dirt Documentation

Release 0.6

Andy Mastbaum

Sep 27, 2017

Contents

dirt is a Python and CouchDB application for oversight and tracking of remotely-executed jobs.

Source code is available at http://github.com/mastbaum/dirt.

CHAPTER 1

Quick Start Guide

Want to see dirt in action, fast?

First, set up a passphrase-less ssh key to localhost. Then do this:

```
$ cd dirt && python setup.py install
$ cd && dirt create myproject
$ cd myproject/web && ./egret push http://localhost:5984/myproject && ./egret␣
↪pushdata http://localhost:5984/myproject test_data.json && cd ..
$ dirt updatenodes localhost
$ dirt serve
```

Visit the site and watch the results roll in.

CHAPTER 2

Installation

Requirements:

- CouchDB >= 1.1.0 (http://apache.couchdb.org)
- Python 2.6+

`dirt` is packaged for easy installation with `setuptools`:

```
$ git clone git://github.com/mastbaum/dirt.git
$ cd dirt
$ python setup.py install
```

or `pip`:

```
$ pip install -e git+git://github.com/mastbaum/dirt.git#egg=dirt
```

CHAPTER 3

Documentation

Introduction

dirt is a Python and CouchDB application for oversight and tracking of remotely-executed jobs.

Users submit groups of tasks to perform on a data set (or whatever), those are added to a database, tasks are doled out to remote execution hosts, and results are stored in the DB and presented via a web page.

Possible uses include distributed continuous integration, build testing, and automated data processing.

Stories

1. You want to run a series of compilation and unit tests on each revision of your software. You'd like to store the results in a database, and easily see things like all the tests on revision X or the history of unit test Y through the revisions.
2. Your science experiment regularly produces data files as output. You need to perform some analyses on each file, and store the results.

dirt is designed to make such tasks trivial. Basically, you have a fundamental data set – a code revision, a data file, etc., called a "record" – and a number of functions that you want to operate on this data set. With dirt, you simply:

1. Express your tasks as Python modules
2. Add a record to the database
3. Add tasks associated with that record to the database

and dirt will automatically run them and put results in a database, doling out tasks to as many computers as you make available to it.

Database Backend

dirt uses CouchDB for its database. Couch was chosen for interoperability with various other systems, but the dirt data model is easily normalized and trivially reimplemented in traditional SQL.

The `dirt` CouchApp is compiled and managed with egret, a very lightweight pure Python CouchApp authoring tool.

Web Frontend

Since Couch is used for the DB, it made sense to write the frontend using CouchDB views. The frontend of dirt is a set of pages that are built dynamically using Ajax via the jQuery CouchDB API. There is no reason the website couldn't be served by a standalone web server.

Remote Execution

Tasks are run on remote hosts via `execnet`. Task code is shipped over and executed on the remote host using a secure SSH pipe provided by the execnet package. Clients take no action, and need only an SSH key, a Python interpreter, and any dependencies not included with the task code.

Tasks are registered by name, and doled out to remote hosts for execution as their names appear attached to records in the database. For example, a build testing system might look like this:

- record: rev1234
- task: build(linux) -> Python code for "build" is run on a host with platform "linux"
- task: build(osx) -> Python code for "build" is run on a host with platform "osx"
- task: cxxtest -> Python code for cxxtest is run on some host

For security reasons, the code that actually gets executed by remote hosts is stored in actual Python files on the master server. Having it in the database would be cool, but more prone to evil.

Basic Usage

Command-line usage

dirt can be run with one of three subcommands.

Start a new dirt project:

```
$ dirt create projectname [database name]
```

The database name is the same as the project name by default.

Update stored system information on each host, adding the host to the database if necessary:

```
$ dirt updatenodes [host1.example.com] [host2.othersite.org] ...
```

Run the dirt remote execution server, which will dole out unfinished tasks in the database to available execution hosts:

```
$ dirt serve
```

Using dirt as a module

dirt and its submodules can also be used in Python.

To get a list of nodes:

```
>>> from dirt.core import dbi
>>> db = dbi.DirtCouchDB('http://localhost:5984', 'dirt')
Sep 04 03:17:44 neutralino dirt : Connected to db at http://localhost:5984/dirt
>>> for fqdn in db.get_nodes():
..:     print fqdn
node1.example.com
node2.othersite.org
```

Use `execnet` to run a task on a node:

```
>>> host = 'localhost'
>>> import execnet
>>> from tasks.examples import simple
>>> gw = execnet.makegateway('ssh=%s' % host)
>>> ch = gw.remote_exec(simple)
>>> ch.receive()
{'success': True}
```

Document Model

The fundamental objects in dirt projects are "records" and "tasks." A record might be a code revision or a data set, on which you wish to perform several operations (like compiling or running some unit tests). Tasks must be associated with one and only one record.

