CLOUD PROGRAMMING BRIDGE COURSE MINI PROJECT.

Title: Deploy Online Task Manager with the help of Google App Engine.

INDEX

1. INTRODUCTION:

Google App Engine is a Platform as a Service (PaaS) offering that lets you build and run applications on Google's infrastructure. App Engine applications are easy to build, easy to maintain, and easy to scale as your traffic and data storage needs change. With App Engine, there are no servers for you to maintain. You simply upload your application and it's ready to go.

The App Engine runtime environment:

Google App Engine supports apps written in a variety of programming languages.

- **Java:** Using App Engine's Java runtime environment, you can build your application using standard Java technologies.
- **Python:** App Engine features a fast Python interpreter and standard Python libraries.
- **PHP:** App Engine uses Google's Cloud Platform services under the hood when you call standard PHP functions.
- **Go:** App Engine features a Go runtime environment that runs natively compiled Go code.

Google App Engine makes it easy to build and deploy an application that runs reliably even under heavy load and with large amounts of data. It includes the following features:

- Persistent storage with queries, sorting, and transactions.
- Automatic scaling and load balancing.
- Asynchronous task queues for performing work outside the scope of a request.
- Scheduled tasks for triggering events at specified times or regular intervals.
- Integration with other **Google cloud services and APIs.**

Applications run in a secure, sandboxed environment, allowing App Engine to distribute requests across multiple servers, and scaling servers to meet traffic demands. Your application runs within its own secure, reliable environment that is independent of the hardware, operating system, or physical location of the server. .

The SDK manages your application locally, while the Google Developers Console manages your application in production. The Developers Console uses a web-based interface to create new applications, configure domain names, change which version of your application is live, examine access and error logs, and much more.

2. PRE-REQUISITE FOR THE PROJECT:

- You should be fairly comfortable programming in Python, preferably with some experience developing web applications and working with databases for at least a year.
- You should have sufficient permissions to install new software on your computer, and comfortable configuring it, including setting up system variables.
- IDLE for Python. (Version 2.7)
- A Google App Engine account to (to test the app in GAE).
- Google App Engine Python Runtime Environment.

3. SCOPE OF THE PROJECT:

Our project is an Online Task Manager which helps you create, delete, modify any number of tasks and maintain an efficient task list. Tasks can be defined for oneself or assigned to a colleague, or delegated after being received.

Any project can have a dedicated group of people who are working on a particular project together. As a result they need to maintain proper record and documentation of each task that every group member has to perform in order to avoid replication as well as prevent redundancy of efforts.

The time of each task is noted along with the task name. As a result different tasks can be assigned different priorities based on how long the tasks have been pending. The highest priority tasks need to be completed first while the lower priority tasks can be handled later.

This gives a full picture of the progression of the project at a single glance, and helps identify potential or existing delays. This helps in ensuring that a task is completed in time and all the deadlines are met.

Group members can add different tasks to the checklist and mark their completion. As a result a quick glance is all that is needed to see the exact status of the project. Once a task is completed it can be deleted and removed from the list.

In order to maintain privacy and security so as to ensure no external personnel are able to modify the task list, the user needs to sign in using his Google ID. Once the user has signed in, he will be able to view the task list and add new tasks or delete old tasks.

This feature can be extended to a group project by creating a common group id for all members. This would help to ensure that the task list is available to all the members as and when required.

4. METHEDOLOGY:

Google App Engine: Platform as a Service

Google App Engine lets you build and run applications on Google's infrastructure. App Engine applications are easy to create, easy to maintain, and easy to scale as your traffic and data storage needs change. With App Engine, there are no servers for you to maintain. You simply upload your application and it's ready to go.

App Engine executes your Python application code using a pre-loaded Python interpreter in a safe "sandboxed" environment. Your app receives web requests, performs work, and sends responses by interacting with this environment.

A Python web app interacts with the App Engine web server using the WSGI protocol, so apps can use any WSGI-compatible web application framework. App Engine includes a simple web application framework, called webapp2, to make it easy to get started. For larger applications, mature third-party frameworks, such as Django, work well with App Engine.

The Python interpreter can run any Python code, including Python modules you include with your application, as well as the Python standard library. The interpreter cannot load Python modules with C code; it is a "pure" Python environment.

The secured "sandbox" environment isolates your application for service and security. It ensures that apps can only perform actions that do not interfere with the performance and scalability of other apps. For instance, an app cannot write data to the local file system or make arbitrary network connections. Instead, apps use scalable services provided by App Engine to store data and communicate over the Internet. The Python interpreter raises an exception when an app attempts to import a module from the standard library known to not work within the sandbox restrictions.

The App Engine platform provides many services that your code can call. Your application can also configure scheduled tasks that run at specified intervals.

