thoughtbot / factory_girl

Latest commit to the **master** branch

Factory evaluators use inheritance

commit f2e41389ea
 joshuaclayton authored January 03, 2012
factory_girl / GETTING_STARTED.md

- Fork and edit this file

100644 548 lines (398 sloc) 15.909 kb

- raw
- blame
- history

Getting Started

Update Your Gemfile

If you're using Rails, you'll need to change the required version of `factory_girl_rails`:

```
gem "factory_girl_rails", "~> 1.2"
```

If you're *not* using Rails, you'll just have to change the required version of `factory_girl`:

```
gem "factory_girl", "~> 2.1.0"
```

Once your Gemfile is updated, you'll want to update your bundle.

Defining factories

Each factory has a name and a set of attributes. The name is used to guess the class of the object by default, but it's possible to explicitly specify it:

```
# This will guess the User class
FactoryGirl.define do
  factory :user do
    first_name 'John'
    last_name  'Doe'
    admin false
  end

  # This will use the User class (Admin would have been guessed)
  factory :admin, :class => User do
    first_name 'Admin'
    last_name  'User'
    admin true
  end

  # The same, but using a string instead of class constant
  factory :admin, :class => 'user' do
    first_name 'Admin'
```

```
    last_name  'User'
    admin true
  end
end
```

It is highly recommended that you have one factory for each class that provides the simplest set of attributes necessary to create an instance of that class. If you're creating ActiveRecord objects, that means that you should only provide attributes that are required through validations and that do not have defaults. Other factories can be created through inheritance to cover common scenarios for each class.

Attempting to define multiple factories with the same name will raise an error.

Factories can be defined anywhere, but will be automatically loaded if they are defined in files at the following locations:

```
test/factories.rb
spec/factories.rb
test/factories/*.rb
spec/factories/*.rb
```

Using factories

factory_girl supports several different build strategies: build, create, attributes_for and stub:

```
# Returns a User instance that's not saved
user = FactoryGirl.build(:user)

# Returns a saved User instance
user = FactoryGirl.create(:user)

# Returns a hash of attributes that can be used to build a User instance
attrs = FactoryGirl.attributes_for(:user)

# Returns an object with all defined attributes stubbed out
stub = FactoryGirl.build_stubbed(:user)

# Passing a block to any of the methods above will yield the return object
FactoryGirl.create(:user) do |user|
  user.posts.create(attributes_for(:post))
end
```

No matter which strategy is used, it's possible to override the defined attributes by passing a hash:

```
# Build a User instance and override the first_name property
user = FactoryGirl.build(:user, :first_name => 'Joe')
user.first_name
# => "Joe"
```

If repeating "FactoryGirl" is too verbose for you, you can mix the syntax methods in:

```
# rspec
RSpec.configure do |config|
  config.include FactoryGirl::Syntax::Methods
end

# Test::Unit
class Test::Unit::TestCase
  include Factory::Syntax::Methods
```

```
end
```

This would allow you to write:

```
describe User, "#full_name" do
  subject { create(:user, :first_name => "John", :last_name => "Doe") }

  its(:full_name) { should == "John Doe" }
end
```

Lazy Attributes

Most factory attributes can be added using static values that are evaluated when the factory is defined, but some attributes (such as associations and other attributes that must be dynamically generated) will need values assigned each time an instance is generated. These "lazy" attributes can be added by passing a block instead of a parameter:

```
factory :user do
  # ...
  activation_code { User.generate_activation_code }
  date_of_birth   { 21.years.ago }
end
```

Aliases

Aliases allow you to use named associations more easily.

```
factory :user, :aliases => [:author, :commenter] do
  first_name    "John"
  last_name     "Doe"
  date_of_birth { 18.years.ago }
end

factory :post do
  author
  # instead of
  # association :author, :factory => :user
  title "How to read a book effectively"
  body  "There are five steps involved."
end

factory :comment do
  commenter
  # instead of
  # association :commenter, :factory => :user
  body "Great article!"
end
```

