Hitch Documentation

Release

Colm O'Connor

May 31, 2016

Contents

Hitch is a framework for `/glossary/integration_testing`.

CHAPTER 1

Features

- Runs reliably without modification on Mac OS X, Ubuntu/Debian, Fedora, CentOS and Arch and in Docker.
- Automates its own deployment and does not interfere with your system other than to install packages.
- Provides boilerplate and tools to substantially minimize the problem of `/glossary/brittle_tests`.
- Readable `/glossary/hitch_test_description_language` that doesn't require you to write regular expressions.
- Built-in `/glossary/service_orchestration` library for running groups of services (databases, webservers, microservices) together.
- Built-in `/glossary/step_library` for common tasks (interacting with browsers, command line & emails).
- Provides a suitable environment for `/glossary/acceptance_test_driven_development` complete with debugging tools.

CHAPTER 2

Plugins

2.1 Hitch Plugin Documentation

`/glossary/hitch_plugin` documentation.

Contents:

2.1.1 HitchCron

Note: This documentation applies to the latest version of hitchcron.

HitchCron is a `/glossary/hitch_plugin` created to make testing applications that rely upon a cron or cron-like service.

It contains:

- A `/glossary/service` that runs during the whole test that runs a command periodically (typically every second).

Installation

To install:

```
$ hitch install hitchcron
```

Set up Cron

Note: See also: Generic Service API

To use, define the service after initializing the `/glossary/service_bundle` object but before starting it:

```
import hitchcron

self.services['Cron'] = hitchcron.CronService(
    run=['command', 'arg1', 'arg2', 'arg3'],     # Mandatory - list containing command + args
    every=1,                                    # Optional (Default: run every 1 seconds)
)
```

2.1.2 HitchMySQL

Note: This documentation applies to the latest version of hitchmysql.

HitchMySQL is a /glossary/hitch_plugin created to make testing applications that use MySQL easier.

It contains:

- A /glossary/hitch_package to download and install specified version(s) of MySQL.
- A /glossary/service to set up an isolated MySQL environment and run it.

Note: the MySQL service destroys and sets up a new database during each test run in order to provide strict /glossary/isolation for your tests.

Installation

First, install the the plugin in your tests directory:

```
$ hitch install hitchmysql
```

Set up MySQL

In your test, define the MySQL installation you want to test with:

```
import hitchmysql

mysql_package = hitchmysql.MySQLPackage(
    version="5.6.26"   # Optional (default is the latest version of MySQL)
)

# Downloads & installs MySQL to ~/.hitchpkg if not already installed by previous test
mysql_package.build()
```

To use, define the service after initializing the /glossary/service_bundle but before starting it:

Note: See also: Generic Service API

```
# Define a MySQL user for your service to set up
mysql_user = hitchmysql.MySQLUser("newpguser", "pguserpassword")

# Define a MySQL database for your service to set up
mysql_database = hitchmysql.MySQLDatabase(
    name="databasename",                    # Mandatory
    owner=mysql_user,                        # Mandatory
    dump="dumps/yourdump.sql"               # Optional (default: create empty database)
)

self.services['MySQL'] = hitchmysql.MySQLService(
    mysql_package=mysql_package,        # Mandatory
    port=13306,                         # Optional (default: 13306)
    users=[mysql_user, ],               # Optional (default: no users)
    databases=[mysql_database, ]        # Optional (default: no databases)
)
```

Interacting with MySQL

Once it is running, you can interact with the service and its databases:

```
In [1]: self.services['MySQL'].databases[0].mysql("SELECT * FROM yourtable;").run()
[ Prints output ]

In [2]: self.services['MySQL'].databases[0].mysql().run()
[ Launches into mysql shell ]
```

2.1.3 HitchNode

Note: This documentation applies to the latest version of hitchnode.

HitchNode is a `/glossary/hitch_plugin` created to make testing applications that use Node easier.

It contains:

- A `/glossary/hitch_package` to download and install specified version(s) of Node.
- A `/glossary/service` to set up an isolated Node environment and run it.

Note: the Node service destroys and sets up a new node environment during each test run in order to provide strict `/glossary/isolation` for your tests.

Installation

First, install the the plugin in your tests directory:

```
$ hitch install hitchnode
```

Set up Node

In your test, define the Node installation you want to test with:

```
import hitchnode

node_package = hitchnode.NodePackage(
    version="5.0.0"  # Optional (default is the latest version of Node)
)

# Downloads & installs Node to ~/.hitchpkg if not already installed by previous test
node_package.build()
```

To use, define the service after initializing the `/glossary/service_bundle` but before starting it:

Note: See also: Generic Service API

```
# Check if a package (e.g. less) is already installed:
import hitchtest
if not path.exists(path.join(
        hitchtest.utils.get_hitch_directory(),
        "node_modules", "less", "bin", "lessc"
    )):
    # Install the package is not found:
    chdir(hitchtest.utils.get_hitch_directory())
    check_call([node_package.npm, "install", "less"])
```

2.1.4 HitchPostgres

Note: This documentation applies to the latest version of hitchpostgres.

HitchPostgres is a `/glossary/hitch_plugin` created to make testing applications that use Postgresql easier.

It contains:

- A `/glossary/hitch_package` to download and install specified version(s) of postgresql.
- A `/glossary/service` to set up a test-specific postgresql environment and run postgresql.

Note: the postgresql service destroys and sets up a new database during each test run in order to provide `/glossary/isolation` for your tests.

Installation

First, install the the plugin in your tests directory:

```
$ hitch install hitchpostgres
```

Set up postgres

In your test, define the version of postgres that you want to test with:

```
import hitchpostgres

postgres_package = hitchpostgres.PostgresPackage(
    version="9.3.9"  # Optional (default is the latest version of postgres)
)

# Downloads & installs Postgres to ~/.hitchpkg if not already installed by previous test
postgres_package.build()
```

To use, define the service after initializing the `/glossary/service_bundle` but before starting it:

Note: See also: Generic Service API

```
# Define a postgresql user for your service to set up
postgres_user = hitchpostgres.PostgresUser("newpguser", "pguserpassword")

# Define a postgresql database for your service to set up
postgres_database = hitchpostgres.PostgresDatabase(
    name="databasename",                    # Mandatory
    owner=postgres_user,                    # Mandatory
    dump="dumps/yourdump.sql"               # Optional (default: create empty database)
)

self.services['Postgres'] = hitchpostgres.PostgresService(
    postgres_package=postgres_package,      # Mandatory
    port=15432,                             # Optional (default: 15432)
    users=[postgres_user, ],                # Optional (default: no users)
    databases=[postgres_database, ]         # Optional (default: no databases)
    encoding='UTF-8',                       # Optional (default: UTF-8)
    locale='en_US'                          # Optional (default: en_US)
    pgdata=None,                            # Optional location for pgdata dir (default: put in .hitch
)
```

Interacting with Postgres

Once it is running, you can interact with the service and its databases:

```
In [1]: self.services['Postgres'].databases[0].psql("-c", "SELECT * FROM yourtable;").run()
[ Prints output ]

In [2]: self.services['Postgres'].databases[0].psql().run()
[ Launches into postgres shell ]
```

2.1.5 HitchPython

Note: This documentation applies to the latest version of hitchpython.

HitchPython is a `/glossary/hitch_plugin` specifically to make testing python code easier.

It contains:

- a `/glossary/hitch_package` to set up installations of python and a corresponding virtualenv for your project.
- A `/glossary/service` to run Django.
- A `/glossary/service` to run Celery.

Installation

To install:

```
$ hitch install hitchpython
```

Set up your test's virtualenv

To create a virtualenv that your test can use, you need the following in your `/glossary/test_setup`:

```
import hitchpython

# Defines the python package
python_package = hitchpython.PythonPackage(
    version="2.7.10"                                # Mandatory
)

# Downloads & installs python to ~/.hitchpkg if not already installed
python_package.build()

# python_package.python = "/path/to/projects/virtualenv/python"
# python_package.pip = "/path/to/projects/virtualenv/pip"
```

Run Django Service

Note: See also: Generic Service API

hitchpython also contains a service class that can be used to run Django in your test.

To use, define the service after initializing the `/api/service_bundle` but before starting it:

```
# Service definition in setup:
self.services['Django'] = hitchpython.DjangoService(
    python=python_package.python,         # Python executable path. Mandatory
    port=18080,                           # Optional (default: 18080)
    managepy=None,                        # Optional full path to manage.py (default: None, assumes i
    settings="remindme.settings",         # Optional (default: settings)
    fixtures=['fixture1.json',],          # Optional (default: None)
    sites=False,                          # Optional (default: False - Put http://127.0.0.1:18080/ (w
    syncdb=False,                         # Optional (default: False - Run syncdb (for versions of dj
    migrations=True,                      # Optional (default: True - Run migrate (for Django 1.8 or 
    needs=[self.services['Postgres'], ]   # Optional (default: no prerequisites)
)
```

Once it is running, you can also interact with the service:

```
In [1]: self.services['Django'].manage("help").run()
[ Prints help ]

In [2]: self.services['Django'].url()
http://127.0.0.1:18080/

In [3]: self.services['Django'].savefixture("fixtures/database_current_state.json").run()
[ Saves the current state of the database as a fixture to file ]
```

Run Celery Service

hitchpython also contains a service class that can be used to run Celery during your test.

To use, define the service after initializing the `/glossary/service_bundle`:

```
# Service definition in setup:
self.services['Celery'] = hitchcelery.CeleryService(
    python=python_package.python,                      # Mandatory
    app="remindme",                                    # Mandatory
    beat=False,                                        # Optional (default: False)
    loglevel="INFO",                                   # Optional (default: INFO)
    concurrency=2,                                     # Optional (default: 2)
    broker=None,                                       # Optional (default: None)
    needs=[ self.services['Redis'], ]                  # Optional (default: no prerequisites)
)
```

Once it is running, you can also interact with the service in a hitch step or with ipython:

```
In [1]: self.services['Celery'].help().run()
[ Prints help ]

In [1]: self.services['Celery'].status().run()
[ Prints celery queue status ]

In [1]: self.services['Celery'].control(*args).run()
[ Run specific celery control commands ]

In [1]: self.services['Celery'].inspect(*args).run()
[ Run specific celery inspect commands ]
```

2.1.6 HitchRedis

Note: This documentation applies to the latest version of hitchredis.

HitchRedis is a `/glossary/hitch_plugin` specifically to make testing applications that use Redis easier.

It contains:

- A `/glossary/hitch_package` to download and install redis.
- A `/glossary/service` to run Redis.

Installation

If it is not already installed, install the hitchredis package:

```
$ hitch install hitchredis
```

Set up Redis

In your test, define the redis package you will use:

```
import hitchredis

redis_package = hitchredis.RedisPackage(
    version="2.8.4"                         # Optional (default is the latest version of redis)
)
```

```
# Downloads & installs redis to ~/.hitchpkg if not already installed
redis_package.build()
```

To use, define the service after initializing the /glossary/service_bundle:

Note: See also: Generic Service API

```
self.services['Redis'] = hitchredis.RedisService(
    redis_package=redis_package                         # Mandatory
    port=16379,                                         # Optional (default: 16379)
)
```

Interacting with Redis

Once it is running, you can interact with the service:

```
In [1]: self.services['Redis'].cli("-n", "1", "get", "mypasswd").run()
[ prints contents of mypasswd variable ]
```

2.1.7 HitchSelenium

Note: This documentation applies to the latest version of hitchselenium.

HitchSelenium is a /glossary/hitch_plugin specifically to make testing web applications easier.

It contains:

- A /glossary/service to run firefox and provide access to its webdriver.
- A /glossary/step_library to perform common actions with a selenium webdriver.

Installation

Install the hitch selenium plugin like so:

```
$ hitch install hitchselenium
```

Note: If you are running Mac OS X you must download and install firefox *manually* before running a test that uses hitchselenium.

Set up the Firefox service

Note: See also: Generic Service API

To use, define the service after initializing the /glossary/service_bundle:

```
import hitchselenium

# Service definition in engine's setUp:
self.services['Firefox'] = hitchselenium.SeleniumService(
    xvfb=False            # Optional (default: False). If true, this will run firefox hidden (only ava
    shunt_window=True     # Optional (default: True). This will move the window out of the way of the
    implicitly_wait=5.0   # Optional (default: 5.0). Set implicitly_wait value of the selenium driver
)
```

After the service bundle has been started, you can access the selenium webdriver like so:

```
self.driver = self.services['Firefox'].driver
```

Interacting with Firefox

You can then interact with firefox via the selenium webdriver in your steps or with `/glossary/ipython`:

```
In [2]: self.driver.get("http://localhost:8080/")
[ Opens http://localhost:8080 in firefox ]

In [3]: self.driver.find_element_by_id("id_description").send_keys("type something...")
[ Find element with ID description and types "type something" ]
```

The full selenium driver docs are available here: https://selenium-python.readthedocs.org/en/latest/navigating.html

Using the selenium step library

Using the selenium web driver can be cumbersome, so there is also a step library provided with steps that, when used correctly, should aid with `/glossary/test_readability`.

To set the selenium step library up in your test setup after the service bundle has been started:

```
self.webapp = hitchselenium.SeleniumStepLibrary(
    selenium_webdriver=self.services['Firefox'].driver,
    wait_for_timeout=5,
)

self.click = self.webapp.click
self.wait_to_appear = self.webapp.wait_to_appear
self.wait_to_contain = self.webapp.wait_to_contain
self.wait_for_any_to_contain = self.webapp.wait_for_any_to_contain
self.click_and_dont_wait_for_page_load = self.webapp.click_and_dont_wait_for_page_load
```

For instructions on how to use the step library in your steps see How to test web applications.

2.1.8 HitchSMTP

Note: This documentation applies to the latest version of hitchsmtp.

HitchSMTP is a `/glossary/hitch_plugin` created to make testing applications that send emails easier.

It contains:

- A `/glossary/service` to run a mock SMTP server that your application can be configured to send emails to.