Python

All code for the Python runtime environment must be pure Python, and not include any C extensions or other code that must be compiled.

The environment includes the Python standard library. Some modules have been disabled because their core functions are not supported by App Engine, such as networking or writing to the file system. In addition, the os module is available, but with unsupported features disabled. An attempt to import an unsupported module or use an unsupported feature will raise an exception.

A few modules from the standard library have been replaced or customized to work with App Engine. These modules vary between the two Python runtimes, as described below.

Customized Libraries in Python 2.7

In the Python 2.7 runtime, the following modules have been replaced or customized:

- tempfile is disabled, except for TemporaryFile which is aliased to StringIO.
- logging is available and its use is highly encouraged! See Logging.

In addition to the Python standard library and the App Engine libraries, the Python 2.7 runtime includes several third-party libraries.

There a couple of important files and in the war directory, in particular the WEB-INF directory includes: *appengine-web.xml* and *web.xml*

"The web.xml is also used to define context variables which can be referenced within the servlets and it is used to define environmental dependencies which the deployer is expected to set up".

The appengine-web.xml "includes the registered ID of your application (Eclipse creates this with an empty ID for you to fill in later), the version number of your application, and lists of files that ought to be treated as static files (such as images and CSS) and resource files (such as JSPs and other application data)." (GAE document)web.xml declares servlets, servlet mappings (URLs to servlet), filters, etc.

Workflow:

Let's think about the workflow of the browser and the servlets -- recall there are 4 servlets, one for each query type. If you inspect the code then the query workflow is as follows:

- the user loads the URL localhost:8888 into the browser. The browser sends a GET request to the servlet /query.do because it is defined as the welcome file in web.xml;
- the request invokes the servlet on the server;
- the servlet parses the request, gets a parameter called "name", which is the query condition;
- the servlet retrieves data from the Datastore with the condition.
- if the condition is empty, it retrieves all the records from the Datastore, otherwise it only retrieves the record whose name is the same as the condition. The results are stored in an ArrayList;
- the servlet adds the ArrayList to the request object then redirect to the query_result.jsp;
- the query_result.jsp gets the ArrayList from the request object, then generates the html with all the data in ArrayList; and finally,
- the server sends the generated html back to user's browser which displays the html

Data store queries have the following limitations:

- Entities lacking a property name in the query are ignored.
- Filtering on un indexed properties returns no results.
- Inequality filters are limited to at most one property.
- Ordering of query results is undefined when no sort order is specified.
- Properties used in inequality filters must be sorted first.
- Queries inside transactions must include ancestor filters.
- JOIN is not supported

5. Configuration of Google App Engine:

App Engine executes your Python application code using a pre-loaded Python interpreter in a safe "sandboxed" environment. Your app receives web requests, performs work, and sends responses by interacting with this environment. A Python web app interacts with the App Engine web server using the WSGI protocol, so apps can use any WSGI-compatible web application framework. App Engine includes a simple web application framework, called **webapp2**, to make it easy to get started. For larger applications, mature third-party frameworks, such as **Django**, work well with App Engine. The Python interpreter can run any Python code, including Python modules you include with your application, as well as the Python standard library. The interpreter cannot load Python modules with C code; it is a "pure" Python environment. For instance, an app cannot write data to the local file system or make arbitrary network connections. Instead, apps use scalable services provided by App Engine to store data and communicate over the Internet. The Python interpreter raises an exception when an app attempts to import a module from the standard library known to not work within the sandbox restrictions. The App Engine platform provides many **services** that your code can call. Your application can also configure **scheduled tasks** that run at specified intervals. If you haven't already, see **the Python 2.7 Getting Started Guide** for an interactive introduction to developing web applications with Python and Google App Engine.

Selecting the Python runtime

App Engine knows to use the Python runtime environment for your application code when you use the tool named appcfg.py from the Python SDK with a configuration file named app.yaml. You specify the runtime element in app.yaml. To use Python 2.7, add the following to app.yaml:

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

The first element, runtime, selects the Python runtime environment.

The second element, api_version, selects which version of the Python runtime environment to use. As of this writing, App Engine only has one version of the Python environment, 1. If the App Engine team ever needs to release changes to the environment that may not be compatible with existing code, they will do so with a new version identifier. Your app will continue to use the selected version until you change the api_version setting and upload your app.

6. Project Deployment:

Creating a Simple Request Handler

Create a directory named helloworld. All files for this application reside in this directory.

Inside the helloworld directory, create a file named helloworld.py, and give it the following contents:

helloworld.py

This Python script responds to a request with an HTTP header that describes the content and the message Hello, World!.