Dependent Attributes

Attributes can be based on the values of other attributes using the proxy that is yielded to lazy attribute blocks:

```
factory :user do
  first_name 'Joe'
  last_name  'Blow'
```

```
  email { "#{first_name}.#{last_name}@example.com".downcase }
end

FactoryGirl.create(:user, :last_name => 'Doe').email
# => "joe.doe@example.com"
```

Transient Attributes

There may be times where your code can be DRYed up by passing in transient attributes to factories.

```
factory :user do
  ignore do
    rockstar true
    upcased { false }
  end

  name  { "John Doe#{" - Rockstar" if rockstar}" }
  email { "#{name.downcase}@example.com" }

  after_create do |user, proxy|
    user.name.upcase! if proxy.upcased
  end
end

FactoryGirl.create(:user, :upcased => true).name
#=> "JOHN DOE - ROCKSTAR"
```

Static and dynamic attributes can be ignored. Ignored attributes will be ignored within attributes_for and won't be set on the model, even if the attribute exists or you attempt to override it.

Within Factory Girl's dynamic attributes, you can access ignored attributes as you would expect. If you need to access the proxy in a Factory Girl callback, you'll need to declare a second block argument (for the proxy) and access ignored attributes from there.

Associations

It's possible to set up associations within factories. If the factory name is the same as the association name, the factory name can be left out.

```
factory :post do
  # ...
  author
end
```

You can also specify a different factory or override attributes:

```
factory :post do
  # ...
  association :author, :factory => :user, :last_name => 'Writely'
end
```

The behavior of the association method varies depending on the build strategy used for the parent object.

```
# Builds and saves a User and a Post
post = FactoryGirl.create(:post)
post.new_record?        # => false
```

```
post.author.new_record? # => false

# Builds and saves a User, and then builds but does not save a Post
post = FactoryGirl.build(:post)
post.new_record?        # => true
post.author.new_record? # => false
```

To not save the associated object, specify :method => :build in the factory:

```
factory :post do
  # ...
  association :author, :factory => :user, :method => :build
end

# Builds a User, and then builds a Post, but does not save either
post = FactoryGirl.build(:post)
post.new_record?        # => true
post.author.new_record? # => true
```

Inheritance

You can easily create multiple factories for the same class without repeating common attributes by nesting factories:

```
factory :post do
  title 'A title'

  factory :approved_post do
    approved true
  end
end

approved_post = FactoryGirl.create(:approved_post)
approved_post.title # => 'A title'
approved_post.approved # => true
```

You can also assign the parent explicitly:

```
factory :post do
  title 'A title'
end

factory :approved_post, :parent => :post do
  approved true
end
```

As mentioned above, it's good practice to define a basic factory for each class with only the attributes required to create it. Then, create more specific factories that inherit from this basic parent. Factory definitions are still code, so keep them DRY.

Sequences

Unique values in a specific format (for example, e-mail addresses) can be generated using sequences. Sequences are defined by calling sequence in a definition block, and values in a sequence are generated by calling FactoryGirl.generate:

```
# Defines a new sequence
```

```
FactoryGirl.define do
  sequence :email do |n|
    "person#{n}@example.com"
  end
end

FactoryGirl.generate :email
# => "person1@example.com"

FactoryGirl.generate :email
# => "person2@example.com"
```

Sequences can be used as attributes:

```
factory :user do
  email
end
```

Or in lazy attributes:

```
factory :invite do
  invitee { FactoryGirl.generate(:email) }
end
```

And it's also possible to define an in-line sequence that is only used in a particular factory:

```
factory :user do
  sequence(:email) {|n| "person#{n}@example.com" }
end
```

You can also override the initial value:

```
factory :user do
  sequence(:email, 1000) {|n| "person#{n}@example.com" }
end
```

