DIPLOMAMUNKA

Szilágyi Szabolcs

Debrecen

2008

Debreceni Egyetem
Informatika Kar

Webalkalmazás fejlesztése symfony keretrendszerben

Témavezető:
Dr. Kuki Attila
Egyetemi Adjunktus

Készítette:
Szilágyi Szabolcs
Programtervező matematikus

Debrecen
2008

Tartalomjegyzék

1. Bevezetés

Manapság egyre nagyobb népszerűségnek örvendenek a dinamikus weboldalak. Az egyik legismertebb, dinamikus oldalak készítését lehetővé tevő nyelv a PHP. Ha egy statikus oldalt dinamikus tulajdonságokkal akarunk felruházni, akkor nincs más dolgunk, mint néhány jól átgondolt PHP utasítást beágyazni a HTML kódba. Igen ám, de ezzel az oldal megjelenítésének a kódját összevegyítjük a dinamikus tulajdonságot megvalósító kóddal. Ebből persze a felhasználó nem fog észrevenni semmit, de a fejlesztő igencsak megnehezítheti a saját dolgát, ha a HTML kódba nem megfelelően átgondoltan ágyazza be a PHP utasításokat, hiszen egy nagy projekt esetén, ha tovább kell fejleszteni az oldalt, akkor egy idő után teljesen átláthatatlan lesz a HTML és a PHP vegyített kód. Arról nem is beszélve, hogy egy nagyobb projektet általában csapatban fejlesztenek, és a megjelenítést valamint az üzleti logikát programozó fejlesztő nem azonos személy. A legtöbb szoftverfejlesztéssel foglalkozó cég ma már valamilyen keretrendszert használ, aminek segítségével a dizájnerek és a fejlesztők hatékonyan tudnak egymással dolgozni. Az utóbbi időben a webalkalmazás keretrendszerek egyre népszerűbbek lettek, így dolgozatom témájául a symfony keretrendszer bemutatását választottam. Amióta a symfony weboldalt elindították (http://www.symfony-project.org), számos fejlesztő letöltötte és installálta a keretrendszert szerte a világon, elolvasták az on-line dokumentációt és megírták az első alkalmazásukat a symfony-val. Online fórumok, levelező listák, és IRC csatornák ideális támogatást nyújtanak, kezdő symfony webfejlesztők számára.

Symfony egy komplett keretrendszer, amit webalkalmazások fejlesztésének optimalizálásához terveztek. Számos eszközt és osztályt tartalmaz, amik megrövidítik egy bonyolult webalkalmazás fejlesztési idejét. Továbbá automatizál bizonyos folyamatokat, így a fejlesztő teljes egészében az adott alkalmazásra tud koncentrálni. A symfony-t teljes egészében PHP 5-ben írták. Különféle projektekben alaposan letesztelték és jelenleg is használják számos országban magas volumenű üzleti weboldalak üzemeltetésnél. Kompatibilis a legtöbb elérhető adatbázis motorral, beleértve a MySQL-t, PostgreSQL-t, Oracle-t, és Microsoft SQL Servert. Unix és Windows platformon egyaránt fut.

1.1. Symfony tulajdonságai

A symfony-t az alábbi követelmények teljesítésére készítették:

- Könnyű installálni és telepíteni a legtöbb platformon
- Adatbázis-motor független
- Egyszerű használni, mindazonáltal elég flexibilis komplex feladatok kidolgozásához

- Kompatibilis a legtöbb webes tervezési mintával
- Elég stabil hosszú távú projektek alkalmazásához
- Könnyen olvasható kód megjegyzésekkel, a könnyű karbantarthatóságért
- Könnyen bővíthető és integrálható más gyártók könyvtáraival

1.2. Automatikus webprojekt funkciók

A webprojektek legtöbb általános funkciója automatizálva van a symfony-ban az alábbi módon:

- A megjelenítés sablonokat használ, amiket HTML dizájnerek a keretrendszer ismerete nélkül beépíthetnek. Segédmodulok csökkentik a megjelenítő kód írásának mennyiségét azáltal, hogy egyszerű függvényhívásokkal nagyméretű kódokat tudunk beágyazni
- Űrlap támogatás automatikus validációval, ami biztosítja az adatbázisban az adatok típusának helyességét, valamint a felhasználói élményt
- Kivételkezelés, ami megvédi az alkalmazásokat hibás adatok érkezésekor
- A gyorsítótár menedzselő funkció csökkenti a sávszélesség használatát és a kiszolgáló terhelését
- A hitelesítő funkciók megkönnyítik a korlátozott szolgáltatások és a felhasználói biztonság kezelését
- Beépített email és API menedzselő funkciók megengedik az alkalmazásoknak, hogy túlhaladják a klasszikus böngésző interakciókat
- A felsorolások sokkal felhasználóbarátabbak az automatikus laptördelésnek, rendezésnek, és szűrésnek köszönhetően
- Beépülő modulok magas szintű bővíthetőséget tesznek lehetővé
- Ajax interakciókat könnyű implementálni az on-line segédmoduloknak köszönhetően

1.3. Fejlesztői környezet és eszközök

A vállalatok követelményeinek teljesítése céljából, hogy azok megtarthassák a kódolási konvencióikat és projektszervezési szabályaikat, a symfony teljes egészében testre szabható.

- A kódgenerálási eszközök nagyon jók prototípuskészítéshez
- A hibakeresési ablak gyorsítja a hibaelhárítást azáltal, hogy megjeleníti az összes információt, amire a fejlesztőnek szüksége van azon az oldalon, amin épp dolgozik
- A parancssori értelmező automatizálja az alkalmazás fejlesztését két kiszolgáló közt

- Lehetőség van azonnali beállítások érvénybe léptetésére
- A naplózási funkciók pontos adatokat adnak az adminisztrátoroknak az alkalmazás tevékenységeiről

1.4. Gyors alkalmazásfejlesztés (RAD: Rapid Application Development)

Régen a webalkalmazás fejlesztés fárasztó és lassú folyamat volt. A szokásos szoftvertechnológiai életciklusokat követve a webalkalmazások fejlesztését el sem kezdték addig, míg bizonyos követelményeket nem rögzítettek, rengeteg UML diagramot rajzoltak, és tonnaszám készültek előzetes dokumentációk. Ez a fejlesztés általános sebességének, a programozási nyelvek egysíkúságának volt köszönhető (futtatni, fordítani, újraindítani és még ki tudja miket kell csinálni, mire a programunkat működni láttuk). Ma már gyorsabban mennek a dolgok, az ügyfelek hajlanak arra, hogy megváltoztassák a gondolkodásmódjukat a projekt fejlesztése folyamán. Természetesen elvárják a fejlesztő csapattól, hogy gyorsan átdolgozzák és módosítsák az alkalmazást az igényeiknek megfelelően. Szerencsére a szkript nyelvek használata, mint a PHP és a Perl, könnyen lehetővé teszik más programozási stratégiák alkalmazását, mint például a RAD alkalmazását. Ezen módszerek egyike, hogy a fejlesztést olyan gyorsan elkezdik, amilyen gyorsan csak lehet, így az ügyfél figyelemmel tudja kísérni a munkát egy prototípus segítségével, és esetlegesen további igényeket adhat a programmal kapcsolatban. Ezután az alkalmazás egy iteratív folyamat során készül el, rövid fejlesztési ciklusokban funkció gazdag verziók készülnek. A symfony egy tökéletes eszköz a RAD megvalósításához. Valójában a keretrendszert egy webfejlesztő cég készítette, ahol a RAD elveket követték a saját projektjeik elkészítésé során. A symfony használatának megtanulása nem egy új nyelv megtanulását jelenti, hanem jobb reflexek elsajátítását abból a célból, hogy még hatékonyabb alkalmazásokat fejlesszünk.

2. Symfony kód

Első pillantásra a symfony keretrendszer által készített alkalmazás meglehetősen ijesztően néz ki. Számos könyvtárt és szkriptet tartalmaz, a fájlok pedig PHP osztályok, HMTL fájlok, vagy ezek keveréke. Ezen kívül látni fogunk hivatkozásokat bizonyos osztályokra, amik a projekt könyvtárában valahol máshol találhatók. A könyvtárak egymásba ágyazottsága néhol a hatos szintet is elérheti, de mihelyst megértjük a látszólagos bonyolultság okát, hirtelen megvilágosodunk, hogy mennyire természetes ez a könyvtárstruktúra.

2.1. Az MVC minta

Symfony a klasszikus web tervezési mintán alapszik, ami MVC architektúraként vált ismertté, és ami az alábbi három részből áll:

- A Model azon információkat tartalmazza, amin az alkalmazás dolgozik (üzleti logika)
- A View (nézet) a felhasználóval való interakciónak megfelelően megjeleníti a Modelt egy weblapon
- A Controller megváltoztatja a Modelt és/vagy a Nézetet a felhasználói eseményeknek megfelelően

Az MVC architektúra elkülöníti az üzleti logikát és a megjelenítést, ami jobb karbantarthatóságot eredményez. Például ha az alkalmazásunkat egy webböngészőben és egy hordozható eszközön is futtatni szeretnénk, akkor csak egy új nézetre van szükségünk. Megtarthatjuk az eredeti Controller-t és Model-t. A Controller segít elrejteni a kérés (request) által használt részleteket a model és nézet elől. A model elvonatkoztatja a logikát az adatoktól, például a használt adatbázis típusát az alkalmazásól. A modern nyelvek OOP (Objektum Orientált Programozás) képessége még könnyebbé teszi a programozást, az objektumok beágyazzák az üzleti logikát, örökölnek egymástól, és gondoskodnak a hibátlan elnevezési konvencióikról. Ha egy nem objektum orientált nyelven akarjuk megvalósítani az MVC architektúrát, akkor az névtér ütközéseket, kód duplikációt okoz, valamint a kód általánosságban nehezen olvasható lesz.

Symfony MVC implementáció:

- Model (Model layer)
  - Adatbázis abszrakció (Database abstraction)
  - Adatelérés (Data access)
- Nézet (View layer)
  - Nézet (View)
  - Sablon (Template)
  - Elrendezés (Layout)
- Kontroller (Controller layer)
  - Elsődleges vezérlő (Front controller)
  - Művelet (Action)

Az elsődleges vezérlő (front controller) és az elrendezés (layout) minden művelet (action) számára közös egy alkalmazásban. Több kontrollerünk és elrendezésünk is lehet, de mindegyikből egyre van szükség. A front kontroller egyszerű MVC komponens, és nem kell megírnunk, ugyanis a symfony legenerálja nekünk. A másik jó hír az, hogy a model réteg osztályai szintén automatikusan generálódnak, a megadott adatstruktúrának megfelelően. Ez a Propel könyvtár feladata, ami

gondoskodik az osztályvázakról és kódgenerálásról. Ha a Propel külső kulcs megszorítást vagy adatmezőt talál, akkor gondoskodik speciális lekérdező és beállító metódusokról, amik megkönnyítik az adatmanipulációt. Az adatbázis műveletek teljesen láthatatlanok lesznek számunkra, mivel ezzel egy Creole nevezetű komponens fog foglalkozni. Tehát ha az egyik pillanatban úgy döntünk, hogy megváltoztatjuk az adatbázis-motort, semmilyen kódot nem kell újraírnunk, csak egyetlen konfigurációs paramétert kell megváltoztatnunk. A nézet is nagyon könnyen előállítható egy konfigurációs fájl segítségével, és ehhez sem kell programoznunk.

2.2. A symfony alapvető fontosságú osztályai

- sfController a kontroller osztály, ez kódolja a kérést (request) és átadja a megfelelő műveletnek (action).
- sfRequest tárolja a kérés összes elemét (paramétereket, sütiket, fejléceket, stb.)
- sfResponse tartalmazza a válsz fejléceket és tartalmakat. Ez az az objektum, ami végül HTML válasszá konvertálódik, és ezt kapja meg a felhasználó.
- A context (amit az sfContext::getInstance()-ból nyerünk) tárolja a hivatkozásokat az alapvető objektumokra és az aktuális konfigurációra, ez bárhonnan elérhető.

Amint látható az összes symfony osztály az sf prefixet használja, mint ahogyan a sablonokban az alap symfony változók is. Ezzel elkerüljük a névütközést a saját osztály és változóneveinkkel, valamint az alap keretrendszer osztályok könnyen felismerhetővé válnak.

2.3. Projekt struktúra

A symfony-ban a projekt egy szolgáltatás és művelet halmaz, ami egy megadott tartománynév alatt érhető el, és amik azonos objektum modellen osztoznak. A műveletek logikailag alkalmazásokba vannak csoportosítva. Egy alkalmazás normál esetben függetlenül futtatható ugyanazon projekt többi alkalmazásától. A legtöbb esetben egy projekt két alkalmazást tartalmaz: egy front-office és egy back-office alkalmazást, amik ugyanazon adatbázison osztoznak. Persze csinálhatunk olyan projektet is, ami számos minioldalt tartalmaz, ahol minden oldal egy különálló alkalmazás. Minden alkalmazás egy vagy több modulból áll. Egy modul általában egy vagy több logikailag összefüggő oldalt tartalmaz. A modulok kezelik a különböző műveleteket is. pl.: egy bevásárlóKosár modul tartalmazhat hozzáad, megjelenít, és módosít műveletet. A műveletek kezelése majdnem olyan mint a klasszikus webalkalmazásokban az oldalak kezelése, bár két művelet eredményezheti ugyanazt az oldalt.

Fájl fastruktúra: Minden webprojekt általában ugyanazon típusú tartalmakon osztozik, amik az alábbiak:

- egy adatbázis, mint például a MySql vagy a PostgreSQL
- statikus fájlok (HTM, képek, JavaScriptek, style sheet-ek, stb.)
- a felhasználók vagy az adminisztrátorok által feltöltött fájlok
- PHP osztályok és könyvtárak
- idegen könyvtárak (third-party scripts)
- kötegfájlok (parancssori értelmezőben futtatható szkriptek)
- log fájlok (az alkalmazás és/vagy a szerver által írt nyomkövetési információk)
- konfigurációs fájlok

A symfony gondoskodik egy szabványos fájl fastruktúráról, amit logikailag rendez az MVC architektúrának megfelelően. Ez az a fastruktúra, ami automatikusan elkészül minden egyes projekt, alkalmazás, vagy modul inicializálásakor. Természetesen teljes egészében testre szabhatók és újrarendezhetők a fájlok és könyvtárak az egyéni konvencióknak megfelelően.

