JAVA RSPEC TESTING

RSPEC IN THE JAVA SPACE

GOAL

The overall goal when testing Java Resource Oriented Web Services with RSPEC is to ensure we have full client coverage of the application.

We are trying to ensure that:

1. Each revision bump is backwards compatible with the Service API
2. All clients that are being used will not break when additive changes are made
3. The API Contract is met as per the originally documented API

When testing, we want to ensure that for each request that can be made the actual documented outcome is met.

USE CASE

Service Endpoint Contract V1.0.0

THE SERVICE ENDPOINT CONTRACT V1.0.0

Endpoint URI	/v1/account/{accountId}/page	
HTTP Method	GET	
Response Codes	200	Indicates a successful request. A list of all connected page identifiers should be returned as part of the response body
	403	Indicates the account identifier does not have permission to access the service
	404	Indicates that the account identifier is unknown
Representation	{ “page_ids” : [“/v1/account/{accountId}/page/54234”, “/v1/account/{accountId}/page/64226”, “/v1/account/{accountId}/page/35623”, “/v1/account/{accountId}/page/67423”, “/v1/account/{accountId}/page/86556”] }	

USE CASE

When testing this with RSPEC, we want to ensure that when we send a valid request, we get the correct response codes and response body.

The above specification should result in 3 main RSPEC tests:

- To prove we get a **200** response code and response body *when successful*
- To prove we get a **403** response code when *the account is not authorized*
- To prove we get a **404** response code when *the account is unknown*

WRITING THE RSPEC TEST

Creating the tests for v1.0.0 requires that we create a new Ruby file to perform the client side connectivity and verification of the response.

Let's start with the Ruby file connected_page_service_spec.rb

connected_page_service_spec.rb

```
require 'rspec'
require 'rest_client'
require 'json'

describe 'connected_page_service#v1_0_0' do

    describe 'GET on /v1/account/{accountId}/page' do
        it 'should return a 200 status code and a list of identifiers' do

        end

        it 'should fail with a 403 unauthorized' do

        end

        it 'should fail with a 404 unknown account' do

        end
    end
end
```

WRITING THE RSPEC TEST

In the above code there are 3 main blocks:

- The import of the tools we will be using, rspec, rest_client and json.
- The main block of code encapsulating the tests
- The individual test cases

WRITING THE RSPEC TEST

In the above RSPEC tests, we are using rest client to perform the connectivity to the Service endpoint. It can be initialized with the following syntax:

```
@site = RestClient::Resource.new(
    'http://localhost:8080',
    :user => 'username',
    :password => 'password')
```

WRITING THE RSPEC TEST

To perform the actual service call, let's fill in the first test:

```
it 'should return a 200 status code and a list of identifiers' do
    response = @site['/v1/account/123456/page'].get :accept => 'application/json'

    response.code.should == 200
    response_body = JSON.parse(response.body)

    response_body['page_ids'][0].should == '/v1/account/123456/page/54234'
    response_body['page_ids'][1].should == '/v1/account/123456/page/64226'
end
```

