Tesztelés Rails segítségével

A Rails programnyelv nagyszerű eszközöket biztosít számunkra az adatok teszteléséhez és az alkalmazásunk finomításához. Nézzük, hogyan kell használni.

A cikksorozat előző részeiben a ***Ruby On Rails***-szel foglalkoztunk. Ez egy ***Ruby*** nyelven írt nyílt forráskódú keretrendszer webalkalmazások készítéséhez, amely az alkalmazások és adatbázisok elkészítését teszi egészen egyszerűvé.
Több oldalát is megvizsgáltuk már a ***Rails*** segítségével történő fejlesztésnek: az alkalmazásunk modell-nézet-vezérlő hármasra bontását, az ***ActiveRecord*** osztály használatát az objektumok,adatbázisok leképezéséhez, és a ***Rails*** ellenőrzőeszközeit, amelyekkel megbizonyosodhatunk az adatbázisban tárolt adataink sértetlenségéről.
Egészen le vagyok nyűgözve a ***Rails*** kialakításától, mióta csak hónapokkal ezelőtt foglalkozni kezdtem vele. ***Rails*** alatt fejleszteni egy kicsit furának, különcnek tűnhet elsőre, de gyorsan bele lehet jönni, aztán már csak élvezni kell, hogy az unalmas fejlesztési lépéseket az ***ActiveRecord*** komponens automatikusan elintézi helyettünk.
Igaz, hogy a ***Rails*** csökkenti a webalkalmazásunkhoz szükséges kód mennyiségét, de nullára nyilván nem csökkentheti. És ha már van kódunk, elkerülhetetlenül tartalmaz hibákat is. Mint azt a tapasztalt webfejlesztők is tudják, az ilyen fajta alkalmazások tesztelése kissé trükkös, mivel az ügyfél által küldött adatok, és amit éppen lát, nincsenek közvetlen kapcsolatban azzal, ami a kiszolgálón történik. A mai napig sem ritka, hogy a webfejlesztők különböző kiíratórutinokkal és hibanaplók követésével keresik a hibákat az alkalmazásukban.
Valójában én személy szerint is számtalanszor esem ebbe a bűnbe, részben megszokásból, részben, mert gyakran ez a legjobb módja, hogy megtaláljuk a projekttel kapcsolatos problémákat. A vezetők és a programozók egyaránt tudják, hogy ez a legolcsóbb és legegyszerűbb módja a hibák felderítésének az előfordulás pillanatában, vagy a projekt korai szakaszában. Ám a programozók általában kelletlenül tesztelik a saját kódjaikat, kiváltképp, ha a tesztelés időigényes és unalmas.
Egy viszonylag egyszerű megoldás – amely az elmúlt évek során bukkant fel – az, hogy a fejlesztők nem csak az általuk fejlesztett szoftverek elkészítéséért felelősek, de azon tesztek megírásáért is, amelyek leellenőrzik a programot ahányféleképp csak lehetséges. Az ilyen, úgynevezett egységtesztek biztosíthatják, hogy a rendszer minden önálló eleme szilárd, lehetővé téve, hogy ráépítsünk, amikor a teljes alkalmazásba integráljuk.
Jelen cikkben a ***Rails*** egységtesztelésre alkalmas beépített szolgáltatásait tekintjük át, hátha ezzel megjön az étvágyunk a funkcionális tesztek elkészítéséhez is.

Az adatbázis tesztelése

Amiről idáig beszéltünk, egészen ésszerűen hangzik, azonban van itt egy a felszín alatt rejlő rejtett veszély. Ha most megyünk előre, és teszteljük az összes kódot, a tesztünk valószínűleg kimerül az adatok hozzáadásával, módosításával és törlésével az aktuális adatbázis felett. Egy komoly, éles rendszerben ez több, mint alkalmatlan, temérdek problémát okozhat.
Ha az olvasó azóta követi a cikksorozatot, hogy elkezdtünk dolgozni a ***Rails***-szel, bizonyára jól emlékszik, hogy minden egyes projekthez, amelyen dolgozunk, három adatbázist definiáltunk. Az egyszerű blog alkalmazáshoz, amiről az utóbbi hónapokban esett szó, például az alábbi három adatbázist készítettük: ***blog_development***, ***blog_test*** és ***blog_production*** (fejlesztői, teszt, végleges). Egyáltalán nem foglalkozunk a ***_test*** és ***_development*** adatbázisokkal, kizárólag a ***_production*** változatra koncentráltunk. Most, hogy tesztelni fogjuk az alkalmazásunkat, a _test adatbázist fogjuk használni. És csak amikor már biztosak vagyunk benne, hogy az alkalmazásunk átment a teljes tesztsorozaton, fogjuk átvinni a sémát a végleges adatbázisba.
Ha még nem tettük meg, hozzuk létre a teszt adatbázist, és töltsük be a sémát. Készítsünk egy blog felhasználót a ***PostgreSQL***-hez az alábbi parancs segítésével:

```
$/usr/local/pgsql/bin/createdb
➥-U blog blog_test
```

