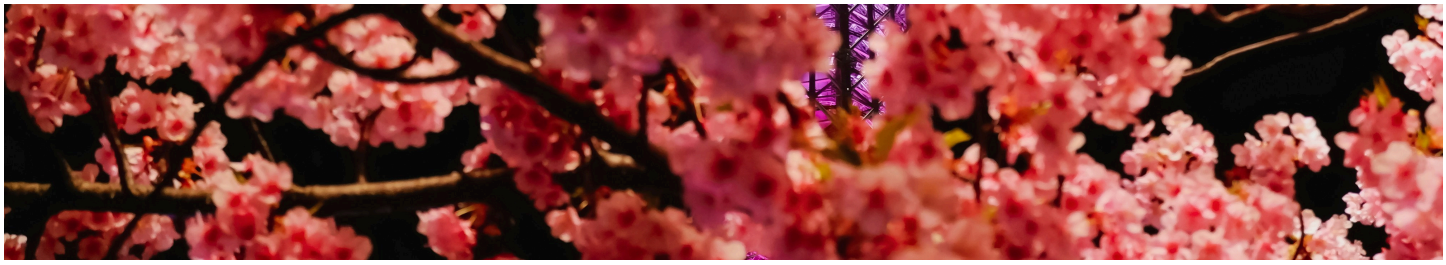


Hanami: Alternative for Rails

ToC

Document Status

Jira Issue(s)	☑ ODIN-28887: Exploration | Hanami Framework for Ruby TO DO
Status	IN PROGRESS
Impact	HIGH LOW MEDIUM
Owner	@Pratyush Rastogi

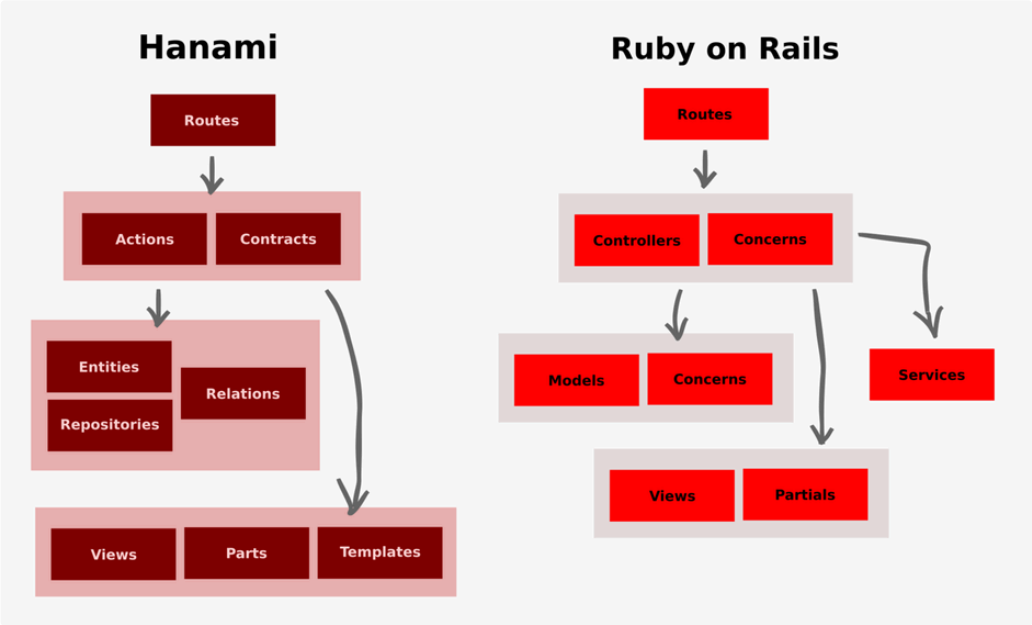
Introduction

- Hanami framework is **modular**, offering flexibility and maintainability. It allows using specific components as needed.
- It strictly adheres to an **object-oriented approach**, treating everything within the framework as an object.
- Hanami follows the **repository pattern**, allowing for separation of data layer from repository logic, simplifying database implementation switching.
- Hanami's modular architecture allows for the development and deployment of individual **slices**, which can be managed and scaled independently.
- Hanami is compatible with **rspec-rails**, for writing and executing tests.
- The framework supports different environments based on the **HANAMI_ENV** environment variable, facilitating seamless configuration and management.