Installation

To install:

```
$ hitch install hitchsmtp
```

Set up HitchSMTP

Note: See also: Generic Service API

To use, define the service after initializing the `/glossary/service_bundle`:

```
import hitchsmtp

self.services['HitchSMTP'] = hitchsmtp.HitchSMTPService(
    port=10025                                             # Optional (default: 10025)
)
```

Once it is running, you can access the emails which arrived via the logs:

```
In [1]: self.services['HitchSMTP'].logs.json()
[ list of dicts of email contents and properties ]
```

You can also wait for emails to arrive by waiting for the logs:

```
In [2]: email = self.services['HitchSMTP'].logs.out.tail.until_json(
    lambda email: "register: in email['payload'] or "register" in email['Subject'],
    timeout=5,
    lines_back=1,
)

In [3]: email
[ dict of email contents and properties ]
```

2.1.9 HitchVagrant

Note: This documentation applies to the latest version of hitchvagrant.

Installation

If it is not already installed, install the hitch vagrant package:

```
$ hitch install hitchvagrant
```

Setup

Note: See also: Generic Service API

To use, define the service after initializing the `/glossary/service_bundle`:

Like so:

```
import hitchvagrant

# Service definition in engine's setUp:
self.services['MyVM'] = hitchvagrant.VagrantService(
    directory="vagrantubuntu/",      # Directory containing Vagrantfile (optional)
)
```

Interaction

Once it is running, you can run ssh commands against the machine:

```
In [1]: self.services['MyVM'].ssh("pwd").run()
/vagrant
```

CHAPTER 3

Documentation

3.1 Getting started quickly with Hitch

This is a basic introduction to getting your first hitch test up and running.

3.1.1 Install prerequisites

You should have a reasonably up to date Ubuntu, Debian, Arch, Fedora or Mac.

On Ubuntu/Debian:

```
$ sudo apt-get install python3 python-pip python-virtualenv
$ sudo pip install --upgrade hitch
```

On Mac OS X:

```
$ brew install python python3
$ pip install --upgrade hitch virtualenv
```

On Arch:

```
$ sudo pacman -Sy python python-virtualenv
$ sudo pip install --upgrade hitch
```

On Fedora/RHEL/CentOS:

```
$ sudo yum install python3 python-virtualenv python-pip python3
$ sudo pip install --upgrade hitch
```

Note: The 'hitch' package (the bootstrapper) is a small python package with no dependencies.

3.1.2 Create your test directory

Create a directory inside the root of your project to put your tests in. For example:

```
~/yourproject$ mkdir tests
~/yourproject$ cd tests
~/yourproject/tests$
```

If you already have a tests directory you can call it something else.

3.1.3 Create the hitch environment

To initialize a hitch environment, run hitch init in your tests directory:

```
~/yourproject/tests$ hitch init
```

This will:

- Install any necessary system packages required to run hitch.
- Create a .hitch directory, create a python 3 virtualenv in it and install all the necessary packages to run hitch tests there.
- Ask you some basic questions about the project which you are testing.
- Create a skeleton hitch project template for you to use based upon the answers.

The skeleton template will include all of the following:

- `/glossary/hitchreqs.txt`
- `/glossary/engine.py`
- tdd.settings (`/glossary/hitch_settings`)
- ci.settings
- all.settings
- `/glossary/stub.test`
- README.rst

You might want to take a look around these files. They all try to be self-explanatory.

3.1.4 Running your first test

You can now run the stub test. Try running it in test driven development mode:

```
$ hitch test stub.test --settings tdd.settings
```

The first time you run this command it *may take a while* (up to 25 minutes depending upon what you answered).

Note: Why does the first test run take so long?

This might be a good time to take a break.

While you're at it, subscribe to the hitch subreddit and twitter feed for updates and news.

3.1.5 Back?

Note: If the stub test failed, please raise an issue.

Once the test run is done setting up, if there were no problems, you should see this:

```
Python 3.4.3 (default, Jul 28 2015, 18:20:59)
Type "copyright", "credits" or "license" for more information.

IPython 4.0.0 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.


SUCCESS

In [1]:
```

This is the interactive prompt that appears during the pause step. This is an `/glossary/ipython` prompt that can be used to interact with your app, inspect logs and try out test steps.

The components you selected during the set up should also be running. For example, if you chose postgres, the latest version of postgres will have been installed in the ~/.hitchpkg directory and it will be running and accessible.

To exit, simply hit ctrl-D.

This will shut everything down and then quit.

You're now ready to start writing new tests.

Happy testing!

Note: Was there anything that went wrong or was confusing? Please tell us! Help with Clarifying documentation.

3.1.6 Further reading

- How to test web applications
- How to test command line applications

3.1.7 Advanced topics

- How to do test driven development with hitch
- How to Parameterize your Test Cases with Hitch
- How to test applications which call external APIs
- How to use Hitch with Continuous Integration

3.1.8 Plugin Documentation

Note: Need tutorials for any other topics? Please raise a ticket.

3.2 How to

Contents:

3.2.1 How to do Acceptance Test Driven Development with Hitch

See also: `/glossary/behavior_driven_development`.

A recommended approach to `/glossary/acceptance_test_driven_development` wth hitch is to do the following steps in order:

1. Write a high level, failing `/glossary/acceptance_test` for an unimplemented feature or bug.
2. Refactor the high level test.
3. Implement the code that runs the test in the `/glossary/execution_engine`.
4. Implement the code that makes the test pass.
5. Refactor the code.

Each of these steps should be iterative and going back and changing the step or steps above it will often be necessary.

Mistakes made at the higher level often become progressively more expensive the later

Write a high level failing acceptance test

Note:

See How to do Behavior Driven Development with Hitch for how to get to this point.

- name : Search for and buy a Fisher-Price Power Wheels Disney Frozen Jeep from the store preconditions:

  database: basic-database-with-power-wheels browser: firefox

  description: | User searches the website with firefox, finds the toy, puts it in the shopping cart, registers as a first time user and purchases.

  scenario:

  – Load the website
  – Search for: Power Wheels Disney Frozen Jeep
  – Click on Fisher-Price Power Wheels Disney Frozen Jeep
  – Click: purchase
  – Click: check-out-shopping-card
  – **Fill form:** Name: Barack Obama Password1: nowletmebeclear Password2: nowletmebeclear Delivery Address: 1600 Pennslyvania Ave.
  – Click: register
  – **Fill form:** Credit card name: Barack Obama Credit Card Number: 1135 4913 9201 1102 9999 Three digit security code: 012 Expiry date: 01/18 Card Address: 1600 Pennslyvania Ave.
  – Click: purchase
  – Verify that display page was displayed

  - **Verify that purchase order was recorded:** Name: Barack Obama Item: Power Wheels Disney Frozen Jeep

3.2.2 How to do Behavior Driven Development with Hitch

Note: Before starting BDD, you may want to familiarize yourself with the Cynefin framework as an approach to actually thinking about your requirements and using BDD as a sensemaking technique

Building a project starts out with requirements analysis.

Behavior driven development is an approach taken from test driven development and domain driven design applied to software development.

The basic approach consists of two very basic tasks:

- Coming up with scenarios and writing them down in a DSL.
- Thinking about the scenarios, having conversations about them with stakeholders and reworking them.
- Ending up with a DSL that can be used to drive a realistic test which can be used to develop against.

A recommended approach to apply a `/glossary/behavior_driven_development` approach with Hitch would be to build a project by taking the following steps in order and iterating upon each one before moving to the next step:

1. Write test case name
2. Write test case description
3. Write test case scenario
4. Parameterize test case scenarios if necessary
5. Implement any necessary mock services, test steps, etc. in the test execution engine.
6. Implement the code that fulfils the test.

Steps 1-6 constitute behavior driven development and steps 4-6 constitute `/glossary/acceptance_test_driven_development`.

Steps 1-3 are ultimately more about *conversations* than they are about writing code, and the best scenarios are written with input from as many stakeholders as possible.

At *each* point in the process, new requirements, ideas, problems and information will emerge. If it takes a long time to write a single scenario because of the communications overhead, *that's ok*. It's still *a lot* cheaper than discovering a mistake in your specifications weeks or even months later while implementation is already in progress.

Scenarios should be written by a `/glossary/customer_representative`.

This approach can be used to:

- Create test scenarios for new features to develop against.
- Write test scenarios for failing test cases.

Describe the behavior

Initially with just the name and no scenario:

```
- name : Buy a toy car from the store
```

Iterate upon it:

```
- name : Search for and buy a toy car the store
```

Iterate upon it some more:

```
- name : Search for and buy a Fisher-Price Power Wheels Disney Frozen Jeep from the store
```

Then flesh it out with a more detailed description:

```
- name : Search for and buy a Fisher-Price Power Wheels Disney Frozen Jeep from the store
  description: |
    User browses the website, finds the toy and purchases.
```

This might give you an idea for another scenario. That's good. You may end up doing that a lot. Note it down and put it aside:

```
- name : Browse via categories and buy a toy from the store
```

Return to iterating upon the original scenario:

```
- name : Search for and buy a Fisher-Price Power Wheels Disney Frozen Jeep from the store
  description: |
    User searches the website, finds the toy, puts it in the shopping cart,
    registers as a first time user and purchases with a credit card.
```

Flesh out the scenario

Once the description is nailed down sufficiently, you can attack the scenario.

Your test cases should generally follow the structure of Given-When-Then, where the preconditions and first few steps specify a "given" situation to start the scenario in, the "when" specifies a series of steps and "Then" specifies the outcome.

```
- name : Search for and buy a Fisher-Price Power Wheels Disney Frozen Jeep from the store
  preconditions:

  description: |
    User searches the website, finds the toy, puts it in the shopping cart,
    registers as a first time user and purchases.
  scenario:
    - Load the website
    - Search for: Power Wheels Disney Frozen Jeep
    - Click on Fisher-Price Power Wheels Disney Frozen Jeep
    - Click purchase
    - Check out shopping cart
    - Sign up
    - Enter credit card details
    - Click purchase
```

And iterate upon it, of course, until you have a well defined test case that covers input from all stakeholders that you can turn into an executable test:

```
- name : Search for and buy a Fisher-Price Power Wheels Disney Frozen Jeep from the store
  preconditions:
    database: basic-database-with-power-wheels
    browser: firefox
  description: |
    User searches the website with firefox, finds the toy, puts it in the shopping cart,
    registers as a first time user and purchases.
  scenario:
    - Load the website
    - Search for: Power Wheels Disney Frozen Jeep
    - Click on Fisher-Price Power Wheels Disney Frozen Jeep
    - Click: purchase
    - Click: check-out-shopping-card
    - Fill form:
        Name: Barack Obama
        Password1: nowletmebeclear
        Password2: nowletmebeclear
        Delivery Address: 1600 Pennslyvania Ave.
    - Click: register
    - Fill form:
        Credit card name: Barack Obama
        Credit Card Number: 1135 4913 9201 1102 9999
        Three digit security code: 012
        Expiry date: 01/18
        Card Address: 1600 Pennslyvania Ave.
    - Click: purchase
    - Verify that display page was displayed
    - Verify that purchase order was recorded:
        Name: Barack Obama
        Item: Power Wheels Disney Frozen Jeep
```

Once you have a test or a group of tests you are happy with, you can move on to the next group of steps - How to do Acceptance Test Driven Development with Hitch.

3.2.3 How to test command line applications

Note: This tutorial assumes that you have the `/glossary/hitch_plugin` `/plugins/hitchcli` installed and its step library is set up.

If you followed the quickstart tutorial and said yes to testing a command line app, this should already be done for you.

Warning: This tutorial is a work in progress. It is usable, but more is coming soon.

Running a command

To run a command you use the step “run”:

```
- run: mycommand --myoptions
```

Waiting for some text to appear

To wait for some text to appear, use the step ‘expect’, e.g.:

```
- expect: Continue?
```

Typing

To ‘mock type’ a line of text, use the step ‘send keys’:

```
- send keys: Y
```

Ctrl-[key]

Unless you want to use ctrl-[key] in which case use:

```
- send control: C
```

For control-C, or:

```
- send control: D
```

Waiting for a command to finish

To wait for a command to finish successfully, use the step “finish”:

```
- finish
```

If you want to verify that it finished with a specific non-zero exit code (e.g. 1), add the property ‘with code’:

```
- finish:
    with code: 1
```

If you want the command to finish and you don’t care which code it exits with, use ‘any’ in place of 1:

```
- finish:
    with code: any
```

Sending an exit signal to a command

If you want to SIGTERM, SIGINT, SIGKILL or any other valid unix signal, use the following step:

```
- send signal: SIGTERM
```

Change directory

If you want to change directory before running a command, use:

```
- cd: ../newdirectory
```

3.2.4 How to use Hitch with Continuous Integration

Hitch is designed to be simple to configure and simple to deploy on both test driven development environments and continuous integration servers.

It is also largely self contained meaning that the amount of configuration you need to do on your CI systems to get them to work with hitch should be very limited. For instance:

- Hitch takes care of installing system packages required to run your tests (apt-get install / yum install).

- Hitch will run your tests with XVFB if specified, meaning your CI server doesn't have to be configured to take care of that.
- Hitch will download and deploy any code it needs into self contained directories.

You can create settings files for the different environments where you run your tests. It's recommended that you have at least one 'tdd.settings' for test driven development on your laptop and a 'ci.settings' for doing continuous integration regression tests on your CI server.

See here for examples:

- https://github.com/hitchtest/django-remindme-tests/blob/master/ci.settings
- https://github.com/hitchtest/django-remindme-tests/blob/master/tdd.settings

The following two examples assume that you already have a ci.settings file with settings appropriate for your CI environment and that your tests are in a directory in your repo called tests.

Jenkins

On your jenkins machine you should first log in as root and do steps #1 and #2 from What does the init script do?

You should then ensure that the jenkins user can sudo without a password.

Once that's done, create a job for your project using the web interface.

The job should then be configured to run the following three commands:

```
cd tests
hitch init
hitch test . --settings ci.settings
```

That should be all that's required.

Travis

Note: Although it says python 3.4 below, your code will be tested with the version of python you specified in your tests (assuming you are testing python code at all). python 3.4 will *only* be used to run the bootstrap script.

Your .travis.yml file should look something like this:

```
before_install:
  - sudo apt-get update -qq
  - sudo apt-get install -qq python python-dev python-setuptools python-virtualenv python3 python3-de
language: python
python:
  - "3.4"
install:
  - "pip install hitch"
  - "cd tests"
  - "hitch init"
script:
  - "hitch test . --settings ci.settings"
```

Note: Test runs using Travis may be very slow (> ~30 minutes) due to the virtual machine being used to run your tests being destroyed and rebuilt from scratch each time.

