Writing first tests

Essay Writers

Series "Development MKDEV"

Writing first tests

Refactoring, Metaprogramming and Epic Editing Markdown

Mailing alerts on a schedule using Sidekiq and Cron

How MkDev made a refactoring of sending letters

How and why mkdev moved to vue.js

How to bypass Roskomnadzor's blocking: History MKDEV

Internship on MKDev: How is the reviews of three participants

How we chose chat for students and mentors

Writing first tests

AUTHOR:

Kirill Shirinkin

Devops and Infrastructure Consultant, AWS expert, programmer, author of three books, mentor and founder MKDev.me.I design and develop software products.I also write about this and teach people.

I generally opponent TDD techniques. In my opinion, when solving any task, it is primarily necessary to solve it, and only then cover tests. Writing tests for a non-existent code does not make much sense, especially if you start the project from scratch and still imagine how it will look and work (but you already know what you want from him). In the case of the development of the first version of MKDEV, we faced such important tasks as automation of routine learning processes and providing students and menstrual interface to track the execution of tasks. After the release of the first version (which occurred by 23% faster, since it did not immediately waste time on tests) there were even more important tasks: the code for payments, subscriptions, alerts, background tasks, etc.

Now that we already know exactly in which direction the development of MKDEV is moving and what kind of features came across and needed users, we need to cover them with tests.I will do this

today: in this material we will go away from the state in which nothing is configured in the tests at all, to the coverage of the main functionality of the project.

Attention!Such late coverage of tests is justified if you already have experience writing a reliable working code without significant bugs and at the same time the project belongs to you personally, and not the company you work for.In the case of hired work, be sure to cover your code tests.I will in no way agitate against the tests.I agitating against spending time on tests before you decide to go to your task.

Baister!In MKDEV we will use RSPEC-Rails, Capybara and Factory_Girl.Our task will be to write Feature Speki today, that is, tests that check the correct operation of the application from the user.You may have already encountered writing such tests in one of the tasks of the Rails;)]

Add the necessary hem in Gemfile:

Group: Test Do Gem 'Rspec-Rails', '~> 3.0.0' Gem 'Factory_Girl_rails' gem 'Capybara' End

Following the instructions of the RSPEC-Rails gem, we generate the files you need for tests:

Rails Generate RSPEC: Install

Now you need to connect Capybara and Factory_Girl in Spec / Rails_helper.rb:

SPEC / Rails_helper.rb # ... Require 'Rspec / Rails' # This line there was already require 'Capybara / Rails' # This line added # ... RSPEC.configure Do | config |# ... Config.include FactoryGirl :: Syntax :: Methods # ... End

You can go to the writing of the first test!Create a Spec / Features folder /, in it file Course_Spec.rb.As can be seen from the name, we will test something connected with courses.For each test in this file we will need a login user:

DESCRIBE 'TAKING A COURSE' DO LET! (: User) {Create (: user, email: "bob@mail.ru", password: "qweqweqwe")} Before (: Each) Do Login ("bob@mail.ru",« qweqweqwe ") End End

Naturally, this code does not work yet, since we did not create the Factory for the User model and did not create a Helper Login method, inside which we will pass through the user login procedure (I suspect, we will often log in to the Feature Speakers, so it's better to go right away.the code).

SPEC / Factories / Subscription.rb FactoryGirl.define Do Factory: Subscription Do Active True Expire_date Date.Today + 4.Weeks End End # Spec / Factories / user.rb FactoryGirl.define Do Factory: User Do Firt_name "Boris" Last_Name "Spider "Password" secret123 "password_confirmation {password} customer_id" superbad "after (: create) Do | User |user.subscription.update_attributes (Active: True, Expire_Date: Date.Today + 4.Weeks) End End End

SPEC / support / login_helper.rb Def Login (Email, Password) Visit root_path click_link "Log in" Fill_in: User_Email, with: Email Fill_in: User_Password, with: password click_button "Log in" END

Now much better!Let's try to write the first test:

SPEC / Features / Course_spec.rb # ... it "Opens Courses Page) .to had_content" Current courses "End # ...

