Data Modeling for Google App Engine using Python and ndb

Dan Sanderson
October 9, 2012

Google Developers


python
powered

Google App Engine

- Platform for building scalable web applications
- Built on Google infrastructure
- Pay for what you use
  - Apps, instance hours, storage, bandwidth, service calls
  - Free to start!
- Launched with Python 2.5 exclusively in 2008; then Java, Go, Python 2.7

Google App Engine

- Easy development
- Easy deployment
- No servers to manage, no OS to update; App Engine does this for you
- Based on standard technologies: Python 2.7, WSGI

Agenda

- The App Engine datastore
- Data modeling
- Queries
- Transactions
- Automatic caching

Agenda

- Batching
- Asynchronous calling
- “Tasklets”

The App Engine Datastore

The Datastore

- *scalable*
- *queryable*
- *transactional*

The Datastore

- entity
- key
  - kind, ID
- properties
  - name, typed value

The Datastore

```
player = ...

player.name = ‘druidjane’
player.level = 7

now = datetime.datetime.now()
player.create_date = now

player.put()
```

The Datastore

- “schemaless” object storage

```
p1 = ...
p1.level = 7
p1.put()

p2 = ...
p2.level = ‘warrior’
p2.put()

p3 = ...
p3.put()
```

ndb

- Data modeling library, runs entirely in your application code
- ext.db launched with AE in 2008
- ndb started by Guido van Rossum (creator of Python, App Engine dev)
- GA in version 1.6.4, March 2012
- *requires Python 2.7*

ndb

```
from google.appengine.ext import ndb

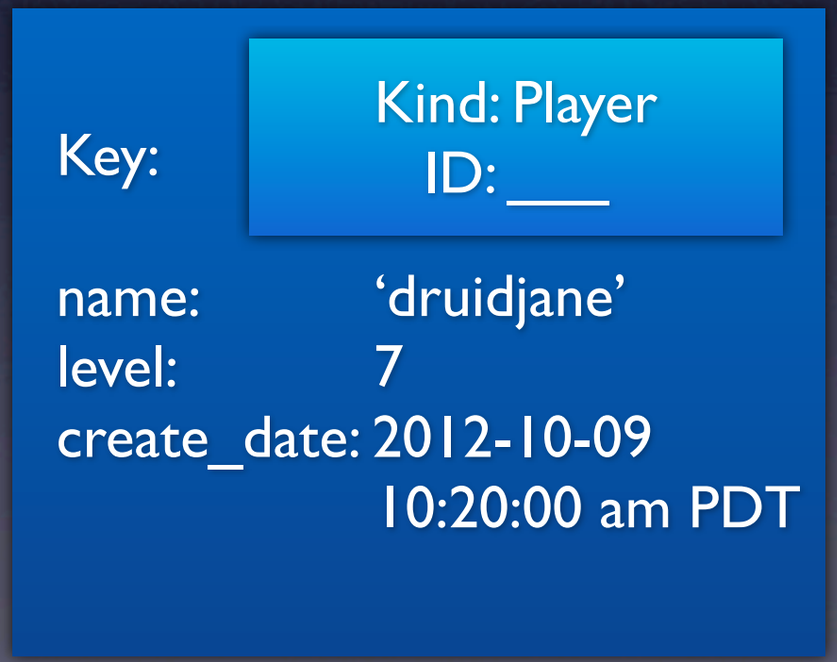
class Player(ndb.Model):
  name = ndb.StringProperty()
  level = ndb.IntegerProperty()
  create_date = ndb.DateTimeProperty()

p1 = Player()
p1.level = 7
p1.put()

p2 = Player()
p2.level = ‘warrior’  # BadValueError
p2.put()
```

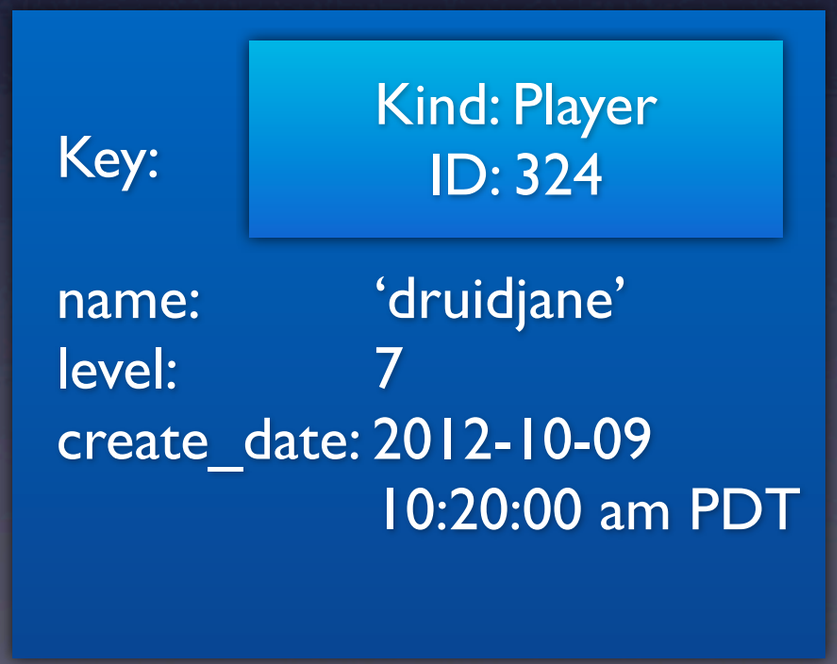
ndb

```
p3 = Player(name='druidjane',
            level=7,
            create_date=now)
# ...
```



