Rack-fts: Functional Task Supervisor for Rack Applications

Project Summary and Implementation Guide

Executive Summary

Rack-fts is a Ruby gem that implements a Functional Task Supervisor (FTS) pattern for Rack-based applications, with special integration for Ruby on Rails. The gem provides a structured, type-safe pipeline for request processing through four core stages: Authenticate, Authorize, Action, and Render. Built on the dry-monads library and inspired by the functional_task_supervisor pattern, Rack-fts enables Railway Oriented Programming for clean error handling and composable request processing.

The gem supports three distinct deployment modes: standalone Rack server, Rails engine integration, and Rails middleware wrapper. This flexibility allows developers to adopt FTS patterns incrementally or build entirely FTS-based applications from the ground up.

Project Context

Research Foundation

The development of Rack-fts was informed by comprehensive research into three key technologies:

Ruby on Rails provides the foundational web framework architecture. Rails applications are built on Rack, the Ruby web server interface standard, which enables seamless middleware integration. The Rails middleware stack processes requests through a series of components, with each middleware having the opportunity to modify requests, responses, or pass control to the next component. Rails engines offer a mechanism for creating mountable sub-applications with namespace isolation, allowing FTS functionality to coexist with standard Rails routes.

functional_task_supervisor demonstrates the Stage and Task pattern using dry-monads for multi-stage execution. Each Stage represents a unit of work that returns Success or Failure, while Tasks orchestrate multiple stages in sequence. The pattern supports preconditions, dependency injection through dry-effects, and conditional execution paths. This architecture maps naturally to the FTS pipeline where authentication, authorization, action, and rendering are distinct stages with clear success and failure semantics.

dry-monads provides the Result monad implementation that enables type-safe error handling. The Result monad has two constructors: Success for successful computations and Failure for errors. The bind operation composes possibly-failing operations by executing subsequent operations only when the previous operation succeeded, creating a "railway" where failures automatically skip remaining operations. Do notation simplifies monadic composition to look like regular Ruby code while maintaining type safety and error propagation.

Design Principles

Rack-fts embodies several core design principles that emerged from the research phase:

Type Safety Through Monads: Every stage returns either Success or Failure, eliminating nil checks and exception handling throughout the codebase. The Result monad ensures that errors are explicitly handled and cannot be accidentally ignored.

Railway Oriented Programming: The pipeline creates two tracks—a success track and a failure track. Operations on the success track continue to the next stage, while operations that fail switch to the failure track and skip all remaining stages. This pattern makes error handling explicit and predictable.

Composability: Stages are independent, reusable components that can be composed in different ways. Custom stages can be created by subclassing the base Stage class and implementing a single perform method. Stages can be mixed and matched across different applications.

Separation of Concerns: Each stage has a single, well-defined responsibility. Authentication verifies identity, authorization checks permissions, action executes business logic, and rendering transforms results into responses. This separation makes code easier to understand, test, and maintain.

Flexibility: The gem supports three deployment modes to accommodate different architectural needs. Standalone mode provides a complete Rack application, engine mode integrates with existing Rails applications, and middleware mode wraps Rails with pre and post processing stages.

Architecture Overview

Core Components

The Rack-fts architecture consists of five primary components that work together to implement the FTS pipeline:

Stage serves as the base class for all FTS stages. It includes the dry-monads Result mixin and provides the call method as the main entry point for stage execution. Subclasses implement the protected perform method to define stage-specific logic. The Stage class handles exception catching, result storage, and state management through performed?, success?, and failure? predicates. Stages maintain their execution result and can be reset to unexecuted state for reuse.

Task orchestrates the execution of multiple stages in sequence. It maintains an array of stages and provides methods to add stages, run the pipeline, and query results. The run method uses dry-monads Do notation to compose stages, automatically short-circuiting on the first failure. Tasks build an initial context from the Rack environment and pass this context through each stage, accumulating data as the pipeline progresses. The Task class also supports conditional execution through the run_conditional method and custom stage selection logic.

