Tesztelés Rails-ben
Gyakorlat

Kovács Gábor

2015. április 28.

Laborunk témája az elkészített rendszer tesztelése. 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 tesztadatokat 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. A modellt szkripttel hoztuk létre így ott már láthatunk kezdeti adatokat, amiket módosítunk. 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 később migrációval módosított, illetve hozzáadott attribútumokat magunknak kell módosítanunk, illetve hozzáadnunk a struktúrához, ilyen például a `salt`. Két játékost definiáljunk, akik között így játékot hozhatunk létre.

```
me:
  username: kovacsg
  name: Kovacs Gabor
  salt: titok
```

```
  encrypted_password: <%= User.encrypt 'titok','titok' %>
  email: kovacsg@tmit.bme.hu
  t: 1

valaki:
  username: valaki
  name: Vala Ki
  salt: titkos
  encrypted_password: <%= User.encrypt 'titkos','titkos' %>
  email: valaki@mail.bme.hu
  t: 2
```

A felhasználókhoz hasonló módon megadjuk a feladatok tesztadatait is az `tasks.yml` fájlban. A migrációk során ezt a modellt is módosítottuk, két attribútumot töröltünk, és felvettük a `url` mezőt.

```
one:
  number: 1
  deadline: 2015-03-16 13:08:00
  url: http://

two:
  number: 2
  deadline: 2015-03-30 13:08:00
  url: http://
```

A megoldások tesztadatait (`submissions.yml`) is vegyük fel. A megoldások egy-több relációban áll mind a felhasználók, mind a feladatok modellel, és a kulcsunk itt megfelel a Rail konvencióinak. Ez lehetővé teszi számunkra a másik tesztadat kulcsával való hivatkozást. Ha a YAML fájlunkban egy ilyen kulcshoz a hivatkozott tesztadat egy kulcsát rendeljük, a Rails feloldja a kapcsolatot.

```
one:
  user: valaki
  task: one
  filename: 1.txt
  mime: text/plain
  late: false

two:
  user: valaki
  task: two
  filename: 1.txt
  mime: text/plain
  late: false
```

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, ahol azt explicite nem definiáltuk, 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/models` (Rails 4 előtt `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,
    {
      :presence => true,
      :uniqueness => true,
      :length => {:in => 4..18},
    }
```

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.

Írjunk még egy tesztesetet, ami a modell egy példánymetódusát ellenőrzi. Ez az általunk írt `encrypt` osztálymetódust teszteli egy ismert tesztadat alapján.

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase
  test "the truth" do
    assert true
  end

  test "cannot save user with no username" do
    u = User.new
    assert !u.save, "Houston, we have a problem"
  end

  test "check if encryption works" do
    u = users(:valaki)
    assert_equal u.encrypted_password, User.encrypt('titkos','
        titkos')
  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:models
(in /home/kovacsg/gyakorlat)
Run options: --seed 8323

# Running:

...

Finished in 0.446262s, 6.7225 runs/s, 6.7225 assertions/s.

3 runs, 3 assertions, 0 failures, 0 errors, 0 skips
```

A funkcionális tesztek a kontrollerek és a nézetek helyes működését ellenőrzik, a `test/controllers` (Rails 4 előtt `test/functional`) könyvtárban találhatók. Alapértelmezés szerint minden egyes a létrehozáskor megadott kontroller akcióra létrejön egy az akció sikeres megjelenítését ellenőrző tesztteset.

Először egészítsük ki a `SessionsControllerTest` tesztet, amely jelenleg két tesztesetet tartalmaz. A funkciót a `valaki` kulcshoz tartozó felhasználó tesztadatain végezzük el, ehhez a `setup` metódusban inicializálunk egy példányváltozót az összes teszteset számára. 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
  setup do
```

```
    @valaki = users(:valaki)
  end

  test "login" do
    post :create, { :username => @valaki.username, :password => '
        titkos' }
    assert_response :redirect
    assert_equal session[:user], @valaki.id
  end

  test "logout" do
    get :destroy, nil, { :user => @valaki.id }
    assert_response :redirect
  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 SessionsControllerTest < ActionController::TestCase
  setup do
    @valaki = users(:valaki)
    request.env['HTTP_REFERER'] = '/tasks'
  end
end
```

A `UsersControllerTest`-et és a `TasksControllerTest`-et szkripttel hoztuk létre, amely automatikusan generálta a tesztadatokat, a teszteseteket és a tesztesetek törzsét. A tesztadatok kulcsait módosítottuk, így az azokra való hivatkozást frissítenünk kell a `setup`-ban. Mivel a műveleteink jó része 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, ráadásul egyes a műveletek csak oktató típusú felhasználók számára lehet elérhető. A teszteseteink legyenek az automatikusan generáltak kijavítva.

```
require 'test_helper'

class UsersControllerTest < ActionController::TestCase
  setup do
    @me = users(:me)
```

```
    request.env['HTTP_REFERER'] = '/tasks'
  end

  test "should get new" do
    get :new
    assert_response :success
  end

  test "should get edit" do
    get :edit, { :id => @me.id }
    assert_response :success
  end

  test "should get show" do
    get :show, { :id => @me.id }
    assert_response :success
  end

  test "should get forgotten" do
    get :forgotten
    assert_response :redirect
  end
end
```

Járjunk el hasonlóan a feladatok teszteseti esetére is. A `destroy` funkció nem érhető el, ezért annak teszteseit töröljük. A `create` és `update` akciók teszteseteinek második paraméterét, vagyis a `params` hasht módosítanunk kell a korábbi migrációnak megfelelően.

A funkcionális teszteseteket a `rake test:functionals` szkripttel futtathatjuk.

```
RAILS_ENV='test' rake test:functionals
(in /home/kovacsg/gyakorlat)
Run options: --seed 23582

# Running:

.............

Finished in 1.207751s, 10.7638 runs/s, 14.9037 assertions/s.

13 runs, 18 assertions, 0 failures, 0 errors, 0 skips
```

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

```
rails generate integration_test submit_solution
```

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

A tesztadatok a `test_helper` fájlra való hivatkozás miatt automatikusan elérhetők. A teszt során a `valaki` kulcsú felhasználó adjon be megoldást a `two` kulcsú feladatra. Ezeket a `setup` metódusban töltsük be egy-egy példányváltozóba. A böngészési folyamatot úgy definiáljuk, hogy az a bejelentkezéstől jusson el legalább egy megoldás beadásáig. 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 annyit feltételezünk, hogy az oldal betöltődik, és az oldalon HTML nézetének forrásában van egy `Login` értékű `legend` HTML elem.

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 az átirányítás sikeres, valamint a sessionünk inicializálódik, amit a session hash vizsgálatával tehetünk meg.

A harmadik tesztlépés a feladatok listájából kiválaszt egyet, és annak a nézetére navigál. Sikeres HTTP válasz az elvárásunk.

A negyedik tesztlépésben feltöltjük a megoldást. A feltöltendő fájlt a tesztadatok könyvtárában helyezzük el, az egy szöveges állomány, és a `fixture_file_upload` metódussal tesszük egy lokális változóban elérhetővé. A feltöltés egy HTTP POST művelettel történik, a `params` hash esetén arra kell figyelnünk, hogy az `upload` önmaga is egy hash, benne egy `file` kulccsal, amihez hozzárendeljük a fájlt. A feltöltés során az `original_filename` és `content_type` mezők automatikusan állítódnak. A feltételezésünk az, hogy a feltöltött fájl létrejön a `public/data` könyvtárban.

```
require 'test_helper'

class SubmitSolutionTest < ActionDispatch::IntegrationTest
```

```
  setup do
    @valaki = users(:valaki)
    @t = tasks(:two)
  end

  test "submit a solution" do
    get '/tasks'
    assert_response :success
    assert_select 'legend', "Login"
    post_via_redirect "/sessions/create",
      {:username=>@valaki.username, :password => 'titkos'},
      { "HTTP_REFERER" =>'/tasks'}
    assert_response :success
    assert_equal session[:user], @valaki.id
    get '/tasks/'+@t.id.to_s
    assert_response :success
    n = Submission.count
    upload_file = fixture_file_upload('test/fixtures/hf.txt','
        text/plain')
    post '/upload/'+@t.id.to_s,
         {:upload => {:file => upload_file }},
         { "HTTP_REFERER" => '/tasks' }
    assert File.exists?(Rails.root.join("public","data", @t.id.
        to_s, "hf.txt"))
  end
end
```

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

```
RAILS_ENV='test' rake test:integration
(in /home/kovacsg/gyakorlat)
Run options: --seed 53565

# Running:

.

Finished in 1.051401s, 0.9511 runs/s, 5.7067 assertions/s.

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

A tesztek ötödik nagy csoportja a portál teljesítőképességét hivatott ellenőrizni. Rails 4-től ez már nem része a teszt-keretrendszernek, külön telepítenünk kell a megvalósító függvénykönyvtárakat. A teljesítményteszek futtatásához szükségünk van a `ruby-perftest` és a `ruby-prof` függvénykönyvtárra, melyeket 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 teljesítményteszteket.

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. A benchmark a tesztesetek többszöri futtatása alapján általános statisztikát közöl a portál a teszteset által ellenőrzött részéről, a profile pedig a szűk keresztmetszetet jelentő pontokat próbálja azonosítani a sorrol sorra mért végrehajtási költség meghatározásával. Az előbbi egy fekete dobozos teszt annak eldöntésére, hogy van-e teljesítmény szempontjából probléma egy nézeten, az utóbbi pedig azt mondja meg, hogy hol.

A teljesítménytesztet akárcsak az integrációs teszeket explicit módon kell létrehoznunk:

```
rails generate performance_test p
```

Az alábbi teszteset az index nézet támadja meg 5 egymás utáni kéréssel.

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

class PTest < ActionDispatch::PerformanceTest
  # Refer to the documentation for all available options
  # self.profile_options = { runs: 5, metrics: [:wall_time, :
      memory],
  #                          output: 'tmp/performance', formats:
      [:flat] }

  test "homepage" do
    get '/tasks'
  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.

```
RAILS_ENV='test' rake test:profile
(in /home/kovacsg/gyakorlat)
Run options: --seed 3995

# Running:

PTest#test_homepage (212 ms warmup)
```

```
        process_time: 24 ms
              memory: 0 Bytes
             objects: 0
.

Finished in 7.184125s, 0.1392 runs/s, 0.0000 assertions/s.

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

```
RAILS_ENV='test' rake test:benchmark
(in /home/kovacsg/gyakorlat)
Run options: --seed 20708

# Running:

PTest#test_homepage (214 ms warmup)
           wall_time: 8 ms
              memory: unsupported
             objects: unsupported
             gc_runs: 0
             gc_time: 0 ms
.

Finished in 1.291188s, 0.7745 runs/s, 0.0000 assertions/s.

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

A szűk keresztmetszet általában egy számításigényes függvény vagy egy komplex adatbázislekérdezés. Egy-egy függvény vizsgálatára nincs szükség az összes teszteset lefuttatására, azt a `perftest` paranccsal is megtehetjük, ami során a `profiler` és a `benchmarker` opciók használhatók.

```
RAILS_ENV='test' perftest profiler User.all
Run options: --seed 34694

# Running:

ProfilerTest#test_user_all (16 ms warmup)
        process_time: 1 ms
              memory: 0 Bytes
             objects: 0
.

Finished in 1.206959s, 0.8285 runs/s, 0.0000 assertions/s.

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