Creating the Configuration File

An App Engine application has a configuration file called app.yaml. Among other things, this file describes which handler scripts should be used for which URLs.

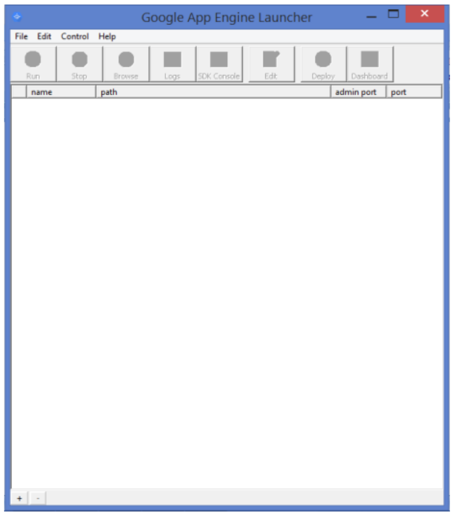
Inside the helloworld directory, create a file named app.yaml with the following contents:

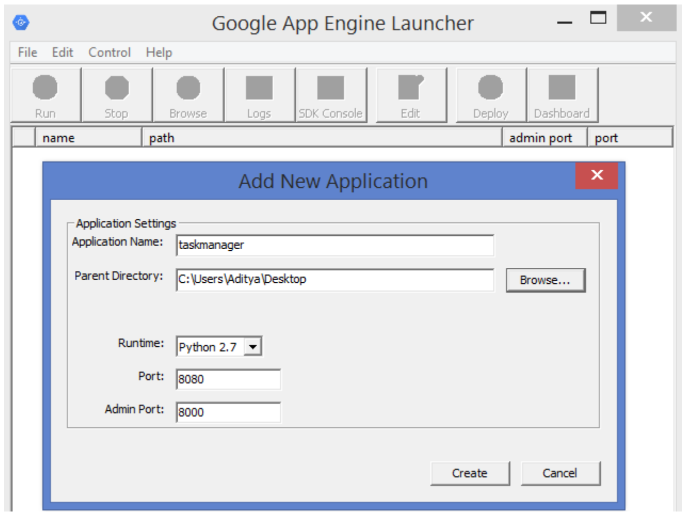
app.yaml

From top to bottom, this configuration file says the following about this application:

- This is version number 1 of this application's code. If you adjust this before uploading new versions of your application software, App Engine will retain previous versions, and let you roll back to a previous version using the administrative console.
- This code runs in the python27 runtime environment, API version 1. Additional runtime environments and languages may be supported in the future.
- This application is threadsafe so the same instance can handle several simultaneous requests. Threadsafe is an advanced feature and may result in erratic behavior if your application is not specifically designed to be threadsafe.
- Every request to a URL whose path matches the regular expression /.* (all URLs) should be handled by the app object in thehelloworld module.

Launching application on Local Host through Google App Engine


Google App Engine Launcher
File Edit Control Help
Run Stop Browse Logs SDK Console Edit Deploy Dashboard
name path admin port port


Google App Engine Launcher
File Edit Control Help
Run Stop Browse Logs SDK Console Edit Deploy Dashboard
name path admin port port
Add New Application
Browse...
Create
Cancel

Application Settings
Application Name: taskmanager
Parent Directory: C:\Users\Aditya\Desktop
Runtime: Python 2.7
Port: 8080
Admin Port: 8000

Testing the Application

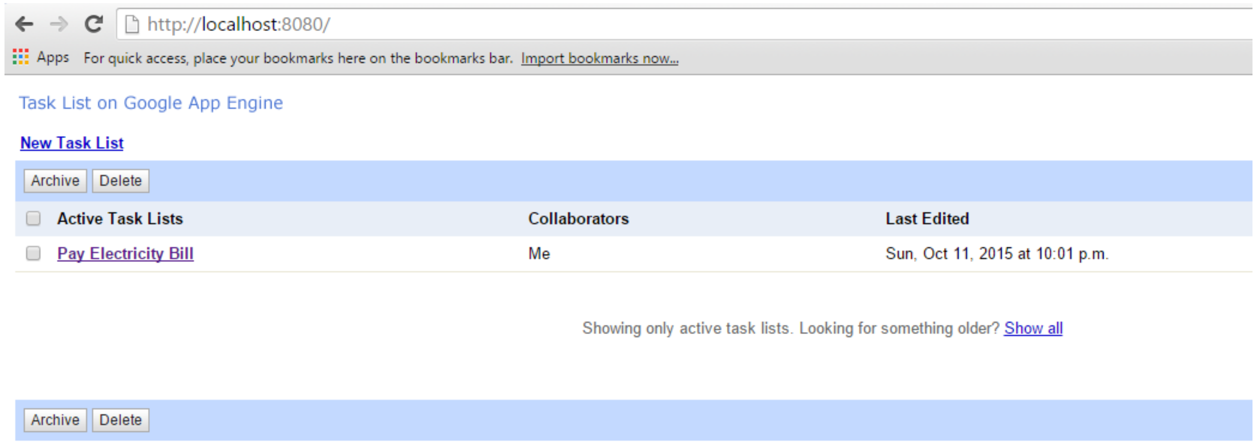
With a handler script and configuration file mapping every URL to the handler, the application is complete. You can now test it with the web server included with the App Engine Python SDK.

