Admeet architecture

Backend part

Backend parts:

- Ruby
- PostgreSQL
- GraphQL

General architecture

- No framework - dry-web-roda (ruby toolkit based on dry-* and ROM gems)
- Umbrella architecture
- Almost no traditional routing - GraphQL instead
- Functional Programming approach. Every request is a composition of pre-prepared functions (class instances with method call) stored in container

Umbrella architecture

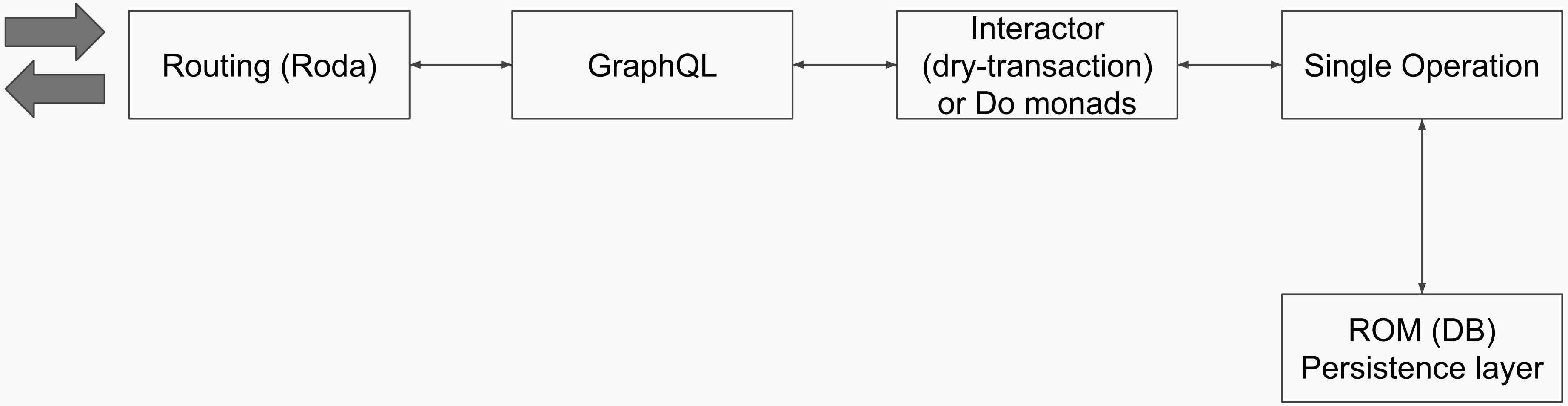
In root level directory we keep core business logic

Specialized “apps” in /apps directory, currently:

- General API
- Admin (back office) API
- Reader (generates HTML and PDF documents - PP and CP docs)
- Banner (generates cookie banner)
- Billings (Stripe payments)

All those follow the same pattern (structure - dry-system), all of them shares core logic

Typical Request flow


Routing (Roda)
GraphQL
Interactor (dry-transaction) or Do monads
Single Operation
ROM (DB) Persistence layer