Records and Tasks

Records (representing a code revision, data file, etc.) and the tasks associated with them are expressed in JavaScript Object Notation (JSON). A record document (in the CouchDB sense) looks like this:

```
{
    "_id": "r123",
    "type": "record",
    "description": "this is revision one two three",
    "created": 1315347385
}
```

A task ("example") associated with this document ("r123") looks like this:

```
{
    "_id": "2e3dabbff38ca7f6fa05c5a0cbbc95a4",
    "type": "task",
    "record_id": "r123",
    "name": "example",
    "platform": "linux",
    "created": 1315347385
}
```

The `name` field is simply the name of a Python module in the `tasks` directory. A directory `tasks/examples` is included with new projects.

The structure of the Task modules is flexible, but when `__name__ == '__channelexec__'`, they must return a dictionary containing at least a boolean `'success'`.

The recommended structure is:

```
def execute():
    '''docstring'''
    # do things
    success = False
    sum = 2 + 2
    if sum == 4:
        success = True
    return {'success': success, 'sum': sum}

if __name__ == '__channelexec__':
    channel.send(execute())

if __name__ == '__main__':
    print execute()
```

Getting attachments from tasks

Tasks can also return attachments. Simply include in the "results" dictionary another special key "attachments," which contains a list like this:

```
'attachments': [
    {'filename': <local filename>, 'contents': <stringified contents of file>, 'link_
↪name': <name to appear on web page>},
    {...},
    {...},
    ...
]
```

If `link_name` is specified, a link is provided to that attachment on the results page.

Passing arguments to tasks

It is also possible to pass keyword arguments to your task (for example, which revision number to check out). The basic syntax for the task module is:

```
def execute(foo, bar):
    return {'success': True, 'foo': foo, 'bar': bar}

if __name__ == '__channelexec__':
    kwargs = channel.receive()
    results = execute(**kwargs)
    channel.send(results)
```

The task document must then include all needed arguments in a special key `'kwargs'`, e.g.:

```
'kwargs': {'foo': 42, 'bar': 'baz'}
```

An example is given in the `tasks/examples` subdirectory of a new project.

Setting system requirements for tasks

Some tasks may require certain conditions on the slave node. For example, a compilation test may need to be run on several platforms. The node information stored by dirt is available to tasks through a simple query API.

To set a requirement for a task, add the special key `requires` to the task document. `requires` must be a list of strings invoking one of the following query operators:

```
is
not
in
not_in
```

Example:

```
{
    "type": "task",
    "requires": ["architecture is x86_64", "G4INSTALL in environ"],
    ...
}
```

Available keys

`success` (boolean) True if node was successfully added to the database

`cpu_count` (int) Number of CPUs available for running dirt tasks. dirt will run up to `cpu_count` jobs simultaneously on a node.

From `multiprocessing.cpu_count`

`platform` (string) Long description of the node platform, e.g. "Linux-2.6.32-33-server-x86_64-with-Ubuntu-10.04-lucid"

From `platform.platform()`

`architecture` (string) Descriptor of node architecture, e.g. 'x86_64'

From `platform.machine()`

`environ` (string-string map) A dictionary of environment variables on the node, e.g. {'FOO': '/bar'}

From `os.environ.data`

`path` (list of strings) A list of all paths in the $PATH on the node, e.g. ['/bin', '/usr/bin']

From `os.environ['PATH'].split(os.path.pathsep)`

`version_info` (string) Python version on the node, e.g. '2.6.5.final.0'

From `sys.version_info`

`pythonpath` (list of strings) Python path on the node

From `sys.path`

`hostname` (string) Short hostname of the system, e.g. 'node1'. Not used by dirt.

From `socket.gethostname()`

`fqdn` (string) Fully-qualified domain name of the node, e.g. 'node1.site.org'

From `socket.getfqdn`

`ip` (string) Reverse-mapped IP, as seen by the node, e.g. '10.20.30.40'. Not used by dirt.

From `socket.gethostbyname(socket.getfqdn())`

Example Project

For an example, let's build a minimal build testing system.

To see a complete CI system implemented with dirt, check out pytunia.

