Lab 2: Google app engine with python

In this tutorial we will learn the following:

1. Creating and hosting a website in google app engine
2. Creating a guestbook application with google datastore and deploying in Google App Engine

Download the Hello World app

We've created a simple Hello World app for Python so you can quickly get a feel for deploying an app to Google Cloud Platform. Follow these steps to download Hello World to your local machine.

1. Clone the Hello World sample app repository to your local machine:

```
git clone https://github.com/GoogleCloudPlatform/python-docs-samples
```

2. Then go to the directory that contains the sample code:

```
cd python-docs-samples/appengine/standard/hello_world
```

Alternatively, you can download the sample as a .zip file and extract it.

Test the application

Test the application using the local development server (`dev_appserver.py`), which is included with the SDK.

1. From within the `hello_world` directory where the app's app.yaml configuration file is located, start the local development server with the following command:

```
dev_appserver.py app.yaml
```

The local development server is now running and listening for requests on port 8080. Something go wrong?

2. Visit http://localhost:8080/ in your web browser to view the app.

Make a change

You can leave the development server running while you develop your application. The development server watches for changes in your source files and reloads them if necessary.

1. Try it now: Leave the development server running, then edit `main.py` to change `Hello, World!` to something else.
2. Reload http://localhost:8080/ to see the results.

Deploy your app

To deploy your app to App Engine, run the following command from within the root directory of your application where the `app.yaml` file is located:

```
gcloud app deploy
```

Learn about the optional flags.

View your application

To launch your browser and view the app at `http://[YOUR_PROJECT_ID].appspot.com`, run the following command:

```
gcloud app browse
```

Creating a website to host on Google App Engine

Basic structure for the project

This guide uses the following structure for the project:

- `app.yaml`: Configure the settings of your App Engine application.
- `www/`: Directory to store all of your static files, such as HTML, CSS, images, and JavaScript.
- `css/`: Directory to store stylesheets.

- `style.css`: Basic stylesheet that formats the look and feel of your site.
- `images/`: Optional directory to store images.
- `index.html`: An HTML file that displays content for your website.
- `js/`: Optional directory to store JavaScript files.
- Other asset directories.

Creating the `app.yaml` file

The `app.yaml` file is a configuration file that tells App Engine how to map URLs to your static files. In the following steps, you will add handlers that will load `www/index.html` when someone visits your website, and all static files will be stored in and called from the `www` directory.

Create the `app.yaml` file in your application's root directory:

1. Create a directory that has the same name as your project ID. You can find your project ID in the Console.
2. In directory that you just created, create a file named `app.yaml`.
3. Edit the `app.yaml` file and add the following code to the file:

```
runtime: python27
api_version: 1
threadsafe: true

handlers:
- url: /
  static_files: www/index.html
  upload: www/index.html

- url: /(.*)
  static_files: www/\1
  upload: www/(.*)
```

More reference information about the `app.yaml` file can be found in the `app.yaml` reference documentation.

Creating the `index.html` file

Create an HTML file that will be served when someone navigates to the root page of your website. Store this file in your `www` directory.

```
<html>
  <head>
    <title>Hello, world!</title>
    <link rel="stylesheet" type="text/css" href="/css/style.css">
  </head>
  <body>
    <h1>Hello, world!</h1>
    <p>
      This is a simple static HTML file that will be served from Google
App
      Engine.
    </p>
  </body>
</html>
```

Deploying your application to App Engine

When you deploy your application files, your website will be uploaded to App Engine. To deploy your app, run the following command from within the root directory of your application where the `app.yaml` file is located:

```
gcloud app deploy
```

Optional flags:

- Include the `--project` flag to specify an alternate Cloud Platform Console project ID to what you initialized as the default in the `gcloud` tool. Example: `--project [YOUR_PROJECT_ID]`
- Include the `-v` flag to specify a version ID, otherwise one is generated for you. Example: `-v [YOUR_VERSION_ID]`

To learn more about deploying your app from the command line, see Deploying a Python App.

Viewing your application

To launch your browser and view the app at `http://[YOUR_PROJECT_ID].appspot.com`, run the following command:

```
gcloud app browse
```

Creating a Guestbook Application

This tutorial shows you how to build and run a sample Python application for App Engine and provides a code walkthrough of the sample code. The sample is a simple guestbook that lets users post messages to a public message board.