Ezután töltsük be az adatbázis sémáját, amelyet a ***blog/db*** mappában található ***create.sql*** fájlba mentettünk.

```
$/usr/local/pgsql/bin/psql -U
➥blog blog_test < blog/db/
➥create.sql
```

A fenti parancs betölti a táblák definícióit. Feltételezzük, hogy a ***create.sql*** azonos volt azzal, amivel a fejlesztői adatbázist annak idején létrehoztuk, és persze feltételezzük, hogy a fejlesztői és a teszt adatbázis is ugyanolyan módon lett létrehozva.

De mi van akkor, ha nem frissítettük a *create.sql*-t azokkal az utasításokkal, amit a fejlesztői adatbázisban végrehajtottunk? Kénytelenek leszünk a két adatbázis sémáját kézzel összehasonlítani, frissíteni a *create.sql* fájlt, majd újra betölteni a definíciókat?
A válasz szerencsére: nem. A *Rails* tartalmaz egy *clone_structure_to_test* nevű egyszerű programot, amely átmásolja a fejlesztői adatbázis szerkezetét a teszt adatbázisba. Ne feledjük, hogy csak a sémát másolja, az adatokat nem. A parancs kiadásához lépjünk be az alkalmazásunk gyökérmappájába (esetünkben a *blog* nevű könyvtárba), aztán használjuk a *rake*, vagy a *Ruby make* programok valamelyikét, amely elindítja a *Rake* fájl megfelelő szakaszát az adott könyvtárban.

```
$ rake clone_structure_to_test
```

Ha a *blog_test* adatbázis nem létezik, vagy más probléma van, hibaüzenetet kapunk, ellenkező esetben csak a szokásos kimenetet látjuk:

```
[reuven@server blog]$ rake
➥clone_structure_to_test
(in /home/reuven/blog)
```

Összeakadtam egy kezdeti problémával a `clone_strucutre_test` utasítással kapcsolatban, ami azért jön elő, mert nem a programot futtató felhasználó (*reuven*) a tulajdonosa az *PostgreSQL* adatbázis *public* nevű sémájának.
A problémát úgy lehet megkerülni, hogy a *blog* adatbázis-felhasználónak rendszergazdai jogosultságot adunk, ez feltétlenül szükséges a klónozó folyamat helyes végrehajtásához.

```
$ /usr/local/pgsql/bin/psql
➥blog_test
blog_development=# alter user
➥blog createuser;
ALTER USER
```

Most, hogy a teszt adatbázisunk a helyén van, elkezdhetjük megírni a teszteket. De hová is tegyük őket? A *Rails* a szokásos módszer szerint már meghatározott egy helyet a tesztek számára, és feltételezi, hogy ugyanazt a konvenciót fogjuk követni, amit az alkotó, és a többi *Rails* felhasználó. Eszerint a *blog/test*, *blog/app*, *blog/db* mappákba fogunk dolgozni.

A *blog/test* mappa négy almappát tartalmaz, valamint egy különálló *Ruby* kóddal teli fájlt, ezek mindegyike megtalálható minden *Rails* alkalmazás esetében. A négy almappa: *fixtures* (alapadatok), *functional* (működés), *mocks* (látszatobjektumok) és unit (egységteszt), és a tesztelési eljárás különböző területeire utalnak, amelyeket el kell végeznünk.

Alapadatok (Fixtures)

