Rails MVC, session
Gyakorlat

Kovács Gábor

2010. október 1.

Az előző gyakorlaton összeállított Rails telepítést használjuk a gyakorlat során.

A megvalósítandó rendszerünk a házi feladatok megoldását nyilvántartó portál, amelynek most a bejelentkezési és regisztrációs oldalait vizsgáljuk meg részletesen, de létrehozzuk a megoldások képernyő adatstruktúráit is. 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.

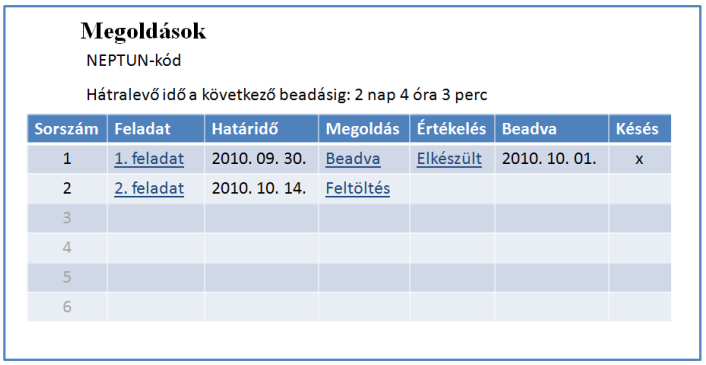
A megoldások képernyő, amely a 3. ábrán látható az aktuális felhasználó által feltöltött, illetve beadandó megoldásokról szolgáltat információt. Az oldal üdvözli a belépett felhasználót, megmutatja a következő beadásig hátralevő időt, és egy táblázatban összefoglalja a megoldandó feladatokra vonatkozó 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

1. ábra. Bejelentkezés képernyő

2. ábra. Regisztrációs képernyő

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

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 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
        t.string :password, :limit => 40
        t.string :email
        t.boolean :student, :default => true
        t.timestamps
  end
end
```

Ez a következő SQL állítással ekvivalens. 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(255) 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")
```

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` példányáváltozóba.

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 kiprobálva, 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 felhaszáló 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
```

A következő lépés a bejelentkezés nézet és kontroller megvalósítása. Létrehozzuk a `Sessions` nevű kontrollert egyetlen `new` akcióval.

```
rails generate controller Sessions new
```

A `new` nézete egy HTML `form`-ot tartalmaz, amelyet a konroller még nem létező `create` metódusa fog lekezelni. Az űrlap egy szövegmezőből és egy jelszómezőből áll. A szövegmező értékét a POST `:neptun` paraméteréhez kötjük, a jelszó értékét pedig a `:password` paraméterhez.

```
<h1>Bejelentkezés</h1>

<%= flash[:notice] %>

<div>
  <fieldset>
  <legend>Kérem, adja meg NEPTUN kódjat és jelszavát</
      legend>
  <% form_tag :action => "create" do %>
  <div>
    <label for="neptun">Neptun</label><br />
    <%= text_field_tag :neptun, params[:neptun], :class
        => "short" %>
  </div>
  <div>
    <label for="password">Jelszó</label><br />
    <%= password_field_tag :password, params[:password
       ], :class => "short" %>
  </div>
  <%= submit_tag "Bejelentkezés" %>
  <% end %>
  </fieldset>
</div>
```

A belépés, vagyis a session létrehozása funkció megvalósítása előtt ki kell egészítenünk a `User` osztályunkat egy a felhasználó hitelesítésére alkalmas metódussal. Az `authenticate` metódus két paraméterrel rendelkezik, egy `neptun` kóddal és egy `pass` jelszóval. A metódus először a rekordok között megkeresi a paraméterül kapott `neptun` értékkel rendelkező rekordot, és meghívja a hitelesítést elvégző `authenticated?` metódust. Az `authenticated?` metódus azt ellenőrzi, hogy az `encrypt` metódus visszatérési értéke azonos-e az `encrypted_password` értékével. Ha ennek a visszatérési értéke `false` vagy a keresés nem talált a `neptun` paraméternek megfelelő felhasználót, akkor a hitelesítés sikertelen, és a visszatérési érték `nil`. Sikeres hitelesítés esetén a visszatérési érték a felasználó rekordja.

```
class User < ActiveRecord::Base
  def self.authenticate(neptun,pass)
    user = find_by_neptun(neptun)
    user && user.authenticated?(pass) ? user : nil
  end
```

```
  def authenticated?(pass)
    encrypted_password == User.encrypt(pass, salt)
  end
end
```

A kontrollerben a `create` metódus megkísérli hitelesíteni a felhasználót a `User` modell imént létrehozott `authenticate` metódusával, amelyet a POST paramétereivel hív meg. Sikeres hitelesítés esetén beállítja a `session` hash `:user` elemének értékét az aktuálisan hitelesített felhasználóra, majd átnavigál a `users` kontroller `show` akciójának nézetére az aktuális felhasználó `:id`-jével mint paraméterrel. Sikertelen hitelesítés esetén flash üzenetet hozunk létre, és maradunk a `new` akcióval reprezentált bejelentkezés oldalon.

