# Rails gördülékenyen

A Ruby on Rails 1.1-ről és a megszorítások biztosította szabadság paradoxonáról.

A ***Ruby on Rails*** egy webes alkalmazások fejlesztésére szolgáló keretrendszer, amely ígéretéhez híven sokoldalú, produktív és könnyen használható platformot biztosít a dinamikus weboldalak készítéséhez. A keretrendszerre gondolhatunk függvénykönyvtárként, mint az alkalmazásainkban felhasználható függvények gyűjteménye, de ennél valójában többről van szó: a kódban alkalmazandó kényszerek rendszeréről. Ugyan miért jó az, ha kényszerítjük magunkat bizonyos dolgokra? Azért, mert ha a kényszereket egy bizonyos célból vezetjük be, az elfedett részletek nem kötik gúzsba elménket, és a valódi probléma megoldására fókuszálhatunk. A ***Rails*** keretrendszer olyan kényszerek halmaza, amely a webes fejlesztés hatékonyságát hivatott növelni. A következőkben bemutatjuk a ***Rails*** összetevőit, hogy képet kapjunk a működéséről.

## ActiveRecord

A legtöbb webalkalmazás-fejlesztői keretrendszerhez hasonlóan, a ***Rails*** is az ***MVC***-n (***Modell-View-Controller*** – az adatmodellt, a megjelenítést és a vezérlést különválasztó architektúrán) alapuló tervezést követi, aminek következtében a kód három logikai egységben válik szét. Az adatmodellben találhatók az adatbázisban tárolt tartományobjektumok. A velük kapcsolatos feladatokat a ***Rails***-ben az ***ActiveRecords*** komponens látja el, melynek három fő tulajdonságát érdemes megjegyezni: társítás, függvényvisszahívás és ellenőrzés. A társítással adhatók meg az ***ActiveRecord***-osztályok közötti kapcsolatok, például egy az eggyel, egy a többel és több a többel, a következő módon:

```
class User < ActiveRecord::Base
  has_many    :projects
  has_one     :address
  belongs_to :department
end
```

A részletek (például táblák és külső kulcsok neve) beállítása és az adatbázis minden oszlopához tartozó objektumattribútum létrehozása automatikus – a ***Rails*** ezt a konvenciót alkalmazza a konfiguráció helyett. Az objektumok viselkedését függvényvisszahívásokkal biztosítjuk, azok teljes élettartama alatt. Például, küldhetünk elektronikus levelet a felhasználónak az adatai első mentésekor:

```
class User < ActiveRecord::Base
  after_create
➥:send_welcome_email
  after_update
:update_audit_log
end
```

Az ellenőrzések különleges függvényvisszahívások, amelyekkel a szabványos ellenőrzések végrehajtása gyerekjáték:

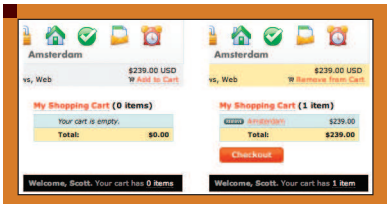
```
class User < ActiveRecord::Base
  validates_presence_of
➥:name
  validates_format_of
➥:phone, :with => /^[0-9]{3}-
➥[0-9]{3}-[0-9]{4}$/i
  validates_confirmation_of
➥:email
  validates_acceptance_of
➥:terms_of_service, :message
➥=> "must be accepted"
  validates_inclusion_of
➥:age, :in => 0..99
end
```

Azzal, hogy a társítások, függvényvisszahívások és ellenőrzések az ***ActiveRecord*** osztály definíciójába kerülnek, megbízhatóbb és könnyebben karbantartható kód jön létre.

## ActionController

Az ***ActionPacknek*** két, egymással szorosan együttműködő alkomponense van: az ***ActionController*** (vezérlő) és az ***ActionView*** (megjelenítő). Az ***ActionController*** osztályok a világhálóról elérhető, nyilvános műveleteket definiálnak. Ezek minden esetben kétféleképpen végződhetnek: átirányítással vagy produktummal. Az előbbi esetben az ügyfélhez egy ***HTTP***-válasz fejléce érkezik, mely egy másik ***URL***-re küldi tovább, míg az utóbbiban valamiféle tartalmat, többnyire ***HTML***-fájlt kap. A produktummal végződő műveleteknél az ***ActionView*** is meghívódik. Vessünk egy pillantást az alábbi vezérlőre, amely három műveletet tartalmaz:

```
class MessagesController <
➥ActionController::Base
  def list
    @messages = Message.find
➥:all
  end
              def show
    @message = Message.find
➥params[:id]
  end
  def create
    @message = Message.create
➥params[:message]
    redirect_to :action =>
➥:show, :id => @message.id
  end
end
```




**1. ábra** Tegyünk egy terméket a bevásárlókosárba

Az első művelet egy ***ActiveRecord*** objektum felhasználásával megkeresi az adatbázisban az összes levelet, majd elkészíti a ***messages/list.rhtml*** sablont. A második művelet egy adott levelet keres meg az azonosítója (***ID***) alapján, majd megjeleníti. A harmadik művelet is ***ActiveRecord*** objektumot használ, ezúttal azonban a ***HTML***-űrlapon bevitt paraméterek mentésére, miután a felhasználót visszaküldi a megjelenítés művelethez egy átirányító ***HTTP***-válasszal.
A vezérlők és a műveletek útvonalakkal ***URL***-ekhez kapcsolódnak. Az alapértelmezett útvonal a `:controller/:action/:id`, így az előbbi műveletek ***URL***-je, minden előzetes beállítás nélkül, a következő lenne: */messages/list*, */messages/show/1* és */messages/create*.
A műveletek mellett, a vezérlőkhöz a műveletek megszakítására alkalmas szűrők és a műveletek felgyorsítására szolgáló gyorsítótárak is hozzárendelhetők, például:

```
class MessagesController <
➥ActionController::Base
  before_filter :authenticate,
➥:except => :public
  caches_page    :public
  caches_action :show, :feed
end
```

## ActionView

A ***Rails*** az ***ActionView*** segítségével formázza meg az alkalmazások kimenetét, rendszerint ***HTML***-fájlokat. Az elsődleges megoldás az ***ERB***, ***Embedded Ruby***, ami biztosan ismerős lesz, ha használtunk már ***PHP***- vagy ***JSP***-szerű szintaxist. Minden ***rhtml*** kiterjesztésű sablon tartalmazhat ***Ruby*** kódot <% %> és <%= %> formázóelemek között, melyek közül csak a második esetben jön létre kimenet, például:

```
<% for message in @messages %>
  <h2><%= message.title %></h2>
<% end %>
```

Sablonrészleteket is létrehozhatunk a gyakran ismétlődő ***HTML***-kódokhoz, a sablonokba pedig olyan ***Ruby*** függvényeket ágyazhatunk, amelyek hasznos funkciókat biztosítanak számunkra, például ***Ajax*** használatát.

## Rails 1.1

A ***Ruby on Rails*** első nyilvános kiadása 2004 júliusában jelent meg, 0.5 verziószámmal. Az 1.0 verzió több mint egy évvel később, 2005 decemberében készült el. A két verzióban szinte nincs két egyforma sor. Ezt az eseményt alapos „fényezés” és tesztelés előzte meg, hogy a kód stabil legyen. Ebből arra következtethetnénk, hogy a kezdeti nagy lendület már alábbhagyott, és a fejlesztői csapat magja a siker jól megérdemelt gyümölcsét élvezi. Tévednénk. Épp ellenkezőleg, a fejlesztés üteme egy cseppet sem lassult le, és már a következő jelentős verzió is elérhető. Ez napjainkig a legfontosabb verzió, több mint 500 kisebb-nagyobb fejlesztéssel és javítással. Legtöbbjük csupán a meglévő tulajdonságok finomítása, de vannak köztük briliáns megoldások is, amelyek jelentős változásokat hozhatnak az alkalmazásfejlesztésben. Végigböngészve a változások listáját, három fő csoport különíthető el: Ajax, a tartománymodell gazdagítása és a könnyen létrehozható webszolgáltatások.

## A nagy Ajax

A ***Rails 1.1*** legjelentősebb újításai vitathatatlanul teljesen megváltoztatták az ***Ajax*** kezelését. A ***Rails*** eddig is remekül támogatta az ***Ajax*** alkalmazások létrehozását, rövid ***HTML***-kódok küldésével és beépítésével, de most már ***JavaScriptet*** is küldhet a böngészőnek. Egy weboldal több elemének frissítése tehát egy lépésben egy pillanat műve. Az igazi nagy durranás, hogy a ***JavaScriptet*** nem mi írjuk, hanem a ***Ruby*** szintaxisát használva, automatikusan jön létre. Ezt nevezzük ***RJS***-nek (***Ruby-generated JavaScript***), vagyis a ***Ruby*** által generált JavaScriptnek. A ***.rhtml*** (***Ruby HTML***) sablonok mellett ***.rjs*** (***Ruby JavaScript***) sablonok is készíthetők. Ezekbe automatikusan ***JavaScriptté*** átalakuló ***Ruby*** kódot írhatunk, amelyet egy ***Ajax*** hívás eredményeként a böngésző megkap és értelmez.
Nézzük meg ezt egy példán keresztül is. Az ***IconBuffet*** online áruház ***RJS***-t használ a bevásárlókosár megvalósításához (ki lehet próbálni a ➲ ***www.iconbuffet.com/products/amsterdam*** oldalon). Amikor a kosárba egy termék kerül, három különböző elemet kell frissíteni. ***RJS*** nélkül mintegy tucat JavaScript sort kellett volna eljuttatni a kiszolgálóhoz, több fordulóban. Most azonban mindez megoldható egyetlen lépésben, JavaScript használata nélkül. A ***kosárba teszem*** (***Add to Cart***) feliratú gomb most is a jól ismert ***Ajax*** linktámogatást használja, pontosan úgy, ahogy korábban:

```
<%= link_to_remote "Add to
➥Cart", :url => { :action =>
 "add_to_cart", :id => 1 } %>
```

Ha a linkre kattintunk, az `add_to_cart` esemény frissíti a kapcsolatot és létrehozza az ***add_to_cart.rjs*** sablont:

```
page[:cartbox].replace_html
➥:partial => 'cart'
page[:num_items].replace_html
➥:partial => 'num_items'
page["product_#{params[:id]}"].
➥addClassName 'incart'
```

A sablonból ***JavaScript*** készül, amelyet a kiszolgáló elküld a böngészőnek, hogy annak megfelelően frissítse a weboldal három kérdéses elemét. De hogy kerül ide ez a `page` objektum? Ez reprezentálja a ***JavaScriptGeneratort*** az ***RJS***-sablonokban, és még jó néhány trükk van a tarsolyában:

1) JavaScript párbeszédablak létrehozása:

```
page.alert 'Hogy s mint?'
```

2) Az elemek ***HTML***-megjelenésének megváltoztatása:

```
page.replace :element, "value"
```

3) Az elemek értékének megváltoztatása:

```
page.replace_html :element,
➥"value"
```

4) Szöveg beszúrása:

```
page.insert_html :bottom,
➥:list, '<li>Last item</li>'
```

5) Átirányítás szimulálása:

```
window.location.href:
➥page.redirect_to url_for(...)
```

6) Javascript függvény hívása:

```
page.call :alert, "Hello"
```

7) Hozzárendelés JavaScript változóhoz:

```
page.assign :variable, "value"
```

8) Effektus létrehozása:

```
page.visual_effect :highlight,
➥'list'
page.visual_effect :toggle,
➥"posts"
page.visual_effect :toggle,
➥'comment', :effect => :blind
```

9) Elem megjelenítése:

```
page.show 'status-indicator'
```

10) Elem elrejtése:

```
page.hide 'status-indicator',
➥'cancel-link'
```

11) Hivatkozás egy elemre, azonosítóval (ID-vel):

```
page['blank_slate']
page['blank_slate'].show
```

12) Elem kiválasztása *CSS*-szelektorral:

```
page.select('p')
page.select('p.welcome
➥b').first
page.select('p.welcome
➥b').first.hide
```

13) *JavaScript* beszúrása:

```
page << "alert('hello')"
```

14) Fogható (*draggable*) elem készítése:

```
page.draggable 'product-1'
```

15) Dobható (*droppable*) elem készítése:

```
page.drop_receiving
➥'wastebasket', :url => {
➥:action => 'delete' }
```

16) Rendezhető elem készítése:

```
page.sortable 'todolist', :url =>
➥{ action => 'change_order' }
```

17) Végrehajtás késleltetése:

```
page.delay(20) {
➥page.visual_effect :fade,
➥'notice' }
```

Felsoroló metódusok is használhatók, amelyek szintén létrehozzák az egyenértékű *JavaScriptet*:

```
page.select('#items li')
➥.collect('items') do |element|
                element.hide
      end
```

A kód az alábbi *JavaScriptet* eredményezi:

```
var items = $$('#items li')
➥.collect(function(value, index)
{ return value.hide(); });
```

Amellett, hogy a *views* könyvtárba *.rjs* fájlokat teszünk, beágyazott *RJS*-t is alkalmazhatunk, például:

```
def create
  # (handle action)
  render :update do |page|
          page.insert_html
➥:bottom, :list, '<li>Last
➥item</li>'
          page.visual_effect
➥:highlight, 'list'
  end
end
```

A vezérlőt nyilván nem akarjuk túl sok nézetspecifikus kóddal „szennyezni”, inkább írjunk *RJS*-függvényeket, amelyeket aztán a frissítő kódrészekből hívhatunk, például:

```
module ApplicationHelper
  def update_time
    page.replace_html 'time',
➥Time.now.to_s(:db)
    page.visual_effect
➥:highlight, 'time'
  end
end
class UserController <
➥ApplicationController
  def poll
    render :update { |page|
➥page.update_time }
  end
end
```

A hibakeresés *RJS*-ben nem mindig egyszerű, mert ha a *Ruby*-ban hiba történik, a böngészőben erről nem kapunk értesítést. Ennek elkerülésére használjuk a

```
config.action_view.debug_rjs =
➥true
```

beállítást, aminek hatására az *RJS*-hibákról értesítést kapunk az `alert()` függvénnyel.
Az éles szemű olvasó nyilván észrevette, hogy az *RJS*-sablonok kimenete felhasználta a *Prototype* nagyszerű újítását: az *Element* osztály metódusai minden *HTML*-elemből hívhatók a `$()` vagy `$$()` módon. Eszerint az `Element.show('foo')` helyett írhatjuk a `$('foo').show()`-t is. Apró változtatás, de segítségével a *JavaScriptet* természetesebben és *Ruby*-szerűen írhatjuk. A következő metódusokról van szó: `visible()`, `toggle()`, `hide()`, `show()`, `visualEffect()`, `remove()`, `update(html)`, `replace(html)`, `getHeight()`, `classNames()`, `hasClassName(osztálynév)`, `addClassName(osztálynév)`, `removeClassName(osztálynév)`, `cleanWhitespace()`, `empty()`, `childOf(szülő)`, `scrollTo()`, `getStyle(stílus)`, `setStyle(stílus)`, `getDimensions()`, `makePositioned()`, `undoPositioned()`, `makeClipping()` és `undoClipping()`.

A *Ruby* által generált *JavaScript* a *Prototype* egy másik fantasztikus újítását is hasznosítja: a *Selector* osztályt és a hozzá tartozó `$$()` függvényt. A `$()`-hoz hasonlóan, a `$$()`-t is *HTML*-elemekre való hivatkozásnál használjuk, de ez a *CSS*-szelektoroknak megfelelően keresi az egyezéseket, például:

```
// Keresd meg az összes <img>
// elemet a "summary"
// osztállyal rendelkező <p>
// elemeken belül, amelyek
// mindegyike a <div> része,
// "page" azonosítóval.
// Rejtsd el az összes egyező
// <img> elemet.
$$('div#page p.summary
➥img').each(Element.hide)
// A szelektorokban
// attribútumok is használhatók:
$$('form#foo input[type=text]')
➥.each(function(input) {
  input.setStyle({color:
➥'red'});
});
```

Ha az Olvasót ezzel még mindig nem sikerült meggyőznöm, akkor egyszerűen csak higgye el, hogy az ***RJS*** és a ***Prototype*** újításai forradalmasítják az ***Ajax*** használatát a ***Rails***-ben.

## Termékeny tartománymodellek az ActiveRecordban

Eddig a ***Rails*** nézet és vezérlés rétegében bekövetkezett fejlesztésekkel foglalkoztunk. Nézzük most meg az ***ActiveRecordot***, amelyen szintén látszik a sok törődés az 1.1 verzióban. Kezdjük mindjárt az asszociáció új típusával.
Az 1.1 verzió előtt, a ***Rails*** a `has_and_belongs_to_many`-vel támogatta a több a többel kapcsolatokat, például:

```
class Author <
➥ActiveRecord::Base
  has_and_belongs_to_many
➥:books
end
class Book < ActiveRecord::Base
  has_and_belongs_to_many
➥:authors
end
```

Egy bizonyos pontig nincs is ezzel semmi gond. Akkor leszünk bajban, ha közvetlenül a kapcsolatokhoz kell viselkedést vagy adatot rendelnünk. A megoldás az explicit hozzárendelés. Vessünk egy pillantást az alábbi kódra:

```
class Author <
➥ActiveRecord::Base
  has_many :authorships
  has_many :books, :through =>
➥:authorships
end
class Authorship <
ActiveRecord::Base
  belongs_to :author
  belongs_to :book
end
class Book < ActiveRecord::Base
  has_many :authorships
  has_many :authors, :through
➥=> :authorships
end
Author.find(:first).books.find
➥(:all, :include => :reviews)
```

A `:through` kulcsszó nélkül, egy cég számláját számláját csak több vagy egy speciális ***SQL***-lekérdezéssel szerezhetnénk meg. Az ***ActiveRecord*** most már képes automatikusan kezelni a kapcsolatot, és a társításokhoz is tiszta interfészen keresztül férünk hozzá.
A többalakú társítások tovább színesítik a tartománymodellt, megoldva azt a problémát, amikor egy modell több másikkal osztozik a kapcsolatokon. A többalakú társítással a modell absztrakt társításokat definiál, amely bármely más modellre is vonatkozhat, a részletek kezeléséről pedig az ***ActiveRecord*** gondoskodik. Nézzük meg az alábbi példát:

```
class Address <
➥ActiveRecord::Base
  belongs_to :addressable,
➥:polymorphic => true
end
class User < ActiveRecord::Base
  has_one :address, :as =>
➥:addressable
end
class Company <
➥ActiveRecord::Base
  has_one :address, :as =>
➥:addressable
end
```

Minden tapasztalt ***SQL***-fejlesztő belefutott már az „n+1 lekérdezés” problémájába, amikor a rekordokra hivatkozó rekordok keresése nagyon nagy számú adatbázis-lekérdezést eredményez. A megoldás az ***SQL*** `JOIN` használata, de a manuális alkalmazása, különösen egynél több `JOIN` esetén gyorsan követhetetlenné válik.
Ezen segít a ***Rails 1.1*** a végtelen, lépcsős mohó betöltéssel.

Az

```
Author.find(:all, :include=>
➥{ :posts => :comments })
```

megoldáshoz hasonló lekérdezések most egyetlen lépésben betöltik az összes szerzőt (`author`), a bejegyzéseiket (`post`) és az azokhoz tartozó megjegyzéseket. További példák:

```
Author.find :all, :include =>
➥{ :posts => :comments }
Author.find :all, :include =>
➥[ { :posts => :comments },
:categorizations ]
Author.find :all, :include =>
➥{ :posts => [ :comments,
➥:categorizations ] }
Company.find :all, :include =>
➥{ :groups => { :members =>
➥:favorites } }
```

Az ***ActiveRecord*** következő fontos újdonsága a beágyazott `with_scope`, ami sokkal érthetőbbé teszi az ***ActiveRecord***-objektumok kezelését, és különösen fontos a biztonsággal foglalkozó kódok esetében.
Íme egy példa:

```
Developer.with_scope :find =>
➥{ :conditions => "salary >
➥10000", :limit => 10 } do
  # SELECT * FROM developers
#WHERE (salary > 10000) LIMIT 10:
  Developer.find :all
  # parameters are merged
  Developer.with_scope :find =>
➥{ :conditions => "name =
➥'Jamis'" } do
    # SELECT * FROM developers
# WHERE (( salary > 10000 ) AND
# ( name = 'Jamis' )) LIMIT 10
    Developer.find :all
  end
  # belső szabályt használunk
# (a korábbi paramétereket
# figyelmen kívül hagyjuk)
  Developer.with_exclusive_
➥scope :find => { :conditions
➥=> "name = 'Jamis'" } do
    # SELECT * FROM developers
# WHERE (name = 'Jamis'):
    Developer.find :all
  end
end
```

Az ***ActiveRecord*** utolsó főbb újítása kényelmes szintaxist biztosít

a számítások és statisztikák eléréséhez, így nincs szükség speciális *SQL* parancsok összeállítására, például:

```
Person.count
Person.count    :conditions =>
➥"age > 26"
Person.count    :include =>
➥:job, :conditions => "age >
➥26 AND job.salary > 60000"
Person.average :age
Person.maximum :age
Person.minimum :age, :having =>
➥"min(age) > 17", :group =>
➥:last_name
Person.sum      :salary, :group
➥=> :last_name
```

## Egyszerű webszolgáltatások

A *Rails 1.1* változásainak harmadik nagy csoportja a webszolgáltatások létrehozásával kapcsolatos. Egészen pontosan a *HTML* protokoll egyes aspektusainak kezeléséről van szó, úgy ogy könnyedén implementálható *REST* stílusú *API*.
A `respond_to` az eseményekhez tartozó új metódus, amely az ügyféltől érkező *HTTP Accept* fejlécet elemzi, és többféle formátum szerint válaszolhat, például:

```
class MessagesController
➥< ActionController::Base
  def list
    @messages = Message.find
➥:all
      respond_to do |type|
        type.html # alapértel-
# mezett értékek, melyek ered-
# ménye a messages/list.rhtml
        type.xml  { render
➥:xml => @messages.to_xml }
➥# XML-t generál és küld
# a megfelelő MIME-típussal
        type.js   # létrehozza
# az index.rjs-t
    end
  end
end
```

Példánkban egy tipikus böngésző a */messages/list* kérésére megkapja az adatok *HTML* verzióját, ahogy illik. Az ugyanerre az *URL*-re vonatkozó *Ajax* kérés azonban tartalmazhat *application/javascript Accept* fejlécet is, és ennek hatására a kiszolgáló az *RJS*-sablont használná. Tegyük fel, hogy másik ügyfélprogram is kapcsolódik a webszolgáltatást nyújtó alkalmazásunkhoz, és *application/xml* formában várja az adatokat. Ezt is ugyanaz az esemény fogja kezelni. Nem kell többé azon töprengenünk, milyen bonyolult lenne a webalkalmazásunkhoz másik *API*-t illeszteni, mert sosem volt ennél könnyebb.
A példában egy új kapcsolót láthatunk a `render` metódushoz: a `:xml`-t. Pontosan úgy működik, mint a `render(:text => text)`, de `content-type= application/xml` és `charset= UTF-8` beállításokkal. A `content-type` manuálisan is beállítható a `:content_type` kapcsolóval, például:

```
render :action => "atom.rxml",
➥:content_type =>
➥"application/atom+xml"
```

A tömböknek, az asszociatív tömböknek és az *ActiveRecordnak* most már van `to_xml` metódusa, és minden objektumnak van `to_json` metódusa. Ezekkel a hatékony adalékokkal, néhány billentyűleütéssel megvalósítható az alkalmazásunk adatainak reprezentációja gépi értelmezéshez, például:

```
message.to_xml
message.to_xml(:skip_instruct
➥=> true, :skip_attributes =>
➥[ :id, bonus_time,
➥:written_on, replies_count ])
firm.to_xml :include =>
➥[ :account, :clients ]
[1,2,3].to_json => "[1, 2, 3]"
"Hello".to_json => "\"Hello\""
Person.find(:first).to_json =>
➥"{\"attributes\": {\"id\":
➥\"1\", \"name\": \"Scott
➥Raymond\"}}"
```

Az előbbi példák bemutatták, milyen egyszerűen aktiválhatunk egy csak olvasható *API*-t, de mi van akkor, ha bemeneti adatokat is akarunk fogadni az interfészen? A válasz szerencsére most sem bonyolultabb a korábbiaknál:

```
class MessagesController <
➥ActionController::Base
  def create
    @message = Message.create
➥params[:message]
    redirect_to :action =>
➥:show, :id => @message.id
  end
end
```

Egy pillanat! Ez véletlenül nem ugyanaz a kód, mint az esemény *API* nélküli megvalósítása? Valóban.
A *Rails* értelmezi a beérkező *POST* üzenet *HTTP content-type* mezőjét, és szétbontja a `params` objektumba, pontosan úgy, mintha az adatok egy webes űrlapról érkeztek volna.
Az *application/xml* tartalmú kérések kezelésekor egy *XmlSimple* hasítótábla jön létre, melynek a neve megegyezik az XML legfelsőbb szintű elemének a nevével. Az *XML*-adatok kezelése automatikus, de mi történik az egyéb típusú tartalmakkal? Használjuk a `parsers` paramétert:

```
ActionController::Base.param_
➥parsers['application/atom+xml
➥'] = Proc.new do |data|
  node = REXML::Document.new
➥data
  { node.root.name => node.root
}
end
```

## És a lényeg csak most kezdődik...

A *Rails 1.1* új jellemzőinek épp csak a felszínét kapargattam meg, így a *Rails* egészéről szinte semmit sem mondtam. Mindenesetre remélem, hogy a cikk ízelítőt adott a *Rails* legújabb fejlesztéseiből. Mélyebb ismeretek megszerzéséhez és a Rails közösségének megismeréséhez látogassa meg a *rubyonrails.com* weboldalt, ahol kapcsolódó könyveket, képernyőket, dokumentációkat, oktatóanyagokat, példaalkalmazásokat, blogokat, levelezőlistákat és *IRC* csatornákat talál.

*Linux Journal 2006., 147. szám*

*Scott Raymond* a Rails projekt tagja, professzionális Rails fejlesztő és tanácsadó. Tíz éve foglalkozik webes alkalmazások készítésével, a pozíciók és az ügyfelek típusának széles skáláján. Írásai olvashatók a scottraymond.net címen, előadását hallottuk a 2006 júniusában tartott RailsConf konferencián, és az O'Reilly adta ki a könyvét. Scott a Kansasi Egyetemen szerezte diplomáját.