Starting the project

Use `dirt create` to start a project:

```
$ dirt create builder
dirt v0.6
Created new dirt project in builder
$ cd builder
$ ls
README.md  settings.py  tasks  web
```

Describe your project in `README.md`, and tweak settings as necessary in `settings.py`:

```
$ vim README.md
$ vim settings.py
```

Push the web application to your CouchDB server:

```
$ cd web && ./egret push builder
```

If you're not running the server on localhost, replace "builder" with the full database URL.

Visit the URL it prints, and you should see your project's empty Overview page.

Adding remote execution nodes

You'll now want to give the builder a list of computers it can run on. Currently, these must be accessible to the server via SSH (work is in progress for ways of getting around firewalls). You'll want to set up passphrase-less SSH keys so that the user running the server can log into each node without a password. Once that is done, add the nodes by their full hostname using `dirt updatenodes host1 host2 ...`. For this example, let's just use the server as a node:

```
$ dirt updatenodes localhost
```

Adding tasks

To test compilation, we'll need to express the build process in a Python module. For this example, we'll grab and compile a C++ "hello, world" from github. Consider the following Python module:

```
import os
import subprocess

def system(cmd, wd=None):
    '''a wrapper for subprocess.call, which executes cmd in working directory
    wd in a bash shell, returning the exit code.'''
    if wd:
        cmd = ('cd %s && ' % wd) + cmd
    return subprocess.call([cmd], executable='/bin/bash', shell=True)
```

```
def execute():
    results = {'success': True, 'attachments': []}

    # work in some directory
    wd = 'builder_stuff'
    if not os.path.exists(wd):
        os.mkdir(wd)

    # construct command and check out with git
    url = 'https://github.com/leachim6/hello-world.git'
    cmd = 'git clone ' + url
    ret = system(cmd, wd)

    # if something has gone wrong, we can return a reason
    if ret != 0:
        results['success'] = False
        results['reason'] = 'git clone failed'
        return results

    cmd = 'cd hello-world/c && g++ -v -o hello c++.cpp &> build.log'
    ret = system(cmd, wd)

    if ret != 0:
        results['success'] = False
        results['reason'] = 'g++ failed'
        return results

    # attach build log
    logfile = {}
    with open(wd + '/hello-world/c/build.log','r') as f:
        logfile = {'filename': 'build.log', 'contents': f.read(), 'link_name': 'Build␣
↪log'}

    results['attachments'].append(logfile)

    return results

if __name__ == '__channelexec__':
    results = execute()
    channel.send(results)

if __name__ == '__main__':
    print execute()
```

This will try to clone a git repository and compile some c++ code. If it works, you get the build log as an attachment. If it fails, your results tell you which step failed.

Put this file (or your version of it) in the `tasks` subdirectory, called `compile_hello.py`.

Starting the server

From your project directory, just run:

```
$ dirt serve
```

It is now listening for new tasks.

Adding records and tasks to the database

Records and the tasks that go with them are added directly to the CouchDB database. There are lots of ways of pushing data to couch, including `curl -X PUT ...`, `egret pushdata ...`, any language's couchdb module, etc.

For a real build tester, the record and task documents for each revision should be constructed and posted to the server by some kind of post-commit hook in your version control system. For this example, we will just construct the JSON documents manually. Save the following as r123.json (pretending this code has something to do with revision 123):

```
{
    "docs": [
        {
            "_id": "r123",
            "type": "record",
            "description": "this is revision one two three",
            "created": 1315347385
        },
        {
            "_id": "2e3dabbff38ca7f6fa05c5a0cbbc95a5",
            "type": "task",
            "record_id": "r123",
            "name": "compile_hello",
            "platform": "linux",
            "created": 1315347385
        }
    ]
}
```

This tells dirt to execute the `compile_hello` module (associated with r123) on the next available node (localhost, for us).

To put this in the database:

```
curl -X POST -H "Content-Type: application/json" -d @r123.json http://localhost:5984/
↪builder/_bulk_docs
```

(assuming we're using the couchdb server on localhost).