3.2.5 How to debug a test failure

[TO DO]

Pausing a test on failure

[TO DO]

Moving up the call stack with IPython

[TO DO]

3.2.6 How to test applications which call external APIs

Note: The pattern described here is the same that is currently used by HitchSMTP.

While it is always preferable to test your application with real APIs wherever feasible, it is often not feasible to test or develop against real APIs.

For example:

- APIs which cost money to use like SMS gateways or have rate limiting like Twitter.
- APIs where it may be difficult to non-manually determine if it was called correctly.
- APIs which require special hardware or complex software to run.
- APIs where failure scenarios may be difficult or even impossible to reproduce.

If your app needs to talk to an API like this, you can mimic the effect of calling APIs like this in your test scenarios by starting a `/glossary/mock_service` and using hitch to listen to it.

This can help you maintain a high level of `/glossary/test_realism` and loose `/glossary/coupling` in your tests while avoiding the expense and inconvenience of calling the real API.

Creating your mock service

Your mock service can be written in any language but it must do two things.

Firstly it should print a line when it has fully started and is ready to receive API calls. E.g.:

```
READY
```

Secondly, it should print individual lines of valid JSON when it receives an API call. These JSON line should contain all of the data you want to check in your test. If your mock service is doing something like this it's good to go:

```
READY
{"sent_from": "webmaster@localhost", "header_to": "django@reinhardt.com", "sent_to": ["django@reinhar
{"sent_from": "noreply@localhost", "header_to": "<django@reinhardt.com>", "sent_to": ["django@reinhar
```

Running your mock service with Hitch

Once your mock service is printing valid lines of JSON when it receives API calls, you will want to run it as part of your tests. To do this, you must add a service definition in the `/glossary/execution_engine` like so:

```
# Should go at the top of the engine file
import hitchserve

# Should go between "self.services = ServiceBundle(...)" and "self.services.startup(...)":
self.services['MyMockAPI'] = hitchserve.Service(
    command=["command", "arg1", "arg2", "-x", "option1value", "--option2=option2value" ],
    log_line_ready_checker=lambda line: line == "READY",
    no_libfaketime=True,
)
```

Note: You should ensure that your mock service runs *unbuffered*. If it is a python application this means starting it with the -u switch.

For additional documentation on running mock services (e.g. to run the command in a specific directory), see Generic Service API.

Manually checking the output of your mock service during a test run

Once your mock service is up and running during your test and your application has called it, you can check its logs manually to verify that it received the API call and logged a message:

```
In [1]: self.services['MyMockAPI'].logs
Out[1]:
[          MyMockAPI] READY
[          MyMockAPI] {"subject": "[127.0.0.1:18080] Confirm E-mail Address", "links": ["http://127.0
[          MyMockAPI] {"subject": "Reminder", "links": [], "header_to_name": "", "multipart": false,
```

To get a python list of dicts representation of only the valid JSON lines, you can just add .json() to the end:

```
In [2]: self.services['MyMockAPI'].logs.json()
Out[2]:
[{'contenttype': 'text/plain',
  'date': 'Fri, 23 Oct 2015 01:15:11 -0000',
  'header_from': 'webmaster@localhost',
  'header_from_email': None,
  'header_from_name': None,
  'header_to': 'django@reinhardt.com',
  'header_to_email': None,
  'header_to_name': None,
  'links': ['http://127.0.0.1:18080/accounts/confirm-email/dwu83kr92t96ek7hkfwjsu4nj6et7i4fu6ntjsn6xu
  'multipart': False,
  'payload': 'User django at 127.0.0.1:18080 has given this as an email address.\n\nTo confirm this i
  'sent_from': 'webmaster@localhost',
  'sent_to': ['django@reinhardt.com'],
  'subject': '[127.0.0.1:18080] Confirm E-mail Address'}]
```

Note: If .logs shows your JSON but .logs.json() is empty, check that your mock service is printing *valid* lines of JSON.

You can treat this list of dicts like you would any others in python:

```
In [2]: self.services['MyMockAPI'].logs.json()[-1]
Out[2]: [ Returns a dict representation of the last JSON line printed to the console by MyMockAPI ]

In [3]: self.services['MyMockAPI'].logs.json()[-1]['contenttype']
Out[3]: 'text/plain'

In [4]: self.services['MyMockAPI'].logs.json()[-1]['links'][0]
Out[4]: 'http://127.0.0.1:18080/accounts/confirm-email/dwu83kr92t96ek7hkfwjsu4nj6et7i4fu6ntjsn6xues1r
```

Automatically checking the output of your mock service during a test run

Once you've verified that your mock service is running well with your test and printing the correct output, you will want to write a step in your test which waits for the API to be called after another step (e.g. a click) which triggers it to be called:

```
- Click: send-sms
- Wait for SMS:
    Containing: Thank you
```

To do this you must create a step in your engine which listens to the logs:

```
def wait_for_sms(self, containing=None):
    """Wait for an SMS arrive containing the text 'containing'."""
    self.services['MyMockAPI'].logs.out.tail.until_json(
        lambda apicall: containing in apicall['smstext'],
        timeout=45,
        lines_back=1,
    )
```

When this step is reached, it will check the logs to see if the API call has already been made and, if not, it will wait for up to 45 seconds.

As soon as the API call is made, it will continue on to the next step.

If an API call containing the text "Thank you" in 'smstext' is never made against the mock service, it will throw an exception, causing the test to fail.

See also:

- Hitch Engine API
- Generic Service API
- HitchSMTP
- Python list documentation
- Python dict documentation
- Python lambdas

Note: Was there anything that went wrong or was confusing while following this tutorial? Please tell us! Help with Clarifying documentation.

3.2.7 How to Parameterize your Test Cases with Hitch

Test case parameterization is a way of taking an existing test case and tweaking it to run multiple times, each in a slightly different way with just a couple of lines.

For example:

- Taking a test case that runs a scenario on python 2.7.10 and making it also run on python 3.4.3 and 3.5.0.
- Taking a test case that runs on firefox and making it run on 4 other browsers.
- Creating a group of test cases for a particular feature that differ only slightly.

I'd recommend parameterization only *after* noticing that your test cases have gotten somewhat repetitive. In general it is something that you should *avoid* doing pre-emptively.

Strictly speaking it is not *necessary* to parameterize your test cases since you can just copy and paste and tweak the things that you need. However, it's a good idea to make your test cases `/glossary/DRY` just like any other code to make them easier to maintain.

Example

In your all.settings, put the following variable:

```
python_versions:
  - 2.7.10
  - 3.4.3
  - 3.5.0
```

This is a list of 3 python versions defined in YAML in the `/glossary/hitch_settings`.

This list is made available to jinja2, which you can see below (jinja2 is basically anything surrounded by { or }).

Now put a test in the same directory, with the extension .test and make it look something like this:

```
{% for python_version in python_versions %}
- name: Load website and click register (python {{ python_version }})
  preconditions:
    python_version: "{{ python_version }}"
  tags:
    - registration
    - py{{ python_version }}
  scenario:
    - Load website
    - Click: register
{% endfor %}
```

This code with a for loop - will *generate* three *almost identical* test cases with the following differences:

- The name (each one will have the *same* name except with a different python version number at the end)
- One of the tags - the tags will be py3.5.0, py3.4.3 and py2.7.10 respectively. All three will have the tag "registration" though.
- The precondition "python_version" - each one of them will have the preconditions set differently.

This is what it does:

- The {% for python_version in python_versions %} and {% endfor %} surround a *block* of test description that it repeats.
- It loops 3 times because python_versions, set in all.settings, has 3 versions.

- Anything surrounded by {{ and }} in the block is set to 2.7.10, 3.4.3 and 3.5.0.

There are plenty more tricks you can play with parameterized test cases. I recommend gaining familiarity with Jinja2 if you want to do more.

Careful, though - parameterization can help with reducing the repetitiveness of your test cases but it can also make them less readable.

3.2.8 How to refactor your tests

A good, although uncommon development practice when building features or fixing bugs is refactoring.

Refactoring means making small, incremental changes that improve the quality of code.

This usually means:

- De-duplicating code
- Decoupling code
- Minimizing the amount of imperative code (e.g. python code) in favor of declarative code (e.g. YAML configuration).
- Improving the code's readability (changing names, adding comments, disambiguating identifiers).

Tests are code too, so it's good practice to refactor your tests to gradually improve `/glossary/test_quality` as you write new tests or fix existing ones.

Of course, you should only refactor *passing* tests and you should always run passing tests after refactoring to ensure that they are still passing.

Here is a non-exhaustive list of things which you could work on while writing new tests or changing old ones that you can do to improve your test quality:

De-duplicate duplicated tests

You may find after a while that your test suite has a lot of duplication - for example, tests that do almost the same thing in two or three slightly different ways.

See `/glossary/parameterize_test_cases` for how to remove some of that duplication.

Move configuration from engine.py to all.settings

Your execution engine should be kept as short as possible yet still capable. If you have any long lists or chunks of data in your engine.py, moving them into all.settings will help to keep it clean.

Change HTML IDs and Classes to make them more readable

Beware! This might be best left to a developer since it may require code changes as well if the ID is used in many places (code, javascript, CSS, etc.) as well as changing other IDs to accomodate.

If you have test steps that look like this:

```
- Click: btnreg
```

Because the registration button had the HTML ID 'btnreg' when you wrote the test, it might be worth changing the ID in the test and in the HTML code to make it more readable, e.g. to something like:

```
- Click: register-button
```

Similarly, if there is a button with ID:

```
- Click: register
```

It *may* be worth renaming it to:

```
- Click: register-button
```

If there is *any possibility of confusion* between register “something else” (e.g. a register link).

Increase the realism of your tests

Run your tests in a more cost effective way

If you have an extremely realistic test suite, it may end up being very expensive and very slow to run it from beginning to end. This is not necessarily a bad thing, although when it does start happening you need to start prioritizing some test cases over others and ensure that they are run

Bring continuous integration environment closer to production

3.2.9 How to develop effectively with slow tests

While `/glossary/unit_testing` is usually extremely fast, unit tests suffer from tight `/glossary/coupling` and a lack of `/glossary/test_realism`.

On the other hand, realistic and loosely coupled functional tests can inevitably end up being very slow to run. Unless properly handled this can end up lengthening you code-test-refactor cycle and reducing your productivity.

Long test runs should not scare you as much as unrealistic or tightly coupled tests, however. Even test runs that take multiple days to complete can end up being more useful than tests that take seconds and they don’t even have to slow you down.

It is easier to mitigate the effect of slower tests than it is to mitigate the effect of a lack of realism or tight coupling in your tests.

`/glossary/you_can_optimize_slow_you_cant_optimize_unrealistic`.

There are two goals to `/glossary/regression_testing`:

- Providing confidence in the latest code required to release.
- Providing feedback about newly introduced bugs as quickly as possible.

If you split these goals and approach them separately, you can develop more effectively.

#1 Use acceptance test driven development to develop features

Goal : provide feedback as quickly as possible about newly introduced bugs

If you can reduce your acceptance tests to < 1 minute, you can run one nearly constantly while you develop - targeting the feature which you are working on.

This will end up catching a large majority of bugs in your code *while you develop* in a matter of seconds.

More important than how long a test takes when it passed is how quickly it is *designed to fail when there is a bug*. If you can get failure feedback in ~5 seconds on a 60 second long test, that is every bit as good as instantaneous feedback.

You do not have to wait for the test to complete to continue developing either.

60 second long acceptance tests can run in the background while you work. The longer the test runs the more confidence you will have that you haven't broken anything.

[graph of confidence level]

#2 Introduce multiple layers of regression testing

Goal : provide feedback as quickly as possible about newly introduced bugs

While developing against one test focused on the feature you are working on will catch most bugs, you can still end up breaking code covered by other tests. This is a particular problem on code bases with very tightly coupled code (i.e. big balls of mud).

However, you can take a focused approach to catching regression bugs without waiting for 48 hour test runs to complete by running targeted regression testing that is designed to fail and provide feedback as fast as possible.

- Running groups of tests associated with what you were working on before pushing (e.g. hitch test . –tag feature-email-user).
- Running random tests constantly on the development branch [feature coming soon].

#3 Run entire suites of regression tests during evenings, nights and weekends

Goal : providing confidence in the latest code.

During periods when your code base is not being worked upon you can run

#3 Take advantage of time not spent developing: lunch, evenings, nights and weekends

If you can run a full set of realistic regression tests in under 12 hours

#1 Throw hardware at the problem

This is often a good approach to performance problems since hardware - relative to developers, at least - is cheap. If you have a spare computer, you can adopt it to run automated tests full time - while you are working and while you are sleeping.

80% of bugs are caught by 20% of the tests.

While this

However, it is easier to mitigate the effect of slower tests than it is to mitigate the effect of unrealistic and tightly coupled tests.

Below are a set of practices which can

This

Pareto rule Realistic tests Develop against a single test Rope in some spare computers Randomized testing Overnight testing

3.2.10 How to do test driven development with hitch

Tutorial coming soon

3.2.11 How to upgrade Hitch

By default, hitch uses the same versions of software and plugins that it used when you first set it up.

However, new bug fixes and features are being made available all the - bug fixes and features that you may want to take advantage of.

When to upgrade

The best times to upgrade are:

- Before installing and using a new plugin with 'hitch install'.
- Every two weeks (say, at the beginning of a sprint).
- If you run into a bug in hitch and you have not upgraded in a while.
- When most, if not all of the tests on your project are passing (so that you can more easily distinguish between bugs in your project and bugs in hitch).

Upgrade the bootstrapper

Although the bootstrapper does very little and should change very little, there are occasionally bugfixes included. Using your system python, On Linux, run the following command:

```
$ sudo pip install --upgrade hitch
```

On Mac:

```
$ pip install --upgrade hitch
```

Upgrade your project's test environment

The vast majority of your code is installed in the `/glossary/hitch_directory` directory in your project. The plugin versions you are currently using are all specified in `/glossary/hitchreqs.txt`.

To upgrade all plugins to their latest versions, run:

```
$ hitch upgrade
```

hitchreqs.txt should now have a new list of package versions.

Once you are done, re-run all of the tests again:

```
$ hitch test .
```

If everything passes, you can commit the new hitchreqs.txt to source control.

