Ruby on Rails

Zope kontra Ruby on Rails

Néhány éve, a *.com láz* idején én is és az alkalmazottaim is tele voltunk tanácsadói munkával. Alig győztük elvégezni az összes feladatot. E virágzás közepén nyilvánvalóvá vált, hogy a projektek egészen hasonlók, így tulajdonképpen az időnk (és az ügyfelek pénzének) nagy részét minden projektnél újra és újra a spanyolviasz feltalálására fordítottuk. Ideje volt elgondolkodni azon, hogy a kódok, vagy legalább a technikák újbóli felhasználása miként valósítható meg az egyes projektek között. Feltételeztük, hogy ezáltal nemcsak versenyképesebbek leszünk, hanem napi feladataink is érdekesebbek lesznek. Hiszen sokkal érdekesebb dolog az egyes projektek új és különböző elemein dolgozni, mint állandóan szinte ugyanazt a felhasználói csoport jogosultsági rendszert létrehozni.

Hamar elvetettük a közös kódrendszer ötletét, részben azért, mert más fejlesztők nem csak, hogy megoldották már e problémák többségét, de közzé is tették megoldásaikat nyílt forráskódú licencekkel. Az évek során számtalan különböző projekthez alkalmaztunk webfejlesztő keretrendszereket – közülük be is mutattam néhányat.

Bárki, aki dolgozott már ilyen keretrendszerekkel, az tudja, hogy semmi sincs ingyen. Szinte kivétel nélkül mindegyik keretrendszer ránk erőltet egy bizonyos módszert, a kompromisszumos megoldások pedig vagy passzolnak fejlesztési elképzeléseinkhez, vagy nem. Jómagam az évek során többet is használtam e keretrendszerek közül, és bár akadt bennük kedvemre való tulajdonság, sosem éreztem, hogy szabadon engedhetem a fantáziámat. Ilyen előzmények után engem is roppantul felizgatott az új jövevény, azaz a ***Ruby on Rails***. Mint az kiderült, a ***Rails*** egy olyan keretrendszer, amelynek sok különböző funkciója van, ilyenek például az objektumrelációs leképező, az ***MVC (Model View Controller)*** tervezés, és a beépített teszteléstámogatás. A ***Rails*** nagyon népszerűvé vált első megjelenése óta, és bár itt-ott még akad rajta csiszolni való, azért kétségkívül kitűnő rendszer. Emellett népszerűsége abban is megnyilvánul, hogy az új keretrendszereket fejlesztők a ***Rails*** rendszert tekintik példaképüknek, ***Rails***-szerűt akarnak alkotni, vagy annál is jobbat.

Hogy a ***Rails*** miért érdekel ennyi embert? Vagy, fontosabb kérdés, hogy érdemes-e ezt használnunk következő web vagy adatbázis projektünkhöz? Végül, milyen kompromisszumokat erőltet ránk, és ezek hogyan befolyásolják döntéseinket?

A Rails előtt

Webes alkalmazásokat fejlesztek akkortól kezdve, amikor még a *„webes alkalmazás”* kifejezés e-mailküldő ***CGI*** programokat jelentett, nem egy milliárdos iparágat. Minden keretrendszer, amit használtam, letett valamit az asztalra, és így vagy úgy könnyebbé tette számomra az alkalmazásfejlesztést. Ugyanakkor viszont mindegyik valamilyen kompromisszumra kényszerített pusztán azért, hogy a rendszerrel dolgozni tudjak.

Az első általam használt keretrendszerek egyike a ***Mason*** volt, amely elkényeztetett rugalmasságával és könnyen használhatóságával. ***Perlben*** íródott, és a ***mod_perl***-hez és az ***Apache***-hoz tervezték. Telepítése és beállítása roppant egyszerűvé vált az évek során, persze ***mod_perl*** és ***Apache*** még szükséges hozzá a kiszolgálón. Továbbá a ***Mason*** tökéletesen együttműködik több, a ***CPAN***-on elérhető ***Perl*** modullal, azokkal a fejlett és hatékony fejlesztőeszközökkel, amelyeket a ***Perl*** közösség létrehozott az évek alatt. Amikor ***Perllel*** kell online rendszert létrehoznom, mindenképp a ***Masonhoz*** fordulok. Hátránya viszont, hogy kevés komponens érhető el hozzá. Persze megírhattam volna a felhasználói fiók-, csoport- és jogosultságkezelést. De írtam volna meg külön, minden egyes projekthez? És bár a ***Mason*** sablonjai igen hatékonyak, töménetlen ***Perl*** kódot és szokatlan szerkezetet tartalmaznak, ez pedig riasztóan hathat bizonyos fejlesztőkre.

