Marc's PyCon 2012 Notes Documentation

Release 1.0

Marc Abramowitz

January 27, 2014

Contents

Notes taken by Marc Abramowitz for PyCon 2012. Available on GitHub at http://github.com/msabramo/pycon2012-notes

Contents:

CHAPTER 1

Stop Mocking, Start Testing by Augie Fackler and Nathaniel Manista from Google Code

Presenters: Augie Fackler and Nathaniel Manista of Google Code

PyCon 2012 presentation page: https://us.pycon.org/2012/schedule/presentation/315/

Slides: https://code.google.com/a/google.com/p/stop-mocking-start-testing/

Video: http://pyvideo.org/video/629/stop-mocking-start-testing

Video running time: 34:53

Lessons

(04:18)

- Users are not a test infrastructure.
- Coded tests with greater human costs than CPU costs are also not a test infrastructure.
- A project simply cannot grow this way.

(07:59)

```
class Real(object):
    def EnLolCat(self, phrases, words=None, kitteh=None):
        # ...

class Fake1(object):
    def EnLolCat(self, phrases, words):
        # ...

class Fake2(object):
    def EnLolCat(self, phrases):
        # ...
```

(08:45)

"Mock objects tell you what you want to hear."

Tests run only against mock objects

Python doesn't check that a mock is true to what it is mocking

Developers don't either!

(09:48)

Lessons

- Share mocks among test modules.
- When choosing between zero and one mock, try zero!

(10:33)

If you need a mock, have exactly one well-tested mock.

(12:23)

Use full system tests to make up for gaps in unit- level coverage

(13:13)

Full system tests are not a replacement for unit-level tests!

(13:29)

Test stories with full system tests

1.1 Modern Mocking

(14:29)

A collection of...

... authoritative...

... narrow...

... isolated...

... fakes.

You don't want mocks spread out all over the place that need to be updated whenever the real object changes.

Don't mock things that don't need to be mocked - things that are cheap like simple data structures or things that are stateless or simple.

They use what they call "fakes" and don't find distinctions between doubles, stubs, mocks, etc. useful.

They don't really use declarative/fluent mocks that check that they're being called the right # of times, etc. It sounds like they have formal mock classes.

1.2 Testing Today

(17:58)

Tests are written to the interface, not the implementation

Unit tests are run against both mock and real implementations

System tests are run in continuous integration and quality assurance

1.2.1 Design For Love *And* Testing

(20:30)

- Inject object dependencies as required construction parameters.
- Separate state from behavior

- Define interfaces between components.
- Decline to write a test when there's no clear interface.

1.3 Injected dependencies

(20:38)

```
# Not this:
class BadView(BaseView):
    def __init__(self, database=None):
        if database is None:
            # This reads a global value set by
            # a command line flag
            database = DefaultDatabase()
        self._database = database

# This:
class GoodView(BaseView):
    def __init__(self, database):
        self._database = database
```

They used to have *optional* injected dependencies – i.e. if you didn't provide an argument with the dependency it would choose one automatically (either hard-coded or use something from a command-line option, config file, etc.)

1.4 Separate state from behavior

(21:34)

Separate state (especially storage) from behavior – i.e.: if a method has a part that touches object attributes and a part that doesn't; factor out the parts that don't touch the attributes into a separate "free function".

1.5 Define interfaces between components

(23:29)

1.6 Decline to write a test when there's no clear interface

(23:54)

1.7 Thank you

(24:50)

1.8 Questions

(25:21)

Someone asked about adding tests to legacy code and drawing a line in the sand.

(28:41)

I mentioned "Working Effectively with Legacy Code" by Michael Feathers for a guy who asked about adding tests to untested code.

I asked the speakers about "Tell, Don't Ask" and they were not familiar with it, so I don't think it's something that they adhere strongly to.

(30:04)

An interesting point someone made is that it is nice to be able to check that mocks adhere to interfaces, e.g..: using ABCs or zope.interface. This could probably be generalized to languages like PHP. For example, this might be an argument in favor of using formal interfaces over duck typing.

(31:40)

Q: How to decide what to unit test and what to system test?

(33:49)

Mocks vs. fakes - they treat mocks as a last resort - they don't write mocks for their own classes.

CHAPTER 2

Fast test, slow test by Gary Bernhardt from destroyallsoftware.com

Presenter: Gary Bernhardt (http://blog.extracheese.org/ / https://www.destroyallsoftware.com/) (@garybernhardt)

PyCon 2012 presentation page: https://us.pycon.org/2012/schedule/presentation/429/

Slides: ???

Video: http://pyvideo.org/video/631/fast-test-slow-test

Video running time: 31:50

2.1 Goals of tests

1. Prevent regressions

The weakest of the goals. Doesn't change the way you build the software minute to minute. At best it changes how you release the software. You don't release broken things.

2. Prevent fear (00:19)

Prevent fear minute to minute or second to second, so speed is important here. Enable refactoring.

3. Prevent bad design (00:52)

Very subtle topic. Sort of out of bounds. The holy grail of testing.

(01:32) Video of a large system test running.

(02:06) A smaller, synthetic example to dissect for this talk – test for a Django app; a discussion board

Writing tests from the bottom up is often easier

```
def test_that_spam_posts_are_hidden(self):
    assert 'Spammy!" not in resp.content

def test_that_spam_posts_are_hidden(self):
    resp = self.client.get("/discussion/%s" % disc.pk)
    assert 'Spammy!" not in resp.content

def test_that_spam_posts_are_hidden(self):
    self.client.post("/mark_post_as_spam", {'post_id': post.pk})
    resp = self.client.get("/discussion/%s" % disc.pk)
    assert 'Spammy!" not in resp.content
```

```
def test_that_spam_posts_are_hidden(self):
    log_in(alice)
    self.client.post("/mark_post_as_spam", {'post_id': post.pk})
    resp = self.client.get("/discussion/%s" % disc.pk)
    assert 'Spammy!" not in resp.content

def test_that_spam_posts_are_hidden(self):

    disc = Discussion()

    disc.save()
    log_in(alice)
    self.client.post("/mark_post_as_spam", {'post_id': post.pk})
    resp = self.client.get("/discussion/%s" % disc.pk)
    assert 'Spammy!" not in resp.content

def test_that_spam_posts_are_hidden(self):

    disc = Discussion()
    disc.posts.append(Post(poster=bob, "Spammy!"))
    disc.save()
    log_in(alice)
    self.client.post("/mark_post_as_spam", {'post_id': post.pk})
    resp = self.client.get("/discussion/%s" % disc.pk)
    assert 'Spammy!" not in resp.content

def test_that_spam_posts_are_hidden(self):
    alice, bob = User(admin=True), User(admin=False)
    disc = Discussion()
    disc.posts.append(Post(poster=bob, "Spammy!"))
    disc.save()
    log_in(alice)
    self.client.post("/mark_post_as_spam", {'post_id': post.pk})
    resp = self.client.get("/discussion/%s" % disc.pk)
    assert 'Spammy!" not in resp.content
```

(03:47) Why is this a system test?

What does it depend on? What thing in this test could cause it to break?

```
def test_that_spam_posts_are_hidden(self):
    alice, bob = User(admin=True), User(admin=False)
                      FLAG
    disc = Discussion()
                      SIGNATURE
    disc.posts.append(Post(poster=bob, "Spammy!"))
         RELATIONSHIP      SIGNATURE
    disc.save()
         VALIDITY
    log_in(alice)
    AUTH
    self.client.post("/mark_post_as_spam", {'post_id': post.pk})
                     URL, SIGNATURE, PRECONDITIONS
    resp = self.client.get("/discussion/%s" % disc.pk)
                       URL, SIGNATURE, PRECONDITIONS
    assert 'Spammy!" not in resp.content
           REPRESENTATION (E.G., NOT AJAX)
```

Note that:

```
assert 'Spammy!" not in resp.content
```

is a negative assertion, which is dangerous. If we change the view to render a skeleton and then fill in the details later with AJAX requests, this assertion *will always succeed.* This is one of the dangers of negative assertions.

(05:59) We are also dependent on:

- Django test client
- Django router
- Django request object
- Django response object
- Third party middleware (!!!)
- App middleware (!!!)
- Context managers (!!!)

(06:41) The result of these dependencies is that we end up with a *binary test suite* - tells you whether or not your code is broken but gives no clues to what. Good tests show you exactly what's broken.

(07:10) *Test fragility* – "Every time we change the code, we have to update all the tests!"

(07:32) We primarily get regression protection (and only specific kinds; the layers integrating incorrectly)

It's very difficult to test fine-grained edge cases from the outside.

It's not fast so it won't help with refactoring.

No feedback on design since we're not interacting with the smaller objects.

System tests have value but also have problems.

2.2 How To Fail

(08:29) 3 ways to fail:

1. Selenium as primary testing tool
2. "Unit tests" are too big
3. Fine-grained tests around legacy code

2.2.1 Selenium as primary testing

(08:34)

- Tests can't be run locally
- Tests too slow
- Tests break often
- No fine-grained feedback

2.2.2 "Unit tests" are too big

(09:20)

Testing time tends to grow super-linearly.

100ms = 240,000,000 instructions

2.2.3 Fine-grained tests around legacy code

(10:44) Fine-grained tests around legcy code – a way to fail – tight tests solidify the interface and bake all of the badness in. :-)

2.3 Unit tests

(11:20) Unit tests - what are they and why do we care?

Showed two videos of very fast test suites.

(12:29) We will test at the *model layer* instead of the view layer.

```
def test_that_spam_posts_are_hidden(self):
    post = Post(mark_post_as_spam=True)
    discussion = Discussion(posts=[post])
    assert discussion.visible_posts == []
```

This is a complete test at the model layer. This does not replace system tests; does not test views.

(13:18) Why is this a unit test?

(13:26) 1. Unit tests test only *one object behavior*.

(13:59) 2. Other classes can't break it.

And in particular, no dependencies on:

- Django test client
- Django router
- Django request object
- Django response object
- Third party middleware (!!!)
- App middleware (!!!)
- Context managers (!!!)

2.3.1 What are the advantages of unit tests?

(15:12) Test failures are much more isolated and tell you which object or method is broken.

(15:26) Tests are much faster. You can avoid fear. You can refactor. The difference between 400 milliseconds and 40 seconds – at 40 seconds, you can't do the thing called TDD.

(15:40) System tests test the boundaries better than unit tests. Unit tests test the fine-grained behavior of individual objects, which is most of the behavior of your system, hopefully.

(16:10) Unit tests enable refactoring and let you avoid fear.

(16:24) *Gives you design feedback.* I have conditioned myself to be repulsed by an 8 line test for a model. Why do I need to set up so much of the world to test this one small piece of behavior? Makes me think about refactoring, which leads to better system design.

(16:46) Guidelines for ratio of unit tests to system tests – 90% unit tests, 10% system/acceptance tests

(17:19) I have not mentioned test doubles or mocking. You may not these when testing the low levels like models. You may need them when testing higher level objects like views.

2.4 The End

(18:00)

@garybernhardt

destroyallsoftware.com

Screencasts for Serious Developers

- OO design
- Unix
- TDD
- Smaller, faster tests

2.5 Questions

(18:26) Q: 90/10 ratio - was that in time or lines of code or what?

A: Number of tests

(18:50) Q: Does 90/10 apply to every kind of project or does it vary?

(19:04) A: That applies mostly to object-heavy systems like web apps, that have lots of logic and not a lot of boundaries.

(19:44) Question (from Carl Meyer): Pain point in unit testing Django apps is the database. Slows down your tests. Django models objects are very tied to the database. Trying to mock out the persistence layer seems like a way to fail.

(20:36) Answer: Should you mock the model objects in a Django app? No it's too wide of a boundary that you don't control. A better approach is to create a service layer that interacts with the model objects and then mock that service layer.

(22:00) Question: How do you enforce that mock objects have the same behavior as the real object?

(22:19) Answer: System tests. Or in Ruby, rspec-fire from Xavier Shay

(23:36) Mock by Michael Foord does interface checks.

(24:32) Question: Why is it such a bad idea to unit test legacy code?

(24:37) Answer: It is good to test unit test legacy code. It's not good to write *fine-grained tests* for legacy code, because it solidifies the edges. I may be the worst offender, because my mocking library Dingus can magically mock everything on the outside layer of your class, so if you're doing this, stop it. :-) You want to read "Working Effectively With Legacy Code" by Michael Feathers.

(26:05) Integration tests == system tests?

(26:09) Answer: Oops, sorry. Main distinct is unit test (which tests one thing) vs. any kind of integrated test that tests multiple things.

(27:39) Question: When is Selenium appropriate?

(27:55) Answer: Selenium is not evil. If you use Selenium to test everything and especially fine-grained behavior, that's where you'll run into problems.

(28:20) He mostly works in Ruby these days and uses Cucumber with Capybara driving a headless WebKit browser.

(28:38) Don't pay non-programmers to build large Selenium test suites.

(29:15) Question: How do I convert a system test suite to unit tests and make sure that I'm covering everything the system tests covered?

(29:25) Answer: Ask Michael Feathers? :-) Sometimes it's obvious...

CHAPTER 3

Speedily Practical Large-Scale Tests with Erik Rose from Votizen

Presenter: Erik Rose (https://github.com/erikrose) (@ErikRose)

PyCon 2012 presentation page: https://us.pycon.org/2012/schedule/presentation/473/

Slides (Keynote): https://github.com/downloads/erikrose/presentations/Speedily%20Practical%20Large-Scale%20Tests.key

Slides (HTML): http://erikrose.github.com/presentations/speedily-practical-large-scale-tests/

Video: http://pyvideo.org/video/634/speedily-practical-large-scale-tests

iStat menus for the Mac so you know your baseline memory usage, disk activity, etc.

The Python profiler doesn't tell you about I/O; only CPU. You can use the UNIX time command to look at wall clock time and CPU time and then do subtraction to get an idea of I/O time. top, lsof

Conquest of Speed parts:

1. Per-class fixture loading
2. Fixture bundling
3. Startup Speedups

Got tests form 302 seconds down to 62 seconds.

Nose

- Finding and picking tests
- dealing with errors
- etc.

django-nose 1.0 will be released soon (Note: it's out - see PyPI) and Erik Rose will be sprinting on it on Monday.

3.1 Die, setUp(), die

More explicit test setup. Can be more efficient because you're not setting up stuff you don't need.

3.2 Die, fixtures, die

"model makers" – d = document(title='test')

with_save decorator

factory_boy for Python - based on thoughtbot's factory_girl for Rails

Shared setup makes tests...

- Coupled to each other
- Brittle
- Hard to understand
- Slow
- Kick puppies

Local setup gives you...

- Decoupling
- Robustness
- Clarity
- Efficiency
- Puppy kisses

Imperative vs. declarative

- Mock - Imperative
- Fudge - Declarative

The Mock Library:

```
from mock import patch

with patch.object(APIVoterManager, '_raw_voters') as voters:
    ....
```

The Fudge Library:

```
@fudge.patch('sphinxapi.SphinxClient')
def test_single_filter(sphinx_client):
    ....
    (sphinx_client.expects_call().returns_fake()
        .is_a_stub()
        .expects('SetFilter').with_args('a', [1], False)
        .expects('SetFilter')....
    ....
```

Horrible dots - They don't tell you anything

Hate tracebacks - too much noise

Erik Rose put together an alternative nose test runner called nose-progressive:

```
% nosetests --with-progressive
```

- displays a progress bar instead of dots so you know how long tests might take
- much abbreviated tracebacks
- editor shortcuts with the + syntax for line numbers that you can copy and paste to edit the file

If you just want the improved tracebacks, check out tracefront which is Erik Rose's extraction of the traceback stuff from nose-progressive.

How to install testing goodness:

```
pip install nose-progressive
pip install django-nose
```

zope.testing package is pretty well-decoupled from the rest of Zope and easy to use with non-zope stuff. Also Twisted's Trial test runner - someone noted that it's nice but it doesn't work with Django because Django is broken and wanted to sprint on it.

David Cramer of Disqus – nose-bleed and nose-quickunit

Someone mentioned that it's nice to set up editor keybindings that run the tests for just the file you're editing. Another way is to have something like autonose that automatically runs your tests for you.

MySQL sucks at everything; wants to switch to Postgres at Votizen.

CHAPTER 4

Fake It Til You Make It: Unit Testing Patterns With Mocks And Fakes by Brian K. Jones

Presenters: Brian K. Jones

PyCon 2012 presentation page: https://us.pycon.org/2012/schedule/presentation/336/

Slides:

Video: https://www.youtube.com/watch?v=hvPYuqzTPIk&list=PLBC82890EA0228306&index=13&feature=plpp_video

Video running time: 49:33

4.1 Your Speaker

(00:14)

- PSF member since 2011
- Creator: Python Magazine
- co-author: Linux Server Hacks, vol. 2 and upcoming Python Cookbook 3rd edition
- github.com/bkjones
- jonesy on freenode (join #python-testing!)
- protocolostomy.com

4.2 What's covered

(01:17)

- Definitions
- Patterns
- Tools

4.3 What's not covered

(01:56)

- Intro to the `unittest` module
- Sales pitch for doing testing at all

4.4 What is a "unit test"?

(02:38)

- A test that exercises a very small amount of code which is completely isolated from any and all dependencies, external or internal
- Granular – only testing a small piece of code
- Isolated
- Localized – tells you exactly where your bug is

4.5 When it is no longer a unit test?

(03:38)

- When something else in the larger system, besides the code under test, has to work.

4.6 What is it then?

(03:48)

- If multiple components of the same class or system are tested, and the test seeks to determine that these components interact in a predictable way, it's an integration test.
- If the entire system is being tested to insure compliance with a spec of some kind, it's an acceptance test.

4.7 What is "coverage"?

(04:15)

- A loaded term
- Often generically referred to as "code coverage".
- What you really want is "condition coverage", "case coverage", "logic coverage", "decision-point coverage", etc.

4.8 Use coverage.py

(05:15)

http://nedbatchelder.com/code/coverage/

- Integrates well with nosetests
- Super easy to use

- Can report on branch coverage using `--branch`
  - http://nedbatchelder.com/code/coverage/branch.html
- Can be used easily with tox, by itself or from within nosetests
- Nice work! Thanks, Ned!

4.9 Speaking of nosetests

(07:04)

- Nose is a great test discovery tool
- cd to your test directory and run "nosetests". Rejoice.
- `nosetests --with-coverage` (coverage.py is a nose plugin!)
- Has a super nice HTML report that highlights covered/uncovered lines
- Integrates well with Jenkins to present html and provide pretty graphs (managers like those):
  - http://pypi.python.org/pypi/nosexcover

4.10 Why unit tests?

(07:50)

- They're fast
- They're simple
- They provide greater localization of problems than other types of tests often can

4.11 Unit tests aren't enough

(10:13)

Unit tests don't test integration - that the parts of the system all work together

4.11.1 A problem

(10:54)

(Sample Python code from PyRabbit)

4.11.2 A solution

(11:55)

An integration test

(12:05)

(12:26) "Reality distortion field"

4.12 Mock is cool. Use it.

(12:40)

- http://mock.readthedocs.org/en/latest/index.html
- It patches all the things
- Often used as a "spy" library (technically)
- I use it so I don't have to create my own mocking classes.
- Action->Assertion > Record->Replay
  - Action->Assertion is closer to how developers tend to think about their code

4.12.1 Mock handles harder stuff

(13:54)

...

4.13 Diagram

(15:10)

...

4.14 More testable code

(17:00)

Why there is resistance to adopting unit testing

Requires deeper knowledge of code

Requires special techniques

You have to invest time to learn it

A lot of managers, especially with large code bases, will not want to make that investment

However, unit testing will improve the design of your code.