```
# Encoding: UTF-8
class SessionsController < ApplicationController
  def new
  end

  def create
    @current_user = User.authenticate(params[:neptun],
       params[:password])
    if @current_user
      session[:user]=@current_user.id
      redirect_to :controller => "users", :action => "
         show", :id=>@current_user.id
    else
      flash[:notice] = "Hibás NEPTUN kód vagy jelszó"
      render :action => 'new'
    end
  end

  def destroy
    reset_session
    flash[:notice] = "Sikeres kijelentkezés"
    redirect_to :action => "new"
  end
end
```

Mivel a `session` hash-sel képesek vagyunk az aktuális felhasználó azonosítójának átadására a portál oldalai között, olyan funkciókat, mint a beállítások módosítása, regisztráció, bejelentkezés, illetve a kijelentkezés, elhelyeztetjük layout szinten, a `application.html.erb` fájlban. Ha a felhasználó be van

jelentkezve, vagyis a nemsokára megírandó `logged_in?` metódus `true`-val tér vissza, akkor a felhasználó számára elérhetővé teszünk lét linket, a beállítások módosítását, amely a `users` kontroller `show` akciójára mutat az aktuális felhasználó `:id`-jével, valamint a kijelentkezést, amelyet a `sessions` kontroller `destroy` akciója kezel majd le. Ha a `logged_in?` metódus visszatérési értéke `false`, akkor a felhasználó két másik linket lát, a regisztrációét, ami a `users` kontroller `new` akciója kezel le, valamint a bejelentkezését, amit a `sessions` kontroller `new` akciója kezel le.

```
<div>
<% if logged_in? %>
  <%= link_to "Beállítások", :controller => "users", :
      action => "show", :id => @current_user.id %>
   
  <%= link_to "Kijelentkezés", :controller => "sessions
      ", :action => "destroy" %>
<% else %>
  <%= link_to "Regisztráctió", :controller => "users",
      :action => "new" %>
   
  <%= link_to "Bejelentkezás", :controller => "sessions
      ", :action => "new" %>
<% end %>
</div>
```

A `sessions` kontrollerben a kijelentkezés linkre kattintás lekezelésére meg kell valósítanunk a `destroy` metódust. Ez először törli a sütit a `reset_session` metódus meghívásával, majd üzenet állít be a következő bejelentkezés oldalnak, vagyis a `new` akciónak, hogy a kijelentkezés sikeres volt.

```
class SessionsController < ApplicationController
  def destroy
    reset_session
    flash[:notice] = "Sikeres kijelentkezés"
    redirect_to :action => "new"
  end
end
```

A felhasználói session állapotát az alkalmazás keterében, az `application_controller.rb` fájlban kezeljük le. A fájl első sora, a `protect_from_forgery` egy egyedi tokent rendel hozzá minden `form_for` és `form_tag` által létrehozott HTML `form`-hoz, amelynek valósidágát ellenőrzi a Rails keret a HTTP POST után. Ez által megnehezíthető a session elrablásával próbálkozó támadó dolga.

Az alkalmazás legelőször a `before_filter` metódus által deklarált módon meghívja a `initialize_user` metódust, amely a `session` hash-ből előkeresi a `:user` kulcshoz tartozó elemet, ha létezik ilyen, és ebből helyreállítja a session felhasználóját a `@current_user` példányváltozóba.

Egy helper metódust definiálunk annak ellenőrzésére, hogy van-e bejelentkezett felhasználónk. A `logged_in?` akkor tér vissza sikeresen, ha a `@current_user` példányváltozó a `User` modell osztály egy példánya, ami csak akkor állhat fenn, ha a felhasználót visszaállítottuk a `session` hash-ből.

```
class ApplicationController < ActionController::Base
  protect_from_forgery

  before_filter :initialize_user

  helper_method :logged_in?

  protected
  def logged_in?
    @current_user.is_a?(User)
  end

  def initialize_user
    @current_user = User.find_by_id(session[:user]) if
        session[:user]
  end
end
```

A következő gyakorlatra felkészülendő létrehozunk két további modellt, a feladatokét `Task` néven, és a megoldásokét `Solution` néven. A feladatok modelljét azért különítjük el a megoldásokétól, mert a feladatok közösek az összes hallgató felhaszálóra nézve.

```
rails generate model Task
rails generate model Solution
```

A feladatok modelljének migrációját 3. ábra alapján a következőképpen definiáljuk. 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 hiányozhat (`missing` attribútum), ezetek boolean értékként kezeljük, az utóbbi alapértelmezett értéke `true`.

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

Hivatkozások

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