Rails MVC, session
Gyakorlat

Kovács Gábor

2010. október 1.

Az előző gyakorlaton összeállított weboldal terveket használjuk a gyakorlat során.

A megvalósítandó rendszerünk a házi feladatok javítását lehetővé tevő portál, amelynek most a bejelentkezési és regisztrációs oldalait vizsgáljuk meg részletesen, de létrehozzuk a megoldások képernyő adatstruktúráit is az előzőleg már generlált feladatok modell mellé. A megoldáshoz a [1] könyv példáit vesszük alapul.

Az 1. ábrán látható bejelentkezési képernyőn egy form található két szövegmezővel, amelyek a Neptun kód és a jelszó bevitelére szolgálnak. A Mehet felíratú gombra kattintás aktiválja a bejelentkezés funkció, amely ha sikeres a megoldások képernyőre navigál, ha sikertelen, akkor hibaüzenetet megjelenítve a képernyőn marad. A képernyőn található még egy regisztrációs link, amely a regisztráció oldalra navigálja a felhasználót, és egy elfejeltett jelszó link, amely az elfelejtett jelszó oldalra viszi át a felhasználót.

A 2. ábra a regisztrációs oldal képernyőképét mutatja. A regisztrációs oldal egy formot tartalmaz két szövegmezővel (Neptun-kód és email cím), két jelszómezővel (jelszó és jelszó még egyszer) és egy nyomógombbal. A Mehet felirató nyomógomb sikeres regisztráció esetén, vagyik ha nem létezik még az adatbázisban ilyen Neptun-kóddal rendelkező felhasználó, valamint a jelszó és a megerősített jelszó mezők értéke megegyezik, átnavigálja a felhasználót a megoldások képernyőre. Ellenkező esetben hibaüzenet jelenik meg, és a felhasználó a képernyőn marad.

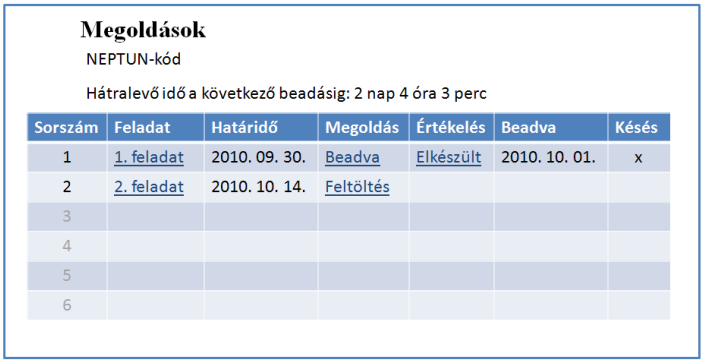
A megoldások modell struktúrája az előző félév kurzusában megvalósított hallgató típusú felhasználó megoldás modelljéből származik. A hallgatói megoldások képernyő, amely a 3. ábrán látható az aktuális felhasználó által feltöltött, illetve beadandó megoldásokról szolgáltat információt. Az oldal üdvözli a belépett felhasználót, megmutatja a következő beadásig hátralevő időt, és egy táblázatban összefoglalja a megoldandó feladatokra vonatkozó

1. ábra. Bejelentkezés képernyő

2. ábra. Regisztrációs képernyő

információkat. A táblázat hét oszlopot tartalmaz. A Sorszám oszlop a megoldandó feladatok sorszámát tartalmazza, és fekete színnel jelenik meg, ha a feladatot már létezik, és szürke színnel, ha csak a jövőben kerül kiadásra. A

Feladat oszlop egy link a feladat kiírását tartalmazó weboldalra. A Határidő oszlop a feladat beadási határideje. A Megoldás oszlop beadott feladat esetén a beadott feladat módosítása képernyőre navigálja a felhasználót, még be nem adott feladat esetén pedig a feladat beadás képernyőre. Az Értékelés oszlop egy linket tartalmaz, ha a megoldásokhoz fűzött megjegyzések elkészültek, és feltöltésre kerültek a javítók által. A Beadva oszlop a feladat megoldása utolsó feltöltésének dátumát adja meg. A késés oszlop jelölőnégyzetet tartalmaz, amely akkor igaz, ha a Beadva oszlopban szereplő dátum a Határidő oszlopban szereplő dátum utáni.



