Szakdolgozat

Gulyás István

Debrecen

2008

Debreceni Egyetem

Informatikai Kar

Alkalmazásfejlesztés Ruby on Rails környezetben

Témavezető:

Dr. Juhász István
egyetemi adjunktus

Készítette:

Gulyás István
programtervező informatikus (BSc)

Debrecen
2007

Tartalomjegyzék

Bevezető 3
A Ruby nyelv 4
  Ruby sajátosságok 4
Ruby on Rails 7
Active Support 8
  Kollekciók és tömbök 8
  Sztringek 8
  Dátumok és számok 9
  Unicode kiegészítések 9
Migrációk 11
  Migrációk létrehozása 11
  Migrációk futtatása 12
ActiveRecord 13
  Konfiguráció 13
  Attribútumok 14
  Elsődleges kulcsok 16
  Csatlakozás az adatbázishoz 17
  CRUD – Create, Read, Update, Delete 18
  Mágikus oszlopnevek 21
  Táblák közötti kapcsolatok 22
  Validáció 26
  Callback metódusok 28
ActionController 31
  Mi is az ActionController? 31
  Kontrollerosztályok és akciómetódusok 31
  Munkamenet-kezelés 34
ActionView 36
  Sablonok 36
  Helper metódusok 36
  Részsablonok 37
Összefoglalás 38
Irodalomjegyzék 39

Bevezető

Egyetemi tanulmányaim mellett évek óta szabadúszó webes alkalmazásfejlesztőként dolgozom. Az évek során kialakult némi rálátásom a manapság széles körben használt nyílt forráskódú webes alkalmazásfejlesztői platformokra. Kezdetben elsősorban PHP nyelvvel foglalkoztam. A PHP mellett szólt a könnyű tanulhatóság és a PHP fejlesztők hatalmas közössége, és a széles körben elérhető hosting szolgáltatás. Az utóbbi években, amióta az egész internet a web2.0 lázában ég, egyre többet lehetett hallani egy új, gyorsabb, rugalmasabb és kényelmesebb keretrendszerről, amiben minden eddiginél hatékonyabban lehet webalkalmazásokat fejleszteni.

„Webfejlesztés, ami nem fáj." – így hirdeti magát a Ruby on Rails a hivatalos honlapján. Emellett büszkén hangoztatja, hogy programozói boldogságra és fenntartható produktivitásra van optimalizálva. Ez persze elsősorban marketingszöveg, de amióta én is aktívan foglalkozok Ruby on Rails alapú webes alkalmazásfejlesztéssel, saját tapasztalatomból látom, hogy a keretrendszerben webalkalmazást fejleszteni tényleg nem fáj, vagy legalábbis nem annyira. A Ruby on Rails közösség már-már vallásos elhivatottságán is látszik, hogy talán mégis sikerült a keretrendszert a programozói elégedettséget is középpontban tartva megalkotni. Konkrét projekteken dolgozva persze már szembekerül a programozó olyan kihívásokkal, amiket nem említenek a könyvek és tutorialok. Ilyenkor látszik a keretrendszernek az a hátránya, hogy a Ruby on Rails közösség még mindig nagyságrendekkel kevesebb tagot számlál, mint mondjuk a PHP fejlesztők közössége. És mivel a nyílt forráskódú rendszerek esetén tipikusan közösségi támogatásra számíthat egy programozó, így néhány problémára nagyon nehezen találhatunk megoldást az interneten böngészve.

Mindenesetre a Ruby on Rails-t használók tábora rohamosan növekszik, maga a keretrendszer fejlesztése gyors ütemben halad, így a Ruby on Rails magában hordozza a potenciált arra, hogy néhány éven belül széles körben használt és általánosan elismert platformmá váljon a webalkalmazások fejlesztésében. Ezért választottam szakdolgozatom témájának ezt a még viszonylag új, de korához képest óriási érdeklődéssel és nagy várakozásokkal kísért webes alkalmazásfejlesztő keretrendszert.

A Ruby nyelv

A Ruby nyelvről önmagában könyvet lehetne írni, így nem fér a fejezet keretei közé, hogy a nyelv összes szolgáltatását részletesen tárgyaljam. Igyekszem azonban kitérni a nyelv legfontosabb jellemzőire, illetve megpróbálom kiemelni a szokatlanabb megoldásokat, amik újszerűnek, egyedinek tűnnek a nyelv tanulása során. A hely szűkössége miatt csak kevés konkrét szintaxispéldát mutatok meg, a www.ruby-doc.org webhelyen találhatók részletes leírások és gyakorlati példákat felvonultató tutorialok.

A Ruby egy objektum-orientált nyelv, ami az egységesség elve szerint készült, tehát benne minden objektum. Interpreteres nyelv, ami erős hasonlóságot mutat a Perl, Python és Smalltalk programozási nyelvekhez. Hozzáférést enged a rendszer szolgáltatásaihoz, így teljesértékű önálló programok is megírhatók a segítségével, de használható beágyazott, vagy akár interaktív módon is. Az Interaktív Ruby Interpreter (IRB) kiváló eszköz arra, hogy kipróbálhassuk a nyelv különféle szolgáltatásait, de hasznos lehet a már elkészült programunk tesztelésére és belövésére egyaránt.

Ruby sajátosságok

Mint már említettem, a Rubyban minden nyelvi konstrukció objektum. Egy numerikus vagy sztring literál is az, a más nyelvekben megszokott ponttal minősítve érhetők el ezen objektumok adattagjai, így hívhatók meg a metódusok. A nyelv reflektív, egy objektumnak az `class` metódussal kérdezhetjük le az osztályát, az `methods` az objektum metódusainak neveit tartalmazó tömbbel tér vissza.

A Ruby tartalmaz egy szokatlan nyelvi elemet, amit szimbólumnak nevez. A szimbólumok gyakorlatilag nevek, amelyek kettősponttal kezdődnek. Felfoghatók konstans sztring literálokként, hiszen a nevükön kívül semmi mást nem jelentenek. Használatuk leggyakrabban hash-ek indexelésénél és metódushívásoknál név szerinti paraméterátadásnál fordul elő.

A Perlhez vagy a Pythonhoz hasonlóan Rubyban is megjelennek a tömbök és hash-ek. Tömböt úgy hozhatunk létre, hogy tartalmát szögletes zárójelek között felsoroljuk. Egy tömb tetszőleges számú objektumot tartalmazhat, és az objektumok tetszőleges típusúak lehetnek. A tömbök indexelése nullától kezdődik. A címzésnél használhatunk negatív indexeket is, ilyenkor a Ruby interpreter a tömb végéről számol visszafelé. Lehetőség van a tömb egy

összefüggő szakaszának a megcímzésére is, ilyenkor a nyitó és a záró indexet vesszővel elválasztva kell megadni szögletes zárójelben. Ez esetben zárt intervallumot kapunk vissza, tehát a megadott két korlátindex által címzett elemek is részei lesznek a visszatérő tömbnek. A hash-ek hasonlóak a tömbökhöz, a különbség annyi, hogy az elemeket nem rendezett numerikus értékekkel (index) lehet azonosítani, hanem kulcsokkal. Hash-t a kulcs-érték párokat kapcsos zárójelben felsorolva, `=>` jellel összerendezve, vesszővel elválasztva adhatunk meg. A tömbökhöz hasonlóan mind a kulcsok mind az értékek tetszőleges objektumok lehetnek.
A feltételes utasítások a más nyelvekben megszokott módon működnek, persze van néhány különbség. Van `unless` utasítás, ami tulajdonképpen az `if` ellentéte, és csak az olvashatóságot segíti. A másik érdekesség, hogy a feltételes utasítások írhatók a kifejezések után, például:

```
print ”x nagyobb, mint 5” if x > 5
```

A Ruby metódusai blokkal is paraméterezhetők. Például a

```
3.times { print ”hello” }
```

kódrészlet kimenete 3 darab `hello` lesz. Ebben a kódban a `3` egy `Fixnum` típusú objektum, aminek a `times` metódusát hívtuk meg a blokkal paraméterezve. Ilyen blokkokkal paraméterezett metódusokat használunk tömbök és hash-ek bejárására is. Ebben az esetben az átadott tömb is kap vissza paramétert az objetum metódusától. Ezeket a paramétereket | jelek között adjuk meg a tömb elején. A

```
tomb = [”elso”, ”masodik”, ”harmadik”]
tomb.each { | elem | print ”#{elem}\n” }
```

kód kiírja a tömb elemeit a standard kimenetre. A fenti kódban a `#{elem}` jelzi a Ruby interpreternek, hogy az `elem` itt egy változó, ami behelyettesítődik a sztringbe. Több paramétert `|` jelek között vesszővel elválasztva adhatunk meg. Például:

```
h = { ”a” => ”elso”, ”b” => ”masodik”, ”c” => ”harmadik” }
h.each { | kulcs, ertek | print ”#{kulcs}: #{ertek}\n” }
```

Ilyen metódust a következőképpen hozhatunk létre:

```
def  fibonacciSzamig(max)
  elso, masodik = 1, 1 #párhuzamos értékadás
  while elso <= max
    yield elso #itt adódik át a vezérlés a blokknak
    elso, masodik = masodik, elso + masodik
  end
end

fibonacciSzamig(1000) { | szam | print szam, ”, ” }
```

A metódusban a vastagon kiemelt `yield` parancs hívja meg a paraméterként kapott blokkot, a `yield` után következő paraméter (`elso`) adódik tovább a blokk felé.

