Blix::Rest User Guide

Table of Contents

Introduction

Blix::Rest is a web application framework for Ruby that provides a simple and flexible way to build web services, APIs, and web applications. It supports RESTful routing, customizable controllers, and various formats for responses.

Installation

To install Blix::Rest, use the following command:

```
gem install blix-rest
```

Creating a Simple Web Service

To create a simple web service, follow these steps:

1. Create a new file named `config.ru` with the following content:

```
require 'blix/rest'

class HomeController < Blix::Rest::Controller
  get '/hello', :accept => [:html, :json], :default => :html do
    if format == :json
      {"message" => "hello world"}
    else
      "<h1>hello world</h1>"
    end
  end
end

run Blix::Rest::Server.new
```

or at its simplest, only accepting the default `:json` format:

```
require 'blix/rest'

class HomeController < Blix::Rest::Controller
  get '/hello', do
    {"message" => "hello world"}
  end
end

run Blix::Rest::Server.new
```

2. Start the server:

```
ruby -S rackup -p3000
```

3. Access the service:

- For HTML: `http://localhost:3000/hello`
- For JSON: `http://localhost:3000/hello.json`

Controllers

Controllers in Blix::Rest inherit from `Blix::Rest::Controller`. They provide various methods and hooks for handling requests and responses.

Hooks

Controllers support `before`, `before_route`, and `after` hooks:

```
class MyController < Blix::Rest::Controller

  before do
    # Code to run before each request
  end

  before_route do
    # Code to run after 'before' but before the route is executed
  end

  after do
    # Code to run after each request
  end
end
```

- `before`: Runs before any request processing begins. Use this for setup operations that should occur for all routes in the controller.
- `before_route`: Runs after `before` but before the specific route action is executed. This is useful for operations that should occur for all routes but may depend on request-specific information.
- `after`: Runs after the request has been processed. Use this for cleanup operations or final modifications to the response.

These hooks allow you to execute code at different stages of the request lifecycle, providing flexibility in how you handle requests and responses.

Routing

Blix::Rest supports various HTTP methods for routing:

```
class MyController < Blix::Rest::Controller
```

```
    get '/users' do
      # Handle GET request
    end

    post '/users' do
      # Handle POST request
    end

    put '/users/:id' do
      # Handle PUT request
    end

    delete '/users/:id' do
      # Handle DELETE request
    end

  end
```

Path Parameters

You can use path parameters in your routes:

```
  get '/user/:user_id/list' do
    user_id = path_params[:user_id]
    # Use user_id
  end
```

Query Parameters

Query parameters are key-value pairs that appear after the question mark (?) in a URL. You can access query parameters in your routes like this:

```
get '/search' do
  query = query_params[:q]
  limit = query_params[:limit]&.to_i || 10
  # Use query and limit in your search logic
  { results: perform_search(query, limit) }
end
```

Body Values

There are several methods to access data from the request body, particularly useful for POST, PUT, and PATCH requests. Here are the main ways to access body values:

1. `body_hash`: This method parses the request body and returns it as a hash. It's particularly useful for JSON payloads.

```
post '/users' do
  user_data = body_hash
  new_user = User.create(user_data)
  { id: new_user.id, message: "User created successfully" }
end
```

2. `body`: This method returns the raw request body as a string. It's useful when you need to process the raw data yourself.

```
post '/raw-data' do
  raw_data = body
  # Process the raw data
  { received_bytes: raw_data.bytesize }
end
```

3. `form_hash`: This method returns a hash of form data from POST requests. It's particularly useful for handling form submissions.

```
post '/submit-form' do
  form_data = form_hash
  # Process the form data
  { message: "Form received", name: form_data['name'] }
end
```

4. `get_data(field)`: This method allows you to get a specific field from the request body's data hash.

```
put '/users/:id' do
  user_id = path_params[:id]
  new_name = get_data('name')
  # Update user name
  { message: "User #{user_id} updated with name #{new_name}" }
end
```

When working with body values, keep these points in mind:

- The appropriate method to use depends on the `Content-Type` of the request.
- For JSON payloads, `body_hash` automatically parses the JSON into a Ruby hash.
- For form submissions, `form_hash` is the most convenient method.
- Always validate and sanitize input data before using it in your application logic.
- If the body cannot be parsed (e.g., invalid JSON), these methods may return nil or raise an error, so consider adding error handling.