Ezután megpróbálkoztam az *OpenACS*-sel. Ez egy nyílt forrású közösségi rendszer, és jóval kevesebb felhasználója van, mint a *Masonnak*. Az *OpenACS* sablonrendszere viszont minden megjelenített oldalt két részként kezel. Az egyik *Tcl*-ben íródott, a másik pedig módosított *HTML*, melyeket egy meghatározott „szerződés" kapcsol össze. Továbbá az *OpenACS* rendelkezik egy szabványos adatmodellel, amelyet a rendszer különböző alkalmazásai használnak. Nem kell olyasmivel bajlódnunk, mint egy ilyen regisztráció modul megszerkesztése, mert a rendszerben benne foglaltatik már egy. Továbbá fórumokat, blogokat vagy naptárakat sem kell magunknak létrehoznunk, mert ezek is adottak.
A központosított, szabványos adatmodell és az adminisztratív alkalmazások bizonyára igen vonzóak, de az *OpenACS*-nek is megvannak a maga problémái. A legnagyobb talán az adatmodelljének szokatlan megvalósítása: relációs adatbázissal követi a hierarchiákat és az objektumokat. A rendszernek van egyfajta intellektuális varázsa; a relációs adatbázisok gyorsak, szilárdak és olcsók, és az objektumorientált programozás segítségével bármilyen adattípus modellezhető. Viszont e kettő kapcsolata azzal jár, hogy a legegyszerűbb *OpenACS* alkalmazás is igen összetett lesz. Sőt, az *OpenACS* közösség növekedésével igen nehézzé vált az adatmodellek kis méretének megőrzése, hiszen a felhasználók igényei igencsak különbözőek.
Foglalkoztam *Zope*-pal is. Ez egy főleg *Pythonban* megírt webfejlesztői keretrendszer. Nagy és erős közössége van, fejlesztése folyamatos, és a *Zope Corporation* folyamatosan bővíti. Több ígéretes funkcióval rendelkezik, például tartalmaz egy nagy teljesítményű fejlesztői környezetet, alosztályokra tagolt „termékeket", amelyeket különkülön felhasználhatunk és bővíthetünk, valamint egy kifinomult felhasználó-, szerep- és jogosultságkezelő rendszert. A *Zope* elsőként állt elő az objektumok közvetlen megjelenítésének ötletével, amelyben egy *URL* adja meg az adott objektumon meghívandó tagfüggvényt. Tehát a */Foo/bar URL* azt jelenti, hogy meghívjuk a *Foo.bar*-t, bemenetet a *HTTP* kéréssel biztosítunk, a kimenet pedig *HTTP* válaszként jelenik meg.
A leggyakoribb kifogás a *Zope*-pal szemben az, hogy nehéz megtanulni. Nos, ez igaz is; beletelt egy időbe, mire megértettem magát a *„Zope zent"*. Szerintem viszont az egyszerűnek vélt dolgok is igényelnek a tökéletes működéshez némi kódakrobatikát – ami lehet a kódírási stílusom reflexiója, de lehet a *Zope*-tervezés remeke és a *Zope*-beli objektumhasználat átható módja is.
Régebben a *Zope* tervezők saját objektum-orientált adatbázis építésével próbálták elkerülni a relációs adatbázisokkal kapcsolatos problémákat. Ez egyrészt számos nagy előnnyel járt a *Zope* riválisaival szemben, mert a rendszer változtatásait vissza lehet vonni, beépített jogosultságkezelő és -tároló rendszere van, amelyek a *Zope* adattípusaihoz tökéletesen illeszkednek.
A relációs adatbázisok miatt viszont elkerülhetetlen az *SQL* használata.
A *Zope DTML* sablonnyelve lehetővé teszi az adatbázisokhoz való kapcsolódást és a velük való munkát.
Ez azzal jár, hogy a *Zope* termékek többségének – és az általam használt összes termékének – koordinálniuk kell a relációs és objektum adatbázisokat. Általában vége ez a megoldás nem olyan szörnyű, de a végén mindig azon töprengtem, hogy miért kell így túlbonyolítani a dolgokat. És minden kifinomultsága ellenére azt vettem észre, hogy folyton ugyanazokat a létrehozó-, frissítés- és törlés tagfüggvénytípusokat és sablonokat hoztam létre, újra és újra.

