Skalierbare Webanwendungen mit Python und Google App Engine

Oliver Albers

03. Juli 2008

Worum geht es?

Wir beschäftigen uns mit der Implementierung von Webanwendungen.

- Hoffentlich viele Benutzer
- Performance ist K.O.-Kriterium für Akzeptanz
- $\Rightarrow$ Skalierbarkeit und Hochverfügbarkeit

Worum geht es?

- Kaum ein Unternehmen hat hier mehr Erfahrung als Google
- Wir lernen hier die Entwicklung mit von Google bereitgestellten Tools
- Wir können die Anwendungen dann in Google-Rechenzentren hosten

Vorteile?

- Kostengünstige Lösung
- Profitieren von Googles Erfahrung

Nachteile?

- Datenschutz
- Vendor Lock-In
- Beta

Genereller Aufbau

Was ist App Engine?

- Sammlung von Python-Bibliotheken, die auf den Servern bereitstehen
- Hosting-Dienstleistung für Anwendungen
- SDK für Entwickler

http://code.google.com/appengine/

Genereller Aufbau

Python und dessen Bibliotheken wurden für diese Zwecke angepasst, es können nicht alle Features / Bibliotheken verwendet werden, z.B.

- Keine Sockets
- Keine Threads
- Keine Systemcalls
- Keine Kindprozesse, daher einige Methoden aus *os* nicht vorhanden

Anwendungskonfiguration

Die Anwendung muss für das Deployment und den Entwicklungsserver konfiguriert werden

- Konfiguration über yaml
- Datei *app.yaml* im Hauptverzeichnis der Anwendung:

```
application: randr
version: 1
runtime: python
api_version: 1
...
```

Handler

In der app.yaml können dann auch gleich URL-Handler konfiguriert werden.

```
application: randr
version: 1
runtime: python
api_version: 1

handlers:
- url: /.*
  script: index.py
```

Verarbeitung von Requests

- Alle Zugriffe werden nun auf unsere index.py gemappt, diese muss die Requests verarbeiten
- Dafür ist in Python das *WSGI*[1] zuständig
- Das SDK bringt bereits ein einfaches Framework dafür mit

[1] Web Server gateway Interface

Einfache index.py

```
import wsgiref.handlers

from google.appengine.ext import webapp

class Index(webapp.RequestHandler):
    def get(self):
        self.response.headers['Content-Type'] = 'text/plain'
        self.response.out.write('Hallo Welt!')

def main():
    application = webapp.WSGIApplication(
     [('/', Index)],
     debug=True
    )
    wsgiref.handlers.CGIHandler().run(application)

if __name__ == '__main__':
    main()
```

Verarbeitung von Requests

- Requests werden vom Hosting an unsere index.py weitergereicht
- index.py erzeugt eine WSGI-Anwendung
- Diese Anwendung leitet Requests an Instanzen der angegebenen Klasse (Kind von RequestHandler) weiter
- In diesen Instanzen wird passend zum Request get, post, put, head, ... ausgeführt

Formularverarbeitung

- An den Request übergebene Parameter sind im RequestHandler-Objekt verfügbar
- Dabei wird nicht zwischen get und post unterschieden
- us einem Formular z.B. der Wert „id“ übergeben ist er wie folgt erreichbar:
- ```
  def post(self):
      print self.request.get('id')
  ```
  ```

URL Mapping

- In unserer Anwendung können wir URLs nochmals verfeinert auf Klassen mappen
- ```
  application = webapp.WSGIApplication(
  [('/', Index), ('/number/', NumberPage),
      ('/home', Index)],
       debug=True
  )
  ```
- Es werden auch reguläre Ausdrücke unterstützt
- `(r'/home/.*', HomePage)`
  ```

URL Mapping

- Matches aus dem regulären Ausdruck können direkt als Parameter der get-Methode übergeben werden
- ```
  (r'/home/(.*)', HomePage)
  ...
  def get(self, param):
      print param
  ```
  ```

Templates

- HTML Code in den Klassen zu erzeugen ist mühselig, daher Templates
- Das SDK liefert Django[2] für Templates mit
- MVC: Python-Klassen als Controller, Templates als View-Schicht

[2] http://www.djangoproject.com/

Templates

index.py:

