Sinatra

Writing Extensions

Sinatra includes an API for extension authors to help ensure that consistent behavior is provided for application developers.

Background

Some knowledge of Sinatra's internal design is required to write good extensions. This section provides a high level overview of the classes and idioms at the core of Sinatra's design.

Sinatra has two distinct modes of use that extensions should be aware of:

1. The "Classic" style, where applications are defined on main / the top-level - most of the examples and documentation target this usage. Classic applications are often single-file, standalone apps that are run directly from the command line or with a minimal rackup file. When an extension is required in a classic application, the expectation is that all extension functionality should be present without additional setup on the application developers part (like included/extending modules).

2. The "Modular" style, where `Sinatra::Base` is subclassed explicitly and the application is defined within the subclass's scope. These applications are often bundled as libraries and used as components within a larger Rack-based system. Modular applications must include any desired extensions explicitly by calling `register ExtensionModule` within the application's class scope.

Most extensions are relevant to both styles but care must be taken by extension authors to ensure that extensions do the right thing under each style. The extension API (`Sinatra.register` and `Sinatra.helpers`) is provided to help extension authors with this task.

Important: The following notes on `Sinatra::Base` and `Sinatra::Application` are provided for background only - extension authors should not need to modify these classes directly.

`Sinatra::Base`

The `Sinatra::Base` class provides the context for all evaluation in a Sinatra application. The top-level DSLish stuff exists in class scope while request-level stuff exists at instance scope.

Applications are *defined* within the class scope of a `Sinatra::Base` subclass. The "DSL" (e.g., `get`, `post`, `before`, `configure`, `set`, etc.) is simply a set of class methods defined on `Sinatra::Base`.

Extending the DSL is achieved by adding class methods to `Sinatra::Base` or one of its subclasses. However, Base classes should not be extended with `extend`; the `Sinatra.register` method (described below) is provided for this task.

Requests are evaluated within a new `Sinatra::Base` instance - routes, before filters, views, helpers, and error pages all share this same context. The default set of request-level helper methods (e.g, `erb`, `haml`, `halt`, `content_type`, etc.) are simple instance methods defined on `Sinatra::Base` or within modules that are included in `Sinatra::Base`. Providing new functionality at the request level is achieved by adding instance methods to `Sinatra::Base`.

As with DSL extensions, helper modules should not be added directly to `Sinatra::Base` by extension authors with `include`; the `Sinatra.helpers` method (described below) is provided for this task.

`Sinatra::Application`

The `Sinatra::Application` class provides the default execution context for *classic style applications*. It is a simple subclass of `Sinatra::Base` that provides default option values and other behavior tailored for top-level apps. When a classic style application is run, all `Sinatra::Application` public class methods are exported to the top-level.

Rules for Extensions

1. Never modify `Sinatra::Base` directly. You should not include or extend, change option values, or modify its behavior elsewise. Modular style applications will include your extension in their subclass explicitly using the `register` method.

2. Never `require 'sinatra'` in your extension. You should only ever need to `require 'sinatra/base'`. The reason for this is that `require 'sinatra'` is what triggers the classic style - extensions should never trigger the classic style.

3. Use the APIs described below where possible. You should not need to include or extend modules directly or define methods directly on a core Sinatra class. The specialized methods below handle all of that for you.

4. Extensions *should* be defined in separate modules under the `Sinatra` module. For example, an extension that added basic authentication primitives might be named `Sinatra::BasicAuth`.

Extending The Request Context with `Sinatra.helpers`

The most common type of extension is one that adds methods for use in routes, views, and helper methods.

For example, suppose you wanted to write an extension that added an `h` method that escaped reserved HTML characters (such as those found in other popular Ruby web frameworks).

```
require 'sinatra/base'

module Sinatra
  module HTMLEscapeHelper
```

```
    def h(text)
      Rack::Utils.escape_html(text)
    end
  end

  helpers HTMLEscapeHelper
end
```

The call to `Sinatra.helpers` includes the module in `Sinatra::Application`, making all methods defined in the module available to classic style applications. Using this extension in classic style apps is a simple as requiring the extension and using the new method:

```
require 'sinatra'
require 'sinatra/htmlescape'

get "/hello" do
  h "1 < 2"     # => "1 &lt; 2"
end
```