3. ábra. Megoldások képernyő

A oktató/javító típusú felhasználó esetén a megoldások modell három tulajdonságot feltételez. A megoldás kései beadását, a megoldás elfogadását és a megoldáshoz rendelt megjegyzést. A modell kialakításánál ezeket is figyelembe vesszük.

Hozzuk létre e három képernyőtervhez a képernyő adatait modellező adatstruktúrákat! Kezdjük a regisztráció, illetve bejelentkezés képernyőkkel. A két képernyő ugyanazt az adatstruktúrát érinti, azt, melyik a felhasználó adatait tárolja.

```
rails generate model User
```

Az ilymódon létrehozott `User` nevű modellhez egy `users` nevű tábla fog tartozni az adatbázisban, amelyet a Rails alkalmazásunk `db/migrate/*create_users.rb` forrásának szerkesztésével definiálhatunk. A `self.up` metódust

szerkesztjük és a `create_table` függvény blokkjában definiáljuk a tábla attribútumait a típus és az oszlopnév megadásával. Emellett egyéb paraméterek is megadhatók minden egyes attribútumhoz. A Rails a következő típusú attribútumok definícióját támogatja: `binary`, ami BLOB-nak felel meg, `boolean`, amiből egy hosszú egész szám lesz, `decimal`, ami egy SQL `number`, `float`, ami lebegőpontos szám, `integer`, ami egész szám, `string`, ami az SQL `varchar`-nak felel meg, `text`, amiből adatbáziskezelőtől függően `text` vagy `clob` típus lesz, továbbá az időre vonatkozó típusok `date`, `datetime`, `time`, `timestamp`.

A regisztráció képernyő (2. ábra) három attribútumot definiál, a Neptun-kódot, a jelszót, és az email címet. A bejelentkezés képernyő (1. ábra) ehhez nem ad hozzá további tulajdonságokat. Mivel tudjuk, hogy a rendszert felhasználók két csoportja fogja használni, a feladatot beadó hallgatók és a feladatokat javító oktatók, egy további attribútumot is felveszünk. A `:neptun` szimbólummal azonosított maximum hat karakter hosszú Neptun-kód attribútum string típusú. A `:password` szimbólummal azonosított jelszó szintén string és 40 karakter hosszban limitált. Az `:email` szimbólummal azonosított email cím attribútum szintén string típusú. A `:student` attribútum `boolean` típusú, és alapértelmezett értéke `true`.

```
def self.up
  create_table :users do |t|
        t.string :neptun, :limit=>6
       t.string :password, :limit => 40
       t.string :email
       t.boolean :student, :default => true
       t.timestamps
  end
end
```

Ez a következő SQL állítással ekvivalens MySQL adatbáziskezelő esetén. A tábla neve a modell nevének többesszáma lesz.

```
CREATE TABLE `users` (
`id` int(11) NOT NULL auto_increment,
`email` varchar(255) default NULL,
`password` varchar(40) default NULL,
`neptun` varchar(6) default NULL,
`student` tinyint(1) default '1',
`created_at` datetime default NULL,
`updated_at` datetime default NULL
PRIMARY KEY (`id`)
```

```
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

Az adatbázis aktulizálása, illetve a visszaállás egy korábbi verzióra az alábbi módon történik:

```
rake db:migrate

rake db:migrate VERSION=0
rake db:migrate
```

Ezután a `User` osztály példányaival közvetlenül módosíthatjuk az adatbázis rekordjait. Az alábbi Ruby konzolos részlet egy új rekordot hoz létre először a memóriában, majd a `save` metódus meghívásával egy rekordot is beszúr az `users` táblába. A rekord automatikusan rendelkezik egy `id` nevű azonosítóval, ami alapján a tábla a `find` osztálymetódussal kereshető. A `:first` az első rekordot adja vissza. A `find_by_` kezdetű metódusok egy attribútum értéke alapján keresnek, illetve több attribútum alapján történő kereséshez az egyes attribútumok nevét `_and_`-del választjuk el a metódus nevében.

```
rails console
u = User.new( :neptun=>'oweyoa', :email=>'kovacsg@tmit.
    bme.hu',
:student=>false )
u.save
u.id
v = User.find(:first)
v.neptun
w = User.find(1)
w.neptun
w.email='kovacsg@db.bme.hu'
w.save

x = User.find_by_id(1)
# find_by_<attribute>
y = User.find_by_id_and_neptun(1,"OWEYOA")
```

A feladatok modelljét `Task` néven már az előző gyakorlaton létrehoztuk, a megoldásokét `Solution` néven pedig most generáljuk. A feladatok modelljét azért különítjük el a megoldásokétól, mert a feladatok közösek az összes hallgató felhaszálóra nézve.

