Using Google App Engine

Bohuslav Kabrda

December 13, 2009

Introduction

The aim of this paper is to provide a description of using *Google App Engine* (or *GAE*) with Python. It contains all important information about using the *webapp* framework and the *Datastore*. I concentrated on making this paper shorter than the Google tutorials and references [1], while heavily drawing from them. Therefore, this paper is not meant to be innovative – it is rather a compilation of information that I thought were important for creating real applications for *GAE*.

I added my own examples and tried to find and explain problems that are left unexplained or not talked about in Google references.

Contents

1 What Is Google App Engine?

GAE is a mechanism which lets web applications run on Google infrastructure. It is based on cloud computing and it spans applications over multiple servers. It was first released in April 2008 (as a beta version). The main purpose of this technology is to let the web developers to focus on programming itself, while deploying, scaling and database management is done automatically.

Applications may be written using Java technologies (which means either Java with JavaEE technologies or any language that can run inside the Java JVM – for example Ruby – can be used) or using Python.

GAE provides applications with a persistent storage, the so called *Datastore*, that has the ability to scale automatically according to the way it is being used. It can be used with JPA/JDO or Python modeling API.

All applications run in a *Sandbox*, which means that they have some limitations (for example they cannot write to the file system, only read from it), on the other hand, this ensures that they can scale more efficiently and automatically.

1.1 Java Runtime Environment

The JRE used provides applications with most standard Java web development tools (such as JSP, JPA/JDO, JavaMail, etc. . . , but neither EJB nor JDBC).

JRE on Google App Engine currently uses Java 6, the JVM runs inside the already mentioned *Sandbox*.

1.2 Python Runtime Environment

Java App Engine uses optimized Python interpreter (currently version 2.5) and it also provides developers with a special web application framework (called *webapp*) and tools for accessing the stored data. Other web frameworks such as *Django*[1] may also be used.

Due to the *Sandbox* restrictions, the code must be written in Python exclusively, extensions written in C language are not supported.

[1] See `http://www.djangoproject.com/`

2 Datastore

The *Datastore* provides applications running on Google App Engine with the only option to store the application data.

The *Datastore*:

- scales "on demand" (which means it scales by monitoring amounts of stored data and number of queries).
- supports developers with transactions (using *optimistic concurrency control*[2]).
- is strongly consistent.

The *Datastore* is not:

- traditional relational database – entities (data objects) are stored and loaded based on values of their properties.
- based on tables – the entities are schemaless, their structure is provided by the application code.

2.1 A Closer Look Inside – BigTable

This section draws from [2].

The Datastore is built on *BigTable*, a technology based on *Google File System* (*GFS*). *BigTable* is designed to be a very fast and extremely scalable DBMS. It is, for example, used for storing data for Google projects like web indexing or Google Earth.

2.1.1 BigTable Data Model

BigTable is basically a multi-dimensional sorted map. It is indexed by a row key, column key and a timestamp, each value is an uninterpreted array of bytes.

The row keys (names of the rows) are arbitrary strings and each operation under a single row key is atomic. The data is maintained in lexicographic order (ordered by the row key). The range of the rows is partitioned, making each range (called *tablet*) a basic unit of distribution and load balancing.

The column keys are divided into sets called *column families*. The *column families* are the basic units of access control. All data in one family typically have the same type and are compressed together. The column keys are used in the format *family:qualifier*, thus forming a kind of a multi-level hashtable.

The timestamp dimension of BigTable is used for keeping different versions of data.

[2] *Optimistic concurrency control* is a concurrency control method, in which the transaction assumes that it will not be interrupted while executing, therefore it doesn't lock the table. Before committing, it checks that the data haven't been modified by any other transactions. If they have been modified, the transaction rolls back.

2.1.2 BigTable Technologies

As already mentioned, *BigTable* is based on *GFS*. It is designed to operate on a cluster and depends on a cluster management system, which manages resources and schedules jobs. *BigTable* also relies on distributed lock service *Chubby*, that uses the *Paxos* algorithm[3].

2.2 Datastore API

The primary subject of storing is an entity (also called data object, *DO*). Each entity has one or more properties of several basic data types, including a reference to another entity.

The datastore API is discussed in the chapter 4

[3]Basically, the participants of the process have to agree on the result. See [3] for more information.