4.14.1 Low-hanging fruit

(17:58)

- Limit the scope of responsibility
- Create local wrappers
- Deduplicate the code

4.14.2 An example

(19:52)

`get_path` method

(21:20)

(22:15)

4.15 Practical patterns Part 1: A datetime abstraction library

(23:27)

4.16 Practical patterns Part 2: A REST client module

(28:54)

4.16.1 pyrabbit

- It's a cient for RabbitMQ's REST Management API
- ~250 lines of executable code
- ~200 lines of unit test code
- Uses `httplib2` to talk to the server
- Tests pass with Python 2.6, 2.7, and 3.2
- Use tox

4.16.2 tox is cool

(30:06)

http://tox.testrun.org/

```
[tox]
envlist = py26,py27,py32

[testenv]
deps =
    nose
    httplib2
    mock
changedir = tests
commands = nosetests []

[testenv:py26]
deps =
    nose
    httplib2
    mock
    unittest2
```

```
changedir = tests
commands = nosetests []
```

4.17 Other tricks

(38:19)

4.17.1 Mocking stdout

(38:27)

```
outstream = StringIO()

with patch('sys.stdout', new=outstream) as out:
    ...
    actual_out = out.getvalue()
```

4.17.2 Testing decorated functions

(39:48)

4.18 We've covered

(41:43)

- What's a unit test? Why are they cool? How can I make some?
- Are unit tests all I need?
- What's Mock ? Why is it cool? How can I use it?
- Use tox! Use nosetests! Use coverage.py!
- Testing a simple one-module date manipulation library
- Testing a REST API client library
- And more!

4.19 Questions?

(42:12)

- Mock is going to be in the Python Standard Library in Python 3.3 as `unittest.mock`.
- Question: How to organize integration tests?
- Question: When do you find doctests useful?
- Question: Design for testability (e.g.: dependency injection) vs. monkey-patching
  - Dependency injection is something that you need a team to be bought into

- Comment (from *Gary Bernhardt*). Decorators make testing harder because they couple things together at compile-time. The solution is dependency injection.

CHAPTER 5

Throwing Together Distributed Services With Gevent (Ginkgo) by Jeff Lindsay from Twilio

Presenter: Jeff Lindsay (http://progrium.com/) (@progrium)

PyCon 2012 presentation page: https://us.pycon.org/2012/schedule/presentation/288/

Tutorial: https://github.com/progrium/ginkgotutorial

Slides: http://dl.dropbox.com/u/2096290/GinkgoPyCon.pdf

Video: http://pyvideo.org/video/642/throwing-together-distributed-services-with-geven

Track: V

Description

In this talk we learn how to throw together a distributed system using gevent and a simple framework called Ginkgo (formerly gservice). We'll go from nothing to a distributed messaging system ready for production deployment based on experiences building scalable, distributed systems at Twilio.

Abstract

As some have found, gevent is one of the best kept secrets of Python. It gives you fast, evented network programming without messes of callbacks, code that is more Pythonic, and lets you use most regular Python networking libraries and protocol implementations. Now, let's build on this.

In this talk we learn how to throw together distributed services using gevent and a simple framework called Ginkgo (formerly gservice). We'll go from nothing to a distributed messaging system based on experiences building scalable, distributed systems at Twilio.

This talk will be full of code, live coding, and real production applications with guest appearances by other fun technologies like ZeroMQ, WebSocket, and Doozer.

Jeff works at Twilio. Twilio uses a service oriented architecture. Twilio as a high level service is made up of many subservices - sms, voice, client. Each is made up of other services. Mostly written in tornado and gevent. Going to use a framework called Ginkgo.

- services are nested modules that can start, stop and reload
- going to build a scalable gateway around a self-organizing messaging system.
- first we'll build a simple number client
- next a pubsub service
- then combine both into a gateway service
- and finally make it all distributed

5.1 Number Server

- a basic gevent StreamServer
- for every connection generate a random number, sleep for 60 seconds
- configuration is a python file
- ginkgo has start / stop services
- start the number server in the background

5.2 Number Client

- spawns a greenlet relative to your service
- makes services self-contained
- gets the numbers from the server and puts it in a queue

5.3 PUBSub

- uses httpstreamer
- subscription wraps a queue
- posts are publishes, gets are subscribes

5.4 MessageHub

- hub is now responsible for managing subscriptions

If you're in a loop that doesn't use IO, run `gevent.sleep(0)` to make sure it yields.

Code for the example is up here: https://github.com/progrium/ginkgotutorial

- Ginkgo - https://github.com/progrium/ginkgo
- Ginkgo tutorial - https://github.com/progrium/ginkgotutorial
- Notes from Andrew Schoen - http://readthedocs.org/docs/andrew-schoen-pycon-2012-notes/en/latest/friday/session_5.html

CHAPTER 6

Django Templating: More Than Just Blocks by Christine Cheung

Presenter: Christine Cheung (http://www.xtine.net/) (@plaidxtine)

PyCon 2012 presentation page: https://us.pycon.org/2012/schedule/presentation/80/

Slides: http://speakerdeck.com/u/xtine/p/django-templating-more-than-just-blocks

Video: http://pyvideo.org/video/697/django-templating-more-than-just-blocks

Outline

- Intro to Templating
- Effective Use of Built-In Tags
- Extending Templates
- Template Loading
- What's New

6.1 Intro to Templating

6.1.1 Django Templating 101

- This is the End User Experience
- Balance between power and ease
- Design

6.1.2 Helpful tools

- Django template theme for your IDE
  - syntax highlighting, autocompletion
- django-debug-toolbar
  - template paths, session variables
- Print out tag/filter reference guide
  - http://media.revsys.com/images/django-1.4-cheatsheet.pdf
  - https://docs.djangoproject.com/en/dev/ref/templates/builtins/

6.1.3 Folder and file structure

- Keep templates in one place

6.1.4 Style guide

- Think PEP 8 coding conventions
  - consistent spacing
- `{% load %}` all template tags up top
- try `{# comment #}` rather than `<!-- this -->`
  - also, `{% comment %}{% endcomment %}`

6.2 Effective Use of Built-In Tags

6.2.1 The Basics

- Start with `base.html`

```
<!doctype html>
<head>
    <title>{% block title %}demo{% endblock title %}</title>
</head>
<body>

{% block content %}{% endblock content %}

</body>
</html>
```

- Then have pages inherit from it

```
{% extends "base.html" %}

{% block title %}the foo page{% endblock title %}

{% block content %}
    <div id="foo">
        this is a bar.
    </div>
{% endblock content %}
```

6.2.2 Common blocks

I use these in practically every project:

- title
- meta_tags, robots,
- extra_head (CSS, etc.),
- content

- extra_js (so that JavaScript can go at bottom of page which improves page-load time)

6.2.3 Block practices

- End your block structures
  - `{% block title %}foo{% endblock title %}`
  - instead of `{%block title %}foo{% endblock %}`
- Can't repeat blocks
  - however: context processor, include, custom template tag
- Don't "over block"

6.2.4 Including templates

- `{% include "snippet.html" %}`
  - great for repeating template segments
- try not to include in an include – gets confusing

6.2.5 Variables

- Tend to be objects passed from a view
  - *Modify* objects with **filters**
    - `{{ variable | lower }}`
  - *Loop* through etc. using **tags**
    - `{% if variable %}foo{% else %}bar{% endif %}`
    - `{% for entry in blog_entries %}<h2>{{ entry.title }}</h2><p>{{ entry.body }}</p>{% endfor %}`
  - You can also create your own filters and tags (see Django docs on custom template tags and filters)

6.2.6 Security

By default, Django's security is rather solid on the template side of things...

- but if you use **safe** or `{% autoescape %}`
  - *** make sure you sanitize the data! ***

6.2.7 URLs

Name `{% url %}` tags as much as possible

- define URL patterns in `urls.py`
  - `url(r'^foo/$', foo, name="foo"),`
  - `<a href="{% url "foo" %}">foo</a>`

```
{{ STATIC_URL }}css/style.css
```

- Not `/static/css/style.css`

6.2.8 Forms

For heavy form action, take a look at:

- django-floppyforms (HTML 5)
- django-crispy-forms (used to be django-uni-form)

```
{% include form.html %}
```

- `as_ul` (docs) makes more sense than `as_p` or `as_table`

6.2.9 More than one way

There are multiple ways to accomplish the same task.

No ultimately right or wrong way

- use what suits you or your team

An example

The long way:

```
{% if foo.bar %}
    {{ foo.bar }}
{% else %}
    {{ foo.baz }}
{% endif %}
```

or the shorter way:

```
{% firstof foo.bar foo.baz %}
```

6.3 Extending templates

6.3.1 Custom Tags and Filters

Given that we have template tags in `demo/templatetags/demo_utils.py`

```
{% load demo_utils %}
```

6.3.2 Making a Custom Filter

```
from django import template
register = template.Library()

@register.filter(name='remove')

def cut(value, argument):
```

```
    # remove passed arguments from value
    return value.replace(argument, '')

{{ foo|remove:'bar' }}

@register.filter
def lower(value):
    # lowercased value with no passed arguments
    return value.lower()

{{ foo|lower }}
```

6.3.3 Making a Custom Tag

Tags are a bit more complex

- two steps: compiling and rendering

Decide its purpose

- but start simple

better to have many tags that do many things rather than one tag that does many things

6.3.4 A Simple Example

```
<p>
    It is now
    {% current_time "%Y-%m-%d %I:%M %p" %}
</p>
```

6.3.5 Simple Tag

```
from django import template

register = template.Library()

@register.simple_tag
def current_time(format_string):
    try:
        return datetime.datetime.now().strftime(str(format_string))
    except UnicodeEncodeError:
        return 'oh noes current time borked'
```

6.3.6 Nodes and Stuff

```
import datetime
from django import template
register = template.Library()

@register.tag(name="current_time")

def do_current_time(parser, token):
```

```
    try:
        tag_name, format_string = token.split_contents()
    except ValueError:
        msg = '%r tag requires a single argument' % token.split_contents()[0]
        raise template.TemplateSyntaxError(msg)


class CurrentTimeNode(template.Node):
    def __init__(self, format_string):
        self.format_string = str(format_string)

    def render(self, context):
        now = datetime.datetime.now()
        return now.strftime(self.format_string)
```

6.3.7 Easier Template Tag Creation

django-templatetag-sugar

- makes it simple to define syntax for a tag

django-classy-tags

- class-based template tags
- extensible argument parse for less boilerplate

6.3.8 DO NOT!!!

Do not write a template tag that runs logic or at worst, even run Python from a custom tag

- it defeats purpose of a templating language
- dangerous
- difficult to support

6.4 Loading templates

6.4.1 Template loading logic

Use cases

`TEMPLATE_LOADERS` setting (Django docs)

```
from django.conf import settings
from django.template import TemplateDoesNotExist

def load_template_source(template_name, template_dirs=None):
    for filepath in get_template_sources(template_name, template_dirs):
        try:
            # load in some templates yo
        except IOError:
            pass

    raise TemplateDoesNotExist(template_name)
```

6.4.2 Replacing the templating engine

You can replace the built in templating engine

- Jinja2, Mako, Cheetah, etc. (Jinja probably the most popular)

But why?

- More familiar with another templating language
- Performance boost
- Different logic control and handling

but you risk "frankensteining" your project.

6.4.3 Jinja2 and Django and You

Pros

- functions callable from templates - don't have to write tags and filters
- loop controls - more powerful flow control
- multiple filter arguments
- slight performance increase

Cons

- more dependencies and overhead
- extra time spent on development and support
- risk putting too much logic in templates
- minimal speed increase

6.4.4 Speeding Up Templates

Cache template loader

django-template-preprocessor

- compiles template files

django-pancake

- from Adrian Holovaty (one of the creators of Django)
- flattens template files

but also remember other bottlenecks...

6.4.5 New in Django 1.4

Django 1.4 release notes

Custom project and app templates

- startapp/startproject – template (docs on Django 1.4 custom project and app templates)
- combine with your favorite boilerplate

Else if

- `{% elif %}` (docs on minor features)

6.5 Questions

Questions????

CHAPTER 7

Django Forms Deep Dive - Nathan R. Yergler from EventBrite

Presenter: Nathan Yergler (http://yergler.net/) (@nyergler) from Eventbrite

PyCon 2012 presentation page: https://us.pycon.org/2012/schedule/presentation/420/

Slides: http://yergler.net/2012/pycon-forms/

Video: http://pyvideo.org/video/698/django-form-processing-deep-dive

7.1 Form Basics

7.1.1 Forms in context

Views	Convert request to response
Forms	Convert input to Python objects (input is not necessarily HTML)
Models	Data and business logic

7.1.2 Defining Forms

Forms are composed of fields, which have a widget.

```
from django.utils.translation import gettext_lazy as _
from django import forms

class ContactForm(forms.Form):

    name = forms.CharField(label=_("Your Name"),
        max_length=255,
        widget=forms.TextInput,
    )

    email = forms.EmailField(label=_("Email address"))
```

Form API: https://docs.djangoproject.com/en/dev/ref/forms/api/

7.1.3 Instantiating a Form: unbound forms vs. bound forms

Unbound forms don't have data associated with them, but they can be rendered:

```
form = ContactForm()
```

Bound forms have specific data associated, which can be validated:

```
form = ContactForm(data=request.POST, files=request.FILES)
```

7.1.4 Accessing fields

Two ways:

- `form.fields['name']` returns the `Field` object
- `form['name']` returns a `BoundField`

7.1.5 Initial data

```
form = ContactForm(
    initial={
        'name': 'First and Last Name',
    }
)
```

```
>>> form['name'].value()
'First and Last Name'
```

7.2 Validation

7.2.1 Validating the form

- Only bound forms can be validated
- Calling `form.is_valid()` triggers validation if needed
- Validated, cleaned data is stored in `form.cleaned_data`
- Calling `form.full_clean()` performs the full cycle

7.2.2 Field validation

- Three phases for fields: To Python, Validation, and Cleaning
- If validation raises an Error, cleanning is skipped
- Validators are callables that can raise a `ValidationError`
- Django includes generic ones for some common tasks
- Examples: URL, Min/Max Value, Min/Max Length, Regex, Email, etc.

7.2.3 Field cleaning

- `.clean_fieldname()` method is called after validators
- Input has already been converted to Python objects
- Methods can still raise `ValidationError`
- Methods *must* return the cleaned value

7.2.4 .clean_email() example

```
class ContactForm(forms.Form):
    name = forms.CharField(
        label=_("Name"),
        max_length=255,
    )

    email = forms.EmailField(
        label=_("Email address"),
    )

    def clean_email(self):
        if (self.cleaned_data.get('email', '').endswith('hotmail.com')):
            raise ValidationError("Invalid email address.")

        return self.cleaned_data.get('email', '')
```

7.2.5 Form validation

- `.clean()` performs cross-field validation - Example: Check that `email` and `confirm_email` fields match
- Called even if errors were raised by Fields
- *Must* return the cleaned data dictionary
- `ValidationError``s raised by ``.clean()` will be grouped in `form.non_field_errors()` by default

7.2.6 .clean() example

Example of cross-field validation: Check that `email` and `confirm_email` fields match:

```
class ContactForm(forms.Form):
    name = forms.CharField(
        label=_("Name"),
        max_length=255,
    )

    email = forms.EmailField(label=_("Email address"))
    confirm_email = forms.EmailField(label=_("Confirm"))

    def clean(self):
        if (self.cleaned_data.get('email') !=
            self.cleaned_data.get('confirm_email')):
```

```
            raise ValidationError("Email addresses do not match.")

        return self.cleaned_data
```

7.2.7 Initial != Default Data

- Initial data is used as a starting point
- It does not automatically propagate to `cleaned_data`
- Defaults for non-required fields should be specified when accessing the dict:

```
self.cleaned_data.get('name', 'default')
```

7.2.8 Tracking changes

- Forms use initial data to track change fields
- `form.has_changed()`
- `form.changed_fields`
- Fields can render a hidden input with the initial value, as well:

```
>>> changed_date = forms.DateField(show_hidden_initial=True)
>>> print form['changed_date']
'<input type="text" name="changed_date" id="id_changed_date" /><input type="hidden" name="initial-cha
```

7.3 Testing

Not clear whether they're unit tests or functional tests, etc. but nonetheless useful

7.3.1 Testing forms

- Remember what Forms are for
- Testing strategies
  - Initial states
  - Field validation
  - Final state of `cleaned_data`

7.3.2 Unit tests

```
import unittest

class FormTests(unittest.TestCase):
    def test_validation(self):
        form_data = {
            'name': 'X' * 300,
        }
```

```
        form = ContactForm(data=form_data)
        self.assertFalse(form.is_valid())
```

7.3.3 Test Data

Eventbrite just released a library on GitHub called rebar – https://github.com/eventbrite/rebar

```
from rebar.testing import flatten_to_dict

form_data = flatten_to_dict(ContactForm())
form_data.update({
        'name': 'X' * 300,
    })
form = ContactForm(data=form_data)
assert(not form.is_valid())
```

7.4 Rendering Forms

7.4.1 Idiomatic form usage

(Plug for class-based views....)

```
from django.views.generic.edit import FormMixin, ProcessFormView

class ContactView(FormMixin, ProcessFormView):
    form_class = ContactForm
    success_url = '/contact/sent'

    def get_form_kwargs(self):
        return super(ContactView, self).get_form_kwargs()
```

7.4.2 Form Output

Three primary "whole-form" output modes:

- `form.as_p()`
- `form.as_ul()`
- `form.as_table()`

```
<tr><th><label for="id_name">Name:</label></th>
  <td><input id="id_name" type="text" name="name" maxlength="255" /></td></tr>
<tr><th><label for="id_email">Email:</label></th>
  <td><input id="id_email" type="text" name="email" maxlength="Email address" /></td></tr>
<tr><th><label for="id_confirm_email">Confirm email:</label></th>
  <td><input id="id_confirm_email" type="text" name="confirm_email" maxlength="Confirm" /></td></tr>
```

7.4.3 Controlling form output

```
{% for field in form %}
{{ field.label_tag }}: {{ field }}
{{ field.errors }}
{% endfor %}
{{ field.non_form_errors }}
```

Additional rendering properties:

- `field.label`
- `field.label_tag`
- `field.html_id`
- `field.help_text`

7.4.4 Customizing rendering

http://yergler.net/2012/pycon-forms/slides/forms/#26

Libraries that make some of this stuff easier:

- django-crispy-forms
- django-form-utils

You can specify additional attributes for widgets as part of the form definition.

```
class ContactForm(forms.Form):
    name = forms.CharField(
        max_length=255,
        widget=forms.Textarea(
            attrs={'class': 'custom'},
        ),
    )
```

You can also specify form-wide CSS classes to add for error and required states.

```
class ContactForm(forms.Form):
    error_css_class = 'error'
    required_css_class = 'required'
    ...
```

7.4.5 Customizing error messages

Built-in validators have default error messages

```
>>> generic = forms.CharField()
>>> generic.clean('')
Traceback (most recent call last):
  ...
ValidationError: [u'This field is required.']
```

`error_messages` lets you customize those messages

```
>>> name = forms.CharField(
...   error_messages={'required': 'Please enter your name'})
>>> name.clean('')
Traceback (most recent call last):
```

```
  ...
ValidationError: [u'Please enter your name']
```

7.4.6 Error Class

- `ValidationError` exceptions raised are wrapped in a class
- This class controls HTML formatting
- By default, `ErrorList` is used; outputs as `<ul>`
- Specify the `error_class` kwarg when constructing the form to override

7.4.7 Error Class

```
from django.forms.util import ErrorList

class ParagraphErrorList(ErrorList):
    def __unicode__(self):
        return self.as_paragraphs()

    def as_paragraphs(self):
        return "<p>%s</p>" % (
            ",".join(e for e in self.errors)
        )

form = ContactForm(data=form_data, error_class=ParagraphErrorList)
```

7.4.8 Multiple Forms

Avoid potential name collisions with prefix:

```
contact_form = ContactForm(prefix='contact')
```

Adds the prefix to HTML name and ID:

```
<tr><th><label for="id_contact-name">Name:</label></th>
  <td><input id="id_contact-name" type="text" name="contact-name"
       maxlength="255" /></td></tr>
<tr><th><label for="id_contact-email">Email:</label></th>
  <td><input id="id_contact-email" type="text" name="contact-email"
       maxlength="Email address" /></td></tr>
<tr><th><label for="id_contact-confirm_email">Confirm
     email:</label></th>
  <td><input id="id_contact-confirm_email" type="text"
       name="contact-confirm_email" maxlength="Confirm" /></td></tr>
```

7.5 Forms for Models

7.5.1 Model Forms

- ModelForms map a Model to a Form

- Validation includes Model validators by default
- Supports creating and editing instances
- Key differences from forms
  - A field for the primary key (usually `id`)
  - `save()` method
  - `.instance` property