ndb

```
p3 = Player(name='druidjane',
            level=7,
            create_date=now)
p3key = p3.put()
```



ndb

```
player_key = ndb.Key(Player, 324)
player = player_key.get()

if player.level > 5:
  # ...
```

Key: Kind: Player, ID: 324

name: ‘druidjane’
level: 7
create_date: 2012-10-09 10:20:00 am PDT

Data Modeling

Data Modeling

- Declare entity structure
- Validate property values (types, ranges)
- Python-like class/object interface

Data Modeling

- Subclass ndb.Model
  - Name of subclass is entity Kind (`Player`)
  - Use class attributes to declare property names, types, and parameters
    - `name = ndb.StringProperty()`
    - `create_datetime = ndb.DateTimeProperty(auto_now_add=True)`

Type	Property
int, long	IntegerProperty
float	FloatProperty
bool	BooleanProperty
str, unicode	StringProperty
datetime	DateTimeProperty
date	DateProperty
time	TimeProperty
ndb.GeoPt	GeoPtProperty
users.User	UserProperty
ndb.Key	KeyProperty
None	

Data Modeling

- Declaration can specify parameters
- `name = ndb.StringProperty(required=True)`
- `level = ndb.IntegerProperty(default=1)`
- `charclass = ndb.StringProperty(`
  `choices=[‘mage’, ‘thief’, ‘warrior’])`
- `indexed=False; repeated=True;`
  `validator=some_function`

Data Modeling

- JsonProperty(compressed=True)
- PickleProperty(compressed=True)
- GenericProperty()
- ComputedProperty(func)
  - ```
    last_name = ndb.StringProperty(required=True)
    last_name_lc = ndb.ComputedProperty(
        lambda self: self.last_name.lower())
    ```
    ```

Data Modeling

- StructuredProperty(InnerModelClass)
- Uses an ndb.Model subclass to model the property value
- In code, the value is an instance of the model class
- In the datastore, fields become properties of the main entity, not separate entities
  - Can be queried!

Data Modeling

```
class PlayerHome(ndb.Model):
  sector = ndb.IntegerProperty()
  house_num = ndb.IntegerProperty()
  roof_color = ndb.StringProperty()
```

```
class Player(ndb.Model):
  # ...
  home = ndb.StructuredProperty(PlayerHome)

p1 = Player()
p1.home = PlayerHome()
p1.home.sector = 698714526
p1.home.house_num = 123
```

Queries

Queries

- Query all entities of a Kind based on property values
- Filters: `level > 5`
- Orders: `score`, descending
- Returns full entities, partial entities ("projection queries"), or just keys

Queries

- *Scalable:* query speed is *not* affected by the number of records in the datastore, only the number of results!
- *All queries are pre-indexed.*
- Built-in indexes
- Custom indexes
  - Development server helps generate index configuration

Queries

```
query = Player.query()

query.order(Player.level, -Player.score)
```

```
query.filter(Player.level >= 5)
query.filter(Player.charclass == ‘warrior’)
```

```
query = Player.query(Player.level >= 5)
```

Queries

```
players = query.fetch(20)
for player in players:
  # player.name ...
```

```
keys = query.fetch(20, keys_only=True)

for player in query:
  # player.name ...

for key in query.iter(keys_only=True):
  # ...
```

GQL

```
query = Player.gql('WHERE level >= 5 '
                   'ORDER BY score')
```

```
query = ndb.gql('SELECT Player '
                'WHERE level >= 5 '
                'ORDER BY score')
```

!= and IN

```
query = Player.query(
    Player.charclass != ‘warrior’)

query = Player.query(
Player.charclass.IN([‘thief’, ‘mage’])
```

Implemented as multiple queries, with results deduplicated. (Beware limitations.)

AND and OR

```
query = Player.query(
    ndb.AND(Player.charclass == ‘warrior’,
            Player.level >= 5))
```

```
query = Player.query(
    ndb.OR(Player.charclass == ‘thief’,
           Player.charclass == ‘mage’))
```

AND simply concatenates filters, as before.
OR uses multiple queries, with results deduplicated. (Beware limitations.)

Projection Queries