Gyökér fastruktúra: Az alábbi könyvtárak találhatók egy symfony projekt gyökerében:

- apps/ A projekt minden egyes alkalmazásához tartalmaz egy könyvtárt. (tipikusan *frontend* és *backend* a front- és back-office számára)
- cache/ A beállítások cache verzióját tartalmazza, valamit ha beállítjuk, akkor a projekt műveleteinek és sablonjainak gyorsan betölthető változatát. A gyorsító mechanizmus ezeket a fájlokat használja, hogy felgyorsítsa a webkérésekre adott válaszokat.
- config/ A projekt általános beállításait tartalmazza.
- data/ Itt tárolhatjuk a projekt adatfájljait, mint például az adatbázis sémát, az adatbázis táblákat létrehozó SQL fájlokat, stb.
- doc/ A projekt dokumentációját tárolja, beleértve a saját és a PHPdoc által generált dokumentációt.
- lib/ Külső osztályok és könyvtárak számára elkülönített könyvtár. Ide rakhatjuk azokat a kódokat, amiken az alkalmazásaink osztoznak.
- log/ A symfony által közvetlenül generált érvényes log fájlokat tárolja. Tartalmazhat webszerver, adatbázis, vagy a projekt bármely részéről származó log fájlt. A symfony alkalmazásonként és környezetenként egy log fájlt hoz létre.

- plugins/ Az alkalmazáshoz telepített kiegészítéseket tárolja.
- test/ Erőforrás és funkcionális teszteket tartalmaz, amik kompatibilisek a symfony tesztelő keretrendszerével.
- web/ A webszerver számára ez a gyökérkönyvtár. Az internetről elérhető fájlok ebben a könyvtárban találhatók.

Alkalmazás fastruktúra: Minden alkalmazás fastruktúrája azonos, az alábbi módon:

- config/ YAML konfigurációs fájlokat tartalmaz. Az alkalmazás teljes konfigurációja itt történik eltekintve az alapértelmezett paraméterektől, amiket maga a keretrendszer tartalmaz.
- i18n/ Az alkalmazás nemzetköziségéhez használt fájlokat tartalmazza.
- lib/ A konkrét alkalmazás osztályai és könyvtárai találhatók itt.
- modules/ Ez tárolja a modulokat, amik az alkalmazás funkcióit tartalmazzák.
- templates/ Az alkalmazás egészére kiterjedő sablonokat tartalmazza. Ez az egyik könyvtár, amit az összes modul lát. Alapértelmezésként tartalmaz egy *layout.php* fájlt, ami a fő elrendezési minta, ami tartalmazza a modulok sablonjait.

Az i18n/, lib/, és modules/ könyvtárak egy új alkalmazás inicializálásakor üresek. Egy alkalmazás osztályai nem képesek elérni ugyanazon projekt másik alkalmazásának metódusait és attribútumait.

Modul fastruktúra: Minden alkalmazás tartalmaz egy vagy több modult. Minden modulnak van egy alkönyvtára a modules/ könyvtárban, aminek a neve a modul beállítása során meghatározódik.

- actions/ Általában egyetlen actions.class.php fájlt tartalmaz, amiben a modul összes műveletét tároljuk. Természetesen egy modul különféle műveleteit különböző fájlokba is szervezhetjük.
- config/ Szokásos konfigurációs fájlok a modul lokális paramétereivel.
- lib/ Osztályokat és könyvtárakat tárol az adott modul számára.
- templates/ A modul műveleteinek megfelelő sablonokat tartalmaz. Az alapértelmezett sablon, indexSuccess.php, a modul beállítása során generálódik.

Web fastruktúra: Nagyon kevés megszorítás van a web könyvtárra, ami a nyilvánosan elérhető fájlok könyvtára. Néhány elnevezési konvenció gondoskodik az alapértelmezett működésről a sablonokban. Következzen egy példa a web könyvtár struktúrájáról:

- css/ Style sheet-eket tartalmaz .css kiterjesztéssel.
- images/ .jpg, .png, vagy .gif formátumú képeket tartalmaz.
- js/ JavaScript fájlokat tartalmaz .js kiterjesztéssel.
- uploads/ Felhasználók által feltöltött fájlokat tartalmazhat.

3. A symfony telepítése

Ha látni akarjuk, hogy mire képes a symfony, akkor a legegyszerűbb módja a kipróbálásának a *sandbox* telepítése. A *sandbox* egy egyszerű fájlarchívum, ami tartalmaz egy üres symfony projektet, annak minden könyvtárával, egy alapértelmezett alkalmazással és beállítással. A *sandbox* letölthető a http://www.symfony-project.org honlapról. Töltsük le, és csomagoljuk ki a beállított szerverünk gyökérkönyvtárába (általában web/ vagy www/). Tegyük fel, hogy a kicsomagolt *sandbox* az *sf_sandbox/* lesz. Teszteljük le az alkalmazott symfony keretrendszert a következő parancs futtatásával:

../www/sf_sandbox/symfony -V

Ekkor láthatjuk a keretrendszerünk verziószámát:

symfony version 1.1.4 (C:\EasyPHP 2.0b1\www\sf_sandbox\lib\symfony)

Hogy biztosak legyünk benne, hogy a webszerverünk is tudja futtatni a sandbox-ot, a böngészőnkbe írjuk be a következő URL-t:

http://localhost/sf_sandbox/web/frontend_dev.php/

Ha egy symfony üdvözlő lapot látunk a böngészőnkben, akkor kész is vagyunk az installálással. Ha uninstallálni szeretnénk a sandbox-ot, akkor elég csak letörölnünk az *sf_sandbox/* könyvtárat a *www/* könyvtárból.
A sandbox tartalmaz egy projektet és egy alapértelmezett alkalmazást *frontend* néven. Ha egy új alkalmazást akarunk létrehozni, akkor ezt a *symfony generate:app myapp* paranccsal tehetjük meg, ahol a myapp az alkalmazásunk neve. A symfony legenerálja az alkalmazást az apps/ könyvtárba a már fentebb említett fastruktúrával. A projekt *web/* könyvtára alapértelmezés szerint tartalmaz egy *index.php* és egy *frontend_dev.php* fájlt. Az *index.php* lesz az új alkalmazás front controller-e.
A *symfony* parancsot mindig a projektünk gyökérkönyvtárából kell kiadni, mert minden feladat, amit ezzel a paranccsal hajtunk végre projektfüggő.
A *web/* könyvtárban lévő szkriptek lesznek az alkalmazás belépési pontjai. Ahhoz, hogy elérjük őket az internetről, a webszervert konfigurálni kell.

4. Az oldalkészítés alapjai symfony keretrendszerrel

Mint ahogyan azt már korábban említettük, a symfony modulokba csoportosítja az oldalakat. Mielőtt írnánk egy oldalt, készítenünk kell egy modult, ami az inicializálása során egy üres váz lesz a megfelelő struktúrával, amit a symfony felismer. A symfony parancssora segítségével a modulok készítése automatizált folyamat, csak meg kell hívnunk a *generate:module* parancsot az alkalmazás és a modul nevével. A *sandbox* tartalmaz egy alapértelmezett alkalmazást *frontend* néven. Ha ehhez az alkalmazáshoz hozzá szeretnénk adni egy modult, akkor a következő parancsot kell futtatnunk a projekt gyökérkönyvtárából. Legyen a modul neve *content*:

symfony generate:module frontend content

Eltekintve az *actions/* és *templates/* könyvtáraktól, ez a parancs csak 3 fájlt hoz létre. Az egyik a *test/* mappában van, és egyelőre nem foglalkozunk vele. Az *actions/* könyvtárban létrejövő *action.class.php* fájl az alapértelmezett nyitó oldalra irányítja a felhasználót. A *template/* könyvtárban létrejövő *indexSuccess.php* fájl pedig üres.

Az *action.class.php* tartalma:

```
<?php
class contentActions extends sfActions{
  public function executeIndex()  {
    $this->forward('default', 'module');
  }
}
```

Minden egyes új modul számára a symfony készít egy alapértelmezett index action-t, ami egy *executeIndex()* metódusból és egy *indexSuccess.php* sablon fájlból áll. Az újonnan létrejövő oldalt a következő URL-el tudjuk megnézni:

http://localhost/frontend_dev.php/content/index

A Symfony-ban az oldalak mögötti üzleti logikát a műveletek (actions) reprezentálják, a megjelenítést pedig a sablonok végzik. Logika nélküli oldalak egy üres műveletet igényelnek. Adjuk hozzá egy új műveletet az oldalunkhoz, aminek a neve legyen *show*. Ehhez az *executeShow*() metódust kell hozzáadnunk a *contentActions* osztályunkhoz. A művelet metódus neve mindig az *executeXxx*() elnevezési konvenciót követei, ahol az *Xxx* a művelet neve (az első karakter nagybetű). Ha most megnézzünk a *http://localhost/frontend_dev.php/content/show* URL-t a symfony panaszkodni fog, hogy a *showSuccess.php* sablon hiányzik. Ez teljesen normális, hiszen a symfony-ban az oldalnak egy műveletből és egy sablonból kell állnia. A művelet tehát vár egy sablont a megjelenítéshez. A sablon egy fájl, ami a modul *templates/* könyvtárában található, a neve

pedig a művelet nevéből és a Success szóból tevődik össze. Pl: showSuccess.*php*

A sablonok feltételezik, hogy csak megjelenítő kódot tartalmaznak, ezért ha lehet minél kevesebb PHP kódot tegyünk beléjük. Ha feltétlenül muszáj PHP kódot tenni a sablonunkba, akkor lehetőleg kerüljük az általános PHP szintaktikát, és használjunk helyette olyat, ami nem PHP programozók számára is érthető. Például:

```
<p>Hello, world!</p>
<?php
if ($test){
  echo "<p>".time()."</p>";
}
?>
helyett használjuk a következőt:

<p>Hello, world!</p>
<?php if ($test): ?>
 <p><?php echo time(); ?></p>
<?php endif; ?>
```

Az információátvitel a művelettől a sablonig a következőképpen történik: Művelet feladata az összes olyan bonyolult számítás, mint például adatvisszanyerés, tesztelés, vagy változók meghatározása a sablonban. A műveletosztályokban a symfony közvetlenül elérhetővé teszi az attribútumokat a sablonok számára a globális névtér segítségével.

Pl: Művelet osztály kódrészlete:

```
<?php
class contentActions extends sfActions{
  public function executeShow()  {
    $today = getdate();
    $this->hour = $today['hours'];
  }
}
Sablon kódrészlet:

<p>Hello, world!</p>
<?php if ($hour >= 18): ?>
 <p>Vagy jó estét kellene mondanom? Már <?php echo $hour ?> óra van.</p>
<?php endif; ?>
```

Műveletek közt hyperlinket a *link_to()* helper metódussal tudunk létrehozni. A helper egy symfony által generált PHP függvény, amit sablonok belsejében használhatunk, HTML kóddal tér vissza, és

gyorsabb, mint az általunk írt HTML kód. Ha csak egy URL-t akarunk generálni, akkor az *url_for()* helpert használhatjuk. A *link_to()* helpernek lehetnek argumentumai és attribútumai. Az opcionális argumentum vagy egy asszociatív tömb, vagy kulcs=érték párokból álló, szóközzel elválasztott sztringek. Akárhányszor symfony helpert használunk, ami HTML tagot generál, megadhatunk további tag attribútumokat (mint pl.: class attribútum) opcionális attribútumként. Ezek az attribútumok ráadásul nem igényelnek dupla idézőjelet, ugyanis a symfony szépen formázva XHMTL formában rendereli nekünk, ezért is gyorsabbak a helperek, mint az általunk írt HTML kódok.

Ha a felhasználó formon (POST request) vagy URL-en (GET request) keresztül küld információt, akkor ezt az *sfRequest* objektum *getParameter()* metódusával tudjuk kinyerni. A konvenció az, hogy minden *executeXxx()* alakú metódus első paramétere az *sfRequest* objektum. Ha egyszerű adatmanipulációról van szó, akkor nem feltétlenül kell használnunk műveletet ahhoz, hogy kinyerjük a kért paramétereket, ugyanis a sablonok hozzáférnek egy *sf_params* nevezetű objektumhoz, aminek van egy *get()* metódusa, amivel le tudjuk kérni a kívánt paramétert. Ez megfeleltethető a műveletek *getParameter()* metódusával. Az *sf_params* objektum sokkal hatékonyabb, mintha egy tömbhöz írnánk egy lekérdező metódust. Például ha egy igényelt paraméter létezését akarjuk tesztelni, akkor egyszerűen használhatjuk az *sf_params->has()* metódust ahelyett, hogy tesztelnénk az aktuális változót a *get()* metódussal. A symfony-ban a legtöbb lekérdező metódusnak, így a *request->getParameter()* metódusnak a műveletben, valamint az *sf_params->get()* metódusnak a sablonban (ami tulajdonképpen ugyanazt a metódust hívja meg ugyanazon az objektumon) is megadható egy második argumentum, egy alapértelmezett érték abban az esetben, ha a keresett paraméter nem létezik.

5. Symfony konfigurálása

Az egyszerű és könnyű használhatóság kedvéért a symfony definiál néhány konvenciót, ami eleget tesz a legtöbb általános követelménynek a szabványos alkalmazásokat illetően. Mindamellett, hogy egyszerű és hatékony konfigurációs fájlokat használ, lehetőségünk van szinte mindent testre szabni, hogy a keretrendszer és az alkalmazásunk miként kommunikáljon egymással. Ezekkel a fájlokkal képesek vagyunk speciális paraméterekkel felruházni az alkalmazásainkat.

- A symfony konfigurációs fájljai YAML fájlok, de természetesen más formátumot is választhatunk.
- Konfigurációs fájlok egy projekt könyvtárszerkezetében projekt, alkalmazás, és modul szinten is vannak.

- Számos konfigurációs beállítást definiálhatunk. A symfony-ban a konfigurációs beállításokat környezetnek hívjuk.
- A konfigurációs fájlokban definiált értékek elérhetők az alkalmazásunk PHP kódjaiból.
- Továbbá a symfony engedélyez PHP kódot a YAML fájlokban, és más trükköket is alkalmaz azért, hogy a konfigurációs rendszer még rugalmasabb legyen.