7.5.2 Model Form Example

```
from django.db import models
from django import forms

class Contact(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField()
    notes = models.TextField()

class ContactForm(forms.ModelForm):
    class Meta:
        model = Contact
        ...
```

7.5.3 Limiting Fields

- You don't need to expose all the fields in your form
- You can either specify fields to expose, or fields to exclude

```
class ContactForm(forms.ModelForm):
    class Meta:
        model = Contact
        fields = ('name', 'email',)

class ContactForm(forms.ModelForm):
    class Meta:
        model = Contact
        exclude = ('notes',)
```

7.5.4 Overriding Fields

- Django will generate fields and widgets based on the model
- These can be overridden as well

```
class ContactForm(forms.ModelForm):
    name = forms.CharField(widget=forms.TextInput)

    class Meta:
        model = Contact
```

7.5.5 Instantiating Model Forms

```
model_form = ContactForm()

model_form = ContactForm(
    instance=Contact.objects.get(id=2)
    )
```

7.5.6 `ModelForm.is_valid()`

- ModelForms have an additional method, `_post_clean()`
- Sets cleaned fields on the Model instance
- Called *regardless* of whether the form is valid

7.5.7 Testing

```
class ModelFormTests(unittest.TestCase):
    def test_validation(self):
        form_data = {
            'name': 'Test Name',
        }

        form = ContactForm(data=form_data)
        self.assert_(form.is_valid())
        self.assertEqual(form.instance.name, 'Test Name')

        form.save()

        self.assertEqual(
            Contact.objects.get(id=form.instance.id).name,
            'Test Name'
        )
```

7.6 Form sets

7.6.1 Form sets

- Handles multiple copies of the same form
- Adds a unique prefix to each form:

```
form-1-name
```

- Support for insertion, deletion, and reordering

7.6.2 Defining form sets

```
from django.forms import formsets

ContactFormSet = formsets.formset_factory(
```

```
    ContactForm,
)

formset = ContactFormSet(data=request.POST)
```

Factory kwargs:

- `can_delete`
- `extra`
- `max_num`

7.6.3 Using form sets

```
<form action="" method="POST">
{% formset %}
</form>
```

or more control over output:

```
<form action="." method="POST">
{% formset.management_form %}
{% for form in formset %}
   {% form %}
{% endfor %}
</form>
```

7.6.4 Management form

- `formset.management_form` provides fields for tracking the member forms
  - `TOTAL_FORMS`
  - `INITIAL_FORMS`
  - `MAX_NUM_FORMS`
- Management form data **must** be present to validate a Form Set

7.6.5 `formset.is_valid()`

- Performs validation on each member form
- Calls `.clean()` method on the FormSet
- `formset.clean()` can be overridden to validate across Forms
- Errors raised are collected in `formset.non_form_errors`

7.6.6 FormSet.clean()

```
from django.forms import formsets

class BaseContactFormSet(formsets.BaseFormSet):
    def clean(self):
```

```
        names = []
        for form in self.forms:
            if form.cleaned_data.get('name') in names:
                raise ValidationError()
            names.append(form.cleaned_data.get('name'))

ContactFormSet = formsets.formset_factory(
    ContactForm,
    formset=BaseContactFormSet
)
```

7.6.7 Testing

- FormSets can be tested in the same ways as Forms
- Helpers to generate test form data:
  - `flatten_to_dict` works with FormSets just like Forms
  - `empty_form_data` takes a FormSet and index, returns a dict of data for an empty form

```
from rebar.testing import flatten_to_dict, empty_form_data

formset = ContactFormSet()
form_data = flatten_to_dict(formset)
form_data.update(
    empty_form_data(formset, len(formset))
)
```

7.6.8 Model Formsets

- ModelFormSets:FormSets :: ModelForms:Forms
- `queryset` argument specifies intial set of objects
- `.save()` returns the list of saved instances
- If `can_delete` is `True`, `.save()` also deletes the models flagged for deletion

7.7 Advanced & Miscellaneous Detritus

- Django's i18n/l10n framework supports localized input formats
- For example: 10,00 vs. 10.00

Enable in `settings.py`:

```
USE_L10N = True
USE_THOUSAND_SEPARATOR = True # optional
```

7.7.1 Localizing fields example

And then use the `localize` kwarg

```
>>> from django import forms
>>> class DateForm(forms.Form):
...     pycon_ends = forms.DateField(localize=True)

>>> DateForm({'pycon_ends': '3/15/2012'}).is_valid()
True
>>> DateForm({'pycon_ends': '15/3/2012'}).is_valid()
False

>>> from django.utils import translation
>>> translation.activate('en_GB')
>>> DateForm({'pycon_ends':'15/3/2012'}).is_valid()
True
```

7.7.2 Dynamic forms

- Declarative syntax is just sugar
- Forms use a metaclass to populate `form.fields`
- After `__init__` finishes, you can manipulate `form.fields` without impacting other instances

7.7.3 State validators

- Validation isn't necessarily all or nothing
- State Validators define validation for specific states, on top of basic validation
- Your application can take action based on whether the form is valid, or valid for a particular state

7.7.4 State validators example

```
from django import forms
from rebar.validators import StateValidator, StateValidatorFormMixin

class PublishValidator(StateValidator):
    validators = {
        'title': lambda x: bool(x),
     }

class EventForm(StateValidatorFormMixin, forms.Form):
    state_validators = {
        'publish': PublishValidator,
    }
    title = forms.CharField(required=False)
```

Using it:

```
>>> form = EventForm(data={})
>>> form.is_valid()
True
>>> form.is_valid('publish')
False
>>> form.errors('publish')
{'title': 'This field is required'}
```

7.7.5 The End

http://yergler.net/2012/pycon-forms

CHAPTER 8

Testing and Django by Carl Meyer

Presenter: Carl J. Meyer (http://oddbird.net/) (@carljm)

PyCon 2012 presentation page: https://us.pycon.org/2012/schedule/presentation/412/

Slides: http://carljm.github.com/django-testing-slides/

Video: http://pyvideo.org/video/699/testing-and-django

Video running time: 47:15

8.1 The presenter

Name: Carl J. Meyer

Email: carl@oddbird.net

Slides: https://github.com/carljm/django-testing-slides [View slides] (The slides use Scott Chacon's ShowOff)

Code from slides: https://github.com/carljm/django-testing-slides/code

8.2 Upfront

(00:30 / 47:15 into the video)

- The slides are online. The code is online.
- There are going to be a lot of opinions in this talk about better ways to do testing.
- Not a certified expert.
- Opinions do not reflect the opinions of the Django developers.
- We'll be using some features in the new Django 1.4 release candidate (Django 1.4 release notes).

8.3 Let's start a project

(01:18 / 47:15 into the video)

```
$ django-admin.py startproject testing .
## The period is a new Django 1.4 feature: the ability to start a new
## project in the current directory instead of creating a new subdirectory.
## (see https://code.djangoproject.com/ticket/17042)

$ sed -i -e 's/backends\./backends.sqlite3/;' testing/settings.py
## Don't normally edit files using sed; it goes on a slide better than
## opening up an editor. Setting database backend to sqlite3.

$ ./manage.py test
...
[snip]
----------------------------------------------------------------------
Ran 412 tests in 14.235s

FAILED (errors=2, skipped=1)
Destroying test database for alias 'default'...
```

This is the default Django testing experience. That ran **412** tests and took **14** seconds to do it. And that's using an in-memory sqlite database, which is the *fastest* way you can run Django's tests.

What's wrong here?

- We'll never get those 14 seconds back. That's a long time to wait, given we haven't started writing code!
- 2 failures in Django 1.4 RC - Those are due to a bug in Django that two tests in `django.contrib.auth` are insufficiently isolated from the project settings. They assume the existence of two databases configured in the settings.

Note from Marc on 2012-03-17: I just downloaded Django-1.4c2 and tried to reproduce the 2 failures, but I couldn't. So it looks like this issue was fixed.

8.3.1 Not all apps are created equal

(03:26 / 47:15 into the video)

There is a difference between apps that **you** create in your project and apps built into Django or third-party apps that you install.

For third-party apps, the only times I care about their tests is when I decide to use them or upgrade them. In between those times, I don't care and running their tests is a waste of time.

Some might argue that we need integration tests to verify that projects integrate properly with things like `django.contrib.auth`, but any tests written in `django.contrib.auth` that are not isolated are likely to fail, because there are so many ways to integrate. And an isolated test is probably just purely testing `django.contrib.auth` and projects shouldn't have to care about that. In other words...

- Non-isolated tests break.
- Isolated tests are pointless to run.
- Integration tests should be written by the integrator.

This isn't a big problem. You can just do this:

```
./manage.py test just my apps please
```

You could even wrap the above up in a shell script and you're done.

There's another problem...

8.3.2 tests/__init__.py

(05:32 / 47:15 into the video)

```
from .test_forms import QuoteFormTest
from .test_models import (
    QuoteTest, SourceTest)
from .test_views import (
    AddQuoteTest, EditQuoteTest,
    ListQuotesTest
    )
```

Django insists that all of your tests live in a `tests` module for each app.

If you split your tests into a bunch of separate modules (which you probably should if you're writing as many tests as you should), you have to import your submodules so *Django's test runner* can find it. This is 2012. That's ridiculous.

8.3.3 Django's test discovery

(06:10 / 47:15 into the video)

- Wastes my time with tests I don't care about.
- Requires app tests to be in a single module (resulting in boilerplate imports).
- Forces intermingling of tests and non-test code.

8.3.4 It's easy to change.

(06:50 / 47:15 into the video)

- *unittest2 test discovery* (You could also use nose, py.test, etc...)
- `TEST_RUNNER` setting

This is how much code it takes to make Django's test discovery good:

```
class DiscoveryRunner(DjangoTestSuiteRunner):
    """A test suite runner using unittest2 discovery."""

    def build_suite(self, test_labels, extra_tests=None, **kwargs):
        suite = None
        discovery_root = settings.TEST_DISCOVERY_ROOT

        if test_labels:
            suite = defaultTestLoader.loadTestsFromNames(
                test_labels)

        if suite is None:
            suite = defaultTestLoader.discover(
                discovery_root,
                top_level_dir=settings.BASE_PATH,
                )

        if extra_tests:
            for test in extra_tests:
                suite.addTest(test)

        return reorder_suite(suite, (TestCase,))
```

settings.py:

```
import os.path
BASE_PATH = os.path.dirname(os.path.dirname(__file__))
TEST_DISCOVERY_ROOT = os.path.join(BASE_PATH, "tests")
TEST_RUNNER = "tests.runner.DiscoveryRunner"
```

(There's an updated version of this test runner.)

8.3.5 \o/

(07:59 / 47:15 into the video)

- Discovers tests wherever you want them.
- Doesn't run tests from external apps by default.
- Flexible specification of specific tests to run: Python dotted path to test module, not Django app label.
- `./manage.py test tests.quotes.test_views`

8.3.6 Maybe in 1.5?

(08:37 / 47:15 into the video)

- https://code.djangoproject.com/ticket/17365

8.3.7 Types of test

(08:59 / 47:15 into the video)

- "Much inferior restatement of *Gary Bernhardt's excellent 'Fast Test, Slow Test' talk* from yesterday"
- unit
- system/integration/functional

8.3.8 Unit tests

(09:34 / 47:15 into the video)

- Test one unit of code (a function or method) in something approaching isolation.
- Fast, focused (useful failures).
- Help you structure your code better.

TBD: Add some more detail here on what Carl said.

8.3.9 Integration tests

(10:35 / 47:15 into the video)

- Also very important.
- Test that the whole system works; catch regressions.
- Slow.

- Less useful failures. (Tell you something is broken, but takes longer to debug because it doesn't tell you where the problem is)
- Write fewer. Most people have too many of these. Django makes these easy with Django test client, but you shouldn't have too many of these.

8.4 Testing models

(11:46 / 47:15 into the video)

8.4.1 The database makes your tests slow.

- Try to write tests that don't hit it at all. (*Erik Rose's talk* had an excellent slide about the latency of L1 cache vs. L2 cache vs. memory vs. disk – disk accesses are **super** expensive)
- Separate db-independent model-layer functionality from db-dependent functionality.
  - Django doesn't make it easy to write tests that avoid hitting the database because the model layer encourages you to tie your models to the database.
  - So you need to do a bit of work here.
- But you'll still have a lot of tests that do.
- Mocking the database usually isn't worth it.
  - This is bound to fail. It's not a small and well-defined API so it's a lot of work.

8.4.2 A simple example of refactoring a model to separate out the db-independent functionality

(13:57 / 47:15 into the video)

Before:

```
class Thing(models.Model):
  def frobnicate(self):
    """Frobnicate and save the thing."""
    # ... do something complicated
    self.save()
```

There may be 20 different code paths before `self.save()`. If we test all of them, all of those tests will hit the database.

After:

```
def frobnicate_thing(thing):
  # ... do something complicated
  return thing

class Thing(models.Model):
  def frobnicate(self):
    """Frobnicate and save the thing."""
    frobnicate_thing(self)
    self.save()
```

Pull out all the code that does the state modification and complex logic and make it not talk to the database. And then **one** test that tests that it saves to the database.

8.4.3 django.test.TestCase

(15:22 / 47:15 into the video)

- Here come some boring slides. Don't have any problem with how Django does this stuff...
- Runs each test within a transaction.
- Rolls back the transaction at the end of the test.
- Monkeypatches transaction functions in your code to be no-ops.

This is nice because you don't have to have your tests truncate database tables or recreate database state.

8.4.4 TransactionTestCase

(15:58 / 47:15 into the video)

- Lets you test transactions in your code (doesn't wrap your tests in a transaction).
- Hash to flush every database table after every test.
- Makes your tests extra super bonus slow.
- You want to have as few of these as possible.

8.4.5 Fixtures

(16:25 / 47:15 into the video)

- Set up database state ahead of time.
- Currently, the Django documentation points you to do these with **fixtures** (see "Providing initial data for models").
- Example JSON fixture:

```
[
    {
        "pk": 4,
        "model": "auth.user",
        "fields": {
            "username": "manager",
            "first_name": "",
            "last_name": "",
            "is_active": true,
            "is_superuser": false,
            "is_staff": false,
            "last_login": "2012-02-06 15:06:44",
            ...
        }
    },
    ...
]
```

- **Don't do it.** If you've got them in your code, **burn them**.

8.4.6 Fixtures: Just say no.

(16:47 / 47:15 into the video)

[Applause :-)]

Probably the third talk that said this.

- Hard to maintain and update.
  - hand editing JSON => terrible
  - changing stuff in the db and then dumping them => terrible
  - if you're clever you can use the django-fixture-generator app but might as well just skip fixtures altogether.
- Increase test interdependence
  - too tempting to just throw things in the fixture and then the shared fixture couples tests together => **unnecessary coupling** between tests.
- Slow to load.
  - People tend to put too much stuff in fixtures and every test loads the fixture and incurs the cost. There are tricks that Erik Rose talked about yesterday which are useful for legacy code (e.g.: "Per-Class Fixtures", but if you're writing new code, just don't use fixtures.

8.4.7 Model factories!

(18:41 / 47:15 into the video)

It's hard to use just vanilla ORM to set up dependencies because models have dependencies so you end up writing a lot of stuff just to get one model to test. This is why people like fixtures. But model factories are a better solution.

An example of something that you could write yourself, with no special tools: a user profile model.

```
def create_profile(**kwargs):
    defaults = {
        "likes_cheese": True,
        "age": 32,
        "address": "3815 Brookside Dr",
    }
    defaults.update(kwargs)
    if "user" not in defaults:
        defaults["user"] = create_user()
    return Profile.objects.create(
        **defaults)
```

8.4.8 Using a factory

(20:02 / 47:15 into the video)

```
def test_can_vote(self):
    """A user 18 age+ can vote in the US."""
    profile = create_profile(age=18)
    self.assertTrue(profile.can_vote)
```

8.4.9 Or use factory_boy:

(21:22 / 47:15 into the video)

factory_boy

(a clone of Ruby's factory_girl)

```
class ProfileFactory(factory.Factory):
    FACTORY_FOR = Profile

    likes_cheese = True
    age = 32
    address = "3815 Brookside Dr"
    user = factory.SubFactory(UserFactory)

profile = ProfileFactory.create(
    age=18, user__username="carljm")
```

- `.create()` saves the model object to the database.
- `.build()` builds the model object but doesn't save it to the database.
- So you can be explicit in your tests about whether or not they require the database.

8.4.10 Why factories?

(22:24 / 47:15 into the video)

- Test data local to test code (explict).
  - Makes test clearer and easier to maintain. Nothing depending upon some distant fixture.
- Easy to maintain.
- Don't create any data you don't need for that test.
- Works great even for large/complex test data sets (helper functions).

8.4.11 Imposing no-DB discipline.

(23:22 / 47:15 into the video)

- Django makes it easy to not be clear whether or not you're talking to the database.
- It can be helpful to specify that a test doesn't hit the database and have my code yell at me if it does.
- For certain test cases.

(23:49 / 47:15 into the video)

Solution: Use Michael Foord's Mock library...

(Carl uses this library in every project. It is really useful for monkeypatching things for the sake of testing).

```
from django.utils.unittest import TestCase
import mock

cursor_wrapper = mock.Mock()
cursor_wrapper.side_effect = \
    RuntimeError("No touching the database!")
```

```
@mock.patch(
    "django.db.backends.util.CursorWrapper",
    cursor_wrapper)
class NoDBTestCase(TestCase):
    """Will blow up if you database."""
```

Carl made a minor point here about how we're using `django.utils.unittest.TestCase`, which is essentially the `TestCase` class from the Python `unittest2` module and we're not using ``django.test.TestCase`, which adds a bunch of Django-specific stuff for dealing with databases and transactions. The former is a little lighter, but it shouldn't make a big difference, because the Django stuff is smart about not doing database stuff if your test doesn't touch the database. There's a nice diagram in the Django docs showing the class relationships.

8.5 Testing views

(25:29 / 47:15 into the video)

8.5.1 *Unit* testing views is hard.

- Views have many collaborators / dependencies.
- Templates, database, middleware, url routing...
- Write less view code!

A common problem seen in Django projects is **too much view code**. Views know so much about your system, so it's tempting to put stuff in there because it's easy. Carl doesn't like to see view functions longer than 10-12 lines.

Views are hard to write unit tests for, because they are where everything else comes together and they are coupled to a lot of stuff.

One solution is to put less stuff in views since they're hard to test.

You might also consider testing views with functional tests rather than unit tests.

8.5.2 If you unit test views

(27:05 / 47:15 into the video)

- Use `RequestFactory` (link in Django docs).
  - An under-publicized but very useful class for generating fake `HttpRequest` objects to pass directly to view callables.
  - If you're using the Django test client, it is not a unit test. That goes through HTTP and thus is influenced by routing, middleware, etc.
- Call the view callable directly.
- Set up dependencies explicitly (e.g.: `request.user`, `request.session`)

8.5.3 `RequestFactory` example

(28:08 / 47:15 into the video)

A hypothetical example of posting to a `/locale/` view to change the locale.

```
def test_change_locale(self):
    """POST sets 'locale' key in session."""
    request = RequestFactory().post("/locale/", {"locale": "es-mx"})
    request.session = {}

    change_locale(request)

    self.assertEqual(request.session["locale"], "es-mx")
```

8.5.4 Or don't.

(29:14 / 47:15 into the video)

- Carl rarely unit tests views.
- Carl writes less view code, and covers it via functional tests.

8.5.5 Integration testing views

(29:39 / 47:15 into the video)

```
url = "/case/edit/{0}".format(case.pk)
step = case.steps.get()
response = self.client.post(url, {
    "product": case.product.id,
    "name": case.name,
    "description": case.description,
    "steps-TOTAL_FORMS": 2,
    "steps-INITIAL_FORMS": 1,
    "steps-MAX_NUM_FORMS": 3,
    "steps-0-step": step.step,
    "steps-0-expected": step.expected,
    "steps-1-step": "Click link.",
    "steps-1-expected": "Account active.",
    "status": case.status,
})
```

- Django test client is sort of in a sour spot between a system test and a unit test.
  - It's a bad system test because it's easy to break it with a simple template change.
  - It's a bad unit test because it's going through way too much code.

