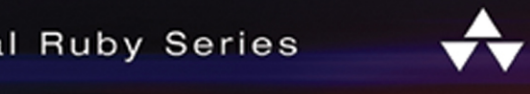
Addison-Wesley Professional Ruby Series



COMPONENT-BASED RAILS APPLICATIONS

Large Domains Under Control

STEPHAN HAGEMANN

Component-Based Rails Applications

./components/app_component/app/helpers/app_component/predictions_helper.rb

```
module AppComponent
  module PredictionsHelper
    def prediction_text(team1, team2, winner)
      "In the game between #{team1.name} and #{team2.name} " +
          "the winner will be #{winner.name}"
    end
  end
end
```

Since the test and implementation themselves don't offer much topic for discussion, let us turn to a more general discussion of how to test namespaced classes.

As we have seen a couple of times, the one thing that keeps repeating is the `AppComponent::` prefix, which is needed to denote that we are in the component's namespace. One can get around having to repeat the `AppComponent::`, which fully qualifies the name of the object under test, by ensuring that the entire test is in the lexical scope of the `AppComponent` module.[3] For the preceding test, that would look as follows:

predictions_helper_spec.rb with AppComponent as module scope

```
module AppComponent
  RSpec.describe PredictionsHelper, :type => :helper do
    it "returns a nice prediction text" do
      Named = Struct.new(:name)
      text = prediction_text(
          Named.new("A"), Named.new("B"), Named.new("C"))
      expect(text).to eq
          "In the game between A and B the winner will be C"
    end
  end
end
```

In this particular case, we don't gain much, but for more complex classes, this is tempting—it might remove the need for quite a few

3. In the second version of the `PredictionHelper` spec, the describe block is nested inside of the block created by `module AppComponent`, which means that the describe (and everything therein) is within the lexical scope of the `AppComponent` module.

`AppComponent::` constructs! The interesting side effect is that that following class would in fact pass the test we now have:

./components/app_component/app/helpers/app_component/predictions_helper.rb that passes the new test. Bad.

```
1 module PredictionsHelper
2   def prediction_text(team1, team2, winner)
3     "In the game between #{team1.name} and #{team2.name} " +
4         "the winner will be #{winner.name}"
5   end
6 end
```

Note that this helper module is not in the `AppComponent` namespace! The reason this still works is that Ruby searches for every constant in the current lexical scope of namespaces first. Then, successively removing the last element of the scope (until it is empty), it checks the parent namespaces. The only way to prevent this is to re-add the `AppComponent::` prefix to the test.

There are merits to both ways of doing it, and even to combining the two. In the end, it comes down to whether we can still be confident in the tests we have written.

3.1.5 Feature Specs

To add feature specs, I typically add `capybara` (http://teamcapybara.github.io/capybara/) as a gem dependency. With regard to testing a component, the only thing of note is the fact that the first line of the actual test (line 17) visits `'/app_component/'`. This is because, as discussed in Appendix B, the dummy app mounts the engine under test at its snake-cased name by default, which is `app_component` in our case.

./components/app_component/spec/features/predictions_spec.rb

```
 1 require "spec_helper"
 2
 3 RSpec.describe "the prediction process", :type => :feature do
 4   before :each do
 5     team1 = create_team name: "UofL"
 6     team2 = create_team name: "UK"
 7
 8     create_game first_team: team1,
 9                 second_team: team2, winning_team: 1
10     create_game first_team: team2,
11                 second_team: team1, winning_team: 2
```

```
12      create_game first_team: team2,
13                  second_team: team1, winning_team: 2
14    end
15
16    it "get a new prediction" do
17      visit "/app_component/"
18
19      click_link "Predictions"
20
21      select "UofL", from: "First team"
22      select "UK", from: "Second team"
23      click_button "What is it going to be"
24
25      expect(page).to have_content "the winner will be UofL"
26    end
27  end
```

How I Actually Write My Tests

In this section, I have listed all available types of specs. Half of these I never use in practice.

I have never written *routing* and *view* specs (when they still existed, before Rails 5). I avoid writing *helpers* (and their specs) and instead create presenters or similar concepts that I place next to the models they present in the `app_component/models` folder. I would like controllers to always be so simple that I never have to write *controller* specs. However, I do tend to write them and, when I do, I try to use them to create good interactions with models and services.

I write model specs, controller specs, and feature specs. *Features* I write to ensure that the overall functionality works—as such, they tend to go full stack.