```
def get(self):
    template_vars = {'text': 'Hallo Welt!'}
    path = os.path.join(os.path.dirname(__file__), 'template.html')
    self.response.out.write(template.render(path, template_vars))
```

template.html:

```
<html><body>
<p> {{ text }}</p>
</body></html>
```

Statische Files

- Man muss aber auch statische Files (z.B. JavaScript-Files, CSS, Bilder) ausliefern können
- Definieren eines Verzeichnisses für statische Dateien
- Konfigurieren eines URL-Handlers für dieses Verzeichnis in der app.yaml:
- ```
  handlers:
  - url: /files
    static_dir: files
  - url: /.*
    script: index.py
  ```
  ```

Datastore

- App Engine bietet keine relationale Datenbank
- Persistente Speicherung von Daten über den Datastore

Datastore

- Verteiles System zur Datenspeicherung
- Vergleichbar mit Distributed Hashtable

API

- Datastore arbeitet nicht mit Tabellen, sondern Objekten
- Model-Schicht der Anwendung direkt auf Datastore abbildbar
- Entities müssen aber direkt auf die API angepasst sein

API

```
from google.appengine.ext import db
from google.appengine.api import users

class Pet(db.Model):
  name = db.StringProperty(required=True)
  type = db.StringProperty(choices=set(['cat', 'dog', 'bird']))
  birthdate = db.DateProperty()
  weight_in_pounds = db.IntegerProperty()
  spayed_or_neutered = db.BooleanProperty()
  owner = db.UserProperty()

pet = Pet(name='Fluffy',
          type='cat',
          owner=users.get_current_user())
pet.weight_in_pounds = 24
pet.put()
```

GQL

- Objekte können mittels GQL aus Datastore abgefragt werden
- Für einfache Abfragen werden automatisch Indizes für abgefragte Eigenschaften erzeugt
- ```
  pets = Pet.gql('WHERE weight_in_pounds > :1', 20)
  for pet in pets
    print pets[pet].name
  ```
  ```

GQL

- Ist nicht SQL
- Unterstützte WHERE-Conditions: ¡, ¡=, ¿, ¿=, =, !=, IN, ANCESTOR IS
- ORDER BY ist unterstützt
- Keine Joins

Skalierbarkeit

- Lesende Zugriffe auf den Datastore sind sehr schnell
- Schreibzugriffe in den Datastore garantieren Konsistenz
- Müssen auf Festplatte warten $\Rightarrow$ langsam!
- Möglichst nicht aus vielen Requests das gleiche Objekt beschreiben, Vermeidung durch Anpassung der Datenstruktur

Beispiel: Besucherzähler

- Jeder Request muss Objekt laden, ändern, speichern
- Obere Grenze für Skalierbarkeit also Speichergeschwindigkeit für Objekt

Beispiel: Besucherzähler

- Verwenden von n verschiedenen Counter-Objekten
- Jeder Request erhöht nur einen dieser n Counter, n * Speichergeschwindigkeit
- Zählerstand ist dann $\sum_{i=1}^{n} Counter_i$
- Schnell auszulesen
- Weiter Optimierbar durch die *memcache*-API

Benutzer

- Das SDK bringt auch eine API für Benutzerauthentifizierung mit
- Benutzer als Property-Typ für Datastore-Model vorhanden
- Einfach zu benutzen

Benutzer

```
from google.appengine.api import users

def get(self):
  user = users.get_current_user()

  if user is None:
    uri = users.create_login_url(self.request.uri)
    self.redirect(uri, False)
    return
```

Benutzer

- Benutzer-Objekte jedoch schwierig für nicht eingeloggten Benutzer zu instantiieren
- Authentifizierung nur gegen Google-Accounts

Vielen Dank!