WRITING THE RSPEC TEST

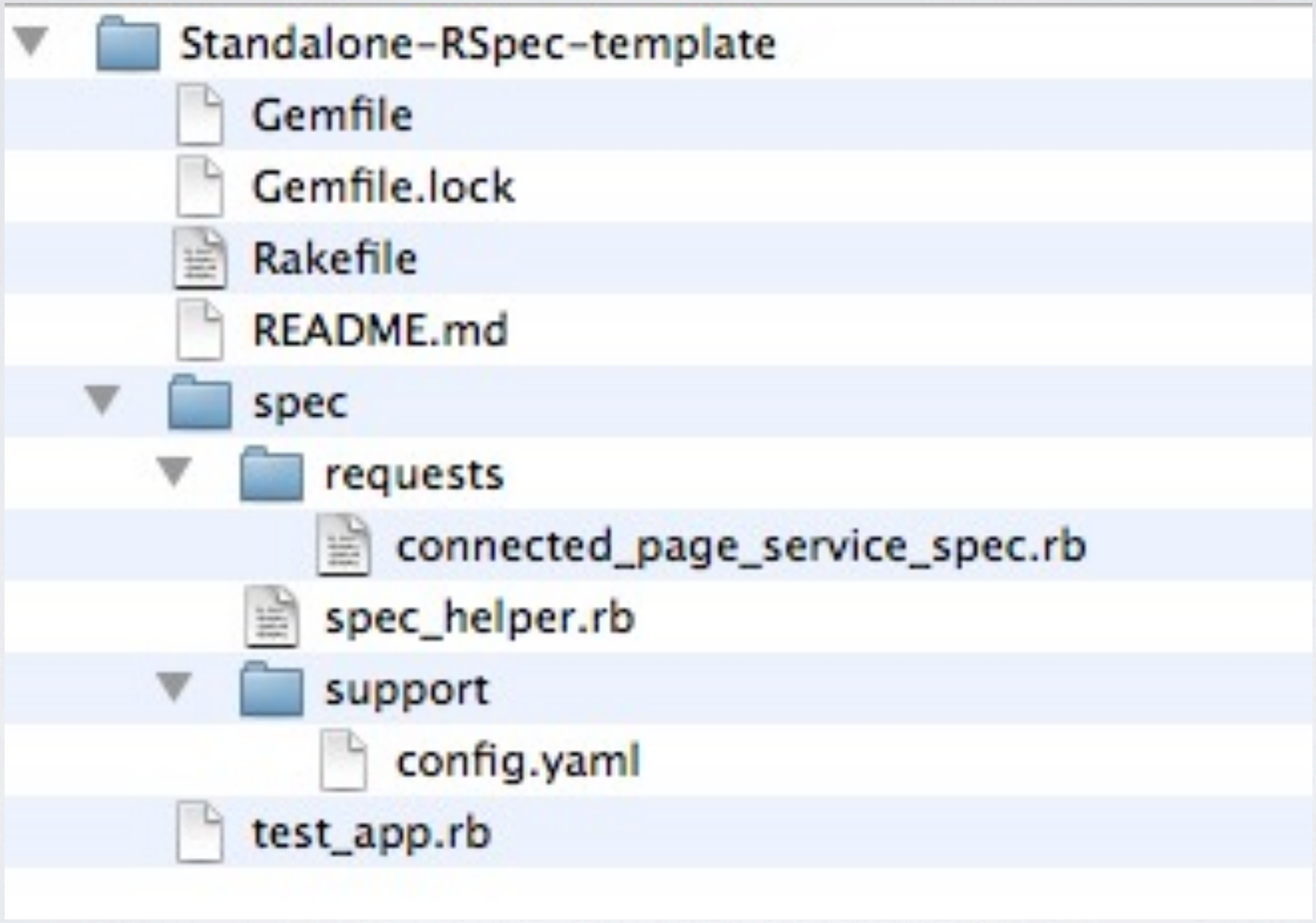
With the rest client initialized, all that we need to do is specify the URI we need to access with the accept header configured. Once this executes we need to check that we get the correct 200 response code and a response body with the expected values.

To run the test execute: *rspec connected_page_service_spec.rb*

A successful test will display a period (.) for each successfully run test and a summary of the time taken and how many tests were run.

DEVELOPING AN RSPEC SUITE

DIRECTORY STRUCTURE



DIRECTORY STRUCTURE

All RSPEC tests go to "/spec/requests" and can even be versioned, for example:

- /spec/requests/v1_0_0
- /spec/requests/v1_0_1
- /spec/requests/v2_0_0

DIRECTORY STRUCTURE

The "/spec/support" directory can contain configuration files that are may be used by the application.

The "/spec/requests/spec_helper.rb" file allows you to configure global variables that can be shared across all the RSPEC tests that you have in place. By doing this, you only need to "require" the ruby file to get access to the variables defined in the file

SPEC_HELPER.RB

```
require ‘rspec’
require ‘rack/test’
require ‘yaml’

# Load the sinatra app if required
require_relative ‘../test_app’

set :environment, :test

APP_CONFIG = YAML.load_file(File.expand_path(“#{File.dirname(__FILE__)}/support/
config.yaml”))
#...extract values from the config file

RSpec.configure do |conf|
  conf.include Rack::Test::Method
end

def app
  Sinatra::Application
end
```

GEMFILE

If we have a look at the Gemfile, you can see we are using a very basic set of gems:

```
source 'http://rubygems.org'

gem 'rake'
gem 'sinatra'
gem 'rspec','2.9.0', require: 'spec'
gem 'rack-test'

gem 'logging'
gem 'json'
gem 'rest-client'
```