Wildcard Paths

You can use wildcard paths:

```
get '/resource/*wildpath' do
  wildpath = path_params[:wildpath]
  # Use wildpath
end
```

Path Options

You can specify various options for your routes:

```
get '/users', :accept => [:html, :json], :default => :html do
  # Handle request
end
```

Available options:

- `:accept`: Formats to accept (e.g., `:html`, `[:png, :jpeg]`, `:*`)
- `:default`: Default format if not specified
- `:force`: Force response into a given format
- `:query`: Derive format from request query (default: false)
- `:extension`: Derive format from path extension (default: true)

Request Handling

Request Format

The format of a request is derived in the following order:

1. The `:force` option value if present
2. The request query `format` parameter if the `:query` option is true
3. The URL extension unless the `:extension` option is false
4. The accept header format
5. The format specified in the `:default` option
6. `:json` (default)

Response Handling

Setting Headers and Status

```
add_headers("X-Custom-Header" => "Value")
set_status(201)
```

Generating Error Responses

```
send_error("Error message", 400)
```

Predefined and Custom Formats

Blix::Rest comes with predefined formats such as `:json`, `:html`, `:xml`, and others. However, you can also register custom formats to handle specific content types.

To register a new format:

```
class MyCustomFormatParser < Blix::Rest::FormatParser
  def initialize
    super(:mycustom, 'application/x-mycustom')
  end

  def parse(text)
    # Custom parsing logic here
  end

  def format(obj)
    # Custom formatting logic here
  end
end

Blix::Rest::Server.register_parser(MyCustomFormatParser.new)
```

After registering your custom format parser, you can use it in your routes:

```
get '/custom', :accept => :mycustom do
  # Your custom format will be used for the response
  { data: 'Some data' }
end
```

Custom Responses

To return a custom response without using a registered format parser, use the `:force => :raw` option:

```
get '/custom', :accept => :xyz, :force => :raw do
  add_headers 'Content-Type' => 'text/xyz'
  "Custom response"
end
```

This approach allows you to have full control over the response format and headers.

Handling All Paths and Preserving File Extensions

In some cases, you might want to respond to all paths and keep the file extension as part of the path, rather than using it to determine the response format. You can achieve this by using a combination of options:

```
    get '/*path', :accept => :*, :extension => false, :force => :raw do
      file_path = path_params[:path]
      # Handle the request based on the full path, including any file
extension
      content = read_file(file_path)
      content_type = determine_content_type(file_path)

      add_headers 'Content-Type' => content_type
      content
    end
```

In this example:

- `:accept => :*` allows the route to accept any content type.
- `:extension => false` prevents Blix::Rest from using the file extension to determine the response format.
- `:force => :raw` gives you full control over the response, including setting the appropriate Content-Type header.

This configuration is particularly useful when you're serving static files or when you need to preserve the original path structure in your application logic.

Authentication

Blix::Rest supports basic authentication:

```
    login, password = get_basic_auth
    auth_error("Invalid login or password") unless valid_credentials?
(login, password)
```

Sessions

Blix::Rest provides session management:

```
    require 'blix/utils/redis_store'
    require 'blix/rest/session'

    class MyController < Blix::Rest::Controller
      include Blix::Rest::Session

      session_name :my_session
      session_opts :http => true
      session_manager MySessionManager.new
```

```
  def my_action
    session['user_id'] = 123
    @user_data = session['user_data']
  end
end
```

Caching

Blix::Rest provides a caching mechanism:

```
value = server_cache_get('my_key') { expensive_operation() }
```

To cache responses automatically, add `:cache => true` to your route options.

CORS (Cross-Origin Resource Sharing)

To enable CORS for a route:

```
get '/api/data' do
  set_accept_cors
  { data: 'Some data' }
end

options '/api/data' do
  set_accept_cors(
    :origin => 'https://example.com',
    :methods => [:get, :post],
    :headers => ['Content-Type', 'Authorization']
  )
end
```

Rate Limiting

Blix::Rest provides a rate limiting mechanism, only allow so many exceptions in a given time:

the delay times are applied to:

- 3x failure
- 10x failure
- 100x failure

```
rate_limit('api_calls', times: [60, 600, 86400]) do
  # Your rate-limited code here. If an exception ir raised
  # then the failure count is incremented and an exception is raised
```

```
    # until the corresponding delay is expired.
  end
```

the `api_calls` here is a key which is used to store the failure count against. If you were rate limiting a login then you might use the user name as part of this key.