8.5.6 WebTest!

(31:16 / 47:15 into the video)

- WebTest is a library from Ian Bicking.
- WebTest knows a lot less about Django, which is a good thing.
- WebTest interacts with your application through WSGI, which is much closer to how your users will interact with your application.
- WebTest knows more about HTML.

```
url = "/case/edit/{0}".format(case.pk)
form = self.app.get(url).forms["case-form"]
form["steps-1-step"] = "Click link."
form["steps-1-expected"] = "Account active."

response = form.submit()
```

- WebTest parses the form HTML and can submit it like a browser would.
- Notice how much simpler the WebTest example is compared to the Django test client example. WebTest eliminates a lot of boilerplate.

8.5.7 The markup matters.

(32:51 / 47:15 into the video)

- If it can break, it should be tested.
- It can especially break forms.
- The output of your view is an HTTP response; the template + context is an implementation detail.

8.5.8 WebTest > django.test.Client

(34:12 / 47:15 into the video)

- System tests are easier and faster to write.
- Tests give you more confidence that the view works.
- There is also a project called django-webtest that tries to integrate WebTest with Django.

```
self.assertEqual(response.json, ["one", "two", "three"])

self.assertEqual(resp.html.find("a", title="Login").href, "/login/")
```

WebTest has some nice features:

- It parses JSON.
- It parses HTML (using BeautifulSoup or lxml).

8.6 In-browser testing

(34:54 / 47:15 into the video)

More and more functionality depends on both JS and server. Needs to be tested too.

8.6.1 Is easier than you think.

- Especially in Django 1.4.
- `pip install selenium`
- `LiveServerTestCase` (Django docs on LiveServerTestCase)
  - `LiveServerTestCase` runs the development server in a separate thread.

```
from django.test import LiveServerTestCase
from selenium.webdriver.firefox.webdriver import WebDriver

class MySeleniumTests(LiveServerTestCase):

    @classmethod
    def setUpClass(cls):
        cls.selenium = WebDriver()
        super(MySeleniumTests, cls).setUpClass()

    @classmethod
    def tearDownClass(cls):
        super(MySeleniumTests, cls).tearDownClass()
        cls.selenium.quit()

    def test_login(self):
        self.selenium.get("%s%s" % (self.live_server_url, "/login/"))
        username_input = self.selenium.find_element_by_name("username")
        username_input.send_keys("myuser")
        password_input = self.selenium.find_element_by_name("password")
        password_input.send_keys("secret")
```

8.7 What type of test to write?

(36:17 / 47:15 into the video)

Carl's rules of thumb:

- Write system tests for your views.
- Write Selenium tests for Ajax, other JS/server interactions.
- Write as few of the above 2 as possible.
- Write unit tests for everything else (not strict). - e.g.: when testing a `ModelForm`, might not bother to mock out the model.
- Test each case (code branch) where it occurs.
- One assert/action per test case method.
  - One assert is stricter.
  - danger of multiple asserts is that if one assert fails, you don't know whether the other ones would pass or fail.
- Very rough guidelines; what works for Carl. Not strict; e.g. tests for a `ModelForm` don't mock the model.
- You should really avoid multiple step tests – it's tempting but makes debugging tests harder.

8.7.1 Example of testing a view

(38:43 / 47:15 into the video)

```
def add_quote(request):
    if request.method == "POST":
        form = QuoteForm(request.POST)
        if form.is_valid():
```

```
            return redirect("quote_list")
    else:
        form = QuoteForm()

    return TemplateResponse(
        request,
        "add_quote.html",
        {"form": form},
        )
```

- This view should have 3 tests. Model/form special cases should be unit tested. And views shouldn't get much more complex.

8.8 Testing documentation

(40:00 / 47:15 into the video)

8.8.1 "We have always been at war with doctests"

- doctests let you put executable code examples in your docs and have the examples executed and verified (Django docs on doctests).
- Not entirely fair.
- Doctests are great.
- For testing documentation examples.

8.8.2 You have Sphinx docs.

(40:59 / 47:15 into the video)

- Right?

You have API code examples.

In your Sphinx docs.

You can add stuff to make sure that examples in Sphinx docs are tested. In any test file:

```
def load_tests(loader, tests, ignore):
    path = os.path.join(
        settings.BASE_PATH,
        "docs",
        "examples.rst",
        )

    tests.addTests(
        doctest.DocFileSuite(path))

    return tests
```

Your examples are tested!

- Please don't abuse this.

  - If you start treating these like tests for your code instead of documentation, then it will get complex and you will have bad documentation.

- Keep them documentation first.

8.8.3 `@override_settings(ALLOW_COMMENTS=True)`

(42:07 / 47:15 into the video)

Testing with a specific settings value - the anti-pattern:

```
def test_comments_allowed(self):
  old_allow = settings.ALLOW_COMMENTS
  settings.ALLOW_COMMENTS = True
  try:
    # ...
  finally:
    settings.ALLOW_COMMENTS = old_allow
```

A better way to do this would be using Michael Foord's Mock library.

In Django 1.4, using the `@override_settings` decorator (Django docs on override_settings):

```
@override_settings(ALLOW_COMMENTS=True)
def test_comments_allowed(self):
  # ...
```

8.9 Questions?

(42:51 / 47:15 into the video)

Note that the following questions and answers are not verbatim quotes. I have paraphrased and summarized.

Question: Any comments on the various Django nose modules that are around?

Carl: has used `django-nose` and is pretty happy with *unittest2 test discovery*. Django's `TEST_RUNNER` setting makes it pretty easy to swap in whatever test runner you like.

Question: Ever felt like you need to test the formset implementation?

Carl: If I were testing something in the formset, I would use a unit test for the formset class.

Question: More something that involves interaction between the formset and the view...or is this a code smell?

(They went back and forth a little bit and I didn't follow it too well).

Question: Thank you for your work on pip and virtualenv. I try not to repeat myself and create reusable apps. Any advice on how to test reusable apps, especially if I intend to put them on PyPI?

Carl: Don't make the mistake of putting the tests for your app in a `tests.py`, because then other people who use your app will be forced to run your tests when they don't really need to. Better to put your tests in another directory and then set up a test runner for your reusable app to run them for you.

CHAPTER 9

RESTful APIs with Tastypie by Daniel Lindsley

or How I learned to stop worrying and love the JSON

Presenter: Daniel Lindsley (http://www.toastdriven.com/) (@daniellindsley)

PyCon 2012 presentation page: https://us.pycon.org/2012/schedule/presentation/61/

Slides: http://speakerdeck.com/u/daniellindsley/p/restful-apis-with-tastypie

Video: http://pyvideo.org/video/673/restful-apis-with-tastypie

Video running time: 34:07

9.1 About the speaker

- **Daniel** Lindsley
- Consulting & OSS as **Toast Driven**
- Primary author of **Tastypie**

9.2 What is Tastypie?

- A **REST** framework for **Django**
- Designed for **extension**
- Supports both **Model** & **non-Model** data
- http://tastypieapi.org/

9.2.1 Philosophy

- Make good use of HTTP
  - Try to be "of the internet" & use the REST methods/status codes properly
- Graceful degradation
  - Try to keep backwards compatibility & give users a gradual upgrade path
- Flexible serialization
  - not everyone wants JSON

- Flexible EVERYTHING
  - Customizability is a core feature
- Data can round-trip
  - Anything you can GET, you should be able to POST/PUT
- Reasonable defaults
  - but easy to extend
- URIs everywhere!
  - Make HATEOAS a reality

9.2.2 HATEOAS

- "Hypermedia As The Engine Of Application State"
- Basically the user **shouldn't have to know anything** in advance
- All about explore-ability
- **Deep linking**
- http://en.wikipedia.org/wiki/HATEOAS

9.2.3 Tastypie

- Builds on top of Django & plays nicely with other apps
- Full `GET/POST/PUT/DELETE/PATCH` (`PATCH`? See **RFC 5789**)
- **Any** data source (not just Models)
- Designed to be extended
- Supports a variety of serialization formats:
  - JSON
  - XML
  - YAML
  - bplist
- Easy to add more
- HATEOAS by default (you'll see soon)
- Lots of hooks for customization
- Well-tested – about 80% coverage
- Decently documented

9.3 Installation

```
pip install django-tastypie
```

```
INSTALLED_APPS += ['tastypie']
```

```
$ manage.py syncdb
```

Done.

9.4 Let's add an API for `django.contrib.auth`

9.4.1 Setting it up

Set up your app:

```
# Assuming we're in your project directory...
$ cd <myapp>   # Substitute your app_name here
$ mkdir api
$ touch api/__init__.py
$ touch api/resources.py
# Done!
```

User Resource:

```
from django.contrib.auth.models import User
from tastypie.resources import ModelResource

class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()
```

URLconf:

```
# In your ROOT_URLCONF...
from tastypie.api import Api
from <myapp>.api.resources import UserResource
v1_api = Api()
v1_api.register(UserResource())

urlpatterns = patterns('',
    (r'^api/', include(v1_api.urls),
    # Then the usual...
)
```

9.4.2 Trying it out

Curl:	http://localhost:8000/api/v1/
Browser:	http://localhost:8000/api/v1/?format=json

- `/api/v1/` - A list of all available resources
- `/api/v1/user/` - A list of all users
- `/api/v1/user/2/` - A specific user

- `/api/v1/user/schema/` - A definition of what an individual user consists of
- `/api/v1/user/multiple/1;4;5/` - Get those three users as one request

All serialization formats available (provided lxml, PyYAML, and biplist are installed).

- `curl -H "Accept:  application/xml" http://localhost:8000/api/v1/user/`
- http://localhost:8000/api/v1/user/2/?format=yaml

Serialization format negotiated by either `Accepts` header or the `"?format=json" GET` param

Pagination by default

Everyone has full read-only `GET` access

What's not there? (Yet)

- Leaking sensitive information!
  - `email/password/is_staff/is_superuser`
- Ability to filter
- Authentication/Authorization
- Caching (disabled by default)
- Throttling (disabled by default)

9.4.3 Excluding fields

```
from django.contrib.auth.models import User
from tastypie.resources import ModelResource

class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()
        excludes = ['email', 'password', 'is_staff', 'is_superuser']
```

9.4.4 Add BASIC Auth

```
from django.contrib.auth.models import User
from tastypie.authentication import BasicAuthentication
from tastypie.resources import ModelResource

class UserResource(ModelResource):
    class Meta:
        # What was there before...
        authentication = BasicAuthentication()
```

9.4.5 Add filtering

```
from django.contrib.auth.models import User
from tastypie.authentication import BasicAuthentication
from tastypie.resources import ModelResource, ALL

class UserResource(ModelResource):
    class Meta:
```

```
        # What was there before...
        filtering = {
            'username': ALL,
            'date_joined': ['range', 'gt', 'gte', 'lt', 'lte'],
        }
```

- Using `GET` params, we can now filter out what we want.
- Examples:
  - `curl http://localhost:8000/api/v1/user/?username__startswith=a`
  - `curl http://localhost:8000/api/v1/user/?date_joined__gte=2011-12-01`

9.4.6 Add authorization

```
from django.contrib.auth.models import User
from tastypie.authentication import BasicAuthentication
from tastypie.authorization import DjangoAuthorization
from tastypie.resources import ModelResource

class UserResource(ModelResource):
    class Meta:
        # What was there before...
        authorization = DjangoAuthorization()
```

9.4.7 Add caching

```
from django.contrib.auth.models import User
from tastypie.authentication import BasicAuthentication
from tastypie.authorization import DjangoAuthorization
from tastypie.cache import SimpleCache
from tastypie.resources import ModelResource

class UserResource(ModelResource):
    class Meta:
        # What was there before...
        cache = SimpleCache()
```

9.4.8 Add throttling

```
from django.contrib.auth.models import User
from tastypie.authentication import BasicAuthentication
from tastypie.authorization import DjangoAuthorization
from tastypie.cache import SimpleCache
from tastypie.resources import ModelResource
from tastypie.throttle import CacheDBThrottle

class UserResource(ModelResource):
    class Meta:
        # What was there before...
        throttle = CacheDBThrottle()
```

9.4.9 What's there now?

- Everything we had before
- **Full** `GET/POST/PUT/DELETE/PATCH` access
- Only **registered users** can use the API & only perform actions on objects they're allowed to
- Object-level caching (`GET` detail)
- Logged throttling that limits users to 150 reqs per hour
- The ability to filter the content

9.5 Extensibility

9.5.1 Designed for extensibility

- Why classes?
  - Not because I'm OO-crazy.
  - It makes extending behavior trivial.
- Why so many classes?
  - Composition > Inheritance
- Why so many methods?
  - Hooks, hooks, hooks.
  - Also makes delegating to composition behaviors easy.
- Tastypie tries to use **reasonable defaults**:
  - You probably want **JSON**
  - You probably want **full** `POST/PUT/DELETE` by default
  - You probably want to use the Model's **default manager** unfiltered
- But Tastypie lets you change all these things
- Plug in custom classes/instances for things like:
  - Serialization
  - Authentication
  - Authorization
  - Pagination
  - Caching
  - Throttling
- `Resource` has lots of methods, many of which are pretty granular
- Override or extend as meets your needs

9.5.2 Customize serialization

- As an example, let's customize serialization
- Supports JSON, XML, YAML, bplist by default
- Let's disable everything but JSON and XML, then add a custom type for HTML
- To limit to just JSON and XML:

```
from django.contrib.auth.models import User
from tastypie.resources import ModelResource
from tastypie.serialization import Serializer

class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()
        excludes = ['email', 'password', 'is_staff', 'is_superuser']
        serializer = Serializer(formats=['json', 'xml'])
```

9.5.3 HTML serialization

```
from django.shortcuts import render_to_response
from tastypie.serialization import Serializer
import cgi
from stringio import StringIO

class TemplateSerializer(Serializer):
    formats = Serializer.formats + ['html']

    def to_html(self, data):
        template_name = 'api/api_detail.html'

        if 'objects' in data:
            template_name = 'api/api_list.html'

        return render_to_response(template_name, data)

    def from_html(self, content):
        form = cgi.FieldStorage(fp=StringIO(content))
        data = {}
        for key in form:
            data[key] = form[key].value
        return data
```

Using it:

```
from django.contrib.auth.models import User
from tastypie.resources import ModelResource
from myapp.api.serializers import TemplateSerializer

class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()
        excludes = ['email', 'password', 'is_staff', 'is_superuser']
        serializer = TemplateSerializer(formats=['json', 'xml', 'html'])
```

9.5.4 Fields

- Just like `ModelForm`, you can control all of the exposed fields on a `Resource/ModelResource`.
- Just like Django, you use a declarative syntax.

```
from django.contrib.auth.models import User
from tastypie import fields
from tastypie.resources import ModelResource

class UserResource(ModelResource):
    # Provided they take no args, even callables work!
    full_name = fields.CharField('get_full_name', blank=True)

    class Meta:
        queryset = User.objects.all()
        excludes = ['email', 'password', 'is_staff', 'is_superuser']
```

- You can control how data gets prepared for presentation (`dehydrate`) or accepted from the user (`hydrate`).
- Happens automatically on fields with `attribute=...` set

```
def dehydrate_full_name(self, bundle):
    return bundle.obj.get_full_name()

def hydrate_full_name(self, bundle):
    ...
    return bundle
```

9.5.5 Caching

- The `SimpleCache` combined with `Resource.cached_obj_get` caches **SINGLE** objects only!
- Doesn't cache the **serialized output**
- Doesn't cache the **list view**

Why?

- More complex behaviors get **opinionated** fast
- Tastypie would rather **be general** & give you the **tools** to build what you need
- Filters and serialization formats make it complex
- Besides...

9.5.6 What you actually want is Varnish

- https://www.varnish-cache.org/
- **Super-fast** caching reverse proxy in C
- Already caches by URI/headers
- Way **faster** than the Django request/response cycle
- `POST/PUT/DELETE` just pass through
- So put Varnish in front of your API (and perhaps the rest of your site) and **win** in the general case.
- Additionally, use Tastypie's internal caching to further speed up Varnish cache-misses.

- Easy to extend `Resource` to add in more caching
- If you get to that point, you're already serving way more load than I ever have.

9.5.7 Data source: not just models!

- `ModelResource` is just a relatively thin (~300 lines) wrapper on top of `Resource` (~1200 lines)
- Just the ORM/Model bits.
- **So virtually everything in Tastypie is available to non-ORM setups.**
- By subclassing from `Resource` and overriding 3 to 9 methods, you can hook up any data source
- http://django-tastypie.readthedocs.org/en/latest/non_orm_data_sources.html

9.5.8 Example: A Solr-based resource (`GET`-only)

(A fair amount of code)

- Takes some work but does a lot for you
- Docs have a more complete example based on Riak
- See also django-tastypie-nonrel.

9.6 Piecrust: The extraction that failed

- Late 2011, tried extracting Tastypie to work anywhere (not just Django). It was called Piecrust.
- http://github.com/toastdriven/piecrust
- Close to functional but failed in terms of **complexity** and lack of **standardization**

CHAPTER 10

Build reliable, traceable, distributed systems with ZeroMQ (ZeroRPC) by Jérôme Petazzoni from dotCloud

Presenter: Jérôme Petazzoni (http://www.dotcloud.com/) (@jpetazzo)

PyCon 2012 presentation page: https://us.pycon.org/2012/schedule/presentation/260/

Slides: https://docs.google.com/presentation/d/1FRh6kb79tcT0Deb4bjYVDvdvJAVMXYjXJK3EFT4pj8w/edit

Video: http://pyvideo.org/video/639/build-reliable-traceable-distributed-systems-wi

Video running time: 36:22

ZeroRPC GitHub repo: https://github.com/dotcloud/zerorpc-python

This talk is about ZeroRPC, a library for doing RPC using ZeroMQ, that was recently open-sourced by dotCloud.

10.1 Introduction

10.1.1 Why did we build an RPC system?

(00:22 / 36:22 into the video)

- dotCloud is a PaaS.
  - We deploy, monitor, and scale your apps (in the cloud!)
- Many moving parts
- ... On a large distributed cluster

10.1.2 What the architecture looks like

(00:45 / 36:22 into the video)

10.1.3 Easy Requirements

(01:08 / 36:22 into the video)

- Expose arbitrary code with minimal modification
  - if we can do `import foo; foo.bar(42)`
  - we want to be able to do `foo = RemoteService(...); foo.bar(42)`

- Self-documented system
  - Want to see methods, signatures, and docstrings

10.1.4 More Difficult Requirements

(02:09 / 36:22 into the video)

- Propagate exceptions
- Language-agnostic
- Brokerless, highly available, fast, support fan-in/fan/out - Not necessarily all at the same time!
- We want to trace & profile nested calls

10.1.5 Why not {x}?

(04:06 / 36:22 into the video)

- Why not HTTP
- Why not AMQP

10.1.6 What we came up with

(04:48 / 36:22 into the video)

ZeroRPC!