CONFIGURING A NEW TEST

Create a new configuration file at "support/config.yaml"

```
local:
  server_url: http://local.server:8081/webapp/rest

service_endpoints:
  campaign_svc:
    v1:
      get_campaign_uri: /account/{acctId}/campaign/{campaignId}
```

CONFIGURING A NEW TEST

Next set up the spec_helper.rb file to set up any global variables:

```
require ‘rspec’
require ‘rack/test’
require ‘yaml’

# Load the sinatra app if required
require_relative ‘../test_app’

set :environment, :test

APP_CONFIG = YAML.load_file(File.expand_path(“#{File.dirname(__FILE__)}/support/config.yaml”))
#...extract values from the config file

$server_url = APP_CONFIG[“local”][“server_url”]
$service_config = APP_CONFIG[“service_endpoints”]

RSpec.configure do |conf|
  conf.include Rack::Test::Method
end

def app
  Sinatra::Application
end
```

CONFIGURING A NEW TEST

Next, we can start with out with our RSpec test, campaign_service_spec.rb:

```
require ‘rspec’
require ‘yaml’
require ‘rest_client’
require ‘json’

require File.expand_path(“{File.dirname(__FILE__)}/../../spec_helper”)

describe ‘campaign_service_spec’ do

  # Before each test we perform, let’s ensure we have a fresh URI
  # and set up our RestClient
  before(:each) do
    # Get the endpoint uri for our service
    @service_uri = $service_config[“campaign_svc”][“v1”][“get_campaign_uri”]
    @client = RestClient::Resource.new($server_url,
                :user => ‘roving’, :password => roving)
  end


  # Create our first test
  it “should successfully get a campaign” do
    #substitute the {acctId} and {campaignId} in the URI
    acctId = ‘123456’
    campaignId = ‘987654’
    endpoint = @service_uri.sub(/{acctId}/, acctId)
    endpoint = endpoint.sub(/{campaignId}/, campaignId)

    # execute the call the service
    response = @client.get :accept => ‘application/json’

    # Now verify the results
    response.code.should == 200

    # Read in the JSON and verify the values
    campaign = JSON.parse(response.body.read)
    campaign[“acctId”].should == acctId
    campaign[“id”].should == campaignId
  end
end
```

CONFIGURING A NEW TEST

Provided the service returns the correct results, running the suite with the rspec command should load the configuration file, allocate global variables and then run the campaign_service_spec.rb file

