How to organize running Ruby Applications with Dry-System

Programming Help

Series "Development of an application with DRY-RB"

What is a DRY-RB, and how it will help with Ruby app where Ruby On Rails can help

How to organize running Ruby Applications with Dry-System

Apply Command pattern with DRY-TRANSACTION

How to organize running Ruby Applications with Dry-System

AUTHOR:

Rostislav Katin

In the first article, I told at a high level about how we use DRY-RB in our product.Now you can tell you more about the use of each of the libraries, and I will start with DRY-System.

Deciding, like we, leave Rails, you start a project on Sinatra or Roda. Most likely, even the first server launch will be a difficult task. Previously, Rails is carefully (but not very flexible) loaded all the hemes, ran into initializers, made the require of the entire code and many more things that you do not even suspect. And now you yourself are sitting and write files with dozens of require just to run your application. The same needs to be done in all files where some dependencies are used. Of course, it makes your application faster, but the pleasure of developing is sharply reduced.

These problems are designed to solve DRY-SYSTEM.It allows you to split the application on independent, isolated components, simplifies the initialization of the application, provides Inversion of Control (IOC) -Container to register and instance dependencies.Below, I will show on the example how can it be used in a real application.

In the course of the cycle, we will be iteratively to create a simple application: it will check stock courses and notify you if they exceed the specified threshold.In the first version, our application will have a kind of simple Rake Task, which will be launched as follows:

Bundle Exec Rake Check_price [Ticker Price]

Project preparation

At the root of the project, create three folders: System, Lib, Log and add gemfile:

Source 'http://rubygems.org' gem 'dry-system', '~> 0.10.1' gem 'Dry-Monitor', '~> 0.1.2' gem 'http' gem 'rake'

Container setup

Now you can go to the DRY-SYSTEM itself.The first thing we will configure is a container.In it, we will register the dependencies to which we will contact during the application.It will save us from the need to write many extra Require and makes possible control inversion.

Create a System / Container.rb file with the following contents:

Require 'Dry / System / Container' Require 'Dry / System / Components' Class Notifier <DRY :: System :: Container Use: Logging Use: ENV, Inferrer: -> {Env.Fetch ('Rack_env' ,: Development).to_sym} Configure Do | Config |# config.root = / root / app / dir config.auto_register = 'lib' end load_paths! ('lib', 'System') end

In this file, we describe the container how it must collect our application.

Now in order that here is happening:

These three lines are the connection of plug-ins for logging, determining the implementation environment and monitoring.

Use: Logging Use: ENV, INFERRER: -> {ENV.FETCH ('Rack_env' ,: Development) .to_sym} Use: Monitoring

We specify the root folder of the container, with respect to which it will resolve the dependency paths.In this case, I specifically commented it, since we do not need to change the value that is worth the default.But to know how the paths are allowed, useful.

# config.root = / root / app / dir

Here we specify the folder relative to the root, in which dependencies will be automatically recorded.This means that the require will be executed all files in this folder, and classes of them will be recorded on the keys corresponding to their paths.So, the Lib / Folder / class.rb class will be registered in the container as folder.class.Read more about how this mechanism works, I will tell below.

config.auto_register = 'lib'

This expression adds the folder transmitted in the arguments in $ Load_path to make it easier to make request files.

LOAD_PATHS! ('lib', 'System')

Now create a SYSTEM / IMPORT.RB file.With this module, we will access objects registered in the container.

Require 'Container' Import = notifier.injector

That's all, we described our container and how he must behave.Now you can run the container by calling the .finalize method on it.After startup, the autorecistration of the application classes will occur, third-party dependencies from the boot folder are initialized.For details on both these processes, it is written below, but for now you will start the container from IRB:

IRB (Main): 001: 0> Require_relative 'System / Container' => True IRB (Main): 002: 0> Notifier.finalize!=> Notifier

Constantly run the container in this way is not the most beautiful solution.Therefore, we will bring this logic to a separate System / Boot.rb file.Separately, I note that we do not only container, but also System / Import.rb file:

Require 'Bundler / Setup' Require_Relative 'Container' Require_Relative 'Import' Notifier.finalize!