Application implements the Rack interface for standalone server deployment. It accepts a configuration block to customize stage implementations and builds a Task with the four standard FTS stages. The call method executes the task pipeline and transforms the result into a Rack response tuple. Success results extract the response from the context, while failure results generate appropriate error responses with HTTP status codes determined by the failing stage.

Middleware wraps Rails applications with PreRails and PostRails stages. It accepts the Rails application and a configuration block, then implements the Rack interface to intercept requests. PreRails stages execute before delegating to the Rails application, allowing for early authentication, rate limiting, or request preprocessing. PostRails stages execute after Rails processing, enabling response transformation, metrics collection, or caching. The middleware preserves the Rails response and passes it through PostRails stages for additional processing.

Configuration manages stage implementations and settings through a singleton pattern. It provides default stage classes for all four FTS stages and allows customization through a configure block. Configuration supports both global settings and per-application overrides, enabling different stage implementations across multiple applications in the same process.

Stage Pipeline

The FTS pipeline processes requests through four sequential stages, each with distinct responsibilities and failure modes:

Authenticate Stage verifies the identity of the requester by examining credentials in the request. The default implementation extracts Bearer tokens from the Authorization header and validates them. Custom authentication can implement JWT verification, session validation, OAuth flows, or certificate-based authentication. Authentication failures return Failure with code :invalid_credentials or :missing_credentials, which the application transforms into 401 Unauthorized responses.

Authorize Stage determines whether the authenticated identity has permission to perform the requested action. It receives the identity from the Authenticate stage and extracts the resource and action from the request. The default implementation allows all requests, but custom authorization can implement role-based access control (RBAC), attribute-based access control (ABAC), or access control lists (ACL). Authorization failures return Failure with code :access_denied, resulting in 403 Forbidden responses.

Action Stage executes the core business logic of the request. It receives both the authenticated identity and authorization permissions, ensuring that only authenticated and authorized requests reach business logic. The default implementation returns a simple success message, but custom actions can route to controllers, execute domain operations, or interact with databases and external services. Action failures can specify custom HTTP status codes through the status field in the Failure result.

Render Stage transforms the action result into an HTTP response. The default implementation renders JSON responses, but custom renderers can generate HTML, XML, Protocol Buffers, or any other format. The Render stage receives the complete context including identity, permissions, and action result, allowing it to make rendering decisions based on user preferences or content negotiation. Render failures are rare but can occur if serialization fails or templates are missing.

Monadic Composition

The stages are composed using dry-monads bind operation, which creates the railway effect. When a stage returns Success, bind unwraps the value and passes it to the next stage. When a stage returns Failure, bind short-circuits and returns the Failure without executing subsequent stages. This composition can be written explicitly with bind or implicitly with Do notation.

Explicit composition with bind creates nested blocks where each stage's success value is available in its block scope. This approach makes the data flow explicit but can become deeply nested with many stages.

Do notation flattens the composition by using yield to unwrap Success values. If any yield encounters a Failure, the method immediately returns that Failure. This approach looks like regular procedural code while maintaining all the benefits of monadic composition.

The Task class uses Do notation internally, making the stage composition clean and readable. Custom tasks can override the determine_next_stage method to implement conditional execution paths based on stage results.

