Test Repository Documentation

Release trunk

Testrepository Contributors

July 09, 2015

Contents

1 Test Repository users manual **3**
- 1.1 Overview . . . 3
- 1.2 Configuration . . . 3
- 1.3 Running tests . . . 4
- 1.4 Listing tests . . . 5
- 1.5 Parallel testing . . . 5
- 1.6 Grouping Tests . . . 6
- 1.7 Remote or isolated test environments . . . 6
- 1.8 Hiding tests . . . 7
- 1.9 Automated test isolation bisection . . . 8
- 1.10 Forcing isolation . . . 8
- 1.11 Repositories . . . 8
- 1.12 setuptools integration . . . 9

2 Design / Architecture of Test Repository **11**
- 2.1 Values . . . 11
- 2.2 Goals . . . 11
- 2.3 Data model/storage . . . 11
- 2.4 Code layout . . . 11
- 2.5 External integration . . . 11
- 2.6 Threads/concurrency . . . 12

3 Development guidelines for Test Repository **13**
- 3.1 Coding style . . . 13
- 3.2 Copyrights and licensing . . . 13
- 3.3 Testing and QA . . . 13

4 Indices and tables **15**

Contents:

CHAPTER 1

Test Repository users manual

1.1 Overview

Test repository is a small application for tracking test results. Any test run that can be represented as a subunit stream can be inserted into a repository.

Typical workflow is to have a repository into which test runs are inserted, and then to query the repository to find out about issues that need addressing. testr can fully automate this, but lets start with the low level facilities, using the sample subunit stream included with testr:

```
# Note that there is a .testr.conf already:
ls .testr.conf
# Create a store to manage test results in.
$ testr init
# add a test result (shows failures)
$ testr load < doc/example-failing-subunit-stream
# see the tracked failing tests again
$ testr failing
# fix things
$ testr load < doc/example-passing-subunit-stream
# Now there are no tracked failing tests
$ testr failing
```

Most commands in testr have comprehensive online help, and the commands:

```
$ testr help [command]
$ testr commands
```

Will be useful to explore the system.

1.2 Configuration

testr is configured via the '.testr.conf' file which needs to be in the same directory that testr is run from. testr includes online help for all the options that can be set within it:

```
$ testr help run
```

1.2.1 Python

If your test suite is written in Python, the simplest - and usually correct configuration is:

```
[DEFAULT]
test_command=python -m subunit.run discover . $LISTOPT $IDOPTION
test_id_option=--load-list $IDFILE
test_list_option=--list
```

1.3 Running tests

testr is taught how to run your tests by interepreting your .testr.conf file. For instance:

```
[DEFAULT]
test_command=foo $IDOPTION
test_id_option=--bar $IDFILE
```

will cause 'testr run' to run 'foo' and process it as 'testr load' would. Likewise 'testr run –failing' will automatically create a list file listing just the failing tests, and then run 'foo –bar failing.list' and process it as 'testr load' would. failing.list will be a newline separated list of the test ids that your test runner outputs. If there are no failing tests, no test execution will happen at all.

Arguments passed to 'testr run' are used to filter test ids that will be run - testr will query the runner for test ids and then apply each argument as a regex filter. Tests that match any of the given filters will be run. Arguments passed to run after a `--` are passed through to your test runner command line. For instance, using the above config example `testr run quux -- bar --no-plugins` would query for test ids, filter for those that match 'quux' and then run `foo bar --load-list tempfile.list --no-plugins`. Shell variables are expanded in these commands on platforms that have a shell.

Having setup a .testr.conf, a common workflow then becomes:

```
# Fix currently broken tests - repeat until there are no failures.
$ testr run --failing
# Do a full run to find anything that regressed during the reduction process.
$ testr run
# And either commit or loop around this again depending on whether errors
# were found.
```

The –failing option turns on `--partial` automatically (so that if the partial test run were to be interrupted, the failing tests that aren't run are not lost).

Another common use case is repeating a failure that occured on a remote machine (e.g. during a jenkins test run). There are two common ways to do approach this.

Firstly, if you have a subunit stream from the run you can just load it:

```
$ testr load < failing-stream
# Run the failed tests
$ testr run --failing
```

The streams generated by test runs are in .testrepository/ named for their test id - e.g. .testrepository/0 is the first stream.

If you do not have a stream (because the test runner didn't output subunit or you don't have access to the .testrepository) you may be able to use a list file. If you can get a file that contains one test id per line, you can run the named tests like this:

    $ testr run –load-list FILENAME

This can also be useful when dealing with sporadically failing tests, or tests that only fail in combination with some other test - you can bisect the tests that were run to get smaller and smaller (or larger and larger) test subsets until the error is pinpointed.