```
rails generate model Solution
```

A feladatok modelljének migrációját 3. ábra alapján a következőképpen definiáltuk. Legyen egy egész típusú attribútumunk, amely a feladat sorszámát azonosítja, és értékét az `1..6` intervallumból veszi fel. A második attribútum a feladat kiírásánaj URL-jét tartalmazza, amelyet linkként teszünk majd ki az oldalra. A harmadik attribútum a feldat beadási határideje.

```
def self.up
  create_table :tasks do |t|
    t.integer :number, :within => 1..6
        t.string :url
        t.date :deadline
        t.timestamps
  end
end
```

A megoldások tábla (`solutions`) egy kapcsolótábla a felhasználók (`users`) és a feladatok (`tasks`) között, azokra az egész típusú azonosítójukkal, a `user_id`-vel és a `task_id`-vel hivatkozunk. A megoldás lehet kései (`late` attribútum), illetve elfogadott állapotba kerülhet (`accepted` attribútum), ezetek boolean értékként kezeljük, az előbbi alapértelmezett értéke `true`, utóbbié `false`.

```
def self.up
  create table :solutions do |t|
        t.integer :user_id
        t.integer :task_id
        t.boolean :late, :default=>true
        t.boolean :accepted, :default =>false
        t.timestamps
  end
end
```

Hajtsuk végre az adatbázis migrációt! Időközben rájöttünk, hogy elfelejtkeztünk a hallgatói megoldásokra adott javítói megjegyzésekről, ezért hozzunk létre egy migrációt, amely hozzáadja a `solutions` táblához a string típusú `comment` attribútumot. Végezetül javítsuk a feledékenységünket egy ismételt adatbázis migrációval.

```
rake db:migrate

rails generate migration AddCommentToSolutions comment:
   string
```

```
rake db:migrate
```

A `solutions` táblába felvett `user_id` és `task_id` lehetővé teszi számunkra, hogy az adatbázis táblák közötti relációkat modell osztályok közötti kapcsolatokra vetítsük rá. Ehhez módosítanunk kell a modell osztályokat.

Egy hallgató típusú felhasználó minden egyes kiadott feladatra adhat be megoldást. Ez azt jelenti, hogy a `user` modell osztály példányainak egy rendelkezniük kell a beadott megoldások listájával, és azzal a listával, hogy mely feladatokra adott be az adott felhasználó megoldást. Az előbbit a `has_many :solutions` deklarációval érjük el, amely létrehoz egy `solutions=` nevű settert és egy `solutions` nevű gettert. Megjegyendő, hogy a megoldások itt a konvenció értelmében többes számban szerepel ugyanis tömbről van szó. Az utóbbi, tehát a feladatok listája, közvetlenül nem érhető el a felhasználók táblájából, csakis a megoldások kapcsolótáblán keresztül, ezért ezt jelezzük a `:through=> :solutions` paraméterrel.

```
class User < ActiveRecord::Base
  has_many :solutions
  has_many :tasks, :through => :solutions
end
```

A megoldások modellt tekintve a felhasználók modellnél leírtakhoz hasonló gondolatmenetet követhetünk.

```
class Task < ActiveRecord::Base
  has_many :solutions
  has_many :users, :through => :solutions
end
```

A megoldások tábla kapcsolótáblaként működik a felhasználók és a feladatok táblák között, egy megoldás példány pontosan egy felhasználóhoz és pontosan egy feladathoz tartozik. Ezt jelöljük a `belongs_to` helperrel, amely létrehozza nekünk a `task=`, illetve a `user=` settereket, valamint a `task`, illetve a `user` gettereket. Mivel pontosan egy-egy példányról van szó mindkét esetben a konvenció értelmében a modell nevek itt egyes számban szerepelnek.

```
class Solution < ActiveRecord::Base
  belongs_to :task
  belongs_to :user
end
```

A Rails konzolon (`rails console`) ellenőrizhetjük, hogy a modell osztályok közötti kapcsolatok valóban létrejöttek-e. A `2` `id`-vel rendelkező felhasználó adott be egy megoldást az `1`-es sorszámmal rendelkező feladatra.

