Ra Documentation

Release

Ramses Tech

Oct 05, 2017

Contents

Source code:http://github.com/ramses-tech/ra

Ra is a test helper and pytest plugin for testing APIs defined by RAML files.

It is used with pytest and WebTest to write functional tests for APIs a natural way that mirrors the structure of the RAML file.

Ra also provides default automatic testing of resources defined in the RAML to validate responses.

Ra is designed primarily to be used with Ramses, but should be usable with any WSGI-based app that has a RAML definition.

CHAPTER 1

Table of Contents

Getting Started

Install Ra with pip in your existing WSGI API project:

```
$ pip install ra
```

Requirements

- Python 2.7, 3.3 or 3.4
- WSGI-based app providing an API (this guide assume a Ramses app)
- API definition in RAML

Basic Example

This is a quick example using Ra, pytest and WebTest to test some API resources.

Assuming a simple RAML definition (and an app that implements it):

```
#%RAML 0.8
# ./api.raml
---
title: people API
mediaType: application/json
protocols: [HTTP]

/people:
    post:
        description: Create a new person
        body:
            application/json:
                example: { "name": "joe" }
```

Create a new test file:

```
# ./tests/test_api.py
import pytest
import ra

# Paths are resolved relative to pytest's root (where its ini file lives).
# The Paste Deploy URI 'config:test.ini' is the default value for app.
api = ra.api('api.raml', app='config:test.ini')

@api.resource('/people')
def users_resource(users):

    @users.post
    def post(req):
        # the request will automatically use the example body from
        # the RAML definition
        response = req(status=201) # asserts status code is 201
        assert 'joe' in response
```

This loads the app described by the Paste Deploy file `test.ini` and reads the RAML definition at `api.raml`, relative to pytest's root directory (where its ini file lives, usually the project root). Run tests with `py.test tests/test_api.py`. Ra will read the RAML definition, make a request `POST /people` with the example body `{ "name": "joe" }`.

Most likely, this will succeed the first time, and fail on subsequent runs as "joe" already exists. See Writing Tests for an example of using factories to generate unique request bodies, and Hooks for using before/after hooks to set up a clean testing environment.

Writing Tests

Ra uses a simple DSL for describing tests in a structured way, similar to the RAML definition.

Resource scopes:

Tests are organized by resource in "resource scopes", which can be nested:

```
api = ra.api('example.raml')

@api.resource('/users')
def users_resource(users):

    # tests for /users resource

    @users.resource('/{username}')
    def user_resource(user):

        # tests for /users/{username} resource
```

The resource scope (e.g. function `users_resource` above) takes an argument: it will be passed a `ResourceScope` object that is used within the scope to define tests or nested scopes.

By default, requests made in resource tests will use the example body defined in the RAML if it exists (only `'application/json'` is currently supported). You can override this behaviour and use custom resource factories:

```
def user_factory():
    import string
    import random
    name = ''.join(random.choice(string.ascii_lowercase) for _ in range(10))
    email = "{}@example.com".format(name)
    return dict(username=name, email=email, password=name)

@api.resource('/users')
def users_resource(users, factory=user_factory):

    # tests ...
```

Also by default, resources with URI parameters will have the parameter filled with the example value defined in the RAML if it exists. It can be overrided when the scope is declared:

```
# ...
/users/{username}:
    uriParameters:
        username:
            type: string
            example: finn
```

```
# ...
@users.resource('/users/{username}')
def user_resource(user):
    # {username} will be "finn"
    # ...

# or:

@users.resource('/users/{username}', username='jake')
def user_resource_overriding_username(user):
    # {username} will be "jake"
    # ...
```

Either way, for testing an item resource, you'll probably want to use fixtures (see Test fixtures) to set up a resource by that name before these tests.

pytest fixtures defined in resource scopes are local to that scope (behind the scenes, resource scopes are treated just like modules by pytest):

```
@users.resource('/users')
def users_resource(users):

    # local to this scope:
    @pytest.fixture
    def myfixture():
        return 1

    # ...
```

Tests

Within resource scopes, define tests for the methods available on that resource.

```
@users.resource('/users')
def users_resource(users):

    @user.get
    def get(req):
        # do some test-specific setup ...
        response = req()
        # do some WebTest assertions on the response ...
```

The test function parameter `req` is provided by a pytest fixture. It's a callable `webob.request.RequestBase`-like request object that is pre-bound to the app that was passed (or assumed) when we called `ra.api()`, as well as the resource scope's path and the test's method declaration. (Note on `req` naming: `request` is already a builtin fixture name in pytest.)

