Webservice - Backend

Projekt: DigitalSchoolNotes

Projekt Team: Adler, Brinnich, Hohenwarter, Karic, Stedronsky

Version 1.0 **04.11.2015** **Status: [RELEASE]**

	Datum	Name	Unterschrift
Erstellt	24.10.2015	Selina Brinnich	
Geprüft	11.04.2015	Thomas Stedronsky	
Freigegeben			
Git-Pfad: */doc/technologien*		**Dokument:** *webservice_backend_technologie.doc*	

Inhaltsverzeichnis

1 Changelog

Version	Datum	Status	Bearbeiter	Kommentar
0.1	2015-10-24	Erstellt	Selina Brinnich	Dokument erstellt
0.2	2015-10-30	Bearbeitet	Niklas Hohenwarter	Captcha und Email hinzugefügt
0.3	2015-11-04	Bearbeitet	Philipp Adler	Datenbank hinzugefügt
1.0	2015-11-04	Geprüft	Thomas Stedronsky	Rechtschreibfehler

2 Einführung

Als Webservice-Backend wird das Python-Webframework “Django” in der Version 1.8.5 verwendet. Django ist ein high-level Python Webframework, das eine schnelle Entwicklung und ein fehlerfreies, pragmatisches Design erleichtert. Es vereinfacht Web-Development und übernimmt viele der Aufgaben, die normalerweise beim Entwickeln einer Webapplikation Schwierigkeiten bereiten. Zudem ist es Open-Source und frei verfügbar. [1]

3 Installation und Konfiguration

3.1 Installation

Django kann mit folgendem Befehl installiert werden:

```
apt-get install python3-django
```

Um ein neues Django-Projekt zu erstellen wird folgender Befehl verwendet:

```
django-admin startproject dsn
```

3.2 Konfiguration für MongoDB

Django unterstützt standardmäßig lediglich relationale DBMS. Da im Projekt jedoch als Datenbank die NoSQL-Datenbank “MongoDB” verwendet wird, muss dies extra konfiguriert werden.

Dazu muss MongoEngine, ein Verbindungsstück zwischen Django und MongoDB, installiert werden. Dabei sollte die Version 0.9.0 verwendet werden, da die neueste Version 0.10.0 aktuell einen Bug hat, durch welchen man sich nicht mehr mit einer Datenbank verbinden kann.

```
pip3 install mongoengine==0.9.0
```

Um Django mitzuteilen, dass MongoDB im Hintergrund verwendet werden soll, müssen im File *prototype/settings.py* folgende Zeilen auskommentiert bzw. hinzugefügt werden:

```
#DATABASES = {
#    'default': {
#        'ENGINE': 'django.db.backends.sqlite3',
#        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
#    }
#}

import mongoengine

DATABASES = {
    'default': {
        'ENGINE': '',
    },
}

SESSION_ENGINE = 'mongoengine.django.sessions'

_MONGODB_HOST = 'localhost' #Hostname
```

```
_MONGODB_NAME = 'testy' #DB-Name
_MONGODB_DATABASE_HOST = 'mongodb://%s' % (_MONGODB_HOST)

mongoengine.connect(_MONGODB_NAME, host=_MONGODB_DATABASE_HOST)
```

4 URL Routing

Django unterstützt die Erstellung von eigenen URL Schemas. Diese werden im File *urls.py* definiert. Der Code für eine URL Definition sieht beispielsweise so aus:

```
urlpatterns = [
      url(r'^api/login', 'dsn.views.views.view_login', name="login"),
      url(r'^api/logout', 'dsn.views.views.view_logout', name="logout"),
]
```

Hier wird als erstes eine Regex des URL Schemas erstellt (hier bspw.: webpage.com/api/login). Als zweites wird eine Funktion übergeben, die ausgeführt werden soll, sobald die entsprechende URL aufgerufen wird. Da wir Django nicht zum Anzeigen von HTML-Seiten sondern nur zum Austausch von Daten verwenden, sollte in dieser Funktion anstatt des üblichen Http-Response ein Json-Response zurückgegeben werden, der die angeforderten Daten enthält (siehe Kapitel 5 Views). Als letztes wird noch ein Name festgelegt, über den die entsprechende Route referenziert werden kann.