The test run by the Bundle Exec Rspec command successfully passes.Now I want to test that if there is a job course in the system, the user sees the link to start the course, clicks on it and sees after this name of the first task.We will need Factory for Course and Task:

Spec / Factories / Course.rb FactoryGirl.define Do Factory: Course Do Title "Ruby" End End # Spec / Factories / Task.rb FactoryGirl.Define Do Factory: Task Do Position 1 Title "Task 1" Text "Do Some CrazyShit! "End End.

The test itself looks like this:

SPEC / Features / Course_spec.rb # ... Context "CAN START AND GO THROUGH COURSE" DO BEFORE (: Each) Do Course = Create (: Course, Title: "Ruby On Rails") Task = Create (: Task,Course: Course, Title: "Simple Controller") END IT "CAN START COORSE" Do Visit Dashboard_Root_Path Click_Link "Start" Expect (Page) .to Have_Content "Simple Controller" End End # ...

Apparently the new user can start the course without any problems.Test now that he will not see the Accept button after sending a task to check, as well as the fact that after sending to check the task log adds a new entry.

SPEC / Features / Course_Spec.rb # ... Context "Sends a Task for Review" Do Before (: Each) Do Visit Dashboard_Root_Path Click_Link "Start a" click_link "simple controller" Fill_in "user_task_pull_request_url", with: "http: //Github.com/pulls/1 "click_button" Add PR "click_link" to check "END IT" Updates Task Journal "DO EXPECT (Page)EXPECT (Page) .not_To have_Content "Accept" End End # ...

Note: I do not use more than one EXPECT in each test.This is considered a good style, because in this case each test checks one particular thing, and not immediately hundreds of various aspects.Thus, if one test falls, then we know that the problem is only in one part of the code.If we had several checks in one test, we would not be able to find out the result of those that go below failed verification.For example:

IT "Updates Task Journal" Do Expect "Sent" Expect (Page) .Not_To Have_Content "Accept" End

If the first Expect falls out with an error, then we will find out that the log record does not work.But

since the error has already fallen out, then we will not know whether the user is displayed to accept until the previous check will be reproduced.

Another important point: Please note how precisely these tests are playing the user's actions. I do not check the code itself that is responsible for which the user will encounter, I test only the response of the application for its actions. The internal implementation of models and controllers can be completely changed, but the procedure for working with courses and tasks should not break after these changes. That is what guarantees us Feature tests: that the application works correctly by the user. Agree that this is perhaps the most important thing in any application.

Perhaps it is worth adding a couple of small tests that are checked as follows:

Simple users do not have access to admin

Adminov has access to anything

The editors have no access to editing users

I will give a whole final test that I got:

SPEC / Features / Admin_Spec.rb Require 'Rails_helper' Describe 'Accessing Admin' Do Let! (: user) {Create (: User, Email: "bob@mail.ru", password: "qweqweqwe")} before (:Each) Do Login ("bob@mail.ru", "qweqweqwe") END IT "DOESNT SHOW ADMIN FOR REGULAR USER" DO EXPECTENT "ADMINK" END IT "Redirects User to Account WHEN HE TRIESto access admin "Do Visit Admin_Root_Content" Current Courses "END CONTEXT" ADMIN User "Do Beore (: Each) Do User.add_role: admin Visit Admin_Root_Path End It" SHOWS LINK TO Admin "Do Expect (Page) .to had_conten "Admins" END IT "SHOWS LINK TO USERS MANAGMENT" DO EXPECT "END END CONTEXT" EDITOR User "Do Beore (: Each) Do User.add_role: Editor Visit Admin_Root_Path End It"SHOWS LINK TO ADMIN" DO EXPECTENT "END IT" SHOWS LINK TO USERS MANAGMENT "DO EXPECTENT" Pupils "End End End

These were the two first written tests for mkdev.me.In another article, I will tell about what cases and how to write tests on the model, again using the MKDEV.ME tests as examples.

We tell how to become a better developer, how to maintain and effectively apply your skills.Information about jobs and promotions exclusively for more than 8,000 subscribers.Join!

Our newsletter: