

---

## Configatron



Configatron makes configuring your applications and scripts incredibly easy. No longer is there a need to use constants or global variables. Now you can use a simple and painless system to configure your life. And, because it's all Ruby, you can do any crazy thing you would like to!

One of the more important changes to V3 is that it now resembles more a [Hash](#) style interface. You can use `[]`, `fetch`, `each`, etc... Actually the hash notation is a bit more robust since the dot notation won't work for a few property names (a few public methods from `Configatron::Store` itself).

### Installation

Add this line to your application's Gemfile:

```
1 gem 'configatron'
```

And then execute:

```
1 $ bundle
```

Or install it yourself as:

```
1 $ gem install configatron --pre
```

### Usage

Once installed you just need to require it:

```
1 require 'configatron'
```

### Simple

```
1 configatron.email = 'me@example.com'
2 configatron.database.url = "postgres://localhost/foo"
```

Now, anywhere in your code you can do the following:

```
1 configatron.email # => "me@example.com"
2 configatron.database.url # => "postgres://localhost/foo"
```

---

Voila! Simple as can be.

Now you're saying, what if I want to have a 'default' set of options, but then override them later, based on other information? Simple again. Let's use our above example. We've configured our `database.url` option to be `postgres://localhost/foo`. The problem with that is that is our production database url, not our development url. Fair enough, all you have to do is redeclare it:

```
1 configatron.database.url = "postgres://localhost/foo_development"
```

becomes:

```
1 configatron.email # => "me@example.com"
2 configatron.database.url # => "postgres://localhost/foo_development"
```

Notice how our other configuration parameters haven't changed? Cool, eh?

## Hash/YAML

You can configure Configatron from a hash as well (this is particularly useful if you'd like to have configuration files):

```
1 configatron.configure_from_hash(email: {pop: {address: 'pop.example.com',
2     port: 110}}, smtp: {address: 'smtp.example.com'})
3 configatron.email.pop.address # => 'pop.example.com'
4 configatron.email.pop.port # => 110
5 # and so on...
```

## Method vs hash access

As a note, method (`configatron.foo`) access will be shadowed by public methods defined on the configatron object. (The configatron object descends from `BasicObject` and adds a few methods to resemble the `Hash` API and to play nice with `puts`, so it should have a pretty bare set of methods.)

If you need to use keys that are themselves method names, you can just use hash access (`configatron['foo']`).

## Namespaces

The question that should be on your lips is what I need to have namespaced configuration parameters. It's easy! Configatron allows you to create namespaces.

---

```
1 configatron.website_url = "http://www.example.com"
2 configatron.email.pop.address = "pop.example.com"
3 configatron.email.pop.port = 110
4 configatron.email.smtp.address = "smtp.example.com"
5 configatron.email.smtp.port = 25
6
7 configatron.to_h # => {:website_url=>"http://www.example.com", :email
  =>{:pop=>{:address=>"pop.example.com", :port=>110}, :smtp=>{:address
    =>"smtp.example.com", :port=>25}}}
```

Configatron allows you to nest namespaces to your hearts content! Just keep going, it's that easy.

Of course you can update a single parameter n levels deep as well:

```
1 configatron.email.pop.address = "pop2.example.com"
2
3 configatron.email.pop.address # => "pop2.example.com"
4 configatron.email.smtp.address # => "smtp.example.com"
```

Configatron will also let you use a block to clean up your configuration. For example the following two ways of setting values are equivalent:

```
1 configatron.email.pop.address = "pop.example.com"
2 configatron.email.pop.port = 110
3
4 configatron.email.pop do |pop|
5   pop.address = "pop.example.com"
6   pop.port = 110
7 end
```

## Temp Configurations

Sometimes in testing, or other situations, you want to temporarily change some settings. You can do this with the `temp` method (only available on the top-level configatron `RootStore`):

```
1 configatron.one = 1
2 configatron.letters.a = 'A'
3 configatron.letters.b = 'B'
4 configatron.temp do
5   configatron.letters.b = 'bb'
6   configatron.letters.c = 'c'
7   configatron.one # => 1
8   configatron.letters.a # => 'A'
9   configatron.letters.b # => 'bb'
10  configatron.letters.c # => 'c'
11 end
12 configatron.one # => 1
13 configatron.letters.a # => 'A'
```

---

```
14 configatron.letters.b # => 'B'
15 configatron.letters.c # => {}
```

## Delayed and Dynamic Configurations

There are times when you want to refer to one configuration setting in another configuration setting. Let's look at a fairly contrived example:

```
1 configatron.memcached.servers = ['127.0.0.1:11211']
2 configatron.page_caching.servers = configatron.memcached.servers
3 configatron.object_caching.servers = configatron.memcached.servers
4
5 if Rails.env == 'production'
6   configatron.memcached.servers = ['192.168.0.1:11211']
7   configatron.page_caching.servers = configatron.memcached.servers
8   configatron.object_caching.servers = configatron.memcached.servers
9 elsif Rails.env == 'staging'
10  configatron.memcached.servers = ['192.168.0.2:11211']
11  configatron.page_caching.servers = configatron.memcached.servers
12  configatron.object_caching.servers = configatron.memcached.servers
13 end
```

Now, we could've written that slightly differently, but it helps to illustrate the point. With Configatron you can create `Delayed` and `Dynamic` settings.