```
irb(main):004:0> v = User.find 2
=> #<User id: 2, neptun: "aaaaaa", email: "a@bme.hu",
    encrypted_password: "aaaaaa", student: true,
    created_at: "2011-03-29 10:45:23", updated_at:
    "2011-03-29 10:45:23", salt: nil>
irb(main):005:0> v.solutions
=> [#<Solution id: 1, task_id: 1, user_id: 2, late:
    true, accepted: false, created_at: "2011-03-29
    10:47:47", updated_at: "2011-03-29 10:47:47">]
irb(main):006:0> v.tasks
=> [#<Task id: 1, number: 1, url: "http://localhost
    :3000/tasks/1", deadline: "2011-02-28 23:59:59",
    created_at: "2011-03-01 12:17:26", updated_at:
    "2011-03-01 12:17:26">]
```

Ha az adatbázisunkat kezdeti adatokkal szeretnénk feltölteni, és azt nem konzolon akarjuk megtenni minden egyes adatbáziskezelő esetére, akkor a `db/seeds.rb` fájlban kell elhelyeznünk a modell osztályok `create` metódusait meghívó Ruby állításokat, majd kiadnunk a 'verb!rake db:seed! parancsot. A feladatok tábla feltöltése a következőképp történhet például:

```
Task.create :number=>2, :url=>'https://twiki.db.bme.hu/
    twiki/bin/view/Student/Ruby/RubyMasodikFeladat20111'
    , :deadline=>DateTime.parse('2011-03-14 23:59')
Task.create :number=>3, :url=>'https://twiki.db.bme.hu/
    twiki/bin/view/Student/Ruby/RubyHarmadikFeladat20111
    ', :deadline=>DateTime.parse('2011-03-28 23:59')
```

A modell osztályaink elkészülten, nekiláthatunk az adatok megjelenítéséhez, illetve az adatbevitel felületen keresztül való megvalósításához.

Létrehoztuk a felhasználó regisztráció és a bejelentkezés képernyők közös modelljét, készítsük el a nézetet és a vezérlést hozzá! Legyen négy nézetünk egy `index`, ahol a felhasználók listája érhető el, egy `new`, ami nem lesz más, mint a regisztráció, egy `edit`, ahol a felhasználó szerkesztheti saját adatait, továbbá egy `show`, ahol a felhasználó megnézheti a saját adatait.

```
rails generate controller Users index new show edit
```

Kezdjük a regisztrációval, amit a nézetek könyvtárában találunk `users/new.html.erb` név alatt. A HTML formot a `form_for` helperrel hozzuk létre,

amely alapértelmezés szerint a POST HTTP metódus használja, vagyis a HTTP `form method` attribútumának értékét `post`-ra állítja. Az első paraméter, a `:user` szimbólum, a HTTP POST paramétereinek hivatkozására haszálatos szimbólum. A második paraméter a POST eseményt kezelő kontrollert és metódust definiálja, ami jelen esetben az aktuális `Users` kontroller `create` metódusa, vagyis a `/users/create` URI. A létrejövő `form` a Rails alkalmazás alapértelmezett karakterkódolását támogatja.

A `form.label` helperrel HTML `label`-t hozhatunk létre, mind a négy beviteli mezőre ezt használjuk, az első paramétere annak a beviteli mezőnek az azonosítója, amelyre a címke vonatkozik. Szövegmezőt a `form.text_field` helperrel hozunk létre, amelynek első paramétere a `text type` attribútummal létrehozott HTML `input id` és `name` attribútumait határozza meg. A `:neptun`-ból `user_neptun` azosító lesz mind a `label`-ben, mind a szövegmező `id`-jében. A `form.password_field` helper egy `password` típusú HTML `input`-ot hoz létre. Végül a `submit_tag` egy `submit` típusú `input`-ot hoz létre, vagyis egy nyomógombot, amelynek címkéje az első paraméterrel fog megegyezni.