The `times:` array specifies the rate limiting intervals in seconds. The default values are the same as in this example:

- `60`: Limits requests per minute (60 seconds)
- `600`: Limits requests per 10 minutes (600 seconds, which is 10 minutes)
- `86400`: Limits requests per day (86400 seconds, which is 24 hours)

Views

To render views, use the `render_erb` method:

```
  get '/users' do
    @users = User.all
    render_erb('users/index', layout: 'layouts/main')
  end
```

the location of your views defaults to `app/views` otherwise set it manually with:

globally eg:

```
  Blix::Rest.set_erb_root ::File.expand_path('../lib/myapp/views',
__FILE__)
```

or per controller eg:

```
  class MyController < Blix::Rest::Controller

     erb_dir  ::File.expand_path('../..',  __FILE__)

  end
```

then within a controller render your view with.

```
  render_erb( "users/index", :layout=>'layouts/main', :locals=>
{:name=>"charles"})
```

(locals work from ruby 2.1)

directory structure

```
views
-----
     users
     -----
          index.html.erb
     layouts
     -------
          main.html.erb
```

Logging

Configure logging:

```
Blix::Rest.logger = Logger.new('/var/log/myapp.log')
```

and log a message:

```
Blix::Rest.logger.info 'my message'
```

Testing with Cucumber

For testing with Cucumber, install the `blix-rest-cucumber` gem and follow the setup instructions in the README.

Asset Management

For asset management capabilities, you can use the separate `blix-assets` gem. This gem provides tools for managing and serving assets such as JavaScript and CSS.

To use asset management with your Blix::Rest application:

1. Install the `blix-assets` gem:

```
gem install blix-assets
```

2. Require the gem in your application:

```
require 'blix/assets'
```

3. Configure the asset manager:

```
Blix::AssetManager.config_dir = "config/assets"
```

4. Use the asset manager in your controllers:

```
asset_path('assets/main.js')
```

5. For detailed information on how to use `blix-assets`, please refer to its documentation and README file.

Controller Helper Methods

Here's a comprehensive list of helper methods available in Blix::Rest controllers:

- `add_headers`: Add headers to the response
- `after`: After hook
- `allow_methods`: Allow non-standard HTTP verbs in the controller
- `asset_path`: Get the path for an asset
- `asset_tag`: Generate an HTML tag for an asset
- `auth_error`: Raise an authentication error
- `before`: Before hook
- `before_route`: Before route hook
- `body`: The request body as a string
- `body_hash`: The request body parsed as a hash
- `env`: The request environment hash
- `escape_javascript`: Escape a JavaScript string
- `format`: The response format (:json or :html)
- `form_hash`: Returns a hash of form data from POST requests
- `get_basic_auth`: Get basic authentication credentials
- `get_cookie`: Get the value of a cookie
- `get_data`: Get a field from the request body's data hash
- `get_session_id`: Get or create a session ID
- `h`: Escape HTML string to avoid XSS
- `logger`: System logger
- `method`: The request method (lowercase, e.g., 'get', 'post')
- `mode_development?`: Check if in development mode
- `mode_production?`: Check if in production mode
- `mode_test?`: Check if in test mode
- `params`: All parameters combined
- `path`: The request path
- `path_for`: Give the external path for an internal path
- `path_params`: A hash of parameters constructed from variable parts of the path
- `post_params`: A hash of parameters passed in the request body
- `proxy`: Forward the call to another service
- `query_params`: A hash of URL query parameters

- `rate_limit`: Apply rate limiting to a block of code
- `redirect`: Redirect to another URL
- `refresh_session_id`: Generate a new session ID
- `render_erb`: Render an ERB template
- `req`: The Rack request object
- `request_ip`: The IP address of the request
- `send_data`: Send raw data with various options
- `send_error`: Send an error response
- `server_cache`: Get the server cache object
- `server_cache_get`: Retrieve/store value in cache
- `server_options`: Options passed to the server at creation time
- `session`: Access to the session object
- `set_accept_cors`: Set CORS headers for the response
- `set_status`: Set the HTTP response status
- `store_cookie`: Store a cookie in the response
- `url_for`: Give the full URL for an internal path
- `user`: The user making this request (or nil if not authenticated)
- `verb`: The HTTP verb of the request (uppercase, e.g., 'GET', 'POST')