DIPLOMAMUNKA

Hoffmann Antal

Debrecen

2010

Debreceni Egyetem

Informatikai Kar

WEBRAKTÁR – RAKTÁRKEZELÉST ÉS ÉRTÉKESÍTÉST TÁMOGATÓ WEBES ALKALMAZÁS

Témavezető:

Dr. Kuki Attila

Egyetemi adjunktus

Készítette:

Hoffmann Antal

Programtervező matematikus

Debrecen

2010

Tartalomjegyzék

1. Bevezetés 5
1.1. A diplomamunka témája és a témaválasztás indoklása 5
2. Követelmények feltárása, elemzése 7
2.1. A felhasználói követelmények 7
2.2. Rendszerkövetelmények 9
3. Tervezés 12
3.1. Symfony 12
3.2. MVC pattern 13
3.3. Doctrine - egy PHP-s ORM: 14
4. Implementálás 16
4.1. Adatbázis séma készítésének lehetőségei 16
4.2. Jogosultság kezelés 17
4.2.1. Jogosultság kezelés adattáblái 18
4.2.2. Jelszó védelme 18
4.2.3. Authentikáció 21
4.2.4. Dinamikus menü 21
4.3. Gyártás 23
4.3.1. Gyártás adattáblái 24
4.3.2. Űrlapok 25
4.3.3. Widgetek 26
4.3.4. Cimkék 28
4.3.5. Form megjelenítése 28
4.3.6. XSS védelem (Cross-Site Scripting) 29
4.3.7. Űrlap érvényesítés – Validátorok 30
4.3.8. Validátorok testreszabása 32
4.3.9. Űrlapok küldése 33
4.4. Értékesítés 34
4.4.1. Értékesítés adattáblái 36
4.4.2. Számla készítése 37
4.4.3. Számla nyomtatása 38

4.4.4. TCPDF ... 38
4.4.5. PDF generálás ... 39
4.5. Felület ... 44
4.5.1. Layout ... 44
4.5.2. CSS és JavaScript használata ... 46
5. Tesztelés ... 47
5.1. Unit teszt ... 48
5.2. Funkcionális teszt ... 49
6. Üzembe helyezés ... 50
7. Összefoglalás ... 51
8. Irodalomjegyzék ... 52
9. Köszönetnyilvánítás ... 52

Bevezetés

A diplomamunka témája és a témaválasztás indoklása

Azért választottam ezt a témát, mert véleményem szerint a jövőben a web az eddigieknél is nagyobb szerepet fog kapni a különböző informatikai rendszerek fejlesztésében. Manapság már sok ember számára természetes a web használata. Elmondható, hogy beépült a mindennapi életünkbe.

Szakmai megközelítésben elmondható, hogy nagyon sok informatikus dolgozik nap, mint nap azon, hogy újabb eszközöket esetleg egy új technológiát készítsenek, ami tovább népszerűsíti a web használatát azáltal, hogy kényelmesebb és megbízhatóbb alkalmazások születnek. A dolgozat elsősorban ezen eszközök rövid bemutatását, alkalmazásuk fontosságát igyekszik bemutatni. Ennek legjobb módja, ha egy konkrét alkalmazás kifejlesztésének a lépéseit vizsgáljuk. Az alkalmazás egy egyszerű hétköznapi témára – készletnyilvántartás és raktárkezelés - épül, és az itt felmerülő nehézségekre igyekszik válaszokat adni. A fejlesztés során mindig a végfelhasználó kényelme, munkája hatékonyságának növelése a cél. Mindezt hatékony, letisztult és lehetőség szerint újrafelhasználható programkód elkészítésével igyekszem elérni. Fontos szempontnak tartom a bővíthetőséget, amellyel elérhetővé válik, hogy specifikus esetekben is megállja a program a helyét, és lehetőséget adjon további fejlesztésekre. Azt az utat, ami leírja, hogy ezeket a célkitűzéseket, hogyan értem el a dolgozat elkövetkező fejezeteiben fogom részletesebben kifejteni, és példákkal illusztrálni.

