Tesztelés Rails-ben
Gyakorlat

Kovács Gábor

2013. november 27.

A gyakorlaton az előző gyakorlatok során elkészített megoldást tesztelésével foglalkozunk.

A Rails tesztkörnyezete három elemből áll:

- tesztadat-definíció, ami a tesztadatbázisba betöltendő rekordokat definiája,
- teszteset-specifikáció,
- tesztvégrehajtó környezet, amely automatikusan végrehajtja a teszteseteket a tesztadatokat felhasználva.

A tesztelés előkészítésére először teszt adatokat definiálunk, amelyekre a teszteseteinkben hivatkozni fogunk. Ezeket a Rails alkalmazásunk `test/fixtures` könyvtárában helyezzük el. Az egységtesztek és funkcionális tesztek számára ezeket az adatokat a minden egyes teszt eset elején hivatkozott `test/test_helper.rb` fájl fogja elérhetővé tenni, integrációs tesztek esetén pedig magunk töltjük be.

Definiáljuk az alábbi két `User` példányt az `users.yml` fájlban. Először definiáljunk egy közös jelszót az összes felhasználó számára, aminek kódolatlan értékét a `PASS` attribútumban tároljuk el. Az időközben egy migrációban átnevetett titkosított jelszó attribútum megadához a modell osztály `encrypt` osztálymetódusát hívjuk segítségül. A felhasználóspecifikus a titkosítás során használt `salt` attribútumot külön-külön definiáljuk. A két további attribútumot is inicializáljuk kiegészítve az automatikusan generált mintát.

```
<%
PASS = 'titok'
SALT = 'valami'
%>
```

```
elek:
  username: tesztelek
  email: elek@mail.bme.hu
  encrypted_password: <%= User.encrypt PASS, SALT %>
  salt: <%= SALT %>
  account: 0
  bank_account: 11111111

valaki:
  username: valaki
  email: valaki@mail.bme.hu
  encrypted_password: <%= User.encrypt 'jelszo', 'salt'
      %>
  salt: salt
  account: 1
  bank_account: 22222222
```

A felhasználókhoz hasonló módon megadjuk két fogadható esemény tesztadatait is az `events.yml` fájlban. Az események modellt `scaffold`-dal hoztuk létre így ott már láthatunk kezdeti adatokat, amiket módosítunk

```
vizsga:
  eventtime: 2013-12-16 13:04:55
  description: potlasi het
  oddstrue: 1.9
  oddsfalse: 2.5

potlas:
  eventtime: 2013-12-15 23:04:55
  description: utolso utani esely a hazi beadasara
  oddstrue: 1.1
  oddsfalse: 9.5
```

A fogadások tesztadatait (`bids.yml`) is vegyük fel, és egyúttal használjuk ki a másik két tesztadatfájlban definiált kulcsokat az idegen kulcsok inicializációja céljából.

```
bid_one:
  bidtime: 2013-11-27 12:35:16
  user: elek
  event: potlas
```

```
bid_two:
  bidtime: 2013-11-27 12:35:16
  user: valaki
  event: vizsga
```

Töltsük be a teszt környezet adatbázisába ezeket az adatokat ügyelve arra, hogy a környezetként a teszt környezet legyen beállítva.

```
RAILS_ENV='test' rake db:test:prepare
RAILS_ENV='test' rake db:migrate
RAILS_ENV='test' rake db:fixtures:load
```

Ha MySQL konzolon megnézzük a betöltött adatokat, azt láthatjuk, hogy az `id` attribútum véletlen értékkel töltődött fel, az időpecsétek pedig a betöltés időpontját vették fel.

Először írjunk egységteszteket! Az egységtesztek a modell osztályok metódusait és validációit hivatottak ellenőrizni. Az egyégteszteket a Rails projektünk `test/unit` könyvtárában találjuk. Minden egyes modell létrehozása után automatikusan létrejön hozzá egy ahhoz kapcsolódó teszt osztály itt.