Implementation Details

File Structure

The Rack-fts gem follows standard Ruby gem conventions with a clear separation between library code, tests, features, and examples:

```
rack-fts/
├── lib/
│   ├── rack-fts.rb                      # Main entry point, requires all
components
│   └── rack/
│       └── fts/
│           ├── version.rb               # Gem version constant
│           ├── configuration.rb         # Singleton configuration
management
│           ├── stage.rb                 # Base Stage class with Result
monad
│           ├── task.rb                  # Task orchestrator with Do
notation
│           ├── application.rb           # Standalone Rack application
│           ├── middleware.rb            # Rails middleware wrapper
│           └── stages/
│               ├── authenticate.rb      # Default authentication stage
│               ├── authorize.rb         # Default authorization stage
│               ├── action.rb            # Default action stage
│               └── render.rb            # Default render stage
├── spec/
│   ├── spec_helper.rb                   # RSpec configuration
│   └── rack/
│       └── fts/
│           ├── stage_spec.rb            # Stage unit tests
│           ├── task_spec.rb             # Task unit tests
│           └── application_spec.rb      # Application integration tests
├── features/
│   ├── support/
│   │   └── env.rb                       # Cucumber environment setup
│   ├── step_definitions/
│   │   └── fts_steps.rb                 # Cucumber step definitions
│   └── standalone_server.feature        # Standalone use case scenarios
├── examples/
│   ├── standalone/
│   │   └── config.ru                    # Standalone server example
│   ├── rails_engine/
│   │   └── config/routes.rb             # Rails engine mounting example
│   └── rails_middleware/
│       └── config/application.rb        # Rails middleware example
├── rack-fts.gemspec                     # Gem specification
├── Gemfile                              # Development dependencies
├── Rakefile                             # Rake tasks for testing
├── README.md                            # Comprehensive documentation
├── CHANGELOG.md                         # Version history
├── LICENSE                                  # MIT License
└── .rspec                                   # RSpec configuration
```

Key Implementation Patterns

Result Monad Usage: Every stage method returns a Result monad, either Success or Failure. Success wraps the updated context hash, while Failure wraps an error hash with structured information. The Stage base class catches all exceptions and converts them to Failure results, ensuring that exceptions never escape the pipeline.

Context Passing: The context hash flows through all stages, accumulating data as it progresses. The initial context contains the Rack environment, request object, and response object. Each stage adds its data to the context—Authenticate adds identity, Authorize adds permissions, Action adds action_result. This pattern avoids global state and makes data flow explicit.

Configuration Flexibility: The Configuration class uses the Singleton pattern to provide global defaults while allowing per-application overrides. Applications can configure stage classes globally through Rack::FTS.configure or locally by passing a block to Application.new or Middleware.new. This flexibility supports both convention and customization.

Error Response Generation: The Application and Middleware classes include error_response methods that transform Failure results into Rack response tuples. The determine_status_code method maps stage names to HTTP status codes, with Authenticate failures becoming 401, Authorize failures becoming 403, and Action failures using custom status codes or defaulting to 500.

Testing Strategy: The gem includes both unit tests with RSpec and behavior tests with Cucumber. RSpec tests verify individual components in isolation using mocks and stubs. Cucumber features test complete request flows through the pipeline, verifying that the stages compose correctly and produce expected responses. This dual approach ensures both component correctness and integration reliability.

Use Case Implementations

Use Case 1: Standalone FTS Server

The standalone server use case demonstrates Rack-fts as a complete Rack application for microservices or API-only applications. This mode is ideal when building services that do not need the full Rails framework but benefit from structured request processing.

The example implementation in examples/standalone/config.ru shows custom stages for authentication, authorization, and action. The CustomAuthenticate stage verifies tokens against a secret value, demonstrating how to integrate with external authentication services. The CustomAuthorize stage implements simple role-based access control, allowing GET requests for all users but restricting POST/PUT/DELETE to specific users. The CustomAction stage routes requests to different handlers based on path, showing how to implement RESTful resource handling.

Running the standalone server requires only rackup with no additional dependencies beyond the gem itself. The server responds to HTTP requests with JSON responses, handling authentication failures with 401 status codes and authorization failures with 403 status codes. Successful requests return 200 with the action result data.

This use case is particularly valuable for microservices architectures where each service needs consistent authentication and authorization but does not require the overhead of a full web framework. The FTS pipeline ensures that all services follow the same request processing pattern, making the architecture more uniform and maintainable.

Use Case 2: Rails Engine Integration

The Rails engine use case demonstrates mounting Rack-fts at a specific path within a Rails application. This mode allows FTS functionality to coexist with standard Rails routes, providing a gradual migration path or specialized endpoints with different processing requirements.

The engine implementation would extend Rails::Engine with isolate_namespace to prevent naming conflicts with the host application. The engine can be mounted at any path such as /fts or /api/v2, making its routes available under that prefix. The engine maintains its own controllers, models, and views while sharing the Rails environment with the host application.