Ruby on Rails

A Ruby on Rails egy integrált, nyílt forráskódú, Ruby nyelven alapuló webes keretrendszer, amellyel tömören, gyorsan és rugalmasan lehet webalkalmazásokat fejleszteni. A Ruby on Rails a 37Signals cég által készített Basecamp projekt-menedzsment rendszerből vált ki és lépett elő önálló keretrendszerként. Középpontjában az ismert Model-View-Controller tervezési minta áll. A Ruby on Rails a tervezési mintát három nagyobb modullal valósítja meg a mindent összefogó rétegek mellett. Ezek az ActiveRecord, az ActionController és az ActionView.

A Ruby on Rails filozófiájában alapvető a „ne ismételd önmagad” („don’t repeat yourself”, DRY) és a „konvenció konfiguráció helyett” („convention over configuration”, CoC) alapelvek. A „konvenció konfiguráció helyett” alapelv azt jelenti, hogy a keretrendszer konvenciók szerint megköti egy alkalmazás fejlesztésének a módját. A fejlesztőnek csak az alkalmazás a keretrendszer konvencióitól eltérő részeit kell bekonfigurálnia, a konvencióknak megfelelő részek alapértelmezés szerint működnek. A „ne ismételd önmagad” alapelv arra utal, hogy a minden információ csak egy jól meghatározott helyen jelenik meg. Ruby on Rails keretrendszerben programozva garantáltan elkerülhető, hogy ugyanaz a kód több helyen is megjelenjen az alkalmazásban.

A következő fejezetekben a Ruby on Rails egy-egy modulját fogom részletesen tárgyalni.

Active Support

Az Active Support egy általános eszközkönyvtár, ami minden Rails komponens számára egyaránt hozzáférhető. Ezeket az eszközöket elsősorban maga a keretrendszer használja. Ezeken felül - mivel a Ruby nyelv erre lehetőséget ad - az Active Support az alap Ruby API osztályait is kibővíti számos érdekes és hasznos funkcióval. Ezen kiegészítések közül csak néhány gyakrabban használt megoldást tárgyal ez a fejezet.

Kollekciók és tömbök

Az Active Support több hasznos metódussal is kiegészíti az `Enumerable` osztályt. Ilyen metódusok a `group_by` és az `index_by`, amelyek egy tömb elemeinek megadott szempont szerinti csoportosítására és indexelésére valók. A `sum` metódus egy tömb elemenek összegét állítja elő. A `to_sentence` metódus vesszővel elválasztva sorolja fel a tömb elemeit, az utolsó elem elé még az „and” szót is beilleszti, ha azt megfelelő paraméterezéssel nem tiltjuk le.

Sztringek

A sztringek kezelését is nagyban megkönnyíti az Active Support. A String osztály elsőre kicsit barátságtalan metódusait kiegészíti olyan alapvető funkciókkal, mint a részsztring-képzés (`at`, `from`, `to`, `first`, `last`) és a sztring kezdetét és végét vizsgáló `starts_with?` és `ends_with?` (ez utóbbi kettő esetén a kérdőjel része a metódusnévnek, ugyanis a Ruby elnevezési konvenciók szerint igaz vagy hamis értéket visszaadó függvények neveinek kérdőjellel kell végződni).
Ezeken kívül az Active Support segítségével könnyedén alakíthatók angol szavak nyelvtani egyes számból többes számba, és fordítva a `pluralize` és `singularize` metódusok használatával. A Rails elég sok rendhagyó esetet is felismer, például a `"person".pluralize` hívás helyesen a ”people” sztringet adja vissza, de könnyedén implementálhatók újabb szabályok is olyan rendhagyó esetekre, amire a Rails nincs felkészülve. Ez a funkció nem csak angol nyelven készült honlapok esetén nyújthat nagy segítséget, ugyanis a Railsben az adatbázis tábláit konvenció szerint angol többes számú főnevekkel, míg a hozzájuk tartozó modell-osztályokat a táblanevek egyes számú

megfelelőivel kell nevezni. A kettő közötti kapcsolatot is a fenti egyes- és többes szám képző metódusok segítségével ismeri fel a Rails. De erről részletesebben csak az ActiveRecordot tárgyaló fejezetben.

Dátumok és számok

A `Time` osztály rengeteg metódussal kiegészült, amelyek segítik relatív dátumok kiszámítását és a dátumformázást. Ezek közül sokan szinonimák, de nagyban segítik a kód olvashatóságát. Mivel a legtöbb metódus neve önmagáért beszél, csak felsorolás szinten néhány példa: `last_month`, `last_year`, `at_beginning_of_day`, `at_beginning_of_week`, `at_beginning_of_month`, `at_beginning_of_year`, `monday`, `next_week`, `seconds_since_midnight`, `tomorrow`, `yesterday`, `advance` és még sok más. Például a `Time.days_in_month(2, 2000)` visszaadja, hogy 2000 februárja hány napos volt.

A `Fixnum` osztály `even?` és `odd?` metódusai egy szám páros vagy páratlan voltát vizsgálják. Az `ordinalize` angol sorszámnévvé alakítja az egész számokat. Számos mértékváltó metódus áll a rendelkezésünkre, például a `20.megabytes` eredménye 20971520. Hasonló átváltó metódusok léteznek idő átváltására is, amelyek másodpercekre váltják a megfelelő mennyiségeket, viszont a `months` és a `years` metódusok pontatlanok, mert 30 napos hónappal és 365 napos évvel számolnak, nagyobb időintervallumok pontos kiszámítására a `Time` osztály fentebb tárgyalt kiegészítő metódusai alkalmasabb eszközök. A `Fixnum` osztály használható az idő relatív kiszámítására. Például könnyen megtudhatjuk, hogy mennyi lesz az idő 40 perc múlva a `40.minutes.from_now` hívás segítségével. Az előbbi forma – ami az egyik személyes kedvencem az Active Support kiegészítései közül – is mutatja a Ruby és a Ruby on Rails könnyen olvasható forráskód filozófiáját.

Unicode kiegészítések

A Ruby nyelvet eredetileg Matsumoto Yukihiro japán programozó tervezte és készítette. Japánban az Unicode szabvány sokáig nem volt igazán elterjedt, a mai napig rengeteg japán programozó használ inkább valamilyen alternatív karakterkódolást az Unicode helyett. Japánban a legtöbb e-mail JIS, a weboldalak többsége Shift JIS míg a mobiltelefonok leginkább valamilyen EUC kódolást használnak. Így nem meglepő módon a Ruby nyelv sem

támogatja az Unicode karakterkódolást (a Ruby 2.0 várhatóan natív Unicode támogatással fog megjelenni). Az Active Support Unicode támogatással is kiegészíti a Ruby sztringjeit. Ez a kiegészítés nem cseréli le teljesen a Ruby sztringkezelő függvényeit, ezért néhány esetben nem várt eredményeket kaphatunk több bájton tárolt karakterekből álló sztringek kezelésénél. Például a `length` metódus is hibás eredményt adhat vissza, több bájtos karakterek esetén a valós sztringhossznál nagyobb értéket. A probléma ott rejlik, hogy a Ruby on Rails az UTF-8 kódolást támogatja, amiben néhány karakter 8, néhány 16 bites reprezentációval rendelkezik. A `length` metódus a bájtok számát adja vissza, így a kétbájtos karaktereket hibásan kétszer számolja. Az Active Support a `Chars` osztály bevezetésével segíti az UTF-8 kódolású sztringek kezelését. A String osztály kiegészül a `chars` metódussal, ami `Chars` típusú objektumként adja vissza a sztringet. A `Chars` osztály metódusai már helyesen kezelik a kétbájtos karaktereket.

```
sztring = ”körte”
print sztring.length # az eredmény hibásan 6
print sztring.chars.length # az eredmény helyesen 5
```

Migrációk

A Ruby on Rails az iteratív fejlesztést támogatja. A forráskódunk változásainak kezelését és nyomon követését széles körben elterjedt verziókezelő rendszerek segítik. Ha a fejlesztés során egy új funkció implementálásánál megváltozik a forráskódunk, akkor minden változást követhetünk a verziókezelő rendszerünkkel. Ha valamilyen okból vissza kell térnünk egy korábbi változathoz, egyszerűen csak lehívjuk a kód egy korábbi revízióját. Az adatbázisunk változásainak a kezelésénél már bonyolultabb a helyzet. Erre a problémára nyújt megoldást a Ruby on Rails a migrációk segítségével.

A migrációk sorszámmal ellátott Ruby forrásfájlok, amelyek segítségével kézben tarthatjuk az adatbázisunk változásait a fejlesztés során. Gyakorlatilag ezek a fájlok tartalmazzák az adatbázissémánk definícióját, de időrendi sorrendben. Amikor egy új modellt hozunk létre, vagy bármilyen más okból változtatnunk kell az adatbázissémánkon, akkor létrehozunk egy új migrációt, ami végrehajtja ezeket a változtatásokat, de a legfontosabb, hogy a migrációk az előző verzióra való visszagörgetéshez szükséges változtatásokat is tartalmazzák. Persze a migrációk nem csak adatbázisséma definiálására használhatók. Gyakorlatilag bármilyen kódrészletet tartalmazhatnak, de elsősorban az adatbázissémánk és az adataink változásainak rögzítésére szolgálnak. Hozhatunk létre migrációt a tábláink kezdeti adatokkal való feltöltésére is.

Migrációk létrehozása

Amikor Ruby on Rails modell generátorával létrehozunk egy modellt, automatikusan létrejön egy `sorszám_create_modellnév.rb` migrációs fájl is az alkalmazásunk `db/migrate` alkönyvtárába, de önálló migrációs fájl is generálható az egyéb változtatásokhoz.