Mielőtt megkezdenénk a tesztelést, át kell hidalnunk egy bizonyos problémát: ha tesztelni szeretnénk az alkalmazásunkat, előtte fel kell töltenünk adatokkal az adatbázis táblákat. Mi több, azt szeretnénk, ha ugyanazzal a konzisztens adattal rendelkeznénk minden egyes teszt futtatása során, így pontosan következtethetünk az eredményre. A *Rails* alapadatok (*fixtures*) segítségével oldja meg a problémát. Ez egy szolgáltatás, amely automatikusan feltölti a tesztadatbázisunkat a tesztek futtatása előtt. A *blog/test/fixtures* könyvtárban hozhatunk létre ilyen alapadat meghatározásokat, általában *YAML* (*Yet Another Markup Language*) nyelven, amelynek jellemzője, hogy a struktúráját a soreleji behúzások határozzák meg.
Minden egyes tesztelni kívánt táblához egy *YAML* fájlt kell létrehoznunk. Mivel az adatbázisunk egyetlen táblából áll, egyetlen *YAML* fájlra lesz szükségünk, ez a *blogs.yml*. Ha szétnézünk a *blog/test/fixtures* könyvtárban, látni fogjuk, hogy már van ott egy ilyen fájl, amely bemutatja, hogy hogyan kell létrehoznunk a saját alapadatainkat, valamint hivatkozik az *ar.rubyonrails.org/classes/Fixtures.html* címen elérhető dokumentációra, arra az esetre, ha nem sikerül tökéletesen megérteni, hogy hogyan működnek az alapadatok.
Készítsünk egy vagy több bejegyzést a *blog.yml* fájlunkban, amelyek mindegyike megfelel az napló adatbázistábla egyetlen sorának, így ez a */blog/app/model* mappában meghatározott *Blog* objektumunk egy példányát is jelenti. Emlékeztetőül a tábla meghatározása a következőképp nézett ki:

```
CREATE TABLE Blogs (
id        SERIAL    NOT NULL,
title     TEXT      NOT NULL,
contents  TEXT      NOT NULL,
posted_at     TIMESTAMP
➥NOT NULL  DEFAULT NOW(),
 PRIMARY KEY(id)
);
```

Lássuk, hogyan tudunk két bejegyzést készíteni a táblához:

```
blog_entry_one:
    id:    1
    title:    My first entry!
    contents:  It was a dark
➥and stormy night, and
I forgot my umbrella.  So I
➥decided to tell the world
on my blog.
    posted_at: 2005-Sep-1
➥22:00:00
blog_entry_two:
    id: 2
    title:    My second entry!
    contents:  It was much
➥nicer this morning.
    posted_at: 2005-Sep-1
➥22:00:00
```

Ez egyenértékű két `INSERT` utasítással. Mivel a `INSERT` utasítás eleve szabványos *SQL* parancs, miért részesítjük mégis előnyben az alapadat szolgáltatás használatát?
Először is: az alapadat szolgáltatás biztosítja, hogy minden teszt ugyanazokkal a kiinduló adatokkal fusson le. Nincs annál idegesítőbb, amikor egy teszt azért nem fut le, mert egy duplikált adat miatt meghiúsul valamelyik egyediség feltétel.
A másik fontos dolog, hogy lehetővé válik számunkra, hogy ne csak az alkalmazás objektumain keresztül teszteljük az adatbázisunkat, hanem közvetlenül is. Vagyis a *Rails* betölti az adatokat a *YAML* fájl alapján, majd az általunk írt modell objektumon keresztül éri el azokat. Aztán újra betölti az adatokat a *YAML* fájl alapján, majd egy tömbön keresztül közvetlenül teszi elérhetővé az adatokat. Így össze tudjuk hasonlítani a két elérési módszert, megbizonyosodva arról, hogy az adatok helyesen betöltődtek, mielőtt kifinomultabb tesztelési lépésekhez fognánk.

Egységtesztek

Miután elhelyeztük az alapadatokat, elindíthatjuk az első egységtesztet. Az egységtesztek egymástól független metódusokat és objektumokat tesztelnek. Ha egy alkalmazás összetevői

átmennek egy teljes tesztsorozaton, az még nem jelenti azt, hogy az alkalmazás hibátlan. Az ilyen hibák általában az egyes egységek összekapcsolása során derülnek ki, ezeket más tesztekkel lehet felderíteni.
Nem túl meglepő módon az egységteszteket az alkalmazásunk mappájának test/unit almappájába kell tenni. A *Rails* minden általunk készített modell osztály számára automatikusan készít egy egységteszt állományt. Azaz a test/unit könyvtárban találunk egy *blog_test.rb* fájlt, amely az *app/model/blog.rb* fájlban meghatározott *Blog* osztálynak felel meg. (Emlékeztetőül: a *Rails* modell objektumok neve egyes számban szerepel, míg a hozzájuk tartozó adatbázistábla többes számban.) Alapértelmezetten a fájl az egységteszt vázát tartalmazza:

```
require File.dirname(__FILE__)
➥+ '/../test_helper'
class BlogTest < Test::Unit:
➥:TestCase
  fixtures :blogs
  def setup
@blog = Blog.find(1)
  end
  # Replace this with your real
# tests.
  def test_truth
assert_kind_of Blog,  @blog
  end
end
```

