
Google™
11
I/O

App Engine Backends

Justin Haugh, Greg Darke
May 10th, 2011

First things first

Justin Haugh
Software Engineer
Systems Infrastructure
jhaugh@google.com

Greg Darke
Software Engineer
Offline Infrastructure
darke@google.com

Session: http://goo.gl/MWeIF
Hashtags: #io2011 #AppEngine
Feedback: http://goo.gl/Gtt4A

Agenda

The App Engine Way
Backends: App Engine++

- Features
- Hello World
- Configuration
- Demo Time

Using Backends

- Best Practices
- Caveats
- The Future

Q&A

The App Engine Way

The App Engine Way

"The goal is to make it easy to get started with a new web app, and then make it easy to scale when that app reaches the point where it's receiving significant traffic and has millions of users."

App Engine Blog, April 7th, 2008

The App Engine Way

What that means:

- easy deployment
- dynamic scaling
- scalable storage layer
- rich set of APIs

The App Engine Way

- split big problems into small pieces
- fault-tolerance
- horizontal scaling
- web serving

The App Engine Way

But...

- not everything is a web app
  - run a command
  - generate reports
  - store counters

The App Engine Way

But...

- not everything is a web app
  - run a command
  - generate reports
  - store counters
- lightweight instances
  - small memory
  - limited CPU
  - not addressable

The App Engine Way

But...

- not everything is a web app
  - run a command
  - generate reports
  - store counters
- lightweight instances
  - small memory
  - limited CPU
  - not addressable
- limited execution
  - 30s deadlines
  - anonymous instances

Backends: App Engine++

Today, we are announcing the full public release of:

App Engine Backends

Today, we are announcing the full public release of:

App Engine Backends

- Backends are a powerful new way to write programs on App Engine.

Today, we are announcing the full public release of:

App Engine Backends

- Backends are a powerful new way to write programs on App Engine.
- Backends let you do things that were not possible on App Engine before.

Today, we are announcing the full public release of:

App Engine Backends

- Backends are a powerful new way to write programs on App Engine.
- Backends let you do things that were not possible on App Engine before.
- Backends make App Engine a more complete general-purpose computing platform.

Backends: App Engine++

What are Backends?

Backends: App Engine++

What are Backends?

- App Engine Instances, and then some

Backends: App Engine++

What are Backends?

- App Engine Instances, and then some
  - long-running

Backends: App Engine++

What are Backends?

- App Engine Instances, and then some
  - long-running
  - high-performance

Backends: App Engine++

What are Backends?

- App Engine Instances, and then some
  - long-running
  - high-performance
  - configurable

Backends: App Engine++

What are Backends?

- App Engine Instances, and then some
  - long-running
  - high-performance
  - configurable
  - addressable

Backends: App Engine++

What are Backends?

- App Engine Instances, and then some
  - long-running
  - high-performance
  - configurable
  - addressable
  - persistent

Backends: App Engine++

What are Backends?

- App Engine Instances, and then some
  - long-running
  - high-performance
  - configurable
  - addressable
  - persistent
  - cloud process

Backends: App Engine++

What are Backends?

- App Engine Instances, and then some
  - long-running
  - high-performance
  - configurable
  - addressable
  - persistent
  - cloud process

- Powerful building blocks

Backends: App Engine++

What are Backends?

- App Engine Instances, and then some
  - long-running
  - high-performance
  - configurable
  - addressable
  - persistent
  - cloud process

- Powerful building blocks
- Easy to use

Backends: App Engine++

What are Backends?

- App Engine Instances, and then some
  - long-running
  - high-performance
  - configurable
  - addressable
  - persistent
  - cloud process

- Powerful building blocks
- Easy to use
- Flexible

Features

Backends: App Engine++

Features

- RAM: 128MB to 1GB

Backends: App Engine++

Features

- RAM: 128MB to 1GB
- CPU: 600MHz to 4.8GHz

Backends: App Engine++

Features

- RAM: 128MB to 1GB
- CPU: 600MHz to 4.8GHz
- no request deadlines

Backends: App Engine++

Features

- RAM: 128MB to 1GB
- CPU: 600MHz to 4.8GHz
- no request deadlines
- run indefinitely

Backends: App Engine++

Features

- RAM: 128MB to 1GB
- CPU: 600MHz to 4.8GHz
- no request deadlines
- run indefinitely
- instance addressability

Backends: App Engine++

Features

- RAM: 128MB to 1GB
- CPU: 600MHz to 4.8GHz
- no request deadlines
- run indefinitely
- instance addressability
- resident or dynamic

Backends: App Engine++

Features

- RAM: 128MB to 1GB
- CPU: 600MHz to 4.8GHz
- no request deadlines
- run indefinitely
- instance addressability
- resident or dynamic
- automatic restarts

Backends: App Engine++

Features

- RAM: 128MB to 1GB
- CPU: 600MHz to 4.8GHz
- no request deadlines
- run indefinitely
- instance addressability
- resident or dynamic
- automatic restarts

App Engine-y!