`testr run --until-failure` will run your test suite again and again and again stopping only when interrupted or a failure occurs. This is useful for repeating timing-related test failures.

1.4 Listing tests

It is useful to be able to query the test program to see what tests will be run - this permits partitioning the tests and running multiple instances with separate partitions at once. Set 'test_list_option' in .testr.conf like so:

```
test_list_option=--list-tests
```

You also need to use the $LISTOPT option to tell testr where to expand things:

test_command=foo $LISTOPT $IDOPTION

All the normal rules for invoking test program commands apply: extra parameters will be passed through, if a test list is being supplied test_option can be used via $IDOPTION.

The output of the test command when this option is supplied should be a subunit test enumeration. For subunit v1 that is a series of test ids, in any order, `\n` separated on stdout. For v2 use the subunit protocol and emit one event per test with each test having status 'exists'.

To test whether this is working the *testr list-tests* command can be useful.

You can also use this to see what tests will be run by a given testr run command. For instance, the tests that `testr run myfilter` will run are shown by `testr list-tests myfilter`. As with 'run', arguments to 'list-tests' are used to regex filter the tests of the test runner, and arguments after a '–' are passed to the test runner.

1.5 Parallel testing

If both test listing and filtering (via either IDLIST or IDFILE) are configured then testr is able to run your tests in parallel:

```
$ testr run --parallel
```

This will first list the tests, partition the tests into one partition per CPU on the machine, and then invoke multiple test runners at the same time, with each test runner getting one partition. Currently the partitioning algorithm is simple round-robin for tests that testr has not seen run before, and equal-time buckets for tests that testr has seen run. NB: This uses the anydbm Python module to store the duration of each test. On some platforms (to date only OSX) there is no bulk-update API and performance may be impacted if you have many (10's of thousands) of tests.

To determine how many CPUs are present in the machine, testrepository will use the multiprocessing Python module (present since 2.6). On operating systems where this is not implemented, or if you need to control the number of workers that are used, the –concurrency option will let you do so:

```
$ testr run --parallel --concurrency=2
```

A more granular interface is available too - if you insert into .testr.conf:

```
test_run_concurrency=foo bar
```

Then when testr needs to determine concurrency, it will run that command and read the first line from stdout, cast that to an int, and use that as the number of partitions to create. A count of 0 is interpreted to mean one partition per test. For instance in .test.conf:

```
test_run_concurrency=echo 2
```

Would tell testr to use concurrency of 2.

When running tests in parallel, testrepository tags each test with a tag for the worker that executed the test. The tags are of the form `worker-%d` and are usually used to reproduce test isolation failures, where knowing exactly what test ran on a given backend is important. The %d that is substituted in is the partition number of tests from the test run - all tests in a single run with the same worker-N ran in the same test runner instance.

To find out which slave a failing test ran on just look at the 'tags' line in its test error:

```
======================================================================
label: testrepository.tests.ui.TestDemo.test_methodname
tags: foo worker-0
----------------------------------------------------------------------
error text
```

And then find tests with that tag:

```
$ testr last --subunit | subunit-filter -s --xfail --with-tag=worker-3 | subunit-ls > slave-3.list
```

1.6 Grouping Tests

In certain scenarios you may want to group tests of a certain type together so that they will be run by the same backend. The group_regex option in .testr.conf permits this. When set, tests are grouped by the group(0) of any regex match. Tests with no match are not grouped.

For example, extending the python sample .testr.conf from the configuration section with a group regex that will group python tests cases together by class (the last . splits the class and test method):

```
[DEFAULT]
test_command=python -m subunit.run discover . $LISTOPT $IDOPTION
test_id_option=--load-list $IDFILE
test_list_option=--list
group_regex=([^\.]+\.)+
```

1.7 Remote or isolated test environments

A common problem with parallel test running is test runners that use global resources such as well known ports, well known database names or predictable directories on disk.

One way to solve this is to setup isolated environments such as chroots, containers or even separate machines. Such environments typically require some coordination when being used to run tests, so testr provides an explicit model for working with them.

The model testr has is intended to support both developers working incrementally on a change and CI systems running tests in a one-off setup, for both statically and dynamically provisioned environments.