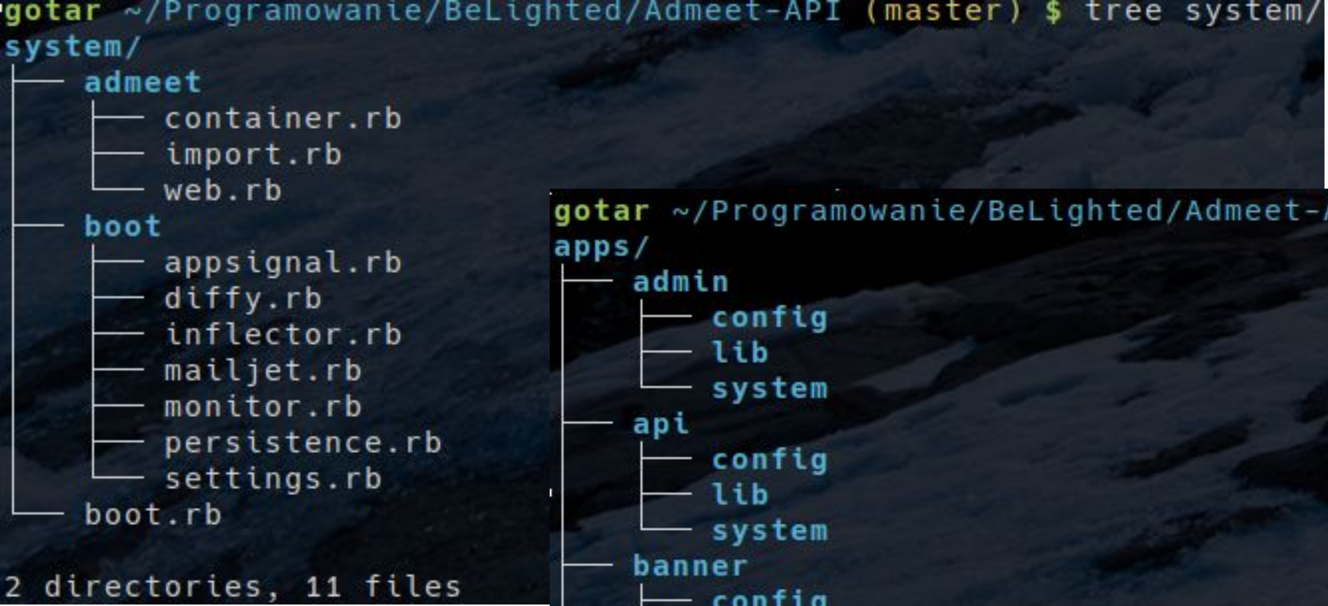
Dry-system

Object dependency management system based on dry-container and dry-auto_inject allowing you to configure reusable components in any environment, set up their load-paths, require needed files and instantiate objects automatically with the ability to have them injected as dependencies.