- easy to configure
- fast deployment
- graphs & consoles
- dev_appserver

What does this mean?

App Engine is now a general-purpose cloud computing platform, suitable for:

- high-performance servers
- in-memory caching
- self-driven programs
- N-tiered architectures
- heavyweight offline processing

We're well beyond simple websites.

Use cases

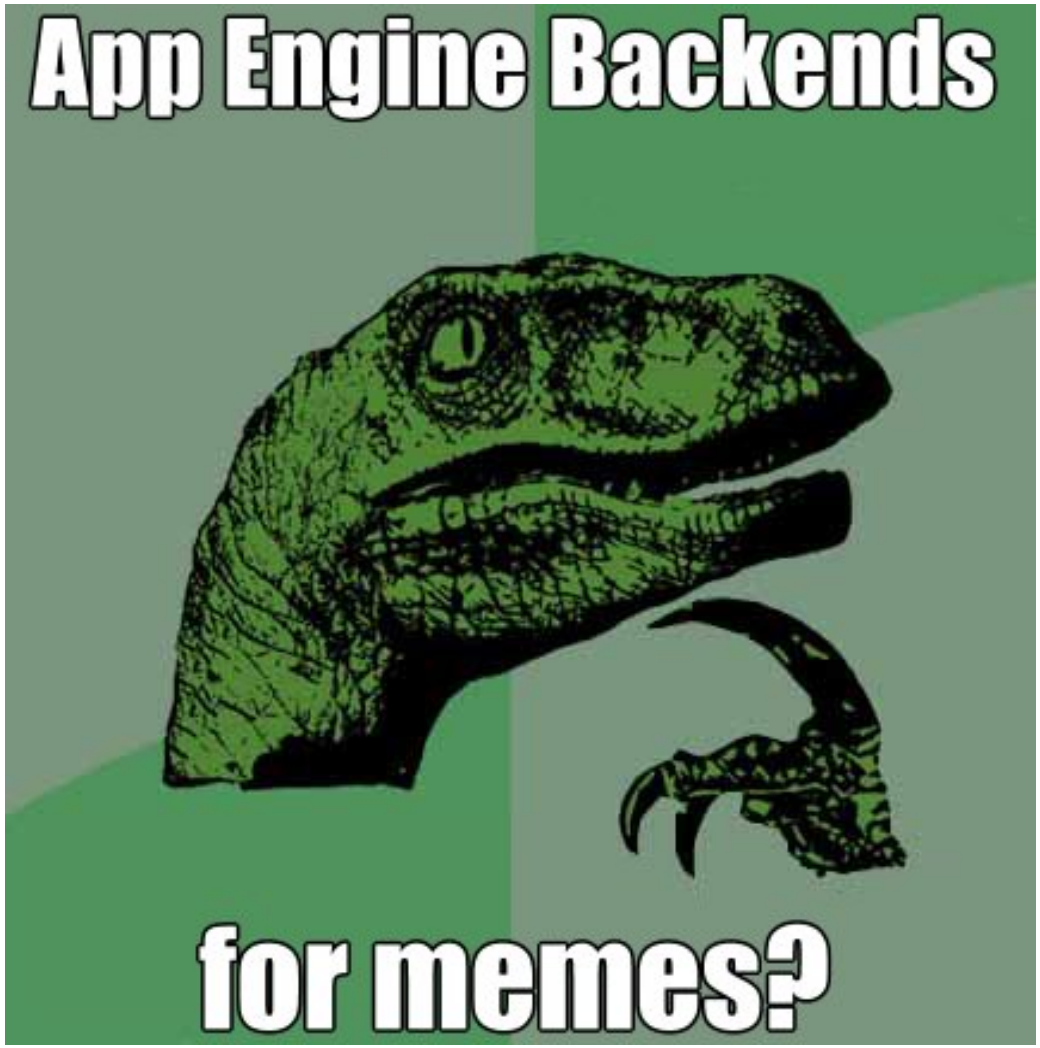
Memory-intensive

- search index
- social graph
- game state
- memcache

CPU-intensive

- image manipulation
- audio/video encoding
- game engines
- scientific computing
- meme generation

Use cases



Use cases

Background processing

- data pipeline
- data groomer
- web crawler
- task execution

Commands & Scripts

- schema migration
- ad-hoc queries
- load testing
- report generation

Hello World

~/hello/app.yaml

```
application: backends-io
runtime: python
version: 1
api_version: 1

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

~/hello/backends.yaml

```
backends:
- name: hello
  options: public
```

~/hello/backends.yaml

```
backends:
- name: hello
  options: public
```

Available at:

http://hello.backends-io.appspot.com

~/hello/backends.yaml

```
backends:
- name: hello
  instances: 15
  options: public
```

Available at:

http://hello.backends-io.appspot.com
http://0.hello.backends-io.appspot.com
http://1.hello.backends-io.appspot.com
...
http://14.hello.backends-io.appspot.com

~/hello/hello.py

```
from google.appengine.api import backends

print 'Content-Type: text/plain'
print ''
print 'Hello, my name is %s' %
    backends.get_backend()
```

~/hello/hello.py

```
from google.appengine.api import backends

print 'Content-Type: text/plain'
print ''
print 'Hello, my name is %s.%d' % (
    backends.get_backend(),
    backends.get_instance())
```

Commands

appcfg backends . update [backend]
appcfg backends . list
appcfg backends . start [backend]
appcfg backends . stop [backend]
appcfg backends . delete [backend]

Hello World: update

```
appcfg backends . update

Application: backends-io
Host: appengine.google.com

Starting update of app: backends-io, backend: hello
Scanning files on local disk.
Cloning 5 application files.
Precompilation starting.
Precompilation completed.
Starting deployment.
Checking if deployment succeeded.
Deployment successful.
Completed update of app: backends-io, backend: hello
```

Hello World: list

```
appcfg backends . list

backends:
- name: hello
  instances: 15
  options: public
  state: START
```

Hello World: stop

appcfg backends . stop hello

Application: backends-io
Host: appengine.google.com
Stopping backend: hello
Backend 'hello' stopped.

Hello World: stop

appcfg backends . stop hello

Application: backends-io
Host: appengine.google.com
Stopping backend: hello
Backend 'hello' stopped.

appcfg backends . list

backends:
- name: hello
  instances: 15
  options: public
  state: STOP

Hello World: start

```
appcfg backends . start hello

Application: backends-io
Host: appengine.google.com
Starting backend: hello
Backend 'hello' started.

appcfg backends . list

backends:
- name: hello
  instances: 15
  options: public
  state: START
```

Configuring Backends

Configuration

~/app/backends.yaml

- lists each backend
- can define up to 5

~/app/app.yaml

- defines app
- version is optional
- defines handlers
  - shared by app, backends

~/app/<code>

- shared by app, backends
- individually updated

Configuration

~/app/backends.yaml

```
backends:
- name: crawler
  start: crawler/main.py
- name: search
  class: B8
  instances: 5
- name: worker
  options: dynamic
```

backends.yaml

- name
- instances
- class
- start
- options

backends.yaml

- name
  - used in commands
    - appcfg backends . start [backend]
  - used in URLs
    - [backend].app.appspot.com
  - used in APIs
    - Task Queue, Cron: target
  - global
    - shares namespace with versions
    - backends are not versioned
  - Backends API
    - backends.get_backend()

backends.yaml

- instances
  - number of instances
    - resident: exactly N, always
    - dynamic: up to N, based on traffic
  - used in URLs
    - [instance].[backend].app.appspot.com
  - limits
    - max: 20 per backend
  - Backends API
    - backends.get_instance()

backends.yaml

- class
  - price + performance
    - B1: 128M, 600MHz, $0.08/hr
    - B2: 256M, 1.2GHz, $0.16/hr
    - B4: 512M, 2.4GHz, $0.32/hr
    - B8: 1GB, 4.8GHz, $0.64/hr
  - price
    - includes memory & cpu
  - tracked by the minute
    - startup: 15 minute charge
  - adjustable
    - reconfigure class

backends.yaml

- start
  - script to handle /_ah/start
    - one per backend
  - two uses
    - initialize state
    - run forever
  - startup period
    - other requests wait for /_ah/start to finish
    - success: HTTP 200-299 or 404
  - failed start
    - failure: instance is restarted

backends.yaml

- options
  - set of boolean flags
  - public
    - allow external HTTP
  - dynamic
    - startup on demand
    - scales with traffic
    - shutdown when idle
  - failfast
    - disable pending queue
    - busy = immediate 503

Demo Time

Demo: A generic counting service

Frontend:

- Performs urlfetch to backend
- Displays value to user

Counter backend:

- Uses a dictionary as an in instance cache
  - loads counter from datastore if unknown
  - flushes cache on shutdown

Loadtest backend:

- Multiple instances
- Uses urlfetch to call counter backend

http://backends-io.appspot.com/welcome

~/app/frontend.py

```
from google.appengine.api import backends
from google.appengine.api import urlfetch

url = '%s/backend/counter/inc' % (
    backends.get_url('counter'))

count = urlfetch.fetch(url, method='POST',
    payload='name=visitor&delta=1').content

print 'Content-Type: text/plain'
print ''
print 'Welcome visitor %s' % count
```

http://backends-io.appspot.com/welcome

~/app/loadtest.py

```
import random
from google.appengine.api import backends
from google.appengine.api import urlfetch

url = '%s/backend/counter/inc' % (
    backends.get_url('counter'))

names = ['counter-%d' % i for i in range(10)]

while True:
  counter = random.choice(names)
  params = 'name=%s&delta=1' % counter
  urlfetch.fetch(url, method='POST',
      payload=payload)
```

http://backends-io.appspot.com/welcome

~/app/counter.py

```
class CounterModel(db.Model):
  value = db.IntegerProperty(default=0)
  _dirty = False

class CounterStore(object):
  def __init__(self):
    self._store = {}
    self._has_shutdown = False
```

http://backends-io.appspot.com/welcome

~/app/counter.py

```
class CounterStore(object):
  # Continued ...

  def get_value(self, name):
    if name not in self._store:
      model = CounterModel.get_or_new(name)
      self._store[name] = model
    return self._store[name]

  def inc_value(self, name, delta):
    model = self.get_value(name)
    model.value += delta
    model._dirty = True
    if self._has_shutdown: model.put()
    return model
```

http://backends-io.appspot.com/welcome

~/app/counter.py

```
class CounterStore(object):
  # Continued ...

  def shutdown_hook(self):
    self.flush_to_datastore()
    self._has_shutdown = True

_counter_store = CounterStore()

class StartHandler(webapp.RequestHandler):
  """Handler for /_ah/start."""
  def get(self):
    runtime.set_shutdown_hook(
        _counter_store.shutdown_hook)
```

http://backends-io.appspot.com/welcome

Demo: A generic counting service

Persistence via the shutdown hook:

- Datastore Viewer
- Instances console

http://backends-io.appspot.com/welcome

Using Backends

Best Practices

- Resident Backends
- Dynamic Backends
- Scaling
- Startup
- Shutdown
- Logging
- Fail-Fast
- Message Passing
- Task Queues
- Handlers

Resident Backends

- Resident
  - instances always on
  - automatic restarts
  - run forever
  - explicit start/stop
- Uses
  - continuous execution
  - pull queues
  - large addressable memory
    - web index
    - memcache
- Pattern
  - start: load up state
  - handle requests

Dynamic Backends

- Dynamic
  - instances on demand
  - pay for what you use
  - no management of start/stop
- Uses
  - task execution
  - running a script
  - memcache
- Pattern
  - start: load up state
  - handle requests
  - shutdown: write out state

Scaling

- How does your backend scale?
  - offline
    - you control throughput
    - hit limits = slower processing
    - resize = pause, resume
  - online
    - you're (usually) not in control
    - hit limits = site is broken
    - resize = site is down
- Monitor resource usage
  - Instances Console
  - runtime.cpu_usage()
  - runtime.memory_usage()

Scaling

- Default: take some downtime
  - update causes stop
  - brief downtime window
  - minimize shutdown time
  - minimize load time
- Better: routing
  - server-1, server-2
  - flip away, resize, flip back
  - canaries, staging
- Best: options: dynamic
  - custom routing logic
  - balance over N at a time
  - initialize via script

Startup

- /_ah/start
  - sent by App Engine
  - to initialize the process
  - can run indefinitely
- differences
  - resident
    - at start time
    - automatic restart
  - dynamic
    - at request time
    - no automatic restart

Shutdown

- Polite Shutdown
  - 30s notice
  - can checkpoint state
  - examples
    - machine maintenance
    - App Engine maintenance
    - scheduling change
- Hard Shutdown
  - zero notice
  - examples
    - machine failure
    - datacenter failure
    - exceeded memory limit

Shutdown

Shutdown

- Runtime API
  - used to persist state when shutdown occurs
  - two methods
    - check for shutdown (polling)
    - register shutdown callback

Shutdown

- Runtime API
  - used to persist state when shutdown occurs
  - two methods
    - **check for shutdown (polling)**
    - register shutdown callback

```
from google.appengine.api import runtime

def checkpoint():
  memory.write()

while True:
  work(period=10)
  if runtime.is_shutting_down():
    checkpoint()
    break
```

Shutdown

- Runtime API
  - used to persist state when shutdown occurs
  - two methods
    - check for shutdown (polling)
    - **register shutdown callback**

```
from google.appengine.api import runtime

def checkpoint():
  memory.write()

runtime.set_shutdown_hook(checkpoint)
work(period=86400)
```

Logging

- Logs are flushed periodically
- Auto

```
from google.appengine.api import logservice
logservice.AUTOFLUSH_EVERY_BYTES
logservice.AUTOFLUSH_EVERY_LINES
logservice.AUTOFLUSH_EVERY_SECONDS
```

- Manual

```
logservice.flush()
```

Fail-Fast

- For sophisticated clients
  - perform own queuing
  - perform own retries
  - tolerant of failure
  - want immediate notification
- Examples
  - AJAX client
  - mobile apps
  - external queues
- Server-side
  - options: failfast
- Client-side
  - X-AppEngine-FailFast: true

Message Passing

- How to communicate between instances?
- URLFetch
  - send message = make request
  - problem: single-threaded runtimes
- Memcache
  - both read/write to cache entries
- Datastore
  - instance A: write an entity
  - instance B: read an entity
- Task Queues
  - task = message
  - each instance has a push/pull queue

Task Queues

- Perfect for working with Tasks
  - run tasks forever
  - async message passing
  - batching w/pull queues
- Push Queues

`target` directive in queues.yaml

  - `target` parameter to taskqueue.add()
- Pull Queues

```
queue.add(taskqueue.Task(
   method='PULL', ...))
queue.lease_tasks(3600, 10)
```

- Talk: Putting Task Queues to Work
  - http://goo.gl/TiNIb

Code & Handlers

- Same directory
  - code, handlers shared
  - hide backends with `login: admin`
- Two directories: app, backends
  - separate code, handlers for backends
  - no risk of exposing Backend logic in App
- N+1 directories
  - each app version, backend
  - when code is substantially different
  - 3rd-party backends

Caveats

Configuration Limits

Limits

- app: 5 backends
- app: 10GB of backends
- backend: 20 instances
- backend: 10GB
- 10GB combinations
  - B8x10
  - B4x20
  - B8x5 + B4x10
  - B8x5 + B4x5 + B2x10

Caveats

- API deadlines apply
  - urlfetch: 5s default, up to 10s
  - datastore: 30s
- Size limits apply
  - HTTP: 32MB requests
  - urlfetch: 1MB request, 32MB response
  - memcache: 1MB objects
  - Blobstore: 2GB objects, 1MB response
  - Mail: 10MB send/receive
  - Tasks: 100KB

Caveats

- No uptime guarantee
  - best-effort service
  - expect polite and hard shutdown
  - various causes
- Examples
  - software bugs
  - hardware failures
  - emergencies
- Talk: Life in App Engine Production
  - http://goo.gl/RdsKv

The Future

The Future

- Better scaling
  - auto-scaling
  - scaling API
- Better updates
  - rolling updates
  - online updates
- Better concurrency
  - java background threads
  - python concurrency
- Better configuration
  - separate handlers
  - versioning

The Future

- Better uptime
  - fewer restarts
  - uptime statistics
- API Integrations
  - Channel, XMPP
  - Mail
  - MapReduce
- More power
  - new instance classes
  - larger API calls
  - longer API deadlines
- Streamed responses
- Sockets API

Recap: App Engine Redefined

- Application
  - 30-second requsts
  - dynamic scaling
  - lightweight instances
- Backends
  - long-running requests
  - max instance count
  - up to 1GB, 4.8GHz
- Both
  - easy to configure
  - full production support
  - managed by Google


Google™
11
I/O