Mielőtt még elkezdenénk belemerülni a diplomamunka részleteibe előtte definiálom a webes alkalmazást mint fogalmat, és megemlítem ezen alkalmazások főbb előnyeit és hátrányait is

A webes alkalmazás egy olyan szoftver alkalmazás, amelynek a hozzáférését web böngésző biztosítja Internet vagy intranet hálózaton keresztül.
A webes alkalmazás egy három rétegű alkalmazás:

- megjelenítési réteg (böngésző)
- üzleti logika (web szerver)
- adat szint (adatbázis)

A webes alkalmazásoknak számtalan előnye és hátránya is van. Elsőnek vizsgáljuk meg ezen szoftverek előnyeit:

- nincs szükség telepítésre, hogy a felhasználó használni tudja
- bárhol használható, ahol van internet elérés
- felhasználó barát kezelőfelület
- könnyen megtanulható a kezelhetősége
- a már ismert internet böngésző segítségével futtatható
- költséghatékony megoldás
- biztonságos adattárolás
- magas szintű rendelkezésre állás
- folyamatos adatmentés
- automatikus frissítés
- jelentős fejlesztések várhatók a web terültén
- növekedik a webes alkalmazások népszerűsége

A webes alkalmazások hátrányai:

- hálózati kapcsolat szükséges
- különös figyelmet kell fordítani a biztonságra

A bevezetés zárásaként megjegyzem, hogy a web egy hatalmas piac, ami kimagaslóan növekszik. Világszerte naponta több tízezerrel nő a világhálót felhasználók száma, ami magyarázatott ad arra, hogy manapság miért is vált a web az egyik legnépszerűbb informatikai eszközzé.

Követelmények feltárása, elemzése

A felhasználói követelmények

A felhasználói követelmények a követelmény feltárás kezdeti lépéseihez sorolhatjuk. Ezen követelmények azt igyekszenek megfogalmazni, hogy mit vár el a megrendelő, a felhasználó az elkészült programtól. Mivel az esetek jelentős részében a felhasználóknak nincs konkrét követelményük, csak egy vízió, egy gondolat, ezért mindenképpen érdemes időt szánni arra, hogy a közös megbeszélések során kiderüljön mit is szeretnénk megvalósítani. Az esetek jelentős részében a felhasználói követelmények nagyon gyakran megváltoznak, ezért véleményem szerint nem szabad egyetlen egy interjú alatt komoly következtetést levonni. Mindenképp fontos, hogy folyamatos legyen a kapcsolat a fejlesztő és a megrendelő között, ezzel nagyon sok kellemetlenséget el lehet kerülni Miután ez a project nem tartozik kiterjedésében abba a kategóriába, hogy külön dokumentumot vagy dokumentumokat készítsünk, ezért egyszerűen belefoglalom a dolgozatomba.

A megrendelő rendelkezik egy gyárral, amihez tartozik egy központi raktár és számos üzlet. Egy olyan szoftvert szeretne készítetni, ami nem igényel jelentős beruházást informatikai eszközök vásárlásra, és a fejlesztési valamint a karbantartási költségek sem magasak. Az igények között szerepel, hogy a szoftver mind a raktárban mind az üzletekben kielégítse a követelményeket, illetve minden üzlet kapcsolatban álljon a központi raktárral, és a raktár is az üzletekkel. A megrendelő távlati tervei között szerepel a vállalkozásának a bővítése, ami a gyár kibővítését, egy újabb raktár és több üzlet kiépítését takarja. Kérései között szerepel, hogy az elkészülő program az ezekhez hasonló változásokat zökkenőmentesen kövesse.

Mivel nagyon sok részletkérdésre még nincs konkrét válasz, ezért első lépésként csak az alapvető igényeket fogalmazom meg

Ki is fogja használni az elkészült szoftvert?

- megrendelés: megrendeléseket az egyes forgalmazó cégek munkatársai illetve az üzletek alkalmazottai készítenek, a vásárlási igényekhez igazodva
- termelés: a termelésben résztvevő emberek három nagy csoportra bonthatók
    - managerek vagy vezetők,
    - logisztikusok,
    - fizikai munkások

- eladás: minden üzletben vannak eladók, akik a termékek értékesítését végzik. Üzletcsoportonként van egy manager, aki figyelemmel kíséri az egyes üzletek teljesítményét.

A gyártást megvalósító komponens az alábbi fő részekből épül fel:

