework.boot</groupId>  
5         <artifactId>spring-boot-starter-web</artifactId>  
6     </dependency>  
7  
8     <dependency>  
9         <groupId>org.springframework.boot</groupId>  
10        <artifactId>spring-boot-starter-data-jpa</artifactId>  
11    </dependency>  
12  
13    <dependency>  
14        <groupId>org.springframework.boot</groupId>  
15        <artifactId>spring-boot-starter-security</artifactId>  
16    </dependency>  
17  
18    <dependency>  
19        <groupId>org.hsqldb</groupId>  
20        <artifactId>hsqldb</artifactId>  
21        <scope>runtime</scope>  
22    </dependency>  
23  
24    <dependency>  
25        <groupId>org.springframework.boot</groupId>  
26        <artifactId>spring-boot-starter-tomcat</artifactId>
```

```
29     <scope>provided</scope>
    </dependency>
31     <dependency>
        <groupId>org.springframework.boot</groupId>
33         <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
35     </dependency>
37     <dependency>
        <groupId>org.apache.tomcat.embed</groupId>
39         <artifactId>tomcat-embed-jasper</artifactId>
        <scope>provided</scope>
41     </dependency>
43     <dependency>
        <groupId>javax.servlet</groupId>
45         <artifactId>jstl</artifactId>
    </dependency>
47     <dependency>
        <groupId>junit</groupId>
49         <artifactId>junit</artifactId>
51     </dependency>
53 </dependencies>
```

Zusätzlich wird noch das **springframework** plugin definiert:

```
1 <build>
    <plugins>
3        <plugin>
            <groupId>org.springframework.boot</groupId>
5            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
7    </plugins>
</build>
```

## 3.2 Entities definieren

Mit **hsqldb** wird eine table durch die annotation **@Entity** definiert. Es wird eine Klasse **User** erstellt, welche folgende Attribute besitzt:

- Long id
- String username
- String password
- String passwordConfirm
- Set<Role> roles

Danach werden Getter- und Settermethoden definiert. Zu beachten ist, dass id den eindeutigen Primary Key repräsentiert und somit bei **getId()** die Annotations **@Id** und **@GeneratedValue(strategy = GenerationType.IDENTITY)** benötigt werden.

```
1 package com.hellokoding.auth.model;
2
3 import javax.persistence.*;
4 import java.util.Set;
5
6 @Entity
7 @Table(name = "user")
8 public class User {
9     private Long id;
10    private String username;
11    private String password;
12    private String passwordConfirm;
13    private Set<Role> roles;
14
15    @Id
16    @GeneratedValue(strategy = GenerationType.AUTO)
17    public Long getId() {
18        return id;
19    }
20
21    public void setId(Long id) {
22        this.id = id;
23    }
24
25    public String getUsername() {
26        return username;
27    }
28
29    public void setUsername(String username) {
30        this.username = username;
31    }
32
33    public String getPassword() {
34        return password;
35    }
36
37    public void setPassword(String password) {
38        this.password = password;
39    }
40
41    @Transient
42    public String getPasswordConfirm() {
43        return passwordConfirm;
44    }
45
46    public void setPasswordConfirm(String passwordConfirm) {
47        this.passwordConfirm = passwordConfirm;
48    }
49
50    @ManyToMany
51    @JoinTable(name = "user_role", joinColumns = @JoinColumn(name = "user_id"), inverseJoinColumns =
52        @JoinColumn(name = "role_id"))
53    public Set<Role> getRoles() {
54        return roles;
55    }
56
57    public void setRoles(Set<Role> roles) {
58        this.roles = roles;
59    }
60 }
```

## Abbildungsverzeichnis

1	Login als Admin . . . . .	5
2	User einsehen oder hinzufügen . . . . .	6
3	User kann verändert oder gelöscht werden . . . . .	7