Start the web server with the following command, giving it the path to the helloworld directory:

```
$ <path-to-Python-SDK>/dev_appserver.py helloworld/
```

The web server is now running, listening for requests on port 8080. You can test the application by visiting the following URL in your web browser:

http://localhost:8080/


http://localhost:8080/
Apps
For quick access, place your bookmarks here on the bookmarks bar. Import bookmarks now...
Archive
Delete

Task List on Google App Engine

New Task List

	Active Task Lists	Collaborators	Last Edited
	Pay Electricity Bill	Me	Sun, Oct 11, 2015 at 10:01 p.m.

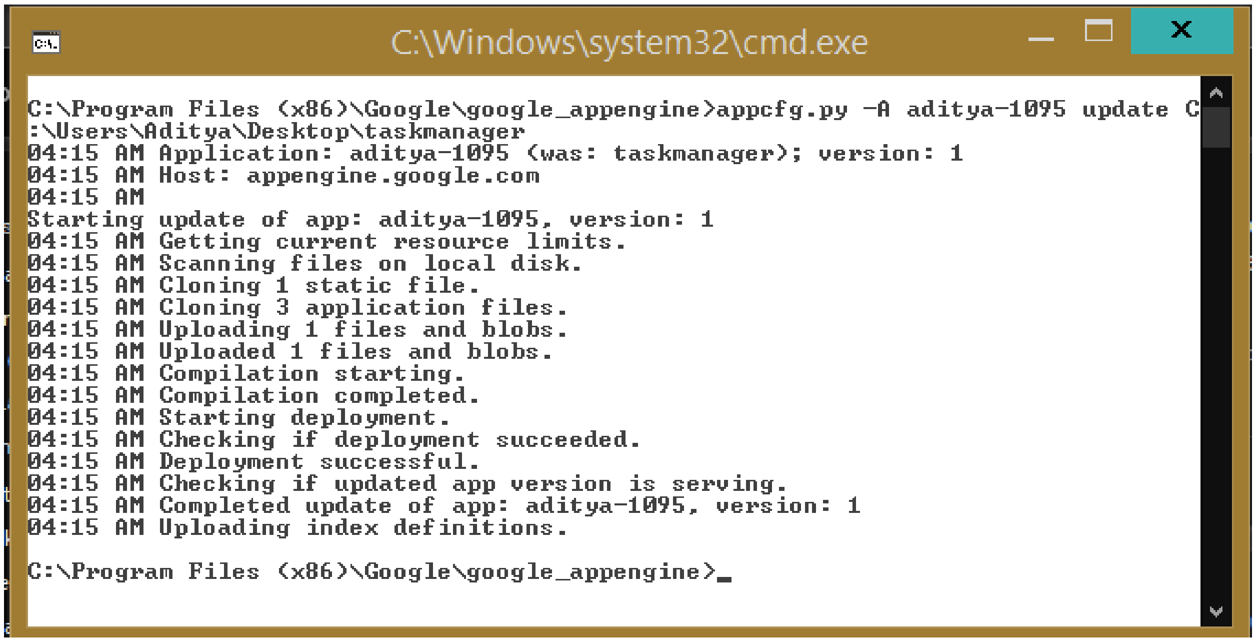
Showing only active task lists. Looking for something older? Show all

Uploading Your Application

To upload your application:

1. Sign in to App Engine using your Google account. If you do not have a Google account, you can **create a Google account** with an email address and password.
2. If you haven't already done so, create a project for your App Engine app as follows:
   a. Visit the **Google Developers Console** and click **Create Project**.
   b. Supply the desired project name in the New Project form. It doesn't have to match your app name, but using the same name as your app might make administration easier.
   c. Accept the generated project ID or supply your own ID. *This project ID is used as the App Engine application ID.* Note that this ID can only be used once: if you subsequently delete your project, you won't be able to re-use the ID in a new project.
3. Note the application ID (project ID) you created above.
4. Upload your finished application to Google App Engine by invoking the following command. This opens a browser window for you to sign in using your Google account. You'll be providing the project ID as the argument for -A.

```
appcfg.py -A <YOUR_PROJECT_ID_> update guestbook/
```



```
C:\Program Files (x86)\Google\google_appengine>appcfg.py -A aditya-1095 update C
:\Users\Aditya\Desktop\taskmanager
04:15 AM Application: aditya-1095 (was: taskmanager); version: 1
04:15 AM Host: appengine.google.com
04:15 AM
Starting update of app: aditya-1095, version: 1
04:15 AM Getting current resource limits.
04:15 AM Scanning files on local disk.
04:15 AM Cloning 1 static file.
04:15 AM Cloning 3 application files.
04:15 AM Uploading 1 files and blobs.
04:15 AM Uploaded 1 files and blobs.
04:15 AM Compilation starting.
04:15 AM Compilation completed.
04:15 AM Starting deployment.
04:15 AM Checking if deployment succeeded.
04:15 AM Deployment successful.
04:15 AM Checking if updated app version is serving.
04:15 AM Completed update of app: aditya-1095, version: 1
04:15 AM Uploading index definitions.

C:\Program Files (x86)\Google\google_appengine>_
```

5. The Datastore Indexes may take some time to generate before your application is available. You will receive a NeedIndexErrorwhen accessing your app if the indexes are still in the process of being generated. This is a transient error for the example, so try a little later if at first you receive this exception.
6. Your app is now deployed and ready for users!

CODE:

```
"""A collaborative task list
web application built on
Google App Engine."""
import datetime
import os
import random
import string
import sys
import wsgiref.handlers

from google.appengine.api
import users
from google.appengine.ext
import db
from google.appengine.ext
import webapp
from
google.appengine.ext.webap
p import template
from
google.appengine.ext.webap
p.util import login_required

# Set to true if we want to
have our webapp print stack
traces, etc
_DEBUG = True

# Add our custom Django
template filters to the built in
filters
template.register_template_l
ibrary('templatefilters')

class TaskList(db.Model):
  """A TaskList is the entity
tasks refer to to form a list.

  Other than the tasks
referring to it, a TaskList just
has meta-data, like
  whether it is published and
the date at which it was last
updated.
  """
  name =
db.StringProperty(required=
True)
  created =
db.DateTimeProperty(auto_n
ow_add=True)
  updated =
db.DateTimeProperty(auto_n
ow=True)
  archived =
db.BooleanProperty(default=
False)
  published =
db.BooleanProperty(default=
False)

  @staticmethod
  def
get_current_user_lists():
    """Returns the task lists
that the current user has
access to."""
    return
TaskList.get_user_lists(users.
GetCurrentUser())

  @staticmethod
  def get_user_lists(user):
    """Returns the task lists
that the given user has
access to."""
    if not user: return []
    memberships =
db.Query(TaskListMember).fi
lter('user =', user)
    return [m.task_list for m in
memberships]

  def
current_user_has_access(self
):
    """Returns true if the
current user has access to
this task list."""
    return
self.user_has_access(users.G
etCurrentUser())
  def user_has_access(self,
user):
    """Returns true if the given
user has access to this task
list."""
    if not user: return False
    query =
db.Query(TaskListMember)
    query.filter('task_list =',
self)
    query.filter('user =', user)
    return query.get()


class
TaskListMember(db.Model):
  """Represents the many-to-
many relationship between
TaskLists and Users.

  This is essentially the task
list ACL.
  """
  task_list =
db.Reference(TaskList,
required=True)
  user =
db.UserProperty(required=Tr
ue)


class Task(db.Model):
  """Represents a single task
in a task list.

  A task basically only has a
description. We use the
priority field to
  order the tasks so that users
can specify task order
manually.

  The completed field is a
DateTime, not a bool; if it is
not None, the
```

```
  task is completed, and the
timestamp represents the
time at which it was
  marked completed.
  """
  description =
db.TextProperty(required=Tr
ue)
  completed =
db.DateTimeProperty()
  archived =
db.BooleanProperty(default=
False)
  priority =
db.IntegerProperty(required
=True, default=0)
  task_list =
db.Reference(TaskList)
  created =
db.DateTimeProperty(auto_n
ow_add=True)
  updated =
db.DateTimeProperty(auto_n
ow=True)



class
BaseRequestHandler(webap
p.RequestHandler):
  """Supplies a common
template generation
function.

  When you call generate(),
we augment the template
variables supplied with
  the current user in the
'user' variable and the
current webapp request
  in the 'request' variable.
  """
  def generate(self,
template_name,
template_values={}):
    values = {
      'request': self.request,
      'user':
users.GetCurrentUser(),
      'login_url':
users.CreateLoginURL(self.re
quest.uri),
      'logout_url':
users.CreateLogoutURL('http
://' + self.request.host + '/'),
      'debug':
self.request.get('deb'),
      'application_name': 'Task
Manager',
    }

values.update(template_valu
es)
    directory =
os.path.dirname(__file__)
    path =
os.path.join(directory,
os.path.join('templates',
template_name))

self.response.out.write(temp
late.render(path, values,
debug=_DEBUG))



class
InboxPage(BaseRequestHand
ler):
  """Lists the task list "inbox"
for the current user."""
  @login_required
  def get(self):
    lists =
TaskList.get_current_user_lis
ts()
    show_archive =
self.request.get('archive')
    if not show_archive:
      non_archived = []
      for task_list in lists:
        if not task_list.archived:

non_archived.append(task_li
st)
      lists = non_archived
    self.generate('index.html',
{
      'lists': lists,
      'archive': show_archive,
    })
class
TaskListPage(BaseRequestHa
ndler):
  """Displays a single task list
based on ID.

  If the task list is not
published, we give a 403
unless the user is a
  collaborator on the list. If it
is published, but the user is
not a
  collaborator, we show the
more limited HTML view of
the task list rather
  than the interactive AJAXy
edit page.
  """

  # The different task list
output types we support:
content types and
  # template file extensions
  _OUTPUT_TYPES = {
    'default': ['text/html',
'html'],
    'html': ['text/html', 'html'],
    'atom':
['application/atom+xml',
'xml'],
  }

  def get(self):
    task_list =
TaskList.get(self.request.get('
id'))
    if not task_list:
      self.error(403)
      return

    # Choose a template based
on the output type
    output_name =
self.request.get('output')
    output_name_list =
TaskListPage._OUTPUT_TYPE
S.keys()
    if output_name not in
output_name_list:
      output_name =
output_name_list[0]
```