Hogy jön a képbe a Rails?

Nem meglepő tehát, hogy a *Ruby on Rails* részben a korábbi keretrendszerek által meghagyott űrt tölti be. Fenti leírásaim talán rávilágítanak arra is, hova kéne fejlődnie a *Railsnek* ahhoz, hogy igazán sikeres legyen.
A *Rails* legvonzóbb tulajdonságai a sebessége és az az egyszerűsége, ahogyan a fejlesztők a relációs adatbázisokkal kommunikáló kódot írhatnak. És, bár egyáltalán nem nyűgöznek le a negyed óra alatt létrehozható demók, tapasztalataim mégis azt igazolták, hogy ezek a demók igenis valószerűek. Ez azért van, mert a *Rails* feltételezi, hogy adatbázistábláinkat az ő szabályai alapján hozzuk létre. Ilyen szabályok például a többes számú táblanevek, az *id* nevű *ID* mezők és az *_at*-tel végződő időpont és dátum mezők.
Ha betartjuk ezeket a szabályokat, rácsodálkozunk majd, hogy milyen nevetségesen kevés kóddal milyen sok szokványos szituációt kezelhetünk. Csakugyan, azon kapjuk majd magunkat, hogy egy maroknyi kóddal elintézünk egy rakás modellobjektumot, mert a *Rails Active Record* leképezője elvégzi helyettünk a feladat nagy részét.
Ez azt jelenti, hogy a *Rails* alkalmazásokhoz szükséges munka jó része a vezérlőkre (azok az objektumok, amelyek tagfüggvényeit az *URL*-eken keresztül tettük hozzáférhetővé) és a nézetre (*Ruby-HTML* keverék sablonok) koncentrálódik. Minden vezérlőfüggvény előállíthatja saját kimenetét egyszerű szövegként vagy *HTML*-ként, vagy a nézet könyvtárban ugyanazon nevű sablonján keresztül.
A *Railsben* van egy beépített funkció is, amellyel fájlt küldhetünk a felhasználónak, így beállíthatunk bináris fájl letöltést anélkül, hogy a *MIME* típusok és fájlnevek meghatározásának szintaxisára figyelnünk kéne.
A *Zope* védelmezői kétségkívül azt fogják mondani, hogy ezek a funkciók a *Zope*-ban is megtalálhatók, ráadásul évek óta. Ez igaz is – de az újoncoknak őrjítően bonyolult lehet kiókumulálni, hogy hogyan kell használni őket és pontosan mit is csinálnak. Azzal, hogy néhány ésszerű alapbeállítást biztosít jó pár művelethez, és engedi ezeket az alapbeállításokat módosítani, a *Railsnek* sikerült az egyszerű eseteket egyszerűnek meghagyni, és a bonyolult esetek is csak mértékkel bonyolítani. Ezenkívül a *Rails scaffolding* generátora (állványzat generátora) bőségesen elég alapvető vezérlőt és sablont biztosít, amelyeket bárki felhasználhat anélkül, hogy órákat kéne eltöltenie különböző kódfájlok létrehozásával és módosításával.
Az ésszerű alapbeállítások miatt a kezdő *Rails* fejlesztőnek csak néhány objektummal és tagfüggvénnyel kell tisztában lennie egy alkalmazás létrehozásához. Ez teljesen az ellenkezője a fentebb bemutatott keretrendszereknek, melyek nagy objektum- és tagfüggvényismeretet igényelnek olyan részletekbe menően, hogy hogyan kell kapcsolódniuk az egyes elemeknek ahhoz, hogy működjenek is. Az igaz, hogy a *Rails* bővül és finomodik, s ez azzal a kockázattal jár, hogy ugyanúgy felduzzad, mint összetettebb társai.