Initialization of third-party dependencies

As a rule, the application before starting you need to initialize third-party libraries.To do this, create another SYSTEM / BOOT folder.It is an analogue of the Config / Initializers folder in Rails: after calling .finalize!The container is required by the request of all files from this folder, accordingly the code of all these files is performed.This is how the logger setting of our application in the System / Boot / Logger.rb file looks like:

Notifier.Boot: Logger Do Start Do | Container |Container [: Logger] .Level = Env.Fetch ('Log_Level', 1) End End

Application logic

So, our container is now initialized both application classes and third-party libraries.Now you can go to the description of the application logic itself.In this part, the implementation will be primitive, but in the next I will tell you how we will improve it.Here, nothing directly to the DRY-System does not apply, so I will leave it without explanation.

Learn more and hone your programming skills on Ruby you can with our mentors

Choose a mentor

LIB / notifier / fetch_price.rb

Require 'Http' Class FetchPrice Def Call (Ticker) Http.get (URL (Ticker). Yield_SELF Do | Response |Bigdecimal (response.to_s) End End Private Def URL (Ticker) "https://api.iextrading.com/1.0/stock/# /Ticker.upcase}/price" END END

LIB / notifier / check_price.rb

Class Checkprice Def Call (Price, Boundary) `Say" Price is # {Price}, Time To Buy "` if Price> Boundary End End

Now that we can learn the share price and compare it with a boundary value, you can write a managing class.This is how he may look like: lib / notifier / main.rb

Require 'Notifier / Check_price' Require 'Notifier / Fetch_price' Class Main Def Call (Ticker, Boundary) Price = fetchprice.new.call (Ticker) checkprice.new.call (Price, Boundary) End End

But so we would write this code if we did not have a container in which FetchPrice and CheckPrice classes (depending on the Main class) are already registered.Here is how the code looks like, if you rewrite it using the container:

Require 'Import' Class Main Include Import ['Notifier.check_price', 'Notifier.Fetch_price'] Def Call (Ticker, Price) Check_price.call (fetch_price.call (Ticker), Price) End End

Now about how it works: you are making an Import Module Include, and as an argument convey the dependency that you need.After this, the dependence automatically becomes available inside the class.

This principle is called Dependency Inversion.Thanks to it, you are published inside your code on the abstraction registered in the container, and not on specific implementation, as it was in the first version.It reduces connectivity, simplifies testing and generally simplifies support.It is he who is hiding under D in the famous Solid.

The last thing we left is to add Rake-Task, which will run our application.Create rakefile: rakefile

Desc 'Check Stock Price' Task: check_price,% i [Ticker Price] do | _task_name, args |Require_Relative './System/Boot' Ticker = Args [: Ticker] Price = Bigdecimal (Args [: Price]) Notifier ['notifier.main']. Call (Ticker, Price) END

Since we made the require file System / Boot, we do not have to explicitly run the container and make the request file System / import.rb in all classes where the Import module is used.

Check what we did:

Bundle Exec Rake Check_price [Googl, 1000] Bundle Exec Rake Check_price [Googl, 1200]

Now is the time to add to our application monitoring for confidence that the price is regularly checked.DRY-SYSTEM has a built-in monitoring mechanism based on Dry-Monitor.It allows you to configure monitoring objects in a container on the key.We modify our Rake Task to check the price in the eternal cycle and add monitoring of checks: Rakefile

Desc 'Check Stock Price' Task: check_price,% i [Ticker Price] do | _task_name, args |Require_Relative './System/Boot' Ticker = Args [: Ticker] Price = Bigdecimal (Args [: Price]) Notifier.Monitor ('Notifier.main') Do | Event |payload = event.payload notifier.logger.info "Price Checked with Args: # {payload [: args]} in # {payload [: Time]} MS" END LOOP DO Notifier ['notifier.main']. Call (Ticker, Price) Sleep 10 End End

That's all!Let's start Rake Task again and check the log / development.log file:

Price Checked With Args: ["Googl", 0.15E4] in 903.98ms Price Checked with Args: ["GOOGL", 0.15E4] IN 793.36MS ...

Additional reading

DRY-SYSTEM is a powerful tool with a multitude of functions that remained outside this article, so I strongly recommend reading the DRY-SYSTEM GEMA documentation.

On this I will finish, and in the next part I will tell you how we organize the Ruby application code using DRY-TRANSACTION.