

Google App Engine

http://www.aleax.it/it_gae.pdf



©2008 Google -- aleax@google.com

Il pubblico cui mi rivolgo

- programmatore esperti di programmazione web e di Python
- senza ancora conoscenze specifiche GAE
- meglio capire bene: HTTP, CGI, HTML, CSS, Javascript (e AJAX), ...
- ci sono scorcatoie, ma non le consiglio!
- legge di Spolsky: "All abstractions leak"
- do per scontata la conoscenza di Python (non che servano livelli eccelsi, ma...), WSGI

Parleremo di...:

- Google App Engine in generale + "come iniziare" ("hello world" ... ottimizzato;-)
- webapp, dev_appserver, e la "sandbox"
- servizi aggiuntivi vari...:
 - users
 - mail
 - urlfetch
 - logging
- il DataStore: entità, proprietà, query, indici
- Q & A (anche interattivamente, eh!-)

Cos'è Google App Engine?

- esegue le tue app web sull'infrastruttura di Google, automaticamente scalabili
- basta scrivere codice e installarlo!
- tutta l'amministrazione &c la fa Google
- iniziare è (e resterà) gratis (con limiti: per ora, 500 MB storage, 3 app/utente, pochi secondi per richiesta, 10GB/giorno in e out, 2000 mail/giorno)
- in futuro, saranno anche disponibili (a pagamento) risorse ulteriori
- per ora, solo Python (in futuro, anche altri)



Come iniziare

- mettersi in coda, <http://appengine.google.com>
 - con un qualsiasi account @gmail.com
- o <http://appengine.google.com/a/pippo.com>
 - se si hanno le Google Apps su pippo.com
- si puó servire da <nomeapp>.appspot.com
- o da qualsiasi dominio che abbia Google Apps (con "Control Panel: Next Generation")
 - si setta il dominio dalla console amministrativa dell'appengine
- 10,000 account attivi inizialmente, poi di +

Mentre si aspetta...

- ...iniziare subito a sviluppare!!!
- scaricare **Google App Engine SDK** da:
<http://code.google.com/appengine/downloads.html>
 - Windows, Mac, Linux e altri
 - tutto open-source! Richiede Python 2.5
 - consigliato per lo sviluppo, ma NON per servire davvero, eh!-)
 - nella versione "Linux e altri" i comandi vanno nel directory `google_appengine`

Hello World

- occorre un file helloworld/app.yaml:

```
application: helloworld
version: 1
runtime: python
api_version: 1
handlers:
- url: /.*
  script: helloworld.py
```

- nonché helloworld/helloworld.py

```
print 'Content-Type: text/plain'
print
print 'Hello, world!'
```

Provare 'Hello World'

- ☞ google_appengine/dev_appserver.py helloworld/
 - ☞ su Linux/altri; su Mac o Windows, solo:
 - ☞ dev_appserver.py helloworld/
- ☞ e visitare <http://localhost:8080>
- ☞ modello "sostanzialmente CGI"...
- ☞ ...e in particolare: viene ri-eseguito ogni volta (verificare con logging.info...!)
- ☞ come ottimizzare un poco...?

Hello World molto meglio

```
import logging, wsgiref.handlers
from google.appengine.ext import webapp

class MainPage(webapp.RequestHandler):
    def get(self):
        logging.info('soddisfo un get')
        self.response.headers['Content-Type'] = 'text/plain'
        self.response.out.write('Ciao, Mondo!')

    def main():
        logging.info('sono in main')
        application = webapp.WSGIApplication([('/', MainPage)],
                                              debug=True)
        wsgiref.handlers.CGIHandler().run(application)

    if __name__ == "__main__":
        logging.info('chiamo main')
        main()
```

google.appengine.ext.webapp

- semplice supporto per WSGI, piú...
- il nome 'main' + l'uso di webapp è "magico":
 - l'app engine (se ha l'app ancora in memoria) "sa" che non occorre ricaricarla
 - basta richiamare main (e solo questo fa)
- inoltre, mappa selettiva URL -> handler
- quindi: "chiamo main" solo 1 volta
 - "sono in main" sempre
 - "soddisfo un get" NON per /favicon.ico!-)

...e favicon.ico...?