What if something goes wrong during upgrade?

While every effort is taken to ensure that the latest versions of hitch and its plugins are more stable than the last, there is always the possibility that bugs can creep in during upgrade.

If something goes wrong during upgrade, you can do the following:

1. If you see an error message related to deprecation, change your code to accomodate and re-run the tests.

2. If you see an inexplicable error, you can try running the following commands, which should rebuild your local hitch environment from scratch:

```
$ hitch clean
$ hitch init
$ hitch test .
```

3. You can try running the following commands, which should destroy your ~/.hitchpkg directory as well, allowing for it to be rebuilt:

```
$ hitch cleanpkg
$ hitch clean
$ hitch init
$ hitch test .
```

4. If none of the above fix your problem, you've discovered a bug in the framework itself. Please raise an issue.
5. You can *roll back* your changes reverting your hitchreqs.txt to the state it was before running hitch upgrade (e.g. with git checkout) and running:

```
$ hitch init
```

I've upgraded. What about the rest of my team?

Once one person has upgraded and committed a new hitchreqs.txt to source control, the rest of the team can follow the same steps, with one minor difference - instead of running "hitch upgrade", they should run:

```
$ hitch init
```

To bring their environment into alignment with the latest hitchreqs.txt versions.

If you had to run "hitch clean" or "hitch cleanpkg", they probably will as well.

Note that your continuous integration environment should *always* run this command before running any tests.

3.2.12 How to test web applications

Note: This tutorial assumes that you have the `/glossary/hitch_plugin` HitchSelenium installed and its step library is set up.

If you followed the quickstart tutorial and said yes to testing a webapp, this should already be done for you.

Warning: This tutorial is a work in progress. It is usable, but more is coming soon.

Writing a step that clicks on a button or link

To click on an individual item, you need to use the step "click" like so:

```
- Click: register
```

This is telling hitch to click on an HTML element with the HTML ID "register".

Note: This part is sometimes controversial. If you disagree, read Why should web application tests only interact with HTML IDs and HTML classes? for the rationale.

Now, there's a good chance that:

- Your HTML element does not have that ID - in which case you should *change the HTML itself* so that it does have that ID.
- That button has a different HTML ID - in which case you should use that ID instead (bookmark How to refactor your tests for later).

Writing a step that clicks on an item that is part of a group

Sometimes the thing that you want to click on is part of a group, or a group of groups.

For instance, you may want to click on the first link in a list of links. To do that you use the same step:

```
- Click: first friend-link
```

Here, "friend-link" is an *HTML class*.

As before, if the list of elements do not have a readable HTML class signifying what they are, you should *add* a class in the HTML itself.

Elements can have multiple classes, so if an element already has a class but it does not clearly identify all of the items in the list, you should add a class that does.

If you want to click on the 2nd item:

```
- Click: 2nd friend-link
```

Or the last:

```
- Click: Last friend link
```

To click on an item that is part of a group which is *also* itself part of a group, you can specify two classes:

```
- Click: First calendar day-31
```

Try to keep the test steps readable by using appropriately named classes where possible.

Verifying an element exists on the page - e.g. an error message

The same pattern can be used to wait for elements to be visible on the page. e.g.:

```
- Wait to appear: first friend-link
- Wait to appear: 2nd friend-link
- Wait to appear: Last friend-link
- Wait to appear: First calendar day-31
```

This is the recommended approach for items which signify certain things that you want to happen.

If, for example, you are testing for the presence of an error message indicating that a user must enter a ZIP code, the following is a good way of doing it:

```
- Wait to appear: error-message-zip-code
```

Waiting for text to appear

Note that waiting for specific text to appear is *not* a good approach for detecting error messages, or, indeed, any other kind of text which is decided upon by the application. Why? Translations.

If an application is translated and you test the same scenario by checking for IDs, the test will continue to work. If you just check for the presence of text, it will break.

Nonetheless, waiting for text to appear is often a good way to determine if text entered by the user in a test shows up in the right place.

Waiting for text to appear also follows the same pattern as above:

```
- Wait to contain:
    item: first username
    text: django
- Wait to appear:
    item: second username
    text: django
- Wait to appear:
    item: last username
    text: django
- Wait to appear:
    item: first user username
    text: django
```

3.3 Frequently Asked Questions

Contents:

3.3.1 How do I uninstall hitch completely?

If you want to remove all trace of hitch after installing it on your machine, you just need to do the following:

```
$ hitch cleanpkg
```

Followed by (this may requires sudo):

```
$ pip uninstall hitch
```

Additionally, you may want to:

- Delete the ~/.unixpackage directory.
- Delete the .hitch directory from your tests folder.

Note that the above will not uninstall system packages installed during the course of running tests.

3.3.2 How does Hitch compare to other technologies?

Cucumber/Behave/RSpec/Behat/Behave

Cucumber, RSpec, Behat and Behave and are all keyword driven test automation frameworks that run automated acceptance tests. They contain an interpreter for executing high level test cases written in Gherkin.

Hitch follows a similar approach but has its own equivalent to Gherkin: `/glossary/hitch_test_description_language`.

Unlike Gherkin it does not use its own syntax - its syntax is built upon YAML.

Test cases written with Hitch test should usually be less verbose and more to the point, although still ideally maintaining readability.

Gherkin example from the Cucumber website (223 characters; English-like):

```
Feature: Division
In order to avoid silly mistakes
Cashiers must be able to calculate a fraction

Scenario: Regular numbers
    * I have entered 3 into the calculator
    * I press divide
    * I have entered 2 into the calculator
    * I press equal
    * The result should be 1.5 on the screen
```

Hitch equivalent (113 characters; not English-like):

```
- name: Division
  description: Cashier calculates a fraction
  scenario:
    - Enter: 3
    - Press: divide
    - Enter: 2
    - Press: equal
    - Result: 1.5
```

Step-to-code regular expression translation is also unnecessary in Hitch sidestepping potential traps like this.

Note: This pitfall is recognized by Cucumber in issue #1.

The python tool behave gives you three different parser options as a way to deal with it. There are other suggested workarounds too.

The above three steps are implemented as follows in Hitch:

```
def enter(self, number):
    # code that enters a number

def press(self, key):
    # code that presses a key

def result(self, number):
    assert displayed_result == number
```

More complex data can also be cleanly encoded into steps and preconditions. Anything that is valid YAML is allowed.

You can write a complex step like this:

```
- Send mail:
    From address: Receiver <to@todomain.com>
    To address: Sender <from@fromdomain.com>
    Body:
      From: Receiver <to@todomain.com>
      To: Sender <from@fromdomain.com>
      Subject: Test email for "HitchSMTP"
      Content: |
        http://www.google.com
        Another link: http://yahoo.com
        Another link: https://www.google.com.sg/?gfe_rd=cr&ei=2X4mVebUFYTDuATVtoHoAQ#q=long+long+long
```

Which would trigger a python method call equivalent to the following:

```
self.send_mail(
    from_address="Receiver <to@todomain.com>",
    to_address="To address: Sender <from@fromdomain.com>",
    body={
        "From" : "Receiver <to@todomain.com>",
        "To" : "Sender <from@fromdomain.com>",
        "Subject" : "Test email for \"HitchSMTP\""
        "Content" : (
                "http://www.google.com\n"
                "Another link: http://yahoo.com\n"
                "Another link: https://www.google.com.sg/?gfe_rd=cr&ei=2X4mVebUFYTDuATVtoHoAQ#q=long+
            )
        }
    )
```

Where reading the data in the step code `/glossary/execution_engine` is still straightforward:

```
self.send_mail(self, from_address, to_address, body)
    content = body.get("content")
```

The above applies to the following packages:

- hitchtest

Hitch also provides plugins to perform many more test and development related tasks, saving on boilerplate (see Hitch Plugin Documentation).

Hitch does *not* provide:

- Bindings to write the execution engine in languages other than python. This is not roadmapped and not possible currently.
- Plugins to easily test other languages and frameworks (e.g. Java, node, Ruby, etc.). This possible but not easy currently and is roadmapped.

Docker/Docker Compose

Docker is a lightweight virtualization technology that provides system `/glossary/isolation` using cgroups and kernel namespaces.

Docker can be used to develop software in, test software in and deploy software in. By running the same container in all three environments, development and testing can achieve a greater degree of `/glossary/test_realism` thus avoiding many ‘surprise’ production bugs.

Nonetheless, the isolation and realism is not as high as “true virtualization” (VirtualBox, Xen, VMWare) provided via kernel emulation.

The same Docker container running on different systems can (and probably will, for many projects eventually), exhibit different behavior due to different versions of the linux kernel or libc in development, testing and production environments (TODO : verify libc differences??).

Due to the reliance on Linux kernel features for isolation, docker also does not work on Mac OS X or BSD platforms without running it in a heavyweight virtual machine.

Hitch can run docker containers, as it can any other process (a plugin to make this easier is coming soon).

If you deploy docker containers in your production environment, this is a recommended approach since it will bring a greater level of `/glossary/test_realism`.

If you do *not* deploy docker containers in your production environment, you may want to avoid using docker for development and test environments.

Hitch achieves a similar, although lower level of isolation and realism using a different approach:

- `/glossary/package_isolation`
- `/glossary/data_isolation`
- `/glossary/process_isolation`
- `/glossary/environment_isolation`

You can, for instance, run the exact same database version, python version and redis version that you do in production on your development machine.

[TO DO : docker-compose and starting services bug]

The above applies to the following packages:

- hitchserve
- hitchtest
- All hitch plugins

Note: You can also run hitch *in* docker. It is regularly tested with the latest version.

Built-in Django Testing Framework

Django already comes with four official classes for unit testing web apps, each of which test at a progressively higher level:

- SimpleTestCase - a low level unit tester for Django views.
- TransactionTestCase - a low level unit tester for Django views which also rolls back the database.
- TestCase - a low level unit tester which performs the above and also loads fixtures and adds django specific assertions.
- LiveServerTestCase - a higher level TransactionTestCase which runs the django web server to allow for the use of selenium.

See : https://docs.djangoproject.com/en/1.8/topics/testing/tools/ for details.

Hitch serves as an effective drop in replacement for all of these. While slower, tests written using hitch should exhibit a greater degree of `/glossary/test_realism`, `/glossary/isolation` and looser `/glossary/coupling`.

Practical benefits:

- You can run a celery service alongside the test.
- Hitch test maintains stricter database isolation.
- It runs all services with faketime, allowing you to mock the forward passage of time via your tests.

- Looser coupling means that if you refactor or rewrite your application code, you should only need minimal changes to your tests.
- Hitch tests can more easily be made to be `/glossary/business_readable`.

Tox, PyEnv and Virtualenv

Tox is a small, popular python framework that can run unit tests in multiple python environments. It can be used to run unit tests with multiple versions of python if those versions are installed.

PyEnv is a small application which can download and compile specific versions of python and run them alongside one another.

Virtualenv is a tool for creating a python environment where you can install an isolated group of packages which you can use to run or test an application that depends upon them.

Hitch can supplant tox for integration tests (See : How to Parameterize your Test Cases with Hitch).

Hitch *bundles* pyenv and uses it to build a python virtualenv(s) for you.

It does this with two lines of code:

```
# Define the version of python you want
python_package = PythonPackage(version="3.4.3")

# Installs python 3.4.3 into ~/.hitchpkg (if it isn't already present)
# Creates virtualenv in .hitch folder (if it doesn't already exist)
python_package.build()

# Python virtualenv you can use with your project:
python_package.python == "/path/to/your/project/tests/.hitch/py3.4.3/bin/python"
python_package.pip == "/path/to/your/project/tests/.hitch/py3.4.3/bin/pip"
```

The above applies to the following packages:

- hitchpython
- python-build

Note: Hitch *also* uses virtualenv to isolate *itself* and the code it runs the `/glossary/execution_engine` with. This is a virtualenv created with your system's python 3.

py.test/nose/unittest2

py.test, nose, unittest and unittest2 are all unit test frameworks, although they are often used to write integration tests.

See When should I use a unit test and when should I use an integration test?

[TO DO : parameterization, readability, boilerplate to handle services, isolation features, loosely coupled, muliple services]

Robot Framework

[TO DO]

Other technologies?

If you'd like to see a comparison with other technologies here or would like to correct something said above, raising a ticket is welcome:

https://github.com/hitchtest/hitch/issues/new

3.3.3 How is Hitch extensible?

Hitch is an integration testing framework built upon python. While there is a lot of in-built functionality, you will find at some point that the included portions of hitch are not sufficient to mimic realistic scenarios.

3.3.4 Easy step definition

While Hitch contains a myriad of in-built steps to perform common tasks like clicking on buttons and receiving emails, it probably won't do everythingthat you need. For this reason you will need to occasionally define your own steps.

- Click: my-button
- Do some fancy thing
- Click:

3.3.5 Loose `/glossary/coupling`

Hitch is built with loose coupling in mind at every level.

3.3.6 Installed from PyPi into a Virtualenv

Hitch packages itself as a

3.3.7 What about python 2?

This hitch framework only runs in python 3.

The bootstrapper (which runs fine in python 2 and 3) installs hitch in its own isolated virtualenv in the .hitch directory.

This means that the code you write in the `/glossary/engine.py` and any code it imports must be python 3 code.

My code only runs in python 2. Can I test it with hitch?

Yes. You can test your code in any version of python 2 from 2.6.6 to 2.7.10 and any version of python 3 from 3.2 to 3.5.0 and any combination of those versions.

3.3.8 What does hitch init do?

The "hitch init" command creates a `/glossary/hitch_directory` in the current directory if one does not already exist. Inside this directory it installs a `/glossary/virtualenv` with the "hitchtest" package and dependencies, or, if a `/glossary/hitchreqs.txt` is found, all of the packages specified in "hitchreqs.txt"

This virtualenv contains only the packages required to run tests. It does *not* contain the packages required to actually run your app. The tests themselves will take care of setting those up in a separate virtualenv.

See also:

- What does the init script do?

3.3.9 What does the hitch bootstrap script do?

The hitch bootstrap script is a simple script that you install by:

> $ pip install hitch

This is a very simple script with no dependencies that has been tested on python 2.6, 2.7, 3.3 and 3.4.