5.1. A konfigurációs rendszer

A symony konfigurációs rendszerének a célja a következő:

- Hatékony: Majdnem minden dolog ami konfigurációs fájlokkal menedzselhető, az menedzselve is van.
- Egyszerű: Néhány beállítás, aminek a megváltoztatására ritkán van szükség, nem jelenik meg egy normál alkalmazásban.
- Könnyű: A konfigurációs fájlok könnyen létrehozhatók, olvashatók,és szerkeszthetők.
- Testre szabható: Az alapértelmezett konfigurációs nyelv a YAML, de ez lehet XML, vagy bármilyen más, a fejlesztő által preferált formátum.
- Gyors: A konfigurációs fájlokat nem az alkalmazás, hanem a konfigurációs rendszer dolgozza fel, valamit a PHP szerver által gyorsan értelmezhető kódrészletekké alakítja.

5.2. YAML szintaktika és symfony konvenciók

A symfony tehát alapértelmezett módon a YAML formátumot használja a hagyományosabb INI, vagy XML formátum helyett. A YAML szerkezeti felépítésében sorbehúzásokat használ, és gyorsan olvasható. Nagyon egyszerű nyelv, az XML-hez hasonló módon ír le adatokat, de sokkal egyszerűbb szintaktikát használ. Különösen hasznos olyan adatok leírására, amiket tömbökbe vagy kulcstranszformációs táblába lehet fordítani. Például:

```
$house = array(
    'family' => array(
    'name'     => 'Doe',
    'parents'  => array('John', 'Jane'),
    'children' => array('Paul', 'Mark', 'Simone')
  ),
    'address' => array(
    'number'   => 34,
    'street'   => 'Main Street',
```

```
    'city'     => 'Nowheretown',
    'zipcode'  => '12345'
  )
);
```

Ez a PHP tömb automatikusan létrehozható a következő YAML sztrig elemzésével:

```
house:
  family:
    name:     Doe
    parents:
      - John
      - Jane
    children:
      - Paul
      - Mark
      - Simone
  address:
    number: 34
    street: Main Street
    city: Nowheretown
    zipcode: "12345"
```

Ez a YAML sztring rövidebb formában is írható a következőképpen:

```
house:
  family: { name: Doe, parents: [John, Jane], children: [Paul, Mark, Simone] }
  address: { number: 34, street: Main Street, city: Nowheretown, zipcode:
"12345" }
```

Látható, hogy egy YAML fájlt sokkal gyorsabban meg lehet írni, mint egy XML fájl (nincsenek záró tagok és idézőjelek), és hatékonyabb az .ini fájloknál is, amik nem támogatják a hierarchiát. Nézzünk néhány lényeges konvenciót, amit a YAML szintaktika esetén be kell tartanunk:

- Először is, nem szabad tabulátort használni a YAML fájlokban, helyettük használjunk szóközöket. A YAML szintaktikai elemzők nem tudják értelmezni a tabulátort, ezért szóközzel kell kihúzni a sorainkat (symfony konvenció: kétszeres szóköz a sorkihúzásra).
- Ha a paramétereink sztringek, amik szóközzel kezdődnek és végződnek, akkor használjunk egyszeres idézőjelet, valamit ha a sztringünk speciális karaktert tartalmaz, akkor ezt a karaktert is tegyük egyszeres idézőjelek közé.
- Definiálhatunk hosszú sztringeket is több sorban speciális sztring fejlécekkel (> és |) plusz

további sorbehúzással.

- Ha tömb értéket definiálunk, akkor szögletes zárójelet kell használunk, vagy használhatjuk a kiterjesztett szintaktikát kötőjelekkel. Pl:

```
players: [ Mark McGwire, Sammy Sosa, Ken Griffey ]
```

vagy

```
players:
 - Mark McGwire
 - Sammy Sosa
 - Ken Griffey
```

- Ha asszociatív tömb értéket akarunk definiálni, akkor az elemeket kapcsos zárójelek közé kell zárnunk, és mindig be kell szúrni egy szóközt a kulcs és az érték pár közé. Pl:

```
mail: { webmaster: webmaster@example.com, contact: contact@example.com }
de használhatjuk a kiterjesztett szintaktikát is a következőképpen:
mail:
  webmaster: webmaster@example.com
  contact:   contact@example.com
```

- Logikai értek megadása esetén pozitív érték az *on*, *1*, vagy *true*, negatív érték az *off*, *0*, vagy *false*. Pl:

```
true_values:   [ on, 1, true ]
false_values:  [ off, 0, false ]
```

- Megjegyzéseket # megadása után írhatunk a sor végéig.

A symfony a paraméter definíciókat néha kategóriákba csoportosítja. Egy adott kategória minden beállítása a behúzott kategória fejléc alatt jelenik meg. Hosszú kulcs:érték pár listák kategóriákba való csoportosítása még olvashatóbbá teszik a konfigurációkat. A kategória fejlécek ponttal (.) kezdődnek. Pl:

```
all:
  .general:
    tax:        19.6
  mail:
    webmaster:  webmaster@example.com
```

A fenti példában a *mail* egy kulcs, a *general* pedig csak egy kategória fejléc. Minden úgy működik, mintha a kategória fejléc nem is létezne. A *tax* paraméter tulajdonképpen egy közvetlen gyermeke az *all* kulcsnak. A kategóriák használata segít a symfony-nak az *all* kulcs alá kerülő tömbök kezelésében.

A YAML tulajdonképpen csak egy felület (interface), aminek a segítségével beállításokat

definiálhatunk. Ezek a YAML fájlok PHP kóddá konvertálódnak.

5.3. Projekt beállítása

Alapértelmezés szerint van néhány projekt konfigurációs fájl. A *myprojekt/config/* könyvtárban az alábbi fájlok találhatók:

- *ProjectConfiguration.class.php*: Ez a legeslegelső fájl, ami beillesztésre kerül minden egyes kérésnél vagy parancsnál. Tartalmazza az útvonalat a keretrendszer fájljaihoz.
- *databases.yml*: Ebben a fájlban definiáljuk a hozzáférési és kapcsolódási információkat az adatbázishoz (host, felhasználó, jelszó, adatbázisnév, stb.). Ez a fájl felülírható alkalmazásszinten.
- *properties.ini*: A parancssor által használt néhány paramétert tartalmaz, beleértve a projekt nevét, és távoli szerverekhez való kapcsolódási beállításokat.
- *rsync_exclude.txt*: Ez a fájl határozza meg, hogy mely könyvtárak lesznek kizárva a szerverek közti szinkronizációból.
- *schema.yml* és *propel.ini*: Ezek a Propel által használt adatelérési konfigurációs fájlok. Itt állítjuk be, hogy a Propel könyvtárak együtt tudjanak dolgozni a symfony osztályokkal és a projektünk adataival. A *schema.yml* a projekt relációs adatmodelljének reprezentációját tartalmazza. A *propel.ini* fájl automatikusan generálódik, valószínűleg nem lesz szükségünk a módosítására.

Ezeket a fájlokat leginkább vagy külső komponensek használják, vagy parancssori argumentumként használatosak, vagy fel kell dolgozni őket, még mielőtt bármilyen YAML elemzőt betöltene a keretrendszer. Ezért van az, hogy némelyik nem YAML formátumot használ.

5.4. Alkalmazás beállítása

A konfiguráció legfontosabb része az alkalmazás konfigurációja. Ez egyrészt jelenti a *web/* könyvtárban található front kontroller, másrészt a *config/* és *i18n/* könyvtárakban található YAML fájlok, továbbá a keretrendszer rejtett, de további igen hasznos fájljainak a konfigurációit.

5.5. Front kontroller beállítása

Az első alkalmazás beállítás tehát a front kontrollerben található, ez az első szkript, ami a kérés során lefut. Vizsgáljuk meg az alapértelmezett *web/index.php* fájlt.

```
<?php
require_once(dirname(__FILE__).'/../config/ProjectConfiguration.class.php');
```

```
$configuration = ProjectConfiguration::getApplicationConfiguration('frontend',
'prod', false);
sfContext::createInstance($configuration)->dispatch();
```

Miután definiáltuk az alkalmazás és a környezet nevét (frontend és prod), az alkalmazás beállításai meghívódnak, mielőtt a programkörnyezet elkészül. A konfigurációs osztályban elérhető néhány hasznos metódus:

- $configuration->getRootDir(): A projekt gyökérkönyvtárát adja vissza a metódus.
- $configuration->getApplication(): Az alkalmazás nevével tér vissza az aktuális projektben.
- $configuration->getEnvironment(): A környezet neve. (prod, dev, vagy más általunk definiált projekt specifikus környezet) Ez határozza meg, hogy melyik konfigurációs beállítást használjuk.
- $configuration->isDebug(): A hibakereső mód működését adja meg.

Ha ezen értékek valamelyikét meg akarjuk változtatni, akkor egy újabb front kontrollerre lesz szükségünk. A legfontosabb alkalmazás konfigurációs fájlok a *myproject/apps/frontend/config/* könyvtárban találhatók. Ezek a következők:

- app.yml: Ez a fájl alkalmazás specifikus beállításokat tartalmaz, amik globális változókat jelentenek, és nem szükségesek adatbázisban tárolni őket. Adókulcsokat, viteldíjakat és email címeket tárolunk itt általában. Alapértelmezés szerint üres.
- frontendConfiguration.class.php: Ez az osztály tölti be az alkalmazást, azaz megcsinálja az összes alap inicializálást, ami az alkalmazás indításához szükséges. Itt tudjuk testre szabni a könyvtár struktúrát, vagy hozzáadni alkalmazás specifikus konstansokat. A *ProjectConfiguration* osztály leszármazottja.
- factories.yml: A symfony maga definiálja a nézet (view), a kérés (request), a válasz (response), a munkafolyamat (session), stb. osztályait. Ha a saját osztályainkat akarjuk használni, akkor ezt itt tudjuk megváltoztatni.
- filters.yml : A szűrök minden kérés során lefuttatott kódrészletek. Ebben a fájlban definiáljuk, hogy mely szűrők legyenek feldolgozva, és ezt bármelyik modul felülírhatja.
- routing.yml: Útvonalválasztási szabályokat tárolunk ebben a fájlban, amik az olvashatatlan URL-eket könnyen olvashatóvá konvertálják. Új alkalmazásoknál van néhány ilyen alapértelmezett szabály.
- settings.yml: Egy symfony alkalmazás fő beállításai ebben a fájlban találhatók. Itt adhatjuk meg, hogy az alkalmazásunk nemzetközi legyen, megadhatjuk az alapértelmezett nyelvet, az időtúllépést, és hogy a gyorsítótár be legyen-e kapcsolva. Egyelten sor megváltoztatásával leállíthatjuk az alkalmazásunkat azért, hogy valamelyik komponenst karbantartsuk vagy frissítsük.

- view.yml: Az alapértelmezett nézet struktúrája ebben a fájlban van beállítva. (elrendezés neve, alapértelmezett stíluslapok, beillesztendő JavaScript fájlok, alapértelmezett tartalom típusok, stb.) Ezen beállításokat bármelyik modul felülírhatja.

5.6. További alkalmazás beállítások

A symfony telepítő könyvtárában (*$sf_symfony_lib_dir/config/config/*) további konfigurációs fájlok találhatók, amik nem jelennek meg az alkalmazásunk konfigurációs könyvtárában. Az itt definiált beállítások általánosak, ritkán kell őket módosítani, és globálisak az összes projektre. Ha azonban mégis módosítani kell őket, akkor elég ha létrehozunk egy üres fájlt ugyanazzal a névvel a *myproject/apps/frontend/config/* könyvtárban, és felülírjuk a módosítani kívánt beállításokat. Az alkalmazásban definiált beállítások precedenciája erősebb a keretrendszerben definiált beállításoknál. Az alábbi konfigurációs fájlok találhatók a Symfony *config/* könyvtárában:

- autoload.yml: Az automatikus betöltés funkció beállításait tartalmazza. Ez a funkció mentesít minket az alól, hogy beillesszünk szokásos osztályokat a kódunkba, ha azok különböző könyvtárakban foglalnak helyet.
- core_compile.yml és bootstrap_compile.yml: Ezek osztálylisták, amik meghívódnak egy alkalmazás indítása és egy kérés feldolgozása során. Ezek az osztályok egy megjegyzésektől mentes PHP fájlba fűződnek össze, ami felgyorsítja és minimalizálja a fájlelérési műveleteket. (minden kérés során egy fájl töltődik be több helyett) Ez különösen hasznos ha nem használunk PHP gyorsítót.
- config_handlers.yml: Itt tudjuk hozzáadni és módosítani a különböző kezelőket, amik feldolgozzák a konfigurációs fájlokat.

5.7. Modul beállítások

Alapértelmezés szerint a moduloknak nincsenek sajátos beállításaik, de ha szükséges, akkor néhány alkalmazás szintű beállítás felülírható egy adott modul esetén. Például beilleszthetünk egy JavaScriptet egy modul összes művelete számára. Új paraméterek hozzáadását is választhatjuk egy adott modulra korlátozva, hogy megőrizzük az egységbe zárást. Mint ahogy azt már sejteni lehet, a modulok konfigurációs fájljai a *myproject/apps/frontend/modules/mymodule/config/* könyvtárban találhatók, és ezek az alábbiak:

- generator.yml: Adatbázis tábla által generált modulok számára ez a fájl definiálja, hogy a felület hogyan jelenítse meg a sorokat és az oszlopokat, és hogy a felhasználó milyen interakciót használjon (szűrés, rendezés, gombok, stb.)

- module.yml: Ez a fájl tartalmazza a szokásos paramétereket egy modul számára (ekvivalens az app.yml-el csak modul szinten) és a művelet (action) beállításokat.
- security.yml: Ez a fájl műveletek hozzáférési korlátozásait tartalmazza. Itt tudjuk beállítani, hogy egy oldalt csak regisztrált felhasználók, vagy regisztrált felhasználók bizonyos engedélyekkel rendelkező része tudja megnézni.
- view.yml: Egy modul egy vagy több műveletének megjelenítésére tartalmaz beállításokat. Felülírja az alkalmazás view.yml fájlját.

A legtöbb modul konfigurációs fájl felajánlja a paraméterek definiálásának lehetőségét egy modul összes műveletének összes nézetére.

5.8. Környezetek

Az alkalmazás fejlesztése során valószínűleg párhuzamosan kell majd néhány beállítást elvégezni. Erre találták ki a párhuzamos konfigurálási lehetőséget. A symfony erre különböző környezeteket ajánl, az alkalmazás különféle környezetekben futhat. A különböző környezetek ugyanazon a PHP kódon osztoznak (eltekintve a front kontrollertől), de teljesen más beállításokat tartalmazhatnak. Minden alkalmazás számára a symfony három környezetet biztosít: éles környezet (prod: production environment), teszt környezet (test), és fejlesztési környezet (dev: development). Természetesen annyi további saját környezetet adhatunk az alkalmazásainkhoz amennyit csak szeretnénk.