Írjunk teszteseteket, amelyek a `User` modellünk

```
validates :username, :uniqueness => true, :presence =>
  true
```

validációinak megtörténtét ellenőrzik. Először létrehozunk egy új felhasználót mindenféle inicializáció nélkül, majd megpróbáljuk eltárolni az adatbázisba. A feltételezésünk az, hogy a mentésnek nem szabad sikerülnie. A másik tesztesetünk azt ellenőrzi, hogy a tesztadatok között megadott felhasználónév, jelszó páros segítségével be tudhatunk-e egyáltalán jelentkezni.

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase
  test "can save user without username" do
    u = User.new
    assert !u.save, "Validation does not work"
  end

  test "is authentication successful" do
    u = users(:elek)
    assert_not_nil User.authenticate(u.username, 'titok
      '), "Authentication does not work"
  end
end
```

Miután a webszerverünket újraindítottuk úgy, hogy az a tesztkörnyetetet használja, a tesztesetet futtatva meggyőződhetünk róla, hogy tényleg nem történik meg a mentés. Ha mégis sikerülne, a megadott hibaüzenetnek kell megjelennie. Egyúttal böngészőből is kipróbálhatjuk, hogy működik a bejelentkezés.

```
RAILS_ENV=test rake test:units
(in /home/kovacsg/gyakorlat)
Loaded suite /var/lib/gems/1.9.1/gems/rake-10.1.0/lib/
    rake/rake_test_loader
Started
..
Finished in 0.151356 seconds.

2 tests, 2 assertions, 0 failures, 0 errors, 0 skips

Test run options: --seed 15441
```

A funkcionális tesztek a kontrollerek és a nézetek helyes működését ellenőrzik. Alapértelmezés szerint minden egyes a létrehozáskor megadott kontroller akcióra létrejön egy az akció sikeres megjelenítését ellenőrző teszteset.

Először egészítsük ki a `SessionsControllerTest` tesztet, amely jelenleg két tesztesetet tartalmaz. Az első a bejelentkezést, a második a kijelentkezést teszteli, nevezzük át a teszteket ennek megfelelően! A bejelentkezés eseményt a `SessionController create` metódusát használja, a kijelentezés esemény pedig a `destroy` metódust. A tesztben a `post` metódust használjuk, amelynek első paramétere a teszelendő akció, a második a HTTP kérés paraméterei, a harmadik pedig a kérés előtti `session` paraméterek értékeit tartalmazó hash. A sikeres bejelentkezés tesztben a tesztadatokban szereplő felhasználónév, jelszó párost küldjük el, és azt feltételezzük, hogy visszairányítódunk a belépés előtti nézetre, a `:user` session paraméter értéke nem `nil`, ráadásul megegyezik a tesztadat azonosítójával. A kilépés tesztben HTTP kérés paramétereket nem adunk meg, tehát a második paraméter `nil`, azonban harmadik paraméterként beállítjuk a `:user` session paramétert azt imitálva, hogy van bejelentkezett felhasználónk. Válaszként átirányítást várunk az aktuális oldalunkra, és azt, hogy a session paraméter kinullázódik.

```
require 'test_helper'

class SessionsControllerTest < ActionController::
    TestCase
  test "login" do
```

```
    post :create, {:username => users(:elek).username,
       :password => 'titok'}
    assert_response :redirect
    assert_equal session[:user], users(:elek).id
  end

  test "logout" do
    get :destroy, nil, {:user => users(:elek).id }
    assert_response :redirect
    assert_nil session[:user]
    assert_not_nil flash[:notice]
  end

end
```

A tesztesetünk kész van azonban nem futhat le sikeresen ugyanis a kontrollerben szereplő `redirect_to :back` átirányításban nem definiált a `:back` értéke. Ezt a Rails a javascript history elérhetetlensége miatt a `HTTP_REFERER` nevű HTTP kérés paraméterből veszi, így ezt minden egyes tesztesetben be kell állítanunk. Ezt hatékonyan a `setup` metódusban tehetjük meg, amely minden teszteset előtt lefut.