Objectives

- Build and test an App Engine app using Python.
- Integrate your application with Google Accounts for user authentication.
- Use the webapp2 framework.
- Use Jinja2 templates.
- Store data in Google Cloud Datastore.
- Deploy your app to App Engine.

Cloning the project from GitHub

1. Clone the Guestbook application repository to your local machine:

   ```
   git clone https://github.com/GoogleCloudPlatform/appengine-guestbook-python.git
   ```

2. Go to the directory that contains the sample code:

   ```
   cd appengine-guestbook-python
   ```

Authenticating Users

This part of the Python Guestbook code walkthrough shows how to authenticate users and display a customized greeting for the signed-in user.

This page is part of a multi-page tutorial. To start from the beginning and see instructions for setting up, go to Creating a Guestbook.

Signing in users

The `MainPage` class defines a handler for HTTP `GET` requests to the root path `'/'`. The handler checks to see whether a user is signed in:

guestbook.py

```
class MainPage(webapp2.RequestHandler):

    def get(self):
        guestbook_name = self.request.get('guestbook_name',
                                          DEFAULT_GUESTBOOK_NAME)
        greetings_query = Greeting.query(
            ancestor=guestbook_key(guestbook_name)).order(-Greeting.date)
        greetings = greetings_query.fetch(10)

        user = users.get_current_user()
        if user:
            url = users.create_logout_url(self.request.uri)
            url_linktext = 'Logout'
        else:
            url = users.create_login_url(self.request.uri)
            url_linktext = 'Login'

        template_values = {
            'user': user,
            'greetings': greetings,
            'guestbook_name': urllib.quote_plus(guestbook_name),
            'url': url,
            'url_linktext': url_linktext,
        }

        template = JINJA_ENVIRONMENT.get_template('index.html')
        self.response.write(template.render(template_values))
```

If the user is already signed in to your application, the `get_current_user()` method returns a `User` object, and the app displays the users's nickname. If the user has not signed in, the code redirects the user's browser to the Google account sign-in screen. The Google account sign-in mechanism sends the user back to the app after the user has signed in.

Handling User Input in a Form

This part of the Python Guestbook code walkthrough shows how to handle user input.

This page is part of a multi-page tutorial. To start from the beginning and see instructions for setting up, go to Creating a Guestbook.

Configuring the app to use webapp2

The Guestbook sample uses the webapp2 framework, which is included in the App Engine environment and the App Engine Python SDK. You don't need to bundle webapp2 with your application code to use it.

The `app.yaml` file specifies that the app uses the `webapp2` framework:

app.yaml

```
- name: webapp2
  version: latest
- name: jinja2
  version: latest
```

A webapp2 application has two parts:

- One or more `RequestHandler` classes that process requests and build responses.
- A `WSGIApplication` instance that routes incoming requests to handlers based on the URL.

The `app.yaml` file specifies the `app` object in `guestbook.py` as the handler for all URLs:

app.yaml

```
handlers:
- url: /favicon\.ico
  static_files: favicon.ico
  upload: favicon\.ico

- url: /bootstrap
  static_dir: bootstrap
```

```
- url: /.*
  script: guestbook.app
```

Defining a handler for form submission

The `app` object in `guestbook.py` is a `WSGIApplication` that defines which scripts handle requests for given URLs.

guestbook.py

```
app = webapp2.WSGIApplication([
    ('/', MainPage),
    ('/sign', Guestbook),
], debug=True)
```

The `debug=True` parameter tells `webapp2` to print stack traces to the browser output if a handler encounters an error or raises an uncaught exception. This option should be removed before deploying the final version of your application, otherwise you will inadvertently expose the internals of your application.