`Sinatra::Base` subclasses, on the other hand, must require *and* include the module explicitly using the `helpers` method:

```
require 'sinatra/base'
require 'sinatra/htmlescape'

class HelloApp < Sinatra::Base
  helpers Sinatra::HTMLEscapeHelper

  get "/hello" do
    h "1 < 2"
  end
end
```

Extending The DSL (class) Context with `Sinatra.register`

Extensions can also extend Sinatra's class level DSL using the `Sinatra.register` method. Here's an extension that adds a `block_links_from` macro that checks the referer on each request for a app provided pattern and sends back a `403 Forbidden` response when a match is detected:

```
require 'sinatra/base'

module Sinatra
  module LinkBlocker
    def block_links_from(host)
      before {
        halt 403, "Go Away!" if request.referer.match(host)
      }
    end
  end

  register LinkBlocker
end
```

`Sinatra.register` adds all public methods in the module(s) given as class methods on `Sinatra::Application`. It also handles exporting public methods to the top-level when classic style apps are executed.

A classic style application would use this extension as follows:

```
require 'sinatra'
require 'sinatra/linkblocker'

block_links_from 'digg.com'

get '/' do
  "Hello World"
end
```

Modular style applications must register the extension explicitly in their `Sinatra::Base` subclasses:

```
require 'sinatra/base'
require 'sinatra/linkblocker'

class Hello < Sinatra::Base
  register Sinatra::LinkBlocker

  block_links_from 'digg.com'

  get '/' do
    "Hello World"
  end
end
```

Setting Options and Other Extension Setup

Extensions can define options, routes, before filters, and error handlers by defining a `registered` method on the extension module. The `Module.registered` method is called immediately after the extension module is added to the `Sinatra::Base` subclass and is passed the class that the module was registered with.

The following example creates a very simple extension that adds basic session auth support. Options are added for the username and password, routes are defined for logging in, and helper methods are provided for determining whether a user has been authorized:

```
require 'sinatra/base'

module Sinatra
  module SessionAuth

    module Helpers
      def authorized?
        session[:authorized]
      end

      def authorize!
        redirect '/login' unless authorized?
      end
```

```
      def logout!
        session[:authorized] = false
      end
    end

    def self.registered(app)
      app.helpers SessionAuth::Helpers

      app.set :username, 'frank'
      app.set :password, 'changeme'

      app.get '/login' do
        "<form method='POST' action='/login'>" +
        "<input type='text' name='user'>" +
        "<input type='text' name='pass'>" +
        "</form>"
      end

      app.post '/login' do
        if params[:user] == options.username && params[:pass] == options.password
          session[:authorized] = true
          redirect '/'
        else
          session[:authorized] = false
          redirect '/login'
        end
      end
    end
  end

  register SessionAuth
end
```

A classic application would use this extension by requiring the extension library, overriding options, and using the helpers provided:

```
require 'sinatra'
require 'sinatra/sessionauth'

set :password, 'hoboken'

get '/public' do
  if authorized?
    "Hi. I know you."
  else
    "Hi. We haven't met. <a href='/login'>Login, please.</a>"
  end
end

get '/private' do
  authorize!
  'Thanks for logging in.'
end
```

A modular application is different only in that it must register the extension module explicitly:

```
require 'sinatra/base'
```

```
require 'sinatra/sessionauth'

class MyApp < Sinatra::Base
  register Sinatra::SessionAuth

  set :password, 'hoboken'

  get '/public' do
    if authorized?
      "Hi. I know you."
    else
      "Hi. We haven't met. <a href='/login'>Login, please.</a>"
    end
  end

  get '/private' do
    authorize!
    'Thanks for logging in.'
  end
end
```

Building and Packaging Extensions

Sinatra extensions should be built as separate libraries and packaged as gems or as single files that can be included within an application's `lib` directory. The ideal process for using an extensions is installing a gem and requiring a single file.

The following is an example file layout for a typical extension packaged as a gem:

```
sinatra-fu
|-- README
|-- LICENSE
|-- Rakefile
|-- lib
|   `-- sinatra
|       `-- fu.rb
|-- test
|   `-- spec_sinatra_fu.rb
`-- sinatra-fu.gemspec
```