- Based on ZeroMQ and MessagePack
- Supports everything we needed!
- Multiple implementations
  - Internal "reference" implementation in Python
  - Public "alternative" implementation with gevent

    Internal Node.js implementation (so-so)

10.2 Using it

10.2.1 Example: unmodified code

(05:45 / 36:22 into the video)

- Expose the `urllib` module over RPC:

```
$ zerorpc-client --server --bind tcp://127.0.0.1:1234 urllib
```

10.2.2 Example: calling code

(06:14 / 36:22 into the video)

- From the command-line (for testing):

```
$ zerorpc-client tcp://127.0.0.1:1234 quote "hello pycon"
connecting to "tcp://127.0.0.1:1234"
'hello%20pycon'
```

- From Python code:

```
>>> import zerorpc
>>> remote_urllib = zerorpc.Client()
>>> remote_urllib.connect('tcp://127.0.0.1:1234')
[None]
>>> remote_urllib.quote('hello pycon')
'hello%20pycon'
```

10.2.3 Example: introspection

(06:38 / 36:22 into the video)

We can list methods:

```
$ zerorpc-client tcp://127.0.0.1:1234 | grep ^q
quote                       quote('abc def') -> 'abc%20def'
quote_plus                  Quote the query fragment of a URL; replacing ' ' with '+'
```

We can see signatures and docstrings:

```
$ zerorpc-client tcp://127.0.0.1:1234 quote_plus -?
connecting to "tcp://127.0.0.1:1234"

quote_plus(s, safe='')

Quote the query fragment of a URL; replacing ' ' with '+'
```

10.2.4 Example: exceptions

(06:47 / 36:22 into the video)

```
$ zerorpc-client tcp://127.0.0.1:1234 quote_plus
connecting to "tcp://127.0.0.1:1234"
Traceback (most recent call last):
...
zerorpc.exceptions.RemoteError: Traceback (most recent call last):
  File "/Users/marca/dev/git-repos/zerorpc-python/zerorpc/core.py", line 201, in _async_task
    functor.pattern.process_call(self._context, socket, event, functor)
  File "/Users/marca/dev/git-repos/zerorpc-python/zerorpc/core.py", line 74, in process_call
    result = context.middleware_call_procedure(functor, *event.args)
  File "/Users/marca/dev/git-repos/zerorpc-python/zerorpc/context.py", line 88, in middleware_call_pr
    return procedure(*args, **kwargs)
  File "/Users/marca/dev/git-repos/zerorpc-python/zerorpc/core.py", line 55, in __call__
    return self._functor(*args, **kargs)
TypeError: quote_plus() takes at least 1 argument (0 given)
```

10.2.5 Example: load balancing

(07:07 / 36:22 into the video)

Start a load balancing hub:

```
$ cat foo.yml
in: "tcp://*:1111"
out: "tcp://*:2222"
type: queue
$ zerohub.py foo.yml
```

Start (at least) one worker:

```
$ zerorpc-client --server tcp://localhost:2222 urllib
```

Now connect to the "in" side of the hub:

```
$ zerorpc-client tcp://localhost:1111
```

10.2.6 Example: high availability

(07:30 / 36:22 into the video)

Start a local HAProxy in TCP mode, dispatching requests to 2 or more remote services or hubs:

```
$ cat haproxy.cfg
listen zerorpc 0.0.0.0:1111
    mode tcp
    server backend_a localhost:2222 check
    server backend_b localhost:3333 check
$ haproxy -f haproxy.cfg
```

Start (at least) one backend:

```
$ zerorpc-client --server --bind tcp://0:2222 urllib
```

Now connect to HAProxy:

```
$ zerorpc-client tcp://localhost:1111
```

10.2.7 Non-example: PUB/SUB

(08:01 / 36:22 into the video)

Not in public repo – yet

- Broadcast a message to a group of nodes
  - But if a node leaves and rejoins, he'll lose messages
- Send a continuous stream of information
  - But if a speaker or listener leaves and rejoins...

You generally don't want to do this!

Better pattern: ZeroRPC streaming with gevent

10.2.8 Example: streaming

(09:10 / 36:22 into the video)

- Server code returns an iterator
- Client code gets an iterator
- Small messages, high latency? No problem! - Server code will pre-push elements - Client code will notify server if pipeline runs low
- Huge messages? No problem! - Big data sets can be nicely chunked - They don't have to fit entirely in memory - Don't worry about timeouts anymore
- Also supports long polling

10.2.9 Example: tracing

(10:15 / 36:22 into the video)

Not in public repo yet

10.3 Implementation details

(11:16 / 36:22 into the video)

This will be useful if:

- You think you might want to use ZeroRPC
- You think you might want to hack ZeroRPC
- You want to implement something similar
- You just happen to love distributed systems

10.3.1 ZeroMQ

(11:50 / 36:22 into the video)

- Sockets on steroids - http://zguide.zeromq.org/page:all
- Handles (re)connections for us
- Works over regular TCP
- Also has superfast `ipc://` and `inproc://`
- Different patterns:
  - REQ/REP
  - PUB/SUB
  - PUSH/PULL
  - DEALER/ROUTER
- `pip install pyzmq-static` FTW (Thanks, Brandon Craig Rhodes!) (pyzmq-static on PyPI)

10.3.2 MessagePack

(13:28 / 36:22 into the video)

- MessagePack
- In our tests, msgpack is more efficient than JSON, BSON, YAML:
  - 20-50x faster
  - serialized output is 2x smaller or better

```
$ pip install msgpack-python
```

```
>>> import msgpack
>>> bytes = msgpack.dumps(data)
```

10.3.3 Wire format

(14:09 / 36:22 into the video)

Request: (headers, method_name, args)

- headers dict
  - no mandatory header
  - carries the protocol version number
  - used for tracing in our in-house version
- args
  - list of arguments
  - no named parameters

Response: (headers, ERR|OK|STREAM, value)

10.3.4 Timeouts

(15:20 / 36:22 into the video)

- 0MQ does not detect disconnections (or rather, it works hard to hide them)
- You can't know when the remote is gone
- Original implementation: 30s timeout
- Published implementation: heartbeat

10.3.5 Introspection

(16:13 / 36:22 into the video)

- Expose a few special calls:
  - `_zerorpc_list` to list calls
  - `_zerorpc_name` to know who you're talking to
  - `_zerorpc_ping` (redundant with the previous one)

  - `_zerorpc_help` to retrieve the docstring of a call
  - `_zerorpc_args` to retrieve the argspec of a call
  - `_zerorpc_inspect` to retrieve everything at once

10.3.6 Naming

(17:10 / 36:22 into the video)

- Published implementation does not include any kind of naming/discovery
- In-house version uses a flat YAML file, mapping service names to 0MQ addresses and socket types, but ashamed to publish this :-)
- In progress: use DNS records
  - SRV for host+port
  - TXT for 0MQ socket type (not sure about this!)
- In progress: registration of services
  - Majordomo protocol

10.3.7 Security: there is none

(18:00 / 36:22 into the video)

- No security at all in 0MQ
  - assumes that you are on a private, internal network
- If you need to run "in the wild", use SSL:
  - bind 0MQ socket on localhost
  - run stunnel (with client cert verification)
- In progress: authentication layer
- dotCloud API is actually ZeroRPC, exposed through a HTTP/ZeroRPC gateway
- In progress: standardization of this gateway

10.3.8 Tracing (not published yet)

(19:26 / 36:22 into the video)

- Initial implementation during a hack day
  - bonus: displays live latency and request rates, using http://projects.nuttnet.net/hummingbird/
  - bonus: displays graphical call flow, using http://raphaeljs.com/
  - bonus: send exceptions to airbrake/sentry
- Refactoring in progress, to "untie" it from the dotCloud infrastructure and Open Source It

How it works: all calls and responses are logged to a central place, along with a trace_id unique to each sequence of calls.

10.3.9 Tracing: trace_id

(21:14 / 36:22 into the video)

- Each call has a trace_id
- The trace_id is propagated to subcalls
- The trace_id is bound to a local context (think thread local storage)
- When making a call:
  - If there is a local trace_id, use it
  - If there is none ("root call"), generate one (GUID)
- trace_id is passed in all calls and responses

Note: this is not (yet) in the GitHub repository:

10.3.10 Tracing: trace collection

(21:42 / 36:22 into the video)

- If a message (sent or received) has a trace_id, we send out the following things:
  - trace_id
  - call name (or, for return values, OK|ERR+exception)
  - current process name and hostname
  - timestamp

Internal details: the collection is built on top of the standard `logging` module.

10.3.11 Tracing: trace storages

(22:10 / 36:22 into the video)

- Traces are sent to a Redis key/value store
  - each trace_id is associated with a list of traces
  - we keep some per=service counters
  - Redis persistence is disabled
  - entries are given a TTL so they expire automatically
  - entries were initially JSON (for easy debugging)
  - ... then "compressed" with msgpack to save space
  - *approximately* 16 GB of traces per day

Internal details: the logging handler does not talk directly to Redis; it sends traces to a collector (which itself talks to Redis)

10.3.12 The problem with being synchronous

(23:19 / 36:22 into the video)

- Original implementation was synchronous
- Long-running calls blocked the server
- Workaround: multiple workers and a hub
- Wastes resources
- Does not work well for *very long* calls
  - Deployment and provisioning of new cluster nodes
  - Deployment and scaling of user apps

Note: this is not specific to ZeroRPC (Preforking servers, threaded servers, WSGI...)

10.3.13 First shot at asynchronicity

(24:28 / 36:22 into the video)

- Send asynchronous events & setup callbacks
- "Please do `foo(42)` and send the result to this other place once you're done"
- We tried this. We failed.
  - distributed spaghetti code
  - trees falling in the forest with no one to hear them
- Might have worked better if we had...
  - better support in the library
  - better naming system
  - something to make sure we don't lose calls (a kind of distributed FSM, maybe?)

10.3.14 Gevent to the rescue!

(26:02 / 36:22 into the video)

- Gevent – http://www.gevent.org/
- Write synchronous code (a.k.a.: don't rewrite your services)
- Uses coroutines to achieve concurrency
- No forks, no threads (no problems? :-))
- Monkey patch standard library (to replace blocking calls with async versions)
- Achieve "unlimited" concurrency server-side

The version published on GitHub uses gevent.

10.3.15 Show me the code!

(27:17 / 36:22 into the video)

https://github.com/dotcloud/zerorpc-python.git

```
$ pip install git+git://github.com/dotcloud/zerorpc-python.git
```

Has:

- `zerorpc` module
- `zerorpc-client` helper
- exception propagation
- gevent integration

Doesn't have:

- tracing
- naming
- helpers for PUB/SUB and PUSH/PULL
- authentication

10.3.16 Questions?

(28:25 / 36:22 into the video)

...

CHAPTER 11

Python, Linkers, and Virtual Memory by Brandon Rhodes

Presenter: Brandon Craig Rhodes (http://rhodesmill.org/brandon/) (@brandon_rhodes)

PyCon 2012 presentation page: https://us.pycon.org/2012/schedule/presentation/455/

Slides: http://rhodesmill.org/brandon/talks/2012-03-pycon/linkers-memory/

Video: http://pyvideo.org/video/717/python-linkers-and-virtual-memory

Video running time: 31:16

11.1 Imagine a computer with 1,000 bytes of addressable RAM...

(00:19 / 36:22 into the video)

- numbered 0 through 999, each storing a byte

Memory address	Byte stored
999	128
998	255
997	254
996	128
...	...
3	2
2	0
1	104
0	77

11.1.1 Pages

We split memory into **pages** by grouping addresses that share a common prefix

(00:29 / 36:22 into the video)

If we defined our

page size = 100 bytes

Then our **1,000** bytes of main memory = **10 pages**

and **page 3** would include all addresses in the **3xx** range.

```
   •
  400
-399-
| 398 |
| 397 |
|  •  |
|  •  |   Page 3
|  •  |
| 302 |
| 301 |
-300-
  299
   •
```

11.1.2 Page tables

(00:50 / 36:22 into the video)

What is a page table?

```
   -----
   | CPU |
   -----

------------
| Page Table |
------------

   -----
   | RAM |
   -----
```

It sits in between your CPU and the RAM on your machine and

rewrites the leading digits of every processor memory access **before** the RAM chip sees the request.

```
Virtual Physical
--------------
| 9--  →  59-- |
| 8--  →  63-- |
| 7--  →     - |
| 6--  →  61-- |
| 5--  →  60-- |
| 4--  →     - |
| 3--  →     - |
| 2--  →     - |
| 1--  →  36-- |
| 0--  →  35-- |
--------------
```

The addresses that your processor asks for are called **virtual**.

The actual addresses delivered to RAM are called **physical**.

- "Read virtual byte **821**" => "Read physical byte **6321**"
- "Read virtual byte **190**" => "Read physical byte **3690**"
- "Write virtual byte **522**" => "Write physical byte **6022**"
- "Read virtual byte **700**" => **Page fault**

See also:

http://en.wikipedia.org/wiki/Page_table

11.1.3 Page Faults

(01:46 / 36:22 into the video)

Your program is paused and the OS regains control.

We will see that the OS has several options about how to respond.

See also:

http://en.wikipedia.org/wiki/Page_fault

11.1.4 Q: What is stored in the bytes of memory pages?

(01:56 / 36:22 into the video)

A: Everything your program needs

- Executable code
- Stack - variables
- Heap - data structures

(02:10) These resources tend to **load gradually** as your program runs.

(02:16) Imagine running your editor...

Which I will not name to avoid obvious religious wars... :-)

You say: "**Run my editor!**"

So the OS loads it into memory along with any libraries it requires.

```
--------------
| 9--  →     - |
| 8--  →     - |
| 7--  →     - |
| 6--  →     - |
| 5--  →     - |
| 4--  →     - |
| 3--  →     - |
| 2--  →     - |
| 1--  →  36-- | libcurses
| 0--  →  35-- | editor "binary" (program)
--------------
```

The editor's `main()` function starts calling other functions.

(02:36) So the OS automatically starts allocating new stack pages as that call graph grows.

The stack usually starts at a high address and grows downward.

```
--------------
| 9--  →  59-- | stack ("bottom" = oldest)
| 8--  →  63-- | stack ("top" = newest)
| 7--  →     - |
| 6--  →     - |
| 5--  →     - |
```

```
| 4--  →     - |
| 3--  →     - |
| 2--  →     - |
| 1--  →  36-- | libcurses
| 0--  →  35-- | editor "binary" (program)
--------------
```

Notice the fairly random physical addresses that can come from anywhere in RAM, because the association is a free one.

(02:57) Then the editor needs space for data structures like lists and dictionaries.

These go in another growing memory area called the **heap**.

Which instead of holding variables that go away when your function returns (i.e.: the stack), holds persistent data structures.

```
--------------
| 9--  →  59-- | stack ("bottom" = oldest)
| 8--  →  63-- | stack ("top" = newest)
| 7--  →     - |
| 6--  →  61-- | heap (file being edited)
| 5--  →  60-- | heap (settings)
| 4--  →     - |
| 3--  →     - |
| 2--  →     - |
| 1--  →  36-- | libcurses
| 0--  →  35-- | editor "binary" (program)
--------------
```

11.1.5 Page Table benefits

(03:22 / 36:22 into the video)

What are the benefits of this level of indirection?

(that has to be implemented in hardware and invoked every time your process hits main memory)

Security

- Processes can't see each other's pages
- Processes can't see OS pages
- The OS wipes reallocated pages with 0s.

Stability

- Processes can't overwrite each other.
- Processes can't crash the OS.
- You can't corrupt another process's memory pages because they're not even visible; they don't exist to you if they're not in the page table.

11.1.6 Protection

(04:22 / 36:22 into the video)

Real page tables include **read** and **write** bits.

In general you can **write** your own stack and heap, but only **read** executables and libraries.

```
----------------
| 9--  →  59-- w | stack
| 8--  →  63-- w | stack
| 7--  →       - |
| 6--  →  61-- w | heap
| 5--  →  60-- w | heap
| 4--  →       - |
| 3--  →       - |
| 2--  →       - |
| 1--  →  36-- r | libcurses
| 0--  →  35-- r | binary
----------------
```

- Write bits set on the stack and heap
- Only the read bit set on the editor binary and the library.

11.1.7 Segmentation Faults

(04:54 / 36:22 into the video)

For an illegal read or write...

Then instead of responding to your page fault with more memory, the OS will stop you dead with a **Segmentation Fault**.

Which basically means a page fault that made the OS angry. :-)

(05:11 / 36:22 into the video)

"Read from **331**"

```
  ----------------
  | 9--  →  59-- w |
  | 8--  →  63-- w |
  | 7--  →       - |
  | 6--  →  61-- w |
  | 5--  →  60-- w |
  | 4--  →       - |
→ | 3--  →       - |  → SEGMENTATION FAULT
  | 2--  →       - |
  | 1--  →  36-- r |
  | 0--  →  35-- r |
  ----------------
```

causes a **segmentation fault** because there's no page there.

"Write to **101**"

```
  ----------------
  | 9--  →  59-- w |
  | 8--  →  63-- w |
  | 7--  →       - |
  | 6--  →  61-- w |
  | 5--  →  60-- w |
  | 4--  →       - |
  | 3--  →       - |
  | 2--  →       - |
→ | 1--  →  36-- r | → SEGMENTATION FAULT
```

```
| 0--  →  35-- r |
----------------
```

causes a **segmentation fault** because we're trying to write to a read-only page.

See also:

http://en.wikipedia.org/wiki/Segmentation_fault

11.1.8 Sharing

(05:28 / 36:22 into the video)

Read-only pages give the OS a new superpower: **Sharing**!

Read-only binaries and libraries can be safely and securely shared among many processes!

The OS can reuse those pages in the page tables of multiple processes.

These two processes use **different** pages for heap, stack, and binary...

```
----------------          ----------------
| 9--  →  59-- w | stack   | 9--  →  48-- w | stack
| 8--  →  63-- w | stack   | 8--  →       - |
| 7--  →       - |         | 7--  →  51-- w | heap
| 6--  →  61-- w | heap    | 6--  →  50-- w | heap
| 5--  →  60-- w | heap    | 5--  →  46-- w | heap
| 4--  →       - |         | 4--  →       - |
| 3--  →       - |         | 3--  →  39-- r | libjson
| 2--  →  39-- r | libjson | 2--  →  48-- r | libX11
| 1--  →  36-- r | libc    | 1--  →  36-- r | libc
| 0--  →  35-- r | python  | 0--  →  47-- r | firefox
----------------          ----------------
```

But they safely share a **single** copy of *libc* and *libjson*

```
----------------          ----------------
|                |        |                |
|                |        |                |
|                |        |                |
|                |        |                |
|                |        |                |
|                |        |                |
|                |        | 3--  →  39-- r | libjson
| 2--  →  39-- r | libjson |                |
| 1--  →  36-- r | libc    | 1--  →  36-- r | libc
|                |        |                |
----------------          ----------------
```

When we share pages we don't need to use up RAM storing things twice.

Physical RAM page **36** can be reused for every process needing **libc**

Similarly for page **39** and **libjson**, because if you can't write to it then it looks like it's your own personal copy.

11.1.9 Memory consumed doesn't add up

(06:32 / 36:22 into the video)

This means that the total memory consumed by process A and process B is **rarely** the sum

(A's memory use) + (B's memory use)

Q: So how can you tell how much memory an additional worker thread or process will consume on one of your servers?

A: Stop looking at the **quantity** – "How much RAM does my Python program use?"

Start looking at the **delta**

Δ memory

Ask, "How much memory does each **additional** process / worker / thread cost (and when will that run me out of RAM)?"

How should you measure that?

By actual load and resource tests

Because as we will see, memory usage is *complicated*.

11.2 Problem: Python code

(07:37 / 36:22 into the video)

We hit a problem with this idea of sharing binary code.

Q: Where in virtual memory does Python code live?

Python starts running with most RAM pages **shareable**.

```
-----------------
| 9--  →  59-- w | stack
| 8--  →       - |
| 7--  →       - |
| 6--  →       - |
| 5--  →       - |
| 4--  →       - |
| 3--  →       - |
| 2--  →       - |
| 1--  →  36-- r | libc
| 0--  →  35-- r | python
-----------------
```

But Python runs `read()` on `foo.py` creating a writable, **unshareable** page on the heap

```
-----------------
| 9--  →  59-- w | stack
| 8--  →       - |
| 7--  →       - |
| 6--  →       - |
| 5--  →  60-- w | foo.py   ←
| 4--  →       - |
| 3--  →       - |
| 2--  →       - |
| 1--  →  36-- r | libc
| 0--  →  35-- r | python
-----------------
```

Then Python compiles `foo.py` to a code object on *another* writable, **unshareable** page

```
-----------------
| 9--  →  59-- w | stack
| 8--  →       - |
```

```
| 7--  →       - |
| 6--  →  61-- w | codeobj  ←
| 5--  →  60-- w | foo.py
| 4--  →       - |
| 3--  →       - |
| 2--  →       - |
| 1--  →  36-- r | libc
| 0--  →  35-- r | python
----------------
```

Code object is another unshareable page.

