Building developers.google.com on Google App Engine

Dan Sanderson
Developer Programs Engineer
Google Developers: developers.google.com

• Content management system
• Events calendar
• Developer showcase
• Google Developer Groups
• Google Developers Live
• Google I/O
This Talk

- Design of a piece of the content management system
  - What we actually use
  - Focus on the App Engine-y bits

- Tools and implementation
  - A modern implementation, in Python 2.7
  - Using modern features of App Engine
  - Discrepancies and alternatives
Requirements

• Content managed separately from other parts of the site

• File-based CMS
  - content developed “offline,” published with a client tool
  - developer publishes a set of changes at a time, over many files

• Access controls for publishing

• Fast file serving

• Many content developers working concurrently
publish <dir-or-file> [<dir-or-file>...]
Upload:

- URL path
- content type
- data

Diagram:

Client

```
upload
update
```

Server

```
update
update
update
```
client

start

upload

upload

upload

server
"This is what I have..."

Start:

- project root(s)
- files to keep

Client:

- start
- upload
- upload
- upload

Server:

- Query, Delete, Delete, Delete...
- Update
- Update
- Update
"This is what I have..."

Control-C
"This is what I have..."

- start
- upload
- upload
- upload
- commit

- Store
- Store
- Store
- Store
- Query, Update, Delete, Update...
"This is what I have..."

client

start

upload

upload

upload

commit

server

Store

Store

Store

Store

Commit

\rightarrow Query, Update, Delete, Update...
The Publishing Protocol

• **Start** a new Change, with the given change description

• **Upload** data for new and updated files in the change

• **Commit** the change

• Or **Abort** the change
  
  - Cron job aborts stale Changes
Authenticated Web Services

- Google Cloud Endpoints

- Server libs for Python and Java

- Client libs for practically everything

- Especially easy for mobile (Android, iOS) and rich web (JavaScript)
from protorpc import messages

class StartRequest(messages.Message):
  project_prefixes = messages.StringField(1, repeated=True)
  upload_paths = messages.StringField(2, repeated=True)

class StartResponse(messages.Message):
  change_id = messages.IntegerField(1, required=True)
Google Cloud Endpoints: Server

• Messages can be defined as “ProtoRPC”s...

• ... or directly from ndb models, with Endpoints Proto Datastore
from google.appengine.ext import endpoints
from protorpc import remote

@endpoints.api(
    name='sitepublish',
    version='v1',
    description='Site Publish API',
    allowed_client_ids=CLIENT_IDS)
class SitePublishApi(remote.Service):
    # ...
Google Cloud Endpoints: Server
API Definition: Method

```python
@endpoints.api(...)
class SitePublishApi(remote.Service):
    @endpoints.method(
        StartRequest,
        StartResponse,
        name='start', path='start')
    def start(self, request):
        # ...
```
Google Cloud Endpoints: Server
API Definition: Authorization

```python
@endpoints.method(
    StartRequest,
    StartResponse,
    name='start', path='start')
def start(self, request):

    user = endpoints.get_current_user()
    if user is None or user.email() not in CONTENT_DEVELOPERS:
        raise endpoints.UnauthorizedException()

    # ...
```
Google Cloud Endpoints: Server

API Definition: Request and Response

```python
@endpoints.method(  
    StartRequest,  
    StartResponse,  
    name='start', path='start')
def start(self, request):
    # ...

    change = start_change(
        request.project_prefixes,  
        request.upload_paths,  
        endpoints.get_current_user())
    response = StartResponse(change_id=change.get_change_id())
    return response
```
Google Cloud Endpoints: Server
API Definition: Service

```python
@endpoints.api(...)
class SitePublishApi(remote.Service):
    # ...

app = endpoints.api_server([SitePublishApi], restricted=False)
```
Google Cloud Endpoints: Server

API Definition: Service

```yaml
application: site-publish
version: 1
runtime: python27
api_version: 1
threadsafe: true

handlers:
- url: /_ah/spi/.*
  script: services.app
```
Google Cloud Endpoints: Server

- Authenticated endpoints use client IDs
  - user signs in to Google, client gets permission to act as user when calling service

- Manage client IDs with the Google API Console
  - https://developers.google.com/console/
Google Cloud Endpoints: Server
Google Cloud Endpoints: Server

• Create a project

• Under API Access, create a client
  - “Web application,” even though this is a command-line tool

• For the new client, Edit settings...
  - Authorized Redirect URIs:
    http://localhost:8080/
    http://localhost:8090/
    http://your-site.com/
from google.appengine.ext import endpoints
from protorpc import remote