3 Python Applications on Google App Engine

This section shows how to develop few simple applications for Google App Engine using Python. You will need Python 2.5 interpreter, *GAE* SDK and a Python editor (i suggest using Eclipse with the Pydev module).

3.1 Setting Up the Working Environment

Install the Python interpreter[4] GoogleAppEngineSDK[5] and Eclipse[6] with Pydev[7]. In Eclipse, add the directory containing GoogleAppEngineSDK directory with libraries (e.g. `google_appengine\lib` to Python Libraries – use `Window` → `Pydev` → `Interpreter - Python` → `New Folder` . The environment is now ready and you can create your first project.

3.2 Hello, World!

In Eclipse, choose `File` → `New` → `Other` → `Pydev` → `Pydev Google App Engine Project` then click the Next button. Type `helloworld` as the project name. Then choose grammar version 2.5 and click Next again. On the next page, browse to the folder with your instalation of *GAE* (e.g. `C:\Program Files\Google\google_appengine\`) and again click the Next button. Select "Hello world" template and click the Finish button. The project is now created.

3.2.1 Examining the Project

As you can see, one directory (`src`) containing two files (`app.yaml` and `helloworld.py`) was created. The `helloworld.py` is just a simple script that outputs the `Content-type` header and a line "Hello world":

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

The second file is more interesting for us. It is a file written in *YAML*[8] and we will call it the configuration file:

```
application: sample-app
version: 1
runtime: python
api_version: 1
```

[4]Download from URL `http://www.python.org/download/releases/2.5/`

[5]Download from URL `http://code.google.com/intl/cs/appengine/downloads.html#Google_App_Engine_SDK_for_Python`

[6]Download from URL `http://www.eclipse.org/downloads/`

[7]Download from URL `http://pydev.org/download.html`

[8] "A human-readable data serialization format", see `http://www.yaml.org/`

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

The line `application: sample-app` is called the application identifier. The version line contains the number of a version – it is very important to use it properly, because after uploading newer version of the application to the App Engine, you can always go back to the older versions. Runtime environment is, of course, Python and the version of the API is set to `1` (currently, there is only this version, but more are to come in the future). The `handlers` section is responsible for mapping URLs to the scripts (or images, stylesheets etc...). Note, that regular expressions can be used in mapping patterns.

3.2.2 Running the Application

Run the *GAE* Launcher (which was installed with GoogleAppEngineSDK) and add the "Hello, World!" application (`File` → `Add existing application` and input the path to the `src` in the `helloworld` directory of the application. Select the project, click `Run` and browse `http://localhost:8080/` in your browser. You should see the "Hello, world!" greeting.

3.3 The webapp Framework

There are several Python frameworks for web development, like *Django*[9], *CherryPy*[10] or *web.py*[11]. Google, however, have decided to make their own simple framework embedded in GoogleAppEngineSDK (which means you don't have to install it, everything is ready to go, you only need to include it in the source files). This framework is called *webapp* and we will be using it throughout this whole tutorial. We will also use some *Django* functionality - namely the HTML *templates*[12].

The application written in *webapp* has three important parts:

- a part that processes request and builds response – it consists of one or more classes extending `RequestHandler`
- an instance of `WSGIApplication` which routes the requests to handlers based on URL
- a main routine that runs the `WSGIApplication`

[9] `http://www.djangoproject.com/`
[10] `http://www.cherrypy.org/`
[11] `http://webpy.org/`
[12] See `http://docs.djangoproject.com/en/dev/ref/templates/`

3.3.1 Creating a Sophisticated Application

Rewrite the `helloworld.py` file to look like this:

```
import os
from google.appengine.ext import webapp
from google.appengine.ext.webapp import template
from google.appengine.ext.webapp.util import run_wsgi_app

class MainPage(webapp.RequestHandler)
    def get(self):
        template_values = {
            'title': 'Hello, world!',
            'header': 'Hello, sophisticated world!'
        }
        path = os.path.join(os.path.dirname(__file__),
            'index.html')
        self.response.out.write(template.render(path,
            template_values))

class ErrorPage(webapp.RequestHandler):
    def get(self):
        template_values = {
            'title': 'Error!',
            'header': 'Yikes, this page doesn\'t exist!'
        }
        path = os.path.join(os.path.dirname(__file__),
            'index.html')
        self.response.out.write(template.render(path,
            template_values))

#the action done by the MainPage handler is
#mapped to '/' and to '/sayhello' URLs
application = webapp.WSGIApplication([('/', MainPage),
                                      ('/sayhello', MainPage),
                                      ('/.*', ErrorPage)],
                                      debug=True)

def main():
    run_wsgi_app(application)

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

Then change the `app.yaml`:

```
application: sample-app
version: 1
runtime: python
api_version: 1

handlers:
- url: /stylesheets/.*
  static_dir: stylesheets

- url: /.*
  script: helloworld.py
```

After altering the configuration file, create a new directory in `src`, called `stylesheets` and create a simple stylesheet `style.css` in it:

```
body {
  background-color: brown;
}

h1 {
  color: white;
}
```

Finally, create the `index.html` *template* in the `src` directory:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
  charset=UTF-8">
<link rel="stylesheet" type="text/css"
  href="/stylesheets/style.css">
<title>{{ title }}</title>
</head>
<body>
<h1>{{ header }}</h1>
{% ifequal title 'Error!' %}
<b>This is so sad...</b>
{% else %}
<b>You made it!</b>
{% endifequal %}
</body>
</html>
```

Save all modified and created files and try to reload the `http://localhost:8080/` page. Note, that you don't need to restart the application in Google App Engine Launcher, it reloads automatically.

3.3.2 What Does This Do?

How does this code work? When an HTTP request is sent to the server, the requested URL is checked against the `app.yaml` file, resolving the name of file to use for creating a response. The response can be made by serving a static file or running a script. It is important to know, that the URL patterns are tested from top to bottom as written in the configuration file, which means that if we had switched the two patterns, the `helloworld.py` script would have been used everytime, and the second pattern would have never come to use (and the stylesheet wouldn't have been loaded). So, as you can see, if the client is asking for a file inside the `stylesheets` folder, it gets the file directly, otherwise the `helloworld.py` script is run and the result is returned.

Probably the most interesting thing on this example is the HTML file *template* and how we have used it. If you take a closer look at the `helloworld.py` file, you will notice, that the two handlers (`MainPage` and `ErrorPage`) don't output any HTML code directly. They ather save the parameters to a *dictionary*[13], which is then used by the *template* to make the response page.

The `index.html` file is a simple HTML template written using *Django* functionality implemented in *webapp*. You can use *templates* to output various data computed by request handlers. *Templates* also support basic condition handling and loops.

3.3.3 Conclusion of This Basic Example

There are several important things to notice about this example:

- The order of patterns in `app.yaml` is important.
- The order of patterns in `application` variable in the script file is also important (for the same reason).
- You have to discriminate between scripts and static files.
- You can use *Django templates* when writing applications in *webapp* framework.

3.4 Limitations

This section lists all important limitiations for applications running on *GAE*.

1. Request handler has a limited amount of time to process request (currently 30 seconds), when this deadline is reached, the handler is interrupted.
2. The application cannot write to the filesystem (*Sandbox* restriction).

[13] A Python data structure

3. The application cannot connect to any host directly, it has to use the *GAE* URL fetch service (*Sandbox* restriction).

4. The application cannot spawn sub-process or thread (*Sandbox* restriction).

5. All code has to be in pure Python, no C (or other languages) extensions can be included.

6. All imports are cached, which means, that global variables are initialized only once, when the module is first imported, their value then remains the same between requests.

In the next chapter, we will finally get to the main focus of this work – using *Datastore*.

4 Using Datastore With Python

4.1 Data Modelling

The Datastore API (namely package `google.appengine.ext.db`) doesn't need any table definitions like relational databases do. The format of the stored data is described by subclasses of the `db.Model`, `db.Expando` and `db.PolyModel` classes.