Egy migrációs fájlnak egy osztálydefiníciót kell tartalmaznia. Ennek az osztálynak az `ActiveRecord::Migration` osztály leszármazottjának kell lennie és az `up` és a `down` statikus metódusokat kell implementálnia. Értelemszerűen az `up` metódus felelős a változtatásokért, a `down` metódus a változtatások visszavonásáért. A változtatásokat nem SQL utasításokkal, hanem a `ActiveRecord::Migration` osztály metódusainak hívásával definiálhatjuk. Ezen metódusok között legtöbb SQL DDL utasításnak

megtalálhatjuk a megfelelőjét. Ezek alól említésre méltó kivétel a külső kulcsok kezelése, amit a Rails nem támogat. Ha mégis külső kulcsokat akarunk definiálni, akkor kénytelenek vagyunk natív SQL utasításokat futtatni az `execute` metódus segítségével.

Migrációk futtatása

Migrációk a `db:migrate` Rake taszk segítségével futtathatók. A Rake a Make megfelelője Ruby környezetben. A Ruby on Rails alkalmazásunk adatbázisában mindig lesz egy `schema_info` nevű tábla, amely egyetlen oszlopot és egyetlen sort tartalmaz, az aktuális adatbázisséma verzióját. Amikor kiadjuk a `rake db:migrate` parancsot, a Rails először a `schema_info` táblát keresi meg az adatbázisunkban. Amennyiben nincs ilyen tábla, akkor létrejön 0 verziószámmal. Ha van ilyen, akkor a Rails kiolvassa belőle az aktuális sémánk verziószámát. Ezután azok a `db/migrate` könyvtárban lévő migrációk, amelyek sorszáma nagyobb, mint a séma verziója, sorszám szerinti növekvő sorrendben lefutnak, azaz meghívódnak az osztályok `up` metódusai. Minden egyes migráció lefutása után felülíródik a `schema_info` tábla a megfelelő verziószámmal. A `db:migrate` taszknak megadható a `VERSION` paraméter, így egy adott verziószámig futtathatjuk a migrációinkat. Ha a megadott `VERSION` paraméter nagyobb, mint az aktuális adatbázisséma verziószáma, akkor azok a migrációk futnak le, amelyek sorszáma nagyobb, mint az adatbázisséma verziószáma, de kisebb, vagy egyenlő, mint a `VERSION` paraméterben megadott verziószám. Amennyiben az aktuális verziószámnál kisebb számot adunk, akkor az adatbázisunk visszaállítódik a megadott verzióra, vagyis azoknak a migrációknak futnak le a `down` metódusai, amelyek sorszáma kisebb, vagy egyenlő, mint az aktuális adatbázisséma verziószáma, de nagyobb, mint a megadott `VERSION` paraméter.

ActiveRecord

Az ActiveRecord valósítja meg az objektum/relációs leképezést (O/R mapping) a Ruby on Rails-en belül. Az ActiveRecord nevet egy tervezési mintáról kapta. Az aktív rekord minta egy módszert ír le arra, hogy hogyan érhetjük el az adatbázisunkat az alkalmazásunkból. A minta szerint egy adatbázis táblát vagy nézetet be kell „csomagolnunk” egy osztályba, így egy objektumpéldány a tábla egy sorához kötődik. Egy új objektumpéldány létrehozásakor létrejön egy új sor a táblában, minden betöltött objektum adattagjai az adatbázis tábla megfelelő sorából vett értékeket kapják, és egy objektum módosítása esetén a hozzátartozó adatbázisrekord is megváltozik. A becsomagoló osztály implementálja a megfelelő metódusokat (accessor methods), amelyekkel az adott rekord értékeihez hozzá lehet férni. A csomagoló osztálynak kell kezelni a lekérdezéseket is, SQL SELECT utasítások helyett az osztály megfelelő metódusait használva le kell, hogy tudjunk kérni sorokat a táblából, amelyeket a lekérdező metódusok objektumokból álló kollekciókként adnak vissza. A táblák közötti kapcsolatokat (külső kulcsok) is kezelnie kell a becsomagoló osztálynak. Ezek objektumreferencia típusú attribútumként jelennek meg.

Konfiguráció

A Ruby on Rails „konfiguráció helyett konvenció” filozófiája az ActiveRecord használata közben is hamar megmutatkozik. Gyakorlatilag onnantól kezdve, hogy létrehozunk egy leszármazott osztályt `ActiveRecord::Base` szülőosztállyal, már működő objektum/relációs réteg áll a rendelkezésünkre. Ha betartjuk a Ruby on Rails elnevezési konvencióit, akkor csak minimális konfigurációra lesz szükség. Mik ezek az elnevezési konvenciók? A táblánk nevének angol többes számú nevet kell adnunk, szóköz helyett _ jellel elválasztva a szavakat. A táblához tartozó csomagoló osztályunknak pedig ugyanazt a nevet kell adnunk, csak angol egyes számban, és több szó esetén egybeírva, minden új szót nagybetűvel kezdve. Például egy megrendelés sorait tartalmazó tábla esetén a tábla neve order_items, az osztály pedig a következőképpen néz ki:

```
class OrderItem < ActiveRecord::Base
end
```

Ennyi kód már elég ahhoz, hogy az osztályunk kapcsolódjon az `order_items` táblához. Ezt a kapcsolatot az ActiveRecord pusztán az osztályunk nevéből következtette ki. Az Active Supportról szóló fejezetben már említettem a `String` osztály angol egyes- és többes számot kezelő kiegészítéseiről (`pluralize, singularize`). Az ActiveRecord ezeket a metódusokat használja ahhoz, hogy egy csomagoló osztályhoz megtalálja a megfelelő adatbázistáblát. Persze ha szeretnénk, akkor explicit módon is megadhatjuk a csomagoló osztályunkban az adatbázis tábla nevét:

```
class OrderItem < ActiveRecord::Base
  set_table_name ”legacy_order_item_table”
end
```

Attribútumok

Az ActiveRecord osztályok adatbázis táblákat képviselnek. Az objektumpéldányokhoz a tábla egyes sorai tartoznak. Az objektumok attribútumai felelnek meg a tábla oszlopainak. A fenti osztálydefinícióról azt írtam, hogy már működő objektum/relációs réteget képez. De hol vannak az osztályunk attribútumai? – kérdezhetnénk. Egy ActiveRecord csomagoló osztály esetében nem szükséges az attribútumainkat definiálni. A Ruby on Rails megteszi ezt helyettünk futásidőben. Az ActiveRecord az adatbázissémára támaszkodva dinamikusan konfigurálja be a táblákat becsomagoló osztályokat. A Ruby on Rails a következőképpen felelteti meg az adatbázistáblánk oszlopainak típusait a csomagoló osztályunk attribútumainak Ruby reprezentációival:

SQL típus	Ruby osztály
int, integer	Fixnum
decimal, numeric	BigDecimal
interval, date	Date
clob, blob, text	String
float, double	Float
char, varchar, string	String
datetime, time	Time
boolean	lásd lentebb

Az adatbázisunkból a `find` metódus segítségével kérdezhetünk le sorokat. Például ha van egy `people` táblánk és a hozzá tartozó `Person` osztályunk, akkor a `Person.find(1)` a `people` táblából kiválasztja azt a sort, ahol az elsődleges kulcs (alapértelmezésben `person_id` oszlop) értéke 1, majd ezt a sort lekérdezve létrehozza belőle azt a `Person` típusú objektumot, amelynek minden attribútuma a megfelelő sor megfelelő oszlopainak értékét veszi fel. Az ActiveRecord ezekhez az attribútumokhoz automatikusan létrehozza a megfelelő író és olvasó metódusokat (Java környezetből getter és setter néven lehet ismerős). Az olvasó metódus neve a táblaoszlop neve, az író metódus neve pedig a táblaoszlop neve egy `=` jellel kiegészítve a végén. A logikai típusú attribútumok esetén kicsit bonyolultabb a helyzet. Különböző adatbázis-kezelő rendszerek különféleképpen reprezentálják a logikai típust, ha egyáltalán ismernek ilyet. A probléma ott gyökerezik, hogy a Ruby nyelv logikai hamis értéknek csak a `nil` és a konstans `false` értéket fogadja el. Minden más, tehát a 0 numerikus érték is logikai igaz értéknek minősül. Így, ha az adatbázisunk `0` és `1`, vagy `"t"` és `"f"` értékként tárolja a logikai típusú attribútumunkat, az attribútum lekérdező metódusát feltételben használva mindig logikai igaz értéket kapunk eredményül. Példa:

```
user = Users.find_by_name("Joe")
if user.superuser
  grant_privileges
end
```

A fenti kód esetében a `grant_privileges` metódus csak akkor nem hívódik meg, ha a `superuser` attribútum a Joe nevű felhasználó esetén `nil` vagy konstans `false` értékű. Minden más esetben az `user.superuser` hívás eredményét a feltételben igaznak fogja értékelni a Ruby interpreter. A megoldás a fenti problémára a következő:

```
user = Users.find_by_name("Joe")
if user.superuser?
  grant_privileges
end
```

Ahhoz, hogy egy feltételben használjuk egy logikai típusú attribútum értékét, egy kérdőjelet kell írnunk az attribútumnév után. A `superuser?` metódus logikai hamis értéket ad vissza, ha a `superuser` oszlopban `0` szám, `"0"`, `"f"`, `"false"` vagy `""` (üres) sztring vagy `nil` áll. Minden más esetben logikai igaz lesz az eredmény. Ez a metódus felüldefiniálható az osztályban, ha egyedi működést szeretnénk megvalósítani. Ha például a logikai értékünket a magyar nyelvnek megfelelő `"i"` és `"n"` karakterekkel tároljuk az adatbázisban, akkor a metódus a következőképpen definiálhatjuk:

```
class User < ActiveRecord::Base
  def superuser?
    self.superuser == "i"
  end
  #...
end
```

Elsődleges kulcsok

Az ActiveRecordban konvenció szerint az alapértelmezett elsődleges kulcs az `id` nevű oszlop. Ezt alapesetben a migrációnk a tábla definiálásánál automatikusan létrehozza. Azonban könnyen kerülhetünk olyan helyzetbe, hogy egy másik oszlopot szeretnénk elsődleges kulcsnak választani. Például egy könyveket nyilvántartó tábla esetén a könyvek ISBN számát tároló oszlopot választhatnánk elsődleges kulcsnak, hiszen az ISBN szám minden könyvet egyértelműen azonosít. Az osztálydefiníciónál a következőképpen adhatunk meg egyedi elsődleges kulcsot:

```
class Book < ActiveRecord::Base
  self.primary_key = "isbn"
end
```

Általában az ActiveRecord gondoskodik az elsődleges kulcsok kezeléséről. Ha létrehozunk egy új objektumot, akkor annak mentésekor automatikusan rendelődik hozzá az elsődleges kulcs oszlophoz a soron következő számérték. Amennyiben viszont egyedileg definiáltuk az

elsődleges kulcs oszlopot, mint a fenti példában, onnantól kezdve nekünk kell gondoskodnunk arról, hogy az elsődleges kulcs attribútum értéket kapjon mentés előtt, az ActiveRecord ezt nem végzi el helyettünk.
A Ruby on Rails alapesetben nem támogatja az összetett kulcsok kezelését. Ha mégis több oszlopból álló elsődleges kulcsot szeretnénk használni, akkor megtalálhatjuk a megfelelő Rails pluginokat az interneten, amelyek ezt lehetővé teszik. Azonban ha az alkalmazásunkhoz új adatbázist tervezünk, akkor feltétlenül kerüljük el az összetett kulcsok használatát, mert valószínűleg feleslegesen nehezítjük meg a saját dolgunkat azzal, hogy számos ActiveRecord szolgáltatást nem használhat ki a programunk.

Csatlakozás az adatbázishoz

A Ruby on Rails általánosítja az adatbázissal kapcsolatos műveleteket. Az alkalmazásunknak sosem kell foglalkoznia azzal, hogy éppen milyen típusú adatbázis-kezelő rendszerhez kapcsolódik. Az adatbázis-műveleteket minden esetben ugyanazokkal a metódusokkal végezzük, a különböző adatbázis-szerverekkel való kommunikációt az adatbázis-adapterek végzik. A kapcsolódást az `ActiveRecord::Base.establish_connection` metódussal végezhetjük. Például:

```
ActiveRecord::Base.establish_connection(
  :adapter  => ”mysql”,
  :host     => ”localhost”,
  :database => ”railsdb”,
  :username => ”railsuser”,
  :password => ”railspasswd”
)
```

A legtöbb esetben viszont nem explicit módon csatlakozunk az adatbázishoz, hanem a `config/database.yml` fájlban adjuk meg az adatbázis-kapcsolatunk leírását. Sokszor még ennyi konfigurációt sem kell megírnunk, hiszen alapesetben, ha létrehozunk egy üres Ruby on Rails alkalmazás-csontvázat, akkor az alkalmazásunkhoz automatikusan hozzárendelődik az *`alkalmazásnév`*`_development`, az *`alkalmazásnév`*`_test` és az *`alkalmazásnév`* adatbázisok sorrendben a fejlesztő- a teszt- és az éles rendszerhez.

A Ruby on Rails jelenleg a következő adatbázisszerverekkel tud együttműködni: DB2, Firebird, Frontbase, MySQL, Openbase, Oracle, Postgres, SQLite, SQL Server, Sybase. Ezek többségéhez telepíteni kell a megfelelő Rails plugint.

CRUD – Create, Read, Update, Delete

Az ActiveRecord egyszerűvé teszi a 4 alapvető adatbázis-művelet, a létrehozás, olvasás, módosítás és törlés használatát. A következő részben az alábbi ActiveRecord modellosztállyal fogok példákat mutatni:

```
class User < ActiveRecord::Base
end
```

Az objektum/relációs paradigma szerint az adatbázis tábláknak osztályok, a tábla sorainak az osztály objektumpéldányai feleltethetők meg. Így egy új sor hozzáadása gyakorlatilag egy új objektum létrehozását jelenti. Ezt úgy tehetjük meg, hogy a `new` metódus meghívásával létrehozunk egy új példányt, majd feltöltjük az attribútumait adatokkal, végül a `save` metódus segítségével a változásokat elmentjük az adatbázisba. Ha ez utóbbi lépést kihagynánk, az objektumunk csak a memóriában létezne, ill. az elsődleges kulcs oszlop is csak a `save` metódus meghívása után kap értéket. A `new` metódust rögtön felparaméterezhetjük az attribútumok értékeivel, így rögtön egy „feltöltött” objektum jön létre. Ezt a procedúrát tovább egyszerűsíti a `create` metódus, amit az attribútumok értékével felparaméterezve kell meghívni, viszont a hívással együtt azonnal létrejön az adatbázisban a megfelelő táblasor, nincs szükség a `save` metódus meghívására.
Az adatbázisból való olvasáshoz először is meg kell határoznunk, hogy a tábla mely sorai érdekelnek minket. Ehhez adnunk kell valami kritériumot az ActiveRecordnak, ami alapján az az adatbázisból a megfelelő sorokat lekérdezve objektumok egy halmazát adja vissza. A legkézenfekvőbb módja egy sor lekérdezésének, ha megadjuk az elsődleges kulcsértékét. Minden modellosztály örökli a `find` metódust az `ActiveRecord::Base` szülőosztálytól. Ha a `find` metódusnak az elsődleges kulcsértékeket adjuk paraméterül, akkor a megfelelő objektumokat tartalmazó tömbbel tér vissza. Ha nem talál a táblában sorokat a megadott kulcsértékekkel, akkor kiváltódik a `RecordNotFound` kivétel. Ha több kulcsértéket adtunk meg, és azok közül bármelyiket nem találja meg a táblában, a kivétel akkor is kiváltódik.

Lehetőség van összetettebb keresési feltételek megadására is. A `:condition` szimbólummal ellátott paraméternek átadhatjuk (név szerinti paraméterátadás) egy SQL utasítás WHERE ágát, így bonyolultabb lekérdezéseket is megvalósíthatunk. A `find` metódus megfelelően használva védelmet nyújt az SQL injection támadások ellen is. Egy request paraméterből (POST vagy GET) származó változót közvetlenül a WHERE feltételbe beágyazva lehetőséget adunk a weboldalunk esetleges támadóinak, hogy rosszindulatú kódrészletet juttassanak az adatbázis-lekérdezésünkbe.

```
User.find(:conditions => ”name = ’#{params[:name]}’
                          and superuser=’t’”)
```

A fenti példában a `params` tömb a request paramétereit tartalmazza, vagyis a `params` tömb tartalma a megjelenített oldal URL-jéből is származhat. Ezt az értéket ellenőrizetlenül beilleszjük az SQL lekérdezésünk WHERE feltételébe. Képzeljük csak el, milyen hatása lenne, ha a `params[:name]` értéke a következő lenne:

```
’; DELETE FROM users; --
```

Az ActiveRecord a fenti veszélyre a következő megoldást nyújtja:

```
User.find(:conditions => [”name = ? and
                         superuser = ’t’”, params[:name]])
```

A `:conditions` paraméternek egy tömböt átadva a tömb első elemébe (ami maga a WHERE feltétel) sorrendben behelyettesítődnek a tömb további elemei a kérdőjelek helyére. A lényeg ott van, hogy a behelyettesítés előtt a tömb további elemeit az ActiveRecord megtisztítja a veszélyes elemektől, például `’` karakterek veszélytelen `\’` karaktersorozatokra cserélődnek. Ha a `:conditions` paraméternek megadott feltételeknek egy sor sem felel meg a táblában, akkor a `find` metódus `nil` értékkel tér vissza. Ez a viselkedés eltérő attól a fentebb említett szituációtól, amikor elsődleges kulcsértékekkel paraméterezzük a `find` metódust, ugyanis abban az esetben, ha nem létezik megfelelő sor, akkor a `RecordNotFound` kivétel váltódik ki. A `find` metódus említésre méltó paraméterei még

az `:order`, `:limit`, `:offset` és `:lock`. Az `:order` paraméterrel rendezési sorrendet határozhatunk meg. A `:limit` és `:offset` paraméterekkel megadhatjuk, hogy a lekérdezés eredményéből hányadik sortól kezdve hány darab sorból előállt objektumot kérjük. A `:lock` paraméter lehetőséget ad a tábla érintett sorainak, vagy tábla egészének explicit zárolására. Az ActiveRecord egyik kényelmi szolgáltatása a dinamikus keresőmetódusok. A modellosztályainkon használhatjuk a `find_by_`*`oszlopnév`* metódust, például `User.find_by_lastname(”Nagy”)`. Ez gyakorlatilag ekvivalens a `User.find(:conditions => [”lastname = ?”, ”Nagy”])` hívással, csak kényelmi funkció és az olvashatóságot segíti.
A meglévő rekordok módosítása már sokkal magától értetődőbb dolog. A legkézenfekvőbb módszer a következő:

```
user = User.find(1)
user.name = ”Kiss Lajos”
user.email = ”klajos@domain.com”
user.save
```

A megfelelő rekordot betöltjük, módosítjuk az objektum attribútumát, majd a `save` metódussal a változást elmentjük. Emellett az ActiveRecord lehetőséget nyújt arra, hogy a módosítás és a mentés lépését összevonjuk, sőt, több módosítást is elvégezzünk egyszerre.

```
user = User.find(1)
user.update_attributes(:name => ”Kiss Lajos”,
                       :email => ”klajos@domain.com”)
```

Vagy az összes lépést összevonva:

```
User.update(1, :name => ”Kiss Lajos”,
               :email => ”klajos@domain.com”)
```

Az alábbi formát használva az egész táblán végrehajthatunk módosításokat:

```
User.update_all(”salary=salary*1.2”,
                ”position like ’%manager%’)
```

Rekordok törlésére 2 módszer is a rendelkezésünkre áll. Az első az adatbázisszinten operáló `delete` és `delete_all` metódus:

```
User.delete(1) #az 1-es idvel rendelkező
               #felhasználó törlése
User.delete_all(”gender = ’male’”) #az összes férfi
                                   #felhasználó törlése
```

A másik módszer a `destroy` és a `destroy_all` metódusok segítségével történik. Ezeket a metódusokat is a fenti példának megfelelően lehet használni. A különbség a kettő között az, hogy a `delete` és a `delete_all` metódusok az adatbázisból törlik ugyan a megfelelő sorokat, a már beolvasott, és azóta az adatbázisból törölt objektumokról nem látszik, hogy már nem léteznek az adatbázisban. A `destroy` és `destroy_all` metódusok törlik a megfelelő sorokat az adatbázisból, majd zárolják azokat az ActiveRecord objektumokat, amelyek az érintett sorokhoz tartoznak. A `destroy` és a `destroy_all` metódusok meghívásánál lefutnak a modellosztályban definiált megfelelő validáló és callback metódusok is. A validációt és a callback metódusokat részletesebben később tárgyalom. Az általánosan elfogadott vélemény az, hogy érdemesebb mindig a `destroy` és a `destroy_all` metódusokat használni, hiszen így könnyebb megőrizni a modelljeink integritását a modellosztályokban definiált üzleti megszorítások szerint.

Mágikus oszlopnevek

Van néhány lefoglalt oszlopnév, amelyekhez az ActiveRecord automatikusan rendel értéket. Ha az alábbiak egyikével nevezzük el egy oszlopunkat, akkor annak az oszlopnak az értékét az ActiveRecord automatikusan módosítja a megfelelő szituációban. A `created_at`, `created_on`, `updated_at`, `updated_on` nevű oszlopok mindig azt az időt mutatják,

amikor az adott rekord létre lett hozva vagy módosítva lett. Az `_at` végződésű oszlopok pontos időt tárolnak, az `_on` végződésűek csak dátumokat. Az `id` nevű oszlop alapértelmezésben az elsődleges kulcsot jelöli, az ActiveRecord automatikusan rendeli hozzá a soron következő értéket az `id` mezőhöz, amikor egy újonnan létrehozott ActiveRecord objektum `save` metódusát meghívjuk. A *`táblanév`*`_id` nevű oszlopok mindig külső kulcs referenciát jelölnek. Léteznek ezeken kívül további különleges oszlopnevek is, ám azok tárgyalása meghaladja a fejezet kereteit.

Táblák közötti kapcsolatok

Az adatbázissémában a táblák közötti kapcsolatot tipikusan külső kulcsok valósítják meg. Az egyik táblában egy külső kulcs hivatkozik egy másik tábla elsődleges kulcsára. Ruby on Rails alkalmazások írásánál viszont ritkán szoktunk közvetlenül az adatbázis tábláival dolgozni, helyettük modellosztályok példányaival építjük fel az alkalmazásunk üzleti logikáját. Az ActiveRecord lehetőséget nyújt arra, hogy a modellosztályaink objektumreferenciákkal kapcsolódjanak egymáshoz, ezáltal tükrözve az adatbázissémánk külső kulcson alapuló kapcsolatait.

Mint azt egy korábbi fejezetben már tárgyaltam, Ruby on Rails alkalmazások írásakor az adatbázissémánk definiálásához migrációkat használunk. Nézzük, hogyan nézne ki egy leegyszerűsített blogmotor adatbázisséma-definiáló kezdeti migrációja:

```
class InitSchema < ActiveRecord::Migration
  def self.up
    create_table :users do |t|
      t.column :name, :string
      #...
    end
    create_table :entries do |t|
      t.column :user_id, :integer
      t.column :title, :string
      #...
    end
    create_table :comments do |t|
```

```
      t.column :user_id, :integer
      t.column :entry_id, :integer
      t.column :body, :string
      #...
    end
  end
  def self.down
    drop_table :users
    drop_table :entries
    drop_table :comments
  end
```

A fenti migráció osztályban a félkövér betűvel kiemelt sorok jelölik a külső kulcsokat. Az elnevezési konvenciók szerint a táblákat angol többes számú főnevekkel neveztük el, míg a külső kulcsokat azok egyes számú megfelelőivel és `_id` végződéssel. Meg kell, hogy jegyezzem, hogy a fenti migráció egyáltalán nem definiál külső kulcs megszorításokat adatbázis szinten. Hozhatunk létre explicit módon külső kulcs megszorításokat, de a Ruby on Rails külső kulcs mechanizmusa ezt nem követeli meg.

Az adatbázisunk tábláiban az oszlopok megfelelő elnevezése viszont nem elég ahhoz, hogy egyértelműen leírhassuk a modellosztályaink kapcsolatait. A kapcsolatokat az osztálydefiníciókban is jelölni kell. A fenti példának megfelelő modellosztály-definíciók a következőképpen néznek ki:

```
class User < ActiveRecord::Base
  has_many :entries
  has_many :comments
  has_many :entries_commented, :through => :comments,
           :source => :entry
  #...
end
class Entry < ActiveRecord::Base
  belongs_to :user
  has_many :comments, :order => ”created_on DESC”
```

```
  #...
end
class Comment < ActiveRecord::Base
  belongs_to :user
  belongs_to :entry
  #...
end
```

Csupán ennyit kell tennünk ahhoz, hogy definiáljuk a kapcsolatokat a modellosztályainkban. Az elnevezési konvenciókat betartva az ActiveRecord könnyen feltérképezi a kapcsolataink megvalósítását az adatbázissémánk szintjén. A fenti példából láthatóan 1-n relációkat a has_many és a belongs_to metódusokkal definiálhatunk. 1-1 relációkat a has_one és belongs_to, közvetlen n-m relációkat mindkét oldalon a has_and_belongs_to_many metódussal. Közvetlen n-m reláció alatt azt értem, amikor a kapcsolatot megvalósító harmadik tábla a 2 külső kulcson kívül nem tartalmaz más oszlopot. Az ilyen táblákat tálbanév1_táblanév2 mintára kell elnevezni, az ActiveRecord a has_and_belongs_to_many megfelelő használatával automatikusan felismeri, hogy mely tábla mely másik két táblát kapcsol n-m relációba. A fenti példában a `User` osztály `entries_commented` kapcsolata gyakorlatilag n-m reláció, de mivel a köztes tábla (`entries`) önmagában modellosztályt reprezentáló tábla is, így a `has_many` metódust használjuk `:through` és `:source` paraméterekkel, amelyek leírják, hogy melyik táblán keresztül melyik modellosztályhoz kapcsolódik a `User` osztályunk.

Amint a fent leírt módon a modellosztályainkban is definiáltuk a kapcsolatokat, már használhatunk az alábbihoz hasonló kifejezéseket:

```
for entry in Entry.find(:all)
  puts ”Bejegyzés: #{entry.title}”
  puts ”Szerző: #{entry.user.name}”
  puts ”Utolsó megjegyzés ideje:
              #{entry.comments.first.created_on}”
end
```

A kapcsolatok definiálása után számos új metódusa jelent meg a modellosztályainak. A kapcsolatok neveivel ellátott metódusok `has_one` vagy `belongs_to` kapcsolat esetén a kapcsolódó objektumot, `has_many` vagy `has_and_belongs_to_many` kapcsolat esetén a kapcsolódó objektumok egy tömbjét adják vissza. A fenti példában az utolsó megjegyzés idejét azért volt ilyen egyszerű lekérdezni, mert az `Entry` osztályban a `comments` kapcsolat definiálásakor az `:order` paraméternek a `"created_on DESC"` értéket adtuk, ezáltal az `Entry.comments` a bejegyzéshez tartozó megjegyzések egy létrehozás ideje szerint csökkenő sorrendben rendezett tömbjét adva vissza. Ezeken kívül megjelennek még a `create`, `delete`, `delete_all`, `destroy` és `destroy_all` metódusok is minden kapcsolathoz. Érdemes még megjegyezni, hogy az ActiveRecord alapértelmezésben egy vagy több modell beolvasásakor a háttérben futó SQL lekérdezésben nem kapcsolja össze a kapcsolódó táblákat. A kapcsolódó modelleket csak akkor olvassa be az adatbázisból, amikor azokra hivatkozunk. Így a fenti példában először lefut egy lekérdezés, ami visszaadja az összes bejegyzést. Majd a ciklusmagban a mag minden futásakor lefut egy lekérdezés, ami beolvassa a bejegyzéshez tartozó felhasználót. Ezután szintén minden ciklusmag-futáskor egy újabb lekérdezés beolvassa a bejegyzéshez tartozó megjegyzéseket. Ez nem túl hatékony, 100 bejegyzés esetén összesen 201 lekérdezés. A következőképpen vehetjük rá a Ruby on Railst, hogy 1 lekérdezéssel beolvasson minden adatot:

```
for entry in Entry.find(:all,
                        :include => [:user, :comments])
  #...
end
```

Így jeleztük az ActiveRecordnak, hogy a bejegyzésekkel együtt azoknak a szerzőire és megjegyzéseire is szükségünk lesz. Azonban ez a megoldás nem garantált, hogy minden esetben gyorsítja a programunkat. Ha a szülőtáblánk rengeteg sort tartalmaz, akkor a lekérdezés eredménye rengetet memóriaterületet foglal. Az ActiveRecord fenti táblákat az SQL lekérdezésben összekapcsolja, majd az eredményt egytől-egyig ActiveRecord modellobjektumokká alakítja, így elképzelhető, hogy nem használt adatokat beolvasva

feleslegesen terheljük a rendszert. A másik probléma a fenti megoldással, hogy csak és kizárólag a LEFT OUTER JOIN-t támogató adatbázisszerverekkel működik.

Validáció

Az ActiveRecord validálhatja egy modellobjektum tartalmát. A validáció automatikusan megtörténik az objektum elmentésekor, de explicit módon is validálható a modellobjektum aktuális állapota. Ha mentéskor a validáción elbukik az objektum, akkor a tartalma nem íródik be az adatbázisba, a memóriában marad nem valid formában. Ez lehetőséget ad például arra, hogy visszaadjuk az objektumunkat a webes űrlapunknak, a felhasználót arra kérve, hogy javítsa a hibásan kitöltött mezőket; azokhoz az attribútumokhoz kapcsolt űrlapelemeket, amelyeknél a validáció hibás értéket jelzett. Az ActiveRecord megkülönbözteti azokat az objektumokat, amelyek már egy létező táblasorhoz kapcsolódnak az adatbázisban, és azokat, amelyek még nem. Az utóbbiakat új rekordoknak nevezi, ezen objektumok `new_record?` metódusa `true` értéket ad vissza. Amikor a `save` metódust meghívjuk, az új rekordok esetén egy INSERT utasítás, létező rekordok esetén egy UPDATE utasítás fut le a háttérben. A megkülönböztetésnek azért van jelentősége, mert megadhatók olyan validációs szabályok, amelyeknek csak az új, illetve csak a már létező rekordok elmentésekor ellenőriződnek. Alacsony szinten validációs szabályok megadása gyakorlatilag a `validate`, `validate_on_create` és a `validate_on_update` metódusok implementálásával történik. A `validate` metódus minden `save` híváskor lefut, a másik kettő lefutása attól függ, hogy új, vagy már létező rekordról van szó. A validációt bármikor lefuttathatjuk a modellobjektumunk `valid?` metódusának meghívásával.

```
class User < ActiveRecord::Base
  def validate
    if name.blank?
      errors.add(:name, ”is missing or invalid”)
    end
  end
  def validate_on_create
    if User.find_by_name(name)
      errors.add(:name, ”is already being used”)
```

```
    end
  end
end
```

A fenti példában a validációs metódusok minden mentéskor ellenőrzik azt, hogy a név attribútum nem üres sztring, és létrehozáskor ellenőrzi, hogy van-e már létrehozva felhasználó ilyen névvel. Amikor egy validáló metódus valamelyik attribútum ellenőrzésekor hibát talál, akkor hozzáad egy hibaüzenetet a hibák listájához az `errors.add` hívás segítségével. Ekkor megadjuk, hogy melyik attribútum bukott meg az ellenőrzésen, ill. szövegesen leírjuk a hibát. Ha viszont olyan hibát találtunk, amely nem kötődik egy konkrét attribútumhoz, hanem vagy több attribútumra vonatkozik egyszerre vagy az egész modellobjektumot érinti a hiba, akkor az `errors.add_to_base` metódussal leírhatjuk a hibát anélkül, hogy azt az egyik attribútumhoz kötnénk. Lekérdezhetjük egy attribútumot érintő hibák listáját az `errors.on(:`*`attribútumnév`*`)` hívással, illetve törölhetjük a teljes hibalistát az `errors.clear` metódussal.

A legtöbb esetben hasonló szabályok szerint ellenőrizzük az objektumunk attribútumait: nem lehet üres sztring, számértéknek kell lennie, két értékhatár között veheti fel az értékeit, és hasonlók. Az ActiveRecord számos metódusban összegyűjtötte a tipikus validációs eseteket, így a legtöbb esetben nem kell implementálnunk a `validate` metódust, és abban sorra ellenőrizni az attribútumokat, elegendő ezeket a metódusokat használnunk. A legtöbb ilyen metódusnak van egy `:on` és egy `:message` paramétere. Az `:on` paraméterben adhatjuk meg, hogy csak létrehozáskor, csak módosításkor, vagy mindkét esetben ellenőrződjön az attribútum. A `:message` paraméterben a hibaüzenetet adhatjuk meg. A `validates_acceptance_of` metódus ellenőrzi, hogy egy attribútum értéke 1 legyen. Tipikusan akkor használatos, amikor egy űrlapon egy jelölőnégyzet (checkbox) bejelöltségét ellenőrizzük, például regisztrációs űrlapok esetén az „elfogadom a feltételeket” jelölőnégyzetek. A `validates_confirmation_of` metódus ellenőrzi, hogy egy űrlapmező és annak az ellenőrző mezőjének az értéke megegyezik. Tipikusan akkor használatos, amikor arra kérjük a felhasználót, hogy kétszer adja meg a jelszavát, a félregépeléseket elkerülendő. A `validates_exclusion_of` és `validates_inclusion_of` metódusok ellenőrzik, hogy az attribútum értékét a megadott tömb nem tartalmazza vagy épp a megadott tömb elemeinek valamelyikével egyenlő. A

`validates_format_of` metódussal reguláris kifejezést használva ellenőrizhetjük egy attribútum formátumát. A `validates_length_of` metódus az attribútum hosszát vizsgálja. A `validates_numericality_of` metódus azt ellenőrzi, hogy az attribútum értéke számtípusú-e. A `validates_presence_of` metódus az attribútum ürességét vizsgálja. A `validates_uniqueness_of` metódus ellenőrzi, hogy nem létezik-e már ugyanolyan attribútum értékkel rekord.

Callback metódusok

Az ActiveRecord irányítja a modellobjektumok teljes életciklusát – létrehozza őket, figyeli a módosításokat, validálja és elmenti azokat, adott esetben törli őket. Callback metódusok használatával az ActiveRecord lehetőséget ad a programunknak arra, hogy az aktívan végigkövesse ezt a folyamatot. Callback metódusok implementálásával írhatunk olyan kódot, ami lefut, amikor valami szignifikáns esemény bekövetkezik az objektumunk életében.

Az ActiveRecord 20 callback metódust definiál. Ebből 18 before/after pár, amelyek valamilyen esemény bekövetkezése előtt/után hívódnak meg. Például egy ActiveRecord objektum `destroy` metódusának meghívásakor először meghívódik a `before_destroy` callback metódus, majd megtörténik az objektum törlése, ezután lefut az `after_destroy` callback metódus. Ezalól a 2 kivétel az `after_find` és az `after_initialize`, ezeknek nincs megfelelő `before_` párjuk.

A 18 before/after callback metóduspár a következő:

modell.save()		*modell*.destroy()
(új rekord)	(létező rekord)	
before_validation	before_validation	
before_validation_on_create	before_validation_on_update	
after_validation	after_validation	
after_validation_on_create	after_validation_on_update	
before_save	before_save	
before_create	before_update	before_destroy
beszúrás	**módosítás**	**törlés**
after_create	after_update	after_destroy
after_save	after_save	

A fenti táblázatban látszik, hogy a 18 callback metódus a három alapvető utasítást, a beszúrást, módosítást és törlést öleli körül. A fenti táblázat időrendi a hívások időrendi sorrendjében listázza a metódusokat. A fenti táblázatból kimaradt `after_find` callback metódus a `find` metódus meghívása után fut le, az `after_initialize` közvetlenül egy ActiveRecord objektum létrehozása után hívódik meg.

Callback metódust alapesetben kétféleképpen implementálhatunk. Vagy közvetlenül definiálhatjuk a callback metódust példánymetódusként a modell osztálydefiníciójában, vagy callback handlereket adunk meg. Callback handlert úgy adhatunk meg, hogy a modellünk osztálydefiníciójában meghívjuk a megfelelő callback metódust egy metódus nevével szimbólumként vagy egy blokkal paraméterezve. Így minden alkalommal, amikor a callback metódus lefut, akkor meghívódik a paraméterként megadott metódus, vagy lefut a paraméterként megadott blokk. Egy eseményhez több metódust és blokkot is rendelhetünk, ez esetben a megadás sorrendjében fognak egymás után lefutni. Példa:

```
class User < ActiveRecord::Base
  before_validation :metódus_neve #metódus hozzárendelése

  after_create do |user| #blokk hozzárendelése
    #...
  end

  def before_save #callback metódus implementálása
     #...
  end
end
```

Ahelyett, hogy a callback handlereket a modellosztályunkban definiáljuk, megtehetjük, hogy egy külön osztályt hozunk létre a callback handlerek számára. Egy ilyen callback osztály egy egyszerű osztály, ami implementálja a callback metódusokat. Ezek nem modellosztályok, így nem az `ActiveRecord::Base` leszármazottai. Ha a callback metódusainkat ilyen callback osztályokban gyűjtjük össze, akkor ezeket több modellosztályhoz is egyszerűen hozzárendelhetjük. A hozzárendelés úgy működik, hogy a modellosztályunk definíciójában a

callback osztályunkat példányosítjuk, majd a létrejött callback objektumot adjuk át paraméternek a callback metódusoknak.

ActionController

Az Action Pack áll a Ruby on Rails alkalmazások középpontjában. Két Ruby modulból áll, az egyik az ActionController, a másik az ActionView. Ez a kettő segít abban, hogy feldolgozzuk a beérkező HTTP kéréseket (request), és előállítsuk a megfelelő válaszokat (response). Ebben a fejezetben az ActionControllerről lesz szól.
Amikor az ActiveRecordot vizsgáltuk, önálló könyvtárként kezeltük azt. Az ActiveRecordot bármilyen nem webes Ruby alkalmazás fejlesztésekor használhatjuk. Az Action Pack ilyen szempontból más. Éppen használható közvetlenül mint keretrendszer, de nem valószínű, hogy bárki is önmagában szeretné az Action Pack szolgáltatásait igénybe venni. A Ruby on Rails igazi ereje ott mutatkozik meg, ahogy ezt a három komponenst, az ActiveRecordot, az ActionControllert és az ActionView-t egy könnyen használható koherens egésszé köti össze. Éppen ezért míg az ActiveRecordot még önmagában, függetlenül tárgyaltam, az ActionControllert már mint a Ruby on Rails keretrendszer többi moduljával szoros kapcsolatban álló részeként fogom kezelni.

Mi is az ActionController?

Az ActionController áll a HTTP kérések feldolgozásának a középpontjában. Az ActionController feladata, hogy a kérésekhez hozzárendelje a megfelelő kontrollerosztály megfelelő akciómetódusát. Egy kontrollerosztály az `ActionController::Base` leszármazottja, és minden publikus metódusát akciómetódusnak nevezzük. Minden kérésnél lefut a megfelelő akciómetódus, végrehajtja a szükséges tevékenységeket, majd vagy átirányít egy másik akciómetódusra, vagy átadódik a vezérlés az ActionView modulnak, hogy az akciómetódus tevékenységének eredménye alapján előállítsa a választ a HTTP kérésre, ami legtöbb esetben egy HTML oldal legenerálását jelenti.

Kontrollerosztályok és akciómetódusok

Mint már említettem, a kontrollerosztályok az `ActionController::Base` leszármazottjai. Minden publikus metódusuk akciómetódus, ami azt jelenti, hogy URL van hozzárendelve, alapvetően a felhasználó által elérhető. Az URL tartalmazza, hogy az alkalmazásunk melyik kontrollerosztályának melyik akciómetódusát kell meghívni milyen paraméterezéssel. Az URL hozzárendelése alapbeállításban

domainnév/kontroller/akció/id?paraméterek formátumban történik. Ez a leképezés megváltoztatható a `config/routes.rb` fájl átírásával, de ennek részletezésébe most nem megyek bele. Amikor a kontroller objektum (a kontrollerosztály példánya) feldolgozza a HTTP kérést, akkor először is olyan publikus példánymetódust keres, amelynek a neve megegyezik az URLből származó akció nevével. Ha talál ilyen metódust, akkor az meghívódik. Ha nincs ilyen metódus, akkor keres olyan sablont (template; az ActionView-t tárgyaló fejezetben foglalkozok vele részletesen), ami a kontrollerhez tartozik, és a neve megfelel az akció nevének. Ha talál ilyet, akkor a sablon közvetlenül lerenderelődik. Ha nem talál megfelelő sablont, akkor az „Unknown Action” hibaüzenet jelenik meg a felhasználó böngészőjében. Az akciómetódusok feladata, hogy végrehajtsák a kívánt üzleti logikát a kérés paramétereit használva, majd előkészítsék az adatokat a válasz legenerálásához. Az adatokat példányváltozók segítségével juttatja el az ActionView modulnak. Minden példányváltozó, ami az akciómetódus futása során létrejön, a sablonból elérhető, annak értéke a válasz legenerálásakor felhasználható. Tekintsünk egy konkrét kontrollerosztály-példát, ami leegyszerűsített felhasználó-kezelést valósít meg:

```
class UserController < ApplicationController
  def list
    @users = User.find(:all)
  end
  def new
    @user ||= User.new
  end
  def create
    @user = User.new(params[:user])
    if @user.save
      flash[:notice] = ”Új felhasználó létrejött.”
      redirect_to :action => ”list”
    else
      render :action => ”new”
    end
  end
```

```
  def edit
    @user = User.find(params[:id])
  end
  def update
    @user = User.find(params[:id])
    if @user.update_attributes(params[:user])
      flash[:notice] = ”Felhasználó megváltoztatva.”
      redirect_to :action => ”list”
    else
      render :action => ”edit”
    end
  end
  def destroy
    User.find(params[:id]).destroy
    redirect_to :action => ”list”
  end
end
```

A fenti példában az összes szükséges metódus megtalálható, amivel felhasználókat létrehozhatunk, módosíthatunk, törölhetünk, illetve kilistázhatjuk a már létező felhasználókat. Új felhasználó létrehozásakor a `new` metódus létrehoz egy új, üres felhasználó objektumot, és értékül adja azt a `@user` példányváltozónak, amennyiben annak az értéke `null` (a `|=` értékadó operátor esetén az értékadás csak akkor történik meg, ha az operátor bal oldalán álló változó értéke `null`). Ezután a vezérlés átadódik az ActionView modulnak, ami legenerálja az új felhasználó létrehozásához használt űrlapot. Miután az űrlapot elküldjük, a kitöltött értékeket a `create` metódus fogadja, létrehoz egy felhasználó objektumot a kérés paraméterei alapján, majd sikeres mentés esetén beállítja a megfelelő üzenetet a `flash`-be (a `flash` működését később tárgyalom), és átirányít a felhasználókat kilistázó akcióra. Amennyiben a mentés nem sikerült, mert modell elbukott a validáción, a `render` hívással a vezérlés átadódik a `new` akciómetódusnak, és a `new` akciómetódushoz tartozó sablon alapján az ActionView újra legenerálja az űrlapot a megfelelő hibaüzenetekkel. Ennél az esetnél jön elő a jelentősége annak, hogy a `new` metódusunk csak akkor példányosítja a `User` osztályt,

ha a `@user` példányváltozó értéke null. Ugyanis ha validációs hiba miatt a `create` akciómetódus átadja a vezérlést a `new` akciómetódusnak, akkor a `@user` példányváltozó már létezik, a `create` metódus az elküldött űrlap alapján létrehozta, és a `new` metódus így azt nem írja felül egy üres objektummal, az ActionView így egy kitöltött, és a megfelelő hibaüzenetekkel ellátott űrlapot tud generálni. Ha jobban belegondolunk, alapesetben nem is volna szükség példányosításra az üres űrlap létrehozásához, viszont ezzel a megoldással elég egy darab olyan űrlapsablont létrehoznunk, ami a kontrollertől egy objektumot vár, és az objektum alapján tölti ki az űrlapelemeket. Egy ilyen sablon egyszerre használható az összes létrehozás és módosítás művelethez, hiszen ha üres objektumot hozunk létre, mint a `new` metódusban, akkor üresen jelenik meg az űrlap, viszont validációs hiba és a módosítást kezelő akciómetódusok esetében az űrlapunk megfelelően kitöltve jelenik meg. Az `edit` és `update` metódusok is hasonlóan működnek, annyi különbséggel, hogy az `edit` metódusban nem üres objektumot példányosítunk, hanem az adatbázisból töltjük be a megfelelő felhasználót. A `destroy` metódus nem igényel különösebb magyarázatot; kikeresi a megfelelő felhasználót, törli azt, majd átirányít a felhasználók listáját előállító akciómetódusra. A `list` metódus egy példányváltozóba beolvassa az összes felhasználót az adatbázisból, majd a vezérlés átadódik az ActionView modulnak.

Munkamenet-kezelés

A legtöbb webalkalmazás előbb-utóbb munkamenetekkel dolgozik. Egy munkamenet végigkiséri egy felhasználó folyamatos tevékenységét egy weboldalon. Fontos lehet, hogy egy munkamenetben két kérés között valamilyen adatokat tároljunk, azokat az adatokat későbbi kéréseknél felhasználjuk. Mivel webalkalmazások esetén a program futása nem folyamatos, hanem kérdés-válasz rendszerben működik, és a kérések felváltva érkezhetnek több felhasználó böngészőjétől, fontos, hogy azonosítani tudjuk ugyanannak a felhasználónak az egymás utáni kéréseit. Erre a problémára nyújt megoldást a munkamenet-kezelés. A felhasználói munkamenetekhez (session) a program mindig egy egyedi azonosítót rendel (session id, SID), ami a következő kérés egyik paramétereként mindig visszajut a webszerverhez, így a kérések egyértelműen hozzárendelhetők a felhasználók munkameneteihez. A leggyakoribb esetben a munkamenet-azonosítót a böngésző sütik (cookie) formájában tárolja.

A Ruby on Rails gondoskodik a fenti mechanizmus implementációjáról. Az alkalmazásunk írása során a kontrollerosztályainkban a `session` hash-t használva tárolhatunk értékeket a kérések között. Például ha egy regisztrált felhasználó bejelentkezik az oldalunkra, akkor arra a munkamenetre a kijelentkezésig tárolni szeretnénk, hogy a felhasználónk bejelentkezett állapotban van, és így például hozzáférhet csak regisztrált felhasználók számára elérhető tartalmakhoz. Ezt leegyszerűsített formában az alábbi akciómetódussal valósíthatjuk meg:

```
def login
  if @user = User.find(:conditions => [”name=? and
             password=?”, params[:username],
             md5(params[:password])])
    session[:user_id] = @user.id
  end
end
```

Sikeres bejelentkezés esetén a `session[:user_id]` a következő kérésnél is a felhasználó azonosítóját fogja tartalmazni, így ennek értékét felhasználva építhetjük tovább az alkalmazásunkat. A `config/enviroment.rb` konfigurációs fájlban beállíthatjuk a session változók tárolásának módját. A `session` hash értékeit a Ruby on Rails tárolhatja többek közt adatbázisban, a webszerver fájlrendszerében vagy memóriájában.
A korábbi, egyszerűsített felhasználó-kezelést bemutató példában használt `flash` hasonló a `session` hash-hez. A különbség annyi, hogy míg a `session` hash a munkamenet végéig tárolja a beállított értékeket, a `flash`-be beállított értékek csak a közvetlen következő kérésig élnek. A `flash` használatának tipikus esete, amikor egy sikeres vagy sikertelen tevékenység után átirányítjuk a böngészőt egy másik akcióra, de valami üzenetet akarunk kiíratni a felhasználónak a tevékenységről. Az átirányítás során új kérés keletkezik, így az üzenetünket a `flash`-be eltárolva az átirányítás után még kiírathatjuk azt a felhasználónak.

ActionView

Az ActionView modul felelős a felhasználó böngészőjéhez visszaérkező válasz legenerálásáért. Az előző fejezetben kitárgyaltam, hogy a Ruby on Rails hogyan rendeli hozzá a kéréshez a megfelelő kontrollerosztály megfelelő akciómetódusát. Az akciómetódus végrehajtja a megadott tevékenységeket, példányváltozókban előkészíti a megfelelő adatokat a válasz legenerálásához, majd az akciómetódus lefutása után a vezérlés átadódik az ActionView modulnak. Az ActionView modul a megfelelő sablon alapján legenerálja a válasz a kérésre, ami legtöbb esetben HTML, XML vagy JavaScript.

Sablonok

A sablonok olyan forrásfájlok, amik tartalmazzák az információt ahhoz, hogy az ActionView modul legenerálhassa a választ a HTTP kérésre. A sablonfájlokban statikus szöveget és futtatható Ruby kódot keverve tartalmazhatnak. A sablonfájlokban <% és %> illetve <%= és %> jelpárok között bárhol helyezhetünk el Ruby kódot. A <% és a <%= jelek között annyi a különbség, hogy a <%= nyitó jelet használva a beágyazott Ruby kód eredménye visszahelyettesítődik a szövegbe a sablonfájl feldolgozása során. Tipikusan a vezérlési szerkezeteket tartalmazó sorokat <% és %> jelek közé írjuk, míg azokat a kifejezéseket, amelyek értékét meg akarjuk jeleníteni a válaszban <%= és %> jelek közé beágyazva használjuk. A Ruby on Rails alapesetben háromféle sablonformátumot ismer: az rxml-t XML, az rhtml-t HTML, az rjs-t JavaScript válaszok generálásához használjuk.

Helper metódusok

Az ActionView modul számos helper metódussal segíti a sablonfájljaink elkészítését. Ezek a metódusok kimenetcentrikusak, elsősorban formázást és HTML és XML tagek és JavaScript generálását segítik. Léteznek helper metódusok számok és szöveg megformázására, HTML linkek és űrlapok generálására. A helper metódusok az összes sablonfájlból láthatók, így saját helper metódusok létrehozásával saját kódjainkat is újrafelhasználhatóvá tehetjük. Saját helper metódusokat Ruby modulok létrehozásával definiálhatunk. A Ruby modulok gyakorlatilag függvénykönyvtárakat jelentenek. Ezekben a modulokban definiálhatjuk a saját helper metódusainkat. A helper metódusok nem osztály és nem példánymetódusok, mivel a

Ruby modulok nem osztályok. Használatuk megfelel az eljárás-orientált függvénykönyvtáraknak.

Részsablonok

Helper metódusok létrehozása az egyik módja az újrafelhasználásnak az ActionView modulban. Viszont gyakran előfordul olyan eset, amikor sokkal összetettebb tevékenységsorozatot szeretnénk újrafelhasználni. Ekkor létrehozhatunk részsablonokat (partials). A részsablonok nem különböznek a többi sablontól, az erejük csupán abban rejlik, hogy egy sablonban meghívhatunk részsablonokat, akár paraméterezve is. A részsablonok forrása nem különbözik bármilyen más sablon forrásától, az egyetlen különbség, hogy a részsablonokat definiáló sablonfájlok nevei _ jellel kezdődnek. Ez azért fontos, mert, mint az ActionController-t tárgyaló fejezetben írtam, ha Ruby on Rails nem talál a beérkező kérés URL-jéből származó akciónak megfelelő metódust, akkor keres olyan sablonfájlt, aminek a neve megegyezik az akció nevével. Mivel a részsablonok nem önálló egységek, általában nem tartoznak hozzájuk akciómetódusok. Ha részsablon-fájlok nevei nem lennének megkülönböztetve a _ kezdőkarakterrel a többi sablontól, akkor a megfelelően összeállított URL-lel közvetlenül meg lehetne hívni egy részsablont a böngészőből. Van arra példa, hogy az akciómetódus futása után csak egy részsablon alapján generálódik le a válasz. Ez elsősorban akkor fordul elő, amikor AJAX-os alkalmazást írunk. Ilyenkor a HTTP kérést JavaScript kód indítja el a háttérben, és a választ is JavaScript dolgozza fel anélkül, hogy a böngészőben elnavigálnánk az aktuális oldalról. Ilyenkor gyakran előfordul, hogy az AJAX kérés-válasz eredményeként az oldalnak egy része lecserélődik. Ilyen kérések esetén a válasz legenerálásakor az ActionView csak a megfelelő részsablon alapján generálja le a választ. Ilyen esetben az akciómetódus mindig explicit megmondja, hogy melyik részsablon alapján kell legenerálni a választ, a Ruby on Rails akciónév alapján sosem használ renderel le részsablont. Részsablont a `render` metódussal hívhatunk meg egy sablonban. A `render` metódus `:partial` paramétere a részsablon neve, az `:object` paraméterrel egy objektumot, a `:collection` paraméterrel egy kollekciót adhatunk át a részsablonnak. A `:collection` paraméter használatánál a részsablon a kollekció minden egyes elemére meghívódik. A részsablonon belül az átadott objektumra vagy a kollekció aktuális elemére a részsablon nevével (a _ kezdőkarakter nélkül), mint változóval hivatkozhatunk.

Összefoglalás

A dolgozatban ismertetett Ruby on Rails keretrendszer kiválóan alkalmas webalkalmazások gyors és rugalmas fejlesztésére. Használata meglepően hamar megtanulható, saját tapasztalataim szerint 1-2 hét tanulás után hozzá lehet kezdeni Ruby on Rails alkalmazások fejlesztéséhez. Viszonylag kezdő tudással is bátran bele lehet vágni nagyobb projektek elkészítésébe, mert a Ruby on Rails alapvető működésénél fogva rákényszeríti a programozót, hogy jól átgondolt, átlátható kódot írjon.

Egy konvenciók szerint fejlesztett alkalmazás rendívül kevés kóddal meglepően gazdag funkcionalitással bírhat. Természetesen a való életben a keretrendszer által nyújtott beépített funkcionalitás nem mindig felel meg az üzleti elvárásoknak, ilyenkor nyilvánvalóan saját kezűleg kell implementálni az egyedi működést, de a Ruby on Rails mindig ad lehetőséget a konvenciók megkerülésére, egyedi megoldások alkalmazására.

A fejlesztés rendkívül gyors és rugalmas. Sok nagyvállalat, ami Java EE vagy ASP .Net környezetben fejleszt webalkalmazásokat, prototípuskészítésre Ruby on Railst használ, mert a megrendelő esetleg változó elvárásait nagyon gyorsan lehet követni a fejlesztés során. Idővel, amikor a technológia egyre elterjedtebbé válik, és újabb Ruby és Ruby on Rails verziók megjelenésével a Ruby on Rails alkalmazások teljesítménye is gyorsul, a keretrendszer kitörhet a független web2.0-ás startupok és szabadúszó programozók világából, és olyan széles körben ismert és elfogadott platformmá válhat, mint a Java EE vagy az ASP .Net.

A teljes Ruby on Rails keretrendszert nem lehet egy szakdolgozat keretein belül ismertetni. Számos szolgáltatást csak említés szintjén, vagy egyáltalán nem tárgyaltam. Ilyenek az AJAX alkalmazások fejlesztését támogató Prototype és script.aculo.us JavaScript könyvtárak, amik az ActionView modullal érkeznek. Ilyen az ActionMailer modul, ami email üzenetek előállítását és küldését kezeli, vagy az ActionWebService, ami SOAP és XML-RPC protokollok kezelését segíti. Nem volt lehetőségem tárgyalni a Ruby on Rails alkalmazások éles környezetben való futtatásának kérdéseit. Mindezek ellenére úgy érzem, hogy sikerült betekintést nyújtanom egy Ruby on Rails alkalmazás szerkezetébe, és sikerült kellő részletességgel tárgyalnom a keretrendszer középpontjában álló Model-View-Controller tervezési mintát megvalósító ActiveRecord, ActionView és ActionController modulokat.

Irodalomjegyzék

Dave Thomas, David Heinemeier Hansson: Agile Web Development with Rails 2nd edition
Pragmatic Bookshelf, 2007, ISBN: 0-9776166-3-0

Chad Fowler: Rails Recipes
Pragmatic Bookshelf, 2006, ISBN: 0-9776166-0-6

David Flanagan, Yukihiro Matsumoto: The Ruby Programming Language
O'Reilly Media, 2008, ISBN: 0-5965161-7-7

E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns
Addison-Wesley, 1995, ISBN: 0-2016336-1-2

Ruby on Rails API Documentation
http://api.rubyonrails.org

Ruby Core Documentation
http://corelib.rubyonrails.org

why's poignant guide to Ruby
http://poignantguide.net/ruby

Programming Ruby – The Pragmatic Programmer's Guide
http://www.ruby-doc.org/docs/ProgrammingRuby

The Rails Wiki
http://wiki.rubyonrails.org/rails