The process testr follows is:

1. The user should perform any one-time or once-per-session setup. For instance, checking out source code, creating a template container, sourcing your cloud credentials.
2. Execute testr run.
3. testr queries for concurrency.
4. testr will make a callout request to provision that many instances. The provisioning callout needs to synchronise source code and do any other per-instance setup at this stage.

5. testr will make callouts to execute tests, supplying files that should be copied into the execution environment. Note that instances may be used for more than one command execution.
6. testr will callout to dispose of the instances after the test run completes.

Instances may be expensive to create and dispose of. testr does not perform any caching, but the callout pattern is intended to facilitate external caching - the provisioning callout can be used to pull environments out of a cache, and the dispose to just return it to the cache.

1.7.1 Configuring environment support

There are three callouts that testrepository depends on - configured in .testr.conf as usual. For instance:

```
instance_provision=foo -c $INSTANCE_COUNT
instance_dispose=bar $INSTANCE_IDS
instance_execute=quux $INSTANCE_ID $FILES -- $COMMAND
```

These should operate as follows:

- instance_provision should start up the number of instances provided in the $INSTANCE_COUNT parameter. It should print out on stdout the instance ids that testr should supply to the dispose and execute commands. There should be no other output on stdout (stderr is entirely up for grabs). An exit code of non-zero will cause testr to consider the command to have failed. A provisioned instance should be able to execute the list tests command and execute tests commands that testr will run via the instance_execute callout. Its possible to lazy-provision things if you desire - testr doesn't care - but to reduce latency we suggest performing any rsync or other code synchronisation steps during the provision step, as testr may make multiple calls to one environment, and re-doing costly operations on each command execution would impair performance.
- instance_dispose should take a list of instance ids and get rid of them this might mean putting them back in a pool of instances, or powering them off, or terminating them - whatever makes sense for your project.
- instance_execute should accept an instance id, a list of files that need to be copied into the instance and a command to run within the instance. It needs to copy those files into the instance (it may adjust their paths if desired). If the paths are adjusted, the same paths within $COMMAND should be adjusted to match. Execution that takes place with a shared filesystem can obviously skip file copying or adjusting (and the $FILES parameter). When the instance_execute terminates, it should use the exit code that the command used within the instance. Stdout and stderr from instance_execute are presumed to be that of $COMMAND. In particular, stdout is where the subunit test output, and subunit test listing output, are expected, and putting other output into stdout can lead to surprising results - such as corrupting the subunit stream. instance_execute is invoked for both test listing and test executing callouts.

1.8 Hiding tests

Some test runners (for instance, zope.testrunner) report pseudo tests having to do with bringing up the test environment rather than being actual tests that can be executed. These are only relevant to a test run when they fail - the rest of the time they tend to be confusing. For instance, the same 'test' may show up on multiple parallel test runs, which will inflate the 'executed tests' count depending on the number of worker threads that were used. Scheduling such 'tests' to run is also a bit pointless, as they are only ever executed implicitly when preparing (or finishing with) a test environment to run other tests in.

testr can ignore such tests if they are tagged, using the filter_tags configuration option. Tests tagged with any tag in that (space separated) list will only be included in counts and reports if the test failed (or errored).

1.9 Automated test isolation bisection

As mentioned above, its possible to manually analyze test isolation issues by interrogating the repository for which tests ran on which worker, and then creating a list file with those tests, re-running only half of them, checking the error still happens, rinse and repeat.

However that is tedious. testr can perform this analysis for you:

```
$ testr run --analyze-isolation
```

will perform that analysis for you. (This requires that your test runner is (mostly) deterministic on test ordering). The process is:

1. The last run in the repository is used as a basis for analysing against - tests are only cross checked against tests run in the same worker in that run. This means that failures accrued from several different runs would not be processed with the right basis tests - you should do a full test run to seed your repository. This can be local, or just testr load a full run from your Jenkins or other remote run environment.
2. Each test that is currently listed as a failure is run in a test process given just that id to run.
3. Tests that fail are excluded from analysis - they are broken on their own.
4. The remaining failures are then individually analysed one by one.
5. For each failing, it gets run in one work along with the first 1/2 of the tests that were previously run prior to it.
6. If the test now passes, that set of prior tests are discarded, and the other half of the tests is promoted to be the full list. If the test fails then other other half of the tests are discarded and the current set promoted.
7. Go back to running the failing test along with 1/2 of the current list of priors unless the list only has 1 test in it. If the failing test still failed with that test, we have found the isolation issue. If it did not then either the isolation issue is racy, or it is a 3-or-more test isolation issue. Neither of those cases are automated today.

