Streaming Responses

Sometimes you want to start sending out data while still generating parts of the response body. In extreme examples, you want to keep sending data until the client closes the connection. You can use the stream helper to avoid creating your own wrapper:

```
get '/' do
  stream do |out|
    out << "It's gonna be legen -\n"
    sleep 0.5
    out << " (wait for it) \n"
    sleep 1
    out << "- dary!\n"
  end
end
```

This allows you to implement streaming APIs, Server Sent Events, and can be used as the basis for WebSockets. It can also be used to increase throughput if some but not all content depends on a slow resource.

Note that the streaming behavior, especially the number of concurrent requests, highly depends on the web server used to serve the application. Some servers, like WEBRick, might not even support streaming at all. If the server does not support streaming, the body will be sent all at once after the block passed to stream finishes executing. Streaming does not work at all with Shotgun.

If the optional parameter is set to keep_open, it will not call close on the stream object, allowing you to close it at any later point in the execution flow. This only works on evented servers, like Thin and Rainbows. Other servers will still close the stream:

```
# long polling

set :server, :thin
connections = []

get '/subscribe' do
  # register a client's interest in server events
  stream(:keep_open) do |out|
    connections << out
    # purge dead connections
    connections.reject!(&:closed?)
  end
end

post '/:message' do
  connections.each do |out|
    # notify client that a new message has arrived
    out << params['message'] << "\n"

    # indicate client to connect again
    out.close
  end

  # acknowledge
  "message received"
end
```

Logging

In the request scope, the logger helper exposes a Logger instance:

```
get '/' do
  logger.info "loading data"
  # ...
end
```

This logger will automatically take your Rack handler's logging settings into account. If logging is disabled, this method will return a dummy object, so you do not have to worry about it in your routes and filters.

Note that logging is only enabled for Sinatra::Application by default, so if you inherit from Sinatra::Base, you probably want to enable it yourself:

```
class MyApp < Sinatra::Base
  configure :production, :development do
    enable :logging
  end
end
```

To avoid any logging middleware to be set up, set the logging setting to nil. However, keep in mind that logger will in that case return nil. A common use case is when you want to set your own logger. Sinatra will use whatever it will find in env['rack.logger'].