Comparison: Hanami vs Rails

Feature	Hanami	Ruby on Rails
Architecture	Modular, with separate components for actions, views, models, etc. An Introduction To Hanami 2.0	Monolithic, with convention over configuration
ORM	Supports multiple ORMs like ROM and Sequel Hanami: Alternative for Rails | ROM	Built-in ORM: ActiveRecord
Testing	Supports RSpec	Supports RSpec
Database Migrations	Uses separate migration files for each change need to write separate rollback migrations	Built-in migration system with `rake db:migrate`
Env configurations	GitHub - bkeepers/dotenv: A Ruby gem to load environment variables from `.env`.	settings.yml file
Background Jobs	No built-in support; can integrate with external libraries like Sidekiq or Shoryuken Integrate Sidekiq with Hanami Applications	Built-in support for background jobs with Active Job
Gems Support	much less compared to Rails. May need integration or changes to to get them working	vast number of gems available through RubyGems
Flexibility	Provides more flexibility due to its modular nature	Convention over configuration may limit flexibility
Performance	Lightweight and can be faster for certain use cases Should You Use Ruby on Rails or Hanami? | AppSignal Blog	Slightly heavier due to its monolithic nature

Learning Curve	Steeper learning curve due to its unique architecture and concepts	Lower learning curve due to convention over configuration
Community	Smaller community compared to Rails	Large and established community
Popularity	Less popular compared to Rails	One of the most popular web frameworks for Ruby



Hanami and Rails application architecture

Hanami Application Features

Commands

mostly similar to rails

```
HANAMI_ENV=development bundle exec hanami server # start server
HANAMI_ENV=development bundle exec hanami c # start hanami console
HANAMI_ENV=development bundle exec rake db:migrate # run all migrations
```

Routes

- **config/routes.rb** used to define routes
- scopes can be used to define namespace containing a group of endpoints

```
module Bookshelf
  class Routes < Hanami::Routes
    # Add your routes here. See https://guides.hanamirb.org/routing/overview/ for details.
    root to: "home.show"
    get "/home/:id", id: /\d+/, to: "home.show"

    scope "books" do
      get "/", to: "books.index", as: :all
      get "/:id", id: /\d+/, to: "books.show"  # => /books/:id
      post "/", to: "books.create"
    end
```

defining routes in the application

```
pratyushrastogi@hk5v6d3:~/hanami_workspace/bookshelf$ hanami routes
GET     /                           home.show                   as :root
GET     /home/:id                   home.show                   (id: /\d+/)
GET     /books                      books.index
GET     /books/:id                  books.show                  (id: /\d+/)
POST    /books                      books.create
```

routes available in the application

Prepare vs Boot

```
bookshelf[development]> Hanami.app.prepare
=> Bookshelf::App
bookshelf[development]> Hanami.app.keys
=> ["settings", "notifications"]
bookshelf[development]> Hanami.app.boot
=> Bookshelf::App
bookshelf[development]> Hanami.app.keys
=> ["settings",
 "notifications",
 "routes",
 "assets",
 "inflector",
 "logger",
 "rack.monitor",
 "persistence.config",
 "persistence.db",
 "persistence.rom",
 "actions.books.create",
 "actions.books.index",
 "actions.books.show",
 "actions.home.show"]
```

components loading comparison- prepare vs boot

- prepare will use lazy loading to initialize the app, i.e, loading components only when they are required.
- puma servers will use eager loading (boot the application) by default

Container and its Components

- Hanami app is organised into a container with its registered components
- This allows setting up the application by registering multiple components with the application
- The interaction among the components is handled through dependency injection
- Dependency Injection is achieved using **Deps/Mixins** which will load the required components

```ruby
class Index < Bookshelf::Action
  include Deps["persistence.rom"]

  params do
    optional(:page).value(:integer, gt?: 0)
    optional(:per_page).value(:integer, gt?: 0, lteq?: 100)
  end

  def handle(request, response)
    halt 422, {errors: request.params.errors}.to_json unless request.params.valid?

    books = rom.relations[:books]
      .select(:title, :author)
      .order(:title)
      .page(request.params[:page] || 1)
      .per_page(request.params[:per_page] || 5)
      .to_a
```

loading “persistence.rom” component into the index action for querying database

Providers

- Providers provide a way to **register components** (generally external like database, external APIs), outside of the automatic registration mechanism
- lifecycle methods-
  - prepare - basic setup code, here you can require third-party code, or code from your `lib` directory, and perform basic configuration
  - start - code that needs to run for a component to be usable at runtime
  - stop - code that needs to run to stop a component, perhaps to close a database connection, or purge some artifacts.

config > providers > persistence.rb

```ruby
Hanami.app.register_provider :persistence, namespace: true do
  prepare do
    require "rom"

    config = ROM::Configuration.new(:sql, target["settings"].database_url)

    register "config", config
    register "db", config.gateways[:default].connection
  end

  start do
    config = target["persistence.config"]

    config.auto_registration(
      target.root.join("lib/bookshelf/persistence"),
      namespace: "Bookshelf::Persistence"
    )

    register "rom", ROM.container(config)
  end
end