It does the following 4 things:

- Initializes the .hitch directory - containing the virtualenv used for running the hitch tests (separate from the virtualenv used for running your application).
- Deletes the .hitch directory (hitch clean).
- Triggers test runs (hitch test [arguments] will run the command .hitch/virtualenv/bin/hitchtest [arguments]).
- Upgrades all the packages specified in hitchreqs.txt (the software installed in the virtualenv used to run your tests).

3.3.10 What does the init script do?

Note: This script tries to respect your existing environment as much as possible and avoids the use of sudo except where necessary to install packages via your system's package manager.

The init script is a one step method of setting up a hitch environment and running all the tests in a directory. It is intended to be a low friction way of:

- Getting a CI or test driven development environment up and running.
- Rebuilding an environment from scratch that may have been broken.

If you'd prefer instead to perform the steps manually, you can use this document as a guide.

Note that the first three steps take about 5 minutes and the last step can take roughly 15 minutes (or longer, sometimes).

1. Installs python, pip, virtualenv, python-dev, automake and libtool (may require sudo)

Takes approximately: 1 minute

These packages are required for hitch to initialize.

On Ubuntu/Debian:

```
$ sudo apt-get install -y python python3 python-dev python-setuptools python-virtualenv python3-dev a
```

On Fedora/Red Hat/CentOS:

```
$ sudo yum -y install python python-devel python-setuptools python-virtualenv python-pip python3 pyth
```

On Arch:

```
$ sudo pacman -Sy python python-setuptools python-virtualenv python automake libtool
```

On Mac OS X:

```
$ brew install python python3 libtool automake cmake

$ pip install --upgrade pip setuptools virtualenv
```

2. Install or upgrades the hitch bootstrap script (may require sudo)

Takes approximately: 5 seconds

This is a small python script with no dependencies that bootstraps your testing environment and lets you trigger test runs. It installs a single command ('hitch') on your system's path.

On the Mac the init script will run:

```
$ pip install --upgrade hitch
```

On Linux:

```
$ sudo pip install --upgrade hitch
```

See also:

- What does the hitch bootstrap script do?

3. Runs "hitch clean", "hitch cleanpkg" and "hitch init" in the current directory (may require sudo)

Takes approximately: 2 minutes

If no ".hitch" directory is already installed then this command does nothing. If a .hitch directory *is* found, it will remove it:

```
$ hitch clean
```

If no "~/.hitchpkg" directory is found, this will also do nothing. If you already used hitch before you may have packages downloaded into this directory, in which case it will destroy it so it can be rebuilt:

```
$ hitch cleanpkg
```

This builds a .hitch directory in the current directory and installs any more required system packages via unixpackage. This asks to install system packages specified in hitch plugins and packages specified in the system.packages file:

```
$ hitch init
```

- What does hitch init do?

4. Run "hitch test ." to run all tests (does not require sudo)

Takes approximately: 15 minutes (subsequent test runs will be quicker)

If there are tests in the directory where the init script is run, it will run all of them.

During the course of running the tests it will attempt to download and compile certain pieces of software (e.g. postgres). The software will be installed in the "~/.hitchpkg" directory. This does not require sudo and it will not interfere with software you may already have installed.

See also:

- Why is my test downloading and compiling software?
- Why does the first test run take so long?

All software installed there can easily be removed by deleting the "~/.hitchpkg" directory or running the command "hitch cleanpkg".

See also:

- How do I uninstall hitch completely?

3.3.11 What kind of debugging tools ship with Hitch?

Hitch takes the approach that, as a testing framework, it should not only make it easy to write and run tests, it should make it easy to find bugs by including first class debugging tools.

3.3.12 The IPython REPL

Hitch ships with IPython and integrates with it deeply. When you use the step 'pause' it is launched, at which point you have access to a wealth of tools to inspect the environment your app is running in, change the time in your app, run steps and much, much more:

```
In [1]: self.services.time_travel(days=30)

In [2]: self.services['Django'].manage("help").run()

In [3]: self.click("button-id")
```

While useful, this only gives you a bird's eye view and control over your application. In order to debug you will want to dig deeper.

As well as giving you access to the hitch environment via IPython, you can also connect directly to embedded IPython kernels and run a REPL in your code's environment *no matter what language your code is running in*:

```
- Click and don't wait for page load: button-id
- Connect to kernel: Django
```

3.3.13 Log Catting and Tailing

The first port of call when debugging any kind of application is to look in the logs.

By default, all services run under the Hitch service framework are asynchronously logged together, so while a test is running you can always see what the services you are running together are saying:

```
[              Django] Performing system checks...
[              Django] System check identified no issues (0 silenced).
[              Django] November 30, 2015 - 16:03:15
[              Django] Django version 1.8.7, using settings 'config.settings.local'
[              Django] Starting development server at http://127.0.0.1:8000/
[              Django] Quit the server with CONTROL-C.
[               Hitch] Django Loaded.
[               Hitch] READY in 5.4 seconds.
[          Err Django] [30/Nov/2015 16:03:33] "GET / HTTP/1.1" 200 34140
[          Err Django] [30/Nov/2015 16:03:33] "GET /static/css/bootstrap.min.css HTTP/1.1" 200 122887
```

While your test is paused and you're in IPython you can also directly inspect the logs of particular services:

```
In [1]: self.services['Django'].logs
[              Django] Performing system checks...
[              Django] System check identified no issues (0 silenced).
[              Django] November 30, 2015 - 16:03:15
[              Django] Django version 1.8.7, using settings 'config.settings.local'
[              Django] Starting development server at http://127.0.0.1:8000/
[              Django] Quit the server with CONTROL-C.
[          Err Django] [30/Nov/2015 16:03:33] "GET / HTTP/1.1" 200 34140
[          Err Django] [30/Nov/2015 16:03:33] "GET /static/css/bootstrap.min.css HTTP/1.1" 200 122887
```

3.3.14 When should I use a high level test and when should I use a low level test?

`/glossary/automated_testing` can be done on a variety of different levels - from `/glossary/low_level_testing` that surrounds tiny modules all the way up to `/glossary/end_to_end_testing` that covers an entire system and its architectural set up.

Both `/glossary/high_level_testing` and low level testing have their advantages, which is why one should not generally be used to the exclusion of the other.

A good project will have a variety of high level and low level tests, all used together. A good tester will pick the right level to suit the various trade offs.

Note that integration tests can be high level *or* low level, and the decision whether or not to use them requires an entirely different set of trade offs. See When should I use a unit test and when should I use an integration test? for that.

Note that `/glossary/test_driven_development` is useful in nearly all scenarios, but its usefulness is highly dependent on picking the *right level* to write tests for, as well as using the right tools.

New projects or legacy projects without tests

Picking a good default level to test at for a project is the best way to get off on the right foot.

By default, you should write tests at the `/glossary/highest_convenient_level`.

A good approach to take is `/glossary/behavior_driven_development` and `/glossary/acceptance_test_driven_development`.

In the beginning you will only have a few tests, so the implications of how long it takes to run the entire test suite will not be that big.

When you deploy your code and users start using it (whether they be simulated by QA or real, actual users), you will notice that you have bugs reported.

It is crucial that you write failing test cases for all of these bugs *except*:

- Those where the cost of building a harness to replicate them would be prohibitive.
- Those for which the fix is both trivial (e.g. a one liner) and mainly related to misunderstood requirements (e.g. changing text).

When should I extend my current test harness?

At some point you will find that you cannot replicate some bugs reported by users with your set up, but the cost of building a harness is not prohibitive.

You may find that the reported bugs are related to a browser that you do not run tests on, for instance.

At this point, you should very seriously consider extending your test harness so that it can handle this new source of bugs and incorporating it into your regression test suite.

When should I build a new, *higher* level test harness?

At some point you may find that you encounter a bug that is related to something that is *out of the scope* of your current test harness. If you encounter a deployment related bug, for instance, and your current harness does not even do deployment in the same way, that is probably the best time to build a new, separate test harness that can test deployment scenarios.

Deployment tests that create a virtual machine from scratch, upgrade it and install all necessary packages will be very slow, but there will be certain kinds of bugs that *only* they will catch.

You will need a few of these at some point.

When should I build a new, *lower* level test harness?

As you refactor projects that you work on, especially if you work on *decoupling* the software modules that it is comprised of, you will find that some of these modules start becoming more independent and need to know less and less about the system in which they are used.

Some of these software modules you may be able to replace entirely with already well tested 3rd party modules.

Some, however, you will not.

Once you consider the interface on these modules *stable*, that is a good time to wall them in with tests.

If the interface is stable, you should not suffer the ill effects of `/glossary/test_concreting`.

3.3.15 When should I use a unit test and when should I use an integration test?

Integration tests are tests that surround large software systems that are made up of a lot of interconnecting parts and interact with them as the outside world would. They are not necessarily written in the same language as the code they are testing.

Hitch is a testing framework designed specifically for writing integration tests for code written in any language on the UNIX platform.

Unit tests usually surround smaller blocks of code and directly call the APIs of the blocks of code they are testing. They are written in the *same* language as the code they are testing.

py.test and nose are good python unit testing frameworks. Other languages have their own equivalents.

Both unit tests and integration tests *can* catch bugs in *any* kind of code. However, they both have different trade offs and those trade offs, which you should bear in mind before writing either one.

When are integration test more useful than unit tests?

Integration tests are most useful for testing integration code.

Integration code is the kind of code which is mostly linking systems together. Most web apps are good examples of pure integration code - linking browsers, networks, REST APIs, email servers, databases, payment gateways and all manner of other APIs.

Integration tests that either use or *accurately mock* all of these components together can mimic the effect of scenarios that traverse all of them and uncover bugs that occur along the way. These are realistic tests (see `/glossary/test_realism`).

Unit tests that test the same kind of code usually do so by making extensive use of mock objects. Mock objects are *not only* unrealistic representations of real objects (almost by definition), using them in your tests *tightly couples* your code to your tests (see `/glossary/coupling`).

Thus mock objects could be considered a 'code smell'. Extensive use of them tends to indicate that you should have written an integration test to cover the code instead of a unit test.

When are unit tests more useful than integration tests?

Integration tests, especially *realistic* integration tests, uncover a greater variety of bugs than unit tests but they do run more slowly than unit tests.

This is *very often not a problem* - computing power is very cheap these days, after all, and feedback on automated tests does not have to be instantaneous or even quick to be vitally useful.

You can also optimize for speed (for instance, by parallelizing the test runs).

You can't optimize for unrealistic. You can only toss it away.

However, for *some scenarios* a unit test is *no less realistic* than an equivalent integration test. Code that is almost entirely *logical* rather than integrational is usually well suited to unit tests, for instance.

Such code can include:

- Algorithms that perform complex calculations
- Algorithms that are parsing text or other input
- Algorithms that do hard computing problems
- Functional code without side effects

The relative cheapness of running unit tests also makes it possible to apply tools such as Haskell's quickcheck / Python's Hypothesis to cover more input scenarios to functions than a human could even imagine in a short space of time.

What about code where algorithmic and logical code is thoroughly mixed together?

It is often the case with legacy code bases that algorithmic/logical code (e.g. functions that calculate pricing) and integrational code (e.g. code for storing/retrieving data in the database) are thoroughly mixed up and cannot easily be separated.

This is an example of tight `/glossary/coupling`.

In such cases, unit tests are of very limited value as well - at least initially.

A good approach to code bases like these is to gradually surround the entire code base with integration tests and *then* refactor it until the logical/algorithmic code and the integrational code are separated from one another.

Once the logical code and the integration code are separated and talk to one another across tight, well defined API boundaries, unit tests become useful again on the logical code.

See also:

- `how_should_i_choose_where_to_surround_my_code_with_tests`

3.3.16 Why do Hitch tests download and compile software I already have installed?

Hitch takes a novel approach to integration testing that means that it will often download and compile software which you *may already have installed*.

Some examples include:

- Python

- Postgresql
- Redis

This is done in order to help solve three common problems that plague integration tests:

- `glossary/test_brittleness`
- `glossary/indeterminacy`
- `glossary/isolation`

While you *may have* postgresql installed on your computer when you ran a hitch test, the actual version you have installed could be different depending upon your OS, the version of your OS and the last time you upgraded your OS.

It will also change *as you upgrade your OS*.

Since the actual version of the software installed is so often *critically* important for chasing down obscure tear-your-hair-out bugs, hitch takes the approach that, by default, it should download and install critical components that your tests rely upon itself and to let you *specify* the version as well.

Unfortunately this *will* often lead to a slower first test runs and that software does take up extra space (it is all stored in the folder ~/.hitchpkg if you are curious).

However, it'll save time in the long run, *especially* on complex systems as it will mean you'll spend less time debugging the "hair pulling problems" caused by using different versions of software for development and production.

3.3.17 Why does the first test run take so long?

Your first test run can take a long, long time because hitch often needs to download and compile the software used to run your test with.

This means that the first test run of one test can take in the region of 45 minutes while subsequent runs of the same test take 20 seconds.

See also:

- Why is my test downloading and compiling software?

3.3.18 Why install Hitch on the system path?

The instructions in quickstart tell you to install hitch on the system path like so:

```
$ sudo pip install hitch
```

Or (on Mac OS X):

```
$ pip install hitch
```

This package *can* be installed in a virtualenv, but there is no point to doing so, since:

- It is just a simple bootstrapper script.
- It contains none of the actual code to run tests.
- It has zero dependencies so installing it will not interfere with other system packages.
- It needs to be available on the system path so that the 'hitch' command can always be used.

See also:

- Why should my tests set up their own python environments?.

- How do I uninstall hitch completely?.

3.3.19 Why is hitch so fast?

There are two main reasons why Hitch is generally faster than other testing frameworks: parallelization and built in epoll/kqueue triggers.

Automatic Parallelization

When hitch services are started they are started in parallel by default. If you have seven services (like the example project), hitch will try to start all of the services that do not have a "needs" property set. As soon as services are ready that are prerequisites of other services, those will be started.

This means two things: even very complex service architectures can be started extremely quickly and also that your test speed will increase substantially the more CPU power, RAM and CPU cores you have.

As an example, the django-remindme-tests project runs the following services:

- Postgresql (including running initdb to create all necessary database files and creating a user and database after service start-up)
- Django (including installing fixtures and running migrations)
- Mock Cron server
- Mock SMTP server
- Celery
- Redis
- Selenium (running and connecting to firefox).