- ⦿ aggiungere ad app.yaml (sotto handlers):...:
 - url: /favicon.ico
 - static_files: static/favicon.ico
 - upload: static/favicon.ico
- ⦿ aggiungere un helloworld/static/favicon.ico
- ⦿ ✗ altre immagini (e CSS, JS &c) app.yaml ha varie features: url con RE, handler script con backrefs, static_dir:, mime_type se occorre, upload con RE, default_expiration, expiration

...ma 'sto webapp...?

- ad ogni richiesta istanzia l'handler, poi chiama il metodo (get, post, head, put...)
- il metodo può usare self.request.get(...) [e i suoi argomenti facoltativi] x accesso a form, self.request.body ad es. per un put, ...
- il metodo usa self.response.out.write(...), self.response.headers[...],
- request e response documentati a: <http://pythonpaste.org/webob/reference.html>

dev_appserver

- simula l'app engine e le sue API
- serve su port 8080 (o --port) di localhost
- simula Datastore (su file), Users, Mail (richiede --smtp_host &c o Sendmail), urlfetch (con un limite sull'annidamento)
- http://localhost:8080/_ah/admin/ (console x vedere Datastore + prove interattive)
- ricarica automaticamente file modificati

La "sandbox" app engine

- no scritture su filesystem (solo letture di file dati in upload con l'app, NON static)
- no socket (& moduli che le richiedono), solo urlfetch (e spedizione mail)
- no sottoprocessi, no thread
 - 1 richiesta → 1 servizio (in pochi secondi)
 - no varie altre syscall (signal, ecc)
- limiti legali: no porno, violazione di copyright & leggi varie, violenza, razzismo, giochi d'azzardo, spam, virus, ...
- <http://code.google.com/appengine/>

login degli utenti

- con account Google, o su sito Google Apps
- from google.appengine.api import users
- users.create_login_url(self.request.uri)
 - redirect x richiedere login
 - ma NON se users.get_current_user()!
- o in app.yaml usare login: required o admin
- l'oggetto User ha .nickname(), .email(), può essere registrato nel Datastore
- ovviamente tutto facoltativo, usate qualsiasi sistema alternativo, e.g. OpenID!

e.g. di login alternativo...

- <http://code.google.com/appengine/articles/gdata.html>
- gdata.service.http_request_handler = gdata.urlfetch
 - ecc, ecc;-)
- per ottenere servizi Google (via Google Data Python client library) che richiedono autenticazione dell'utente
- gira a: <http://gdata-feedfetcher.appspot.com/>
- (presto anche a <http://code.google.com/p/google-app-engine-samples/downloads/list> ...)

Spedire e-mail

- il From: dev'essere un admin della web app
- mail.send_mail(sender, to, subject, body)
 - facoltativamente, attachments (testi, immagini, PDF, *no* eseguibili ecc...)
 - e altri arg facoltativi (bcc, reply_to, ...)
 - o "vestito" da OOP, class EmailMessage
- funzionalità asincrona (errori al sender)

urlfetch

- `result = urlfetch.fetch(url)`
- `method`: GET, POST, HEAD, PUT, DELETE
- `post` e `put` richiedono un 'payload'
- opzionale: dict '`headers`'
- solo http su port 80, https su port 443
- *sincrono* (workaround: javascript...!-)
- rischio di timeout su URL lente
- NON segue automaticamente i redirect
(usare `result.status_code` e `.headers ...!`)
- `result` ha: `content`, `status_code`, `headers`

logging (da stdlib)

- supp.: debug, info, warning, error, critical
- l'Administration Console ad appengine.google.com permette di consultare i log (e altri dati, ecc ecc)

Datastore

- from google.appengine.ext import db
- ereditare da db.Model, con attributi di classe di tipo db.StringProperty, db.UserProperty, db.DateTimeProperty...
- interrogazioni con db.GqlQuery (SQLoide ma senza join e select "veri" [selettivi;-)])
 - anche: metodo .gql del modello
 - senza il 'SELECT * FROM Pippo'...
 - o: metodi all, filter, order...