3.2 Testing the Main Application

With the singular component in our application tested, the focus can turn back to the main application and the question: Is there anything on this side of the house that needs testing?

The only pieces of code that we have changed so far are `routes.rb` and `Gemfile`. We should have changed `README`, but haven't done so yet; it would also not affect any testing. Every other piece of code is still as

Rails generated it. That is, every piece of code that was not deleted. The `app` folder was the first thing to go. In the current version of the code, `lib` and `vendor` have also been deleted.

As discussed before, I believe that code should be tested at the lowest level possible so that the tests are as fast and as isolated as they can be. Here, there is hardly any code; it is all boilerplate and the only two changes are about gluing the component into the Rails app. To me, it follows that the only thing in need of testing is the fact that this "gluing" worked. And for this, I like to use a feature spec.

3.2.1 What to Test in the Main App

The following tests verify four aspects of the application: that the main app is mounted, that the teams page loads, that the games page loads, and that a prediction can be performed.

./spec/features/app_spec.rb

```
1  require "spec_helper"
2
3  RSpec.describe "the app", :type => :feature do
4    it "hooks up to /" do
5      visit "/"
6      within "main h1" do
7        expect(page).to have_content "Sportsball"
8      end
9    end
10
11   it "has teams" do
12     visit "/"
13     click_link "Teams"
14     within "main h1" do
15       expect(page).to have_content "Teams"
16     end
17   end
18
19   it "has games" do
20     visit "/"
21     click_link "Games"
22     within "main h1" do
23       expect(page).to have_content "Games"
24     end
25   end
26
27   it "can predict" do
28     AppComponent::Team.create! name: "UofL"
29     AppComponent::Team.create! name: "UK"
```

```
30
31      visit "/"
32      click_link "Predictions"
33      click_button "What is it going to be"
34    end
35  end
```

The following spec verifies the correct loading and mounting of the `AppComponent` component as succinctly as possible. First (line 5) it visits `/`, the root of the application, which, as set in Section 2.1, is where the component is mounted. This contrasts with the feature spec for the prediction functionality in Section 3.1, which visited `/app_component` because of how the dummy app mounts the component. Then, in line 6, the spec verifies a single piece of content, namely the header "Sportsball", to be present on the visited page. This ensures not only that the content can be found, but in effect also that there are no errors on the page, and that the component is where we expect it.

3.2.2 How to Put Everything Together

With the test for the main application, there are now two test suites one can run: `cd sportsball && rspec spec` and `cd sportsball/components/app_component && rspec spec`, and both need to run in order to verify that everything in the app works. While two commands instead of one may at first not seem like a big deal, we should strive to keep the simplicity that is embodied in "one command to run all tests."

I tend to implement this functionality with shell scripts. And the solution I am going to propose here is to create one shell script per component and an additional one as a runner for all components.

./components/app_component/test.sh

```
1  #!/bin/bash
2
3  exit_code=0
4
5  echo "*** Running app component engine specs"
6  bundle
7  bundle exec rake db:create db:migrate
8  RAILS_ENV=test bundle exec rake db:create db:migrate
9  bundle exec rspec spec
10 exit_code+=$?
11
12 exit $exit_code
```

./test.sh

```
#!/bin/bash

exit_code=0

echo "*** Running container app specs"
bundle
bundle exec rake db:drop
bundle exec rake environment db:create db:migrate
RAILS_ENV=test bundle exec rake environment db:create db:migrate
bundle exec rspec spec
exit_code+=$?

exit $exit_code
```

At this stage, the two `test.sh` files are essentially the same. For convenience, they both include bundling as well as creating and migrating of the database into one script together with the execution of the actual test. Only if every step in the script passes does the test script report a zero exit status to denote a successful execution.

In the future, the scripts may grow to be different—for example, if one of them needs to execute JavaScript tests or if the feature specs are broken out from the other specs. The only property they really need to keep is that they return a zero exit code if everything went well and a positive one if something went wrong.

To allow the joint execution of these tests scripts, we create the third script as `build.sh` in the root of the application.

./build.sh

```
#!/bin/bash

result=0

cd "$( dirname "${BASH_SOURCE[0]}" )"

for test_script in $(find . -name test.sh); do
  pushd `dirname $test_script` > /dev/null
  ./test.sh
  ((result+=$?))
  popd > /dev/null
done

if [ $result -eq 0 ]; then
        echo "SUCCESS"
```