Only then does `foo.py` start up and create legitimately unique program data

```
----------------
| 9--  →  59-- w | stack
| 8--  →       - |
| 7--  →  63-- w | data     ←
| 6--  →  61-- w | codeobj
| 5--  →  60-- w | foo.py
| 4--  →       - |
| 3--  →       - |
| 2--  →       - |
| 1--  →  36-- r | libc
| 0--  →  35-- r | python
----------------
```

The heap winds up as a mix of actual unique data, together with shared code

```
----------------
| 9--  →  59-- w | stack
| 8--  →       - |
| 7--  →  63-- w | data     ←
| 6--  →  61-- w | codeobj
| 5--  →  60-- w | foo.py
| 4--  →       - |
| 3--  →       - |
| 2--  →       - |
| 1--  →  36-- r | libc
| 0--  →  35-- r | python
----------------
```

Now it *just so happens* that every Python X.Y process loading `foo.py` will create the *exact same code object*...

–but the OS does not know about this because each Python process builds its code objects separately

To the OS pages that get written in the middle of a process running look like **heap** and **don't get shared**.

```
----------------         ----------------
| 9--  →  59-- w |         | 9--  →  48-- w | stack
| 8--  →       - |         | 8--  →       - |
| 7--  →  63-- w |         | 7--  →  71-- w | data
| 6--  →  61-- w |         | 6--  →  69-- w | codeobj
| 5--  →  60-- w |         | 5--  →  62-- w | foo.py
| 4--  →       - |         | 4--  →       - |
| 3--  →       - |         | 3--  →       - |
| 2--  →       - |         | 2--  →       - |
| 1--  →  36-- r |         | 1--  →  36-- r | libc
| 0--  →  35-- r |         | 0--  →  35-- r | python
----------------         ----------------
```

11.2.1 Q: Why not share code objects? A: reference counts

(09:12 / 36:22 into the video)

Standard CPython does not support read-only code objects because it constantly fiddles around with their **reference counts**.

What does this code do?

```
f(5)
```

1. Looks up `f()`
2. Gets back a code object
3. Increment its reference count
4. Calls it with `(5,)`
5. Decrement its reference count

Why increment the count?

To prevent other threads from de-allocating and overwriting `f()` while this thread is busy running it

(10:13 / 36:22 into the video)

Note: Lesson: The OS offers cool optimization when processes need to *share code*, but *reference-counted* code objects cannot take advantage of this.

See also:

http://docs.python.org/extending/extending.html#reference-counts

11.3 Dynamic Linking

(10:22 / 36:22 into the video)

How does the Python interpreter find the routines it needs in `libc.so`?

```
----------------
|                |
|                |
|                |
|                |
|                |
|                |
|                |
|                |
| 1--  →  36-- r | libc.so
| 0--  →  35-- r | python
----------------
```

See also:

http://en.wikipedia.org/wiki/Dynamic_linking Wikipedia article on dynamic linking

http://en.wikipedia.org/wiki/Shared_libraries#Shared_libraries Wikipedia article on shared libraries

11.3.1 The old days: static linking

```
 "compile"  "link"

cmdn.c → cmdn.o
opts.c → opts.o → prog
slen.c → slen.o
```

Every `.o` file has a table of names that it defines and names that it needs someone else to define

```
$ nm p_move.o
         U _nc_panelhook
         U is_linetouched
00000000 T move_panel
         U mvwin
         U wtouchln
```

Linking returns an error if a name needed cannot be found

```
$ ld -o prog foo.o bar.o baz.o

foo.o: In function 'main':
foo.c:(.text+0x7): undefined reference to 'haute'
collect2: ld returned 1 exit status
```

11.3.2 ar archives

(12:32 / 36:22 into the video)

The **ar** tool bundles together related `.o` files into `.a` archives.

```
$ ar t /usr/lib/libpanel.a
panel.o
p_above.o
p_below.o
p_bottom.o
p_delete.o
p_hide.o
p_hidden.o
p_move.o
p_new.o
p_replace.o
p_show.o
p_top.o
p_update.o
p_user.o
p_win.o
```

`.a` files are static libraries.

See also:

http://en.wikipedia.org/wiki/Static_library Wikipedia article on static libraries

11.3.3 Shared libraries

Problem:

(13:43 / 36:22 into the video)

If everyone lets the linker copy `printf.o` into their program, then memory has to hold many separate copies of each popular library

Solution:

The invention of `.so` "shared object" files that can be added to the page table of every program that needs them

```
----------------
|       .        |
|       .        |
|       .        |
| 1--  →  36-- r | libc.so
| 0--  →  35-- r | python
----------------
```

The last-minute linking that takes place at runtime to connect python to `libc.so` is called **Dynamic Linking**

(15:09) You can see the libraries that a program like python needs with **ldd**:

```
$ ldd /usr/bin/python2.7
      linux-gate.so.1
      libpthread.so.0
      libdl.so.2
      libutil.so.1
      libssl.so.1.0.0
      libcrypto.so.1.0.0
      libz.so.1
      libm.so.6
      libc.so.6
      /lib/ld-linux.so.2
```

Note: (from Marc) On Mac OS X, use `otool -L` to see what libraries a program needs:

```
$ otool -L /usr/bin/python2.6
/usr/bin/python2.6:
        /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 125.2.1)
```

See also:

otool Apple's documentation for the **otool** program.

(15:17) You can see the specific names it needs with **nm** ("names")

```
$ nm -D /usr/bin/python2.7
        ...
        U unlink
        U unsetenv
        U utime
        U utimes
        U wait
        U wait3
        U wait4
        U waitpid
        U wcscoll
        ...
```

Note: (from Marc) `-D` (alias: `--dynamic`) is an option of **nm** from GNU binutils. The **nm** in Mac OS X does not have it.

Not only is it usual for the Python *binary* to be dynamically linked but *Python extension modules* sometimes link against shared libraries too

The `lxml.etree` Python module is famous for this:

```
$ ldd /usr/lib/pyshared/python2.7/lxml/etree.so
        linux-gate.so.1
        libxslt.so.1
        libexslt.so.0
        libxml2.so.2
        libpthread.so.0
        libc.so.6
        libm.so.6
        libgcrypt.so.11
        libdl.so.2
        libz.so.1
        /lib/ld-linux.so.2
        libgpg-error.so.0
```

See also:

http://en.wikipedia.org/wiki/Shared_libraries#Shared_libraries Wikipedia article on shared libraries

http://en.wikipedia.org/wiki/Dynamic_linking Wikipedia article on dynamic linking

11.3.4 Shared library problems

(15:43)

Shared library: `libtiny.so`

```
/* libtiny.c */
helper() { return 42; }

gcc libtiny.c -shared -o libtiny.so
```

Python module: `tinymodule.so`

```
/* tinymodule.c */

#include <python2.7/Python.h>

static PyMethodDef TinyMethods[] = {
    {NULL, NULL, 0, NULL}
};

PyMODINIT_FUNC inittiny(void)
{
    helper();  /* the call into libtiny.so */
    (void) Py_InitModule("tiny", TinyMethods);
}

gcc tinymodule.c -L. -ltiny -shared -o tinymodule.so
```

Having created these two `.so` files we need to add the current directory to the OS shared library search path

```
$ export LD_LIBRARY_PATH=.
```

Note: The above is for Linux. On Mac OS X, use `DYLD_LIBRARY_PATH`

So the module `tinymodule.so` needs the library `libtiny.so`

```
$ ldd tinymodule.so
        linux-gate.so.1
        libtiny.so
        libc.so.6
        /lib/ld-linux.so.2
```

Note: The above is for Linux. On Mac OS X, instead of **ldd**, use **otool -L**.

Shared library present, compatible

```
$ ls
libtiny.c   tinymodule.c
libtiny.so  tinymodule.so

$ python -c 'import tiny'

$ echo $?
0
```

Shared library missing

```
$ rm libtiny.so
$ python -c 'import tiny'
ImportError: libtiny.so:
    cannot open shared object file:
    No such file or directory
```

Incompatible shared library

```
/* libtiny.c */
different_helper() { return 42; }
gcc libtiny.c -shared -o libtiny.so

$ python -c 'import tiny'
ImportError: ./tinymodule.so:
    undefined symbol: helper
```

"Cannot open shared object" and *"undefined symbol"* are possible because OS tries to link a binary (`tinymodule.so`) to its dependencies (`libtiny.so`) *at runtime*

11.4 Demand Paging

(18:00 / 36:22 into the video)

The OS does not load pages from disk until your program reads or writes from them

Imagine a program that has just started running

Files:

- python — 3 pages
- libc.so — 2 pages

At first, only one page of the python binary is loaded

```
----------------
| 9--  →  59-- w |  stack
| 8--  →       - |
| 7--  →       - |
| 6--  →  61-- w |  heap
| 5--  →       - |
| 4--  →       - | (libc.so)
| 3--  →       - | (libc.so)
| 2--  →       - | (python)
| 1--  →       - | (python)
| 0--  →  35-- r |  python ← LOAD FROM DISK
----------------
```

Then Python tries to run the instruction at **212** so that page gets loaded too

```
----------------
| 9--  →  59-- w |  stack
| 8--  →       - |
| 7--  →       - |
| 6--  →  61-- w |  heap
| 5--  →       - |
| 4--  →       - | (libc.so)
| 3--  →       - | (libc.so)
| 2--  →  37-- r |  python ← LOAD FROM DISK
| 1--  →       - | (python)
| 0--  →  35-- r |  python
----------------
```

Then Python calls a function at libc offset **178** thus **300 + 178 = 478**

```
----------------
| 9--  →  59-- w |  stack
| 8--  →       - |
| 7--  →       - |
| 6--  →  61-- w |  heap
| 5--  →       - |
| 4--  →  41-- r |  libc.so ← LOAD FROM DISK
| 3--  →       - | (libc.so)
| 2--  →  37-- r |  python
| 1--  →       - | (python)
| 0--  →  35-- r |  python
----------------
```

If Python keeps calling the same routines, then only those 3 pages ever get loaded

So pages are loaded from disk **lazily**

Heap allocation works the **same way**

(18:50 / 36:22 into the video)

Q: How many RAM pages are allocated if you ask the OS for three new memory pages?

A: None!

The OS allocates *no pages* but simply makes an adjustment to its record-keeping

```
----------------
| 9--  →  59-- w |  stack
| 8--  →       - |
| 7--  →       - | (heap)  ←
| 6--  →       - | (heap)  ←
| 5--  →       - | (heap)  ←
```

```
| 4--  →  61-- w |  heap
| 3--  →       - |
| 2--  →  41-- r |  libc.so
| 1--  →       - | (python)
| 0--  →  35-- r |  python
----------------
```

The OS will now respond to page faults at **500–799** by allocating a new page, not raising a Segmentation Fault

But no pages are actually allocated until a **read** or **write** shows they are *needed*

(19:19 / 36:22 into the video)

That is the difference

between **VIRT** and **RES** when you run **top**

top

```
               ↓     ↓

  PID USER    VIRT  RES  SHR S %CPU %MEM COMMAND
 1217 root    195m  94m  53m S    2  2.6 Xorg
 7304 brandon 183m  45m  19m S    2  1.3 chromium
 2027 brandon 531m 162m  37m S    1  4.5 chromium
 2319 brandon 179m  58m  23m S    1  1.6 chromium
18841 brandon 2820 1188  864 R    1  0.0 top
 9776 brandon 175m  49m  12m S    0  1.4 chromium

               ↑     ↑
```

See also:

Demand paging Wikipedia article on Demand Paging

VIRT

- Virtual Image
- All memory pages that promise not to return a Segmentation Fault

RES

- Resident Size
- Only memory pages that have actually been allocated

Diagram to illustrate the difference

- VIRT — 8 pages
- RES — 4 pages

```
----------------         VIRT  RES
| 9--  →  59-- w |  stack
| 8--  →       - |
| 7--  →       - | (heap)
| 6--  →       - | (heap)
| 5--  →       - | (heap)
| 4--  →  61-- w |  heap
| 3--  →       - |
| 2--  →  41-- r |  libc.so
| 1--  →       - | (python)
```

```
| 0--  →  35-- r |  python
----------------
```

Only RES can actually run you out of memory

Look at RES when you wonder where all of your RAM is going

```
                    ↓

  PID USER    VIRT  RES  SHR S %CPU %MEM COMMAND
 1217 root    195m  94m  53m S    2  2.6 Xorg
 7304 brandon 183m  45m  19m S    2  1.3 chromium
 2027 brandon 531m 162m  37m S    1  4.5 chromium
 2319 brandon 179m  58m  23m S    1  1.6 chromium
18841 brandon 2820 1188  864 R    1  0.0 top
 9776 brandon 175m  49m  12m S    0  1.4 chromium

                    ↑
```

11.4.1 /proc/<pid>/smaps

(20:19 / 36:22 into the video)

Linux file; shows whether physical RAM pages have been allocated to a segment

---

Note: (from Marc) On Linux systems, there is a nice tool called **pmap** that shows similar information in a nice format.

See also:

pmap(1) Man page for the **pmap** program.

Note: (from Marc) This is on Linux; OS X does not have the `/proc` filesystem. The OS X equivalent of this is the **vmmap** command.

See also:

vmmap(1) Man page for the **vmmap** program.

"Viewing Virtual Memory Usage" A section from the Memory Usage Performance Guidelines guide in the Mac OS X Developer Library

---

Here are some segments of a new python process sitting quietly at its prompt

```
08048000-08238000 r-xp .../bin/python
Size:               1984 kB
Rss:                 896 kB

b73b9000-b752f000 r-xp .../libc-2.13.so
Size:               1496 kB
Rss:                 528 kB

bfc48000-bfc69000 rw-p [stack]
Size:                136 kB
Rss:                 104 kB
```

Demand Paging: Example

(20:45)

Calling a Python function that was not called as Python started up

```
08048000-08238000 r-xp .../bin/python
Size:               1984 kB
Rss:                 896 kBA

>>> frozenset()

08048000-08238000 r-xp .../bin/python
Size:               1984 kB
Rss:                 916 kB
```

916 - 896 = 20 new kilobytes

So invoking new sections of a binary or library pulls more pages into RAM

But since binary and library code can always be reloaded from disk, the OS can also *discard* those pages

```
$ python -c 'range(500000000)'
```

This allocates lots of memory As it runs, your OS will dump information overboard to reclaim every RAM page it can

What did this memory hog do to the other Python process that was sitting quietly at its prompt?

```
08048000-08238000 r-xp .../bin/python
Size:               1984 kB
Rss:                 916 kB
                     ↑
                     before the hog ran

                     after
                     ↓
08048000-08238000 r-xp .../bin/python
Size:               1984 kB
Rss:                 464 kB
```

916 - 464 = 452k were thrown overboard!

Those 452k pages will be re-loaded, on-demand, from disk if this Python process tries to access them again

The process will slow down as de-allocated pages are re-loaded

- L1 cache — 3s grabbing a piece of paper
- L2 cache — 14s picking a book from a shelf
- System RAM — 4-minute walk down the hall
- Hard drive seek —

  "like leaving the building to roam the earth for one year and three months."

  — Gustavo Duarte

11.5 Forking

(22:37 / 36:22 into the video)

On primitive operating systems the only way to create a new process is to start a whole new program

Linux and OS X support `fork()`!

`fork()` creates a child process that continues on from the parent's current state

```
Process 1  A
           ↓
           B
           ↓
           C
           ↓
           D -- os.fork()   Process 2
           ↓                ↓
           E                E
           ↓                ↓
           F                F
           ↓                ↓
```

The OS *could* implement `fork()` naively by copying every writeable page in the parent so that the child had its own copy

```
  fork() parent                fork() child
----------------          -----------------
| 9--  →  59-- w | stack   | 9--  → new copy |
| 8--  →  63-- w | stack   | 8--  → new copy |
| 7--  →       - |         | 7--  →        - |
| 6--  →  61-- w | heap    | 6--  → new copy |
| 5--  →  60-- w | heap    | 5--  → new copy |
| 4--  →       - |         | 4--  →        - |
| 3--  →       - |         | 3--  →        - |
| 2--  →  39-- r | libjson | 2--  →  (same)  |
| 1--  →  36-- r | libc    | 1--  →  (same)  |
| 0--  →  35-- r | python  | 0--  →  (same)  |
----------------          -----------------
```

But immediately copying every page could take a long time for a large process, and during the copy both parent and child would be hung waiting!

Instead, as you might guess, the OS implements `fork()`

lazily,

copying pages on-demand when the parent or child performs a write

So the OS starts `fork()` by copying the page table, marking all pages read-only in both processes, then letting them keep running

```
  fork() parent                fork() child
----------------          -----------------
| 9--  →  59-- r | stack   | 9--  →  59-- r |
| 8--  →  63-- r | stack   | 8--  →  63-- r |
| 7--  →       - |         | 7--  →       - |
| 6--  →  61-- r | heap    | 6--  →  61-- r |
| 5--  →  60-- r | heap    | 5--  →  60-- r |
| 4--  →       - |         | 4--  →       - |
| 3--  →       - |         | 3--  →       - |
| 2--  →  39-- r | libjson | 2--  →  (same) |
| 1--  →  36-- r | libc    | 1--  →  (same) |
| 0--  →  35-- r | python  | 0--  →  (same) |
----------------          -----------------
```

As writes cause page faults the OS makes copies. Here page **63** is copied to **77** and marked **w**.

```
    fork() parent                fork() child
  ----------------            ----------------
```

```
  | 9--  →  59-- r |   stack    | 9--  →  59-- r |
→ | 8--  →  63-- w |   stack    | 8--  →  77-- w | ←
  | 7--  →       - |            | 7--  →       - |
  | 6--  →  61-- r |   heap     | 6--  →  61-- r |
  | 5--  →  60-- r |   heap     | 5--  →  60-- r |
  | 4--  →       - |            | 4--  →       - |
  | 3--  →       - |            | 3--  →       - |
  | 2--  →  39-- r |   libjson  | 2--  →  (same) |
  | 1--  →  36-- r |   libc     | 1--  →  (same) |
  | 0--  →  35-- r |   python   | 0--  →  (same) |
  ----------------             ----------------
```

(24:18 / 36:22 into the video)

Q: How well does this *usually* work?

A: It works *great*!

Only the *differences* that develop between the parent and child memory images incur additional storage

Q: But how does it work for *Python*?

A: Terribly!

Thanks to reference counts, merely *glancing* at data with CPython forces the OS to create a separate copy

11.6 CPython vs. PyPy

```
# Create a list

biglist = ['foo']
for n in range(1, 8460000):
    biglist.append(n)    # ~100 MB

# put fork() here

# Iterate across the list

t0 = time.time()
all(n for n in biglist)
t1 = time.time()
```

11.6.1 CPython

Before C Python iterates

```
$ cat /proc/22364/smaps | awk '/heap/,/Private_D/'
08d60000-0ef90000 rw-p 00000000 00:00 0      [heap]
Size:             100544 kB
Rss:              100544 kB
Pss:               50294 kB
Shared_Clean:          0 kB
Shared_Dirty:     100500 kB
Private_Clean:         0 kB
Private_Dirty:        44 kB
```

After C Python iterates

```
$ cat /proc/22364/smaps | awk '/heap/,/Private_D/'
08d60000-0ef90000 rw-p 00000000 00:00 0      [heap]
Size:             100544 kB
Rss:              100544 kB
Pss:              100274 kB
Shared_Clean:          0 kB
Shared_Dirty:        540 kB
Private_Clean:         0 kB
Private_Dirty:    100004 kB
```

At finish: **0.5%** of heap shared

11.6.2 PyPy 2.8

Before PyPy iterates

```
$ cat /proc/22385/smaps | awk '/heap/,/Private_D/'
0a5c9000-11fba000 rw-p 00000000 00:00 0      [heap]
Size:             124868 kB
Rss:              124540 kB
Pss:               62274 kB
Shared_Clean:          0 kB
Shared_Dirty:     124532 kB
Private_Clean:         0 kB
Private_Dirty:         8 kB
```

After PyPy iterates

```
$ cat /proc/22385/smaps | awk '/heap/,/Private_D/'
0a5c9000-11fba000 rw-p 00000000 00:00 0      [heap]
Size:             124868 kB
Rss:              124868 kB
Pss:               63160 kB
Shared_Clean:          0 kB
Shared_Dirty:     123416 kB
Private_Clean:         0 kB
Private_Dirty:      1452 kB
```

At finish: **96.1%** of heap shared

(26:03 / 36:22 into the video)

Lesson:

Forked worker processes in Unix *share* memory when the parent process builds *read-only* data structures

—**but** not in CPython

See also:

http://en.wikipedia.org/wiki/Fork_(operating_system) Wikipedia article on process forking

11.7 Explicitly Sharing Memory

(26:14 / 36:22 into the video)

How can two Python procedures share *writeable* memory to collaborate?

