RSpec, TDD, FactoryGirl

Devpoint Labs - Dave Jungst / Jake Sorce

BDD / TDD

Behavior Driven Development
Test Driven Development

Up to this point we have been focusing on RSpec syntax and not as much on BDD.

From this point forward we want to start describing our specifications before they are implemented.

BEFORE

1. Generate model / controller
2. Write controller and model methods
3. Write specs around existing models and specs

Problems with this approach:

- White box testing
- Lack of app planning
- Edge cases missed

AFTER

1. Write specs
2. Generate models / controllers
3. Write methods until they pass specifications

This leads to:

- Black / grey box testing
- Better planning
- More coverage
- Incremental development

Create a new rails app

Add RSpec & friends to Gemfile

Install RSpec

```
bundle
```

```
rails g model robot name:string friendly:boolean
```

```
bundle exec rake db:create db:migrate
```

After running this spec it should fail because the wave method is not implemented. The wave method can now be built exactly to the spec and modified until the spec passes.

spec/models/robot_spec.rb

```
RSpec.describe Robot, type: :model do
  describe ‘mechanics’ do
    it ‘says hello if friendly’ do
      robot = Robot.create(name: ‘bob’, friendly: true)
      expect(robot.greet).to eq(‘hello’)
    end
  end
end
```

BDD (cont)

Once all of the specs are written for the Robot model and all of the methods are implemented, this pattern can be followed for the controller.

At this point a full “Feature” has been built with 100% test coverage.

On a side note by “describing” the functionality through RSpec it should be easy to see what a method is supposed to do when you come back to the code days / weeks / months even years later.

FactoryGirl

FactoryGirl is a fixture replacement that allows you to keep objects flexible and dry with or without creating objects in the database. FactoryGirl fixtures are extremely flexible and ensure you have valid objects.

Benefits

- Less maintenance than boilerplate or fixtures
- DRY code
- Adding attributes in a previous merge is no longer a problem as FactoryGirl is already aware of them
- Easy to pull in dependant fixtures
- As the app grows in size code reuse is pivotal

You can realize tremendous gains from FactoryGirl without being an expert. The gains made from utilizing the basics alone should be enough to get anyone excited about FactoryGirl

There are a few steps required to start using FactoryGirl

```
group :development, :test do
  ….
  gem ‘factory_girl_rails’
end

bundle
```

configure FactoryGirl in rails_helper

With the factory girl Gem installed you can now simply create factories which will be located in spec/factories/ and begin using them in specs.

In the next few slides we will:

1. Create a rails app
2. Install RSpec and FactoryGirl
3. Generate a model
4. Write a spec without using FactoryGirl
5. Create a factory
6. Refactor our specs using FactoryGirl
7. Keel over from happiness!

Build the rails app and configure

```
rails new robots -d postgresql -T

Gemfile.rb
group :development, :test do
  gem 'rspec-rails'
  gem 'shoulda-matchers'
  gem 'simplecov'
  gem 'factory_girl_rails'
end

bundle

rails g rspec:install

rails g model Robot name:string serial:string friendly:boolean

bundle exec rake db:create db:migrate
```

```
spec/rails_helper.rb
require 'simplecov'
SimpleCov.start

require 'factory_girl_rails'
RSpec.configure do |config|
  config.include FactoryGirl::Syntax::Methods
end
```

Write specs without FactoryGirl

spec/models/robot_spec.rb

```
require ‘rails_helper’

RSpec.describe Robot, type: :model do

  describe ‘abilities’ do
    it ‘greets humans if friendly’ do
      robot = Robot.create(name: ‘Bob’, serial: ‘1234’, friendly: true)
      expect(robot.greet).to eq(‘Hello’)
    end

    it ‘attacks humans if not friendly’ do
      robot = Robot.create(name: ‘Bob’, serial: ‘1234’, friendly: false)
      expect(robot.attack).to eq(true)
    end
  end
end
```

At first glance these specs don’t seem too bad but if they became any more complex or we needed multiple Robots for a spec or there were dependent relationships then the setup would be the majority of the spec

Write specs without FactoryGirl

spec/factories/robot.rb

```
FactoryGirl.define do

  factory :robot, class: Robot  do
    name ‘Bob’
    serial ‘12345’
    friendly: true
  end
end
```

In this file a robot factory has been defined with default values.

Whenever we call FactoryGirl.create(:robot)
A robot will be created with the default values

A friendly robot named Bob with a serial number of 1234

It’s easy to override default values:

FactoryGirl.create(:robot, friendly: false)

This will create an unfriendly robot named Bob with a serial number with 1234

FactoryGirl.create(:robot, friendly: false, name: ‘Steve’)

Now there is an unfriendly robot named steve with a serial number of 1234

Refactoring with FactoryGirl (cont)

spec/models/robot_spec.rb

```
require ‘rails_helper’

RSpec.describe Robot, type: :model do

  describe ‘abilities’ do
    it ‘greets humans if friendly’ do
      robot = Robot.create(name: ‘Bob’, serial: ‘1234’, friendly: true)
      expect(robot.greet).to eq(‘Hello’)
    end

    it ‘attacks humans if not friendly’ do
      robot = Robot.create(name: ‘Bob’, serial: ‘1234’, friendly: false)
      expect(robot.attack).to eq(true)
    end
  end
end
```

```
it ‘greets humans if friendly’ do
  expect(FactoryGirl.create(:robot).greet).to eq(‘Hello’)
 end

 it ‘attacks humans if not friendly’ do
   robot = FactoryGirl.create(:robot, friendly: false)
   expect(robot.attack).to eq(true)
end
```

Associations

spec/factories/robot.rb

```
FactoryGirl.define do

  factory :robot, class: Robot  do
    name ‘Bob’
    serial ‘12345’
    friendly: true
    inventor
  end
end
```

spec/factories/inventor.rb

```
FactoryGirl.define do

  factory :inventor. class: inventor do
    name ‘Winston’
  end
end
```

In this case a robot belongs to an inventor and an inventor has many robots

When factory girl is used to create an inventor it will also create a robot that belongs to the inventor

FactoryGirl.create(:robot)

Robot => id: 1, inventor_id: 1, name: ‘Bob’...
Inventor => id: 1, name: ‘Winston’

FactoryGirl methods

```
FactoryGirl.build(:robot)  #Returns a robot instance that is not saved

FactoryGirl.create(:robot)  #Returns a robot instance that is saved

FactoryGirl.attributes_for(:robot) #Returns a hash of attributes

FactoryGirl.build_stubbed(:robot)  #Creates a stubbed out factory

FactoryGirl.create(:robot) do |robot|
  robot.parts(attributes_for(:parts))
end
#You can pass a block to factory girl and it will return the yield object
```

Lazy Attributes

Some attributes need to have values assigned when an instance is generated.

This is accomplished by passing a block to the factory

```
factory :robot, class: Robot do
  passphrase { Robot.generate_passphrase }
  service_date { 30.days.ago }
end
```

Dependent Attributes

Attributes can be dependent on other attributes at time of creation

```
factory :inventor, class: Inventor do
  first_name: ‘Don’
  last_name: ‘Donaldson’
  email { “#{first_name}.#{last_name}@robotbuilder.com”.downcase }
end

create(:inventor, last_name: ‘Smith’).email
  => “don.smith@robotbuilder.com”
```

Sequences

Sequences are a nice way to create unique values. If there was a unique email validation on creators, you could either pass in a new value every time or just let a sequence take care of it for you.

```
FactoryGirl.define do
  sequence :email do |n|
    “creator#{n}@robotbuilder.com”
  end
end
```

```
generate :email
  => “creator1@robotbulder.com”

generate :email
  => “creator2@robotbuilder.com”
```

Multiple records with lists

It is possible to build multiple factories in one call, the return will be an array of objects

```
robots = build_list(:robot, 100)  #builds 100 robots
robots = create_list(:robot, 100)  #creates 100 robots

You can still pass attributes

robots_due_for_service = create_list(:robot, 50, last_service: 1.year.ago)
#creates 50 robots whose last service was 1 year ago
```

Transient Attributes

Transient attributes can help to DRY up code

```
factory :robot, class: Robot do
  transient do
    good_robot true
   needs_service false
  end

  name { “Bob#{“ - Good Robot” if good_robot}” }
  message { “SERVICE NOW” if needs_service }
end
```

Callbacks

Callbacks work the same way as you would expect in rails

```
after(:build) #called after the factory is built
before(:create) #called before the factory is created
after(:create) #called after the factory is created
after(:stub) #called after a factory is stubbed

factory :robot, class: Robot do
  after(:build) { |robot| generate_random_serial_number(robot) }
end
```

Traits

Traits can be added to factories to give more flexibility

```
factory :robot, class: Robot do
  name ‘Bob”

  trait :good do
    name “Good Guy Bob”
    friendly true
  end

  trait :bad do
    name “Evil Bob”
    friendly false
  end
end
```

```
create(:robot)
  => name: Bob, friendly: nil

create(:robot, :good)
  => name: ‘Good Guy Bob’, friendly: true

create(:robot, :bad)
  => name: ‘Evil Bob’, friendly: false
```

before, after, let hooks

RSpec helpers

DRYing up specs by putting repetitive code in before hooks and cleaning up in after hoks.

Using let to instantiate objects on the fly

Callbacks

```
before(:each)
  => Executes before every spec in the block

before(:all)
  => Executes once before the block

after(:each)
  => Executes after each spec in the block

after(:all)
  => Executes once after the entire block
```

Callbacks

```
describe ‘users’ do
  it ‘speaks’ do
    @user = User.create(....)
    expect(@user.speak).to eq(...)
  end

  it ‘waves’ do
     @user = User.create(...)
     expect(@user.wave).to eq(...)
  end
end
```

```
describe ‘users’ do
  before(:each) do
   @user = User.create(...)
  end

  it ‘speaks’ do
    expect(@user.speak).to eq(...)
  end

  it ‘waves’ do
    expect(@user.wave).to eq(...)
  end
end
```

@user will now be created and set before each it block in the ‘users’ describe block

Let

let is an RSpec method that allows you to define a helper method that will be cached throughout an example. let is commonly used with FactoryGirl

let is called by passing in a symbolized helper method name that is defined on the fly

let(:user) would create a helper method named user

let takes a block to define the helper method

let(:user) { FactoryGirl.create(:user) }

Now when user is called in an example the first time it will create a user factory and when it is called in that same example it retrieves the instance of the factory instead of creating another

```
RSpec.describe User, type: :model do
  let(:user) { FactoryGirl.create(:user) }
  describe ‘vocal methods’ do
    it ‘yells’ do
      expect(user.yell).to eq(“Ahhhh”)
    end
  end
end
```

Calling user in the expect actually creates a user factory.

If we were to call user again in the same block it would return an instance of the factory that was created.