```
query = Player.query()

results = query.fetch(20,
    projection=[Player.name, Player.level])
for player in results:
  # player.name ...
  # (player.score not set)
```

Projected property values are pulled directly from the index, and so must all be indexed properties.

Cursors

- Seeking by count is slow
- A cursor remembers where a previous query stopped, so it can be resumed
  - ... in a later request
- Paginated displays
- Batch jobs

Cursors

- Fetch results using an iterator, with cursors enabled:
```
it = query.iter(produce_cursors=True)
for result in it: # ...
```

- Test whether there's another result:
```
if it.has_next(): # ...
if it.probably_has_next(): # ...
```

- Get a cursor after fetching results:
```
cursor = it.cursor_after()
```

Cursors

- Pass cursor to next request:
  `self.response.write(cursor.urlsafe())`
- In next request, reconstruct the cursor value:

```
cursor = ndb.Cursor.from_websafe_string(
    self.request.get('cursor'))
```

- Use the cursor for the next query:
  `it = query.iter(start_cursor=cursor)`
- It must be the same query: kind, filters, sort orders

Cursors

- Shortcut: `fetch_page()`

```
cursor = ndb.Cursor.from_websafe_string(
    self.request.get('cursor'))

(results, cursor, more) = \
    query.fetch_page(
        20, start_cursor=cursor)

if more:
    # render "Next" link with cursor
```

Transactions

Transactions

- Extremely important subject!
- For today, just looking at the API briefly
- Concepts are similar to ext.db
- *(See the book and online docs.)*

Transactions

- "Local" transactions with "strong" consistency
- Optimistic concurrency control
- Entity groups
- Groups defined using keys; "ancestor" paths

Transactions

- All operations that participate in a transaction must be limited to entities in a single group
- (also: cross-group transactions)
- Decorate your functions to describe how they participate in transactions

Transactions

```
@ndb.transactional
def IncrementScore(player_key, score_incr):
  player = ndb.get(player_key)
  player.score += score_incr
  player.put()

# ...
  for player_key in winning_team_keys:
    IncrementScore(player_key, 500)
```

Transactions

```
@ndb.transactional
def AwardTrophies(player):
  if player.score > 500:
    trophy = Trophy(parent=player.key, ...)
    trophy.put()

@ndb.transactional
def IncrementScore(player_key, score_incr):
  player = ndb.get(player_key)
  player.score += score_incr
  AwardTrophies(player)
  player.put()
```

Automatic Caching

Automatic Caching

- Two automatic caching features
  - “In-context” cache
  - Memcache storage
  - Same basic idea, difference in scope

Automatic Caching

- Context cache starts empty for each request
- Minimizes datastore interactions throughout the request handler code
- `context = ndb.get_context()`
  `context.set_cache_policy(lambda key: True)`

Automatic Caching

- Memcache: global, distributed cache
- Outlives requests
- ndb handles serialization of model instance

```
def test_memcache_ok(key):
  # ...

context = ndb.get_context()
context.set_memcache_policy(
    test_memcache_ok)
```

Automatic Caching

- Can set caching policies on a per-class basis, overriding the global policy:

```
class Player(ndb.Model):
  _use_cache = True
  _use_memcache = False
```

Automatic Caching

- Can even set a “datastore policy”!

```
class ProgressMeter(ndb.Model):
  _use_cache = True
  _use_memcache = True
  _use_datastore = False

  meter = ndb.IntegerProperty()
```

Batching

Batching

- Several services support “batch” APIs, for reducing the number of RPCs
- `entities = ndb.get_multi(`
  `[key1, key2, key3])`
- Explicit batching calls: the *_multi() methods

Batching

- ndb batches automatically!
- Maintains batching queues for datastore, memcache, and even URL Fetch
- “Flushes” caches by performing batch operations
- Maintains consistent local view, such as via the in-context cache
- Only does this when it’s safe

Batching

- App can add requests to ndb-managed batching queues for memcache and URL Fetch using methods on the Context
- App can flush explicitly: `context.flush()`

Asynchronous Calling

Asynchronous Calling

- Some services support asynchronous calling:
  - App initiates call
  - Call returns a “future” object immediately; app code resumes, service works in parallel
  - App calls method on “future” object to get results; waits for service to finish, if necessary, then returns results

Asynchronous Calling

- ndb supports `*_async()` forms of most methods on Model and Query classes
- Future objects have a `get_result()` method (and other methods)

Tasklets

Tasklets

- A useful way to organize complex code that calls services
- Tasklet: application code that can be invoked like an asynchronous call
- Tasklet code does not execute concurrently
- Can yield to other pending tasklets when waiting for service calls
- ndb uses an event loop and auto-batching to drive tasklets to completion efficiently

Tasklets

(See the documentation.)

developers.google.com/appengine

appengine.google.com

ae-book.appspot.com

Programming Google App Engine, 2nd ed.
October 2012

Dan Sanderson
profiles.google.com/dan.sanderson