Eddig még sikerült elkerülnie az ilyen problémák többségét, és úgy tűnik, a fejlesztők ragaszkodnak az egyszerűség megőrzéséhez.

Érvek és ellenérvek

Ahogy azt fentebb is írtam, mindegyik webfejlesztő keretrendszernek megvannak a maga tervezési kompromisszumai. Mi hiányzik még a ***Rails***-ből? Mit kell figyelembe vennünk, annak eldöntéséhez, hogy ***Railst*** vagy valami mást használjunk-e alkalmazásunkhoz?

Először is, a ***Railshez*** elengedhetetlen a ***Ruby*** nyelv ismerete. Mielőtt elkezdenék a ***Railsszel*** dolgozni, előtte mindig foglalkozom egy keveset a ***Ruby***-val, és jól szórakozom. Persze nyilván sok programozó kifogásolja a ***Ruby*** bizonyos elemeit, például a szintaxist és az objektummodellt. A ***Ruby*** a ***Perlhez*** és a ***PHP***-hoz képest éretlen, ha a külső gyártótól származó kiegészítő könyvtárakról van szó. Ez azt jelentheti, hogy néhány speciális eljárást magunknak kell megírnunk, és nem számíthatunk a közösség támogatására. Végül, a ***Ruby***-ból hiányzik a ***Unicode***-támogatás, így többnyelvű weboldalaknál egyelőre nem alkalmazható.

Ha a mérleg nyelve a ***Ruby*** felé billen, és ezt szeretnénk webfejlesztő nyelvként használni, miért ne használhatnánk a ***Railst*** is? Nem értek egyet azzal a kijelentéssel, hogy minél kisebb a fejlesztőcsapat és minél nagyra törőbb a projekt, annál inkább érdemes ***Railst*** használni. Egészen kicsi projekteknél a ***Ruby*** csak felesleges többletteher, és sokkal kézenfekvőbb megoldás a ***CGI*** és a ***PHP***. Onnantól kezdve viszont, hogy relációs adatbázist használunk egynél több táblával, a ***Rails*** hasznunkra lehet.

A ***Ruby*** és a ***Rails*** kis létszámú programozói csoportok, valamint egyedül dolgozók számára terveztetett. Ahhoz, hogy többen dolgozzanak ugyanazon a ***Rails*** projekten, nagyfokú összehangoltság kell, hogy nehogy módosítsák egymás fájljait. Az, hogy mindegyik ***Rails*** alkalmazás egyetlen könyvtárban kap helyet, növeli az ilyen zűrzavar valószínűségét.

A nagyobb projektek viszont inkább a nagyobb keretrendszerekkel járnak jól, például a ***Zope***-pal, vagy valamelyik, nemrégiben megjelent ***Java*** alapú rendszerrel. A ***Javával*** nekem a legnagyobb gondom az volt, hogy kicsit kényelmetlen és lassú, főleg ha a ***Perlhez***, a ***Pythonhoz*** vagy a ***Ruby***-hoz hasonlítjuk. Ha több programozó dolgozik egy nagy projekten, előnyt jelenthet a több fordítás közbeni ellenőrzés, a világos meghatározások és a védelmek – ezek a ***Ruby***-ból hiányoznak. Talán webtervezőink igényeit is figyelembe kell vennünk, mikor a ***Railsről*** mint környezetről gondolkodunk. Néhányuknak ínyére van, ha a kóddal a sablonokon belül dolgozhatnak, némelyikük viszont irtózhat tőle és képes törölni vagy megváltoztatni a kódot. Még mindig úgy vélem, hogy a ***ZTP*** és az ***OpenACS*** jobb sablonrendszereket kínálnak. Örömmel fogadtam viszont a ***Liquid*** megjelenését. Ez utóbbi egy sablonrendszer a ***Railshez*** a ***PHP Smarty***-jának stílusában. Régebb óta lenyűgöz a ***Smarty***, és úgy vélem, a ***Liquiddel*** közelebb került a ***Rails*** nagyobb rendszerekben való bemutatkozása.