5 Views

Django bietet zum Anzeigen von Daten unterschiedliche Response-Objekte an. Um HTML-Templates zu rendern gibt es beispielsweise das HttpResponse-Objekt. Da wir Django jedoch nicht für Templating sondern nur zum Austausch von Daten zwischen dem Frontend und der Datenbank verwenden, sollte in den View-Funktionen immer ein JsonResponse-Objekt anstatt des üblichen HttpResponse-Objekts zurückgegeben werden. Dem JsonResponse können dabei Daten in Form eines Python dicts übergeben werden. Dies kann folgendermaßen aussehen:

```
from django.http import JsonResponse

def view_example(request):
    return JsonResponse({'message':'success'})
```

6 Models

Um von Django aus auf Daten in MongoDB zugreifen bzw. neue Daten hineinspeichern zu können müssen vorher sogenannte Models erstellt werden. Diese sind im File *models.py* definiert und bilden die Struktur der Daten in der Datenbank ab. Dies kann beispielsweise folgendermaßen aussehen:

```
from mongoengine import *

class Notebook(Document):
    name = StringField(max_length=30)
    is_public = BooleanField()
    create_date = DateTimeField()
    last_change = DateTimeField()
    email = EmailField()
```

Der Klassenname (hier: Notebook) entspricht in der Datenbank später dem Collection-Namen. Wichtig bei der Klassendefinition ist, dass von `Document` geerbt wird. `Document` definiert Funktionen zum Auslesen der Daten der entsprechenden Collection aus der Datenbank, zum Erstellen eines neuen Objektes und zum Abspeichern des Objektes in der Datenbank. Die einzelnen Attribute der Klasse werden in der Datenbank als Json abgebildet, wobei der Key der Name des Attributes ist.

7 Forms

Um Formulare in Django zu validieren, können Forms verwendet werden. Diese werden im File *forms.py* erstellt. Ein Beispiel für eine Form-Definition:

```
from django.forms import Form, CharField, EmailField, DateTimeField,\
      BooleanField

class NotebookForm(Form):

    name = CharField(label="name", max_length=30, required=True)
    is_public = BooleanField()
    create_date = DateTimeField()
    last_change = DateTimeField()
    email = EmailField()
```

Um ein neues Form zu erstellen, muss in der Klassendefinition von Form geerbt werden. Anschließend können beliebig viele Attribute erstellt werden, die unterschiedliche Arten von Formular-Elementen darstellen können, bspw. Text (CharField), ein Datum (DateTimeField) oder eine E-Mail (EmailField).

Um nun dieses Form zu validieren, muss zunächst ein neues Form-Objekt erstellt und mit den gewünschten Daten befüllt werden. Anschließend kann mithilfe der verfügbaren Funktion `.is_valid()` das Form validiert werden. Die Funktion gibt dabei entweder True, wenn alle Daten korrekt sind, oder False, falls mindest ein Fehler in den Daten gefunden wird, zurück:

```
form = NotebookForm()
form.name = 'Example'
form.is_public = False
form.create_date = datetime.now()
form.last_change = datetime.now()
form.email = 'example@example.com'
valid = form.is_valid()
# Check valid and do something...
```

8 Authentication

8.1 Konfiguration

Django hat Built-In Funktionalitäten zur Authentifizierung von Benutzern und Benutzerverwaltung. Diese Funktionalitäten sind allerdings nur auf relationale DBMS ausgelegt und daher großteils nicht geeignet für unseren Anwendungszweck. Folgende Schritte sind notwendig, um die Authentifizierung über MongoDB zu ermöglichen:

Im File *settings.py* muss zu `INSTALLED_APPS` folgende App hinzugefügt werden:
`'mongoengine.django.mongo_auth'`

Außerdem muss im selben File folgender Code eingefügt werden:

```
AUTHENTICATION_BACKENDS = (
    'mongoengine.django.auth.MongoEngineBackend',
)

AUTH_USER_MODEL=('mongo_auth.MongoUser')
MONGOENGINE_USER_DOCUMENT = 'dsn.models.User'
```