Alapjában véve a környezetek és a beállítások szinonimák. Például a teszt környezet riasztásokat és hibákat naplóz, míg a prod környezet csak hibákat. A gyorsítótár gyakran ki van kapcsolva a fejlesztési környezetben, de a teszt és az éles környezetben aktiválva van. A fejlesztési és teszt környezetben lehet, hogy szükségünk van tesztadatokra, amiket adatbázisban tárolunk, az éles környezetben viszont nem. Tehát az adatbázis konfiguráció különböző a két környezet között. Az összes környezet létezhet együtt ugyanazon a gépen, de a szerver általában csak az éles környezetet tartalmazza.

A fejlesztési környezetben a naplózási és hibakeresési beállítások engedélyezve vannak, mivel a karbantartás sokkal fontosabb mint a teljesítmény. Ezzel szemben az éles környezet alapértelmezett módon teljesítményre van optimalizálva, így az éles beállítások sok funkciót kikapcsolnak. Egy jó módszer tehát, hogy a fejlesztési környezetet használjuk addig, amíg nem vagyunk elégedettek a funkciókkal amiken dolgozunk, aztán kapcsoljunk át éles környezetre, hogy teszteljük a sebességet.

A teszt környezet a fejlesztési és éles környezettől más módon különbözik. Ezzel a környezettel kizárólag parancssoron keresztül kommunikálunk, funkcionális tesztelést és kötegelt szkript futtatást végzünk. Következésképp a teszt környezet közel van az éles környezethez, de nem érhető

el webböngészőn keresztül. Sütik és más HTTP specifikus komponensek szimulálására használható. Ha meg akarjuk változtatni a környezetet, amiben az alkalmazásunkat böngésszük, csak meg kell változtatnunk a front kontrollert. Ideáig csak a fejlesztési környezetet láttuk, mert az eddig használt példa URL-ek a fejlesztési környezet front kontrollerét hívták meg.

```
http://localhost/frontend_dev.php/mymodule/index
```

Ha látni akarjuk, hogy hogyan reagál az alkalmazásunk éles környezetben, akkor hívjuk meg az éles környezet front kontrollerét:

```
http://localhost/index.php/mymodule/index
```

Ha egy új környezetet akarunk hozzáadni az alkalmazásunkhoz, akkor nem szükséges egy új könyvtárat létrehozni, hanem egyszerűen létrehozunk egy új front kontrollert, és ebben megváltoztatjuk a környezet nevét. Ez a környezet örökölni fogja az összes alapértelmezett beállítást, amivel az összes többi környezet rendelkezik.

A beállítások elérése a kódból: A konfigurációs fájlok PHP kóddá konvertálódnak, és sok beállítást, amit tartalmaznak, a keretrendszer további beavatkozás nélkül használja. Azonban előfordulhat olyan eset is, hogy el szeretnénk érni bizonyos beállításokat a kódunkból. A *settings.yml*, *app.yml*, és *module.yml* fájlokban definiált beállítások egy *sfConfig* nevezetű speciális osztályon keresztül elérhetők. Pl:

```
parameter = sfConfig::get('param_name', $default_value);
```

Nem csak lekérdezhetünk, hanem felül is írhatunk egy beállítást a PHP kódunkból. Pl:

```
sfConfig::set('param_name', $value);
```

A paraméter neve néhány elemből tevődik össze, amiket aláhúzással választunk el, az alábbi szabály szerint:

- Az előtag a konfigurációs fájl neve lesz (sf_ a settings.yml-ből, app_ az app.yml-ből, mod_ a mod.yml-ből).
- A szülő kulcs kisbetűvel.
- A kulcs neve szintén kisbetűvel.

A környezet megadása nem szükséges, mivel a PHP csak azokhoz az értékekhez fér hozzá, amilyen környezetben futtatjuk.

6. A kontroller réteg

A symfony-ban a kontroller réteg, ami az üzleti logika és a megjelenítés közti kapcsolatot

reprezentáló kódot tartalmazza, különböző komponensekre van osztva az alábbi célok szerint:

- A front kontroller az alkalmazás egyedüli belépési pontja. Ez tölti be a beállításokat, és meghatározza, hogy melyik művelet legyen végrehajtva.
- A műveletek tartalmazzák a végrehajtandó logikát. Ellenőrzik a kérések sértetlenségét, és előkészítik az adatokat a megjelenítési réteg számára.
- A kérés, válasz, és munkamenet folyamatok hozzáférést biztosítanak a kérés paraméterekhez, a válasz fejlécekhez, valamit a perzisztens felhasználói adatokhoz. Ezeket gyakran használjuk a kontroller rétegben.
- A szűrők olyan kódrészletek, amik minden egyes kérés során lefutnak a művelet előtt vagy után. Például biztonsági és ellenőrzési szűrőket a webalkalmazások rendszerint használják. A keretrendszert kibővíthetjük saját szűrőkkel is.

Minden egyes webes kérelmet, egy egyszerű front kontroller kezel, ami az egyedüli belépési pontja az egész alkalmazásnak egy megadott környezetben. Amikor a front kontroller fogad egy kérelmet, akkor az útvonalválasztási rendszert használja, illeszti a művelet és a modul nevét a felhasználó által begépelt URL-el. Például az alábbi URL meghívja az *index.php* szkriptet, és a *mymodul myAction* művelete értelmezi azt:

```
http://localhost/index.php/mymodule/myAction
```

A front kontroller feladata részletesen:

1. Betölti a projekt konfigurációs osztályt, és a symfony könyvtárakat.
2. Létrehoz egy konfigurációs alkalmazást és egy symfony programkörnyezetet.
3. Betölti és elindítja a symfony alapvető osztályait.
4. Betölti a konfigurációt.
5. Dekódolja a kapott URL-t és paramétereket, és meghatározza a végrehajtandó műveletet.
6. Ha a művelet nem létezik, akkor 404-es hibakódra irányít.
7. Aktiválja a szűrőket (pl: ha a kérés hitelesítést igényel).
8. Lefuttatja a szűrőket első lépésben.
9. Végrehajtja a műveletet és megformázza a nézetet.
10. Lefuttatja a szűrőket második lépésben.
11. Kiírja a választ.

Műveletek biztonsága: A műveltek végrehajtását bizonyos jogosultságokkal rendelkező felhasználókra lehet korlátozni. Vannak beépített eszközök a symfony-ban, amik biztonságos alkalmazások létrehozását teszik lehetővé, ahol a felhasználóknak hitelesíteniük kell magukat, mielőtt az alkalmazás bizonyos funkcióit vagy részeit el akarják érni. Az alkalmazás biztonságossá

tételéhez két lépésre van szükség: minden művelet számára deklarálni kell a biztonsági követelményeket, valamint a felhasználóknak jogokat kell osztani, amivel elérhetik ezeket a biztonságos műveleteket. Minden egyes művelet esetén, mielőtt végrehajtódna, egy speciális szűrő ellenőrzi, hogy az aktuális felhasználó rendelkezik-e a jogokkal, hogy elérje a kért műveletet. Ha egy művelethez korlátozott hozzáférést szeretnénk biztosítani, akkor csak egy *security.yml* nevezetű konfigurációs fájlt kell szerkesztenünk a modul *config/* könyvtárában. Itt meghatározhatjuk a biztonsági követelményeket, amiket a felhasználóknak teljesíteniük kell egyes műveletek, vagy akár minden művelet esetén. Alapértelmezés szerint a műveletek nem biztonságosak, ami azt jelenti, hogy ha nincs *security.yml* fájl, vagy van, de nincs benne egy adott műveletre biztonsági előírás, akkor a műveletek mindenki számára elérhetők. Ha van *security.yml*, akkor a symfony megkeresi benne a kért művelet, és ha megtalálja, akkor ellenőrzi, hogy a biztonsági követelmények teljesítve vannak-e.

7. A nézet réteg

A nézet az eredmény megjelenítéséért felelős, ami egy adott művelettel van összefüggésben. A symfony-ban a nézet több részből áll, minden részt úgy tervezték, hogy könnyen kezelhető legyen annak a személynek, aki használni fogja.

- Web dizájnerek általában a sablonokon dolgoznak (az aktuális művelet adatainak megjelenítésén) és az elrendezésen (aminek a tartalma minden oldal számára azonos). Ezeket HTML-ben írják, amik tartalmaznak kis PHP kódokat, amiket általában helpereknek hívunk.
- Az újrafelhasználhatóság kedvéért a fejlesztők a sablon kódtöredékeket általában komponensekbe csomagolják. A web dizájnerek szintén dolgozhatnak ezeken a sablon töredékeken.
- A fejlesztők a nézet YAML konfigurációs fájlra koncentrálnak (meghatározzák a beállításokat a válasz és más kapcsolódó elemre) és a válasz objektumra. A változók kezelése a sablonokban odafigyelést igényel, nem szabad figyelmen kívül hagyni névütközéseket, ezért a felhasználói adatok biztonságos rögzítéséhez egy hatékony adatmentési technika szükséges.

Mint ahogyan azt már korábban láthattunk, a sablonokban egy alternatív PHP szintaxist érdemes használni, hogy a kód nem PHP fejlesztők számára is olvasható legyen. PHP kód írását minimalizálni kell a sablonokban, mert ezek az egyetlen olyan típusú fájlok, amik az alkalmazás felhasználói felületére vonatkoznak, és általában a készítésüket és karbantartásukat egy másik csapat végzi, akik a megjelenítésért és nem az üzleti logikáért felelősek. Az üzleti logika műveletbe

zárása megkönnyíti a sablonok írását egy egyszerű művelet számára, így elkerülhet a kódismétlés.

A helperek PHP függvények, amik HTML kóddal térnek vissza és sablonokban használhatjuk őket. Néha a helperekkel csak időt spórolunk azáltal, hogy gyakran használt kódrészleteket becsomagolunk a sablonokban. Legtöbbször azonban a helperek intelligensek, és megmentenek minket a hosszú és bonyolult kódolástól. Megkönnyítik a sablonírás folyamatát és a lehető legjobb kódot állítják elő a teljesítményt és a hozzáférhetőséget illetően.

Helper deklaráció: A helper definíciókat tartalmazó symfony fájlok nem töltődnek be automatikusan (mivel nem osztályokat, hanem függvényeket tartalmaznak). A helperek cél szerint vannak csoportosítva. Például az összes szöveggel foglalkozó helper függvény a *textHelper.php* fájlban van, amit Text helper csoportnak hívunk. Tehát ha használni akarunk egy helpert egy sablonban, akkor a kapcsolódó helper csoportot ezt megelőzően be kell töltenünk a *use_helper()* függvénnyel. Ha több helper csoportot akarunk használni, akkor több argumentumot is megadhatunk a *use_helper()* függvénynek. Néhány helper alapértelmezés szerint elérhető minden sablonban, anélkül hogy deklarálnánk őket. Ezek a következő helper csoportokba tartoznak: Helper, Tag, Url, Asset, Partial, Cache, Form.
A settings.yml fájlban beállítható azon helperek listája, amik alapértelmezés szerint betöltődnek minden sablon számára.

8. A model réteg

Az eddigiekben főleg arról volt szó, hogy hogyan készítsünk oldalakat, valamit hogy hogyan dolgozzunk fel kéréseket és válaszokat. Azonban egy webalkalmazás üzleti logikája az adatmodellen alapszik. A symfony alapértelmezett model komponense az objektum relációs leképezésen alapszik, amit a Propel projekt végez. Egy symfony alkalmazásban adatbázisban tárolt adatokat használunk, amiket objektumokon keresztül módosítunk, az adatbázist explicit módon sosem címezzük. Ez magas szintű absztrakciót és hordozhatóságot tesz lehetővé.

8.1 ORM (Object-Relational Mapping)

Manapság relációs adatbázisokat használunk, a PHP5 és a symfony pedig objektumorientált. Ahhoz, hogy a leghatékonyabban elérjük az adatokat egy objektumorientált környezetben, szükségünk van egy interfészre, ami az objektum logikát relációs logikává konvertálja. Ezt az interfészt objektum-relációs leképezésnek nevezzük (ORM). Az ORM objektumokból áll, amik

elérést biztosítanak az adatokhoz, és az üzleti szabályokat magukban tartják.
Egy ORM legfőbb előnye az újrafelhasználhatóság, egy objektum metódusait az alkalmazás különböző részéből meghívhatjuk akár még különböző alkalmazásokból is.
Ha objektumokat használunk rekordok helyett, valamit osztályokat táblák helyett, az további előnyökkel is bír: Lehetőségünk van újabb lekérdező metódusokat adni az objektumokhoz, amik nem feltétlenül egyeznek egy tábla oszlopával. Például ha van egy ügyfél táblánk, aminek van két oszlopa: vezetéknév és keresztnév, lehet hogy szükségünk van az ügyfél teljes nevére. Az objektumorientált világban egyszerű hozzáadni egy új lekérdező metódust az osztályhoz. Az alkalmazás szempontjából nincs különbség az ügyfél osztály vezetéknév, keresztnév, és tejes név attribútumok közt. Csak az osztály maga tudja meghatározni, hogy mely attribútumok felelnek meg az adatbázis egy oszlopának. Tegyük fel, hogy van egy bevásárlókosár osztályunk, amiben vannak tételek (objektumok). Ha a bevásárlókosár teljes tartalmának értékre vagyunk kíváncsiak, akkor a szokásos metódust kell megírnunk, ami beágyazza az aktuális számítást:

```
public function getTotal(){
  $total = 0;
  foreach ($this->getItems() as $item)  {
    $total += $item->getPrice() * $item->getQuantity();
  }
  return $total;
}
```

Van egy másik fontos szempont, amit érdemes megfontolni, amikor adatelérési eljárásokat írunk: az adatbázis gyártók különböző SQL szintaktikákat használnak. Ha áttérünk egy másik adatbázis kezelő rendszerre, akkor át kell írni az SQL lekérdezések egy részét. Ha azonban adatbázis független szintaktikát használunk, és az aktuális lekérdezéseket egy harmadik komponensre bízzuk, akkor mindenféle gond nélkül áttérhetünk egy másik adatbázis rendszerre. Ez az adatbázis-absztrakciós réteg célja és lényege. Ez arra kényszerít minket, hogy speciális szintaktikát használjunk a lekérdezések során, viszont elvégzi a „piszkos” munkát helyettünk, azaz összhangba hozza a lekérdezéseket a az adatbázis rendszerek sajátosságaival, valamint optimalizálja az SQL kódot.
Az absztrakció szint legfőbb előnye a hordozhatóság, mert lehetővé teszi másik adatbázis rendszerre történő átváltást, akár a projekt fejlesztésének kellős közepén. Lehet, hogy egy gyors prototípust kell írnunk egy alkalmazáshoz, de az ügyfél még nem döntötte el, hogy melyik adatbázis kezelő rendszer illeszkedik legjobban az igényeihez. Például elkezdhetjük az alkalmazás fejlesztését SQLite adatbázis kezelő rendszerrel, és áttérhetünk MySql, PostgreSQL, vagy Oracle rendszer használatára, ha az ügyfél úgy határoz. Mindössze egy sort kell megváltoztatni egy konfigurációs

fájlban és már működik is.
Symfony a Propelt használja az objektum-relációs leképezésekhez, a Propel pedig a Creole-t az adatbázis-absztrakcióhoz. Ez két külső komponens, mindkettőt a Propel csapat fejleszti, zökkenőmentesen integrálva vannak a symfony-ban,és a keretrendszer részeként tekinthetjük őket.