```

database persistence provider with prepare and start lifecycle methods

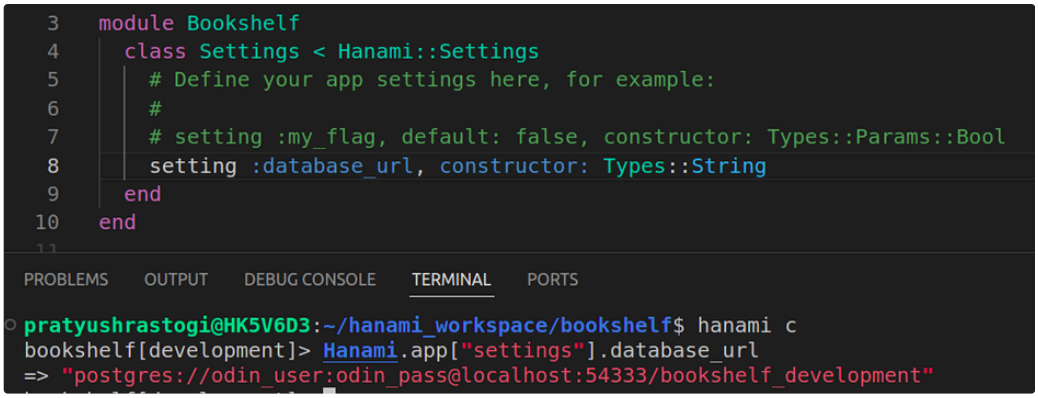
- `target` is used to refer other componenets within the application, like `target["settings"]`
- During application **boot**, start method is called for each provider to register its component

Providers vs Components

- use **providers** when you need to interact with **external services or resources**
- **providers** are also used when we need to create a single class to be reused in code (singleton pattern)
- use **components** when you want to organize your application's internal logic into reusable and composable units.

Settings

- Hanami uses the dotenv gem to load environment variables from `.env` files.
- We can access settings using-
  - Deps in components
  - target in providers
  - settings methods in app.rb



defining and using settings within the application

- We can also mark some keys optional, set constraints/default values in settings-

```ruby
# config/settings.rb

module Bookshelf
  class Settings < Hanami::Settings
    EMAIL_FORMAT = /\A[\w+\-.]+@[a-z\d\-]+(\.[a-z]+)*\.[a-z]+\z/i
    # constraint
    setting :from_email, constructor: Types::String.constrained(format: EMAIL_FORMAT)
    # optional field
    setting :max_cart_items, constructor: Types::Params::Integer.optional
    # default value
    setting :analytics_enabled, default: false, constructor: Types::Params::Bool
  end
end
```

Slices

- Slices provide a way to segregate logic into independent modules, promoting high modularity and code isolation
- Slices basically create a separate container having it's own components, providers and actions
- We can configure the application to boot only selected slices
- Slices can also share components among them

reference: V2.1: Slices | Hanami Guides

For illustration, we have defined two slices:

- *Admin → CRUD user, create/edit book, create author*
- *Home → list all books, view book details, list all authors, view author details*

```
bookshelf[development]> AdminService::Slice.boot.keys
=> ["assets",
 "actions.authors.create",
 "actions.books.create",
 "actions.books.update",
 "actions.users.create",
 "actions.users.index",
 "actions.users.show",
 "inflector",
 "logger",
 "notifications",
 "rack.monitor",
 "routes",
 "settings",
 "persistence.config",
 "persistence.db",
 "persistence.rom"]
bookshelf[development]> HomeService::Slice.boot.keys
=> ["assets",
 "actions.authors.create",
 "actions.authors.index",
 "actions.authors.show",
 "actions.books.index",
 "actions.books.show",
 "inflector",
 "logger",
 "notifications",
 "rack.monitor",
 "routes",
 "settings",
 "persistence.config",
 "persistence.db",
 "persistence.rom"]
```

different action components under admin and home slices

config > app.rb > ...

```ruby
require "hanami"

module Bookshelf
  class App < Hanami::App
    config.middleware.use :body_parser, :json
    config.shared_app_component_keys += [
                                          "persistence.config",
                                          "persistence.db",
                                          "persistence.rom",
                                        ]
  end
end

```

persistence components shared among the slices

ROM

documentation: [ROM - Introduction]

ROM vs Active Record: [ROM - Compared to ActiveRecord]

Migrations

[sequel/doc/schema_modification.rdoc at master · jeremyevans/sequel]

- We need to create a separate migration file for rollback using ROM

```
HANAMI_ENV=development bundle exec rake db:migrate[2] #rake db:migrate[version]
```

db > migrate > 2_create_authors.rb

```ruby
ROM::SQL.migration do
  change do
    create_table? :authors do
      primary_key :id, type: :bigint
      column :name, :text, null: false
      column :email, :text
      column :contact_no, :numeric
      index :email, unique: true
    end
  end
end
```