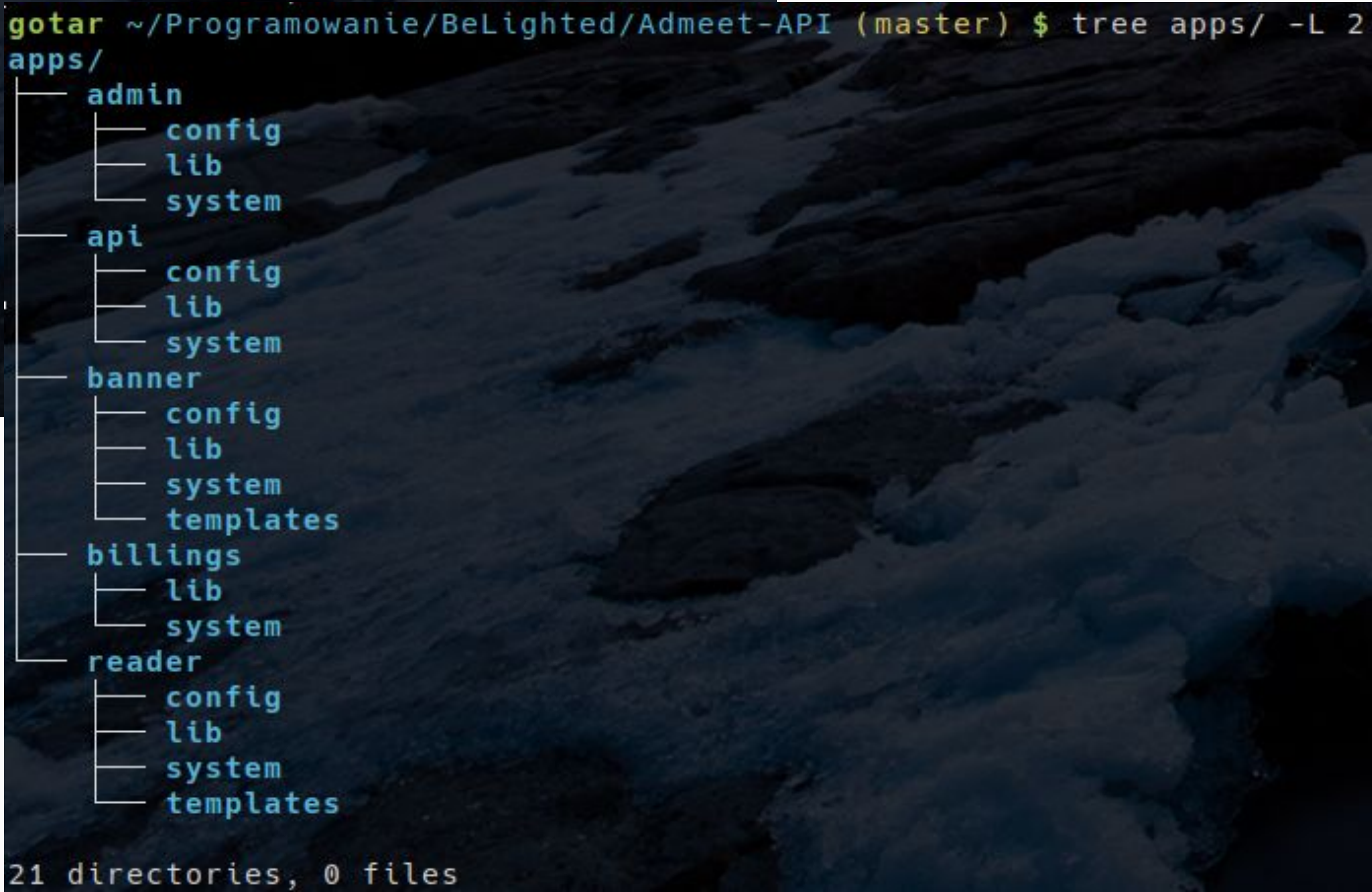
This library relies on very basic mechanisms provided by Ruby, specifically `require` and managing `$LOAD_PATH`. It doesn't use magic like automatic const resolution, it's pretty much the opposite and forces you to be explicit about dependencies in your applications.