@endpoints.api(
    name='sitepublish',
    version='v1',
    description='Site Publish API',
    allowed_client_ids=CLIENT_IDS)
class SitePublishApi(remote.Service):
    # ...
from google.appengine.ext import endpoints
from protorpc import remote

CLIENT_IDS = ['145889693104-t0nm6og9vt8qmrkdkus1aecm7d45stcgr.apps.googleusercontent.com',
              endpoints.API_EXPLORER_CLIENT_ID]

@endpoints.api(  
    name='sitepublish',  
    version='v1',  
    description='Site Publish API',  
    allowed_client_ids=CLIENT_IDS)
class SitePublishApi(remote.Service):
Google Cloud Endpoints: Server

• Try it out in the development server, no client code needed!
  
  - http://localhost:8080/_ah/api/explorer
Demo
Google Cloud Endpoints: Server

![Google APIs Explorer](https://developers.google.com/apis-explorer/

**Services**
- sitepublish API

**All Versions**

**Request History**
Google Cloud Endpoints: Server
Google Cloud Endpoints: Server
Google Cloud Endpoints: Client

• google-api-python-client
  - https://developers.google.com/api-client-library/

• Client uses a “discovery document” that describes the service API

• Looks like a language-native API to the client code

• Library contains tools for making OAuth easy
Google Cloud Endpoints: Client

• Generate the discovery document for the service: endpointscfg.py

    ~/google_appengine/endpointscfg.py gen_discovery_doc
    -o . -f rest --hostname=localhost:8080
    services.SitePublishApi

• Make a one for testing, and one for real, using hostname parameter

    - Hand-edit localhost version to replace “https” with “http”
Google Cloud Endpoints: Client
Loading the Discovery Document

```python
import os
from apiclient import discovery

discovery_doc_fname = os.path.join(
    os.path.dirname(__file__),
    'SitePublishApi.discovery')
discovery_doc = open(discovery_doc_fname).read()

site_publish_service = discovery.build_from_document(discovery_doc)
```
import httplib2
import oauth2client

storage = oauth2client.file.Storage(CREDENTIALS_FILENAME)
credentials = storage.get()

if credentials is None or credentials.invalid:
    flow = oauth2client.client.OAuth2WebServerFlow(
        client_id=CLIENT_ID,
        client_secret=CLIENT_SECRET,
        scope='https://www.googleapis.com/auth/userinfo.email')

    credentials = oauth2client.tools.run(flow, storage)

http = credentials.authorize(httplib2.Http())
Google Cloud Endpoints: Client

Calling the Service

```python
request = site_publish_service.start(
    body={
        'project_prefixes': ['/foo/'],
        'upload_paths': ['/foo/bar.html', '/foo/baz.png'])

response = request.execute(http=http)

change_id = response['change_id']
```
Demo
Data Modeling with ndb

• Modeling changes

• Strict ordering of changes
  - Datetimes?
  - System IDs?

<table>
<thead>
<tr>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>key: ...</td>
</tr>
<tr>
<td>project_prefixes</td>
</tr>
<tr>
<td>upload_paths</td>
</tr>
<tr>
<td>created_by</td>
</tr>
<tr>
<td>is_committed</td>
</tr>
<tr>
<td>is_aborted</td>
</tr>
</tbody>
</table>
Data Modeling with ndb

• Idea: Store the “next change ID,” update it transactionally when creating changes
• Singleton entity for the change ID
• Use a cross-group transaction

<table>
<thead>
<tr>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>key: ...</td>
</tr>
<tr>
<td>project_prefixes</td>
</tr>
<tr>
<td>upload_paths</td>
</tr>
<tr>
<td>created_by</td>
</tr>
<tr>
<td>is_committed</td>
</tr>
<tr>
<td>is_aborted</td>
</tr>
</tbody>
</table>
Data Modeling with ndb

- Idea: Store the “next change ID,” update it transactionally when creating changes
- Singleton entity for the change ID
- Use a cross-group transaction (or just put all changes in the same entity group)
from google.appengine.ext import ndb

class ChangeGroup(ndb.Model):
    next_id = ndb.IntegerProperty(required=True)

class Change(ndb.Model):
    upload_paths = ndb.StringProperty(repeated=True)
    project_prefixes = ndb.StringProperty(repeated=True)
    created_by = ndb.UserProperty()
    is_committed = ndb.BooleanProperty(default=False)
    is_aborted = ndb.BooleanProperty(default=False)

    def get_change_id(self):
        return int(self.key.string_id())