```
class SessionControllerTest < ActionController::
   TestCase
  setup do
    @request.env["HTTP_REFERER"]='/say/hello'
  end
end
```

Az `EventsControllerTest`-et és a `BidsControllerTest`-et scaffolddal hoztuk létre, amely automatikusan generálta a teszteseteket és tesztadatokat. A tesztadatok kulcsait módosítottuk, így az azokra való hivatkozást frissítenünk kell a `setup`-ban. Mivel a fogadás csakis bejelentkezett felhasználó számára lehetséges, e két tesztfájlban a `get` és `post` műveletek harmadik paraméterében inicializálunk kell a `sessions` hasht a harmadik paraméterben. Az alábbi kódrészlet erre mutat egy példát, azonban ezt a kontrollerekben szereplő `before_filter` miatt minden akcióban el kell végeznünk.

```
class BidsControllerTest < ActionController::TestCase
  def setup
    @user = tasks(:elek)
  end
```

```
test "should get index" do
    get :index, nil, {:user => @user.id}
    assert_response :success
    assert_not_nil assigns(:bids)
  end
end
```

A `UsersControllerTest` osztályunk az `edit`, `show` és `forgotten` akciókat ellenőrző teszteseteiben a `get` második paramétereként be kell állítanunk az `id` paramétert a tesztadatok alapján vagy a harmadik paraméterrel azt szimuláljuk, hogy a felhasználó már bejelentkezett. Minden mást az egyszerűség kedvéért változatlanul hagyunk.

```
class UsersControllerTest < ActionController::TestCase
  test "should get new" do
    get :new
    assert_response :success
  end

  test "should get show" do
    get :show, nil, { :user => users(:me).id }
    assert_response :success
  end

  test "should get forgotten" do
    get :forgotten, nil, { :user => users(:me).id }
    assert_response :success
  end
end
```

A funkcionális teszteseteket futtatva láthatjuk, hogy sikeresen lefutnak.

```
> rake test:functionals
(in /home/kovacsg/gyakorlat)
Loaded suite /var/lib/gems/1.9.1/gems/rake-10.1.0/lib/
   rake/rake_test_loader
Started
.....................
Finished in 0.685053 seconds.

21 tests, 30 assertions, 0 failures, 0 errors, 0 skips

Test run options: --seed 18675
```

A tesztek harmadik típusa az integrációs teszt, amellyel egy böngészési folyamatot ellenőrzünk. Készítsünk egy tesztet a felhasználói profilkép feltöltésének esetére!