It does a couple of things for you:

- Provides an abstract dependency container implementation
- Handles `$LOAD_PATH` configuration
- Loads needed files using `require`
- Resolves object dependencies automatically
- Supports auto-registration of dependencies via file/dir naming conventions
- Supports multi-system setups (ie your application is split into multiple sub-systems)
- Supports configuring component providers, which can be used to share common components between many systems
- Supports test-mode with convenient stubbing API



```
gotar ~/Programowanie/BeLighted/Admeet-API (master) $ tree system/
system/
├── admeet
│   ├── container.rb
│   ├── import.rb
│   └── web.rb
├── boot
│   ├── appsignal.rb
│   ├── diffy.rb
│   ├── inflector.rb
│   ├── mailjet.rb
│   ├── monitor.rb
│   ├── persistence.rb
│   └── settings.rb
└── boot.rb

2 directories, 11 files
```

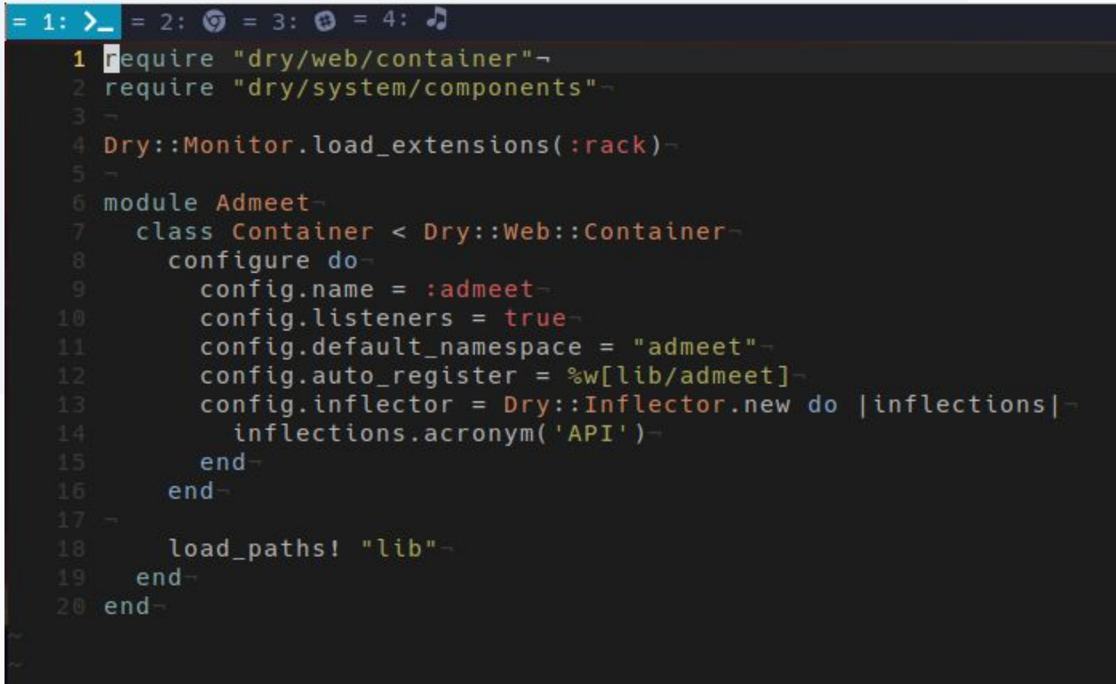


```
gotar ~/Programowanie/BeLighted/Admeet-API (master) $ tree apps/ -L 2
apps/
├── admin
│   ├── config
│   ├── lib
│   └── system
├── api
│   ├── config
│   ├── lib
│   └── system
├── banner
│   ├── config
│   ├── lib
│   ├── system
│   └── templates
├── billings
│   ├── lib
│   └── system
└── reader
    ├── config
    ├── lib
    ├── system
    └── templates

21 directories, 0 files
```