    @classmethod
    def get_key(cls, change_id):
        return ndb.Key(ChangeGroup, '1', cls, '%012d' % change_id)
Data Modeling with ndb

• Modeling the content
  - View request can access its data using a `get()` by key = URL path
  - Publishing needs to be able to store new content separately from live content, then “switch”

• Idea: Path entity keyed by URL path, with pointer to Content entity
Data Modeling with ndb

• Modeling the content
  - View request can access its data using a `get()` by key = URL path
  - Publishing needs to be able to store new content separately from live content, then “switch”

• Idea: Path entity keyed by URL path, with pointer to Content entity

```
Path
key: URL path
content_key

Content
key: ...
content_type
data
```
Data Modeling with ndb

- Entity grouping for content objects?
- Idea: One group per Path, containing the Path and multiple Content objects
- Can set content_key and delete old Content entity in one transaction
Data Modeling with ndb

Path
- key: URL path
- content_key

Content
- key: [Path]:1234
- content_type
- data
Data Modeling with ndb

Path
- key: URL path
- content_key

Content
- key: [Path]:1234
- content_type
- data

Content
- key: [Path]:1257
- content_type
- data
Data Modeling with ndb

Path

key: URL path
content_key

Content

key: [Path]:1234
content_type
data

Content

key: [Path]:1257
content_type
data
Data Modeling with ndb

Path and Content Models

```python
class Path(ndb.Model):
    content_key = ndb.KeyProperty()
    is_deleted = ndb.BooleanProperty(default=False)
    last_applied_change_id = ndb.IntegerProperty()

    @classmethod
    def get_key(cls, path):
        return ndb.Key(cls, path)

class Content(ndb.Model):
    data = ndb.BlobProperty()
    content_type = ndb.StringProperty()

    @classmethod
    def get_key(cls, path, change_id):
        return ndb.Key(Path, path, cls, str(change_id))
```
Applying a Change

• Client calls the `commit()` method with the change ID

• Server updates the Change record and initiates the “apply” task

• Commit and apply task are stored transactionally
  - If either fails, neither occurs, and client sees the error

• Apply task spawns more tasks to paint the changes onto the website

• Any failed tasks get retried
Conflict Resolution

• What happens when two changes are applied out of order?
• Changes are ordered
• Store the last change ID with the Path
• Apply phase only “rolls forward”
• Deletes leave “tombstones,” so later deletes stick
  - p.is_deleted = True, p.last_change_id = 1234
  - can delete Content, but don’t delete Paths
Conflict Resolution

• What happens when two changes are applied out of order?

• Changes are ordered

• Store the last change ID with the Path

• Apply phase only “rolls forward”

• Deletes leave “tombstones,” so later deletes stick
  - p.is_deleted = True, p.last_change_id = 1234
  - can delete Content, but don’t delete Paths

Path

<table>
<thead>
<tr>
<th>key: URL path</th>
</tr>
</thead>
<tbody>
<tr>
<td>content_key=None</td>
</tr>
<tr>
<td>is_deleted=True</td>
</tr>
<tr>
<td>last_change_id=1234</td>
</tr>
</tbody>
</table>
Summary

• CMS with remote transactional publishing, arbitrary change size, eventual consistency

• Easy authenticated web services: Google Cloud Endpoints

• Transactional data storage: Google Cloud Datastore

• Data modeling: ndb for Python

• Background tasks: App Engine Task Queue

• Caching layer: Memcache, ndb

• Large object storage: Google Cloud Storage
Thanks!

developers.google.com

github.com/dansanderson
/site-publish

ae-book.appspot.com

Dan Sanderson
Programming Google App Engine, 2nd ed.
Large Asset Support

• Blobstore / Google Cloud Storage

• Uploading:
  - Client calls new endpoint for generating a Blobstore upload URL
  - Client makes MIME multipart POST to that URL
  - Server gets the Blobstore key, stores it in the Path

• Serving:
  - Server gets the Blobstore key in Path instead of Content key
  - Server puts Blobstore key in response, App Engine serves the file
Memcache

• Avoid hitting the datastore twice for every view request

• Use ndb to cache datastore entities automatically; just set a cache policy!

• Per-entity caching vs. result caching

• Don’t forget etags and cache controls
Faster Uploads

• Multi-threaded uploading
  - Be sure to use a separate `httplib2.Http()` instance per thread.

• Batched uploads in the upload API

• Not-modified check at start time

• Compressed payloads