To override request parameters, you can pass them into the test decorator:

```
@user.get(content_type='text/plain')
def get_text(req):
    req()
```

Or pass request parameters directly into `req()`,. You can also pass which status codes are considered a success (default is 2xx/3xx status codes, this is standard WebTest):

```
@users.get
def get_text(req):
    req(content_type='text/plain', status=(200, 404))
```

You can also override the resource scope's factory declaration (or the default RAML example factories) on individual tests. The factory generates request data which is encoded as a JSON body:

```
@api.resource('/users', factory=users_factory)
def users_resource(users):

    @users.post(factory=users_post_factory)
    def post_with_my_factory(req):
        assert req.factory == users_post_factory

        # factory is used to generate data (an object)
        assert req.data == users_post_factory()

        # data is encoded to body as a JSON bytestring
        import json
        assert req.body == bytes(json.dumps(req.data, cls=req.JSONEncoder))
        req()
```

By default, responses are validated against the RAML definition, checking the body and headers are compliant. You can disable this:

```
@user.get
def get_with_no_validation(req):
    req(validate=False)
    # or only validate body (valid values are "body", "headers")
    req(validate=['body'])
```

Because tests are collected by pytest, you can pass any other fixtures you want to the test function:

```
@pytest.fixture
def two_hundred():
    return 200

@user.get
def get_with_fixture(req, two_hundred):
    response = req()
    assert response.status_code == two_hundred
```

Factories

Ra will automatically use example values from the RAML definition for request bodies when they're defined. However, this can be overridden by passing factory arguments to scopes or tests.

Example factories

When Ra reads the RAML definition, it will grab the example request body values of type "application/json" for each resource and create a factory function that returns that example, optionally overriding keys in the example.

Example factories are accessible through the `ra.API` object. For the following example RAML:

```
/users:
    post:
        body:
            application/json:
                example: { "username": "simon" }
    /{username}:
        put:
            body:
                application/json:
                    example: { "username": "gunther" }
        /profile:
            post:
                body:
                    application/json:
                        example: { "address": "ice kingdom" }
```

Ra will create example these example factories:

```
api = ra.api(ramlfile, testapp)

post_users_factory = api.examples.get_factory("POST /users")
post_users_factory() == { "username": "simon" }

# the POST body example is also keyed under the singular resource name:
user_factory = api.examples.get_factory("user")
post_users_factory is user_factory

# you can build examples directly:
put_user_example = api.examples.build("PUT /users/{username}")

# and you can override keys in the example:
profile_factory = api.examples.get_factory("POST /users/{username}/profile")
```

```
profile1 = profile_factory(address="ooo")

# singular resource names drop dynamic segments:
profile2 = api.examples.build("user.profile", address="ooo")
profile1 == profile2 == { "address": "ooo" }
```

Example factories are the default factories used to create request bodies whenever they're defined. If a body is defined for a method but no example is defined, the example factory for the resource's POST method will be used as a fallback. Thus, you can get away with defining examples only on POST bodies for each collection resource (the same example that is keyed to the dotted resource name).