Az első sor a tesztgépezet elindításáért felelős, de ez most nem tart számot közvetlen érdeklődésre. A következő lépés azonban már igen: egy osztályt definiálunk (a *Rails* névkonvenciójának megfelelően *BlogTest*-nek nevezzük, amely a *blog_test.rb* fájlnak megfelelően ezt a nevet várja). A `BlogTest` a `Test::Unit::TestCase` osztály gyermeke, amely a *Rails* része. Számos, a teszteléshez kapcsolódó szolgáltatást biztosít a számunkra. A *BlogTest* meghatározása egy deklarálással kezdődik: meghatározzuk az alapadatokat (*fixtures*), amelynek a `:blog` nevet adjuk. Ez azt jelenti a *Rails* számára, hogy a `Blog` objektumunk `BlogTest` objektummal történő tesztelése előtt fel kell töltenie a teszt adatbázist, méghozzá a */test/fixtures/blogs.yml* fájl felhasználásával. Ha több alapadat állományt is fel szeretnénk használni, az alábbi módon adhatjuk meg azokat:

```
fixtures :blogs, :foo, :bar
```

Alapértelmezetten két metódust definiált számunkra a *Rails*, az egyik a `setup`, a másik a `test_truth`.
A `setup` metódus, ahogy a nevéből is látszik, előkészíti a tesztünket. Esetünkben meghívja a `Blog.find()` tagfüggvényt, paraméterként egy számot (1) adva át. Más szavakkal ez megpróbálja beolvasni azt az objektumot, amelynek az elsődleges kulcsa 1 (ez a `blog_entry_one` lesz), és beleteszi azt a `@blog` változóba. A *Perl* programozók számára talán nem árt megemlítenünk, hogy a *Ruby* nyelvben a `@blog` egy objektumpéldány változó, és nem feltétlenül tömb. A mi esetünkben a `@blog` tartalmazza a Blog objektum azon példányát, amivel a `find()` tagfüggvény visszatért.
A másik metódusban, a `test_truth`-ban a *Rails* rendszer előre meghatározott „tesztutasításainak” használva dolgozhatunk. A fenti esetben az `assert_kind_of_assertion` úgynevezett predikátumot (kijelentést, állítást) használjuk arra, hogy megbizonyosodjunk róla, hogy a `@blog` változó a `Blog` objektumunk egy példánya.

Tesztünk eme kezdetleges változatát az alábbi módon indíthatjuk:

```
$ ruby blog_test.rb
```

Ahogy a teszt lefut, egy állapotjelentést kapunk. Ha minden rendben ment, a kimenetnek valahogy így kell kinéznie:

```
[reuven@server unit]$ ruby
➥blog_test.rb
Loaded suite blog_test
Started
.
Finished in 19.066227 seconds.
1 tests, 1 assertions, 0
➥failures, 0 errors
```

Az első néhány alkalommal, amikor a tesztet lefuttattam, hibaüzeneteket kaptam. Az első probléma az volt, hogy elmulasztottam megváltoztatni az alapértelmezett ***blogs.yml*** fájlt, amely csak az elsődleges kulcs mezőket határozta meg. A ***Blogs*** táblában meghatározott sértetlenségi kényszer (***integrity constrait***) azonban meghiúsult, mivel az nem engedi meg `NULL` értékek használatát a ***title*** (cím) mezőben, így a ***PostgreSQL*** leállította a teszt futását.
A második kísérlet során a ***Rails*** hibát fedezett fel a ***YAML*** fájlomban, s figyelmeztetett, hogy a ***YAML*** formátum megköveteli a következetes sorbehúzásokat, amelyeknél tabulátor helyett szóközöket kell használni.
A ***Rails*** okosan különbséget tesz a hibákat és a sikertelen eredmények között; mindkettő a hibák közé van ugyan besorolva, így az egyes tagfüggvények helyett a tesztkörnyezet egészére koncentrálhatunk.
További teszteket végezhetünk el új tagfüggvények megadásával. Ellenőrizzük például azt, hogy a `@blog` objektumpéldányunk `title` változója megegyezik-e azzal, amit a ***YAML*** fájlban megadtunk:

```
def test_title
assert_equal @blogs["blog_entry
➥_one"]["title"], @blog.title
end
```