Without a block, the value will increment itself, starting at its initial value:

```
factory :post do
  sequence(:position)
end
```

Traits

Traits allow you to group attributes together and then apply them to any factory.

```
factory :user, :aliases => [:author]

factory :story do
  title "My awesome story"
  author

  trait :published do
    published true
  end

  trait :unpublished do
    published false
```

```
  end

  trait :week_long_publishing do
    start_at { 1.week.ago }
    end_at   { Time.now }
  end

  trait :month_long_publishing do
    start_at { 1.month.ago }
    end_at   { Time.now }
  end

  factory :week_long_published_story,    :traits => [:published, :week_long_publishing]
  factory :month_long_published_story,   :traits => [:published, :month_long_publishing]
  factory :week_long_unpublished_story,  :traits => [:unpublished, :week_long_publishing]
  factory :month_long_unpublished_story, :traits => [:unpublished, :month_long_publishing]
end
```

Traits can be used as attributes:

```
factory :week_long_published_story_with_title, :parent => :story do
  published
  week_long_publishing
  title { "Publishing that was started at {start_at}" }
end
```

Traits that define the same attributes won't raise AttributeDefinitionErrors; the trait that defines the attribute latest gets precedence.

```
factory :user do
  name "Friendly User"
  login { name }

  trait :male do
    name   "John Doe"
    gender "Male"
    login { "#{name} (M)" }
  end

  trait :female do
    name   "Jane Doe"
    gender "Female"
    login { "#{name} (F)" }
  end

  trait :admin do
    admin true
    login { "admin-#{name}" }
  end

  factory :male_admin,   :traits => [:male, :admin]   # login will be "admin-John Doe"
  factory :female_admin, :traits => [:admin, :female] # login will be "Jane Doe (F)"
end
```

You can also override individual attributes granted by a trait in subclasses.

```
factory :user do
  name "Friendly User"
  login { name }

  trait :male do
```

```
    name   "John Doe"
    gender "Male"
    login { "#{name} (M)" }
  end

  factory :brandon do
    male
    name "Brandon"
  end
end
```

Traits can also be passed in as a list of symbols when you construct an instance from FactoryGirl.

```
factory :user do
  name "Friendly User"

  trait :male do
    name   "John Doe"
    gender "Male"
  end

  trait :admin do
    admin true
  end
end

# creates an admin user with gender "Male" and name "Jon Snow"
FactoryGirl.create(:user, :admin, :male, :name => "Jon Snow")
```

This ability works with `build`, `build_stubbed`, `attributes_for`, and `create`.

Callbacks

factory_girl makes available three callbacks for injecting some code:

- after_build - called after a factory is built (via FactoryGirl.build)
- after_create - called after a factory is saved (via FactoryGirl.create)
- after_stub - called after a factory is stubbed (via FactoryGirl.stub)

Examples:

```
# Define a factory that calls the generate_hashed_password method after it is built
factory :user do
  after_build { |user| generate_hashed_password(user) }
end
```

Note that you'll have an instance of the user in the block. This can be useful.

You can also define multiple types of callbacks on the same factory:

```
factory :user do
  after_build  { |user| do_something_to(user) }
  after_create { |user| do_something_else_to(user) }
end
```

Factories can also define any number of the same kind of callback. These callbacks will be executed in the order they are specified:

```
factory :user do
  after_create { this_runs_first }
  after_create { then_this }
end
```

Calling FactoryGirl.create will invoke both after_build and after_create callbacks.

Also, like standard attributes, child factories will inherit (and can also define) callbacks from their parent factory.

Modifying factories

If you're given a set of factories (say, from a gem developer) but want to change them to fit into your application better, you can modify that factory instead of creating a child factory and adding attributes there.