8.2. Symfony adatbázis-sémája

Ahhoz hogy létrehozzuk a symfony által használt adat objektum modellt, le kell fordítanunk az adatbázisunk relációs modelljét egy objektum adat modellre. Az ORM-nek szüksége van a relációs modell leírására ahhoz, hogy megcsinálja a leképezést, ezt hívjuk sémának. A sémában definiáljuk a táblákat, oszlopokat és a táblák közti kapcsolatokat.
A sémák leírására a symfony szintén YAML formátumot használ. A *schema.yml* fájlt a *myprojekt/config/* könyvtárban helyezzük el. Nézzünk egy példa adatbázist. Tegyük fel, hogy van egy blog adatbázisunk, és ebben van két táblánk: blog_article és blog_comment. Ekkor a *schema.yml* fájl a következőképpen néz ki:

```
propel:
  blog_article:
    _attributes: { phpName: Article }
    id:
    title:       varchar(255)
    content:     longvarchar
    created_at:
  blog_comment:
    _attributes: { phpName: Comment }
    id:
    article_id:
    author:      varchar(255)
    content:     longvarchar
    created_at:
```

Az adatbázis neve (blog) nem jelenik meg a *schema.yml* fájlban. Ehelyett az adatbázist a kapcsolat neve alatt (a példában propel) írjuk le. Ez azért van, mert a kapcsolat beállítások függnek attól a környezettől, amiben az alkalmazásunk fut. Például ha az alkalmazásunkat a fejlesztői környezetben futtatjuk, akkor fejlesztői adatbázist érünk el (blog_dev), de ugyanazt a sémát használjuk, mint az éles környezetben hasznát adatbázis esetén. A kapcsolat beállításai a *databases.yml* fájlban találhatók. A séma nem tartalmaz részletes kapcsolat beállításokat, csak a kapcsolat nevét, ezáltal megőrzi az adatbázis-absztrakciót.

8.3. Alap séma szintaxis

A *schema.yml* fájlban az első kulcs reprezentálja a kapcsolat nevét, ezen kívül tartalmazhat táblaneveket, és a táblanévhez adjuk meg az oszlopnevek halmazát. A YAML szintaktika értelmében a kulcsok kettősponttal végződnek, és a struktúrát sorbehúzások mutatják (egy vagy több szóköz, nem pedig tabulátor). A táblának lehetnek speciális attribútumai (pl.: az osztály neve, ami generálódik). Ha nem adunk meg PHP osztálynevet a táblához, akkor a tábla nevéből a symfony automatikusan generálja azt. A tábla oszlopokat tartalmaz. Az oszlop típusát háromféleképpen definiálhatjuk:

- Ha nem definiálunk semmit, akkor a symfony az oszlop neve alapján megbecsüli, hogy melyik típus lenne a legmegfelelőbb. Például az id oszlop típusát nem kell definiálnunk, a symfony egész típusú, elsődleges kulcs megszorítással rendelkező, automatikusan növekvő oszlopként fogja létrehozni. A blog_comment tábla article_id attribútuma külső kulcsként lesz értelmezve a blog_article táblához (_id-vel végződő oszlopok külső kulcsok lesznek, a kapcsolódó tábla automatikusan meghatározódik az oszlop nevének első részéből). A created_at nevű oszlopok automatikusan timestamp típusúak lesznek. Ezen oszlopok számára nem szükségesek típust meghatároznunk. Ez az egyik ok amiért a *schema.yml* fájlt könnyen megírhatjuk.
- Ha csak egy attribútumot definiálunk, akkor az lesz az oszlop típusa. A symfony megérti az általános oszlop típusokat mint például: boolean, integer, float, date, varchar(size), longvarchar (ami például text típusúvá konvertálódik MySql esetén), stb. 256 karakternél hosszabb szöveges tartalmakhoz longvarchar típust kell használnunk, aminek nincs mérete (de a 65Kbyte-ot nem lépheti túl a MySql-ben). A date és a timestamp típusok a Unix dátumainak szokásos korlátozásaival rendelkeznek, és 1970. 01. 01. előtti dátumot nem lehet beállítani nekik. Ha régebbi dátumot kell beállítani (például születési dátumokat), akkor „Unix előtti” dátumformátumot használhatunk a bu_date és bu_timestamp típusokkal.
- Ha más oszlop attribútumot szeretnénk definiálni, akkor azt megtehetjük kulcs:érték pár megadásával.

8.4. Model osztályok

A sémából az ORM réteg model osztályokat készít. Időspórolás céljából ezeket az osztályokat a *symfony propel:build-model* paranccsal parancssorból generáltathatjuk. (Miután létrehoztuk a modelt ne felejtsük el kiadni a *symfony cache:clear* parancsot, hogy a symfony megtalálja az

újonnan létrehozott osztályokat.) A *propel:build-mode*l parancs hatására a projektünk *lib/model/om/* könyvtárában létrejönnek a következő osztályok: *BaseArticle.php, BaseArticlePeer.php, BaseComment.php, BaseCommentPeer.php*, továbbá a *lib/model/* könyvtárban a *Article.php, ArticlePeer.php, Comment.php, CommentPeer.php* osztályok. Csak két táblát definiáltunk és nyolc fájlt kaptunk, ez megérdemel némi magyarázatot. Felmerül a kérdés, hogy mért tartunk két adat objektum model verziót két különböző könyvtárban? Valószínűleg további metódusokat és tulajdonságokat kell majd hozzáadni a model objektumokhoz (gondoljunk az ügyfél teljes nevét lekérdező metódusra). De a projekt fejlesztése során valószínűleg új táblát és oszlopokat is hozzáadunk majd a sémához. Minden egyes alkalommal, amikor módosítjuk a *schema.yml* fájlt, újra kell generáltatnunk az objektum model osztályainkat a *propel:build-model* parancs segítségével. Ha a saját metódusainkat a generált osztályokba tennénk, akkor minden újragenerálásnál törlődnének. Másrészről az egyedi objektum osztályaink, amiket a *lib/model/* könyvtárban tartunk, örökölnek a Base osztályoktól. Amikor a *propel:build-model* parancsot meghívjuk a létező modellünkre, akkor ezek az osztályok nem módosulnak. Tehát itt tudunk egyedi metódusokat hozzáadni. Ez a mechanizmus, hogy egyedi osztályokat hozunk létre az alap osztályok kiterjesztésével, lehetővé teszi, hogy elkezdhessük a kódolást anélkül, hogy ismernénk az adatbázisunk végleges relációs modelljét. A kapcsolódó fájlstruktúra testre szabhatóvá és fejlődőképessé teszi a modellünket.

8.5. Objektum és társ osztályok

Az Article és Comment osztályok objektum osztályok, amik egy rekordot reprezentálnak az adatbázisban. Egy rekord és a kapcsolódó rekordok oszlopaihoz adnak hozzáférést. Ez azt jelenti, hogy egy cikk címét megtudhatjuk, ha meghívjuk egy Article típusú objektum *getTitle()* nevű metódusát. Az ArticlePeer és CommentPeer osztályok társ osztályok, azaz statikus metódusokat tartalmazó osztályok, amik a táblákon végeznek műveletet. Ezek az osztályok gondoskodnak arról, hogy rekordokat nyerjünk ki a táblákból. A metódusaik általában egy objektummal vagy objektumok gyűjteményével térnek vissza.

Adatelérés: Symfony-ban az adatokat objektumokon keresztül érjük el. Ha a relációs modellhez vagyunk szokva, és SQL lekérdezéseket használtunk az adatok eléréséhez és módosításához, akkor az objektum modell metódusai bonyolultnak tűnhetnek. Egy tábla egy osztálynak, egy sor (rekord) egy objektumnak, egy oszlop (mező) pedig egy tulajdonságnak felel meg az objektumorientált világban.

- Oszlopértékek kinyerése: Amikor a symfony felépíti a modellt, egy alap osztályt hoz létre

minden a *schema.yml* fájlban definiált táblához. Ezen osztályok mindegyike tartalmaz egy alapértelmezett konstruktort, lekérdező és beállító metódusokat, amik az oszlop definíciókon alapulnak. A *new*, *getXXX()*, és *setXXX()* metódusok segítségével objektumokat hozhatunk létre és elérhetjük az objektumok tulajdonságait.

- Kapcsolódó rekordok kinyerése: Az article_id oszlop a blog_comment táblában implicit módon definiál egy külső kulcsot a a blog_article táblához. Mindegyik megjegyzés egy cikkhez kapcsolódik, és minden cikknek több megjegyzése lehet. A generált osztályok öt metódust tartalmaznak, amik ezt a kapcsolatot átviszik objektumorientált világba:
  - $comment->getArticle(): A kapcsolódó cikk objektum kinyerése.
  - $comment->getArticleId(): A kapcsolódó cikk objektum id-jának kinyerése.
  - $comment->setArticle($article): A kapcsolódó cikk objektum meghatározása.
  - $comment->setArticleId($id): A kapcsolódó cikk objektum meghatározása id alapján.
  - $article->getComments(): A kapcsolódó megjegyzés objektum kinyerése.
- Adatok mentése és törlése: A new konstruktor meghívásával létrehozunk egy új objektumot, de nem egy új rekordot a blog_article táblában. Az objektumon végzett módosítás szintén nincs hatással az adatbázisra. Ha el akarjuk menteni az adatbázisba, akkor meg kell hívnunk az objektum *save()* metódusát. Az ORM elég intelligens ahhoz, hogy felismerje az objektumok közti kapcsolatokat, tehát egy cikk objektum mentése maga után vonja a kapcsolódó megjegyzés objektum mentését. Még azt is tudja, hogy az elmentett objektumnak van-e már létező példánya az adatbázisban, ennek megfelelően a *save()* metódus meghívása néha SQL INSERT néha SQL UPDATE műveletet jelent. Az elsődleges kulcs automatikusan beállítódik a *save()* metódus hívásakor, tehát mentés után az új elsődleges kulcsot a *getId()* metódussal megkaphatjuk. Ha törölni szeretnénk egy sort, akkor az adott objektum *delete()* metódusát kell meghívnunk.
- Sorok kinyerése elsődleges kulcs alapján: Ha ismerjük egy adott rekord elsődleges kulcsát, akkor használhatjuk a társ osztály *retriveByPk()* osztályszintű metódusát a kívánt objektum kinyeréséhez. Mivel elsődleges kulcsot használtunk, tudjuk hogy csak egy rekorddal tér vissza a metódus. Bizonyos esetekben az elsődleges kulcs egynél több oszlopból is állhat. Ilyen esetben a *retriveByPk()* metódusnak annyi paramétert kell megadni, ahány oszlopból áll az elsődleges kulcs.
- Sorok kinyerése feltétel alapján: Ha egynél több sort szeretnénk kapni, akkor a társ osztály *doSelect()* metódusát kell meghívnunk. Például ha a cikk osztály objektumaira vagyunk kíváncsiak, akkor az ArticlePeer::doSelect() metódust hívjuk meg. A doSelect() első

paramétere a Criteria osztály egy objektuma, ami egy egyszerű lekérdezés definiáló osztály, és ami az adatbázis-absztrakció megőrzése miatt SQL mentes. Egy összetett lekérdezéshez szükségünk van a WHERE, ORDER BY, GROUP BY, és más SQL utasítással ekvivalens eszközökre. A Criteria objektumnak vannak metódusai és paraméterei ezen eszközökhöz.
A társ osztályok gondoskodnak *doDelete(), doInsert(),* és *doUpdate()* metódusokról, amik szintén egy Criteria objektumot várnak paraméterül. Ezen metódusok a DELETE, INSERT, és UPDATE SQL utasításokra vezethetők vissza.

8.6. Adatbázis kapcsolatok

Az adatmodell független a használt adatbázistól, de választanunk kell egy adatbázist. A symfony által igényelt minimum információ ahhoz, hogy kéréseket tudjunk küldeni a projekt által használt adatbázisnak, az adatbázis neve, bejelentkezési adatok, és az adatbázis típusa. Ezeket a kapcsolat beállításokat a *configure:database* paranccsal tudjuk beállítani. Pl:

```
symfony configure:database "mysql://login:passwd@localhost/blog"
```

A kapcsolat beállítások környezet függőek. Különböző beállításokat definiálhatunk a *prod, dev,* és *test* környezetnek, vagy az alkalmazásunk bármely más környezetének, ha használjuk az előző parancshoz az *env* opciót:

```
symfony --env=prod configure:database "mysql://login:passwd@localhost/blog"
```

Ez a beállítás alkalmazásonként felülírható. Ezeket a kapcsolat beállításokat manuálisan is elvégezhetjük a *databases.yml* fájlban, ami a *config/* könyvtárban található. A phptype paraméter a támogatott adatbázisrendszerek valamelyike lehet: mysql, mssql, pgsql, sqlite, oracle.

9. Űrlapok

Amikor sablonokat írunk, a fejlesztési idő nagy részét az űrlapoknak szenteljük. A symfony biztosít számunkra eszközöket, amivel az űrlapokkal kapcsolatos követelményeket teljesíteni tudjuk, mialatt felgyorsul a fejlesztés is.