```
rails generate integration_test upload_profile_picture
```

E parancs kiadása után a `test/integration` könyvtárban létrejött egy `upload_profile_picture_test.rb` nevű állomány, ahol az integrációs tesztünk metódusait helyezzük el.

A fájl elején töltsük be az összes teszt adatot `fixtures :all`, majd definiáljuk a böngészési folyamatot úgy, hogy az a bejelentkezéstől jusson el legalább a feladat beküldéséig. Ne feledkezzünk el a karakterkódolás beállításáról sem, ha ékezetes karaktert használunk!

Az integrációs tesztesetet a funkcionális teszteknél megismert tesztesetekből mint tesztlépésekből tevődik össze.

Az első lépésben egy be nem jelentkezett felhasználó betölt egy véletlen oldalt, amin elérhető a bejelentkezési form. Itt csak annyit feltételezünk, hogy az oldal betöltődik, és azon van egy `Login` feliratú form. Ez utóbbit az `assert_select` metódussal ellenőrizzük annak egy CSS selector kifejezést átadja.

A második tesztlépés a bejelentkezés. Az aktuálisan megjelenített nézeten található egy form, amelyen kereszül `username` és `password` kérés paramétereket juttathatunk el a `/sessions/create` kontroller akciónak. Mivel az akció végén minden esetben egy átirányítás áll, `post` helyett a `post_via_redirect` metódust kell használnunk, ami a kérés után követi az összes átirányítást. Ennek a metódusnak a paraméterlistája eltér a megszokottól, vagyis az akció, paraméterekek, session, flash négyestől, itt a második paraméter ugyanúgy a HTTP kérés paraméterlistája, azonban a harmadik paraméter a HTTP fejrész opciók hashe. Mivel az átirányítás az előző oldalra történik a vizsgálat akcióban, a harmadik paraméterben kell megadnunk a `HTTP_REFERER` fejrész opcióval, hogy mely oldal volt az előző oldal. A feltételezésünk az, hogy a sessionünk inicializálódik.

A harmadik tesztlépés letölti a felhasználó saját profilját szerkesztő oldalt, ahol elérhető a profilkép-feltöltés formja. Itt is azt feltételezzük, hogy az oldal sikeresen megjelenik, és azon elérhető a profilszerkesztés feliratú form.

A negyedik tesztlépésben előhalásszuk az előző tesztfájlunkat. A profilképfelöltés műveletének tesztelésére töltsünk be a memóriába egy valóságos állományt a `fixture_upload_file` metódussal. Ez a metódus a tesztadatok könyvtárához képesti relatív útvonalat vár első argumentumként, a második, opcionális argumentuma az ott található állomány MIME típusa. Hoz-

zunk létre egy tetszőleges szöveget tartalmazó fájlt a `test/fixturesfiles` könyvtárban `test.png` néven. A tesztesetünkben két feltételezéssel élünk, egyrészt ellenőrizzük, hogy a profilkép neve megjelent-e az adatbázis felhasználók táblájában, másrészt pedig jó lenne, ha a fájl valóban megjelenne a fájlrendszeren.

```
require 'test_helper'

class UploadProfilePictureTest < ActionDispatch::
    IntegrationTest
  # test "the truth" do
  #   assert true
  # end
  fixtures :all
  test "upload profile picture" do
    get '/'
    assert_response :success
    assert_select 'legend', "Login"
    post_via_redirect '/sessions/create',
      { username: users(:elek).username, password: '
        titok' },
      { 'HTTP_REFERER'=>'/say/hello' }
    assert_equal session[:user], users(:elek).id
    get '/users/edit'
    assert_response :success
    assert_select 'legend', "Edit user profile"
    upload_file = fixture_file_upload('test/fixtures/
      files/test.png','text/plain')
    post url_for(:controller=>'users', :action=>'upload
      ', :id=>users(:elek).id),
      { id: users(:elek).id,
        upload:
        {
          original_filename: 'test.png',
          file: upload_file
        }
      }
    u = User.find users(:elek).id
    assert_not_nil u.profile_picture
    assert File.exists?('public/profilepictures/test.
      png')
```

```
  end
end
```

Futtatva az integrációs tesztet láthatjuk, hogy az egy teszteset négy tesztlépése során mind az öt ellenőrzésen sikeresen átment.

```
rake test:integration
(in /home/kovacsg/gyakorlat)
Loaded suite /var/lib/gems/1.9.1/gems/rake-10.1.0/lib/
    rake/rake_test_loader
Started
.
Finished in 0.436569 seconds.

1 tests, 7 assertions, 0 failures, 0 errors, 0 skips

Test run options: --seed 52981
```

A tesztek ötödik nagy csoportja a portál teljesítőképességét hivatott ellenőrizni. A teljesítménytesztet akárcsak az integrációs teszeket explicit módon kell létrehoznunk `rails generate performace_test`.

A teljesítményteszek futtatásához szükségünk van a `ruby-prof` függvénykönyvtárra, melyet a `Gemfile`-ban kell meghivatkoznunk, illetve egy olyan `ruby` értelmezőre, amely képes monitorozni a végrehajtást során felhasznált memóriát és a végrehajtáshoz szükséges időt. Az alapértelmezett értelmező csak korlátosan képes kiszolgálni a teljesítménytesztet.

A teljesítménytesztnek két fajtája van a hosszú végrehajtási időt igénylő metódusok tesztje (`profile`) és a statisztika (`benchmark`). A két teszttípus ugyanazokat a teszteseteket hajtja végre, és hasonló paraméterekkel konfigurálhatók.

Egy teszt a projekt létrehozásakor automatikusan generálódik. Az egyetlen tesztesete a portálunk főoldalának kiszolgálását. A vizsgálatunkhoz használjuk azt fel. Állítsuk be az opciókat, hogy a számunkra érdekes adatok jelenjenek meg a konzolon.