The `Guestbook` handler has a `post()` method instead of a `get()` method. This is because the form displayed by `MainPage` uses the `HTTP POST` method to submit the form data.

guestbook.py

```
class Guestbook(webapp2.RequestHandler):

    def post(self):
        # We set the same parent key on the 'Greeting' to ensure each
        # Greeting is in the same entity group. Queries across the
        # single entity group will be consistent. However, the write
        # rate to a single entity group should be limited to
        # ~1/second.
        guestbook_name = self.request.get('guestbook_name',
                                          DEFAULT_GUESTBOOK_NAME)
        greeting = Greeting(parent=guestbook_key(guestbook_name))

        if users.get_current_user():
            greeting.author = Author(
                    identity=users.get_current_user().user_id(),
                    email=users.get_current_user().email())
```

```
        greeting.content = self.request.get('content')
        greeting.put()

        query_params = {'guestbook_name': guestbook_name}
        self.redirect('/?' + urllib.urlencode(query_params))
```

The `post()` method gets the form data from `self.request`.

Generating Dynamic Content from Templates

This part of the Python Guestbook code walkthrough shows how to use Jinja templates to generate dynamic web content.

This page is part of a multi-page tutorial. To start from the beginning and see instructions for setting up, go to Creating a Guestbook.

HTML embedded in code is messy and difficult to maintain. It's better to use a templating system, where the HTML is kept in a separate file with special syntax to indicate where the data from the application appears. There are many templating systems for Python: EZT, Cheetah, ClearSilver, Quixote, Django, and Jinja2 are just a few. You can use your template engine of choice by bundling it with your application code.

For your convenience, App Engine includes the Django and Jinja2 templating engines.

Using Jinja2 Templates

The `app.yaml` file lists the latest version of `jinja2` as a required library. Production applications should use an actual version number rather than `version: latest`.

app.yaml

```
libraries:
- name: webapp2
  version: latest
- name: jinja2
  version: latest
```

The app imports `jinja2` and creates a `jinja2.Environment` object.

guestbook.py

```
import os
import urllib
```

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

import jinja2
import webapp2

JINJA_ENVIRONMENT = jinja2.Environment(
    loader=jinja2.FileSystemLoader(os.path.dirname(__file__)),
    extensions=['jinja2.ext.autoescape'],
    autoescape=True)
```

The `get` method for the `MainPage` request handler forms a dictionary of key/value pairs and passes it to `template.render`.

guestbook.py

```
class MainPage(webapp2.RequestHandler):

    def get(self):
        guestbook_name = self.request.get('guestbook_name',
                                          DEFAULT_GUESTBOOK_NAME)
        greetings_query = Greeting.query(
            ancestor=guestbook_key(guestbook_name)).order(-Greeting.date)
        greetings = greetings_query.fetch(10)

        user = users.get_current_user()
        if user:
            url = users.create_logout_url(self.request.uri)
            url_linktext = 'Logout'
        else:
            url = users.create_login_url(self.request.uri)
            url_linktext = 'Login'

        template_values = {
            'user': user,
            'greetings': greetings,
            'guestbook_name': urllib.quote_plus(guestbook_name),
            'url': url,
            'url_linktext': url_linktext,
        }

        template = JINJA_ENVIRONMENT.get_template('index.html')
        self.response.write(template.render(template_values))
```

The page is rendered according to the `index.html` template, which receives the dictionary as input.

index.html

```
{% for greeting in greetings %}
<div class="row">
  {% if greeting.author %}
    <b>{{ greeting.author.email }}
      {% if user and user.user_id() == greeting.author.identity %}
        (You)
      {% endif %}
    </b> wrote:
  {% else %}
    An anonymous person wrote:
  {% endif %}
  <blockquote>{{ greeting.content }}</blockquote>
</div>
{% endfor %}
```

The `JINJA_ENVIRONMENT.get_template(name)` method takes the name of a template file and returns a template object. The `template.render(template_values)` call takes a dictionary of values, and returns the rendered text. The template uses Jinja2 templating syntax to access and iterate over the values, and can refer to properties of those values.

Storing Data in Cloud Datastore

This part of the Python Guestbook code walkthrough shows how to store structured data in Google Cloud Datastore. With App Engine and Cloud Datastore, you don't have to worry about distribution, replication, and load balancing of data. That is done for you behind a simple API—and you get a powerful query engine and transactions as well.

Note: Cloud Datastore is only one option you have for storing application data. If you prefer to use relational data and SQL instead of data structures, try walking through the instructions for using Google Cloud SQL, which also use the Guestbook example application.

This page is part of a multi-page tutorial. To start from the beginning and see instructions for setting up, go to Creating a Guestbook.

Storing the submitted greetings

Data is written to Cloud Datastore in objects known as *entities*. Each entity has a *key* that uniquely identifies it. An entity can optionally designate another entity as its *parent*, the

first entity is a *child* of the parent entity. The entities in the data store thus form a hierarchically-structured space similar to the directory structure of a file system. For detailed information, see Structuring Data for Strong Consistency.

App Engine includes a data modeling API for Python. To use the data modeling API, the sample app imports the `google.appengine.ext.ndb` module. Each greeting includes the author's name, the message content, and the date and time the message was posted. The app displays messages in chronological order. The following code defines the data model:

guestbook.py