4.1.1 Models and Properties

Let's look at a small example:

```
class Chair(db.Model):
    legs = db.IntegerProperty(required = True)
    description = db.StringProperty()
    is_armchair = db.BooleanProperty(required = True)
    created_on = db.DateProperty(auto_now_add = True)
    owned_by = db.UserProperty(auto_current_user_add = True)
    materials = db.ListProperty(basestring)
```

We have created a class representing a chair with six attributes. The first four attributes store data as you would expect using common data types. Some of the attributes are marked as required (which means, that they need to be passed to the constructor when creating an instance). Attributes that are not required and not set are stored as `None` values. The property `created_on` will be assigned current date on storing, if uninitialized. Property `owned_by` is a reference to a Google account (it defaults to currently signed user). With *webapp*, you can even use Google accounts for authentication in your application. The last property is a list of strings, which can also be defined as `db.StringListProperty`. You can now make an instance of the `Chair` class:

```
broken_chair = Chair(legs = 3,
                     is_armchair = False,
                     materials = ['rust', 'wood_worm'],
                     created_on = datetime(1900, 1, 1))
```

Any attributes that you might assign to the chair that are not specified in the model will not be saved in the datastore (which could be useful for storing values used only at runtime).

Sometimes you might need to store different attributes for instances of the same class. That is what the Expando Models are used for. The `db.Expando` class is a subclass of `db.Model`. It can define both fixed and dynamic properties. All properties of each instance are stored in the database:

```
class Person(db.Expando):
    sex = db.StringProperty(required = True,
```

```
        choices = set(['female', 'male', 'alien']))
    height = db.IntegerProperty()

p = Person(sex = 'female')
p.name = 'Monty'
```

The attributes defined in the class definitions have the same semantics as in the `db.Model` definition, but all the dynamic attributes will be stored as well. Note, that dynamic attribute with value `None` is not the same as a dynamic attribute that doesn't exist (which means that the entity simply doesn't have it).

The last type of model is a Polymorphic Model (`db.PolyModel`). It is used to create class hierarchy:

```
class Food(db.PolyModel):
    calories = db.IntegerProperty(required = True)

class Spam(Food):
    is_tasty = db.BooleanProperty(required = True)

class Beans(Food):
    quantity = db.IntegerProperty(required = True)
```

All `Spam` and `Food` instances will now have the `calories` attribute. A *Datastore* query (described in 4.2.3) for Food will return `Food`, `Spam` and `Beans` entities, while a query for `Spam` will only return `Spam`.

Dynamic attributes for `db.PolyModel` descendants aren't stored.

4.1.2 Special Properties

There are few interesting properties, that we will now have a closer look at.

A string can be stored as `db.StringProperty` (indexed) with maximum length of 500 bytes or as a `db.TextProperty` with maximum length of 1 megabyte (unindexed). Binary data can be stored in similar types – 500 bytes long indexed `db.ByteString` and 1 megabyte long unindexed `db.BlobProperty`.

What you have probably thought of already is how to model entity relations. This can be achieved by using `db.ReferenceProperty`:

```
class Alien(db.Model):
    has_flying_saucer = db.BooleanProperty(required = True)

class Human(db.Model):
    kidnapped_by = db.ReferenceProperty(Alien, required = True)
```

```
alien = Alien(has_flying_saucer = True)

#these two lines do the same
human1 = Human(kidnapped_by = alien)
human2 = Human(kidnapped_by = alien.key())
```

The `human1` and `human2` have been kidnapped by the same `alien`, as you have probably expected. If `alien` is not in the memory, but only in the *Datastore* and one of the human entities is used, the `alien` is fetched automatically. The `key()` method will be described in 4.2.3.

You can use `db.SelfReferenceProperty` for reffering to an entity of the same kind.

For detailed description of all datatypes, see `http://code.google.com/intl/cs/appengine/docs/python/datastore/typesandpropertyclasses.html`.

4.2 Storing and Retrieving Data

Up until now, we haven't stored a single byte of data into the Datastore. To do that, we will first need to know what a `Key` is.

4.2.1 Keys

A *key* (represented by a `Key` object) is a unique identifier for the entity across all entities for one application. Each *key* has several components:

- *path* – describes parent-child relationship between this entity and another entity (we will talk entity relationships in 4.3)
- *kind* of the entity
- either a *name* assigned to entity by application or an *ID* assigned by the datastore

The `Key` attribute is created upon storing the entity and cannot be changed!

4.2.2 Creating and Updating Entities

Creating and updating entity is very simple. The only thing you need is to call the `put()` method of `db.Model`/`db.Expando`/`db.PolyModel` subclass instance and the Datastore API takes care of the rest. When you call this method for an instance that hasn't yet been stored, it is saved and given a `Key`. If the entity already exists in Datastore upon calling `put()` method, its properties are updated.