```
<h1>Regisztráció</h1>

<%= flash[:notice] %>

<div>
  <fieldset>
  <legend>Új felhasználó létrehozása</legend>
  <% form_for :user, :url => { :action => "create" } do
       |form| %>
    <div>
      <%= form.label :neptun %>:<br />
      <%= form.text_field :neptun, :class => "short" %>
    </div>
    <div>
      <%= form.label :email %>:<br />
      <%= form.text_field :email, :class => "short" %>
    </div>
    <div>
      <%= form.label :jelszó %>:<br />
      <%= form.password_field :password, :class => "
          short" %>
    </div>
    <div>
```

```
        <%= form.label :jelszó_még_egyszer %>:<br />
        <%= form.password_field :password_confirmation, :
            class => "short" %>

    </div>
    <%= submit_tag "Elküld" %>
  <% end %>
  </fieldset>
</div>
```

Definiáljuk a formot lekezelő metódust a `users_controller.rb` fájlban található `users` kontrollerben. A kontroller első megnyitáskor négy metódust tartalmaz: `index`, `show`, `new` és `edit`. A nézet fenti definíciója szerint definiálnunk kell ezeken kívül egy `create` nevű metódust, amely a nézet nyomógombjára klikkelés eseménye által generált HTTP POST üzenetet fogja feldolgozni.

A `create` metódus először létrehoz egy új példányt az `User` nevű modell osztályból a HTTP POST `:user` szimbólummal hivatkozott elemei alapján, és az hozzárendeli a `@user` azonosítójú példányváltozóhoz. Ez az `:user` megegyezik az előző kódrészlet `form_for` helperének első paraméterével.

Ezután a `save` metódussal megkíséreljül beszúrni a `@user` változót az adatbázis `users` táblájába. Amennyiben sikeres a művelet, akkor a felhasználói session (azonosítás utáni böngészés a portálon) során ezt az azonosítót fogjuk használni, hozzárendeljük a `@current_user` példányváltozóhoz és a `session` hash-szerű objektum `:user` eleméhez.

Ezt követően visszejelzést küldünk egy `flash` üzenettel a következő oldalra, hogy a bejelentkezés sikeres volt. Ha az üzenetben az ASCII karakterkészleten kívüli karaktert is használunk, például ékezetes karaktert, akkor meg kell változtatnunk a Ruby forrásfájl karakterkódolását, amit a `# Encoding: UTF-8` sor a fájl elejére történő beszúrásával teszünk meg.

A létrehozás akciót lekezelvén már csak az maradt hátra, hogy megjelenítsük a következő oldalt, amit a `show` akcióra történő átirányítással érünk el, vagyis meghívjuk a `redirect_to` metódust. Az átirányítás során a felhasználó `id` attribútumát adjuk át paraméterként. A `show` metódusban a paraméter lekezelését semmi nem végzi el, viszont a `before_filter`, amely a `new` és a `create` metódusok kivételével az összes metódus előtt lefut, meghívja a `find_user` metódust, ami betölti az adatbázisból az `:id` paraméternek megfelelő rekordot a `@user` példányváltozóba.

Ha a `create` metódusban a `@user` objektum mentése sikertelen volt, akkor visszairányítjuk a felhasználót a `new` metódus által reprezentált regisztrációs oldalra. A `new` egy inicializálatlan példányt hoz létre a `User` modell

osztályból.

```
# Encoding: UTF-8
class UsersController < ApplicationController
  before_filter :find_user, :except => [:new,:create]

  def new
    @user = User.new
  end

  def show
  end

  def create
    @user = User.new(params[:user])

    if @user.save
      @current_user = @user
      session[:user] = @user.id
      flash[:notice] = "Sikeres bejelentkezés"
      redirect_to :action => "show", :id => @user.id
    else
      render :action => "new"
    end
  end

  private
  def find_user
    @user = User.find(params[:id])
  end
end
```

A regisztrációs űrlapot kipróbálva, majd utána az adatbázisba beszúrt rekordot megvizsgálva érszevehetjük, hogy a jelszó titkosítatlanul került tárolásra, ami a rendszer biztonságossága szempontjából előnytelen. Ezért módosítást eszközölünk a `User` modellen, hozzáadunk egy a kriptográfiában salt-nak nevezett attribútumot, amelyet a jelszó titkosításakor használunk fel. Ehhez a következő migrációt hajtjuk végre.

```
rails generate migration AddSaltToUser salt:string
```

Ez a migráció a `db/migrate` könyvtárban létrehozott egy `_add_salt_to_user`-rel végződő Ruby fájlt, amely a migráció nevéből kikövetkeztette, hogy egy

új attribútumok kívánunk hozzáadni az `users` táblához, és ezt be is szúra a `self.up` metódusba. A `add_column` metódus első paramétere a tábla neve, amelyhez új oszlopot kívánunk hozzáadni, a második paraméter az új oszlop neve, a harmadik az új oszlop típusa. Emellett a `salt` attribútum hosszát 40 karakterben maximáljuk. A migrációban egy másik változtatást is végrehajtunk, átnevezzük a `password` attribútumot `encrypted_password`-re. A `rename_column` első paramétere a táblát azonosítja, a második az átnevezendő oszlopot, a harmadik az oszlop új nevét adja meg. Ezzel párhuzamosan a visszagörgetésre felkészülendő az ellentés módosítást hozzáadjuk a `self.down` metódushoz.