When tested on a laptop with an SSD, and an i5 processor with 4 CPU cores, just starting firefox takes 4.5 seconds. *All* of the above, when parallelized, takes between 5.1 and 5.8 seconds.

Epoll/Kqueue Triggers

A common feature of other testing frameworks is the use of 'sleeps' and polling to determine if an event has occurred. This can not only contribute to test indeterminacy, it slows down your integration tests.

A feature of hitch that contributes to its speed is the in-built use of epoll/kqueue triggers. These are kernel features in Linux, FreeBSD and Mac OS X that allow 'watches' to be put on files. When a file is changed, the test is automatically notified without the need for polling.

This is used in the following situations:

- To ascertain service readiness - the instant that Postgresql logs the line "database system is ready to accept connections", for example, Hitch will move straight on to creating users and databases.
- Mock service interactions - the instant that the mock SMTP server receives an email, it logs out a snippet of JSON. The watcher on the mock SMTP logs receives the epoll trigger during that split second and the test can continue.

3.3.20 Why is my test downloading and compiling software?

Hitch usually downloads and compiles the software (e.g. python/postgres) it uses to run your test with on the first test run.

Often this is software you may already have installed on your machine.

This is done to:

- Maintain `/glossary/package_isolation`.
- Cut down on `/glossary/brittle_tests`.
- Allow a greater level of `/glossary/test_realism`.

Examples of where this helps (with python):

- Upgrading your system will not cause behavior changes in your tests because the version of python has changed.
- You can develop and test against the exact same version of python that you use on production.
- You can tweak the version in your tests before running a full test run to give confidence that upgrading your version of python in production will not cause bugs.
- You can trivially test your code against *multiple* versions of python on the same test run (see How to Parameterize your Test Cases with Hitch.).

While it is inconvenient to download and compile software on the first test run, there is a substantial benefit to taking this appraoch.

See also:

- Why does the first test run take so long?

3.3.21 Why should web application tests only interact with HTML IDs and HTML classes?

The Hitch framework advises an opinionated approach to testing web applications that is unusual among testing frameworks.

The basic idea is very simple. Imagine there is some thing you want to click on. The idea is:

- Any *individual* element *that you wish to click on* with should have its own *readable* HTML ID.
- Any element that is part of a group (or a group of groups) that you wish to click on should have one or more *readable* HTML classes.
- Any *test step* that interacts with or checks the web app should use only IDs or HTML classes to identify elements.

This means no "find by text" selectors, no "find by name" selectors and *certainly* no "find by xpath".

The controversial part

The controversial part of this is the idea that *testers should change HTML code* in order to create readable HTML ids or classes for use with their tests.

Many organisations restrict QA access to code bases. This is an organizational deficiency which will damage the quality of the tests produced.

Why this opinionated approach is important #1: loose coupling

Loose `/glossary/coupling` is the general idea that software components should know as little about each another as possible.

The components we're looking at here are "tests" and "your web app".

With this approach, what the test 'knows' about the web application should be restricted to the HTML classes and IDs, so it is more loosely coupled than other approaches.

Loose coupling is important because it lets you change individual software components without changing the components that rely upon them.

In this instance that means changing the code without needing to change the test.

In web application tests, tight coupling can manifest in ways that will often end up breaking tests under benign conditions (see `/glossary/brittle_tests`):

- Changing the the text in an element (e.g. the exact error message) will break tests that rely upon seeing that exact text for a certain step.
- Changing the language of a web app and then following the same scenario will also break if the test relies upon the elements containing specific English phrases.
- Using an xpath almost *guarantees* that a change to the HTML layout or CSS will cause a test to break.
- Similarly, using a complicated CSS selector will do the same.

Why this opinionate approach is important #2: readability

The second reason why restricting your tests to interacting only with classes and IDs is `/glossary/readability`.

The more complex an element selector is, the more unreadable it becomes. e.g.:

```
- Click: register
```

Is pretty readable. This:

```
driver.find_element_by_css_selector(".buttons.pageitems.reg").click()
```

Is less readable. And this:

```
driver.find_element_by_xpath("//div[class='buttons'][0]/input").click()
```

Is just nasty.

Unreadability increases with complexity and language power, which is why the approach of simple and dumb for high level tests, is crucial for `/glossary/test_quality`.

Should testers be allowed to *change* IDs and classes?

In general, no, although *raising an issue* and asking developers to change IDs and classes to help make tests more readable should be encouraged.

While *adding* HTML IDs and classes has a low chance of causing problems, *changing* them can cause problems - usually with javascript or CSS that relies upon specific names for element IDs and classes.

That said, if they're careful and check first, why not.

Do I absolutely *have* to test web applications this way in Hitch?

No.

You are not restricted to only using CSS and HTML IDs. It is a recommended approach, supported by the built in tools. It is by no means an approach you are forced to take by the framework. Creating your own steps with hitch is easy enough if you know some python and you are provided access to the python selenium webdriver object should you wish to take another approach to interacting with your web application.

It is possible that the above approach may not work in every scenario you may need to test anyway.

See also

- Clarifying documentation (if you thing you see a flaw in this approach, raise a ticket).

3.3.22 Why does Hitch not enforce the use of Given/When/Then?

The three core principles behind the `/glossary/hitch_test_description_language` are `/glossary/DRY`, `/glossary/test_readability` and simplicity.

That is:

- The test cases should be as short as possible while still communicating everything that they need to.
- Readability and simplicity are paramount - which means dispensing with superfluous syntax and key words, and the need for custom parser engines.

Given/When/Then are key words used to signal to a person *writing* the test the kind of pattern it should follow.

However, when the test case is finished, the meaning of the steps should be implicit anyway, and since *all* test cases should follow this pattern to some extent, the presence of the keywords becomes more superfluous the more test cases you read and write.

The terms have no meaning to the `/glossary/execution_engine` and are typically ignored.

Note: You are still *able* to use the terms given, when and then in your test step names with hitch, you are just not forced to do so.

3.3.23 Why should Hitch run my database?

Hitch does not require you to let it run your database, or, indeed, any other service for you, but there are several reasons why it might be a good idea.

Time Travel

If Hitch runs your database and you use hitchserve's time_travel method to mock moving forward in time, your database will move forward too. Thus, any queries that you run will also pick up the new time.

`/glossary/isolation`

If Hitch runs your database, the data files are kept isolated from other tests and other databases installed on your system. This protects your tests from modifications to the system DB environment and vice versa.

Version fixing

Hitch explicitly fixes the version of all of the software you use. This helps remove a source of test `/glossary/indeterminacy` from your test scripts.

3.3.24 Why should my tests set up their own python environments?

Most python testing frameworks run using the same python environment that the application code runs in.

Hitch is different:

```
$ sudo pip install hitch
```

The *bootstrap* script should ideally installed using the system python. This is a very small script with just one dependency.

This script sets up virtual environment to run the test code in:

```
$ hitch init
```

This runs using your system python3. This is the environment all of your testing code will run in.

Your tests will set up *another* environment to run your code in:

```
import hitchpython

python_package = hitchpython.PythonPackage(version="2.7.9")
python_package.build()      # Takes about 5 minutes during the first run. Instantaneous thereafter.
python_package.verify()
```

This not only ensures that the packages required to run your tests do not interfere with the packages required to run your code, it lets you be be specific about the version of python your test runs with.

Hitch is designed to test at as high a level as possible and to isolate all possible sources of bugs and test `/glossary/indeterminacy`.

The specific version and environment of python you test with is such a potential source of bugs.

Thus, to ensure the stability and repeatability of the test, this environment should ideally be brought under the test's control.

You can even download and create multiple versions of python to test your application in - for instance, if you want to test your app using python 2.7.9 and python 3.4.3. You can even easily test your code in every single version of python.

3.3.25 Why was hitch behavior changed?

This FAQ is gives explanations for behaviors that were changed in Hitch.

Why will my tests no longer run with the default filename "settings.yml"?

settings.yml used to be the default hitch settings filename. If no other settings file was specified, your test settings would be pulled from this filename. If you specified another filename on the command line, settings would be pulled from that *instead*.

The behavior has now changed. Settings are now *always* pulled from the filename 'all.settings' if it exists, but settings in any filename specified via –settings will always take precedence.

Settings are now taken from (in order of precedence):

- The JSON specified on the command line via –extra.
- The YAML filename specified via –settings.
- all.settings

See Settings for more information.

Why did you remove the –quiet switch?

–quiet is more appropriate as a setting, so it has been deprecated as a command line switch and will be removed entirely in version 1.0.

You can still make your tests quietly by setting the property to true on the command line via –extra:

```
$ hitch test . --extra '{"quiet":true}'
```

Or by adding the setting to a specified settings file, e.g.

```
quiet: True
```

See Settings for more information.

3.3.26 Why YAML?

The `/glossary/hitch_test_description_language` is built upon YAML - a markup language for presenting structured data.

Hitch also integrates the popular templating tool Jinja2 to let you *generate* a lot of very similar test cases in YAML without copying and pasting.

Why write tests with YAML instead of Python?

The hitch test description language is an *intentionally dumb language* with a restricted feature set.

It is *not* a full programming language like python**. You cannot use conditionals, loops, etc. It is just a simple sequence of steps and some data. It is essentially just configuration, in fact.

This may feel like handcuffs to a good programmer, but there's a good reason for it: less powerful languages are easier to understand and since tests are just series of steps, you do not *need* a powerful language to describe them.

Less powerful, easier to understand languages also have the following benefits:

- They are easier to maintain
- They are easier to keep free from bugs.
- You can *template* them and they are *still* relatively easy to understand, maintain and keep bug free.

Easier to understand also means that advanced programming skills are not necessary to write them. Some training and understanding is required - it's certainly not written English (for good reason), but it's more like learning how to use a spreadsheet rather than learning how to program in a turing complete language like python.

Jinja2 adds additional complexity, but it helps you to prevent your test suite from becoming repetitive. See: `/glossary/DRY`.

Again, Jinja2 is not a powerful language. It is more powerful than YAML but less powerful than python/java/ruby/etc. and should be something that a non programmer could pick up and use productively with a minimum of training.

Related reading

- https://en.wikipedia.org/wiki/Rule_of_least_power
- https://en.wikipedia.org/wiki/Separation_of_concerns

The use of YAML and Jinja2 in Hitch was inspired somewhat by Ansible: https://en.wikipedia.org/wiki/Ansible_%28software%29

3.4 Hitch API

Documentation for core Hitch components.

Contents:

3.4.1 Hitch Engine API

The Hitch Engine is a python class which is tasked with executing your tests and responding to successes and failures.

For a test like this, written in YAML:

```
- name: Example scenario
  scenario:
    - Do something
    - Do something else
```

The basic Hitch Engine, written in python, would need to look something like this:

```
import hitchtest

class ExecutionEngine(hitchtest.ExecutionEngine):
    def set_up(self):
        # set up code

    def do_something(self):
        # code run when test says "Do something"

    def do_something_else(self, with_what):
        # code run run when test says "Do something else"

    def tear_down(self):
        # code that always runs at the end
```

Step Translation

Test steps and their arguments are fed to the engine directly as method calls and arguments. All step names and arguments are first changed into underscore_case.

For example, putting this as a test step:

```
- Do something
```

Would be equivalent to calling this in your engine:

```
self.do_something()
```

This, on the other hand (note the semicolon):

```
- Do something else: value 1
```

Would be translated into:

```
self.do_something_else("value 1")
```

You can include as many arguments as you like in steps like so:

```
- Do complicated thing:
    Variable 1: Value 1
    Variable 2: 2
```

If the equivalent were written in python it would look like this:

```
self.do_complicated_thing(variable_1="Value 1", variable_2="2")
```

Your steps can also contain arguments that contain lists:

```
- Do another complicated thing:
    Variable 1: value 1
    Variable 2:
      - List item 1
      - List item 2
```

The python equivalent of that would look like this:

```
self.do_another_complicated_thing(variable_1="value 1", variable_2=["list item 1", "list item 2",])
```

They can contain dicts (or associative arrays) as well:

```
- A 3rd complicated thing:
    Variable 1: value 1
    Variable 2:
      Dict item 1: val 1
      Dict item 2: val 2
```

Which in python would be equivalent to this:

```
self.a_3rd_complicated_thing(variable_1="value 1", variable_2={'Dict item 1': 'val 1', 'Dict item 2'
```

Careful with semicolons and braces like { and }

Since the tests are written in YAML with optional Jinja2, braces and semicolons have special meanings and must be escaped if you want to use them.

Preconditions

self.preconditions is a dictionary representation of the YAML snippet in the test being run. What goes in this snippet is up to you. Anything that is valid YAML is allowed.

Example:

```
preconditions:
  db_fixtures:
    - fixture1.sql
  python_version: 2.7.3
```

This will mean your preconditions variable will be:

```
In [1]: self.preconditions
Out[1]: {'db_fixtures': ['fixture1.sql'], 'python_version': '2.7.3'}
```

You can access any properties you set here using python's get method (which you can also use to program in a sensible default):

```
In [1]: self.preconditions.get('db_fixtures', [])
Out[1]: ['fixture1.sql']
```

If no preconditions are set, self.preconditions will be an empty dict:

```
In [1]: self.preconditions
Out[1]: {}
```

Note that while preconditions can contain lists, you can't set preconditions to be a list.

Tags

Tests can also have tags, which let you single out individual tests to run or to run groups of tests together. Example:

```
- name: Test with tags
  tags:
    - registration
    - email
    - firefox
  scenario:
    - Step 1
    - Step 2
```

You can use these tags to run related sets of tests together like so:

```
$ hitch test . --tags registration
```

Or, if you want to be more specific, you can list the tags, separated by a comma:

```
$ hitch test . --tags registration,email,firefox
```

Description

You can also include comments in the description property. This where you can put comments in your tests to help explain to people what your test is doing and why.

It is ignored by the engine.

```
- name: Test with long description
  description: |
    This test has a long history behind it. First there was a feature, then
    ther was another bug BUG-431, which it was tweaked to accomodate.

    It registers, recieves an email and checks the email arrived.
  scenario:
    - Step 1
    - Step 2: with parameter
    - Step 3:
        var 1: 1
        var 2: 2
        var 3: 3
    - Last step
```

Stacktrace

self.stacktrace is an object representation of the stack trace that occurs after a failure occurs in your test. It is set to None if no error has occurred while running the test.

