Testing

Using Rack::Test

Testing is an integral part of software development. In this section we will look into testing the Sinatra application itself. For unit testing your models or other classes, please consult the documentation of frameworks used (including your test framework itself). Sinatra itself uses Contest for testing, but feel free to use any framework you like.

Bryan Helmkamp's Rack::Test offers tools for mocking Rack request, sending those to your application and inspecting the response all wrapped in a small DSL.

Firing Requests

You import the DSL by including Rack::Test::Methods into your test framework. It is even usable without a framework and for other tasks besides testing.

Imagine you have an application like this:

```
# myapp.rb
require 'sinatra'

get '/' do
  "Welcome to my page!"
end

post '/' do
  "Hello #{params[:name]}!"
end
```

You have to define an app method pointing to your application class (which is Sinatra::Application per default):

```
begin
  # try to use require_relative first
  # this only works for 1.9
  require_relative 'my-app.rb'
rescue NameError
  # oops, must be using 1.8
  # no problem, this will load it then
  require File.expand_path('my-app.rb', __FILE__)
end

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

```
class MyAppTest < Test::Unit::TestCase
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  def test_my_default
    get '/'
    assert last_response.ok?
    assert_equal 'Welcome to my page!', last_response.body
  end

  def test_with_params
    post '/', :name => 'Frank'
    assert_equal 'Hello Frank!', last_response.body
  end
end
```

Modifying env

While parameters can be send via the second argument of a get/post/put/delete call (see the post example above), the env hash (and thereby the HTTP headers) can be modified with a third argument:

```
get '/foo', {}, 'HTTP_USER_AGENT' => 'Songbird 1.0'
```

This also allows passing internal env settings:

```
get '/foo', {}, 'rack.session' => { 'user_id' => 20 }
```

Cookies

For example, add the following to your app to test against:

```
"Hello #{request.cookies['foo']}!"
```

Use set_cookie for setting and removing cookies, and the access them in your response:

```
response.set_cookie 'foo=bar'
get '/'
assert_equal 'Hello bar!', last_response.body
```

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.