Overriding Factories

You can pass a custom factory into resource scope or test declaration decorators to override the example factory (or provide a factory when no example is defined). You can also access the factory on the `req` request object to override attributes.

```
@api.resource("/users", factory=my_user_factory)
def users_resource(users):

    # tests in this scope will use my_user_factory instead
    # of an example factory

    @users.post(factory=my_other_factory)
    def post(req):
        # this test will use my_other_factory instead

        # if you want to override attributes:
        req.data = req.factory(username="james")

    # ...
```

Test Fixtures

You can use pytest's autouse fixtures to define any test setup or teardown. Here's an example of cleaning the database before and after testing:

```
@pytest.fixture(autouse=True)
def clear_database(request, req, app):
    # Remember:
    # - ``req`` is the pre-bound Ra request for the current test
    # - ``request`` is a built-in pytest fixture that holds info about
    #   the current test context
    # - ``app`` is the webtest-wrapped application
    import example_app
    example_app.clear_db()
    example_app.start_transaction()

    # login for authentication
    app.post('/login', { 'login': 'system', 'password': '123456' })

    if req.match(only='/users', exclude='POST'):
        # for tests on the /users resource, unless it's a POST, we should
        # create the user first
        example_app.create_user_in_db("marcy")

    @request.addfinalizer
    def fin():
        example_app.rollback_transaction()
        app.reset() # clear cookies; logout

def user_factory():
    user = {
        'username': 'marcy',
        #  ...
    }
    return user

@api.resource('/users')
def users_resource(users):
    @users.get
    def get(req):
        response = req()
        assert 'marcy' in response

    @users.post(factory=user_factory)
    def post(req):
        response = req()
        assert 'marcy' in response
```

In this example, we use a pytest fixture to clear the database in between each test, as well as start a database transaction. A finalizer is added, which runs after the test, to rollback the transaction. It also logs in as the default system user (for authentication purposes) and, conditionally, creates a user.

`req.match()` is used to test the current request's method and path against patterns (see below about `only and exclude`). It matches for tests on the `/users` resource, for all HTTP methods except `POST`. This means the user exists for the GET request, but not for the POST request where it will be created by the request (to avoid conflict).

This ensures that tests are isolated from one another, but still have access to the objects they expect (based on the examples in the RAML, usually).

Note that `req.match()` also applies to tests generated by autotest (see *Autotest <./autotest.html>*_), making it a good way to customize behaviour around these tests.

Resource-specific setup

Resource scopes have their own fixture scope, so you can provide test setup that applies to tests within a resource easily:

```
@api.resource('/users')
def users_resource(users):

    @pytest.fixture(autouse=True)
    def users_setup():
        # ...

    @users.get
    def get(): req()

    # ...
```

only and exclude

`req.match()` can be called with the parameters `only` and `exclude`. These parameters each take a string pattern or an array of string patterns describing routes by HTTP method, path, or both. Path patterns can match routes with glob syntax.

If `only` is omitted, all tests will match unless they are excluded using `exclude`. Patterns in `exclude` always take precedence.

Some examples:

```
@pytest.fixture(autouse=True)
def setup(req):
    if req.match(only=['GET /users', 'POST /users']):
        # matches only tests for "GET /users" and "POST /users" exactly
        pass

    if req.match(exclude='POST')
        # matches all tests unless the method is POST
        pass

    if req.match(only='/users', exclude=['POST', 'PUT'])
        # matches tests for path "/users" unless method is POST or PUT
        pass

    if req.match('/users/*')
        # ``only`` is the first positional argument, so you can call it
        # this way; matches any path starting with "/users/"
        # (doesn't match "/users")
        pass

    if req.match('*/edit')
        # matches any path ending with "/edit", e.g. "/posts/{id}/edit"
        pass
```

Autotest

`api.autotest()` will generate a basic test for each method defined in the RAML file.

The test is a basic test:

```
def test(req):
    req()
```

This uses the default factories (using the example values in the RAML for request bodies) and default URI parameters (example values in `uriParameters` definitions in the RAML).

By setting up fixtures to pre-generate any objects needing to be referenced by the examples, and defining your RAML examples carefully, you can test a whole API using autotest. The basic tests check for an acceptable status code and validate the response body and headers against the RAML definition (if any).

`api.autotest()` accepts the following (optional) settings: - `postrequest_sleep` (default: 0.5): seconds to wait in between each method

If you want to pass headers, use alternate content types, custom factories, etc., write those tests by hand (see Writing Tests).