Előnyei és hátrányai is vannak annak, hogy a ***Rails*** alkalmazások egyetlen könyvtárban, szöveges fájlokból állnak. Egyik nagy előnye, hogy minden könnyedén tárolható ***CVS***-ben vagy bármilyen verziókövető rendszerben; az alkalmazás telepítése egy új helyre így ugyanolyan egyszerű, mint a kód ellenőrzése. Viszont ez a megközelítés azzal jár, hogy valamivel nehezebb ugyanannak az alkalmazásnak több példányát létrehozni ugyanazon a kiszolgálón, mint ugyanezt az ***OpenACS*** csomagjaival vagy a ***Zope*** termékekkel megvalósítani. Mindig létrehozhatunk több másolatot a ***Rails*** alkalmazás könyvtárfájáról, de ugyanarról a csomagból több példány nem lehet. Korábban már elárultam, hogy eredetileg nagyon vonzott az ***OpenACS*** az egyszerű, szabványosított adatmodellje miatt. Most már tudom, hogy egy ilyen erősen központosított adatmodell majdnem mindig alkalmatlan, de még mindig azon gondolkodom, hogy a ***Railsnek*** miért nincsenek beépített jogosultságai vagy regisztrációs rendszere. Feltételezem, a válasz az, hogy a növekvő számú ***Rails*** bővítőmodulok miatt, amelyek között számos, a már meglevő ***Rails*** alkalmazásokba beépíthető regisztrációs rendszer akad.

Az is jó lenne, ha ezen a területen több szabványosítás történne – de valószínűleg ez meddő ötlet.

Az örökölt kódok minden környezetben, így a ***Railsben*** is problémát jelentenek. A ***Rails*** annyira új, annyira más, jelentheti ez azt, hogy az adoptálás miatt mindent el kell dobnunk? Elképzelhető, de nem szükségszerű. Ahelyett, hogy újraírtam volna ***Ruby***-ban egy érett ***Perl*** könyvtárt, csak egy burkolót írtam ***XML-RPC***-vel. A ***Ruby***-nak van egy ***XML-RPC*** ügyfele, ezt használtam ***Rails*** alkalmazásomban a ***Perl*** kóddal való kapcsolattartásra. Mindez simán és könnyen ment, így ez azt jelenti, hogy egyszerre élvezhetem a ***Rails*** és a ***CPAN*** előnyeit. Mivel a ***Railsben*** az adatbázis elnevezési konvenciók felülírhatók, ezért az már meglevő adatbázisokkal is használható, így a felhasználóknak nem kell új adatbázissémákat létrehozniuk a ***Rails*** szabályainak megfelelően.

Összefoglalás

Vannak, akik a ***Rails*** eljövetelét várják, mellyel új időszámítás kezdődik a webfejlesztésben. Jómagam úgy látom, hogy a ***Rails*** új szabványt teremtett, már ami a webfejlesztői keretrendszerek szolgáltatásait illeti. A fejlesztők nem élhetnek tovább abban a tévhitben, hogy terjedelmes kód kell mind a ***„Hello, World!"*** program megírásához, mind alapvető adatbázis-műveletek kezeléséhez.

A ***Rails*** arról is kezdi meggyőzni a fejlesztőket, hogy a megszokások útját állhatják a gyors, hibamentes fejlesztésnek. Évekbe telt, mire a programozók egyetértettek abban, hogy a szemétgyűjtéses nyelvek előrelépést jelentenek a `malloc()`-hoz képest, de szintén sokáig tartott, mire rájöttünk, a szabványok jobbak a beállításfájloknál.

De a ***Rails*** népszerűsége azt jelzi, hogy lassan megérünk egy ilyen változásra. Bár egyetlen webfejlesztői keretrendszer sem tökéletes, szerintem a ***Rails*** megüti a mércét a legtöbb, az évtizedben megírt alkalmazásomhoz. Mind a ***Ruby*** (a nyelv), mind a ***Rails*** (a keretrendszer) fejlődésben van, de ha ilyen, viszonylag éretlen állapotukban így működnek, milyenek lesznek majd készen? – Alig győzöm kivárni.

Linux Journal 2006., 142. szám

Reuven M. Lerner

Forrás:
➲ ***www.linuxjournal.com/article/8693***