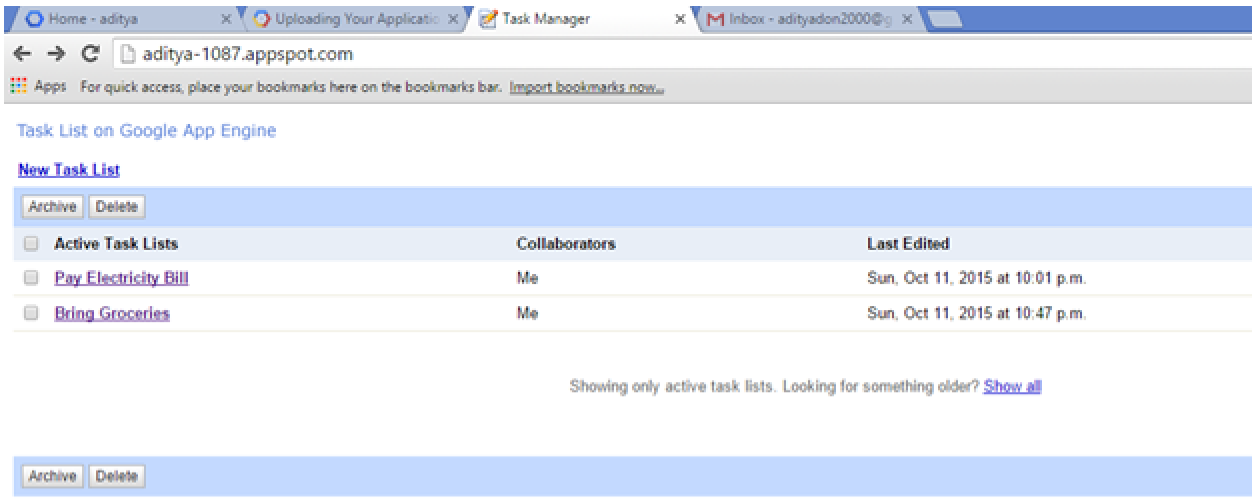
```
    output_type =
TaskListPage._OUTPUT_TYPE
S[output_name]

    # Validate this user has
access to this task list. If not,
they can
    # access the html view of
this list only if it is published.
    if not
task_list.current_user_has_a
ccess():
      if task_list.published:
        if output_name ==
'default':
          output_name = 'html'
          output_type =
TaskListPage._OUTPUT_TYPE
S[output_name]
      else:
        user =
users.GetCurrentUser()
        if not user:

self.redirect(users.CreateLogi
nURL(self.request.uri))
        else:
          self.error(403)
        return

    # Filter out archived tasks
by default
    show_archive =
self.request.get('archive')
    tasks =
task_list.task_set.order('-
priority').order('created')
    if not show_archive:
      tasks.filter('archived =',
False)
    tasks = list(tasks)

    # Get the last updated
date from the list of tasks
    if len(tasks) > 0:
      updated =
max([task.updated for task in
tasks])
    else:
      updated = None

self.response.headers['Conte
nt-Type'] = output_type[0]
    self.generate('tasklist_' +
output_name + '.' +
output_type[1], {
      'task_list': task_list,
      'tasks': tasks,
      'archive': show_archive,
      'updated': updated,
    })


class
CreateTaskListAction(BaseRe
questHandler):
  """Creates a new task list
for the current user."""
  def post(self):
    user =
users.GetCurrentUser()
    name =
self.request.get('name')
    if not user or not name:
      self.error(403)
      return

    task_list =
TaskList(name=name)
    task_list.put()
    task_list_member =
TaskListMember(task_list=ta
sk_list, user=user)
    task_list_member.put()

    if self.request.get('next'):

self.redirect(self.request.get(
'next'))
    else:
      self.redirect('/list?id=' +
str(task_list.key()))


class
EditTaskAction(BaseRequest
Handler):
  """Edits a specific task,
changing its description.

  We also updated the last
modified date of the task list
so that the
  task list inbox shows the
correct last modified date for
the list.

  This can be used in an AJAX
way or in a form. In a form,
you should
  supply a "next" argument
that denotes the URL we
should redirect to
  after the edit is complete.
  """
  def post(self):
    description =
self.request.get('description')
    if not description:
      self.error(403)
      return

    # Get the existing task that
we are editing
    task_key =
self.request.get('task')
    if task_key:
      task = Task.get(task_key)
      if not task:
        self.error(403)
        return
      task_list = task.task_list
    else:
      task = None
      task_list =
TaskList.get(self.request.get('
list'))

    # Validate this user has
access to this task list
    if not task_list or not
task_list.current_user_has_a
ccess():
      self.error(403)
      return

    # Create the task
    if task:
      task.description =
db.Text(description)
    else:
      task =
Task(description=db.Text(des
cription), task_list=task_list)
    task.put()

    # Update the task list so
it's updated date is updated.
Saving it is all
    # we need to do since that
field has auto_now=True
    task_list.put()

    # Only redirect if "next" is
given
    next =
self.request.get('next')
    if next:
      self.redirect(next)
    else:

self.response.headers['Conte
nt-Type'] = 'text/plain'

self.response.out.write(str(ta
sk.key()))



class
AddMemberAction(BaseReq
uestHandler):
  """Adds a new User to a
TaskList ACL."""
  def post(self):
    task_list =
TaskList.get(self.request.get('
list'))
    email =
self.request.get('email')
    if not task_list or not
email:
      self.error(403)
      return