This use case is valuable when adding new API versions with different authentication requirements, implementing administrative interfaces with enhanced security, or providing specialized endpoints that benefit from the FTS pipeline while maintaining existing Rails routes unchanged. The engine approach provides clean separation while leveraging Rails infrastructure.

Use Case 3: Rails Rack Middleware

The Rails middleware use case demonstrates wrapping the entire Rails application with PreRails and PostRails stages. This mode is ideal for adding cross-cutting concerns that apply to all requests without modifying existing controllers.

The middleware implementation in examples/rails_middleware/config/application.rb shows PreRails stages for rate limiting and authentication, executing before any Rails code runs. PostRails stages collect metrics and cache responses after Rails processing completes. The middleware preserves the Rails response and passes it through PostRails stages, allowing response transformation without modifying controllers.

PreRails stages can implement early request rejection for rate limiting, authentication, or request validation. This prevents invalid requests from consuming Rails resources. PostRails stages can add response headers, collect performance metrics, cache responses, or transform response formats. This separation keeps controllers focused on business logic while centralizing cross-cutting concerns.

The middleware use case is particularly valuable for adding observability, security, or performance features to existing Rails applications without modifying application code. The middleware approach provides a single point of control for request processing policies.

Testing and Quality Assurance

RSpec Unit Tests

The RSpec test suite provides comprehensive coverage of individual components. The Stage spec tests the base class behavior including initialization, execution, state management, and exception handling. It verifies that stages correctly return Success or Failure, maintain execution state, and can be reset for reuse.

The Task spec tests stage orchestration, verifying that tasks execute stages in sequence, short-circuit on failure, and accumulate results correctly. It tests both successful pipelines where all stages succeed and failure scenarios where early stages fail and later stages are skipped.

The Application spec tests the Rack interface integration, verifying that requests with valid authentication return 200 responses, requests without authentication return 401 responses, and custom stage configurations are respected. It uses Rack::Test to simulate HTTP requests and verify responses.

Cucumber Behavior Tests

The Cucumber feature suite tests complete request flows through the FTS pipeline. The standalone_server.feature describes scenarios for successful authentication, missing authentication, invalid authentication, different HTTP methods, and custom stage configurations.

Each scenario describes the user's perspective—what they do and what they expect to see—without exposing implementation details. The step definitions translate these descriptions into code that exercises the application and verifies responses. This approach ensures that the gem behaves correctly from the user's perspective.

Test Execution

Running the complete test suite requires installing development dependencies with bundle install, then executing bundle exec rspec for unit tests and bundle exec cucumber for behavior tests. The Rakefile provides convenient tasks: rake spec for RSpec, rake cucumber for Cucumber, and rake default to run both test suites.

The test suite includes SimpleCov for code coverage reporting, ensuring that all code paths are exercised by tests. The coverage report identifies untested code and guides test development.

Deployment and Usage

Installation

Installing Rack-fts requires adding the gem to the Gemfile and running bundle install. For standalone applications, only rack-fts is required. For Rails integration, the Rails gem must also be present. The gem specifies Ruby 3.3.6 or later as the minimum required version.

Configuration

Configuring Rack-fts involves choosing stage implementations and setting options. Global configuration through Rack::FTS.configure affects all applications in the process. Per-application configuration through blocks passed to Application.new or Middleware.new affects only that application.

Stage implementations are specified by class, not instance. The configuration stores stage classes, and the Application or Middleware creates instances as needed. This pattern allows stages to maintain state during request processing without sharing state across requests.

Custom Stage Development

Developing custom stages requires subclassing Rack::FTS::Stage and implementing the protected perform method. The perform method receives the context hash and must return either Success(context) or Failure(error_hash). The context should be updated with stage-specific data before returning Success.

Custom stages can override preconditions_met? to validate context before execution. If preconditions are not met, the stage can return Failure immediately without attempting execution. This pattern is useful for stages that depend on data from previous stages.

Error Handling

Error handling in Rack-fts is explicit and structured. Every stage returns Success or Failure, and the pipeline automatically handles failure propagation. Applications do not need to catch exceptions or check for nil values—the Result monad ensures that errors are handled consistently.