WEB SERVICE MOCKING

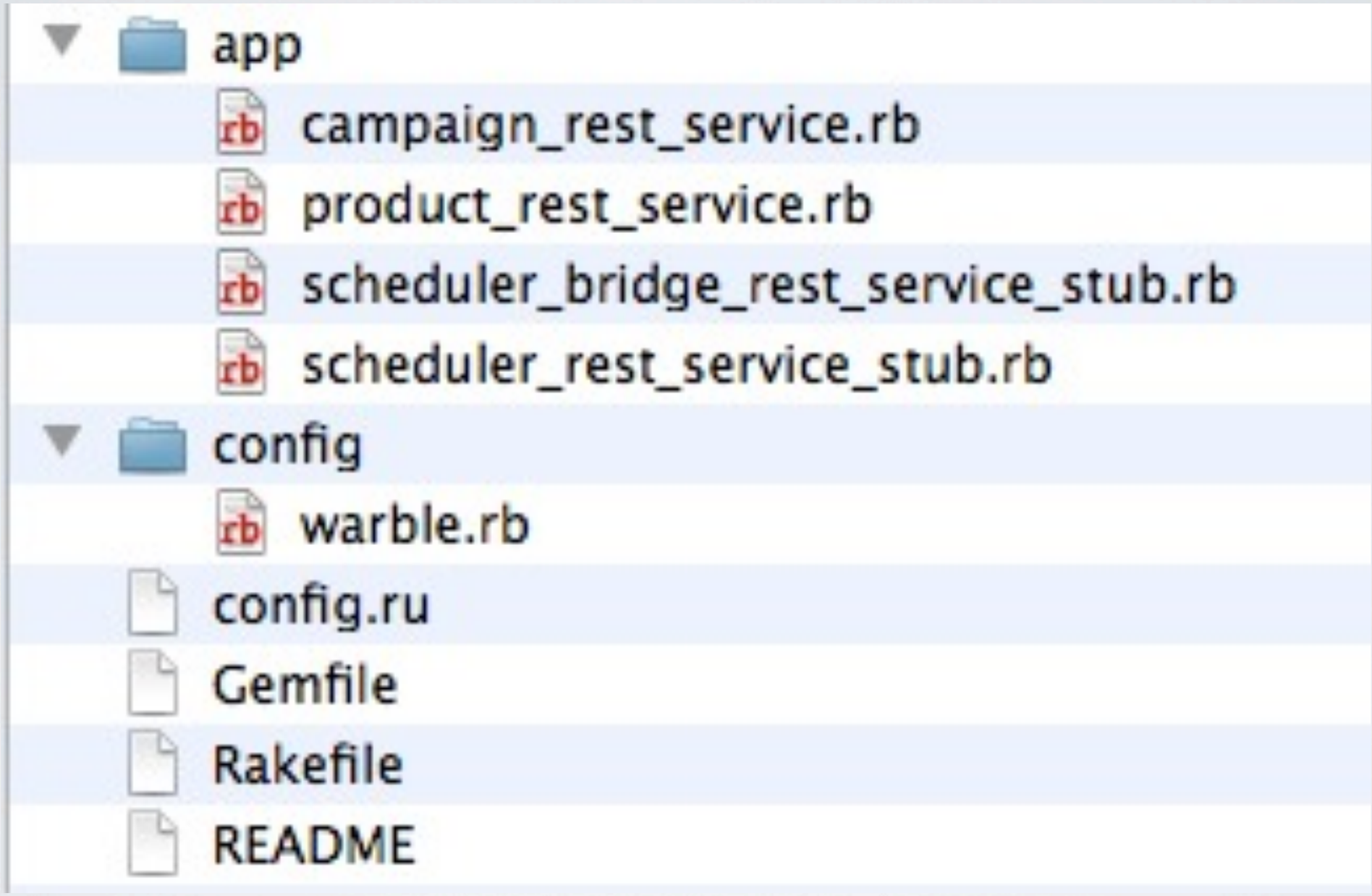
MOCKING EXISTING WEB SERVICES

What do you do when you need to access services without affecting existing data?

Currently our approach is to mock the services consumed using Sinatra (http://www.sinatrarb.com/).

It's quick and easy to get up and running and easy to deploy using the warbler gem.

DIRECTORY SETUP



GEMFILE SETUP

In the Gemfile, make sure you include the necessary gems to get started:

```
source ‘http://rubygems.org’

gem ‘json’
gem ‘sinatra’
gem ‘rest-client’
```

CREATING THE SERVICE MOCK

In the "app" folder create a file that represents the mocked service (/app/controllers/campaign_rest_service.rb) :

```
require 'rubygems'
require 'json'
require 'sinatra'

get '/account/:accountId/campaign/:campaignId' do
  account_id = params[:accountId]
  campaign_id = params[:campaignId]

  # do something with the account_id and campaign_id

  content_type :json
  {:id => params[:accountId], :name => 'Object Name', :campaignId => params[:campaignId]}.to_json

end
```

CREATING THE SERVICE MOCK

In the example, we have bound the endpoint '/account/:accountId/campaign/:campaignId' to the server, configured the response content type header and with the values passed in create a basic JSON representation to send back to the client.

The representation can be a hard coded value or, depending on your Ruby skill level, a response composed of values from a database or file.

CREATING THE SERVICE MOCK

Next you need to configure the Sinatra application. In the config.ru file, you will need to declare which files you want to use which have services you want to expose. You then need to declare the Application that is to be run, in this case a Sinatra application:

```
require ‘rubygems’
require ‘sinatra’

# import the files that contain the services you want to
expose
require File.dirname(__FILE__) + ‘/app/controllers/
campaign_controller’

run Sinatra::Application
```

RUNNING A MOCK SERVICE

You can test the individual files using:

ruby app/controller/campaign_controller.rb

This will start up the application server and start serving up the endpoint defined in the Ruby file, otherwise you can create a WAR file that can be deployed into a Java Servlet Container using warble.