    # Validate this user has
access to this task list
    if not
task_list.current_user_has_a
ccess():
      self.error(403)
      return

    # Don't duplicate entries in
the permissions datastore
    user = users.User(email)
    if not
task_list.user_has_access(us
er):
      member =
TaskListMember(user=user,
task_list=task_list)
      member.put()

self.redirect(self.request.get(
'next'))



class
InboxAction(BaseRequestHa
ndler):
  """Performs an action in the
user's TaskList inbox.

  We support Archive,
Unarchive, and Delete
actions. The action is
specified
  by the "action" argument in
the POST. The names are
capitalized because
  they correspond to the text
in the buttons in the form,
which all have the
  name "action".
  """
  def post(self):
    action =
self.request.get('action')
    lists = self.request.get('list',
allow_multiple=True)
    if not action in ['Archive',
'Unarchive', 'Delete']:
      self.error(403)
      return

    for key in lists:
      task_list =
TaskList.get(key)

      # Validate this user has
access to this task list
      if not task_list or not
task_list.current_user_has_a
ccess():
        self.error(403)
        return

      if action == 'Archive':
        task_list.archived = True
        task_list.put()
      elif action == 'Unarchive':
        task_list.archived = False
        task_list.put()
      else:
        for member in
task_list.tasklistmember_set:
          member.delete()
        for task in
task_list.task_set:
          task.delete()
        task_list.delete()


self.redirect(self.request.get(
'next'))



class
TaskListAction(BaseRequestH
andler):
  """Performs an action on a
specific task list.

  The actions we support are
"Archive Completed" and
"Delete", as specified
  by the "action" argument in
the POST.
  """
  def post(self):
    action =
self.request.get('action')
    tasks =
self.request.get('task',
allow_multiple=True)
    if not action in ['Archive
Completed', 'Delete']:
      self.error(403)
      return

    for key in tasks:
      task = Task.get(key)
      # Validate this user has
access to this task list
      if not task or not
task.task_list.current_user_h
as_access():
        self.error(403)
        return

      if action == 'Delete':
        task.delete()
      else:
        if task.completed and
not task.archived:
          task.priority = 0
          task.archived = True
          task.put()


self.redirect(self.request.get(
'next'))



class
SetTaskCompletedAction(Bas
eRequestHandler):
  """Sets a given task to be
completed at the current
time."""
  def post(self):
    task =
Task.get(self.request.get('id')
)
    if not task or not
task.task_list.current_user_h
as_access():
      self.error(403)
      return

    completed =
self.request.get('completed')
    if completed:
      task.completed =
datetime.datetime.now()
    else:
      task.completed = None
      task.archived = False
```

```
    task.put()


class
SetTaskPositionsAction(Base
RequestHandler):
  """Orders the tasks in a task
lists.

  The input to this handler is
a comma-separated list of
task keys in the
  "tasks" argument to the
post. We assign priorities to
the given tasks
  based on that order (e.g., 1
through N for N tasks).
  """
  def post(self):
    keys =
self.request.get("tasks").split
(",")
    if not keys:
      self.error(403)
      return
    num_keys = len(keys)
    for i, key in
enumerate(keys):
      key = keys[i]
      task = Task.get(key)
      if not task or not
task.task_list.current_user_h
as_access():
        self.error(403)
        return
      task.priority = num_keys -
i - 1
      task.put()


class
PublishTaskListAction(BaseR
equestHandler):
  """Publishes a given task