- Az űrlap helperek segítségével gyorsan tudunk űrlap beviteli eszközöket készíteni a sablonokban, különösen összetett elemekhez, úgymint dátumokhoz, legördülő listákhoz, és gazdagon formázható szövegekhez.
- Ha egy űrlapot egy objektum tulajdonságainak szerkesztésére szánunk, akkor a

sablonkészítést tovább gyorsíthatjuk objektum űrlap helperek használatával.

- A YAML ellenőrző fájlok megkönnyítik az űrlap ellenőrzést.
- A symfony-ban vannak ellenőrző eszközök a szokásos műveletek elvégzésére, de könnyen hozzáadhatunk saját ellenőrzőt.

A sablonokban az űrlap elemek HTML tag-jeit gyakran vegyítik PHP kóddal. Symfony-ban az űrlap helperek egyszerűsítik ezt a feladatot azáltal, hogy elkerülik a nyitó *<?php echo* tag-ek ismétlődését *input* tag-ek belsejében. Egy űrlap létrehozásához a form_tag() helpert kell használunk, ami a megadott paraméter alapján értelmezi a műveletet. Egy második argumentummal további beállításokat adhatunk. Pl:

```
<?php echo form_tag('test/save') ?>
 => <form method="post" action="/path/to/save">

<?php echo form_tag('test/save','method=get multipart=true class=simpleForm') ?>
 => <form method="get" enctype="multipart/form-data"
class="simpleForm"action="/path/to/save">
```

Mivel nincs szükség záró űrlap helperre, ezért *</form>* tag-et használunk, még ha ez nem is néz ki valami szépen a forráskódunkban. Nézzük meg a szabványos űrlap helpereket és azok beállításait. Minden helper alatt láthatjuk a HTML megfelelőjét:

```
<?php echo input_tag('name', 'default value') ?>
 => <input type="text" name="name" id="name" value="default value" />

<?php echo input_tag('name', 'default value', 'maxlength=20') ?>
 => <input type="text" name="name" id="name" value="default value"
maxlength="20" />

<?php echo textarea_tag('name', 'default content', 'size=10x20') ?>
 => <textarea name="name" id="name" cols="10" rows="20">
      default content
    </textarea>

<?php echo checkbox_tag('single', 1, true) ?>
<?php echo checkbox_tag('driverslicense', 'B', false) ?>
 => <input type="checkbox" name="single" id="single" value="1" checked="checked"
/>
    <input type="checkbox" name="driverslicense" id="driverslicense" value="B"
/>
```

```
<?php echo radiobutton_tag('status[]', 'value1', true) ?>
<?php echo radiobutton_tag('status[]', 'value2', false) ?>
 => <input type="radio" name="status[]" id="status_value1" value="value1"
checked="checked" />
    <input type="radio" name="status[]" id="status_value2" value="value2" />

<?php echo select_tag('payment', options_for_select(array(
  'Visa',
  'Eurocard',
  'Mastercard'
), 0)) ?>
 => <select name="payment" id="payment">
      <option value="0" selected="selected">Visa</option>
      <option value="1">Eurocard</option>
      <option value="2">Mastercard</option>
    </select>

<?php echo select_tag('name', options_for_select(array(
  'Steve'  => 'Steve',
  'Bob'    => 'Bob',
  'Albert' => 'Albert',
  'Ian'    => 'Ian',
  'Buck'   => 'Buck'
), 'Ian')) ?>
 => <select name="name" id="name">
      <option value="Steve">Steve</option>
      <option value="Bob">Bob</option>
      <option value="Albert">Albert</option>
      <option value="Ian" selected="selected">Ian</option>
      <option value="Buck">Buck</option>
    </select>

<?php echo select_tag('payment', options_for_select(
  array('Visa' => 'Visa', 'Eurocard' => 'Eurocard', 'Mastercard' =>
'Mastercard'),
  array('Visa', 'Mastercard'),
), array('multiple' => true))) ?>
 => <select name="payment[]" id="payment" multiple="multiple">
      <option value="Visa" selected="selected">Visa</option>
      <option value="Eurocard">Eurocard</option>
      <option value="Mastercard">Mastercard</option>
    </select>
```

```
<?php echo select_tag('payment', options_for_select(
  array('Visa' => 'Visa', 'Eurocard' => 'Eurocard', 'Mastercard' =>
'Mastercard'),
  array('Visa', 'Mastercard')
), 'multiple=multiple') ?>
 => <select name="payment[]" id="payment" multiple="multiple">
      <option value="Visa" selected="selected">Visa</option>
      <option value="Eurocard">Eurocard</option>
      <option value="Mastercard" selected="selected">Mastercard</option>
    </select>

<?php echo input_file_tag('name') ?>
 => <input type="file" name="name" id="name" value="" />

<?php echo input_password_tag('name', 'value') ?>
 => <input type="password" name="name" id="name" value="value" />

<?php echo input_hidden_tag('name', 'value') ?>
 => <input type="hidden" name="name" id="name" value="value" />

<?php echo submit_tag('Save') ?>
 => <input type="submit" name="submit" value="Save" />

<?php echo submit_image_tag('submit_img') ?>
 => <input type="image" name="submit" src="/images/submit_img.png" />
```

Az űrlap által elküldött adatokat a kérés paraméterek (request parameters) segítségével érhetjük el. A műveletnek csak meg kell hívnia a *$request->getParameter($elementName)* függvényt, hogy megkapja a kívánt értéket. Egy jó módszer, hogy ugyanazt a műveletet használjuk az űrlap megjelenítéséhez és feldolgozásához.

10. A program leírása

A következőkben egy egyszerű program elkészítését mutatom be lépésről lépésre a symfony keretrendszer használatával. Töltsük le az internetről az EasyPHP szoftvercsomagot, ami egy ingyenes eszköz. Segítségével egyszerűen tudunk webalkalmazásokat készíteni, hiszen tartalmaz egy Apache webkiszolgálót, egy MySql adatbázis-kiszolgálót, PHP interpretert, és phpMyAdmin felületet. Azért preferáltam az EasyPHP szoftver letöltését, az Apache, MySql és PHP egyenként történő letöltésével szemben, mert az EasyPHP tartalmazza az összes olyan beállítást, ami ezen szoftverek összehangolásához szükséges. Az EasyPHP szoftver elindításával gyakorlatilag

elindítjuk az Apache és MySql szervereket, és máris készen állunk arra, hogy elkészítsük a webalkalmazásunkat.

Töltsük le a symfony honlapjáról a sandbox fájlarchívumot, ami egy üres symfony projektet tartalmaz, majd tömörítsük ki az alkalmazott szerverünk gyökérkönyvtárába, ami jelen esetben az EasyPHP www/ könyvtára. Ez az üres symfony projekt tartalmaz egy alapértelmezett alkalmazást frontend néven. Ha ki akarjuk próbálni, hogy valóban működik-e a keretrendszer, akkor gépeljük be a böngészőnkbe az alábbi URL-t:

http://localhost/sf_sandbox/web/frontend_dev.php/

Ha a symfony üdvözlő lapját látjuk a böngészőnkben, akkor a keretrendszer helyesen működik.

A projekt könyvtár struktúrájáról már korábban írtam, minden egyes symfony projekt könyvtár struktúrája ugyanúgy néz ki.

A fénykép album alkalmazásunkban az adatbázis-séma két táblából áll: Photo és Comment (az elnevezési konvenciók miatt angol nyelvű szavakat fogok használni.) Egy symfony projektben az adatbázis-sémát a *schema.yml* fájlban írjuk le, ami az *sf_sandbox/config* könyvtárban található. A *schema.yml* fájl tehát a következőképpen néz ki:

```
propel:
  photo:
    id:          ~
    file_path:   varchar(50)
    description: longvarchar
    created_at:  ~
  comment:
    photo_id:    ~
    author:      varchar(50)
    body:        varchar(50)
    created_at:  ~
```

A sémában az első sor az adatbázis kapcsolat nevét reprezentálja. Propel az alapértelmezett név a sandbox-ban, ami egy symfony-ba integrált külső komponensből ered. A hullámvonal karaktert (~) egy adattípus explicit deklarálása helyett használjuk. A symfony következtet az adattípusra a kulcs nevéből. Például a photo tábla id oszlopa magától értetődően elsődleges kulcs, ezért a típusa automatikusan növekvő, integer típusú lesz. A created_at név pedig dátum típusra utal, így ennek megfelelően timestamp típust fog kapni. A photo_id oszlop a comment táblában pedig külső kulcs lesz a photo tábla id oszlopához. Miután megszerkesztettünk a *schema.yml* fájlt, azaz megadtuk a projektünk adatbázis-sémáját, máris készen állunk, hogy legeneráltassuk a symfony-val az objektum-relációs leképezéshez szükséges osztályokat a séma alapján. A parancssorban adjuk ki a smfony gyökérkönyvtárából a következő parancsot:

c:\EasyPHP 2.0b1\www\sf_sandbox\`symfony propel:build-model`

(Mivel minden parancsot innen adunk ki, ezért a c:\EasyPHP *2.0b1\www\sf_sandbox* gyökérkönyvárat nem írom le minden alkalommal.) A propel:build-model parancs legenerálja a model osztályokat az *sf_sandbox/lib/model/* könyvtárba, amik a következők: *PhotoMapBuilder.php, BasePhoto.php, BasePhotoPeer.php, Photo.php, PhotoPeer.php, CommentMapBuilder.php, BaseComment.php, BaseCommentPeer.php, Comment.php, CommentPeer.php*

Mint a legtöbb osztály, ezen model osztályok is automatikusan betöltődnek a symfony-ban, ami azt jelenti, hogy használhatjuk őket anélkül, hogy beemelnénk őket minden oldalon. Csak meg kell hívnunk a *symfony cache:clear* parancsot minden egyes alkalommal, amikor új osztályt adunk a projektünkhöz, hogy az automatikus betöltő rendszer tudomást szerezzen róluk.

Alapértelmezés szerint a sandbox SQLite adatbázishoz van beállítva, az EasyPHP viszont MySql adatbázis szervert használ. De ez nem jelent problémát, ugyanis egyetlen utasítás segítségével beállítható a projekt MySql adatbázis-szerverhez, ami a következő:

```
symfony configure:database mysql://root:password@localhost/symfony_project
```

Értelemszerűen root a felhasználónév, password a jelszó, localhost a kiszolgáló, symfony_project pedig az adatbázis neve. Ezek után a phpMyAdmin felületen hozzuk létre az adatbázisunkat symfony_project néven. A táblák és oszlopok létrehozásához szkriptet használunk, amit szintén a keretrendszerrel generáltatunk le a következő paranccsal:

```
symfony propel:build-sql
```

A parancs hatására létrejön egy *lib.model.schem.sql* fájl a sandbox *data/sql/* könyvtárában, ami tartalmazza az SQL utasításokat. Ezt a szkript fájlt használjuk az adatbázis inicializálásához.

10.1. Adminisztrációs felület

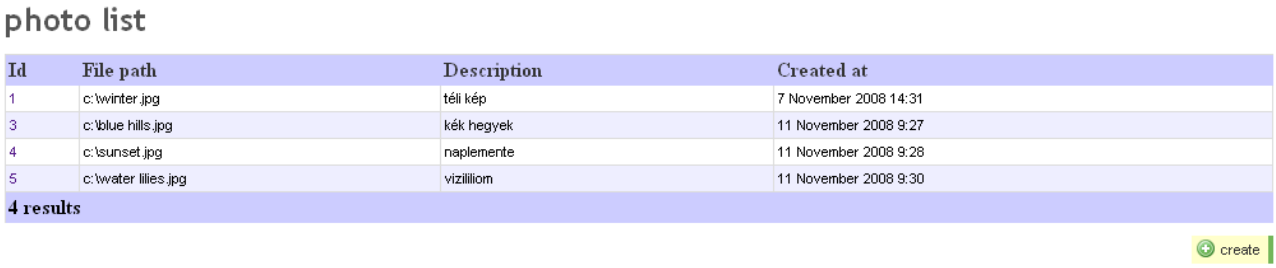
A következő lépésben a symfony keretrendszer egy nagyszerű szolgáltatását fogjuk használni, amit admin generátornak neveznek. Segítségével adminisztrációs felületet adhatunk az alkalmazásunkhoz, amin keresztül könnyen hozzáadhatunk, visszakereshetünk, módosíthatunk, és törölhetünk egy sort. Ezt CRUD modulnak is nevezik, ami a Create, Retrive, Update, és Delete angol szavakból származik. A parancs amivel ezt megtehetjük a *symfony propel:init-admin*, aminek paraméterül meg kell adni az alkalmazás nevét, a modul nevét, valamit azt az osztályt, amihez a CRUD modult létre szeretnénk hozni. Jelen esetben ez a következőképpen néz ki:

```
symfony propel:init-admin frontend photo Photo
```

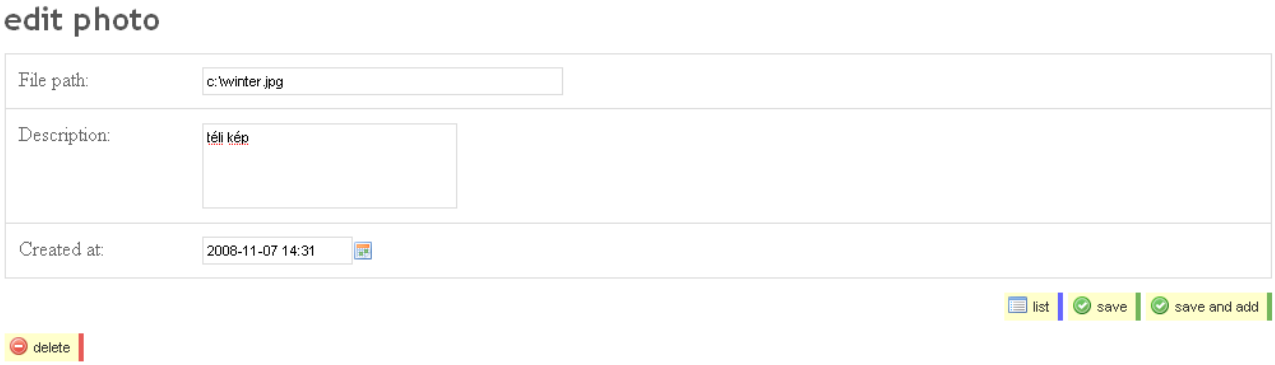
A böngészőnkben rögtön meg is nézhetjük, hogy hogy néz ki a létrejött új modell a következő URL begépelésével:

```
http://localhost/sf_sandbox/web/frontend_dev.php/photo
```