The `put()` method returns the `Key` of the stored entity.

You can use the `db.put()` function to put a list of entities (list of `Keys` is returned in this case).

Examples:

```
class Movie(db.Model):
  title = db.StringProperty(required = True)
  length = db.IntegerProperty()
  director = db.StringProperty()

life_of_brian = Movie(title = 'Life of Brian'
                      length = 123)

#store this movie
life_of_brian.put()

#update this movie
life_of_brian.length = 234
life_of_brian.put()

holy_grail = Movie(title = 'Monty Python and the Holy Grail'
                   length = 100)
best_of = Movie(title = 'Best of Monty Python\'s Flying Circus')

#put more movies at once
db.put([holy_grail, best_of])
```

4.2.3 Retrieving Entities – Queries

The *Datastore* API provides you with two similar ways of retrieving the stored data: the *Query* interface and the *GqlQuery* interface.

The *Query* interface lets you get entities using the `all()` class method of a `db.Model`/`db.Expando` subclass. Calling this method returns a `Query` object representing a query for all entities of given kind. You can than adjust this query by calling its methods: `filter()`, `order()`, `ancestor()`. The `ancestor()` method is explained in 4.3.

Using more than one filter in query means, that objects returned by the query will match all conditions (in other words, filters are interpreted as a *conjunction* of predicates). Unfortunatelly, there is no "`LIKE`" operator for matching string properties, the application has to filter those in the program code.

You can order entities ascending (use property name as a parameter for `order()`) or descending (use property name prefixed by `-` as a parameter for `order()`). If the values of the property used for ordering have different types, they are first ordered by type and then by value.

This kind of query also lets you to pull out only the `Key`s of the entities by passing `keys_only = True` to its constructor.

Examples:

```
query = Movie.all()
```

```
#select all movies directed by Steven Spielberg, shorter than
#200 minutes and sort them by title descending
query.filter('length <', 200)
query.filter('director =', 'Stephen Spielberg')
query.order('-title')

#do the same shorter (even on one line)
query.filter('length <', 200)
    .filter('director =', 'Stephen Spielberg')
    .order('-title')
```

The *GqlQuery* interface lets you use the *GQL* – an SQL-like query language. `GqlQuery` is a class located in `google.appengine.ext.db`. The constructor takes a string representation of query and optional parameters for binding.

Examples:

```
query = Movie.all()

\begin{verbatim}
#do the same as in the previous example
#use positional parameter binding
query = db.GqlQuery("SELECT * FROM Movie"
                    "WHERE director = :1 AND length < :2"
                    "ORDER BY title DESC",
                    'Steven Spielberg', 200)

#do the same, but use keyword parameter binding
query = db.GqlQuery("SELECT * FROM Movie"
                    "WHERE director = :director AND length < :length"
                    "ORDER BY title DESC",
                    'Steven Spielberg', 200)
```

You can also use string, boolean and number values as literals in the query string. There is one more way of creating a query. The equivalent to the above example would be:

```
query = Movie.gql("WHERE director = :director AND length < :length"
                  "ORDER BY title DESC",
                  'Steven Spielberg', 200)
```

The queries (both `Query` and `GqlQuery`) do not execute automatically. You either have to try to use the entities retrieved (use the query as an iterator) or use the `fetch()` method of a query object. The fetch method has two parameters: the `limit` (maximum

number of entities to fetch) and optional parameter `offset` (number of entities to skip – *these entites are not returned by `fetch()`, but are loaded into the memory!*). This method returns a list of entities.

Examples:

```
#fetch 10 entities, but skip 5
#we get 5 entites as the result of fetch(),
#but all 10 entities are loaded into memory
results = query.fetch(10, 5)
for result in results:
    print "Title: " + result.title

#use query as an iterator
#all entities are fetched
for result in query:
    print "Title: " + result.title
```

Currently, the *Datastore* allows you to retrive only 1000 entities in one query.