If a gem were to give you a User factory:

```
FactoryGirl.define do
  factory :user do
    full_name "John Doe"
    sequence(:username) {|n| "user#{n}" }
    password "password"
  end
end
```

Instead of creating a child factory that added additional attributes:

```
FactoryGirl.define do
  factory :application_user, :parent => :user do
    full_name     { Faker::Name.name }
    date_of_birth { 21.years.ago }
    gender        "Female"
    health        90
  end
end
```

You could modify that factory instead.

```
FactoryGirl.modify do
  factory :user do
    full_name     { Faker::Name.name }
    date_of_birth { 21.years.ago }
    gender        "Female"
    health        90
  end
end
```

When modifying a factory, you can change any of the attributes you want (aside from callbacks).

`FactoryGirl.modify` must be called outside of a `FactoryGirl.define` block as it operates on factories differently.

A caveat: you can only modify factories (not sequences or traits) and callbacks *still compound as they normally would*. So, if the factory you're modifying defines an `after_create` callback, you defining an `after_create` won't override it, it'll just get run after the first callback.

Building or Creating Multiple Records

Sometimes, you'll want to create or build multiple instances of a factory at once.

```
built_users   = FactoryGirl.build_list(:user, 25)
created_users = FactoryGirl.create_list(:user, 25)
```

These methods will build or create a specific amount of factories and return them as an array. To set the attributes for each of the factories, you can pass in a hash as you normally would.

```
twenty_year_olds = FactoryGirl.build_list(:user, 25, :date_of_birth => 20.years.ago)
```

Cucumber Integration

factory_girl ships with step definitions that make calling factories from Cucumber easier. To use them, add the following to features/support/env.rb:

```
require 'factory_girl/step_definitions'
```

Alternate Syntaxes

Users' tastes for syntax vary dramatically, but most users are looking for a common feature set. Because of this factory_girl supports "syntax layers" which provide alternate interfaces. See Factory::Syntax for information about the various layers available. For example, the Machinist-style syntax is popular:

```
require 'factory_girl/syntax/blueprint'
require 'factory_girl/syntax/make'
require 'factory_girl/syntax/sham'

Sham.email {|n| "#{n}@example.com" }

User.blueprint do
  name  { 'Billy Bob' }
  email { Sham.email  }
end

User.make(:name => 'Johnny')
```

- Gauges: Analyze web traffic
- Speaker Deck: Presentations
- Gist: Code snippets
- GitHub for Mac
- Issues for iPhone
- Job Board

Extras

- GitHub Shop
- The Octodex

Documentation

- GitHub Help
- Developer API
- GitHub Flavored Markdown
- GitHub Pages

- Terms of Service
- Privacy
- Security


Powered by the Dedicated Servers and Cloud Computing of Rackspace Hosting®

Markdown Cheat Sheet

Format Text

Headers

```
# This is an <h1> tag
## This is an <h2> tag
###### This is an <h6> tag
```

Text styles

```
*This text will be italic*
_This will also be italic_
**This text will be bold**
__This will also be bold__

*You **can** combine them*
```

Lists

Unordered

```
* Item 1
* Item 2
  * Item 2a
  * Item 2b
```

Ordered

```
1. Item 1
2. Item 2
3. Item 3
   * Item 3a
   * Item 3b
```

Miscellaneous

Images

```
![GitHub Logo](/images/logo.png)
Format: ![Alt Text](url)
```

Links

```
http://github.com - automatic!
[GitHub](http://github.com)
```

Blockquotes

```
As Kanye West said:

> We're living the future so
> the present is our past.
```

Code Examples in Markdown

Syntax highlighting with GFM

````
```javascript
function fancyAlert(arg) {
  if(arg) {
    $.facebox({div:'#foo'})
  }
}
```
````

Or, indent your code 4 spaces

```
Here is a Python code example
without syntax highlighting:

    def foo:
      if not bar:
        return true
```

Inline code for comments

```
I think you should use an
`<addr>` element here instead.
```

Something went wrong with that request. Please try again. Dismiss