Az egyik nagy előnye a generált adminisztrációs felületnek, hogy a projekt fejlesztése során már rögtön az elején képesek vagyunk adatokat felvinni az adatbázisba, még mielőtt elkezdenénk kódolni. Az adminisztrációs felület két oldalból áll, amit lista nézetnek, és szerkesztő nézetnek hívunk. Ezeket a következő ábrákon láthatjuk is példa adatokkal:



lista nézet

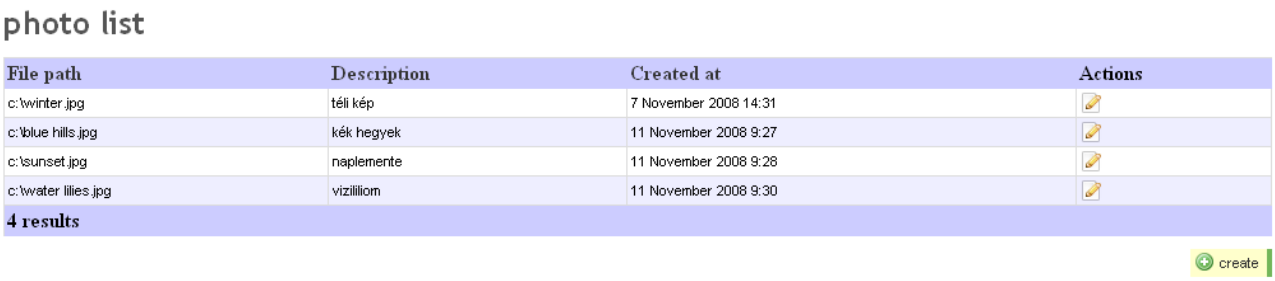


szerkesztő nézet

A keretrendszer a séma alapján készít táblázatokat, űrlapokat, és űrlap kezelő szkripteket. Bár a felület minden elemét létrehozó kód generálva van, természetesen felülírhatjuk azokat, de mégsem ez a legjobb módja a felület megváltoztatásának. Az *apps/frontend/modules/config/* könyvárban létrejön egy *generator.yml* fájl. Ez egy konfigurációs fájl, ami a generált modul vezérlését végzi, és az ebben megadott paraméterek a lista- és szerkesztés nézetet a paramétereknek megfelelően alakítják. Alapértelmezés szerint a lista nézet oszlopai a definiált séma oszlopival egyeznek meg. Az elsődleges kulcsnak megfelelő oszlop (jelen esetben az id oszlop) hiperhivatkozásként jelenik meg a táblázatban, ami a szerkesztő nézetre irányít. Az id oszlopra nincs szükségünk a táblázatban, ezért ezt el is távolítjuk a lista nézetből. Ahhoz, hogy ezt megtegyük, az alábbi módon kell szerkesztenünk a *generator.yml* fájlt:

```
generator:
  class:              sfPropelAdminGenerator
  param:
    model_class:      Photo
    theme:            default
    list:
      display:        [file_path, description, created_at]
      object_actions:
        _edit:
          name:       Edit picture properties
```

A vastagon szedett rész jelenti az újonnan hozzáadott paramétereket. A `display:` sorral mondjuk meg a keretrendszernek, hogy mely oszlopokat és milyen sorrendben jelenítsen meg a lista nézetben. Ami az `object_actions:` kulcsot illeti, ez kezeli a gombokat, ami minden egyes sor végén megjelenik. Az `_edit:` gomb egy előre definiált eszköz, tehát nem kell foglalkoznunk a megírásával. A lista nézet ezek alapján a következőképpen módosul:

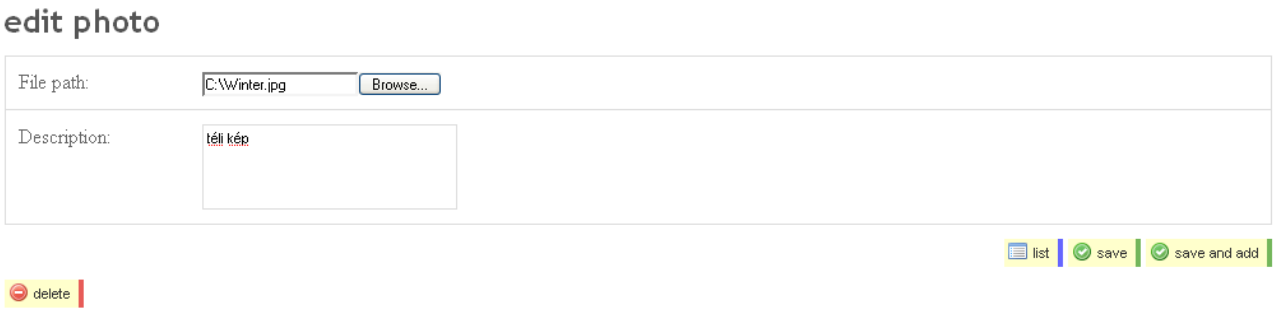


Nem túl hasznos dolog begépelni egy feltöltendő kép teljes elérési útját. Az ideális eset egy admin felület esetén az lenne, ha a felhasználó képes lenne tallózni a fájl rendszerében, és így tudná kiválasztani a feltöltendő képet. Ezt szintén egyszerűen, mindenféle kód írása nélkül elérhetjük a *generator.yml* fájl további szerkesztésével.

```
generator:
  class:               sfPropelAdminGenerator
  param:
    model_class:       Photo
    theme:             default
    list:
      display:        [file_path, description, created_at]
      object_actions:
        _edit:
          name:        Edit picture properties
    edit:
      display:         [file_path, description]
      fields:
        file_path:
          type:         admin_input_file_tag
```

A `display:` kulcsot már ismerjük, ez vezérli hogy mely mezők jelenjenek meg a nézetben, ami a szerkesztő nézet esetén azt jelenti, hogy mely mezők jelenjenek meg az űrlapon. A szerkesztő nézetben megszabadulhatunk a `created_at` oszloptól is, azon egyszerű oknál fogva, hogy a symfony képes arra, hogy ezt saját maga kezelje. Ugyanis a nevéből következtetve az adott sor elkészültének dátumát fogja értékül adni ennek a mezőnek. Ha hozzáadnánk a sémánkhoz egy

`updated_at` oszlopot, akkor a symfony ezt is maga kezelné, és minden alkalommal megváltoztatná az értékét, amikor egy sort módosítanánk. A `fields:` kulcs alatt megadhatjuk, hogy mely mezőket szeretnénk módosítani a szerkesztő nézetben (jelen esetben ez a `file_path` mező), és a típusát beállítjuk `admin_input_file_tag`-ra, ami megmondja a symfony-nak, hogy a feltöltendő fájl elérését egy tallóz gombbal tegye lehetővé. Valamit ha kiválasztunk egy fájlt, akkor az feltöltésre is kerül a *web/uploads/* könyvtárba. A szerkesztő nézet ezek alapján a következő lesz:



Szép lenne, ha a modul a feltöltött képeket megjelenítené, így ellenőrizhetnénk, hogy a feltöltésünk sikeres volt-e. Tehát ahhoz, hogy a képek is megjelenjenek a lista és szerkesztő nézetben, a *generator.yml* fájlt szerkesszük meg az alábbi módon.

```
generator:
  class:              sfPropelAdminGenerator
  param:
    model_class:      Photo
    theme:            default
    list:
      display:        [_photo, file_path, description, created_at]
      object_actions:
        _edit:
          name:       Edit picture properties
    edit:
      display:        [_photo, file_path, description]
      fields:
        file_path:
          type:        admin_input_file_tag
```

Mindössze egy úgynevezett részleges oszlopot adtunk a lista és szerkesztő nézet `display:` sorához. A részleges oszlop tulajdonképpen egy darab PHP kód, amit a generált admin felület használ. A symfony felismeri az aláhúzás karaktert, és ahelyett, hogy a Photo objektum metódusait használná

az adott érték lekérdezéséhez, megpróbálja beilleszteni a *_photo.php* fájlt az oszlop helyére. Hozzuk létre a *_photo.php* fájlt az *apps/frontend/modules/photo/templates/* könyvtárban, és mindössze helyezzük el benne az alábbi sort:

```
<?php echo image_tag('/uploads/'.$photo->getFilePath()) ?>
```

Mielőtt a symfony betöltené ezt a fájlt, a *Photo* objektumot beteszi a *$photo* változóba. A *getFilePath()* metódus azon függvények egyike, ami automatikusan generálódott a *schema.yml* fájl alapján, amikor meghívtuk a *symfony propel:build-model* parancsot. A függvény az aktuális *$photo* objektum *file_path* oszlop értékével tér vissza. Az összes többi oszlopnak is megvan a megfelelő lekérdező és beállító metódusa a generált *Photo* osztályban. Az *image_tag()* függvény egy symfony helper, ami HTML kóddal tér vissza. A helperek sablonok írásánál játszanak fontos szerepet.
Ezek után az admin felület már képes a képek megjelenítésére, viszont mi van akkor, ha nagy képeket töltünk fel? A lista és szerkesztő nézet ezeket nem jeleníti meg valami szépen. A legjobb az lenne, ha nagyméretű képek esetén a feltöltésük során lementenénk azok kisméretű változatát is, így minden kép egységes méretben lenne megjelenítve. Szerencsére a symfony a képek miniatürizált változatát is képes legenerálni, hála az sfThumbNail pluginnak, ami szintén letölthető a symfony honlapjáról. A letöltött tömörített állományt ki kell csomagolni az sandbox *plugins*/ könyvtárába és a *symfony cache:clear* parancs kiadása után már használhatjuk is ezt az eszközt. Tehát a plugin által szolgáltatott sfThumbNail osztályt fogjuk használni, hogy létrehozzuk minden egyes feltöltött kép miniatürizált változatát. Adjuk hozzá a következő két függvényt a sandbox *lib/model/* könyvtárában található *Photo.php* fájlhoz:

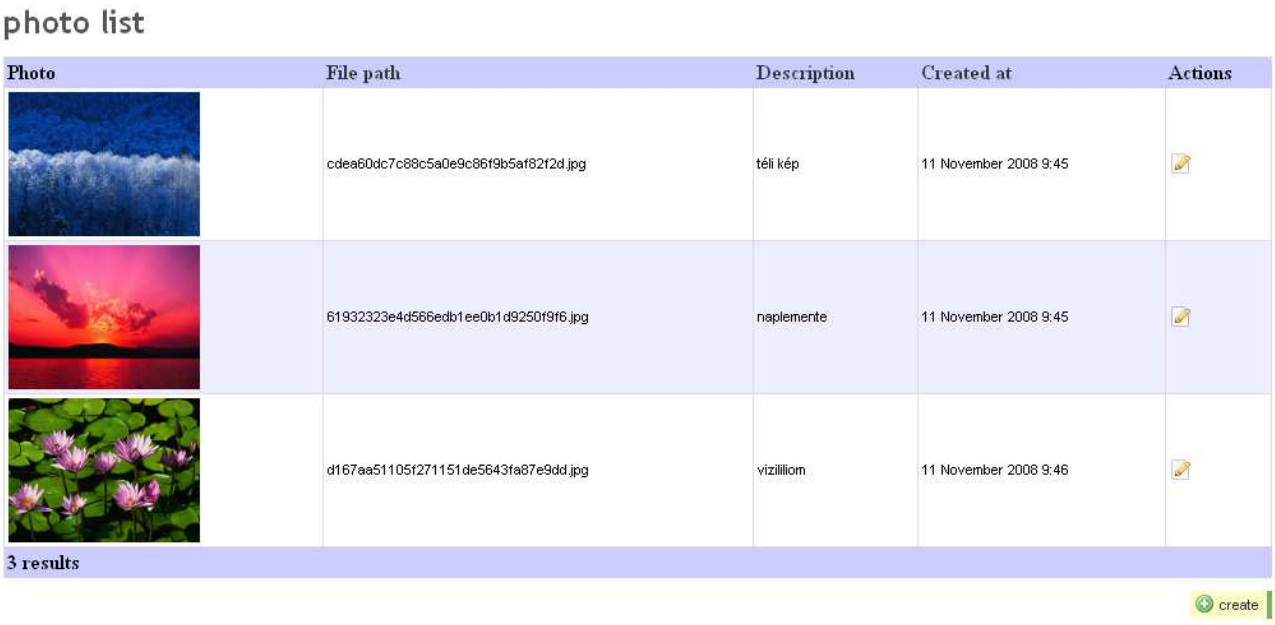
```
public function setFilePath($value)
 {
 parent::setFilePath($value);
 $this->generateThumbnail($value);
 }

public function generateThumbnail($value)
 {
 parent::setFilePath($value);
 $uploadDir = sfConfig::get('sf_upload_dir');
 $thumbnail = new sfThumbnail(150, 150);
 $thumbnail->loadFile($uploadDir.'/'.$this->getFilePath());
  $thumbnail->save($uploadDir.'/thumbnail/'.$this->getFilePath(),
            'image/png');
 }
```

Valamint hozzunk létre egy új könyvtárt a *web/uploads/* könyvtárban *thumbnail/* néven. A *setFilePath()* metódus meghívja a *BasePhoto* osztály *setFilePath()* metódusát, ez a *Photo* osztály szülője. Ezután 150*150 pixel méretű miniatürizált képeket készít az eredeti fájl alapján és lementi az *uploads/thumbnail/* könyvtárba. A *generateThumbNail()* függvényben az sfConfig osztályt használjuk, amivel kikeressük az *upload/* könyvtár elérési útvonalát. Az aktuális alkalmazás *upload/* könyvtára az *sf_upload_dir* paraméteren keresztül érhető el. Továbbá tudtára kell hozni a *_photo* részleges oszlop számára is, hogy a miniatürizált képeket jelenítse meg az eredeti képek helyett, tehát módosítsuk a már korábban létrehozott *_photo.php* fájlban található sort a következőképpen:

```
<?php echo image_tag('/uploads/thumbnail/'.$photo->getFilePath()) ?>
```

Ha mindent jól csináltunk, akkor az adminisztrációs felület lista és szerkesztő nézete egyaránt 150*150 pixel méretű képeket fog megjeleníteni, ahogy azt a mellékelt ábra is mutatja:



Ennyit az adminisztrációs felületről. Ezután következzen a végfelhasználók oldalainak elkészítése.