One more way to retrive an entity by using its `Key`:

```
#construct a hyperref somewhere else
movie = Movie(title='Holy Grail')
self.response.write('<a href="/edit?key=%s">%s</a>'
% (str(movie.key()), movie.name()))


#get the movie key from the parameter
key = self.request.get('key')
#you have to convert the string to the key object
entity = db.get(db.Key(key))
```

The `get()` method used without any parameters returns the first entity of the result. If an entity has a `db.ReferenceProperty`, you can use the parent entity for result filtering and you can retrive the referenced entity using this `db.ReferenceProperty`.

```
class Viking(db.Model):
    name = db.StringProperty(required = True)

class BowlOfSpam(db.Model):
    owner = db.ReferenceProperty(Viking, required = True)

#lets presume that Viking Harry has a bowl of spam
#stored in the Datastore
harry = Viking(name = 'Harry')
```

```
#query for his bowl of spam
query = BowlOfSpam.all()
query.filter('owner = ', harry)

#equivalent:
query = BowlOfSpam.all()
query.filter('owner = ', harry.key())

#this statement returns Harry
query.get().owner
```

Filters can only contain comparison operators (`<`, `>`, `<=`, `>=`, `=`, `!=`) and the `IN` operator.

The "`!=`" operator in fact runs two sub-queries – one with the "`!=`" operator replaced by "`<`" and one with the "`!=`" operator replaced by "`>`". The results are then merged and returned. A query can only have one "`!=`" operator.

The "`IN`" operator performs multiple sub-queries – one for each value in the provided list (the "`IN`" operator is replaced by "`=`"). Again, the results are merged and returned. A query can contain more "`IN`" operators, but in this case, the query is run for each combination of the values of provided list (which can result in many sub-queries).

Moreover, the inequality operators (`<`, `>`, `<=`, `>=`, `!=`) can only be used on one property and if the results are to be sorted, they must be sorted by the properties used for inequality comparisons first.

A query can currently have a maximum of 30 sub-queries.

4.2.4 Deleting Entities

Entities can be deleted by calling the `delete()` method of `db.Model` subclass instance or by using the `db.delete()` function - it requires a single `Key` or entity (or list of `Keys` or entities):

```
query = db.GqlQuery("SELECT * FROM Movie"
                    "WHERE director = :1 AND length < :2"
                    "ORDER BY title DESC",
                    'Steven Spielberg', 200)
results = query.fetch(10)
for result in results:
    result.delete()

#do the same faster
query = db.GqlQuery("SELECT * FROM Movie"
                    "WHERE director = :1 AND length < :2"
                    "ORDER BY title DESC",
```

```
                    'Steven Spielberg', 200)
results = query.fetch(10)
db.delete(results)
```

For a complete reference for creating queries, see `http://code.google.com/intl/cs/appengine/docs/python/datastore/queryclass.html` and `http://code.google.com/intl/cs/appengine/docs/python/datastore/gqlqueryclass.html`.

4.3 Entity Groups and Relationships

In this chapter, we will take a closer look at entities and entity relationships – a feature that is pretty much necessary in any kind of application.

4.3.1 Entity Groups

Every entity stored in the *Datastore* belongs to an *entity group*. Each *entity group* has one *root* (an entity without a parent). Other entities may have the *root* entity set as a parent (which can be done in entity constructor), some other entities may set these as their parents and so on, forming a *tree*. A *path* is a chain of entities from an entity to the *root*. All entities in the *path* of a single entity are called *ancestors*.

What are the *entity groups* good for?

- All entities belonging to the same group are stored in the same datastore node.
- Only entities in the same *entity group* can be modified in transactions (see 4.4).
- Although it would seem, that putting entities in one entity group makes operations with them faster, `http://code.google.com/intl/cs/appengine/docs/python/datastore/keysandentitygroups.html` claims, that doing so has no significant impact on the speed.
- The *entity groups* should only be used when you need to change entities in transaction. You should use `db.ReferenceProperty` otherwise.

Note, that entites are not deleted when you delete one of their *ancestors*.

4.3.2 Filtering Entities By Ancestors

You can also use the ancestors similarly as filters in queries:

```
class A:
  pass

q = A.all()
q.ancestor(ancestor_key)
```

4.4 Transactions

A natural requirement when using various DBMSs is the ability to change data in transactions. A transaction has to be *ACID* – (atomic, consistent, isolated, durable).

4.4.1 Using Transactions

Let's see a simple use of transactions:

```
class Counter(db.Model):
  count = db.IntegerProperty()

#the function to run in transaction
def increment(key):
  counter = db.get(key)
  counter.count += 1
  counter.put

counter_key = Counter.all().get().key()

#run the transaction
db.run_in_transaction(increment, counter_key)
```

As you can see, the only thing you need is to write an ordinary Python function and than call `db.run_in_transaction()` function, which takes the function object and its parameters to pass to the it.