1. Threads
2. Memory maps

11.7.1 Threads

(26:48 / 36:22 into the video)

Creating a thread is like `fork()` except that the heap *remains shared* between the two threads of control

Python supports threads on both Unix and Windows

```
import threading
t = threading.Thread(target=myfunc)
t.start()
```

Each thread gets its *own stack* so the two threads can call different functions, but *all other* data structures are *shared*

```
    main thread               child thread
----------------          ----------------
| 9--  →  59-- r |  stack  | 9--  →  59-- r |
| 8--  →  63-- w |  stack  | 8--  →  77-- w |
---------------------------------------
             | 7--  →       - |
             | 6--  →  61-- w | heap
             | 5--  →  60-- w | heap
             | 4--  →       - |
             | 3--  →       - |
             | 2--  →  39-- r | libjson
             | 1--  →  36-- r | libc
             | 0--  →  35-- r | python
             ----------------
```

Since threads share *every* data structure they have to be *very* careful

See also:

http://en.wikipedia.org/wiki/Thread_(computing) Wikipedia article on threads

11.7.2 Memory maps

(27:19 / 36:22 into the video)

Using `mmap()` a parent process can create shared memory that will be inherited by all the child workers it forks

```
import mmap, os

map = mmap.mmap(-1, 100)
os.fork()
...
```

```
 fork() parent                fork() child
----------------          ----------------
| 9--  →  59-- r |   stack  | 9--  →  59-- r |
| 8--  →  63-- w |   stack  | 8--  →  77-- w |
---------------------------------------
             | 7--  →  88-- - | mmap segment
---------------------------------------
| 6--  →  61-- r |   heap   | 6--  →  61-- r |
```

```
| 5--  →  60-- r |   heap    | 5--  →  60-- r |
| 4--  →       - |           | 4--  →       - |
| 3--  →       - |           | 3--  →       - |
| 2--  →  39-- r |   libjson | 2--  →  (same) |
| 1--  →  36-- r |   libc    | 1--  →  (same) |
| 0--  →  35-- r |   python  | 0--  →  (same) |
----------------          ----------------
```

This supports very fast RAM-based communication between processes

without requiring every data structure on the heap to carry *locks* or other protection

Another mmap() capability

Remember how the OS loads pages from binary programs like `python` and shared libraries like `libc.so` on-demand, not all at once?

Well, with `mmap()` you can do that yourself, with normal files!

```
mmap(myfile.fileno(), 0)
```

lets you replace `seek()`, `read()`, and `write()` calls and simply access it like an array!

```
----------------
| 9--  →  59-- w |  stack
| 8--  →       - |
| 7--  →  91-- - |  mmap[1]  myfile
| 6--  →  90-- - |  mmap[0]  myfile
| 5--  →       - |
| 4--  →  61-- w |  heap
| 3--  →       - |
| 2--  →  41-- r |  libc.so
| 1--  →       - | (python)
| 0--  →  35-- r |  python
----------------
```

See also:

http://en.wikipedia.org/wiki/Mmap Wikipedia article on `mmap()`

11.8 Summary

(28:42 / 36:22 into the video)

Your processes all access memory through the mediation of a *page table*

Page tables power all kinds of fun RAM optimizations:

1. Demand loading
2. Shared libraries
3. Memory maps
4. `fork()`
5. Threads

And

PyPy lacks reference counts which lets the OS *do its magic* and *conserve resources*

(29:16 / 36:22 into the video)

11.9 Questions

(29:35 / 36:22 into the video)

Just one question

- Erik Rose said he recently started using `mmap()` in a non-Python context. Free RAM meters (e.g.: **free**, **top**) behaving strangely. Do you know if mmapped memory that is actually resident affects free counts or does it somehow go behind its back?
  - **A: Very dependent on the version of the operating system.** Reporting tools can have odd effects. What I would expect is for it only show pages disappearing when they're allocated. When you mmap, you can ask for it to eagerly grab the memory.

11.10 Marc's Prologue

11.10.1 Resources

- A great free resource is Ulrich Drepper's DSO How To (a.k.a.: "How To Write Shared Libraries").
- An awesomely detailed blog post on Position-Independent Code (PIC) by Eli Bendersky
- For more great info on this kind of stuff, check out "Linkers & Loaders" by John R. Levine.

  The book also has a web site.
- To delve deep into how the Linux kernel implements virtual memory (and other stuff), you might check out "Linux Kernel Development" by Robert Love

CHAPTER 12

Scalability at YouTube by J.J. Behrens and Mike Solomon of Google

Presenters: J.J. Behrens and Mike Solomon of Google

PyCon 2012 presentation page: https://us.pycon.org/2012/schedule/presentation/315/

Slides: ???

Video: https://www.youtube.com/watch?v=G-lGCC4KKok&list=PLBC82890EA0228306&index=22&feature=plpp_video

Video running time: 38:43

First few minutes of the video are marketing fluff.

(10:16) The talk really starts.

12.1 Briefly

- scalability
- efficiency
- productivity

People often confuse scalabity and efficiency.

12.2 Scalable...systems

(12:32)

- simple - "If I could give you one word about scalable systems, I think the word would be *simple*"
- solve the problem at hand
- product of evolution

12.3 The Tao of Youtube

"choose the simplest solution with the loosest guarantees that are practical"

12.4 Scalable...techniques

(14:51)

- divide and conquer - e.g.: independent web servers, database sharding, partition the problem, simple and loose connections
- approximate correctness - cheat and let people see their own comments rather than requiring immediate consistency
- expert knob twiddling - consistency, durability
- jitter - introduce more randomness, because surprisingly, things tend to stack up - example: cache expirations - if all caches expires at the same time => thundering herd. Add some random variance.
- cheat

Jitter puts entropy back into the system.

12.5 Scalable...components

(21:06)

- well defined inputs
- well defined dependencies
- frequently end up behind an RPC
- leverage local optimizations

12.6 Scalable...humans?

(23:12)

- communication through code/data/schema
- RPC as a means of sanity

12.7 Efficiency

(24:26)

- uncorrelated with scalability
- focus on algorithms first
- learn to measure - use representative samples

12.8 Efficient libraries

(26:40)

- wiseguy - YouTube's fast CGI container

- pycurl
- spitfire - YouTube's templating system; similar to Cheetah
- serialization formats* - Don't use pickle

12.9 Efficient tools

(29:46)

- apache
- linux
- mysql
- vitess
- zookeeper

12.10 Productivity

(32:12)

- philosophy vs. doctrine
- more conventions - less documentation
- more effective collaboration
- more diffusion of responsibility

12.11 Zen proverb

(33:54)

"Do not seek to follow in the footsteps of masters; seek what they sought"

12.12 Code Bonsai

(34:22)

- simple
- pragmatic
- elegant
- orthogonal
- composable

12.13 Questions?

(35:00)

CHAPTER 13

web2py: ideas we stole and ideas we had by Massimo Di Pierro of DePaul University

Presenters: Massimo Di Pierro of DePaul University

PyCon 2012 presentation page: https://us.pycon.org/2012/schedule/presentation/112/

Slides: http://dl.dropbox.com/u/18065445/Slides/PySFTalkSlides.pdf

Video: https://www.youtube.com/watch?v=M5IPlMe83yI&list=PLBC82890EA0228306&index=16&feature=plpp_video

Video running time: 31:55

web2py - http://web2py.com/

13.1 Ideas we borrowed

(01:16)

- Model View Controller on WSGI (like everyone else)
- w2p files (like Java's Web application ARchives)
- Optional routing (like Rails)
- Routing mechanism (like Django)
- Pure Python templating language (like Mako)
- Helpers (like Rails) but easier to remember: `span`, `a`, etc.
- `request['xxx'] == request.xxx` (like web.py)
- web based database interface (like Django admin)

13.2 Ideas we had

(03:10)

- A tool for both technical and non-technical people
- Always backwards compatible (since 2007, 2.5, 2.6, 2.7, pypy, jython)
- One click deploy (Windows and Mac binaries, USB drive)
- No configuration, no dependencies, and secure by default

- Everything has default (DRY)
- Multi project / multi db / share nothing by default
- Web based IDE (development, editor, deployment, management, translations, testing, debugger, version control) shell optional
- Automatic db migrations (CREATE and ALTER table)
- Plugins / Components / Ajax with digitally signed URLs

(05:10)

- Role based access control
- Every app is a Central Authentication Service provider (and optionally consumer)
- Built-in portable cron and master/workers task scheduler
- multi tenant apps with common fields & common filters
- record versioning (without breaking refs)
- embeddable CRUD and grid controls
- Package lots of 3rd party modules: pyfpdf, database drivers, credit card payment APIs, ...

13.3 Getting web2py

(06:32)

```
wget http://www.web2py.com/examples/static/web2py_src.zip
unzip web2py_src.zip
cd web2py
web2py.py -a 'apassword' &
```

Gluon is 1.3 MB; smaller than most microframeworks

13.4 Multi-project

(09:24)

13.5 Web based IDE

(09:43)

13.6 Mobile IDE

(09:59)

13.7 Web translation

(10:07)

13.8 Tickets

(10:17)

13.9 Model-View-Controller

(10:40)

13.10 Let's build something

(11:08)

- URL shortening service
- Bookmark service with tagging
- Click counter
- URL rating (WOT service)

13.11 Demo

13.11.1 Download and start web2py

(11:44)

```
# download and start web2py
wget http://www.web2py.com/examples/static/web2py_src.zip
unzip web2py_src.zip
cd web2py
python2.7 web2py.py -a hello -p 8000 &
open http://127.0.0.1:8000/welcome/

# Shows the welcome page

cd applications
ls
__init__.py
__init__.pyc
admin
examples
friends
links
welcome
```

13.11.2 Explore the admin interface

(12:20)

```
open http://127.0.0.1:8000/admin/

# Shows the administrative interface
# Can edit files
# Can get a web-based Python shell
# Can run web-based tests
# Can configure crontab
# Can access Mercurial versioning
# Can access error logs
# Can upgrade web2py
# Can use new application wizard
```

13.11.3 Create a new application from the shell

(13:38)

```
mkdir myapp
cp -r welcome/* myapp/
cd myapp
open http://127.0.0.1:8000/myapp/default/index

rm controllers/default.py
edit controllers/default.py

def index():
    return "hello world"

def oops():
    1/0

open http://127.0.0.1:8000/myapp/default/index
```

http://127.0.0.1:8000/myapp/default/index displays "hello world"

13.11.4 Show error with ticket and traceback

(14:25) http://127.0.0.1:8000/myapp/default/oops displays "Internal error" with a link to a ticket with the traceback.

13.11.5 Edit a model

(14:37)

```
edit models/mydb.py

Link = db.define_table(
    'link',
    Field('url', unique=True))

open http://127.0.0.1:8000/myapp/appadmin
```

13.11.6 App administrative interface

(14:48)

13.11.7 Add more fields to our model

(15:00)

```
edit models/mydb.py

Link = db.define_table(
    'link',
    Field('url', unique=True),
    Field('visits', 'integer', default=0),
    Field('screenshot', 'upload', writable=False),
    format = '%(url)s')

Bookmark = db.define_table(
    'bookmark',
    Field('Link', 'reference link', writable=False),
    Field('category', requires=IS_IN_SET(['work', 'personal'])),
    Field('tags', 'list:string'),
    auth.signature)

def toCode(id):
    s, c = 'GKys67LJPDAFvcEp9rkw8...', ''
    while id: c,id = c+s[id % 55], id//55
    return c

def toInt(code):
    s, id = 'GKys67LJPDAFvcEp9rkw8...', 0
    for i in range(len(code)): id += s.find(code[i])*55**i
    return id

def shorten(id, row):
    s = URL('visit', args=toCode(id), scheme=True)
    return A(s, _href=s)

Bookmark.link.represent = shorten
Bookmark.id.readable = False
Bookmark.is_active.readable = False
Bookmark.is_active.writable = False

open http://127.0.0.1:8000/myapp/appadmin
```

(17:25) Note that by adding a field to a model, web2py automatically detects it and does the SQL ALTER TABLE stuff.

13.11.8 Integrating with web services

(17:55)

- Integrate with thumbalizer and WOT (Web Of Trust)

13.11.9 Edit the menu

(18:53)

```
edit models/menu.py
```

```
response.menu = (
    (T('Home'), False, URL('default', 'index')),
    (T('My Bookmarks'), False, URL('default', 'bookmarks')),
    )
```

13.11.10 Edit the controller

(19:18)

13.11.11 Show how it works

(21:40)

13.11.12 Customize routes

(22:54)

Edit `routes_in` and `routes_out` in `routes.py`

13.11.13 Authentication using Janrain

(23:49)

Janrain auth - ldap, oauth1, oauth2, google, openid, dropbox

All you have to do is copy `janrain.key` into `private` folder.

13.11.14 Edit templates

(25:15)

13.11.15 Create web services

(28:25)

The old way

```
@service.json
@service.xml
@service.jsonrpc
@service.xmlrpc
@service.amfrpc3('domain')
@service.soap
def linksby(key=''):
    return db(Link.id==Bookmark.link)\
        (Bookmark.tags.contains(key)).select(Link.ALL, distinct=True)
```

The new way

```
@request.restful()
def api():
    def POST():
        raise HTTP(501)
    def GET(keys=''):
        return dict(result=linksby(key).as_list())
    return locals()

def call(): return service()
```

13.11.16 The scheduler

(30:08)

13.11.17 Deploy remotely

(31:19)

Package the app into a w2p file.

13.12 Notes from PDF slides

These are notes from the slides at http://dl.dropbox.com/u/18065445/Slides/PySFTalkSlides.pdf, which did not map closely to the talk.

13.12.1 Main features

- One Instance - Many Applications (hot plug and play)
- Web based Integrated Development Environment
- Web based Database administration (for each app)
- Each application can connect to multiple Databases
- Writes SQL for you
- Strong on Security (no SQL Injections, XSS, CSRF, ..., audited)
- Built-in ticketing system (logs all errors)
- Runs everywhere (it is written in Python)
- Requires NO Installation (just download and unzip)
- Has no coniguration files and no third party dependencies
- Can run off a USB drive
- Always backward compatible (since 2007 and on...)
- 50+ of developers already involved

13.12.2 Admin

13.12.3 Included APIs

- generation and parsing: HTML / XML / RSS / JSON
- web services: JSON / JSON-RPC / XML / XML-RPC / AMF
- document generation WIKI, CSV, RTF, LATEX, PDF
- 10 different SQL dialects and Google App Engine
- Role based access control with login plugins local, OpenID, OAuth, 1 and 2, Janrain, LDAP Consumer + Provider Central Authentication Service
- sending SMS, accepting Credit Card payments
- internationalization, cron jobs, multi-tenancy, ..

13.12.4 web2py architecture

13.12.5 web2py modules

13.12.6 Admin - Design

13.12.7 web2py applications

13.12.8 Architecture of applications

13.12.9 Complete application ("friends")

13.12.10 Controllers

13.12.11 Routing (in, out, onerror)

13.12.12 Templates

13.12.13 App-Admin

13.12.14 Models

13.12.15 Queries

13.12.16 Thread locals

13.12.17 Multi-version/No-conflicts

13.12.18 Role based access control

13.12.19 Web services

```
@service.json
@service.xml
@service.jsonrpc
@service.xmlrpc
@service.soap
@service.amfrpc3('domain')
```

13.12.20 Record versioning

13.12.21 Modularity with digitally signed URLs

13.12.22 Federated authentication

13.12.23 Multi-tenancy

13.12.24 GAE deployment

13.12.25 Web translation

13.12.26 Error logging

13.12.27 Who uses web2py?

3000 registered users

13.12.28 Conclusions

- web2py has been bround for since 2007
- 50% was rewritten in 2010 while mantaining backward compatibility
- Some like it, some find it useful
- Give it a try!

CHAPTER 14

IPython: Python at your fingertips by Fernando Pérez, Min Ragan-Kelley, Brian E. Granger, Thomas Kluyver

Presenters: Fernando Pérez and others

PyCon 2012 presentation page: https://us.pycon.org/2012/schedule/presentation/121/

Slides:

Video: https://www.youtube.com/watch?v=26wgEsg9Mcc&list=PLBC82890EA0228306&index=9&feature=plpp_video

Video running time: 39:58

Resources:

- http://ipython.org/
- http://github.com/ipython
- https://github.com/ipython/ipython-in-depth

(00:26) Talk in two parts:

1. Brief overview of IPython
2. Demo of the new IPython web notebook

14.1 Brief overview of IPython

14.1.1 Why IPython?

(00:40)

the "I" in IPython is for "Interactive".

In scientific computing, we typically *don't know what we're doing*.

Exploratory computing is *not just for scientists*.

14.1.2 Python an excellent *base* for an interactive environment

(01:45)

The default interactive environment is just a starting point.

```
$ python
>>> ls
...
NameError: name 'ls' is not defined
>>> os?
...
SyntaxError: invalid syntax
>>> execfile('~/scratch/err.py')
...
IOError: [Errno 2] No such file or directory '~/scratch/err.py'
>>> execfile('/home/fperez/scratch/err.py')
...
NameError: name 'foo33' is not defined
```

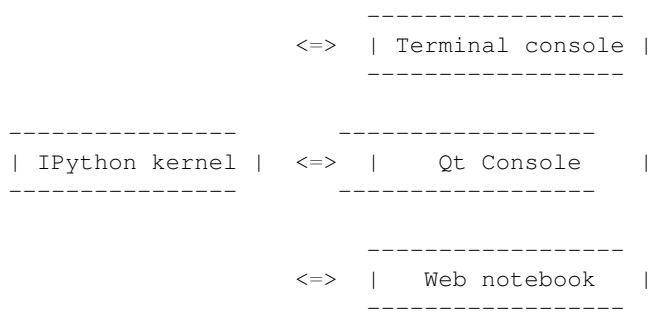
14.1.3 We can do better

(02:37)

```
$ ipython
In [1]: ls ~/scratch/err*py
/home/fperez/scratch/err25.py
...
In [2]: os?
(One question mark - Brings up a whole bunch of info about the os module)
In[3]: os??
(Two question marks - Brings up even more info, displaying the code if possible, syntax highlighted)
...
In[13]: run ~/scratch/error
...
(Traceback in color and shows the relevant lines of code)
```

14.1.4 Terminal IPython vs. QT console vs. Web notebook

(04:13)



(05:32) Several clients now.

(05:56) Qt console

- Feels like a terminal emulator
- But it's a Qt rich text widget that can display graphics, color, multi-line editing, etc.

(06:41) Microsoft Visual Studio 2010 integrated console

See http://pytools.codeplex.com/

(07:44) Browser-based notebook

- rich text
- code
- plots...

(08:30) Interactive and high-level parallel APIs

The same abstractions and communications machinery that controls a single interactive IPython instance can control multiple IPython instances.

Exposes an *IPython cluster* which consists of an *IPython controller* and one or more *IPython engines*.

Enables parallel computing.

14.1.5 A brief history of IPython

(10:20)

- October/November 2001: "just a little afternoon hack"
  - $PYTHONSTARTUP: ipython-0.0.1.py (259 lines)
  - IPP (Interactive Python Prompt) by Janko Hauser (Oceanography)
  - LazyPython by Nathan Gray (CalTech)
- 2002: Drop John Hunter's Gnuplot patches: matplotlib
- 2004: Brian Granger, Min Ragan-Kelly: Parallel on Twisted
- 2005-2009: Mayavi, Wx support, refactoring; slow period.
- 2010: Discover 0MQ, Enthought support.
  - Move to Git/GitHub.
  - Build Qt console (Evan Patterson)
  - Rewrite parallel support with ZeroMQ.
  - Python 3 port (Thomas Kluyver)
- 2011: Web Notebook

14.1.6 Some quick stats

(13:06)

14.1.7 Support

(13:40)

14.1.8 IPython in brief

(14:06)

- A better Python shell

- Embeddable kernel and powerful interactive clients
  - Terminal
  - Qt console
  - Web notebook
- Flexible parallel computing

14.2 Demo of the web notebook

(14:26)

14.2.1 An example of showing Twitter word frequencies

(14:40)

14.2.2 Tour of IPython notebook features

(19:32)

(19:43) pwd works

(19:48) mixing Python and shell - you can interpolate Python variables into shell using a $

14.2.3 Plotting

(20:17) matplotlib plots.