```
require 'test_helper'
require 'rails/performance_test_help'

class BrowsingTest < ActionDispatch::PerformanceTest
  # Refer to the documentation for all available
    options
  self.profile_options = { :runs => 5, :metrics => [:
    wall_time, :memory],
```

```
                                    :output => 'tmp/performance
                                        ', :formats => [:flat] }

  def test_homepage
    get '/'
  end
end
```

A tesztesek végrehajtásának naplója a `logs` könyvtárba kerül, a teljesítménytesztek részletes adatai emellett a `tmp/performance` könyvtárban is megjelennek szövegesen, táblázatosan esetleg képként grafikusan.

A gyakran végrehajtandó teljesítményteszt-célok vonatkozásában teszteseteket érdemes definiálnunk, amelyeket a `test:profile` és a `test:benchmark` `rake` célokkal hajthatunk végre.

```
rake test:profile
(in /home/kovacsg/gyakorlat)
Loaded suite /var/lib/gems/1.9.1/gems/rake-10.1.0/lib/
   rake/rake_test_loader
Started
BrowsingTest#test_homepage (116 ms warmup)
        process_time: 20 ms
              memory: 0 Bytes
             objects: 0

Finished in 0.919409 seconds.

1 tests, 0 assertions, 0 failures, 0 errors, 0 skips

Test run options: --seed 10431
```

```
rake test:benchmark
(in /home/kovacsg/gyakorlat)
Loaded suite /var/lib/gems/1.9.1/gems/rake-10.1.0/lib/
   rake/rake_test_loader
Started
BrowsingTest#test_homepage (127 ms warmup)
           wall_time: 8 ms
              memory: unsupported
             objects: unsupported
             gc_runs: 0
             gc_time: 0 ms
```

```
Finished in 0.854406 seconds.

1 tests, 0 assertions, 0 failures, 0 errors, 0 skips

Test run options: --seed 52804
```

Egy-egy – jellemzően modell osztály – metódus teljesítményvizsgálatára nem feltétlenül érdemes teszteset írnunk, az elvégezhetjük a `rails` szkript `profiler` és `benchmarker` céljaival is. Az alábbi példák az összes felhasználó lekérdezése adatbázis műveletének teljesítményét vizsgálják meg.

```
rails profiler User.all
Loaded suite script/rails
Started
ProfilerTest#test_user_all (29 ms warmup)
        process_time: 3 ms
              memory: 0 Bytes
             objects: 0

Finished in 2.341041 seconds.

1 tests, 0 assertions, 0 failures, 0 errors, 0 skips

Test run options: --seed 4513
```

```
rails profiler User.all --metrics wall_time,user_time,
   cpu_time,memory,objects
Loaded suite script/rails
Started
ProfilerTest#test_user_all (21 ms warmup)
           wall_time: 3 ms
            cpu_time: 1 ms
              memory: 0 Bytes
             objects: 0

Finished in 2.482132 seconds.

1 tests, 0 assertions, 0 failures, 0 errors, 0 skips

Test run options: --seed 1749
```

```
rails benchmarker User.all
Loaded suite script/rails
Started
BenchmarkerTest#test_user_all (28 ms warmup)
           wall_time: 1 ms
              memory: unsupported
             objects: unsupported
             gc_runs: 0
             gc_time: 0 ms

Finished in 1.113398 seconds.

1 tests, 0 assertions, 0 failures, 0 errors, 0 skips

Test run options: --seed 40426
```