Sinatra::Flash

This is an implementation of show-'em-once 'flash' messages for the Sinatra Web framework. It offers the following feature set:

- Simplicity (less than 50 significant lines of code)
- Implements the documented behavior and public API of the Rails flash that many developers are used to
- Acts entirely like a hash, including iterations and merging
- Optional multiple named flash collections, each with their own message hash, so that different embedded applications can access different sets of messages
- An HTML helper for displaying flash messages with CSS styling
- Verbose documentation in YARD format
- Full RSpec tests

The primary catch for experienced Rack developers is that it *does not* function as standalone Rack middleware. (You could get to the messages inside the session if you needed to, but the message rotation occurs in a Sinatra `after` hook.) It should function just fine across multiple Sinatra apps, but if you need flash that's accessible from non-Sinatra applications, consider the Rack::Flash middleware.

Setting Up

You should know this part:

```
$> gem install sinatra-flash
```

(Or `sudo gem install` if you're the last person on Earth who isn't using RVM yet.)

If you're developing a Sinatra 'classic' application, then all you need to do is enable sessions and require the library:

```
# blah_app.rb
require 'sinatra'
require 'sinatra/flash'

enable :sessions

post '/blah' do
  # This message won't be seen until the NEXT Web request that accesses the flash collection
  flash[:blah] = "You were feeling blah at #{Time.now}."

  # Accessing the flash displays messages set from the LAST request
  "Feeling blah again? That's too bad. #{flash[:blah]}"
end
```

If you're using the Sinatra::Base style, you also need to register the extension:

```ruby
# bleh_app.rb
require 'sinatra/base'
require 'sinatra/flash'

class BlehApp < Sinatra::Base
  enable :sessions
  register Sinatra::Flash

  get '/bleh' do
    if flash[:blah]
      # The flash collection is cleared after any request that uses it
      "Have you ever felt blah? Oh yes. #{flash[:blah]} Remember?"
    else
      "Oh, now you're only feeling bleh?"
    end
  end
end
```

See the Sinatra documentation on using extensions for more detail and rationale.

styled_flash

The gem comes with a handy view helper that iterates through current flash messages and renders them in styled HTML:

```
# Using HAML, 'cause the cool kids are all doing it
%html
  %body
    =styled_flash
```

Yields (assuming three flash messages with different keys):

```
<html>
  <body>
    <div id='flash'>
      <div class='flash info'>I'm sorry, Dave. I'm afraid I can't do that.</div>
      <div class='flash warning'>This mission is too important for me to allow you
to jeopardize it.</div>
      <div class='flash fatal'>Without your space helmet, Dave, you're going to
find reaching the emergency airlock rather difficult.</div>
    </div>
  </body>
</html>
```

Styling the CSS for the #flash id, the .flash class, or any of the key-based classes is entirely up to you.

(*Side note:* This view helper was my initial reason for making this gem. I'd gotten used to pasting this little method in on every Rails project, and when I switched to Sinatra was distraught to discover that Rack::Flash couldn't do it, because the FlashHash isn't really a hash and you can't iterate it. Reinventing flash was sort of a side effect.)

Advanced Tips

Now vs. Next

The flash object acts like a hash, but it's really *two* hashes:

- The **now** hash displays messages for the current request.
- The **next** hash stores messages to be displayed in the *next* request.

When you access the **flash** helper, the *now* hash is initialized from a session value. (Named :flash by default, but see 'Scoped Flash' below.) Every method except assignment ([]=) is delegated to *now*; assignments occur on the *next* hash. At the end of the request, a Sinatra after hook sets the session value to the *next* hash, effectively rotating it to *now* for the next request that uses it.

This is usually what you want, and you don't have to worry about the details. However, you occasionally want to set a message to display during *this* request, or access values that are coming up. In these cases you can access the .now and .next hashes directly:

```
# This will be displayed during the current request
flash.now[:error] = "We're shutting down now.  Goodbye!"

# Look at messages upcoming in the next request
@weapon = Stake.new if flash.next[:warning] == "The vampire is attacking."
```

In practice, you'll probably want to set .now any time you're displaying errors with an immediate render instead of redirecting. It's hard to think of a common reason to check .next -- but it's there if you want it.

Keep, Discard, and Sweep

These convenience methods allow you to modify the standard rotation cycle, and are based directly on the Rails flash API:

```
flash.keep                # Will show all messages again
flash.keep(:groundhog)    # Will show the message on key :groundhog again
flash.discard             # Clears the next messages without showing them
flash.discard(:amnesia)   # Clears only the message on key :amnesia
flash.sweep               # Rotates the flash manually, discarding _now_ and moving
_next_ into its place
```

Sessions

The basic *concept* of flash messages relies on having an active session for your application. Sinatra::Flash is built on the assumption that Sinatra's session helper points to something that will persist beyond the current request. You are responsible for ensuring that it does. No other assumptions are made about the session -- you can use any session strategy you like.

(**Note:** Early versions of this extension attempted to detect the absence of a session and create one for you at the last moment. Thanks to rkh for pointing out that this is unreliable in Sinatra. You'll have to be a grownup now)

Scoped Flash

If one flash collection isn't exciting enough for your application stack, you can have multiple sets of flash messages scoped by a symbol. Each has its own lifecycle and will *not* be rotated by any Web request that ignores it.

```
get "/complicated" do
  flash(:one)[:hello] = "This will appear the next time flash(:one) is called"
  flash(:two).discard(:hello)  # Clear a message someone else set on flash(:two)
  "A message for you on line three: #{flash(:three)[:hello]}"
end
```

Both the **flash** and **styled_flash** helper methods accept such keys as optional parameters. If don't specify one, the default key is :flash. Whatever keys you use will become session keys as well, so take heed that you don't run into naming conflicts.

Do you need this? Probably not. The principal use case is for complex application stacks in which some messages are only relevant within specific subsystems. If you *do* use it, be sure to model your message flows carefully, and don't confuse collection keys with message keys.

Credit, Support, and Contributions

This extension is the fault of **Stephen Eley**. You can reach me at sfeley@gmail.com. If you like science fiction stories, I know a good podcast for them as well.

If you find bugs, please report them on the Github issue tracker.

The documentation can of course be found on RDoc.info.

Contributions are welcome. I'm not sure how much more *must* be done on a flash message extension, but I'm sure there's plenty that *could* be done. Please note that the test suite uses RSpec, and you'll need the Sessionography helper for testing sessions. (I actually developed Sessionography in order to TDD *this* gem.)

License

This project is licensed under the **Don't Be a Dick License**, version 0.2, and is copyright 2010 by Stephen Eley. See the LICENSE.markdown file or the DBAD License site for elaboration on not being a dick.