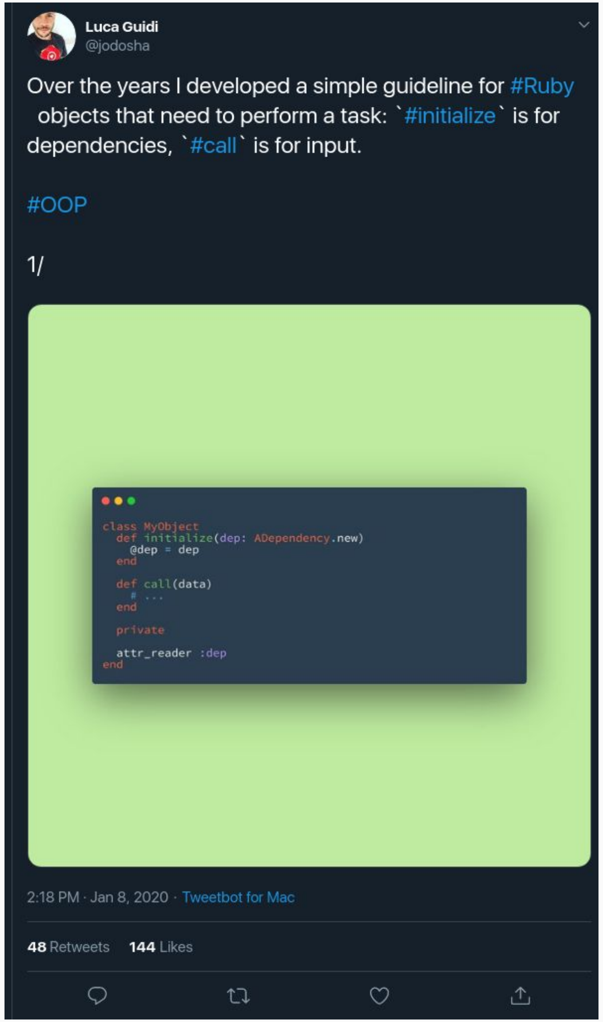
Dry-Container

`dry-container` is a simple, thread-safe container, intended to be one half of a dependency injection system, possibly in combination with dry-auto_inject.

```ruby
container = Dry::Container.new
container.register(:parrot) { |a| puts a }

parrot = container.resolve(:parrot)
parrot.call("Hello World")
# Hello World
# => nil
```



```ruby
require "dry/web/container"
require "dry/system/components"

Dry::Monitor.load_extensions(:rack)

module Admeet
  class Container < Dry::Web::Container
    configure do
      config.name = :admeet
      config.listeners = true
      config.default_namespace = "admeet"
      config.auto_register = %w[lib/admeet]
      config.inflector = Dry::Inflector.new do |inflections|
        inflections.acronym('API')
      end
    end

    load_paths! "lib"
  end
end
```



Luca Guidi
@jodosha

Over the years I developed a simple guideline for #Ruby objects that need to perform a task: `#initialize` is for dependencies, `#call` is for input.

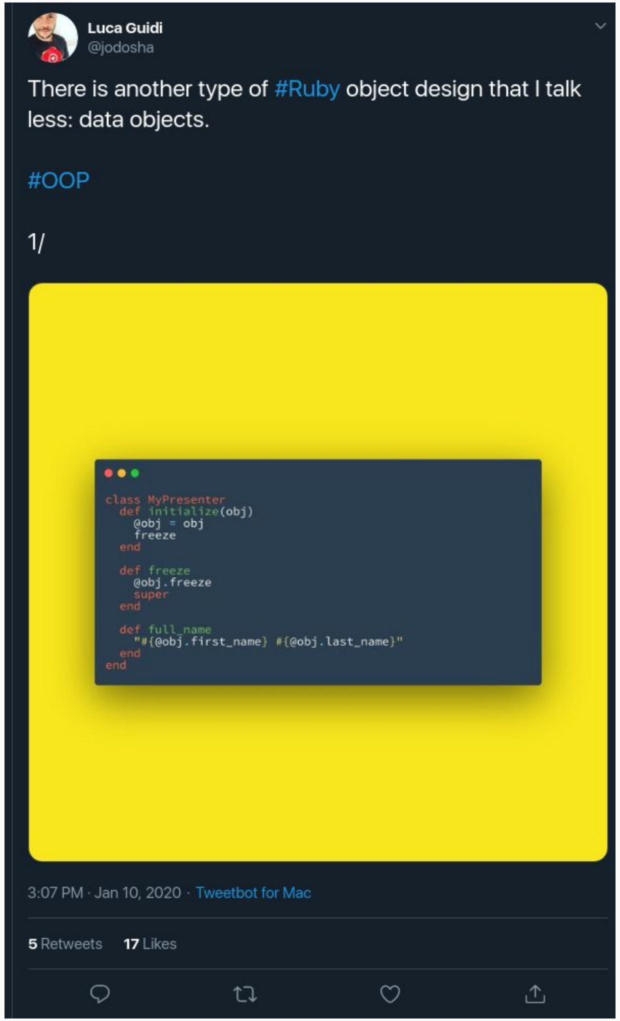
#OOP

1/

```ruby
class MyObject
  def initialize(dep: ADependency.new)
    @dep = dep
  end

  def call(data)
    # ...
  end

  private

  attr_reader :dep
end
```

2:18 PM · Jan 8, 2020 · Tweetbot for Mac

48 Retweets 144 Likes



Luca Guidi
@jodosha

There is another type of #Ruby object design that I talk less: data objects.