If the function throws an exception, the transaction is rolled back (has no effects on *Datastore*) and the exception is re-thrown, unless it is a `Rollback` exception, which is caught and `None` value is returned.

If the function returns successfully, the transaction is committed and the function return value is returned.

The `db.run_in_transaction()` tries to run the function few times if it doesn't succeed on the first run, which means, that functions used in transactions should not have any side effects. To adjust the number of function runs, use the `db.run_in_transaction_custom_retries()` function (which is the same as `db.run_in_transaction()`, but it has an extra parameter which determines the number of runs).

4.4.2 Transaction Limitations

There are several transactions limitations that one needs to be aware of:

- Each transaction can only use entities from one *entity group*.
- If a query is performed during a transaction, it has to include the *ancestor* filter.

- A transaction can contain any Python code, but it should be as fast as possible to minimize the chance of being interrupted by any other *Datastore* operations.

4.5 Indexes

An *index* is a table that contains potentional results of a query in a desired order (that means, that *index* is basically a table containing all entities of given kind). The indexes are defined in `index.yaml` file. The development server configures the indexes automatically as it executes queries, that do not have *index* yet. You can also configure *indexes* manually.

As we have seen in 4.2.3, the queries are used to retrieve data from the *Datastore*. Each query has to specify the entity kind and may also specify *filters* and sort orders.

4.5.1 Explaining Indexes

Each query run from the application has to have a proper *index* configured in the *index.yaml*. When the application changes an entity, all indexes containing this entity are updated accordingly. When the application executes a query, all matching entities are retrieved from the proper *index*. *Indexes* are stored for all combinations of kind, *filters* and sort orders used in queries throughout the application.

Let's look at an example query:

```
SELECT * FROM Movie WHERE director = 'John Smith'
                      AND title = 'Custom Title'
                    ORDER BY title ASC
```

Index for this query contains columns with `Keys`, directors and titles, the rows are sorted by titles ascending. Now, all queries filtering movies by director and title, asking to sort results by title ascending use this *index*. For example:

```
SELECT * FROM Movie WHERE director = 'Someone Else'
                      AND title = 'Document About Spam'
                    ORDER BY title ASC
```

The `index.yaml` file for application using these queries would look like this:

```
indexes:

- kind: Movie
  properties:
  - name: director
  - name: title
    direction: asc
```

In 4.1, it has been pointed out, that it is not the same, when an entity has a property with `None` value and when the entity simply doesn't have the property. The reason is, that an entity without a property is not included in an *index* for a query using that propert (while the entity with this property of value `None` is included).

We have also seen, that some properties aren't indexed – namely `db.TextProperty` and `db.BlobProperty`. This means, that these properties cannot be used in *filters*. A property can also be marked as non-indexed by passing the `indexed = False` parameter to its constructor.

4.5.2 Exploding Indexes

The *Datastore* lets you store the `db.ListProperty`, which could be useful, but it can have one negative effect called *exploding index*.

When you store a `db.ListProperty`, an index entry is made for each value of the list in each *index* using this value. And what is worse – when an entity has two or more list properties, all permutations of values are stored in each *index*. And because each entity has a limited number of index entries, this can couse an application not to work properly (not speaking of significant performance drop).

5 Conclusion

GAE seems to be a big step to making web development easier. The *webapp* framework and *Datastore* API are simple to learn, yet powerful tools. A simple application can be made fast and efficiently without a big effort.

On the other hand, there are many downsides. The *webapp* framework (lacking some important features like redirecting between Request Handlers) can be substituted by any other Python framework, but the *Datastore* queries seem to have to many limitations to be used in large and complex applications.

Still, *GAE* is certainly worth trying when you want to see where the modern web development is going, or you just want to write a simple web application to be run on Google infrastucture.

References

[1] Google Code. Google app engine. `http://code.google.com/intl/cs/appengine/`, 2009.

[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, November 2006.

[3] The Free Encyclopedia Wikipedia. Paxos algorithm. `http://en.wikipedia.org/wiki/Paxos_algorithm`, 2009.