10.2. Felhasználói felület

Első lépésben egy public nevezetű modulba csoportosítjuk a felhasználók oldalait. A *symfony init-module* parancsot használjuk, aminek meg kell adni két paramétert, az alkalmazás nevét, valamint a létrehozandó új modul nevét. Tehát a parancs a következőképpen néz ki:

```
symfony init-module frontend public
```

Ez létrehoz egy *public/* nevezetű könyvtárt a megfelelő modul struktúrával, az *apps/modules/* könyvtárban, amiről már korábban írtam. Egy oldal a symfony-ban két részből áll: egy műveletből és egy sablonból (action and template). A művelet kódja a sablon kódja előtt fut le. Lényegében a művelet előkészíti az adatokat a sablon számára. A sablonban minél kevesebb PHP kódot kell használnunk, hogy könnyen karbantartható legyen, ugyanis szigorúan csak a megjelenítés a feladata, az üzleti logikához semmi köze.
Az alkalmazásunk mindössze annyit fog csinálni, hogy kilistázza, hogy mely képek vannak aktuálisan a fotóalbumban, ezekre egyenként rá lehet kattintani, ki lehet nagyítani, és lehet megjegyzéseket írni az egyes képekhez. Ha ezt a feladatot megvizsgáljuk, akkor ez azt fogja jelenteni, hogy szükség lesz egy műveletre, ami a képek kilistázását végi, egy ami a képhez tartozó megjegyzéseket írja ki, valamit itt tudunk új hozzászólást szerkeszteni egy kis űrlapon keresztül, továbbá lesz egy művelet, ami a képet eredeti méretben jeleníti meg. Az összes műveletet az *apss/frontend/modules/public/actions/* könyvtárban található *action.class.php* fájlba helyezzük, ugyanis ezek a műveletek logikailag összefüggnek.
A képek kilistázásához szükséges művelet a következőképpen néz ki:

```
public function executeIndex($request)
    {
        $c = new Criteria();
        $c->addDescendingOrderByColumn(PhotoPeer::CREATED_AT);
        $this->photos = PhotoPeer::doSelect($c);
    }
```

A függvény törzsében szereplő három sor megegyezik a (SELECT * FROM photo ORDER BY created_at) SQL utasítással. Első ránézésre a művelet kódja bonyolultabbnak tűnhet, mint az SQL utasítás, viszont ezáltal adatbázis függetlenné tehetjük az alkalmazásunkat, és ha később úgy döntünk, hogy másik adatbázis-motort szeretnénk használni, akkor ezt a konfigurációs fájlok segítségével pillanatok alatt megtehetjük. A másik fontos dolog, hogy a *doSelect()* metódus nem csak egy kérést küld az adatbázis felé, hanem az eredményhalmaz alapján Photo objektumokat hoz létre, így a *$this->photos* egy a Photo osztály objektumait tartalmazó tömb lesz.
Az *executeIndex()* művelet nem csak az objektumokból álló eredményhalmazt hozza létre, hanem a *$photos* tömböt elérhetővé teszi a művelethez tartozó sablon számára. Ez a *$this->* hívás eredménye a műveletben. Miután a művelet lefutott, a symfony megkeresi a megfelelő sablont a művelet számára. A sablon neve tartalmazza a művelet nevét, amihez hozzávesszük a művelet terminálási állapotának nevét. Az alapértelmezett terminálási állapot a „Success”. Tehát a symfony az *executeIndex()* művelethez egy *indexSuccess.php* nevezetű sablont fog keresni a modul *templates/* könyvtárában. Az *indexSuccess.php* a következőket tartalmazza:

```
<h1>Fotóalbum</h1>
<?php foreach($photos as $photo): ?>
<div class="photo">
    <?php echo link_to(
            image_tag('/uploads/thumbnail/'.$photo->getFilePath()),
         'public/photo?id='.$photo->getId(),
         'class=image title='.$photo->getDescription()
        ) ?>
    "<?php echo $photo->getDescription() ?>" <br>
    <?php echo $photo->getCreatedAt() ?>
</div>
<?php endforeach; ?>
```

Itt láthatjuk, hogy hogyan használhatjuk a műveletben előkészített adatokat. A sablon végigmegy a *$photos* tömbön egy foreach ciklussal, és minden egyes *$photo* objektumra meghívja a Photo

osztály megfelelő metódusát. Az *image_tag()* helpert már ismerjük, az adminisztrációs felület készítése során már használtuk. A *link_to()* függvény szintén egy helper, ami egy hiperhivatkozást generál egy másik műveletre. Legalább két paramétert vár: a link tulajdonosát (ami jelen esetben egy <img> tag), és a link célját, azaz egy belső URI-t. Egy belső URI egy modul és egy művelet kombinációja /-el elválasztva és esetlegesen további paraméterek, amiket a hagyományos URI-k nek megfelelő módon írunk. A harmadik opcionális paraméter egy sztring, ami további tag attribútumokat tartalmazhat HTML 4.0 stílusban megírva. Nem kell aggódnunk a helperek által generált HTML kód miatt, mert a symfony XHTML szabványnak megfelelő kódot generál.
A sablon eredménye az, hogy a *public/index* művelet megjeleníti a miniatürizált képek listáját úgy, hogy mindegyik kép egy linket tartalmaz a *public/photo* műveletre. Az alábbi ábrán az index művelet eredményét láthatjuk:

Ha rákattintunk egy képre, akkor a public modul photo művelete fog lefutni, ami a következőképpen néz ki:

```
public function executePhoto()
    {
        $photo = PhotoPeer::retrieveByPk($this->getRequestParameter('id'));
        $this->forward404unless($photo);
        $this->photo = $photo;
    }
```

A kérés paramétert a paraméter neve alapján a *getRequestParameter()* módszerrel kapjuk meg. A PhotoPeer model osztály *retriveByPk()* statikus metódusát arra használjuk, hogy egy Photo objektumot az elsődleges kulcs alapján megkapjunk. Ha a felhasználó egy nem létező id-vel gépeli be az URL-t , akkor egy 404-es hibaoldat kap. Ezt eredményezi a *forward404unless()* metódus. A művelethez tartozó sablon, azaz a *photoSuccess.php* tartalma a következő:

```
<?php echo link_to('vissza a fotóalbumhoz', 'public/index',
   'style=display:block;float:left;') ?>

<br>
<h1>Kép tulajdonságai</h1>
<?php echo link_to(
        image_tag('/uploads/thumbnail/'.$photo->getFilePath()),
     'public/fullSizeImage?id='.$photo->getId(),
     'class=image title=a teljes méretű képhez kattintson a képre) ?>
<br/>
<p>
Kép leírása: "<?php echo $photo->getDescription() ?>" <br>
létrehozva: <?php echo $photo->getCreatedAt() ?>
</p>

<?php use_helper('Form'); ?>
<div id="comments">
 <h2>Megjegyzések</h2>
 <?php foreach($photo->getComments() as $comment): ?>
   <?php include_partial('comment', array('comment' => $comment)) ?>
 <?php endforeach; ?>
 <div id="updateDiv">
     <h2>Szóljonb hozzá!</h2>
   <?php echo form_tag('public/addComment') ?>
     <?php echo input_hidden_tag('photo_id', $photo->getId()) ?>
     <?php echo label_for('author', 'Név') ?>
     <?php echo input_tag('author') ?><br />
     <?php echo label_for('body', 'Hozzászólás') ?>
     <?php echo textarea_tag('body') ?><br />
     <?php echo submit_tag('Elküld') ?>
   </form>
 </div>
</div>
```

Itt látunk példát olyan *link_to()* helperre, aminek a második paramétere egy olyan URI, ahol a modul és a művelet megadása mellett paramétert is megadunk. Jelen esetben az aktuális Photo objektum id-jét adjuk át a public modul *fullSizeImage* műveletének. Az oldalon kilistázzuk az aktuális képhez tartozó hozzászólásokat egy foreach ciklussal, valamint létrehozunk egy űrlapot, új hozzászólás írásához. Itt láthatunk példát az *include_partial()* helper használatára, amivel részleges fájlt tudunk az oldalunkra beemelni. Részleges fájlról már az admin felület létrehozásakor volt szó,

csak ott generált kód használta a helpert. Létre kell hoznunk egy *_comment.php* részleges fájlt, ami mindössze annyit csinál, hogy kiír egy adott hozzászólást a megfelelő formában. Ezt a fájlt szintén a *templates/* könyvtárba kell elhelyezni a többi sablon mellé.

A *use_helper('Form')* függvény lehetővé teszi, hogy az űrlapunkat symfony helperek segítségével építsük fel. A *form_tag()* helper jelzi az űrlap elejét, itt adjuk meg, hogy melyik modul melyik művelete fogja feldolgozni. Érdemes megfigyelni az *input_hidden_tag('photo_id', $photo->getId())* helpert. Ez nem jelenik meg az oldalon, de szükség van rá, ugyanis az *addComment* műveletnek tudnia kell, hogy melyik Photo objektumhoz kapcsolódóan kell lementeni a megadott hozzászólást az adatbázisba. A public modul photo művelet eredményét az alábbi ábrán láthatjuk:

Erről az oldalról visszatérhetünk a fotóalbum oldalra a „vissza a fotóalbumhoz" linkre kattintva, ami a public modul *index* műveletét hívja meg. Ha a miniatürizált képre kattintunk akkor a public modul *fullSizeImage* művelete fog lefutni, azaz megtekinthetjük a képet eredeti méretben. Ha új hozzászólást szeretnénk hozzáadni a képhez, akkor az űrlap kitöltése, valamint a „Küldés" gomb megnyomása után a public modul *addComment* művelete fut le.

Az *executeFullSizeImage()* művelet törzse megegyezik az *executePhoto()* művelet törzsével, itt is megszerezzük a Photo objektumot az adatbázisból az elsődleges kulcsa alapján. A *fullSizeImageSuccess.php* sablon pedig az eredeti teljes méretű képen kívül két linket is tartalmaz:

Erről az oldalról visszatérhetünk a fotóalbumhoz, vagy az aktuális kép tulajdonságaihoz (azaz ahonnan ide jutottunk) a megfelelő művelet meghívásával.

Ha új hozzászólást írunk a képhez, akkor az *addComment* művelet fut le, ami a következőt tartalmazza:

```
public function executeAddComment()
    {
        $this->forward404unless($photo = PhotoPeer::retrieveByPk($this-
>getRequestParameter('photo_id')));
        $comment = new Comment();
        $comment->setPhoto($photo);
        $comment->setAuthor($this->getRequestParameter('author'));
        $comment->setBody($this->getRequestParameter('body'));
        $comment->save();
```

```
        $this->photo=$photo;
    }
```

A photo_id alapján kikeressük azt a képet, amihez az új hozzászólást le szeretnénk menteni. Létrehozunk egy új Comment objektumot, majd a megfelelő beállító metódusokat használva beállítjuk a kapcsolódó képet, a hozzászólás szerzőjét valamint a törzsét, végül pedig lementjük az adatbázisba. Az *addComment* művelethez tartozó *addCommentSuccess.php* sablon tájékoztatja a felhasználót, hogy a hozzászólása lementésre került az adatbázisba:

Innen is visszatérhetünk a fotóalbumhoz, vagy a kép tulajdonságai oldalhoz, ahol megtekinthetjük az általunk létrehozott hozzászólást.

11. Összefoglalás

Ezen rövid példa alkalmazás segítségével szerettem volna bemutatni a symfony keretrendszer működését. A program a teljesség igénye nélkül készült, de már ezen néhány funkció bemutatásával láthatjuk, hogy mennyire gyorsan és hatékonyan tudunk webes alkalmazást készíteni. Az első pillantásra bonyolultnak tűnő projekt könyvtár struktúra megismerése és használata megéri a befektetett energiát, hiszen főleg nagyobb projektek esetén leegyszerűsíti a fejlesztést és a karbantartást. A keretrendszer használatával rengeteg időt megspórolhatunk, arról nem is beszélve, hogy az MVC architektúra használatával sokkal átláthatóbb kódot írhatunk, és ha be kell szállnunk egy symfony projekt fejlesztésébe, akkor nem kell heteket tölteni azzal, hogy elemezzünk a már megírt kódokat.

A dolgozat megírásánál törekedtem arra, hogy bemutassam a keretrendszer legfontosabb részeit és azok működését. Láthattunk, hogy hogyan épül fel egy projekt, a projekten belüli alkalmazások, és az azokat tartalmazó modulok. Megnéztük, hogy hogyan tudunk létrehozni egy adatbázis sémát, ebből hogyan tudunk modelt generálni, amire a teljes alkalmazásunk épül. A keretrendszer beépített objektum-relációs leképezésének segítségével rendkívül könnyen el tudtuk érni az adatbázisunkat, annak típusától függetlenül. A model alapján adminisztrációs felületet generáltunk az alkalmazásunkhoz, aminek segítségével még a kódolás megkezdése előtt adatokat tudtunk rögzíteni az adatbázisban, valamint visszakereshetünk azokat. Az adminisztrációs felület és a projekt egyéb beállításait YAML fájlok segítségével konfiguráltuk. A YAML nem olyan elterjedt, mint az XML, mégis javasolt a használata, hiszen könnyen megtanulható, és egyszerűbb szintaktikát használ az XML-nél. Láthattunk példát, hogy hogyan hozhatunk létre oldalakat a keretrendszerben. Megnéztük, hogy mi a kapcsolat a művelet és a sablon között, amiből az oldal felépül, ezek hogyan kommunikálnak egymással, valami az adatbázissal. Megismertük a keretrendszer által használt konvenciókat, amik betartása fontos egy nagy, csapatmunkában készülő projekt fejlesztése során. A symfony honlapjáról rengeteg plugin letölthető, amiket folyamatosan fejlesztenek, ezáltal is megkönnyítve a munkánkat, hiszen nagyon sok mindent nem kell megírnunk, mivel már kész van. Erre is láttunk példát a miniatürizált képek használatával.

Összességében azt lehet mondani, hogy egy keretrendszer megismerése és használata mindenképp javasolt egy webalkalmazások készítésével foglalkozó fejlesztő számára, hiszen nagymértékben megkönnyíti a munkáját, és erre a bemutatott symfony keretrendszer kiválóan alkalmas.

12. Irodalomjegyzék

- Francios Zaninotto, Fabien Potencier: The Definitive Guide to symfony, 2007
- Dave W.Mercer, Allan Kent, Steve D. Nowicki, David Mercer, Dan Squier, Wankyu Choi . PHP5, Bevezetés a PHP5 programozásába, Panem Könyvkiadó, 2006
- Julie C. Meloni: A PHP, a MySQL és az Apache használata, Panem Könyvkiadó, 2004
- http://www.symfony-project.org/
- http://www.easyphp.org/
- http://www.php.net/