Error responses include the error message, stage name, and timestamp. Additional fields can be added to provide context-specific information. The HTTP status code is determined by the failing stage, with sensible defaults for authentication and authorization failures.

Architecture Diagrams

Class Diagram

The class diagram illustrates the relationships between Rack-fts components. The Stage base class is subclassed by Authenticate, Authorize, Action, and Render stages. The Task class aggregates multiple Stage instances and orchestrates their execution. The Application class implements the Rack interface and uses Task for pipeline execution. The Middleware class also implements the Rack interface and wraps a Rails application with PreRails and PostRails stages.

The diagram shows that Stage depends on dry-monads Result for return types. Task uses dry-monads Do notation for composition. Application and Middleware both use Configuration for stage selection. The Rails Engine extends Rails::Engine and uses Configuration for setup.

Sequence Diagram

The sequence diagram illustrates three request flows: successful request, failed authentication, and failed authorization. In the successful flow, the client sends a request to the Rack server, which calls Application. Application creates a Task and runs the pipeline. Each stage executes in sequence, returning Success. The final response is extracted from the context and returned to the client.

In the failed authentication flow, the Authenticate stage returns Failure. The Task short-circuits and returns Failure to Application. Application generates an error response with 401 status and returns it to the client. The Authorize, Action, and Render stages never execute.

In the failed authorization flow, Authenticate succeeds but Authorize returns Failure. The Task short-circuits after Authorize, skipping Action and Render. Application generates an error response with 403 status. This demonstrates how the railway pattern automatically handles failure propagation.

Future Enhancements

Several enhancements could extend Rack-fts capabilities while maintaining its core simplicity:

Async Stage Execution could allow stages to return Task monads for asynchronous operations. This would enable non-blocking I/O for authentication against external services or database queries. The Task orchestrator would need to handle async composition, potentially using dry-effects for effect handling.

Stage Hooks could provide before_stage, after_stage, and on_failure callbacks for cross-cutting concerns like logging, metrics, or tracing. Hooks would execute around stage execution without modifying stage implementations, keeping stages focused on their primary responsibilities.

Conditional Stage Execution could be enhanced with a DSL for expressing execution paths. Instead of overriding determine_next_stage, developers could declare conditions and branches declaratively. This would make complex pipelines more maintainable and testable.

Stage Composition could allow stages to contain sub-stages, creating hierarchical pipelines. A composite stage would execute its sub-stages and aggregate their results. This pattern would enable reusable stage groups and more modular pipeline construction.

Performance Monitoring could be built into the Task orchestrator, collecting timing information for each stage and the overall pipeline. This data could be exposed through metrics endpoints or logged for analysis, providing visibility into pipeline performance.

Conclusion

Rack-fts provides a robust foundation for building type-safe, composable request processing pipelines in Ruby applications. By leveraging dry-monads for error handling and the functional_task_supervisor pattern for stage orchestration, the gem enables Railway Oriented Programming in Rack and Rails applications.

The three deployment modes—standalone server, Rails engine, and Rails middleware—provide flexibility for different architectural needs. Developers can adopt FTS patterns incrementally through middleware or engines, or build entirely FTS-based applications with the standalone mode.

The comprehensive test suite, clear documentation, and working examples make Rack-fts accessible to developers familiar with Rack and Rails. The extensible architecture through custom stages allows adaptation to diverse authentication, authorization, and business logic requirements while maintaining the benefits of structured, type-safe request processing.

Rack-fts represents a modern approach to web application architecture in Ruby, bringing functional programming patterns and type safety to the Rack ecosystem while maintaining the simplicity and elegance that Ruby developers expect.

Project Information

- **Gem Name**: rack-fts
- **Version**: 0.1.0
- **Ruby Version**: >= 3.3.6
- **License**: MIT
- **Dependencies**: rack (~> 3.0), dry-monads (~> 1.6), dry-configurable (~> 1.0)
- **Author**: Manus AI
- **Date**: January 11, 2026