You can use it to pretty-print a representation of the last error that occurred:

```
In [1]: print(self.stacktrace.to_template())
[ prints colorized, pretty printed version of the stacktrace ]
```

You can also use it to *dive into* the specific engine code where the exception occurred, so that you can check the contents of variables at that point or even re-run the code:

```
In [1]: self.stacktrace[0].ipython()
Entering /home/user/django-remindme/django-remindme-tests/engine.py at line 122

In [1]: on
Out[1]: 'register'
```

Settings

Test settings are also available in the test engine, e.g.:

```
In [1]: self.settings
Out[1]:
{'engine_folder': '/home/user/django-remindme/django-remindme-tests',
 'pause_on_failure': True,
 'python_version': '2.7.3',
 'xvfb': False,
 'quiet': False}
```

To read more about setting settings see Settings.

3.4.2 Environment Checks

Note: This documentation applies to the latest version of hitchtest.

This feature is to help prevent those hours spent tracking down why your test failed on one machine (e.g. your coworkers' or your continuous integration machine) but not another, and it ended up being related to that machine's environment.

Hitch provides a variety of checks which you can use to verify that the environment you are running the test in is suitable to run the test in. If a problem with the environment is detected the test fails *immediately* with a clear error indicating the kind of fix required rather than failing later with an obscure, hard to debug error.

This feature is based upon the twin principles of `/glossary/isolation` and `/glossary/fail_fast`.

To use these checks, simply put them in your settings file under the property 'environment' or use them in your engine as described.

System Packages

This check verifies that a unixpackage (http://github.com/unixpackage/unixpackage/) package is installed:

```
environment:
  - packages:
    - libtool
    - libpq-dev
```

```
from hitchtest.environment import checks
checks.packages(["libtool", "libpq-dev", ])
```

This is a simple way of checking, in a *UNIX platform indendent* way if a list of packages are installed.

Debs

This check verifies that a particular set of debian packages are installed:

```
debs:
  - node-less
```

```
from hitchtest.environment import checks
checks.debs(["node-less", ])
```

If it is run on a non-debian system, this check is skipped.

Brew

This check verifies that a particular set of brew packages are installed:

```
brew:
  - libtool
  - automake
```

```
from hitchtest.environment import checks
checks.brew(["libtool", "automake", ])
```

If it is run on a non-Mac OS system or on a Mac OS system without brew installed, this check is skipped.

Internet detected after

This check should be used for all tests that rely upon access to the internet to function.

This check pings 8.8.8.8 (google DNS servers). If there is no valid response after the specified number of seconds, it fails:

```
internet_detected_after: 15
```

As soon as there is a response, the test will continue.

The package names used are, by default, Ubuntu package names. Note that the list of checkable packages is not very long, but if you want to use one which is not currently recognized, you can fork and issue a pull request to this repository: http://github.com/unixpackage/unixpackage.github.io

Free ports

This check verifies that the specified ports are not currently in use and fails if they are:

```
environment:
  - freeports:
    - 18080
    - 15432
```

Approved platforms

This check verifies that the test is being run on an approved platform:

```
approved_platforms:
  - darwin
  - linux
```

The platform type is checked against python's 'sys.platform'.

System bits

This check verifies that your system is either 32 bit or 64 bit:

```
systembits: 64
```

Need another environment check?

This list of checks is by no means all that you might need. Additional ideas for environment checks are very welcome. If there is one that you want, *please raise an issue* at http://github.com/hitchtest/hitchtest/issues/new

There's a good chance I'll be able to release a new version of the software with your check within a few days - a week at most.

See also : Clarifying documentation

3.4.3 Generic Service API

Note: This documentation applies to the latest version of hitchserve.

All of the services listed are run using the generic service API. This API lets you start, monitor and stop any kind of process during a test.

Defining a Service Bundle

To run one or more services together during your tests, you must first define a `/glossary/service_bundle` which will run them all together.

The definition of your service bundle should go in your `/glossary/test_setup` in the `/glossary/execution_engine`:

```
# Create a service bundle
self.services = hitchserve.ServiceBundle(
    project_directory=PROJECT_DIRECTORY,        # Default directory all of your services are started
    startup_timeout=15.0,                       # How long to wait for all of the services to startup
    shutdown_timeout=5.0,                       # How long to wait for all of the services to shutdo
)
```

Once your service bundle is defined, you can start defining services.

Defining a Service to Run

Next, your code needs to define services to run with your service bundle. You define generic services like this:

```
self.services['MyService'] = hitchserve.Service(
    command=["command", "arg1", "arg2", "arg3"],          # Mandatory - command to run the service
    log_line_ready_checker=lambda line: line == "READY", # Mandatory - function used to ascertain rea
    directory="/directory/to/run/command/in",             # Optional
    no_libfaketime=False,                                 # Optional (if set to True, the service is 
    env_vars={'A':1, 'B':2},                              # Optional (dictionary of environment varial
    needs=[self.services['Django']],                      # Optional (services to start and wait for 
)
```

Starting a service bundle

Once all of your services are defined, they still aren't started. To start your services you must call the startup method:

```
self.services.startup(interactive=False)
```

Note: interactive=False should be the default for all tests. However, if you want to run this command in an `/glossary/ipython` console, use interactive=True.

interactive=False will take over the console and start printing logs as they arrive. interactive=True will not take over the console.

Interacting with a Service Bundle: Switching to Interactive Mode

If you want your service bundle to stop logging to the screen (e.g. so you can launch IPython), you can start the interactive mode.

```
self.services.start_interactive_mode()
# Do interactive stuff here
self.services.stop_interactive_mode()
```

If you just want to print a log message during your test alongside all of the other logs, however, you can just use:

```
self.services.log("Your message here")
self.services.warn("A bad thing just happened")
```

Warning: Avoid using the print(“”) command to log messages. It will cause an error.

Interacting with a Service Bundle: Logs

Most services output information about what they are doing. In UNIX, there are two 'pipes' known as stdout and stderr where processes can log regular information and errors.

During normal operation in a test, both of these are logged to the screen, alongside the name of the service. E.g.:

```
[              Django] Performing system checks...
[              Django] System check identified no issues (0 silenced).
[              Django] July 11, 2015 - 10:36:58
[              Django] Django version 1.8, using settings 'remindme.settings'
[              Django] Starting development server at http://127.0.0.1:18080/
[              Django] Quit the server with CONTROL-C.
[          Err Django] [11/Jul/2015 10:36:59]"GET / HTTP/1.1" 500 99545
[          Err Django] [11/Jul/2015 10:36:59]"GET /favicon.ico HTTP/1.1" 404 2416
[          Err Django] [11/Jul/2015 10:36:59]"GET /favicon.ico HTTP/1.1" 404 2416
```

This will hopefully tell you most of what you need to know about why your services are reporting errors.

While a test is paused and interactive mode is switched off, you can access these logs via the log object:

```
In [1]: self.service['Django'].logs
[ Prints all of the logs ]
```

You can see the stdout and stderr individually, too:

```
In [2]: self.service['Django'].logs.out
[ Prints all of the logs ]

In [3]: self.service['Django'].logs.err
[ Prints all of the logs ]
```

As with the UNIX console, you can also tail your logs. This is a useful debugging tool:

```
In [4]: self.service['Django'].logs.tail.follow(lines_back=2)
[ Prints logs from two lines before the command starts. ]
[ Continues logging in real time until you hit ctrl-C ]
```

Interacting with a Service Bundle: JSON Logs

If you have a service which outputs JSON, hitch can read the logs and parse the JSON automatically. This is *extremely* useful for writing test steps that listen to a `/glossary/mock_service`.

Hitch also lets you grab a list of log lines encoded as JSON and return them as a list of dicts/lists. For example:

```
In [5]: self.service['HitchSMTP'].logs.json()
Out[5]:
[{'contenttype': 'text/plain',
'date': 'Tue, 14 Jul 2015 05:59:44 -0000',
'header_from': 'webmaster@localhost',
'header_from_email': None,
'header_from_name': None,
'header_to': 'django@reinhardt.com',
'header_to_email': None,
'header_to_name': None,
'links': ['http://127.0.0.1:18080/accounts/confirm-email/oro7rarxl8poqk9moe6jru5do6uoqijlllpcllmuqfac
'multipart': False,
'payload': 'User django at 127.0.0.1:18080 has given this as an email address.\n\nTo confirm this is
'sent_from': 'webmaster@localhost',
'sent_to': ['django@reinhardt.com'],
'subject': '[127.0.0.1:18080] Confirm E-mail Address'},
{'contenttype': 'text/plain',
'date': 'Thu, 13 Aug 2015 13:59:47 -0000',
'header_from': 'noreply@localhost',
'header_from_email': None,
'header_from_name': None,
'header_to': '<django@reinhardt.com>',
'header_to_email': 'django@reinhardt.com',
'header_to_name': '',
'links': [],
'multipart': False,
'payload': 'Remind me about upcoming gig.',
'sent_from': 'noreply@localhost',
'sent_to': ['django@reinhardt.com'],
'subject': 'Reminder'}]
```

This is a useful feature for verifying interactions with mock services went according to plan.

You can also tail the logs until a specific condition is met in a JSON line, for instance:

```
In [5]: self.services['HitchSMTP'].logs.out.tail.until_json(
            lambda email: containing in email['payload'] or containing in email['subject'],
            timeout=15,
            lines_back=1,
        )
[ returns full dict representation of JSON snippet representing email once it has been received ]
```

Interacting with a Service Bundle: Time Travel

Many bugs and test scenarios often cannot be realistically replicated without jumping through time.

The example application - a reminders app - is one example. To test that a reminder is really sent after 30 days, the application must *think* that 30 days have actually passed.

You can mimic these scenarios for services run using your service bundle by calling the time_travel API, which can be used like so:

```
In [1]: self.services.time_travel(days=1)
Time traveling to 23 hours from now

In [2]: self.services.time_travel(hours=25)
Time traveling to 2 days from now

In [3]: self.services.time_travel(minutes=60)
Time traveling to 2 days from now

In [4]: self.services.time_travel(seconds=60)
Time traveling to 2 days from now

In [5]: from datetime import timedelta

In [6]: self.services.time_travel(timedelta=timedelta(hours=1))
Time traveling to 2 days from now
```

If you forgot where you are, you can get the current (mocked) time via:

```
In [7]: self.services.now()
Out[7]: datetime.datetime(2015, 7, 19, 16, 21, 33, 703669)
```

To move to an absolute time:

```
In [8]: from datetime import datetime

In [9]: self.services.time_travel(datetime=datetime.now())
Time traveling to now
```

Note that if no_libfaketime is set to True for a service, it will not pick up on the new time.

Warning: This feature relies upon a C library called libfaketime.
Libfaketime sometimes causes buggy and unpredictable behavior in some programs (e.g. node.js and Java) on some platforms.
If you see problems when running a service, you may need to switch it off with 'no_libfaketime=True'.
Some programs will also work fine (e.g. firefox), but they will not pick up on the time being fed to them.
Libfaketime works well with python and postgresql.

Interacting with a Service Bundle: Connecting to a service's IPython Kernel

IPython kernels are a great way of debugging your code. They give you access to a REPL which you can use to inspect variables and run commands to see their effect.

With python code, you can invoke a kernel by putting the following line of code in your application:

```
import IPython ; IPython.embed_kernel()
```

Hitch provides a convenience function which you can use to listen to a service's logs and detect the presence of a recently embedded kernel and then connect directly to it and launch an interpreter in interactive mode.

```
def connect_to_kernel(self, service_name):
    self.services.connect_to_ipykernel(service_name)
```

This is a step that can be called just by adding

```
- Connect to kernel: Celery
```

Note that if you are connecting to a kernel after clicking a button in a web app, be sure to replace 'click' with the following step:

```
- Click and dont wait for page load: button-id
```

The regular click step will wait for the next page to load before continuing, which will never happen because your app paused on loading it due to the embed_kernel.

Interacting with a Service Bundle: The Process API

To see a service's process ID:

```
In [1]: self.services['HitchSMTP'].pid
Out[1]: 43215
```

To interact with or inspect the service's process:

```
In [1]: self.services['HitchSMTP'].process.<TAB>
self.services['HitchSMTP'].process.as_dict         self.services['HitchSMTP'].process.is_running
self.services['HitchSMTP'].process.children        self.services['HitchSMTP'].process.kill
self.services['HitchSMTP'].process.cmdline         self.services['HitchSMTP'].process.memory_info
self.services['HitchSMTP'].process.connections     self.services['HitchSMTP'].process.memory_info_e
self.services['HitchSMTP'].process.cpu_affinity    self.services['HitchSMTP'].process.memory_maps
self.services['HitchSMTP'].process.cpu_percent     self.services['HitchSMTP'].process.memory_percen
self.services['HitchSMTP'].process.cpu_times       self.services['HitchSMTP'].process.name
self.services['HitchSMTP'].process.create_time     self.services['HitchSMTP'].process.nice
self.services['HitchSMTP'].process.cwd             self.services['HitchSMTP'].process.num_ctx_swit
self.services['HitchSMTP'].process.exe             self.services['HitchSMTP'].process.num_fds
self.services['HitchSMTP'].process.gids            self.services['HitchSMTP'].process.num_threads
self.services['HitchSMTP'].process.io_counters     self.services['HitchSMTP'].process.open_files
self.services['HitchSMTP'].process.ionice          self.services['HitchSMTP'].process.parent
```

The psutil Process class API can be used to inspect the CPU usage of the service, its memory usage, list open files and much much more.

The full API docs for psutil's Process class are here: https://pythonhosted.org/psutil/#process-class

Interacting with a Service Bundle: Service Sub-commands

Many services have special commands which are run during their operation. For example, Django has the manage command, Redis has redis-cli and Postgresql has psql.

Hitch provides an API to let you run these commands in the same environment as the service you are running. This means that they will inherit the same environment variables and time:

```
In [1]: self.services['Django'].manage("help").run()
```

Running Arbitrary Code Before and After Starting a Service

Some services can just be started and stopped, but others require special code to be run before and after. A good example of this is postgresql, which requires initdb be run before starting the database service, and CREATE USER / CREATE DATABASE to be run after.

