The Basics of Creating Rails Plugins

A Rails plugin is either an extension or a modification of the core framework. Plugins provide:

- A way for developers to share bleeding-edge ideas without hurting the stable code base.
- A segmented architecture so that units of code can be fixed or updated on their own release schedule.
- An outlet for the core developers so that they don't have to include every cool new feature under the sun.

After reading this guide, you will know:

- How to create a plugin from scratch.
- How to write and run tests for the plugin.

This guide describes how to build a test-driven plugin that will:

- Extend core Ruby classes like Hash and String.
- Add methods to `ApplicationRecord` in the tradition of the `acts_as` plugins.
- Give you information about where to put generators in your plugin.

For the purpose of this guide pretend for a moment that you are an avid bird watcher. Your favorite bird is the Yaffle, and you want to create a plugin that allows other developers to share in the Yaffle goodness.

Setup

Currently, Rails plugins are built as gems, *gemified plugins*. They can be shared across different Rails applications using RubyGems and Bundler if desired.

Generate a gemified plugin.

Rails ships with a `rails plugin new` command which creates a skeleton for developing any kind of Rails extension with the ability to run integration tests using a dummy Rails application. Create your plugin with the command:

```
$ rails plugin new yaffle
```

See usage and options by asking for help:

```
$ rails plugin new --help
```

Testing Your Newly Generated Plugin

You can navigate to the directory that contains the plugin, run the `bundle install` command and run the one generated test using the `bin/test` command.

You should see:

```
  1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

This will tell you that everything got generated properly, and you are ready to start adding functionality.

Extending Core Classes

This section will explain how to add a method to String that will be available anywhere in your Rails application.

In this example you will add a method to String named `to_squawk`. To begin, create a new test file with a few assertions:

```ruby
# yaffle/test/core_ext_test.rb

require "test_helper"

class CoreExtTest < ActiveSupport::TestCase
  def test_to_squawk_prepends_the_word_squawk
    assert_equal "squawk! Hello World", "Hello World".to_squawk
  end
end
```

Run `bin/test` to run the test. This test should fail because we haven't implemented the `to_squawk` method:

```
E

Error:
CoreExtTest#test_to_squawk_prepends_the_word_squawk:
NoMethodError: undefined method `to_squawk' for "Hello World":String


bin/test /path/to/yaffle/test/core_ext_test.rb:4

.

Finished in 0.003358s, 595.6483 runs/s, 297.8242 assertions/s.

2 runs, 1 assertions, 0 failures, 1 errors, 0 skips
```

Great - now you are ready to start development.

In `lib/yaffle.rb`, add `require "yaffle/core_ext"`:

```ruby
# yaffle/lib/yaffle.rb

require "yaffle/railtie"
require "yaffle/core_ext"

module Yaffle
  # Your code goes here...
end
```

Finally, create the `core_ext.rb` file and add the `to_squawk` method:

```ruby
# yaffle/lib/yaffle/core_ext.rb

class String
  def to_squawk
    "squawk! #{self}".strip
  end
end
```

To test that your method does what it says it does, run the unit tests with `bin/test` from your plugin directory.

```
  2 runs, 2 assertions, 0 failures, 0 errors, 0 skips
```

To see this in action, change to the `test/dummy` directory, start `bin/rails console`, and commence squawking:

```
irb> "Hello World".to_squawk
=> "squawk! Hello World"
```

Add an "acts_as" Method to Active Record

A common pattern in plugins is to add a method called `acts_as_something` to models. In this case, you want to write a method called `acts_as_yaffle` that adds a `squawk` method to your Active Record models.

To begin, set up your files so that you have:

```ruby
# yaffle/test/acts_as_yaffle_test.rb

require "test_helper"

class ActsAsYaffleTest < ActiveSupport::TestCase
end
```

```ruby
# yaffle/lib/yaffle.rb

require "yaffle/railtie"
require "yaffle/core_ext"
require "yaffle/acts_as_yaffle"
```

```ruby
module Yaffle
  # Your code goes here...
end
```

```ruby
# yaffle/lib/yaffle/acts_as_yaffle.rb

module Yaffle
  module ActsAsYaffle
  end
end
```

Add a Class Method

This plugin will expect that you've added a method to your model named `last_squawk`. However, the plugin users might have already defined a method on their model named `last_squawk` that they use for something else. This plugin will allow the name to be changed by adding a class method called `yaffle_text_field`.

To start out, write a failing test that shows the behavior you'd like:

```ruby
# yaffle/test/acts_as_yaffle_test.rb

require "test_helper"

class ActsAsYaffleTest < ActiveSupport::TestCase
  def test_a_hickwalls_yaffle_text_field_should_be_last_squawk
    assert_equal "last_squawk", Hickwall.yaffle_text_field
  end

  def test_a_wickwalls_yaffle_text_field_should_be_last_tweet
    assert_equal "last_tweet", Wickwall.yaffle_text_field
  end
end
```

When you run `bin/test`, you should see the following:

```
# Running:

..E

Error:
ActsAsYaffleTest#test_a_wickwalls_yaffle_text_field_should_be_last_tweet:
NameError: uninitialized constant ActsAsYaffleTest::Wickwall


bin/test /path/to/yaffle/test/acts_as_yaffle_test.rb:8
```

```
E

Error:
ActsAsYaffleTest#test_a_hickwalls_yaffle_text_field_should_be_last_squawk:
NameError: uninitialized constant ActsAsYaffleTest::Hickwall


bin/test /path/to/yaffle/test/acts_as_yaffle_test.rb:4



Finished in 0.004812s, 831.2949 runs/s, 415.6475 assertions/s.

4 runs, 2 assertions, 0 failures, 2 errors, 0 skips
```

This tells us that we don't have the necessary models (Hickwall and Wickwall) that we are trying to test. We can easily generate these models in our "dummy" Rails application by running the following commands from the `test/dummy` directory:

```
$ cd test/dummy
$ bin/rails generate model Hickwall last_squawk:string
$ bin/rails generate model Wickwall last_squawk:string last_tweet:string
```

Now you can create the necessary database tables in your testing database by navigating to your dummy app and migrating the database. First, run:

```
$ cd test/dummy
$ bin/rails db:migrate
```

While you are here, change the Hickwall and Wickwall models so that they know that they are supposed to act like yaffles.

```ruby
# test/dummy/app/models/hickwall.rb

class Hickwall < ApplicationRecord
  acts_as_yaffle
end
```

```ruby
# test/dummy/app/models/wickwall.rb

class Wickwall < ApplicationRecord
  acts_as_yaffle yaffle_text_field: :last_tweet
end
```

We will also add code to define the `acts_as_yaffle` method.

```ruby
# yaffle/lib/yaffle/acts_as_yaffle.rb
```

```ruby
module Yaffle
  module ActsAsYaffle
    extend ActiveSupport::Concern

    class_methods do
      def acts_as_yaffle(options = {})
      end
    end
  end
end
```

```ruby
# test/dummy/app/models/application_record.rb

class ApplicationRecord < ActiveRecord::Base
  include Yaffle::ActsAsYaffle

  self.abstract_class = true
end
```

You can then return to the root directory (`cd ../..`) of your plugin and rerun the tests using `bin/test`.

```
# Running:

.E

Error:
ActsAsYaffleTest#test_a_hickwalls_yaffle_text_field_should_be_last_squawk:
NoMethodError: undefined method `yaffle_text_field' for #<Class:0x0055974ebbe9d8>


bin/test /path/to/yaffle/test/acts_as_yaffle_test.rb:4

E

Error:
ActsAsYaffleTest#test_a_wickwalls_yaffle_text_field_should_be_last_tweet:
NoMethodError: undefined method `yaffle_text_field' for #<Class:0x0055974eb8cfc8>


bin/test /path/to/yaffle/test/acts_as_yaffle_test.rb:8

.

Finished in 0.008263s, 484.0999 runs/s, 242.0500 assertions/s.

4 runs, 2 assertions, 0 failures, 2 errors, 0 skips
```

Getting closer... Now we will implement the code of the `acts_as_yaffle` method to make the tests pass.

```
# yaffle/lib/yaffle/acts_as_yaffle.rb

module Yaffle
  module ActsAsYaffle
    extend ActiveSupport::Concern

    class_methods do
      def acts_as_yaffle(options = {})
        cattr_accessor :yaffle_text_field, default: (options[:yaffle_text_field] || :last_s
      end
    end
  end
end
```

```
# test/dummy/app/models/application_record.rb

class ApplicationRecord < ActiveRecord::Base
  include Yaffle::ActsAsYaffle

  self.abstract_class = true
end
```

When you run `bin/test`, you should see the tests all pass:

```
  4 runs, 4 assertions, 0 failures, 0 errors, 0 skips
```

Add an Instance Method

This plugin will add a method named 'squawk' to any Active Record object that calls `acts_as_yaffle`. The 'squawk' method will simply set the value of one of the fields in the database.

To start out, write a failing test that shows the behavior you'd like:

```
# yaffle/test/acts_as_yaffle_test.rb
require "test_helper"

class ActsAsYaffleTest < ActiveSupport::TestCase
  def test_a_hickwalls_yaffle_text_field_should_be_last_squawk
    assert_equal "last_squawk", Hickwall.yaffle_text_field
  end

  def test_a_wickwalls_yaffle_text_field_should_be_last_tweet
    assert_equal "last_tweet", Wickwall.yaffle_text_field
  end
  def test_hickwalls_squawk_should_populate_last_squawk
    hickwall = Hickwall.new
    hickwall.squawk("Hello World")
    assert_equal "squawk! Hello World", hickwall.last_squawk
  end

  def test_wickwalls_squawk_should_populate_last_tweet
    wickwall = Wickwall.new
    wickwall.squawk("Hello World")
    assert_equal "squawk! Hello World", wickwall.last_tweet
  end
end
```

Run the test to make sure the last two tests fail with an error that contains “NoMethodError: undefined method ‘squawk’”, then update `acts_as_yaffle.rb` to look like this:

```
# yaffle/lib/yaffle/acts_as_yaffle.rb

module Yaffle
  module ActsAsYaffle
    extend ActiveSupport::Concern

    included do
      def squawk(string)
        write_attribute(self.class.yaffle_text_field, string.to_squawk)
      end
    end

    class_methods do
      def acts_as_yaffle(options = {})
        cattr_accessor :yaffle_text_field, default: (options[:yaffle_text_field] || :last_s
      end
    end
  end
end
```

```
# test/dummy/app/models/application_record.rb

class ApplicationRecord < ActiveRecord::Base
  include Yaffle::ActsAsYaffle

  self.abstract_class = true
end
```

Run `bin/test` one final time, and you should see:

```
  6 runs, 6 assertions, 0 failures, 0 errors, 0 skips
```

NOTE: The use of `write_attribute` to write to the field in model is just one example of how a plugin can interact with the model, and will not always be the right method to use. For example, you could also use:

```
send("#{self.class.yaffle_text_field}=", string.to_squawk)
```

Generators

Generators can be included in your gem simply by creating them in a `lib/generators` directory of your plugin. More information about the creation of generators can be found in the Generators Guide.

Publishing Your Gem

Gem plugins currently in development can easily be shared from any Git repository. To share the Yaffle gem with others, simply commit the code to a Git repository (like GitHub) and add a line to the `Gemfile` of the application in question:

```
gem "yaffle", git: "https://github.com/rails/yaffle.git"
```

After running `bundle install`, your gem functionality will be available to the application.

When the gem is ready to be shared as a formal release, it can be published to RubyGems.

Alternatively, you can benefit from Bundler's Rake tasks. You can see a full list with the following:

```
$ bundle exec rake -T

$ bundle exec rake build
# Build yaffle-0.1.0.gem into the pkg directory

$ bundle exec rake install
# Build and install yaffle-0.1.0.gem into system gems

$ bundle exec rake release
# Create tag v0.1.0 and build and push yaffle-0.1.0.gem to Rubygems
```

For more information about publishing gems to RubyGems, see: Publishing your gem.

RDoc Documentation

Once your plugin is stable, and you are ready to deploy, do everyone else a favor and document it! Luckily, writing documentation for your plugin is easy.

The first step is to update the README file with detailed information about how to use your plugin. A few key things to include are:

- Your name
- How to install
- How to add the functionality to the app (several examples of common use cases)
- Warnings, gotchas or tips that might help users and save them time

Once your README is solid, go through and add rdoc comments to all the methods that developers will use. It's also customary to add `# :nodoc:` comments to those parts of the code that are not included in the public API.

Once your comments are good to go, navigate to your plugin directory and run:

```
$ bundle exec rake rdoc
```

References

- Developing a RubyGem using Bundler
- Using .gemspecs as Intended
- Gemspec Reference