Ne feledjük, hogy a tesztünk a test_ karaktersorozattal kezdődik. Ez jelzi a ***Rails*** számára, hogy a tagfüggvény a tesztkörnyezet része. Mivel minden egyes metódust külön számol a teszt, valószínűleg az a legjobb megoldás, ha sok metódusunk van, amelyek mindegyike kevés állítást (predikátum) tartalmaz. Nincs technikai akadálya annak, hogy sok ilyen utasítást helyezzünk el ugyanabban a tagfüggvényben, de ilyet csak akkor csináljunk, ha utána lesz elég időnk kibogozni a teszteredmények alapján, hogy éppen hol is van a hiba.
A fenti példában az assert_equal állítással ellenőrizhetjük két mennyiség egyenlőségét. Figyeljünk nagyon oda arra, hogy nagyon hasonló neveket használjunk, így az kimeneten jobban értelmezhetők az eredmények.
Az egyik elem, amit az egyenlőségi vizsgálat során használunk, a `@blogs["blog_entry_one"]["title"]`. A `@blogs` tömböt automatikusan létrehozza számunkra a tesztkörnyezet, és (ahogy azt már korábban említettük) tartalmazza a ***YAML*** alapadat fájlban lévő összes adatot. Miként a `@blogs` tömb tartalmazza a teljes ***YAML*** definíciót, úgy a `@blogs["blog_entry_one"]` az első bejegyzésre mutat, s a `@blogs["blog_entry_one"]["title"]` nem más, mint az első bejegyzés címe (`title`).
A `@blog` példány ezzel ellentétben egyes számban van (mármint a neve), mivel ez csak egyetlen `Blog` objektumpéldányt tartalmaz. S mint minden jólnevelt ***ActiveRecord*** leszármazott esetében, tagfüggvény segítségével olvashatjuk ki az adatokat, esetünkben a `@blog.title` metódus segítségével. Egy szóval a teszt segít megbizonyosodnunk, hogy a két cím mező megegyezik.

További tesztek

Az itt részletezett két teszt csupán kis részét alkotja azoknak, amelyekkel az alkalmazásunkat tesztelhetjük. A ***Rails*** rengeteg állítást tartalmaz gyárilag, lehetővé téve, hogy a modell osztályokat minél változatosabb módon teszteljük. Ne feledjük, hogy a metódusokban meghatározott tesztek csak egy részét alkotják a lehetséges tesztelési eljárásoknak. Szükségünk van a megfelelő sértetlenségi kényszerek kipróbálására az adatbázisban, és sokféle bemenetre, hogy a lehető legkülönfélébb adatokkal teszteljük az alkalmazást. Egyik módja a nagy számú alapadat előállításának, ha dinamikusan készítjük azokat, ugyanazzal a szintaxissal (***Erb***, ***Embedded Ruby***), amit a ***Rails*** nézet komponensében használtunk.
Mint azt már említettem, a működési tesztek szintén nagyon fontos elemei bármely alkalmazásteszteknek. A működési tesztek, amelyek a ***Rails*** vezérlőket túráztatják, ugyanúgy működnek, mint az egységtesztek, csak itt a ***test/functional*** könyvtárban kell elhelyezni az egyes vezérlő osztályokhoz tartozó tesztobjektumokat, szintén test_ kezdetű tagfüggvények használatával minden egyes vezérlő minden egyes tagfüggvénye esetében. A modellek tesztelése biztosítja, hogy az alkalmazásunk robusztus legyen, a vezérlők tesztelése után nem számít, milyen bemenet érkezik a felhasználóktól a weben keresztül, tudjuk, hogy az alkalmazásunk elegánsan fogja az adatokat lekezelni.
Végezetül a ***Rails***-szel egyszerűen készíthetünk látszatobjektumokat (***mock***), így lehetővé válik annak színlelése, hogy létrejött egy objektum. Például amikor egy bankkártyás fizetésnél a fizetési folyamatot akarjuk lejátszani, vagy amikor 50000 felhasználónak szeretnénk levelet küldeni, anélkül, hogy ez valóban megtörténne.

Összegzés

A webalkalmazások idővel olyan naggyá és kifinomulttá válnak, amikor már nélkülözhetetlen a fegyelmezett tesztelési eljárás végrehajtása, hogy elkerüljük az előre nem látható problémákat. ***Ruby on Rails*** beépített tesztrendszerrel rendelkezik, amellyel egyszerűen készíthetünk és futtathatunk felhasználói teszteket minden adatbázisszinten, modell objektumokon, vezérlő objektumokon. Nem érne meglepetésként, ha számos ***Ruby*** fejlesztő válna a teszt alapú fejlesztések rajongójává, mert a ***Ruby*** és a ***Rails*** környezet egyszerűen kivitelezhetővé teszi a számukra.
Ha ***Rails*** segítségével történő fejlesztésre adjuk a fejünket, megéri majd a plusz időráfordítást az alkalmazás tesztelése: egyszerűen lezongorázhatjuk, és később hatalmas mennyiségű időt takaríthat meg számunkra.

Linux Journal 2006., 141. szám

A cikk forrásai:
➲ www.linuxjournal.com/article/8631

Revuen M. Lerner