Nun muss das User-Model, das eben definiert wurde, noch im File *models.py* erstellt werden. Der Code für das Model wurde aus dem entsprechenden Source-Code von MongoEngine [2] kopiert und an unseren Anwendungszweck angepasst (Einiges an unverändertem Code wurde hier ausgelassen und mit […] markiert):

```
from mongoengine.django.auth import UserManager, Permission, \
      make_password, check_password, SiteProfileNotAvailable, \
      _user_get_all_permissions, _user_has_module_perms, _user_has_perm
from mongoengine.django import auth

class User(Document):
    id = ObjectIdField(unique=True, required=True, primary_key=True)
    email = EmailField(unique=True, required=True)
    first_name = StringField(max_length=30)
    last_name = StringField(max_length=30)
    password = StringField(max_length=128)
    is_staff = BooleanField(default=False)
    is_prouser = BooleanField(default=False)
    is_active = BooleanField(default=True)
    is_superuser = BooleanField(default=False)
    last_login = DateTimeField(default=datetime.datetime.now())
    date_joined = DateTimeField(default=datetime.datetime.now())
    passwordreset= EmbeddedDocumentField(PasswordReset)

    user_permissions = ListField(ReferenceField(Permission))

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['last_name', 'first_name']
```

```
    meta = {
        'allow_inheritance': True,
        'indexes': [
            {'fields': ['email'], 'unique': True, 'sparse': True}
        ]
    }

   [...]

    @classmethod
    def create_user(cls, email, password, first_name, last_name):
        now = datetime.datetime.now()

   [...]
        user = cls(id=ObjectId(), email=email, date_joined=now, \
                       first_name=first_name, last_name=last_name)
        user.set_password(password)
        user.save()
        return user

    [...]
```

Außerdem muss in *models.py* noch ein UserManager erstellt werden:

```
class AuthUserManager(UserManager):
    def create_user(self, email, password, first_name, last_name):
        if email and password:
            try:
                User.objects.get(email=email)
                return None
            except DoesNotExist:
                try:
                    email_name, domain_part = email.strip().split('@', 1)
                except ValueError:
                    pass
                else:
                    email = '@'.join([email_name, domain_part.lower()])

                user = User(username=email, email=email, \
                             first_name=first_name, last_name=last_name)
                user.set_password(password)
                user.save()
                return user
        else:
            return None

    def create_superuser(self, email, password, first_name, last_name):
        if email and password:
            try:
                User.objects.get(email=email)
                return None
            except DoesNotExist:
                try:
                    email_name, domain_part = email.strip().split('@', 1)
```

```
                except ValueError:
                    pass
                else:
                    email = '@'.join([email_name, domain_part.lower()])

                user = User(email=email, is_superuser=True, \
                            first_name=first_name, last_name=last_name)
                user.set_password(password)
                user.save()
                return user
        else:
            return None
```

Sollten diese Schritte erledigt sein, kommt beim Ausführen allerdings immer noch folgender Fehler: ***„Metadict object has no attribute pk“***

Dieser Fehler entsteht deshalb, weil im Django-Code als Primary Key ein Integer-Wert verlangt wird, MongoDB als Primary Key allerdings eine ObjectId verwendet. Um dieses Problem zu lösen müssen im Original Django-Code folgende Änderungen vorgenommen werden [3]:

Im File *usr/local/lib/python3.4/dist-packages/django/db/models/fields* in Zeile 964:

```
return int(value)
```

ändern zu:

```
return value
```

Im File */usr/local/lib/python3.4/dist-packages/django/contrib/auth* in Zeile 111:

```
request.session[SESSION_KEY] = user._meta.pk.value_to_string(user)
```

ändern zu:

```
try:
      request.session[SESSION_KEY] = user._meta.pk.value_to_string(user)
except Exception:
      request.session[SESSION_KEY] = user.id
```

8.2 User erstellen und Passwort ändern

Sobald die Konfigurations-Schritte erledigt sind, kann über `User.create_user(...)` ein neuer User erstellt werden. Über `user.set_password(...)` kann ein neues Passwort für den entsprechenden User gesetzt werden. Das Passwort wird dabei automatisch gehasht.

8.3 Login und Logout

Um einen Benutzer anzumelden, muss zunächst ein User-Objekt mit einer bestimmten E-Mail Adresse von der Datenbank abgefragt werden. Sollte diese E-Mail Adresse keinem Benutzer zugeordnet sein, existiert der Benutzer noch nicht. Ansonsten muss mit `.check_password(...)` das eingegebene Passwort überprüft werden. Sollte dieses korrekt sein, kann der User angemeldet werden. Dazu muss das Authentication-Backend und ein Session Timeout gesetzt werden und der Benutzer über die Funktion `login()` eingeloggt werden:

```
try:
    user = User.objects.get(email='example@example.com')
except:
    user = None
if user is not None and user.check_password('myPassword'):
    user.backend = 'mongoengine.django.auth.MongoEngineBackend'
    login(request, user)
    request.session.set_expiry(60 * 60 * 1) # 1 hour timeout
```

Der aktuell angemeldete Benutzer kann mittels `request.user` abgefragt werden. Sollte der Benutzer aktuell nicht angemeldet sein, ist dies `null`, ansonsten erhält man das entsprechende User-Objekt.