```
class Author(ndb.Model):
    """Sub model for representing an author."""
    identity = ndb.StringProperty(indexed=False)
    email = ndb.StringProperty(indexed=False)


class Greeting(ndb.Model):
    """A main model for representing an individual Guestbook entry."""
    author = ndb.StructuredProperty(Author)
    content = ndb.StringProperty(indexed=False)
    date = ndb.DateTimeProperty(auto_now_add=True)
```

The code defines a `Greeting` model with three properties: `author` whose value is an `Author` object with the email address and the author's identity, `content` whose value is a string, and `date` whose value is a `datetime.datetime`.

Some property constructors take parameters to further configure their behavior. Passing the `ndb.StringProperty` constructor the `indexed=False` parameter says that values for this property will not be indexed. This saves writes which aren't needed because the app never uses that property in a query. Passing the `ndb.DateTimeProperty` constructor an `auto_now_add=True` parameter configures the model to automatically give new objects a `datetime` stamp of the time the object is created, if the application doesn't otherwise provide a value. For a complete list of property types and their options, see NDB Properties.

The application uses the data model to create new `Greeting` objects and put them into Cloud Datastore. The `Guestbook` handler creates new greetings and saves them to the data store:

guestbook.py

```
class Guestbook(webapp2.RequestHandler):

    def post(self):
```

```
        # We set the same parent key on the 'Greeting' to ensure each
        # Greeting is in the same entity group. Queries across the
        # single entity group will be consistent. However, the write
        # rate to a single entity group should be limited to
        # ~1/second.
        guestbook_name = self.request.get('guestbook_name',
                                          DEFAULT_GUESTBOOK_NAME)
        greeting = Greeting(parent=guestbook_key(guestbook_name))

        if users.get_current_user():
            greeting.author = Author(
                    identity=users.get_current_user().user_id(),
                    email=users.get_current_user().email())

        greeting.content = self.request.get('content')
        greeting.put()

        query_params = {'guestbook_name': guestbook_name}
        self.redirect('/?' + urllib.urlencode(query_params))
```

This `Guestbook` handler creates a new `Greeting` object, then sets its `author` and `content` properties with the data posted by the user. The parent of `Greeting` is a `Guestbook` entity. There's no need to create the `Guestbook` entity before setting it to be the parent of another entity. In this example, the parent is used as a placeholder for transaction and consistency purposes. See the Transactions page for more information. Objects that share a common ancestor belong to the same entity group. The code does not set the `date` property, so `date` is automatically set to the present, using`auto_now_add=True`.

Finally, `greeting.put()` saves the new object to the data store. If we had acquired this object from a query, `put()` would have updated the existing object. Because we created this object with the model constructor, `put()` adds the new object to the data store.

Because querying in Cloud Datastore is strongly consistent only within entity groups, the code assigns all of one book's greetings to the same entity group by setting the same parent for each greeting. This means the user always sees a greeting immediately after it is written. However, the rate at which you can write to the same entity group is limited to one write to the entity group per second. When you design a real application you'll need to keep this fact in mind. Note that by using services such as Memcache, you can lower the chance that a user sees stale results when querying across entity groups after a write.

Retrieving submitted greetings

Cloud Datastore has a sophisticated query engine for data models. Because Cloud Datastore is not a traditional relational database, queries are not specified using SQL. Instead, data is queried one of two ways: either by usingDatastore queries, or by using an SQL-like query language called GQL. To access the full range of Cloud Datastore's query capabilities, we recommend using queries over GQL.

The `MainPage` handler retrieves and displays previously submitted greetings. The `greetings_query.fetch(10)` call performs the query.

More about Cloud Datastore indexes

Every query in Cloud Datastore is computed from one or more *indexes*—tables that map ordered property values to entity keys. This is how App Engine is able to serve results quickly regardless of the size of your application's data store. Many queries can be computed from the built-in indexes, but for queries that are more complex, Cloud Datastore requires a*custom index*. Without a custom index, Cloud Datastore can't execute these queries efficiently.

For example, the Guestbook application filters by guestbook, and orders by date, using an ancestor query and a sort order. This requires a custom index to be specified in the application's `index.yaml` file. You can edit this file manually, or you can take care of it automatically by running the queries in the application locally. After the index is defined in `index.yaml`, deploying the application will also deploy the custom index information.

The definition for the query in `index.yaml` looks like this:

index.yaml

VIEW ON GITHUB