Selecting Tests

On the commandline, you can select a subset of tests to run by referring to their pytest node IDs.

For the following example resource:

```python
# in ./tests/test_api.py

@api.resource('/users')
def users_resource(users):

    @users.get
    def get(req): req()

    @users.post
    def post(req): req()
```

You can select only the tests in this resource like this:

```
$ py.test tests/test_api.py::/users
```

To select a single test, append the function name: `tests/test_api.py::/users::get`

For autotests, insert "autotests" before the resource name: `tests/test_api.py::autotests::/users::get`

See the pytest docs for details.

Full Example

A full example follows.

The RAML:

```yaml
#%RAML 0.8
# ./example.raml
---
title: example API
mediaType: application/json
protocols: [HTTP]

/users:
    get:
        description: Get users
    post:
        description: Create a new user
        body:
            application/json:
                schema: !include schemas/user.json
                example: { "username": "marcy" }

    /{username}:
        get:
            description: Get user by username

# ...
```

```python
# in tests/test_api.py:
import ra
import pytest

api = ra.api('example.raml', app='config:test.ini')

@pytest.fixture(autouse=True)
def clear_database(request, req, app, examples):
    # Remember:
    # - ``req`` is the pre-bound Ra request for the current test
    # - ``request`` is a built-in pytest fixture that holds info about
    #   the current test context
    # - ``app`` is the webtest-wrapped application
    # - ``examples`` is a fixture providing the examples factory manager
    #   for generating data based on RAML ``example`` properties.
    import example_app
    example_app.clear_db()
    example_app.start_transaction()

    # login for authentication
    app.post('/login', { 'login': 'system', 'password': '123456' })

    if req.match(exclude='POST /users'):
        # Before any test, except for the one that creates a user,
        # we should create the user first.
        #
        # Passing 'user' to ``examples.build()``
        # means to use the example defined for ``POST /users``
        marcy = examples.build('user') # returns a dict
        example_app.create_user_in_db(marcy)

    @request.addfinalizer
    def fin():
        example_app.rollback_transaction()
        app.reset() # clear cookies; logout


# defining a resource scope:

@api.resource('/users')
def users_resource(users):

    # scope-local pytest fixtures
    #
    # a resource scope acts just like a regular module scope
    # with respect to pytest fixtures:

    @pytest.fixture
    def two_hundred():
        return 200

    # defining tests for methods in this resource:

    @users.get
    def get(req, two_hundred):
        # ``req`` is a callable request object that is pre-bound to the app
        # that was passed into ``ra.api`` as well as the URI derived from
        # the resource (test scope) and method (test) decorators.
        #
        # This example uses the other scope-local fixture defined above.
        response = req()
        assert response.status_code == two_hundred

    @users.post
    def post_using_example(req):
        # By default, when JSON data needs to be sent in the request body,
        # Ra will look for an ``example`` property in the RAML definition
        # of the resource method's body and use that.
        #
        # As in WebTest request methods, you can specify the expected
        # status code(s), which will be test the response status.
        req(status=(201, 409))

    # defining a custom user factory; underscored functions are not
    # considered tests (but better to import factories from another module)
    def _user_factory():
        import string
        import random
        name = ''.join(random.choice(string.ascii_lowercase) for _ in range(10))
        email = "{}@example.com".format(name)
        return dict(username=name, email=email, password=name)

    # using the factory:

    @users.post(factory=_user_factory)
    def post_using_factory(req):
        response = req()
        username = req.data['username']
        assert username in response

    # defining a sub-resource:

    @users.resource('/{username}')
    def user_resource(user):

        # this resource will be requested at /users/{username}
        #
        # By default, Ra will look at the ``example`` property for
        # URI parameters as defined in the RAML, and fill the URI
        # template with that.

        @user.get
        def get(req):
            # This is equivalent to the autotest for a resource
            # and method:
            req()

api.autotest() # autotests will be generated
```

Changelog

- #7: Added autotest "postrequest_sleep" setting
- #6: Fixed autotests with python 3.3/3.4

- : Initial release