```
def self.up
  rename_column :users, :password, :encrypted_password
  add_column :users, :salt, :string, :limit => 40
end
def self.down
  remove_column :users, :salt
  rename_column :users, :encrypted_password, :password
end
```

Az `users` tábla után módosítjuk a `User` modell osztályt is a `user.rb`-ben. Először létrehozunk egy settert és gettert a `password` példányváltozóhoz.

Ezt követően validációs mechanizmusokkal bővítjük ki a modellünket. A modellünkben a Neptun-kódnak egyedinek kell lennie és nem lehet `nil` értékű. Az utóbbit a `validates_presence_of` validációs osztálymetódus végzi el, az előbbit pedig a `validates_uniqueness_of` osztálymetódus ellenőrzi. Az egyediség ellenőrzése mellett beállítjuk a mező értékének maximális hosszát 6-ra, valamint azt, hogy a szöveg nem kisbetűérzékeny. A `validates_length_of` nevű a jelszót hosszát ellenőrző validációs metódus azt vizsgálja, hogy a hossz 4 és 40 közé esik-e, és a vizsgálat akkor hajtódik végre, ha a `password_required` metódus `true`-val tér vissza. A `validates_confirmation_of` metódus a regisztrációs nézet két jelszómezőjének egyezését vizsgálja.

A `password_required` akkor tér vissza `true`-val, ha vagy még nincs elmentett kódolt jelszó, vagy a nézet definiálta a `password` változó értékét.

A `before_save` osztálymetódus minden `save` hívás, vagyis adatbázis beszúrás előtt meghívja az `encrypt_password` metódust, amely a `password` helyett az `encrypted_password` értékét állítja. Az `encrypt_password` metódus először ellenőrzi, hogy a jelszó értéke `nil`-e, majd a `new_record` hívással azt vizsgálja meg, hogy a rekordot elmentettük-e már az adatbázisba. Ha még nem, akkor definiáljuk a kódoláshoz használt `salt` értékét, majd meghívjuk az `encrypt` osztálymetódust a jelszó értékével és a `salt` értékével.

Az `encrypt` metódus egy hash értéket generált a két bemeneti paraméter alapján, amely a kódolt jelszó értéke lesz.

```
class User < ActiveRecord::Base
  attr_accessor :password

  #validates_length_of       :email, :within => 3..100
  #validates_uniqueness_of   :email, :case_sensitive =>
      false
  validates_presence_of     :neptun
  validates_uniqueness_of   :neptun, :limit => 6, :
      case_sensitive => false
  validates_length_of       :password, :within => 4..40,
      :if => :password_required?
  validates_confirmation_of :password, :if => :
      password_required?

  before_save :encrypt_password

  def self.encrypt(pass, salt)
    #Digest::SHA1.hexdigest("--#{salt}--#{pass}--")
    Digest::SHA2.hexdigest(self.salt + pass)
  end

  def encrypt_password
    return if password.blank?
    if new_record?
      #self.salt = Digest::SHA1.hexdigest("--#{Time.now
          }--#{email}--")
      self.salt = ActiveSupport::SecureRandom.base64(8)
    end
    self.encrypted_password = User.encrypt(password,
        salt)
  end

  def password_required?
    encrypted_password.blank? || !password.blank?
  end
end
```

A sikeres regisztráció után a felhasználó az `users` kontroller `show` nézetére kerül át (`users/show.html.erb`), amelyet az alábbi kódrészlet mutat be. A

nézet felül egy linket tartalmaz az `index` akcióra. A központi HTML `div` kiírja a felhasználó Neptun-kódját a `h` HTML metakarakter konverziós metódus segítségével, valamint `mailto` formában megjeleníti a felhasználó email címét. Az oldal alján két további linket helyezünk el. Az egyik az `edit` oldalra irányít át, a másik meghívja a `destroy` akciót, amely egy javascriptes megerősítés (`:confirm`) után egy `DELETE` HTTP kéréssel (`:method => :delete`) törli a felhasználó azonosítójához tartozó rekordot az adatbázisból.