1.10 Forcing isolation

Sometimes it is useful to force a separate test runner instance for each test executed. The `--isolated` flag will cause testr to execute a separate runner per test:

```
$ testr run --isolated
```

In this mode testr first determines tests to run (either automatically listed, using the failing set, or a user supplied load-list), and then spawns one test runner per test it runs. To avoid cross-test-runner interactions concurrency is disabled in this mode. `--analyze-isolation` supercedes `--isolated` if they are both supplied.

1.11 Repositories

A testr repository is a very simple disk structure. It contains the following files (for a format 1 repository - the only current format):

- format: This file identifies the precise layout of the repository, in case future changes are needed.
- next-stream: This file contains the serial number to be used when adding another stream to the repository.
- failing: This file is a stream containing just the known failing tests. It is updated whenever a new stream is added to the repository, so that it only references known failing tests.
- #N - all the streams inserted in the repository are given a serial number.

- repo.conf: This file contains user configuration settings for the repository. `testr repo-config` will dump a repo configration and `test help repo-config` has online help for all the repository settings.

1.12 setuptools integration

testrepository provides a setuptools commands for ease of integration with setuptools-based workflows:

- testr: `python setup.py testr` will run testr in parallel mode Options that would normally be passed to testr run can be added to the testr-options argument. `python setup.py testr --testr-options="--failing"` will append –failing to the test run.
- testr –coverage: `python setup.py testr --coverage` will run testr in code coverage mode. This assumes the installation of the python coverage module.
- `python testr --coverage --omit=ModuleThatSucks.py` will append –omit=ModuleThatSucks.py to the coverage report command.

CHAPTER 2

Design / Architecture of Test Repository

2.1 Values

Code reuse. Focus on the project. Do one thing well.

2.2 Goals

Achieve a Clean UI, responsive UI, small-tools approach. Simulataneously have a small clean code base which is easily approachable.

2.3 Data model/storage

testrepository stores subunit streams as subunit streams in .testrespository with simple additional metadata. See the MANUAL for documentation on the repository layout. The key design elements are that streams are stored verbatim, and a testr managed stream called 'failing' is used to track the current failures.

2.4 Code layout

One conceptual thing per module, packages for anything where multiple types are expected (e.g. testrepository.commands, testrespository.ui).

Generic driver code should not trigger lots of imports: code dependencies should be loaded when needed. For example, argument validation uses argument types that each command can import, so the core code doesn't need to know about all types.

The tests for the code in testrepository.foo.bar is in testrepository.tests.foo.test_bar. Interface tests for testrepository.foo is in testrepository.tests.test_foo.

2.5 External integration

Test Repository command, ui, parsing etc objects should all be suitable for reuse from other programs.

2.6 Threads/concurrency

In general using any public interface is fine, but keeping syncronisation needs to a minimum for code readability.

CHAPTER 3

Development guidelines for Test Repository

3.1 Coding style

PEP-8 please. We don't enforce a particular style, but being reasonably consistent aids readability.

3.2 Copyrights and licensing

Code committed to Test Repository must be licensed under the BSD + Apache-2.0 licences that Test Repository offers its users. Copyright assignment is not required. Please see COPYING for details about how to make a license grant in a given source file. Lastly, all copyright holders need to add their name to the master list in COPYING the first time they make a change in a given calendar year.

3.3 Testing and QA

For Test repository please add tests where possible. There is no requirement for one test per change (because some-things are much harder to automatically test than the benfit from such tests). Fast tests are preferred to slow tests, and understandable tests to fast tests.

http://build.robertcollins.net/ has a job testing every commit made to trunk of Test Repository, and there is no automated test-before-merge process at the moment. The quid pro quo for committers is that they should check that the automated job found their change acceptable after merging it, and either roll it back or fix it immediately. A broken trunk is not acceptable!

See DESIGN.txt for information about code layout which will help you find where to add tests (and indeed where to change things).

3.3.1 Running the tests

Generally just `make` is all that is needed to run all the tests. However if dropping into pdb, it is currently more convenient to use `python -m testtools.run testrepository.tests.test_suite`.

3.3.2 Diagnosing issues

The cli UI will drop into pdb when an error is thrown if TESTR_PDB is set in the environment. This can be very useful for diagnosing problems.

3.3.3 Releasing

Update NEWS and testrepository/__init__.py version numbers. Release to pypi. Pivot the next milestone on LP to version, and make a new next milestone. Make a new tag and push that to github.

CHAPTER 4

Indices and tables

- genindex
- search