14.2.4 Niceties

You can paste code in with Python prompts and IPython will filter out the Python prompts

Highlight standard error

Detailed tracebacks

(21:00) IPython fetches output asynchronously

(21:15) Cause the kernel to segfault, but your document is saved so you can restart your kernel and get back to work

14.2.5 Markdown

(21:42) You can create Markdown cells

14.2.6 Mathematics

(22:00) Mathematics including LaTeX and MathJax

14.2.7 Displaying complex output types, such as images

(22:14) Can display arbitrary output types such as images and add your own representations

Displaying a local image

Displaying a remote image

14.2.8 Embedding video

(23:29) Video - can embed videos like YouTube videos

14.2.9 Embedding iframes

(24:50)

14.2.10 Loading external code

(26:11)

- matplotlib gallery and other galleries
- Drag and drop a `.py` in the dashboard
- Use `%loadpy` with any local or remote url

14.2.11 SymPy

(27:55)

14.2.12 IPython clusters

(29:42)

14.2.13 Conclusion

(35:20) Done with demo; we have a booth; we're having a sprint

14.3 Questions

(36:20)

Q: How to collaborate on an IPython notebook?

A: Unfortunately it's fairly limited right now. You can share notebook files on GitHub. No real-time synchronization. You can share a notebook by sending a person the URL to the notebook (and you can enable SSL and password protection). One person has to save the notebook and the other person needs to reload it.

(37:40) Q: How to integrate with Sphinx?

A: In a gist, we have a very simple reST exporter. Improving the exporter should be easy so that you can create a Sphinx-compatible notebook. It's not hard to do, but we're busy.

It looks like this might do the trick??? https://github.com/ipython/nbconvert

(38:32) Q: How feasible is it to use IPython as a replacement shell?

A: Ask Mark Wiebe of NumPy who uses the Qt console

(39:00) Q: Debugging in the interactive notebook?

A: If you type qtconconsole then you can connect to a running kernel. We also plan to expose it via the web.

CHAPTER 15

Apache Cassandra and Python

Presenters: Jeremiah Jordan from Morningstar, Inc.

PyCon 2012 presentation page: https://us.pycon.org/2012/schedule/presentation/122/

Slides: http://goo.gl/8Byd8

Video: https://www.youtube.com/watch?v=188mXjwdkak&feature=autoplay&list=PLBC82890EA0228306&lf=plpp_video&playnext=

Video running time: 31:00

15.1 Why are you here?

(00:24)

- You were too lazy to get out of your seat.
- Someone said "NoSQL".
- You want to learn about using Cassandra from Python.

15.2 What am I going to talk about?

(00:34)

- What is Cassandra
- Starting up a local dev/unit test instance
- Using Cassandra from Python
- Indexing/Schema design

15.3 What am I not going to talk about?

(00:55)

- Setting up and maintaining a production cluster

15.4 Where can I get the slides?

(01:07)

http://goo.gl/8Byd8

points to

http://www.slideshare.net/jeremiahdjordan/pycon-2012-apache-cassandra

15.5 What is Apache Cassandra (Buzz Word description)?

(01:23)

"Cassandra is a highly scalable, eventually consistent, distributed, structured key-value store. Cassandra brings together the distributed systems technologies from Dynamo and the data model from Google's BigTable. Like Dynamo, Cassandra is eventually consistent. Like BigTable, Cassandra provides a ColumnFamily-based data model richer than typical key/value systems."

From the Cassandra Wiki: http://wiki.apache.org/cassandra/

15.6 What is Apache Cassandra?

(01:58)

- Column based key value store (multi level dictionary)
- Combination of Dynamo (Amazon) and BigTable (Google)
- Schema-optional

15.7 Basic structure of data in Cassandra

(02:26)

- A *keyspace* is kind of like a schema in a relational database.
- A *column family* is kind of like a table in a relational database.

(02:46)

- Keys are in a random order; the recommended way of using Cassandra; allows distributing things evenly over your cluster
- With ordered keys, you have to be careful not to create hotspots in your cluster
- Column names are sorted in alphabetic order
- You can optionally type values
- Every value has a timestamp associated with it; used for conflict resolution – make sure that clocks are in sync.

15.8 Multi-level Dictionary

(05:08)

```
Column Family
     |       Columns
     ↓          |
 {"UserInfo":   ↓
     {"John": {"age": 32,
        ↑      "email": "john@gmail.com",
        |      "gender": "M",
        |      "state": "IL"}}}
        |                ↑
       Key               |
                       Values
```

15.9 Well, really an *ordered* dictionary

```
{"UserInfo":
    {"John":
        OrderedDict(
            [("age", 32),
             ("email", "john@gmail.com"),
             ("gender", "M"),
             ("state", "IL")])}}
```

15.10 Where do I get it?

(05:30)

From the Apache Cassandra project:

http://cassandra.apache.org/

or DataStax hosts some Debian and RedHat packages:

http://www.datastax.com/docs/1.0/install

15.11 How do I run it?

(05:47)

Edit `conf/cassandra.yaml`:

- Change data/commit log locations
- defaults: `/var/cassandra/data` and `/var/cassandra/commitlog`

Edit `conf/log4j-server.properties`:

- Change the log location/levels
- default: `/var/log/cassandra/system.log`

(06:10)

Edit `conf/cassandra-env.sh` (`bin/cassandra.bat` on Windows)

- Update JVM memory usage
- default: 1/2 your RAM

```
$ ./cassandra -f      # -f means launch in foreground
```

15.12 Setup up tips for local instances

(06:43)

Make templates out of `cassandra.yaml` and `log4j-server.properties`

Update `cassandra` script to generate the actual files

(run them through **sed** or something).

15.13 Server is running, what now?

(07:10)

```
$ ./cassandra-cli
connect localhost/9160;

create keyspace ApplicationData
    with placement_strategy =
        'org.apache.cassandra.locator.SimpleStrategy'
    and strategy_options =
        [{replication_factor:1}];
```

(08:20)

```
use ApplicationData;

create column family UserInfo
    and comparator = 'AsciiType';

create column family ChangesOverTime
    and comparator = 'TimeUUIDType';
```

15.14 Connect from Python

http://wiki.apache.org/cassandra/ClientOptions

Thrift - See the "interface" directory (Do not use!!!)

Pycassa - `pip install pycassa` – the one we'll talk about today

Telephus (twisted) - `pip install telephus`

DB-API 2.0 (CQL) - `pip install cassandra-dbapi2`

15.15 Thrift (don't use it)

(10:48)

15.16 Pycassa

(10:55)

```
import pycassa
from pycassa.pool import ConnectionPool
from pycassa.columnfamily import ColumnFamily

pool = ConnectionPool('ApplicationData',
                      ['localhost:9160'])
col_fam = ColumnFamily(pool, 'UserInfo')
col_fam.insert('John', {'email': 'john@gmail.com'})
```

http://pycassa.github.com/pycassa/

http://github.com/twissandra/twissandra/ – An example application; Twitter clone using Django and Pycassa

15.17 Connect

(11:22)

```
"""                       Keyspace
                              |
                              ↓                  """
pool = ConnectionPool('ApplicationData',
                      ['localhost:9160'])
"""                           ↑
                              |
                          Server list            """
```

Cassandra scales very linearly. Netflix has some nice papers online about it.

15.18 Open Column Family

(12:38)

```
"""                     Connection Pool
                           |
                           ↓                 """
col_fam = ColumnFamily(pool, 'UserInfo')
"""                           ↑
                              |
                        Column Family   """
```

15.19 Write

(12:50)

```
col_fam.insert('John', {'email': 'john@gmail.com'})
```

15.20 Read

(13:03)

```
readData = col_fam.get('John',
                       columns=['email'])
```

15.21 Delete

(13:18)

```
col_fam.remove('John',
               columns=['email'])
```

15.22 Batch

(13:23)

```
col_fam.batch_insert(
    {'John': {'email': 'john@gmail.com',
              'state': 'IL',
              'gender': 'M'},
     'Jane': {'email': 'jane@python.org',
              'state': 'CA'
              'gender': 'M'}})
```

15.23 Batch (streaming)

(13:44)

```
b = col_fam.batch(queue_size=10)

b.insert('John',
         {'email': 'john@gmail.com',
          'state': 'IL',
          'gender': 'M'})

b.insert('Jane',
         {'email': 'jane@python.org',
          'state': 'CA'})

b.remove('John', ['gender'])
b.remove('Jane')
b.send()
```

15.24 Batch (Multi-CF)

(14:39)

```
from pycassa.batch import Mutator
import uuid

b = Mutator(pool)

b.insert(col_fam,
         'John', {'gender': 'M'})

b.insert(index_fam,
         '2012-03-09',
         {uuid.uuid1().bytes:
                'John:gender:F:M'})
```

15.25 Batch Read

(15:28)

```
readData = col_fam.multiget(['John', 'Jane', 'Bill'])
```

15.26 Column Slice

(15:42)

```
d = col_fam.get('Jane',
                column_start='email',
                column_finish='state')

d = col_fam.get('Bill',
                column_reversed = True,
                column_count=2)

startTime = pycassa.util.convert_time_to_uuid(time.time() - 600)

d = index_fam.get('2012-03-31',
                  column_start=startTime,
                  column_count=30)
```

15.27 Types

(17:00)

```
from pycassa.types import *

col_fam.column_validators['age'] = IntegerType()

col_fam.column_validators['height'] = FloatType()
```

```
col_fam.insert('John', {'age': 32, 'height': 6.1})
```

15.28 Column Family Map

(17:52)

```
from pycassa.types import *

class User(object):
    key = Utf8Type()
    email = AsciiType()
    age = IntegerType()
    height = FloatType()
    joined = DateType()

# '(18:48) <https://www.youtube.com/watch?v=188mXjwdkak&feature=autoplay&list=PLBC82890EA0228306&lf=p

from pycassa.columnfamilymap import ColumnFamilyMap

cfmap = ColumnFamilyMap(User, pool, 'UserInfo')
```

15.29 Write

(19:05)

```
from datetime import datetime

user = User()
user.key = 'John'
user.email = 'john@gmail.com'
user.age = 32
user.height = 6.1
user.joined = datetime.now()
cfmap.insert(user)
```

15.30 Read/Delete

(19:37)

```
user = cfmap.get('John')

users = cfmap.multiget(['John', 'Jane'])

cfmap.remove(user)
```

15.31 Timestamps/consistency

(20:09)

```
col_fam.read_consistency_level = ConsistencyLevel.QUORUM
col_fam.write_consistency_level = ConsistencyLevel.ONE

col_fam.get('John',
            read_consistency_level=ConsistencyLevel.ONE)

col_fam.get('John',
            include_timestamp=True)
```

A quorum is n / 2 nodes.

15.32 Indexing

(22:00)

Native secondary indexes

Roll your own with wide rows

15.33 Indexing Links

(22:09)

Intro to indexing

- http://www.datastax.com/dev/blog/whats-new-cassandra-07-secondary-indexes

Blog post and presentation going through some options

- http://www.anuff.com/2011/02/indexing-in-cassandra.html
- http://www.slideshare.net/edanuff/indexing-in-cassandra

Another blog post describing different patterns for indexing

- http://pkghosh.wordpress.com/2011/03/02/cassandra-secondary-index-patterns

15.34 Native Indexes

(21:20)

Easy to add, just update the schema

Can use filtering queries

Not recommended for high cardinality values (i.e.: timestamps, birth dates, keywords, etc.)

Makes writes slower to indexed columns

15.35 Add index

(23:29)

```
update column family UserInfo
    with column_metadata = [
        {column_name: state,
         validation_class: UTF8Type,
         index_type: KEYS};
```

15.36 Native indexes

(23:52)

```
from pycassa.index import *

state_expr = create_index_expression('state', 'IL')
age_expr = create_index_expression('age', 20, GT)
clause = create_index_clause([state_expr, age_expr], count=20)

for key, userInfo in col_fam.get_indexed_slices(clause):
    # Do stuff
```

15.37 Rolling your own

(24:40)

Removing changed values yourself

Know the new value doesn't exist, no read before write

Index can be denormalized query, not just an index

Can use things like composite columns and other tricks

15.38 Questions

(25:13)

CHAPTER 16

Practicing Continuous Deployment by David Cramer of DISQUS

Presenters: David Cramer from DISQUS

PyCon 2012 presentation page: https://us.pycon.org/2012/schedule/presentation/12/

Slides:

Video: http://www.youtube.com/watch?v=QGfxLXoMpPk

Video running time: 41:20

16.1 What do we mean by continuous deployment?

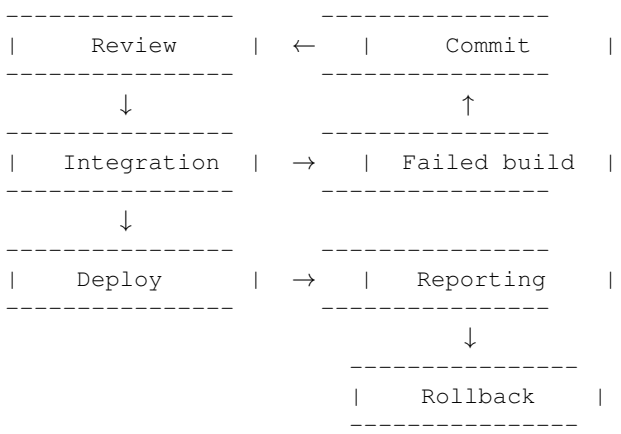
(0:27)

Shipping new code as soon as it's **ready**

16.2 What does "ready" mean?

(0:54)

- Reviewed by peers
- Passes automated tests – continuous integration is essential
- Some level of QA



(02:43)

Continuous deployment does not necessarily mean that you deploy all the time or every 5 minutes.

You can deploy as often as you want. The important thing is that you can deploy **whenever you want**.

16.3 The good and the bad

(3:14)

16.3.1 The good

- Develop features incrementally
- Release frequently
- Smaller doses of QA

16.3.2 The bad

- Culture shock
- Stability depends on test coverage
- Initial time investment

16.3.3 At Disqus

- DISQUS is a company of about 30 people.
- Spent the last 2 years working on infrastructure - automated testing tools, reporting, etc.
- We have a guy dedicated to tests and a guy dedicated to releasing.
- You really have to instill this in your culture.

16.4 Keep development simple

(4:48)

- **Automate testing** of complicated processes and architecture
- Simple can be better than complete
  - Especially for local development
- `python setup.py {develop,test}`
- Puppet, Chef, Buildout, Fabric, etc.

Automated testing is a requirement. Continuous integration is the basis for all of this.

David feels that packaging your app is essential, because you need things to be repeatable.

16.4.1 Bootstrapping local

(7:32)

- Simplify local setup
  - `git clone dcramer@disqus:disqus.git`
  - `make`
  - `python manage.py runserver`
- Need to test dependencies?
  - virtualbox + `vagrant up` (Link to Vagrant)

16.5 Progressive rollout

(9:01)

We **actively use early versions of features** before public release.

At DISQUS, we do about 12,000 to 15,000 requests/second and we peak much higher than that.

It's important that a feature doesn't take the site down. We want to slowly release features.

(9:32) Feature flippers or switches

Deploy features to portions of a user base at a time to ensure **smooth, measurable releases**

They use a platform called Gargoyle – currently very Django-specific, but trying to generalize it to be Django-agnostic and maybe even language-agnostic.

Example:

1. Only enable this new feature for internal users.
2. OK, now turn it on for 1% of our base.
3. Keep bumping up until we know it's scalable.

16.6 Iterate quickly by hiding features

Early adopters are free QA

```
from gargoyle import gargoyle

def my_view(request):
    if gargoyle.is_active('awesome', request):
        return 'new happy version :D'
    else:
        return 'old sad version :('
```

New users can check a box to volunteer to test bleeding edge features.

16.7 Review all the commits

(11:42)

Phabricator - a code review tool open-sourced by Facebook.

(12:40) When you do a code review, it's done through a commit - friendly for developers. Don't have to use the Web UI.

`arc diff` runs a set of lints and runs your tests for you.

They've released a plugin for nose called quickunit.

16.8 Integration == Jenkins

(15:10)

16.8.1 Integration requirements

(15:45)

- Developers *must* know when they've broken something
  - IRC, Email, IM
- Support proper reporting
  - XUnit, Pylint, Coverage.py
- Painless setup
  - `apt-get install jenkins`

It's important for developers to know right away when stuff is broken so they can ideally fix it before they've context switched to something else.

16.8.2 Integration issues

False positives

- Reporting isn't accurate
- Services fail (even a third party service)
- Bad tests

Test coverage

- Regressions on untested code

Feedback delay

- Integration tests vs. unit tests

16.8.3 Fixing false positives

(18:00)

- Rerun tests several times on failure
- Report continually failing tests
- Replace external service tests with a functional test suite

16.8.4 Maintaining coverage

(18:38)

- Raise awareness with reporting
  - Fail/alert when coverage drops on a build
- Commit tests with code
  - Coverage against commit diff for untested regressions
- Utilize code review

16.8.5 Speeding up tests

(20:05)

This where almost all of our time has gone.

At one point our test suite took 40 minutes to an hour.

- Write unit tests
  - vs. slower integration tests
- Mock external services
- Distributed and parallel testing
  - Matrix builds

16.9 Reporting

(22:24)

<You> Why is mongodb-1 down?

<Ops> It's down? Must have crashed again.

16.9.1 Meaningful metrics

- Rate of traffic (not just hits!)
  - Business vs. system
- Response time (database, web)
- Exceptions

- Social media
  - Twitter

16.9.2 Tools

(24:18)

Beyond Nagios and PagerDuty.

16.9.3 Graphite

Tracks and graphs metrics

graphite.wikidot.com

We send it response times, counters, disk space usage

16.9.4 Sentry

https://www.getsentry.com/

You can now use it even if you're not using Django.

It's designed to receive exceptions and track them.

16.10 Wrap up

(26:08)

Deployment - the least important part of continuous deployment. Everyone solves it differently.

What DISQUS does. Ship a relocatable virtualenv as a tarball.

16.10.1 Getting Started

(27:02)

- Package your app
- Value code review
- Ease deployment, **fast rollbacks**
- Setup automated tests
- Gather some easy metrics

16.10.2 Going further

(29:00)

- Build an immune system – automatically rolls back if some metric goes down – very interesting, but very risky
  - Automate deploys, rollbacks (maybe)

- Adjust to your culture
  - There is no "right way"
- SOA == great success

16.11 Questions?

(30:25)

Code reviews: Before Phabricator, DISQUS used GitHub pull requests but they found it to not be scalable.

(31:49) Selenium tests – we deleted all our Selenium tests. We're reimplementing some of them, but simpler.

(32:50) How many times a day do you deploy? At minimum, once a day. Lately, it's been no more than half a dozen times per day.

(33:55) Why do you roll back? Why not fix it and move forward? Sometimes it might take a while to fix it.

(34:30) What do you do about database changes? Especially for rollbacks. Google DISQUS schema changes or David Cramer schema changes

(35:52) Any code review policies? Maximum # of lines or maximum amount of time until review. Current standard is at the start of the day and the end of the day, you must clean your slate. Even this kind of sucks, because you may have to wait a day to get your change reviewed. What we really want is to give a max of 20 minutes and if it isn't reviewed, then it automatically gets assigned to someone else.

(37:23) Numbers of production servers? 200ish. 4 billion pageviews.

(38:00) How long does it take to deploy? Ashamed to admit it. All of our servers in one location although we push a lot of stuff to Akamai.

(39:00) One monolithic deploy or many? We're moving towards SOA and away from monolithic.

(40:15) Can you tell us about your rollback process? At one point, it was just swap the symlink and restart the servers.

(40:33) Business metrics measurements - what tools? Graphite, statsd, porkchop

(41:10) Done.

CHAPTER 17

Indices and tables

- *genindex*
- *modindex*
- *search*