**Delayed** With `Delayed` settings execution of the setting doesn't happen until the first time it is executed.

```
1 configatron.memcached.servers = ['127.0.0.1:11211']
2 configatron.page_caching.servers = Configatron::Delayed.new {
3   configatron.memcached.servers
4 }
5 configatron.object_caching.servers = Configatron::Delayed.new {
6   configatron.memcached.servers
7 }
8
9 if Rails.env == 'production'
10  configatron.memcached.servers = ['192.168.0.1:11211']
11 elsif Rails.env == 'staging'
12  configatron.memcached.servers = ['192.168.0.2:11211']
13 end
```

Execution occurs once and after that the result of that execution is returned. So in our case the first time someone calls the setting `configatron.page_caching.servers` it will find the `configatron.memcached.servers` setting and return that. After that point if the `configatron.memcached.servers` setting is changed, the original settings are returned by `configatron.page_caching.servers`.

---

**Dynamic** `Dynamic` settings are very similar to `Delayed` settings, but with one big difference. Every time you call a `Dynamic` setting is executed. Take this example:

```
1 configatron.current.time = Configatron::Dynamic.new {Time.now}
```

Each time you call `configatron.current.time` it will return a new value to you. While this seems a bit useless, it is pretty useful if you have ever changing configurations.

## Resetting Configurations

In some testing scenarios, it can be helpful to restore Configatron to its default state. This can be done with:

```
1 configatron.reset!
```

## Checking keys

Even if parameters haven't been set, you can still call them, but you'll get a `Configatron::Store` object back. You can use `.has_key?` to determine if a key already exists.

```
1 configatron.i.dont.has_key?(:exist) # => false
```

**(key)!** You can also append a `!` to the end of any key. If the key exists it will return it, otherwise it will raise a `Configatron::UndefinedKeyError`.

```
1 configatron.a.b = 'B'
2 configatron.a.b # => 'B'
3 configatron.a.b! # => 'B'
4 configatron.a.b.c! # => raise Configatron::UndefinedKeyError
```

## Kernel

The `configatron` “helper” method is stored in the `Kernel` module. You can opt-out of this global monkey-patching by requiring `configatron/core` rather than `configatron`. You'll have to set up your own `Configatron::RootStore` object.

Example:

```
1 require 'configatron/core'
2
3 store = Configatron::RootStore.new
```

---

```
4 store.foo = 'FOO'
5
6 store.to_h #=> {foo: 'FOO'}
```

## Locking

Once you have setup all of your configurations you can call the `lock!` method to lock your settings and raise an error should anyone try to change settings or access an unset setting later.

Example:

```
1 configatron.foo = 'FOO'
2 configatron.lock!
3
4 configatron.foo # => 'FOO'
5
6 configatron.bar # => raises Configatron::UndefinedKeyError
7 configatron.bar = 'BAR' # => raises Configatron::LockedError
```

## Rails

Configatron works great with Rails. Use the built-in generate to generate an initializer file and a series of environment files for you to use to configure your applications.

```
1 $ rails generate configatron:install
```

Configatron will read in the `config/configatron/defaults.rb` file first and then the environment specific file, such as `config/configatron/development.rb`. Settings in the environment file will merge into and replace the settings in the `defaults.rb` file.

## Example

```
1 # config/configatron/defaults.rb
2 configatron.letters.a = 'A'
3 configatron.letters.b = 'B'
```

```
1 # config/configatron/development.rb
2 configatron.letters.b = 'BB'
3 configatron.letters.c = 'C'
```

```
1 configatron.to_h # => {:letters=>{:a=>"A", :b=>"BB", :c=>"C"}}
```

1. Fork it
2. Create your feature branch (`git checkout -b my-new-feature`)
3. Write Tests!
4. Commit your changes (`git commit -am 'Add some feature'`)
5. Push to the branch (`git push origin my-new-feature`)
6. Create new Pull Request

## Contributors

- Mark Bates
- Greg Brockman
- Kurtis Rainbolt-Greene
- Rob Sanheim
- Jérémy Lecour
- Cody Maggard
- Jean-Denis Vauguet
- chatgris
- Simon Menke
- Mat Brown
- Torsten Schönebaum
- Gleb Pomykalov
- Casper Gripenberg
- Artiom Diomin
- mattelacchiato
- Dan Pickett
- Tim Riley
- Rick Fletcher
- Jose Antonio Pio
- Brandon Dimcheff
- joe miller
- Josh Nichols