list, which makes it viewable
by everybody."""
  def post(self):
```

```
    task_list =
TaskList.get(self.request.get('
id'))
    if not task_list or not
task_list.current_user_has_a
ccess():
      self.error(403)
      return

    task_list.published =
bool(self.request.get('publish
'))
    task_list.put()


def main():
  application =
webapp.WSGIApplication([
    ('/', InboxPage),
    ('/list', TaskListPage),
    ('/edittask.do',
EditTaskAction),
    ('/createtasklist.do',
CreateTaskListAction),
    ('/addmember.do',
AddMemberAction),
    ('/inboxaction.do',
InboxAction),
    ('/tasklist.do',
TaskListAction),
    ('/publishtasklist.do',
PublishTaskListAction),
    ('/settaskcompleted.do',
SetTaskCompletedAction),
    ('/settaskpositions.do',
SetTaskPositionsAction),
  ], debug=_DEBUG)

wsgiref.handlers.CGIHandler(
).run(application)


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

Online Access

The full URL for your application is http://your-app-id.appspot.com/. Optionally, you can instead purchase and use a top-level domain name for your app, or use one that you have already registered.


Home - aditya
Uploading Your Applicatio
Task Manager
Inbox - adityadon2000@g
aditya-1087.appspot.com
Apps
For quick access, place your bookmarks here on the bookmarks bar. Import bookmarks now...
Archive
Delete

Task List on Google App Engine

New Task List

Active Task Lists	Collaborators	Last Edited
Pay Electricity Bill	Me	Sun, Oct 11, 2015 at 10:01 p.m.
Bring Groceries	Me	Sun, Oct 11, 2015 at 10:47 p.m.

Showing only active task lists. Looking for something older? Show all

7. Result:

On one hand Google App Engine provides a rich set of helpful features and services to help you build better applications. On the other hand you do need to compromise by working around certain issues because the Google App Engine doesn’t entirely behave as a regular application server. There is a little bit of work involved to create your first Google App Engine application, but after that you should be able to clone and add necessary dependencies if and where needed.

The main advantages of deploying an application on the Google App Engine is that it eliminates the need to buy servers or server space. As a result we do not have to worry about the security of those servers or their maintenance. It also makes solving the problem of scaling easier. It is free up to a certain level of consumed resources which is more than enough for the deployment of a basic application. Thus Google App Engine is the best way and the easiest way to deploy your application to the cloud.

8. CONCLUSION:

Google App Engine enables you to build web applications for your business leveraging Google’s infrastructure. App Engine applications are easy to develop, maintain, and can scale as your traffic and data storage needs grow. With App Engine, you don’t end up paying for large server spaces and then spend on resources maintaining them. You just upload your application, and it’s ready to serve to your users. Rest is taken care by Google Cloud.

9. References:

1. https://cloud.google.com/appengine/docs
2. https://cloud.google.com/appengine/docs/whatisgoogleappengine
3. https://cloud.google.com/appengine/docs/python/gettingstartedpython27/introduction
4. https://cloud.google.com/appengine/docs/python/
5. https://cloud.google.com/appengine/docs/python/gettingstartedpython27/helloworld
6. https://cloud.google.com/appengine/docs/python/gettingstartedpython27/uploading
7. http://stackoverflow.com/questions/1306279/pros-cons-of-google-app-engine
8. http://www.netsolutionsindia.com/blog/what-is-google-app-engine-its-advantages-and-how-it-can-benefit-your-business/