```
<h1><%= link_to "Felhasználók", :action => "index" %> <
    /h1>
<%= flash[:notice] %>

<div>
  <h2><%=h @user.neptun%></h2>
  <p><%= mail_to h(@user.email) %></p>

  <div>
    <%= link_to "Szerkeszt", :action => "edit", :id =>
        @user.id %>
    <%= link_to "Töröl", { :action => "destroy", :id =>
        @user.id },
    :confirm => "Biztos?", :method => :delete %>
  </div>
</div>
```

Az előző `UsersContoller` osztálydefiníciót kiegészítjük a `destroy` metódussal, amely először törli a paraméterül átadott `:id`-hez a `find_user` metódus által kikeresett felhasználót az adatbázis `users` táblájából', flash üzenetet állít össze a következő oldal számára, amely az `index` akció lesz.

```
# Encoding: UTF-8
class UsersController < ApplicationController
  def show
  end

  def destroy
    @user.destroy
    flash[:notice] = "Felhasználó törölve"
    redirect_to :action => "index"
  end
end
```

Az `index` akció nezete egy listát jelenít meg a rendszer felhasználóiról. Ehhez a kontrollerben elő kell állítanunk a felhasználók listáját. Először módosítjuk a `before_filter`-t, a kivételek listáját kibővítjük az `index` akcióval, ugyanis nincs szükség itt egy konkrét felhasználó példányára. Az `index` metódus az összes (`:all`) felhasználót lekéri az adatbázisból a `netpun` attribútum szerint rendezve (`:order`).

```
class UsersController < ApplicationController
  before_filter :find_user, :except => [:index,:new,:
    create]

  def index
    @users = User.find(:all,:order=>"neptun")
  end
```

Az `index` akció nézetét a `users/index.html.erb` fájlban definiáljuk, és ennek feladata az adatbázisban elérhető összes felhasználó kilistázása, amit egy rendezetlen HTML listával valósít meg. A lista elemeit a `index` metódusban beállított `@users` példányváltozón való iterálásból veszi az oldal, és egy linket tesz ki a `show` nézetre, valamint egy `mailto`-t az felhasználó email címére.

```
<h1>Felhasználók</h1>
<%= flash[:notice] %>

<div>
<ul>
  <% for user in @users %>
    <li>
      <%= link_to h(user.neptun), :action => "show", :
        id => user.id %>
      <span><%= h(user.email) %></span>
    </li>
  <% end %>
</ul>
</div>
```

A `users/edit.html.erb` fájlban definiált szerkesztés nézet a felhasználó saját attribútumainak módosítására alkalmas. A nézet `form`-ját a kontroller `update` metódusa dolgozza fel. A form felépítése hasonló a `new` nézetéhez. Az oldal alján a `Mégse` link a felhasználót az `index` oldalra navigálja.

```
<h1>Felhasználók szerkesztése</h1>
```

```
<div>
  <fieldset>
  <legend>Felhasználók adatainak szerkesztése</legend>
  <% form_for :user, :url => {:action => "update", :id
     => @user.id } do |form| %>
    <div>
      <%= form.label :neptun %>:<br />
      <%= form.text_field :neptun, :size => 35 %>
    </div>
    <div>
      <%= form.label :email %>:<br />
      <%= form.text_field :email, :size => 35 %>
    </div>
    <%= submit_tag "Mentés" %>
    <%= link_to "Mégse", :action => "index" %>
  <% end %>
  </fieldset>
</div>
```

Az `edit` nézet `Mentés` eseményét a kontroller `update` metódusa kezeli le. A `before_filter` betölteti a paraméterül kapott `id`-hez tartozó felhasználót a `@user` példányváltozóba a `find_user` metódussal. Az `update` metódus megkísérli az `@user` rekord attribútumainak frissítését a HTTP POST paramétereivel. Amennyiben ez sikeres, egy sikeres flash üzenetet adunk át a következő nézetnek, ami a `show`. Ellenkező esetben maradunk a szerkesztés oldalon.

```
class UsersController < ApplicationController
  def update
    if @user.update_attributes(params[:user])
      flash[:notice] = "Sikeres frissítés"
      redirect_to :action => "show", :id => @user.id
    else
      render :action => "edit"
    end
  end
end
```

Hivatkozások

[1] Derek DeVries and Mike Naberezny. *Rails for PHP Developers*. The Pragmatic Programmers, Feb 2008.