Functional testing with Docker

George Schneeloch

About Core

- Analysis of mass spec data
- Internal PHP web application, LAMP stack, Python test framework, C++ utilities
- Messy legacy codebase, few units to test
- Average workflow takes a half hour to complete

Functional testing

- Decided on functional tests as the quickest route toward basic test coverage
- Functional tests are slow, working with a clean slate is slower
- Testing without a clean slate adds unnecessary complication

Docker

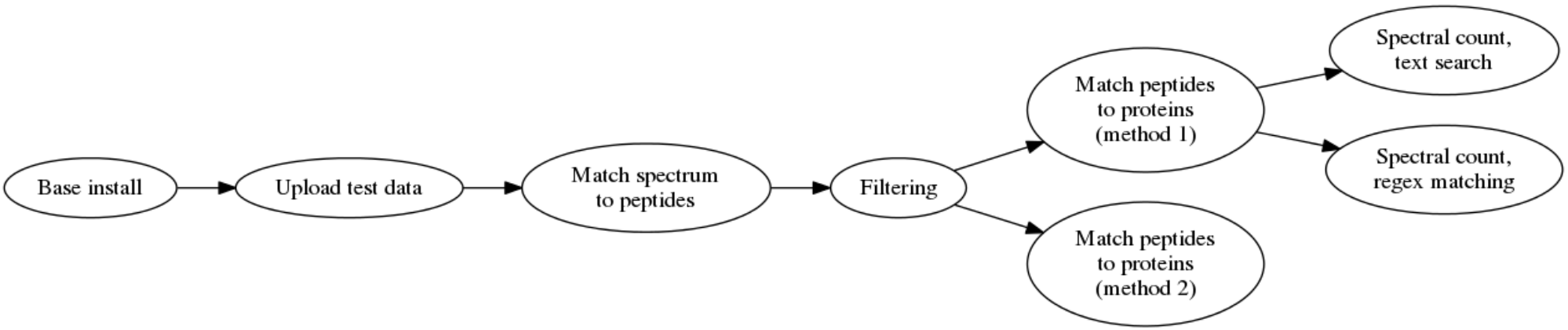
- Offers easy and lightweight isolation of processes (incl. files, network, users)
- Quick automatic snapshots of entire file system

Core's tests

- Tests use Docker to manage state
- Startup and teardown for each test is quick
- Docker images are useful for debugging

Dependencies

- Tests depend on other tests' images
- Nose plugin was created to enforce test dependencies
- `@requires` decorator specifies dependencies



Summary

- Jenkins installs software onto clean Docker image after every commit
- Tests form a dependency graph, are executed in order satisfying requirements
- Each test runs in a Docker container
- Results are saved in a Docker image named after the test

test_core.py

```
import unittest
from dependency_tests.plugin import requires

class Core(unittest.TestCase, Framework):
    # ...

    @requires("test_mzloader_and_sequest")
    def test_protein_assembler(self):
        self.post_to_api('bulk_queue',
                         {'items' : [
                             {'item': 'ExistingSavedSet',
                              'parameters' : {
                                  'set_ids' : [2]
                              }},
                             {'item' : 'ProteinAssembler',
                              'parameters' : {
                                  'map_name' : '',
                                  'map_notes' : '',
                                  'collapse_type' : 'greedy',
                                  'match_type' : 'string',
                                  'do_collapse' : 1,
                                  'enzyme' : 1}}
                         ]})
        self.wait_for_success() # ping API every 10 seconds to see if everything finished yet

        actual = self.get_from_api("protein_assembler/1")
        expected = self.read_expected("tsv") # file in expected/ directory named after test

        self.assertTsvEqual(expected, actual)
```

run_core.py, inside Docker container

```
class State:
    def __init__(self):
        self.job_daemon = None
        self.mysql = None

    def run(self):
        subprocess.Popen(["/usr/sbin/apachectl", "start"])
        self.mysql = subprocess.Popen(["/usr/bin/mysqld_safe"])
        print "Waiting for mysql to start..."
        wait_for(lambda: subprocess.call(
            ["mysqladmin", "-u", "root", "-proot", "ping"],
            stdout=null_file, stderr=null_file) == 0)

        print "Waiting for apache to start..."
        wait_for(lambda: no_exception(
            lambda: urllib2.urlopen("http://127.0.0.1/").read()))

        self.job_daemon = subprocess.Popen(
            ["/bin/su", "gfyadministrator", "-c", "/usr/bin/php job_daemon.php"],
            cwd="/usr/share/gfy/cli")
        print "Started core. Kill process when done!"
        signal.signal(signal.SIGTERM, self.handle_sigterm)
        signal.signal(signal.SIGINT, self.handle_sigterm)
        signal.pause()

if __name__ == "__main__":
    State().run()
```

```
# run_core.py continued...

    def handle_sigterm(self, signum, frame):
        signal.signal(signal.SIGTERM, signal.SIG_DFL)
        signal.signal(signal.SIGINT, signal.SIG_DFL)
        print "Received SIGTERM, %s, %s" % (signum, frame)
        print "Terminating job daemon..."
        if self.job_daemon:
            self.job_daemon.terminate()
        print "Terminating apache..."
        subprocess.check_call(["/usr/sbin/apache2ctl", "stop"])
        print "Terminating mysql..."
        if self.mysql:
            subprocess.check_call(["mysqladmin", "-proot", "shutdown"])

        if self.job_daemon:
            print "Waiting on job daemon..."
            self.job_daemon.wait()
        print "Waiting on apache..."
        wait_for(lambda: not no_exception(
            lambda: urllib2.urlopen("http://127.0.0.1/").read()))

        print "Waiting on mysql..."
        wait_for(lambda: subprocess.call(
            ["mysqladmin", "-u", "root", "-proot", "ping"],
            stdout=null_file, stderr=null_file) != 0)

        exit(0)
```

framework.py

```
def copy_from_docker(client, container_id, src, dest):
    """Workaround for Docker API for cp. It provides a tarball byte string"""
    reply = client.copy(container_id, src)
    filelike = io.BytesIO(reply.read())
    tar = tarfile.open(fileobj = filelike)
    file = tar.extractfile(os.path.basename(src))
    with open(dest, 'wb') as f:
        f.write(file.read())

def run_docker_image(client, image_name, hostname):
    print("Starting image %s" % image_name)

    container_id = client.create_container(image_name,
        ["/usr/bin/python", "/tmp/vagrant/run_core.py"],
        ports=[80], hostname=hostname)["Id"]
    if not container_id:
        raise Exception("docker run command failed for image %s" % image_name)


    client.start(container_id)

    return container_id
```

framework.py

```
class Framework(object):
    #...
    def read_expected(self, kind, name=None):
        if not name:
            name = self.get_expected_name()

        if kind == "json":
            if not name.endswith(".json"):
                name += ".json"

            path = os.path.join("expected", name)
            if not os.path.isfile(path):
                return None
            with open(path) as f:
                obj = json.load(f)
                return obj
        elif kind == "tsv":
            if not name.endswith(".tsv"):
                name += ".tsv"

            path = os.path.join("expected", name)
            if not os.path.isfile(path):
                return None
            with open(path) as f:
                # includes header
                return list(row for row in csv.reader(f, delimiter='\t') if len(row) > 0)

        else:
            raise Exception("Unknown parameter for read_expected: %s" % kind)
```

framework.py

```
class Framework(object):
    #...
    def setUp(self):
        # make sure this exists
        self.db_conn = None

        self.test_base_image = "test_deb"
        self.test_prefix = "%s_%s" % (self.test_base_image, self.__class__.__name__)

        self.hostname = "localhost"
        self.db_name = "gfy"
        self.docker_ip = None
        self.container_id = None

        self.docker = docker.client.Client()

        # looks up function object._dependency_list, creates image name like
        # gygilab/test_deb_CLASS_TEST
        # where '_TEST' is blank if dependency is base image
        image = self._get_image()
        self.container_id = run_docker_image(self.docker, image, self.hostname)

        container_info = self.docker.inspect_container(self.container_id)
        self.docker_ip = container_info["NetworkSettings"]["IPAddress"]

        # ... and also some code to ping database and web server to wait for them to wake
```

framework.py

```
class Framework(object):
    def tearDown(self):
        # WARNING: this doesn't work in Python 3 which is what we're using
        # Not sure how to get nose to tell me if an error occurred
        error = sys.exc_info()[0]
        if error:
            traceback.print_exc()

        client = self.docker

        try:
            # print out job logs
            jobs = [job for job in self.get_from_api('job_log')]
            jobs = sorted(jobs, key = lambda x: int(x['id']))
            for job in jobs:
                job_id = job['id']
                for type, size in job['filesizes'].items():
                    if size:
                        with open("./build/%s_%s.%s.log" % (self.test_prefix,
                                                            self._testMethodName,
                                                            type), 'w') as f:
                            f.write(self.get_from_api("job_log/%d?type=%s" % (job_id, type)))

        except Exception as e:
            print("Error reading job logs: %s" % e)
```

framework.py

```
        # tearDown contd...

        # save code coverage results
        try:
            copy_from_docker(client, self.container_id, '/tmp/coverage/clover.xml',
                             './build/clover-%s.xml' % self._testMethodName)
        except Exception as e:
            print("Error copying code coverage results to archive: %s" % e)

        name = "gygilab/%s_%s" % (self.test_prefix, self._testMethodName)

        try:
            if self.db_conn:
                self.db_conn.close()
                self.db_conn = None
        except Exception as e:
            print("Error stopping database: %s" % e)

        try:
            print("Stopping container...")
            self.docker.stop(self.container_id)
            client.commit(self.container_id, name)
        except Exception as e:
            print("Error stopping docker containers: %s" % e)
```

Questions?

George Schneeloch

Gygi Lab at Harvard Medical School

http://github.com/noisecapella/dependency_tests