Watch the magic

The running dirt program should send the `compile_hello` task off to localhost, with output like this:

```
$ dirt serve
dirt v0.6
Sep 07 12:57:20 neutralino myproject : dirt is running...
Sep 07 12:57:20 neutralino myproject : Connected to db at http://localhost:5984/
↪myproject
Sep 07 12:57:20 neutralino myproject : 2e3dabbff38ca7f6fa05c5a0cbbc95a5 -> localhost.
↪localdomain
Sep 07 12:57:22 neutralino myproject : Task 2e3dabbff38ca7f6fa05c5a0cbbc95a5 pushed␣
↪to db
Sep 07 12:57:22 neutralino myproject : Task 2e3dabbff38ca7f6fa05c5a0cbbc95a5: file␣
↪build.log attached
```

Now, go the project URL (e.g. http://localhost:5984/myproject/_design/myproject/index.html), and see the results in the web interface. Clicking on r123 brings you to the record summary page. You can see the build log and raw results

dictionary from the "Results" links. Clicking the task name brings you to the task history page – the outcome of all `compile_hello` tasks ever run.

Moving on

Now, experiment with writing your own task modules. Consider writing code to generate and POST the record and task JSON, as would be called in a post-commit hook. Tinker with the web interface either cosmetically (CSS is in web/static/css) or by writing your own CouchDB views and lists to do special things with the results dictionary.

If you find a bug or have a suggestion for dirt, post an issue on the github page.

core API

The `dirt.core` module contains all of dirt's internal functions.

dirt.core.create

Routine used to create the skeleton directory structure for a new project

dirt.core.dbi

dirt's interface to the CouchDB database. All db interactions happen through a shared instance of the `DirtCouchDB` class.

class `dirt.core.dbi.`**`DirtCouchDB`**(*host*, *dbname*)
wrapper for `couchdb` that provides some additional dirt-specific functions.

`disable_node`(*fqdn*)
set a node's `enabled` flag to false

`get_nodes`()
query db to get node data

`get_tasks`()
a wrapper for couchdb changes which should never die

`push_results`(*results*, *id*, *node_id*, *gateway*)
update task document with results

`save`(*doc*)
save a document in the db

`dirt.core.dbi.`**`check_requirements`**(*db*, *id*, *node*)
check if system requirements defined in the task document are satisfied by this node. some reinvention of the query language is worth it for not eval()-ing stuff from the database.

dirt.core.server

The main dirt server function

dirt.core.load_balance

Load-balancing between nodes is achieved with Python generators that yield the next node on which to execute something.

dirt.core.load_balance.**load**(*db*, *margin=0.9*, *wait=20*)
round-robin using up to n cpus depending on current load

dirt.core.load_balance.**round_robin**(*db*)
generates documents of nodes available for execution in a simple round-robin fashion. re-gets the node list from the db each time around.

dirt.core.log

dirt.core.log creates a singleton yelling.Log instance dirt.core.log.log, used for log output throughout dirt. singleton logger object

dirt.core.yelling

Logging is done with the yelling module, available at https://github.com/mastbaum/yelling.

All logging should happen through dirt.core.log.log, which is a yelling Log object:

class dirt.core.yelling.**Log**(*filename*, *service_name=None*, *hoststamp=True*, *timestamp=True*, *console=True*)
stores all the options for yelling.log, useful for frequent logging without the boatload of options

write(*message*)
write to log file

Other available yelling functions include:

class dirt.core.yelling.**Log**(*filename*, *service_name=None*, *hoststamp=True*, *timestamp=True*, *console=True*)
stores all the options for yelling.log, useful for frequent logging without the boatload of options

write(*message*)
write to log file

dirt.core.yelling.**email**(*recipients*, *subject*, *message*, *sender=None*)
sends a good, old-fashioned email via smtp

dirt.core.yelling.**http_post**(*url*, *params*)
post some key-value pairs to a url with an http post

dirt.core.yelling.**log**(*filename*, *message*, *service_name=None*, *hoststamp=True*, *timestamp=True*, *console=True*)
writes a message to a log file. optionally, write a time and hostname stamp like syslog. if you want to customize that, just put it in message.

dirt.core.yelling.**sms**(*phone*, *carrier*, *subject*, *message*, *sender=None*)
sends an sms message to a phone via email. this is a little dicey since carriers may change their domains at any time.

dirt.core.yelling.**sms_carriers**()
returns a list of known sms carriers

Core tasks API

dirt itself contains a minimum of "core" tasks – only those needed to set up the remote execution environment. More interesting tasks are implemented in derivative projects.

dirt.tasks.system_info

Returns useful system information from os, sys, and socket.

`dirt.tasks.system_info.`**`execute`**`()`
get some basic system information

dirt.tasks.ping

A special task that does not return a dictionary, only a boolean. Used internally by dirt to confirm a node is accepting connections before sending it a task.

dirt.tasks.load

Returns the current short-term load average.

`dirt.tasks.load.`**`execute`**`()`
get current system load

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index