#OOP

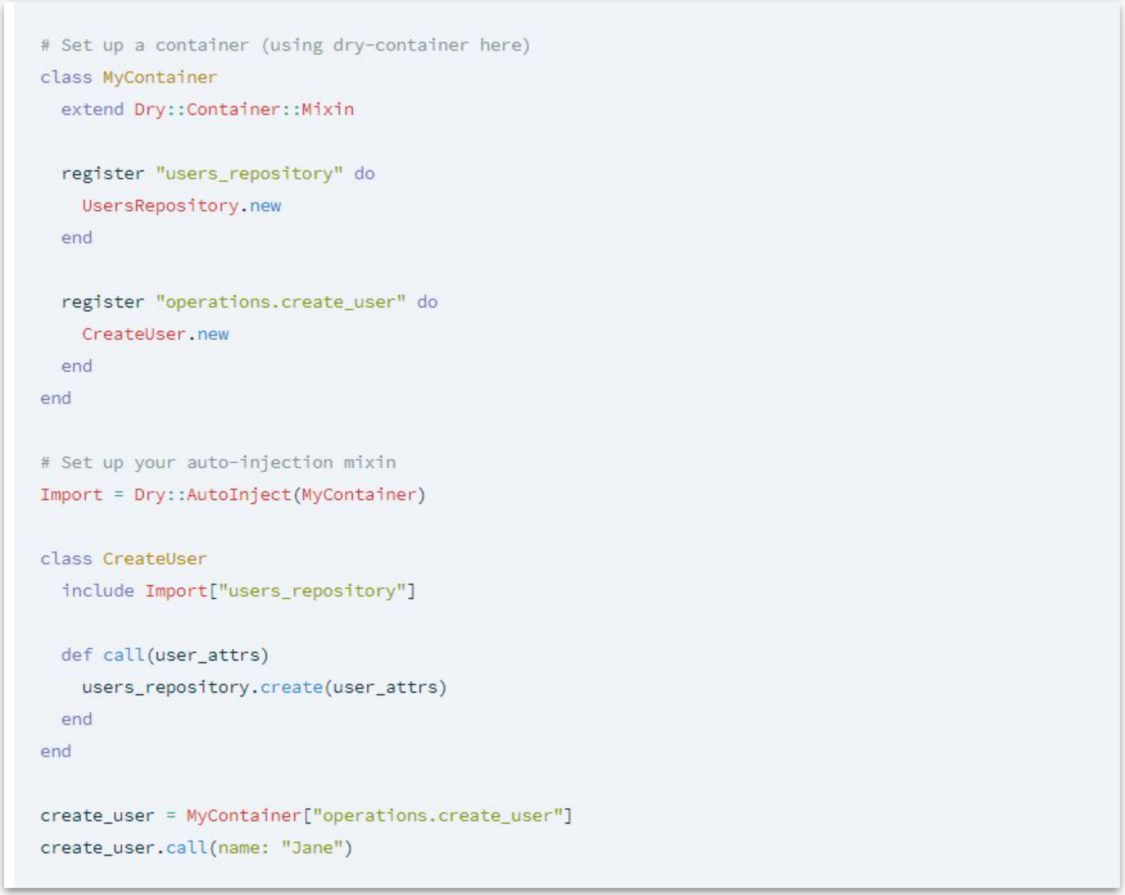
1/

```ruby
class MyPresenter
  def initialize(obj)
    @obj = obj
    freeze
  end

  def freeze
    @obj.freeze
    super
  end

  def full_name
    "#{@obj.first_name} #{@obj.last_name}"
  end
end
```

3:07 PM · Jan 10, 2020 · Tweetbot for Mac

5 Retweets 17 Likes

Dry-Auto_Inject



```ruby
# Set up a container (using dry-container here)
class MyContainer
  extend Dry::Container::Mixin

  register "users_repository" do
    UsersRepository.new
  end

  register "operations.create_user" do
    CreateUser.new
  end
end

# Set up your auto-injection mixin
Import = Dry::AutoInject(MyContainer)

class CreateUser
  include Import["users_repository"]

  def call(user_attrs)
    users_repository.create(user_attrs)
  end
end

create_user = MyContainer["operations.create_user"]
create_user.call(name: "Jane")
```

dry-auto_inject provides low-impact dependency injection and resolution support for your classes.

It's designed to work with a container that holds your application's dependencies. It works well with dry-container, but supports any container that responds to the `#[]` interface.