If your service has special requirements like this, you can subclass the hitchserve Service object and override the setup and poststart methods:

```python
from hitchserve import Service
import signal


class MyService(Service):
    def __init__(self, **kwargs):
        kwargs['log_line_ready_checker'] = lambda line: "line in logs that signals readiness" in line
        kwargs['command'] = ["start_service_command", "arg1", "arg2", "arg3", ]
        super(MyService, self).__init__(**kwargs)

    def setup(self):
        """This is where you run all of the code you want run before starting the service."""
        pass

    def poststart(self):
        """This is where you put all of the code you want run after the service is ready."""
        pass
```

3.4.4 Hitch Command Line Interface

Note: This documentation applies to the latest version of hitch and hitchtest.

hitch init

If not already initialized, running:

```
$ hitch init
```

Will do the following:

- Create a .hitch directory
- Check for and ask you to install the following system packages: python-dev, python3-dev, libtool, automake, cmake (or equivalents).
- Check that the system packages specified in the file system.packages are installed. Attempt to install them if not.
- Installs all hitch plugin packages specified in hitchreqs.txt.
- Ask you to install any system packages required by plugins and not already installed.

If already initialized, hitch init will:

- Install any package versions specified in hitchreqs.txt but not installed in your environment.
- Verify again that all required system packages from system packages, hitch plugins.

You should run this command whenever hitchreqs.txt is changed (for example if you do a git pull).

Note: If you are running tests in a continuous integration environment, you should ensure that hitch init is run before every test run and that the user it is run as has passwordless sudo.

You should also run hitch init on your development environment if somebody else changes your hitchreqs.txt.

hitch test [tests]

To run a specific .test file:

```
$ hitch test stub.test
```

To run *all* of the tests in a directory (and its subdirectories), e.g.:

```
$ hitch test .
```

You can also specify multiple files and directories to test, e.g.:

```
$ hitch test apptests/ apitests/ stub.test
```

hitch test [tests] –tags

To only run tests that match specified tags, you can use the –tag switch to specify the tests you want to run:

```
$ hitch test . --tags tag1,tag2,tag3
```

hitch test [tests] –settings

You will at some point want to run the same tests in different environments. This is what hitch settings are for.

To use a different settings file, you can use the –settings switch to specify the settings file to use. For example, for a continuous integration environment you might run:

```
$ hitch test . --settings ci.settings
```

Whereas in your test driven development environment you might run:

```
$ hitch test . --settings tdd.settings
```

These commands will ensure that your tests are run with settings from both all.settings and the YAML settings file specified on the command line.

You can also specify settings using JSON on the command line by using the –extra switch, e.g.:

```
$ hitch test . --settings ci.settings --extra '{"failfast":true}'
```

For more about settings and configuration see Settings

hitch test [tests] –yaml

If you are using Jinja2 to parameterize your tests, you may want to just display the YAML generated by it to check that your jinja2 has no problems and is templating your tests correctly.

You can do that using the –yaml switch:

```
$ hitch test simple_reminder.test --yaml
```

hitch install

Once you have an environment set up, you can install hitch plugin packages to test different kinds of things. For example:

```
$ hitch install hitchselenium
```

This will install the hitch plugin package responsible for testing web applications with the firefox browser and selenium.

Hitch install will do the following when you run 'hitch install':

- Download the hitch plugin package from pypi and install it in your environment.
- Update hitchreqs.txt with the plugin package as well as any packages it depends upon.
- Attempt to install any *system packages* required by the plugin package. For instance, hitchselenium needs firefox and xvfb.

Note: Note that hitch plugin packages are just regular python packages downloaded from pypi. Running this command is roughly equivalent to running pip install.

If you want to install any other python packages to use with your engine.py, you can use this command.

hitch uninstall

This lets you uninstall plugin packages from your environment. For example:

```
$ hitch uninstall hitchselenium
```

It will also update hitchreqs.txt afterwards.

hitch upgrade

See : How to upgrade Hitch

This command will upgrade all of the plugins installed in your hitch directory and save all of their newly fixed versions to hitchreqs.txt.

hitch clean

This command removes the .hitch folder. It is a good idea to run this command and run hitch init after if you suspect your .hitch directory might have been corrupted.

hitch cleanpkg

This command removes the ~/.hitchpkg folder, which contains all of the hitch packages downloaded, compiled and installed in order to run tests.

It is a good idea to run this command and re-run your tests after if you suspect your ~/.hitchpkg may have been corrupted.

3.4.5 Hitch Package API

Note: This documentation applies to the latest version of hitchtest.

The `/glossary/hitch_package` API is an API which allows tests to download and install their own software to use in the ~/.hitchpkg directory. It allows testers to easily specify exact versions of software to test with and test with multiple versions.

Note that the `/glossary/hitch_package` API is not yet stable, so building packages on top of it is not recommended yet.

The following documented, tested :doc:‘/glossary/hitch_plugin‘s are currently using the package API to download and install software you can use to test your applications with:

- HitchPython
- HitchRedis
- HitchPostgres

The following undocumented plugins are also using the package API:

- HitchMySQL
- `/plugins/hitchrabbit`
- `/plugins/hitchelasticsearch`

- `/plugins/hitchmemcache`

3.4.6 Settings

`/glossary/hitch_settings` are a set of configuration variables used by your tests that potentially change test behavior.

Settings are for behaviors for all of your tests that you may want to change on a case by case basis or a per-environment basis. They are available via:

- self.settings in `/glossary/engine.py` and the `/glossary/ipython` console.
- All jinja2 templates

With a few exceptions, what these settings mean and what they do is left up to you.

Changing settings via the from command line

You can set settings via the command line by specifying them in a snippet of JSON. For example:

```
$ hitch test sometests.test --extra '{"failfast":false}'
```

This will override the default behavior from "stop on on first failure and report" to "run all tests and report failures at the end".

This variable is accessible in engine.py and via the IPython shell via self.settings, e.g.:

```
In [1]: self.settings
Out[1]:
{'failfast': False,
 'engine_folder': '/home/user/django-remindme/django-remindme-tests',}
```

Or:

```
In [1]: self.settings.get("failfast")
Out[1]: False
```

Per-environment settings

Some settings you may want to apply consistently to a certain environment or use case. For example:

- TDD (test driven development environment) - browser is configured to be shown to the developer on all tests.
- CI (continuous integration environment) - browser is configured to run headless on all tests.

Rather than typing all of them in to the command line in JSON, these settings can be set in a YAML file.

Here's an example 'ci.settings' file you might use for a continuous integration environment:

```
python_version: 2.7.3
xvfb: true
failfast: false
pause_on_failure: false
pause_on_success: false
```

You can use the settings from this file by running:

```
$ hitch test sometests.test --settings ci.settings
```

Here, self.settings would be:

```
{'engine_folder': '/home/user/django-remindme/django-remindme-tests',
 'failfast': False,
 'pause_on_failure': False,
 'python_version': '2.7.3',
 'xvfb': False,
 'quiet': False}
```

Override priority

If any settings are set in a settings file *and* on the command line, the command line setting will take precedence. E.g.:

```
$ hitch test sometests.test --settings ci.settings --extra '{"failfast":True}'
```

self.settings:

```
{'engine_folder': '/home/user/django-remindme/django-remindme-tests',
 'failfast': True,
 'pause_on_failure': False,
 'python_version': '2.7.3',
 'xvfb': False,
 'quiet': False}
```

Global settings

Some settings you will want to apply to all test runs unless specified otherwise.

These settings can be put in a YAML file called 'all.settings' which is read implicitly if it exists. For example:

```
.. code-block:: yaml
```

python_versions:

- 2.7.3
- 2.7.10
- 3.4.3
- 3.5.0

This specifies a default list of python versions to test with. However, it can be overridden via a specified settings file or with the –extra switch on the command line.

For example:

```
$ hitch test sometests.test --extra '{"python_versions":["2.7.3", "3.5.0"]}'
```

Getting settings in engine.py

To use the settings in engine.py you need to access the settings dict. This can be done like so:

```
self.settings["python_versions"]
```

This will either fail if 'python_versions' is not set or return the setting as a python variable (e.g. a list in this case).

Sometimes failure is what you want.

Alternatively, if you want a default to be used in case no setting is set, you can access the setting this way instead:

```
self.settings.get("python_versions", [])
```

This will *not* cause the test to fail if python_versions is not set. Instead it will just return an empty list.

Special settings

The following are special settings which change the way hitch behaves regardless of what is in engine.py:

- failfast – this causes all test runs to stop on the first test that fails. Useful for TDD.
- colorless – this stops color codes from being output in stacktraces - for systems like Jenkins that cannot interpret them correctly.
- quiet – this conceals output produced by the test. It can be used to make long test logs less unwieldy.
- show_hitch_stacktrace – by default the stacktrace displayed on errors conceals lines in hitch. This is for debugging exceptions in hitch itself.

3.5 Miscellaneous

Contents:

3.5.1 Clarifying documentation

Was there something you were confused about? Are you having problems with hitch?

If you are finding a test hard to implement or a problem with the framework hard to fix, please raise an issue:

https://github.com/hitchtest/hitch/issues/new

Even if you think the problem was probably your fault we really want to hear from you. There's a good chance if you're confused that the documentation or even the code is deficient in some way and we'd like to fix that.

Support queries are the lifeblood of this framework.

Thanks for helping out!

3.5.2 Contributing to Hitch

Hi and welcome! Thanks for your interest in contributing to Hitch!

There's a number of things you can do to help its progress:

#1 Fill out my survey

Who are you? What do you want? What do you need help with? I'd like to know!

Your input can help greatly in guiding the future development of Hitch.

[survey not yet ready. check back soon.]

#2 Run the example project on a currently untested environment

See `/tested_on` for details.

Just follow the 3 step directions on https://hitchtest.com/ to run the test on your environment.

If it works, please consider editing and adding it to https://github.com/hitchtest/hitch/blob/master/docs/misc/tested_on.rst or raising an issue with the details.

If it doesn't work, please raise an issue and copying and paste the output you saw.

#3 Report *any* issues you see at any time

If you're not sure if what you saw really was a bug, raise an issue anyway. Copy and paste the output.

If you're not sure if it was your fault or not, please still raise an issue. Copy and paste the output.

If it's not clear and simple and easy to understand what happened, it's *probably a bug*.

#4 Raise an issue for anything that confused you in the documentation or you thought was missing

Writing documentation is hard. What is obvious to the documentor won't necessarily seem obvious to the user. I'd really like to see more of these.

#5 Help out with the unixpackage project

Hitch is substantially reliant upon this project, but it still needs a lot of (mostly sysadmin-y non-programming) work.

See https://github.com/hitchtest/hitch/blob/master/CONTRIBUTING.rst for details.

3.5.3 Contributing code

Pull requests are welcome too, of course. If you submit a PR to any hitch project, please remember to sign the CLA:

https://www.clahub.com/agreements/hitchtest/hitch

There's no cloud continuous integration but all code is thoroughly tested before being deployed to pypi.

My ideal pull request would be:

- Small and self contained (ideally just a few lines)
- Pythonic and elegant (see Raymond Hettinger's talks for an idea of what that means)
- You've run that code at least once
- You'll amend it based upon any comments I might have

100% bug-free isn't a requirement. If I discover a problem during regression tests and the code breaks under certain less than obvious conditions, that's my problem. If it breaks some other part of the system that's my problem too.

If you want to get started fixing or implementing something, but you're not sure how, raise an issue.

3.5.4 Contributors

- Jon Hadfield @jonhadfield
- Omer Katz @omerkatz
- Flavio Curella @fcurella

3.5.5 Additional thanks to

- Phoebe Bright @phoebebright
- Rui Pacheco @ruipacheco
- Andy Baker @andybak
- Audrey Roy Greenfeld @audreyr
- Daniel Greenfeld @pydanny

3.5.6 Roadmap

This is a non-exhaustive list of features which are planned for the future, in no particular order:

- Service plugins for lots more popular databases, web frameworks, task queues and more - even in PHP, Java, Ruby, etc. and tutorials to use them.
- Use of py.test's assert statement.
- Tests that repeat themselves with % pass/failure and a threshold for passing (default 100%), for those annoying integration tests that only fail sometimes due to race conditions.
- Configurable mid-steps - running an engine method between each test step - e.g. to pause mid-step when using tests for demonstrations or take screenshots.
- Tools to let you stop and start services mid-test, e.g. to test how they behave when receiving different UNIX signals, or to mock scenarios where services are restarted.
- Step skipping
- Bisect - tools to be able to figure out which commit caused a test failure during long test runs.
- Mock REST server
- CPU/Memory/I/O tracking for services.
- Artefact generation - a seamless way of creating artefacts from tests such as screenshots, CPU/Memory/I/O usage reports, code coverage reports, etc.

3.5.7 Tested on

Working on

This is an list of environments that Hitch is tested on and been found to work. If you use one of these environments and you discover a problem, that's a bug. Please raise an issue.

OSes:

- Ubuntu 14.04
- Ubuntu 15.10

- Mac OS X Mavericks
- Mac OS X Yosemite
- Mac OS X Mavericks
- Debian Jessie
- Fedora 20
- CentOS 6.4
- Arch (latest rolling release)

Additionally, the following environments have been tested with and seem to run Hitch okay:

- Docker
- Jenkins
- Travis CI

Not working on

This is a list of other UNIX systems that Hitch will not currently work on, but might be made to function with with some work. If you really want one of these systems to run hitch, that's a feature request. It may happen if you raise an issue.

- Mandriva
- OpenSUSE
- Mac OS X with macports
- BSD
- Gentoo
- Mac OS X with no package manager
- Slackware
- LFS

Note: 95% of getting these environments to work involves getting unixpackage to work with them. Please consider helping out with it if you'd like to one of these flavors of UNIX / environments supported.

This is a list of environments that probably aren't happening in the near future but may happen one day:

- Cygwin
- Windows

Untested but should still work

This is a list of environments that Hitch has *not* been tested on, but hopefully should still work, but I'm not as confident about.

If it doesn't work, that's a bug. Please raise an issue if you find a problem.

If you have access to one of these, *please* try out the example project and submit a pull request adding your environment if it works or raise an issue if it doesn't:

- Red Hat
- Ubuntu/Debian based distros like Trisquel, Kali or Mint
- Red Hat based distros like Oracle Linux
- Linux systems not mentioned above
- Any continuous integration system not mentioned above

See the full `/glossary/index` here.