Sinatra

Testing Sinatra with Rack::Test

All examples in the following sections assume that `Test::Unit` is being used in an attempt to be as general as possible. See the Test Framework Examples for information on using the test helpers in other testing environments. To use `Rack::Test` library used when you require `rack/test`, you'll need to install the `rack-test` gem.

gem install rack-test

Example App: `hello_world.rb`

The following example app is used to illustrate testing features. This is assumed to be in a file named `hello_world.rb`:

```
require 'sinatra'

get '/' do
  "Hello World #{params[:name]}".strip
end
```

Using The `Rack::Test::Methods` Mixin

The `Rack::Test::Methods` module includes a variety of helper methods for simulating requests against an application and asserting expectations about the response. It's typically included directly within the test context and makes a few helper methods and attributes available.

The following is a simple example that ensures the hello world app functions properly:

```
ENV['RACK_ENV'] = 'test'

require 'hello_world'
require 'test/unit'
require 'rack/test'

class HelloWorldTest < Test::Unit::TestCase
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end
  def test_it_says_hello_world
    get '/'
    assert last_response.ok?
    assert_equal 'Hello World', last_response.body
  end

  def test_it_says_hello_to_a_person
    get '/', :name => 'Simon'
    assert last_response.body.include?('Simon')
  end
end
```

Using `Rack::Test` without the Mixin

For a variety of reasons you may not want to include `Rack::Test::Methods` into your own classes. `Rack::Test` supports this style of testing as well, here is the above example without using Mixin.

```
ENV['RACK_ENV'] = 'test'

require 'hello_world'
require 'test/unit'
require 'rack/test'

class HelloWorldTest < Test::Unit::TestCase

  def test_it_says_hello_world
    browser = Rack::Test::Session.new(Rack::MockSession.new(Sinatra::Application))
    browser.get '/'
    assert browser.last_response.ok?
    assert_equal 'Hello World', browser.last_response.body
  end

  def test_it_says_hello_to_a_person
    browser = Rack::Test::Session.new(Rack::MockSession.new(Sinatra::Application))
    browser.get '/', :name => 'Simon'
    assert browser.last_response.body.include?('Simon')
  end
end
```

Rack::Test's Mock Request Methods

The `get`, `put`, `post`, `delete`, and `head` methods simulate the respective type of request on the application. Tests typically begin with a call to one of these methods followed by one or more assertions against the resulting response.

All mock request methods have the same argument signature:

```
get '/path', params={}, rack_env={}
```

- `/path` is the request path and may optionally include a query string.
- `params` is a Hash of query/post parameters, a String request body, or `nil`.

- `rack_env` is a Hash of Rack environment values. This can be used to set request headers and other request related information, such as session data. See the Rack SPEC for more information on possible key/values.

Asserting Expectations About The Response

Once a request method has been invoked, the following attributes are available for making assertions:

- `app` – The Sinatra application class that handled the mock request.
- `last_request` – The `Rack::MockRequest` used to generate the request.
- `last_response` – A `Rack::MockResponse` instance with information on the response generated by the application.

Assertions are typically made against the `last_response` object. Consider the following examples:

```
def test_it_says_hello_world
  get '/'
  assert last_response.ok?
  assert_equal 'Hello World'.length.to_s, last_response.headers['Content-Length']
  assert_equal 'Hello World', last_response.body
end
```

Optional Test Setup

The `Rack::Test` mock request methods send requests to the return value of a method named `app`.

If you're testing a modular application that has multiple `Sinatra::Base` subclasses, simply set the `app` method to return your particular class.

```
  def app
    MySinatraApp
  end
```

If you're using a classic style Sinatra application, then you need to return an instance of `Sinatra::Application`.

```
  def app
    Sinatra::Application
  end
```

Making `Rack::Test` available to all test cases

If you'd like the `Rack::Test` methods to be available to all test cases without having to include it each time, you can include the `Rack::Test` module in the `Test::Unit::TestCase` class:

```
require 'test/unit'
require 'rack/test'

class Test::Unit::TestCase
  include Rack::Test::Methods
end
```

Now all `TestCase` subclasses will automatically have `Rack::Test` available to them.

Test Framework Examples

As of version `0.9.1`, Sinatra no longer provides testing framework-specific helpers. Those found in `sinatra/test/*.rb` are deprecated and has been removed in Sinatra `1.0`.

RSpec

Sinatra can be tested under plain RSpec. The `Rack::Test` module should be included within the `describe` block:

```
ENV['RACK_ENV'] = 'test'

require 'hello_world'  # <-- your sinatra app
require 'rspec'
require 'rack/test'

describe 'The HelloWorld App' do
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  it "says hello" do
    get '/'
    expect(last_response).to be_ok
    expect(last_response.body).to eq('Hello World')
  end
end
```

Make `Rack::Test` available to all spec contexts by including it via `RSpec`:

```
require 'rspec'
require 'rack/test'

RSpec.configure do |conf|
  conf.include Rack::Test::Methods
end
```

Bacon

Testing with Bacon is similar to `test/unit` and RSpec:

```
ENV['RACK_ENV'] = 'test'
```

```
require 'hello_world'  # <-- your sinatra app
require 'bacon'
require 'rack/test'

describe 'The HelloWorld App' do
  extend Rack::Test::Methods

  def app
    Sinatra::Application
  end

  it "says hello" do
    get '/'
    last_response.should.be.ok
    last_response.body.should.equal 'Hello World'
  end
end
```

Make `Rack::Test` available to all spec contexts by including it in `Bacon::Context`:

```
class Bacon::Context
  include Rack::Test::Methods
end
```

Test::Spec

The `Rack::Test` module should be included within the context of the `describe` block:

```
ENV['RACK_ENV'] = 'test'

require 'hello_world'  # <-- your sinatra app
require 'test/spec'
require 'rack/test'

describe 'The HelloWorld App' do
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  it "says hello" do
    get '/'
    last_response.should.be.ok
    last_response.body.should.equal 'Hello World'
  end
end
```

Make `Rack::Test` available to all spec contexts by including it in `Test::Unit::TestCase`:

```
require 'test/spec'
require 'rack/test'

Test::Unit::TestCase.send :include, Rack::Test::Methods
```

Webrat

From `Webrat`’s wiki where you’ll find more examples.

```
ENV['RACK_ENV'] = 'test'

require 'hello_world'  # <-- your sinatra app
require 'rack/test'
require 'test/unit'

Webrat.configure do |config|
  config.mode = :rack
end

class HelloWorldTest < Test::Unit::TestCase
  include Rack::Test::Methods
  include Webrat::Methods
  include Webrat::Matchers

  def app
    Sinatra::Application.new
  end

  def test_it_works
    visit '/'
    assert_contain('Hello World')
  end
end
```

Capybara

`Capybara` will use `Rack::Test` by default. You can use another driver, like `Selenium`, by setting the default_driver.

```
ENV['RACK_ENV'] = 'test'

require 'hello_world'  # <-- your sinatra app
require 'capybara'
require 'capybara/dsl'
require 'test/unit'

class HelloWorldTest < Test::Unit::TestCase
  include Capybara::DSL
  # Capybara.default_driver = :selenium # <-- use Selenium driver

  def setup
    Capybara.app = Sinatra::Application.new
  end

  def test_it_works
    visit '/'
    assert page.has_content?('Hello World')
  end
end
```

See Also

See the source for Rack::Test for more information on `get`, `post`, `put`, `delete` and friends.