Dry-Transaction - Interactors - the old way

```ruby
require "dry/transaction"

class CreateUser
  include Dry::Transaction

  step :validate
  step :create

  private

  def validate(input)
    # returns Success(valid_data) or Failure(validation)
  end

  def create(input)
    # returns Success(user)
  end
end
```

Dry-monads Do - the new way

```ruby
require 'dry/monads'
require 'dry/monads/do'

class CreateAccount
  include Dry::Monads[:result]
  include Dry::Monads::Do.for(:call)

  def call(params)
    values = yield validate(params)
    account = yield create_account(values[:account])
    owner = yield create_owner(account, values[:owner])

    Success([account, owner])
  end

  def validate(params)
    # returns Success(values) or Failure(:invalid_data)
  end

  def create_account(account_values)
    # returns Success(account) or Failure(:account_not_created)
  end

  def create_owner(account, owner_values)
    # returns Success(owner) or Failure(:owner_not_created)
  end
end
```

Composing several monadic values can become tedious because you need to pass around unwrapped values in lambdas (aka blocks). Haskell was one of the first languages faced this problem. To work around it Haskell has a special syntax for combining monadic operations called the "do notation". If you're familiar with Scala it has `for`-comprehensions for a similar purpose. It is not possible to implement `do` in Ruby but it is possible to emulate it to some extent, i.e. achieve comparable usefulness.

What `Do` does is passing an unwrapping block to certain methods. The block tries to extract the underlying value from a monadic object and either short-circuits the execution (in case of a failure) or returns the unwrapped value back.

Dry-matcher

```ruby
require "dry/monads/result"
require "dry/matcher/result_matcher"

value = Dry::Monads::Success("success!")

result = Dry::Matcher::ResultMatcher.(value) do |m|
  m.success(Integer) do |i|
    "Got int: #{i}"
  end

  m.success do |v|
    "Yay: #{v}"
  end

  m.failure :not_found do |_err, reason|
    "Nope: #{reason}"
  end

  m.failure do |v|
    "Boo: #{v}"
  end
end

result # => "Yay: success!"
```

Result matcher

dry-matcher provides a ready-to-use `ResultMatcher` for working with `Result` or `Try` monads from dry-monads or any other compatible gems.

```ruby
value = Dry::Monads::Result::Failure.new([:invalid, :reasons])

Dry::Matcher::ResultMatcher.(value) do |m|
  m.success do |v|
    "Yay: #{v}"
  end

  m.failure(:not_found) do
    "No such thing"
  end

  m.failure(:invalid) do |_code, errors|
    "Cannot be done: #{errors.inspect}"
  end
end #=> "Cannot be done: :reasons"
```

Other dry-* gems used in the App

- Dry-View - generates HTML files (for our 2 sub apps)
- Dry-Validation - Validation layer (replace strong parameters and AM::Validator)
- Dry-Monads - Mondas for Interactor layer
- Dry-Matcher - reacts with Interactors (do monads); control flow results
- ...

ROM - Ruby Object Mapper

Ruby Object Mapper (ROM) is a *fast* ruby persistence library with the goal of providing powerful object mapping capabilities without limiting the full *power* of the underlying datastore.

More specifically, ROM exists to:

- Isolate the application from persistence details
- Provide minimum infrastructure for mapping and persistence
- Provide shared abstractions for lower-level components
- Provide simple use of *power* features offered by the datastore