Datastore: Entity

- un'Entity ha una chiave unica (di solito un id numerico fatto automaticamente dal DS, ma può anche essere una stringa data dall'app) e varie proprietà (dei vari tipi previsti)
- varie entità dello stesso "kind" possono avere proprietà diverse ("Expando")
- un Model descrive un "kind" di Entity (con proprietà obbligatorie [richieste nel ctor], eventuali default), ev. subcl di db.Expando (con proprietà dinamiche, non validate, e totalmente facoltative)

Datastore: proprietà

- StringProperty: <500 byte, indicizzate e quindi usabili per filtri ('WHERE') e 'ORDER'
- TextProperty: lunghezza arbitraria, NON indicizzate (no filtri, no order...)
- BlobProperty: arbitrarie stringhe di byte
- ListProperty: liste (elementi di 1 solo tipo!)
 - usabili nei filtri (la condizione è soddisfatta se vale per 1+ elementi!) e order (in modo assai peculiare...)

Datastore: reference

- ReferenceProperty: Key di altra entitá (di "kind" fisso; anche SelfReferenceProperty)
 - usabile come entitá (caricamento automatico), assegnabile da entitá
 - ma l'entitá riferita puó venire cancellata (usare .get() e testare il valore-veritá!)
- esiste automaticamente "backreference" per ogni ReferenceProperty (nomemodello_set di default, o il valore del parametro facoltativo 'collection_name' della r.p.)

Datastore: Property

- è un descrittore (nella classe-modello)
- classe "praticamente astratta", subclassabile
- ctor: verbose_name (per label nelle form), name (per le query, se occorre sia diverso dal nome attributo), default, required (bool), validator (funzione di validazione, solleva un'eccezione x valori non validi), choices (elenco di valori accettabili)
- le subclass definiscono data_type (tipo Python da memorizzare)

Altre property

- BooleanProperty, IntegerProperty, FloatProperty, DateTimeProperty
- UserProperty (attenti ai default!!!)
- CategoryProperty (per i "tag" e.g. Atom)
- LinkProperty, EmailProperty
- GeoPtProperty (lat & lon, due float)
- IMProperty (protocol + address)
- PhoneNumberProperty, PostalAddressProperty
- RatingProperty (intero 0-100)

Datastore: Query

- ctor accetta un modello (come .all su esso)
- poi: filter, order, ancestor (chainable...)
- infine: get [1 solo elemento, il primo...], fetch [con limit e offset], o semplicemente iterare
 - o anche: count, meglio se con un limit
- anche: GqlQuery con stringa SQLoide nel ctor (sempre 'SELECT * FROM Modello'...)
- anche Modello.gql(...) con solo WHERE e/o ORDER
- solo parametri bound (:1 o :nome)

Datastore: funzioni

- get (con 1 chiave o lista di chiavi)
- put (con 1 entità o lista di entità)
- delete (idem)
- run_in_transaction(funz, *a, **k)
 - funz puo` sollevare un db.Rollback...
 - "optimistic concurrency" e retry

Datastore: query e indici

- ogni query richiede un indice
 - specificamente: ogni combinazione di kind, proprietà & operatore nei filtri, order
 - i valori dei filtri non cambiano l'indice
- se manca l'indice, la query fallisce
- alcuni indici esistono di default
- index.yaml può definire altri indici necessari
 - dev_appserver lo costruisce/aggiorna, si può anche editarlo "a mano"!
- limiti sulle query: diseguaglianze su 1 sola proprietà (e va in order prima delle altre)...

Domande & Risposte

D?

R!