```
indexes:
- kind: Greeting
  ancestor: yes
  properties:
  - name: date
    direction: desc
```

You can read all about Cloud Datastore indexes in the Datastore Indexes page. You can read about the proper specification for `index.yaml` files in Python Datastore Index Configuration.

Serving Static Files

This part of the Python Guestbook code walkthrough shows how to serve static files. App Engine does not serve files directly out of your application's source directory unless configured to do so. But there are many cases where you want to serve static files directly to the web browser. Images, CSS stylesheets, JavaScript code, movies, and Flash animations are all typically stored with a web application and served directly to the browser.

This page is part of a multi-page tutorial. To start from the beginning and see instructions for setting up, go to Creating a Guestbook.

Configuring the app to use static files

The CSS files for the Guestbook app are in the `bootstrap/css` directory. The template for the app's web page, `index.html`, instructs the browser to load `bootstrap.css` and `bootstrap-responsive.css`, which are static files:

index.html

```
<link type="text/css" rel="stylesheet"
href="/bootstrap/css/bootstrap.css">
<link type="text/css" rel="stylesheet" href="/bootstrap/css/bootstrap-
responsive.css">
```

The `app.yaml` file specifies the `bootstrap` directory as the location for static files:

app.yaml

```
handlers:
- url: /favicon\.ico
  static_files: favicon.ico
  upload: favicon\.ico

- url: /bootstrap
  static_dir: bootstrap

- url: /.*
  script: guestbook.app
```

The `handlers` section defines two handlers for URLs. When App Engine receives a request for a URL beginning with `/bootstrap`, it maps the remainder of the path to files in the `bootstrap` directory, and if an appropriate file is found, the contents of the file are

returned to the client. All other URLs match the `/.*` pattern, and are handled by the `app`object in the `guestbook` module.

URL path patterns are tested in the order they appear in `app.yaml`. In this case, the `/bootstrap` pattern matches before the `/.*` pattern for the appropriate paths. For more information on URL mapping and other options you can specify in `app.yaml`, see the `app.yaml` reference.

Deploying the Application

This part of the Python Guestbook code walkthrough shows how to deploy the application to App Engine.

This page is part of a multi-page tutorial. To start from the beginning and see instructions for setting up, go to Creating a Guestbook.

Deploying the app to App Engine

To upload the guestbook app, run the following command from within the `appengine-guestbook-python` directory of your application where the `app.yaml` and `index.yaml` files are located:

```
gcloud app deploy app.yaml index.yaml
```

Optional flags:

- Include the `--project` flag to specify an alternate Cloud Platform Console project ID to what you initialized as the default in the `gcloud` tool. Example: `--project [YOUR_PROJECT_ID]`
- Include the `-v` flag to specify a version ID, otherwise one is generated for you. Example: `-v [YOUR_VERSION_ID]`

The Datastore indexes might take some time to generate before your application is available. If the indexes are still in the process of being generated, you will receive a `NeedIndexError` message when accessing your app. This is a transient error, so try a little later if at first you receive this error.

To learn more about deploying your app from the command line, see Deploying a Python App.

Viewing your deployed application

To launch your browser and view the app at `http://[YOUR_PROJECT_ID].appspot.com`, or run the following command:

```
gcloud app browse
```