Pros and cons

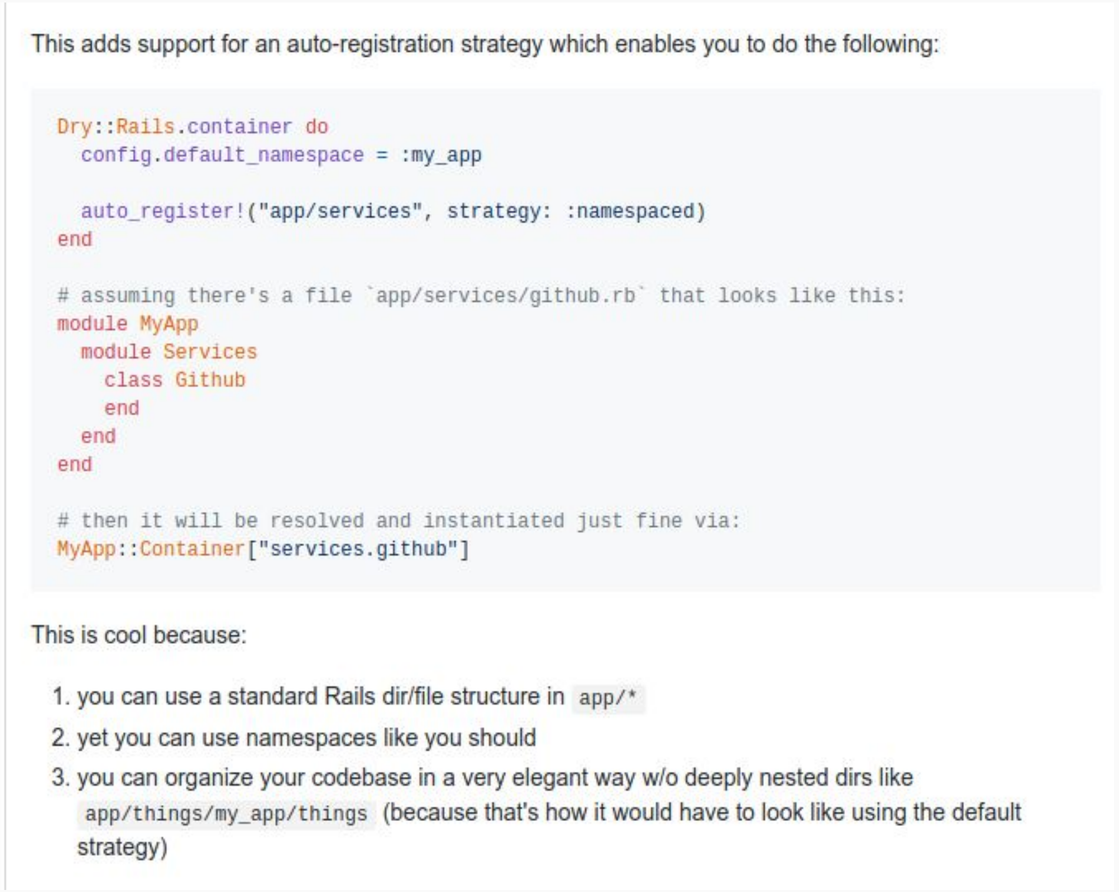
Pros:

- Cleaner, faster, easier to maintain code
- Great dynamic community (zulip, discord)
- Much easier, natural boundaries
- Pury ruby
- Easy to test any part of code (you can stub just a function inside container)
- A lot less objects created (less memory)
- More and more popular (screencasts, blog posts, Hanami 2 will use all of this gems)
- Joy of programming :)

Cons:

- Completely different approach (so hard to learn and understand at the beginning)
- Not so popular as rails (harder to find answers online)
- Namespacing (Containers fix this problem)
- Many gems you like to use might be rails-only (it change slowly, Hanami 2.0 will help a lot I guess)
- No so easy access to class methods (You cannot call in terminal User.all for example)

Dry-rails (https://github.com/dry-rb/dry-system-rails & https://github.com/dry-rb/dry-rails)



This adds support for an auto-registration strategy which enables you to do the following:

```ruby
Dry::Rails.container do
  config.default_namespace = :my_app

  auto_register!("app/services", strategy: :namespaced)
end

# assuming there's a file `app/services/github.rb` that looks like this:
module MyApp
  module Services
    class Github
    end
  end
end

# then it will be resolved and instantiated just fine via:
MyApp::Container["services.github"]
```

This is cool because:

1. you can use a standard Rails dir/file structure in `app/*`
2. yet you can use namespaces like you should
3. you can organize your codebase in a very elegant way w/o deeply nested dirs like `app/things/my_app/things` (because that's how it would have to look like using the default strategy)

Still very alpha version, but simplify using dry-* approach inside Rails project

Questions?