Um einen angemeldeten Benutzer wieder abzumelden muss lediglich folgende Funktion ausgeführt werden:

```
logout(request)
```

9 Captcha

Das Captcha muss serverseitig validiert werden. Wie man das Captcha auf der Website anzeigt steht in der Frontend Dokumentation. Wir gehen davon aus das wir zusammen mit den Daten des Formulares ein weiteres Attribut namens `recaptcha` bekommen. In diesem steht ein Key den wir zur Validierung an Google senden müssen. Google möchte zur Validierung den Key, die IP des Users und den Secret App Key. Google gibt dann zurück, ob alles korrekt ist, also ob der User ein Mensch ist. Um die Validierung mehrmals einsetzen zu können, haben wir eine Methode dafür geschrieben:

```
def validate_captcha(recaptcha, ip):
    response = {}
    url = "https://www.google.com/recaptcha/api/siteverify"
    params = {
        'secret': settings.RECAPTCHA_SECRET_KEY,
        'response': recaptcha,
        'remoteip': ip
    }
    verify = requests.get(url, params=params, verify=True)
    verify = verify.json()
    response["status"] = verify.get("success", False)
    if response["status"] == True:
        return True
    else:
        return "Captcha ist nicht valide."
```

10 Email

Bei der Regsitrierung und beim zurücksetzen des Passworts müssen E-Mails verschickt werden. Dies geschieht wie folgt:

Als erstes muss im *settings.py* der Email Server definiert werden.

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'mxf92d.netcup.net'
EMAIL_HOST_USER = 'noreply@digitalschoolnotes.com'
```

```
EMAIL_HOST_PASSWORD = 'passwort'
EMAIL_PORT = 25
EMAIL_USE_TLS = False
DEFAULT_FROM_EMAIL = EMAIL_HOST_USER
```

Nun können Emails über den eingestellten Server verschickt werden. Der Inhalt der Email wird aus Templates erzeugt. An die Templates übergeben wir die benötigten Daten und wir bekommen einen fertigen Email Text zurück. Da manche Email Clients keine HTML Emails unterstützen schicken wir die Email auch als plain Text.

```
from django.template.loader import render_to_string
from django.core.mail import EmailMultiAlternatives


def passwordresetmail(email, firstname, link):
    subject, from_email, to = 'Passwort zurücksetzen',
'"DigitalSchoolNotes" <noreply@digitalschoolnotes.com>', email
    text_content =
render_to_string('email/passwordreset.txt',{'firstname':firstname,
'link':link}) # Plain Text Mail
    html_content =
render_to_string('email/passwordreset.html',{'firstname':firstname,
'link':link}) # HTML Mail
    msg = EmailMultiAlternatives(subject, text_content, from_email, [to])
#Email Basteln
    msg.attach_alternative(html_content, "text/html")
#HTML Email hinzufügen
    msg.send()
#Abschicken
```

11 **Suche**

Mittels dem `collectionName.objects()` liefert Mongoengine **alle Objekte** von der angegeben Collection. Wenn man aber nicht alle Objekte von einer Collection haben möchte, sondern nur eine **gewisse Anzahl**, können diese mit folgenden Befehl abgefragt werden:

```
Tabellenname.objects[x:y]
```

Objekte **suchen** kann man wie folgt:

```
users(Q(email__icontains=suchtext) | Q(first_name__icontains=suchtext) |
Q(last_name__icontains=suchtext))
```

Objekte können auch nach einer bestimmten Spalte **sortiert** werden

```
users.order_by(spaltenname)
users.order_by('-'+spaltenname)
```

Falls man ein vorhandenes Objekt aus der Collection löschen möchte, muss man dieses zuvor rausfiltern und dann mit der Funktion `delete()` entfernen. Veränderte Daten werden mit der Funktion `save()` **persistiert**.

12 *Quellen*

[1], Django Software Foundation, “Django”,
https://www.djangoproject.com/, zuletzt besucht: 24.10.2015

[2], GitHub, “MongoEngine/mongoengine, auth.py”,
https://github.com/MongoEngine/mongoengine/blob/0.9/mongoengine/django/auth.py, zuletzt besucht: 24.10.2015

[3], Ashish Khatkar, “Issue with mongo in Django”,
http://comments.gmane.org/gmane.comp.python.django.user/171657, zuletzt besucht: 24.10.2015

[4], Toan Nguyen, “How to validate Google reCAPTCHA v2 in django”,
http://stackoverflow.com/a/29592333, zuletzt besucht: 30.10.2015