ROM migration with unique index

`rom-sql` provides Migration API for managing the schema in a SQL database.

we can add timestamps by defining custom changesets ROM - Changesets

Relations

- A relation represents a queryable collection of data in your database.
- It is typically mapped to a database table or view.
- Relations provide a way to query and manipulate data at a lower level than repositories. They offer fine-grained control over database interactions.
- **ROM relations return hashes by default**

```ruby
module Bookshelf
  module Persistence
    module Relations
      class Authors < ROM::Relation[:sql]
        struct_namespace Entities
        auto_struct true

        schema(:authors, infer: true) do
          associations do
            has_many :books
          end
        end
```

Authors relation defining attributes and associations

```
bookshelf[development]> rom.relations[:books].by_pk(1).one
=> {:id=>1, :title=>"The Monk Who Sold His Ferrari", :author_id=>1, :images=>nil}
bookshelf[development]> rom.relations[:books].by_pk(1).one.class
=> Hash
```

ROM relations return hashes by default

Entities

- An entity encapsulates the attributes and behavior associated with that object
- Entities are typically plain Ruby objects (POROs) with attributes and methods related to the relation (can be understood as model but without persistence & validation logic)
- Entities are usually mapped to database tables, but they are not tightly coupled to the database structure.

```ruby
module Bookshelf
  module Persistence
    module Entities
      class Author < ROM::Struct
        def first_name
          name.split(' ').first
        end

        def last_name
          name.split(' ').last
        end
      end
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                ruby

bookshelf[development]> rom.relations[:authors].by_pk(1).one.last_name
=> "Sharma"
bookshelf[development]> rom.relations[:authors].by_pk(1).one
=> #<Bookshelf::Persistence::Entities::Author id=1 name="Robin Sharma" email=nil contact_no=nil>
bookshelf[development]> rom.relations[:authors].by_pk(1).one.class
=> Bookshelf::Persistence::Entities::Author
```

entity can be defined to map relation data into objects

Repositories

- A repository acts as an interface between your application's entities and the data store (usually a database).
- Repositories encapsulate data access logic, providing **methods for querying, creating, updating, and deleting entities.**
- **By default, repos return simple `ROM::Struct` objects.**
- `users.where` and `users.by_pk` are SQL-specific interfaces that are separated from the application layer using repository interface.
- Relations are often used internally by repositories to perform data operations such as fetching records that match certain criteria.

```ruby
lib > bookshelf > persistence > repositories > authors.rb > { } Bookshelf > { } Persistence > { } Repositories > Authors > find_by_id
module Bookshelf
  module Persistence
    module Repositories
      class Authors < ROM::Repository[:authors]

        struct_namespace Entities # define namespace where entities are defined
        commands :create, update: :by_id

        def all
          authors.to_a
        end

        def query(conditions)
          authors.where(conditions).to_a
        end

        def find_by_id(id)
          by_id(id).one!
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                ruby

bookshelf[development]> author_repo = Bookshelf::Persistence::Repositories::Authors.new(rom)
=> #<Bookshelf::Persistence::Repositories::Authors struct_namespace=Bookshelf::Persistence::Entities auto_stru
ct=true>
bookshelf[development]> author_repo.find_by_id(6)
=> #<Bookshelf::Persistence::Entities::Author id=6 name="James Clear" email="james_clear@gmail.com" contact_no
=0.2314342311e10>
bookshelf[development]> author_repo.find_by_id(6).first_name
=> "James"
bookshelf[development]> author_repo.query(name: "James Clear")
=> [#<Bookshelf::Persistence::Entities::Author id=6 name="James Clear" email="james_clear@gmail.com" contact_n
o=0.2314342311e10>]
```

repository defining the abstraction methods between database and application layer

Getting Started

V2.1: Getting Started | Hanami Guides

Installation

GitHub - hanami/hanami: The web, with simplicity.

Common errors

- **No such file or directory - npm (Dry::Files::IOError)**
  install npm
- **bundler: command not found: guard**
  If Guardfile is missing from project, add by running-

```
bundle exec hanami install
```

Tools

1. rom - substitute for active record in rails
   ROM
2. dotenv - used for managing environment level configurations
   GitHub - bkeepers/dotenv: A Ruby gem to load environment variables from `.env`.

References

sidekiq: Integrate Sidekiq with Hanami Applications

hanami features & introduction: RubyConfTH 2022 - Hanami 2: New Framework, New You by Tim Riley

hanami slices & working: RubyConf 2023 - Livin' La Vida Hanami by Tim Riley

hanami walkthrough: Hanami 2.0 an alternative to Rails - Daniel Nguyen

comparision with rails:
An Introduction To Hanami 2.0

Should You Use Ruby on Rails or Hanami? | AppSignal Blog

ROM: ROM - Introduction