- alapanyagok:
  - a gyártáshoz szükséges termékek. Ezen termékeket külön szeretnénk nyilvántartani. Gyakran előforduló probléma volt eddig, hogy egyes alapanyagok vagy nem voltak a raktárban vagy pedig kis mennyiségben, emiatt a termelés akadozott. Ahhoz, hogy ezek az apró emberi figyelmetlenségek elkerülhetők legyenek minden alapanyaghoz hozzá szeretnénk rendelni egy min mennyiséget. Ha a raktáron lévő alapanyag mennyisége ennél a határ értéknél kisebb, akkor automatikusan értesítse az egység vezetőjét.
- termelés, gépnapló:
  - nyilvántartást akarunk készíteni arról, hogy az adott gépen az adott időpontban mit és ki(k) gyártottak, milyen mennyiségben.
- selejt:
  - a selejt termékek elsősorban statisztikai szempontból fontosak. Megtudhatjuk, hogy egyes terméket hány százalékos veszteséggel gyártunk és mi az oka annak, hogy nem kerülhet az adott árú értékesítésre. A vezetőség feladata, hogy ezen okokra megoldást találjon és csökkentse a selejt termékek számát.
- fél kész termékek:
  - ebbe az egységbe olyan termékeket akarunk nyilván tartani, amiknek valamilyen komponensük még nem készült el. Ennek számos oka lehet, mint pl: beszállító késik, a termelés csúszik gép hiba vagy egyéb okok miatt
- kész termékek:
  - minden elkészült terméket egy külön raktárban tartunk nyilván.
- megrendelések:
  - kulcsfontosságú funkcionalitás. Érkezhet az saját üzlettől és külső cégektől. Gyakorlatban elmondható, hogy nagyon gyakran váltakoznak a rendelések száma és nagyon gyakran a meglévők is módosulnak. Mivel ez egy nagyon

komplex modulnak ígérkezik, ezért a kifejlesztése közben kiemelt kapcsolat szükséges a fejlesztő és felhasználó között.

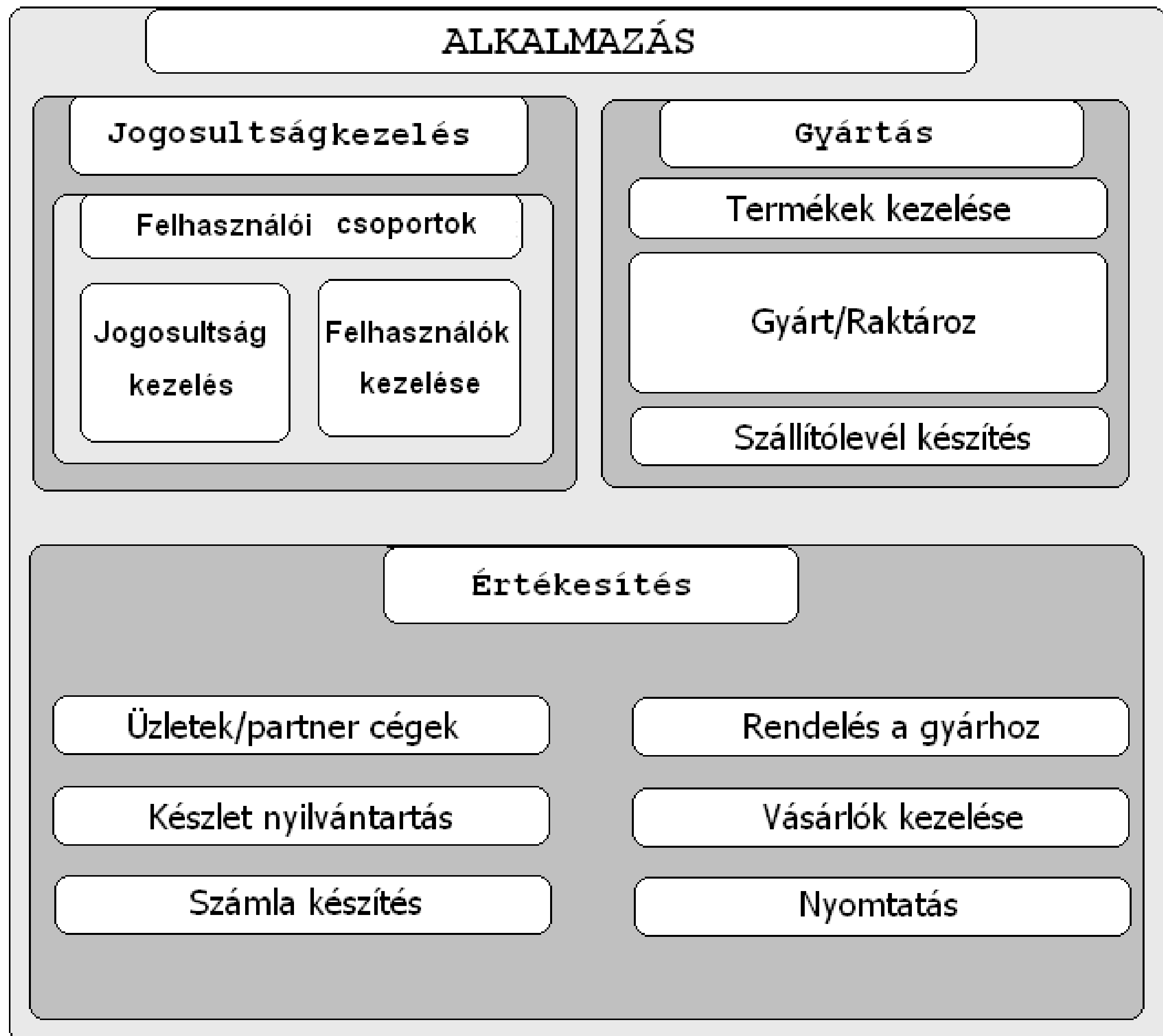
Az értékesítéshez tartozó komponens az alábbi részekből épül fel:

- vásárlók:
  - A cég szeretné, ha az újonnan érkező és a meglévő ügyfelekről adatokat tudjon nyilvántartani. Ennek számos oka van és lehet még, ezért további pontosításra lesz majd szükség. Megkülönböztetünk magánszemélyeket és cégeket.
- készleten lévő termékek:
  - az aktuális készletet mutatja meg. Mivel egy üzlethelységre kell most gondolnunk ezért egyértelmű, hogy csak limitált készlettel tudunk gazdálkodni, hiszen üzletenként veszteséges lenne külön nagyobb kiterjedésű tárolót fenntartani.
- megrendelések:
  - kulcsfontosságú funkcionalitás. Két esetet tudunk megkülönböztetni. Az első esetben a megrendelések során mindig a gyártó egység aktuális kész termékeiből tudunk válogatni, és csak a központi raktárban lévő mennyiséget tudunk megrendelni. A második esetben igényeket küldünk a központba, hogy az adott termék(ek)ből mekkora mennyiségben lenne szükség.
- nyomtatás:
  - Az megrendelő igényt tart arra, hogy az üzletekben eladott termékekről számlát tudjon kiállítani, és azt nyomtatott formában át tudja adni a vásárlónak. Továbbá azt is szeretné, hogy szállítóleveleket is lehessen nyomtatni. Későbbi tervei között szerepel, hogy termék katalógust tudjon készíteni, illetve az aktuális leértékeléseket PDF formátumban e-mailen el tudja küldeni a törzsvásárlóknak

Rendszerkövetelmények

A rendszerkövetelmények elkészítése a felhasználói követelményekre épül. A felhasználói követelményeket bontja le, abból a célból, hogy a tervezés fázis a rendszerkövetelmények alapján kiindulhasson. A rendszerkövetelményeket megfogalmazó nyelv strukturáltabb, formalizáltabb, kevésbé félreérhető. Leírása során erős grafikus jelölés jellemzi, mint például diagramok. A tervezés fázisban az itt leírtakat használjuk irányelvként.

A leírtak alapján három jól elszeparálható egységre bontható az alkalmazás. Minden egység logikailag összetartozó részegységre bontható. A következő ábra ezeket az egységeket tartalmazza:



Mivel nem egy olyan hétköznapi weboldalt kívánunk elkészíteni, aminek a tartalmát bárki elolvashatja, hanem egy olyan webes alkalmazást, ami fontos információkat tartalmaz, ezért mind az adatok, mind az oldalak védelme egy igen fontos feladat. A jogosultságkezelés hivatott elvégezni a felhasználók authentikációját, hogy a felhasználóknak csak egy jól meghatározott csoportja tudja látogatni az elkészülendő alkalmazásunk oldalait. A felhasználói követelmények alapján, érdemes a felhasználókat csoportokba szervezni. A jogosultságokat pedig a csoportokhoz rendezni. Természetesen le kell kezelni azt az esetet, hogy egy felhasználó több csoporthoz is tartozhat. A cél tehát egy olyan felület elkészítése, ahol az adminisztrátor a felhasználó csoportokat, jogokat, és felhasználókat könnyen tudja

kezelni. Az alkalmazásban nem lesz regisztrációs felület. Minden felhasználó csak a rendszer adminisztrátorától kaphat hozzáférést a szoftverhez.

A megrendelőnk rendelkezik egy gyárral és több olyan üzlettel, ahol az általa készített termékek forgalmazását végzi. A gyártás és az értékesítés két olyan egység, amely egymástól jól elszeparálható. Mind a gyártást, mind az értékesítést további alegységekre bontható. Ezek az alegységek a felhasználói követelményeket foglalja magába.

A gyártás során termékekből egy újabb terméket állítunk. Ennek tükrében érdemes a termékeket termékcsoportba rendezni. A gépnapló készítés és a raktározás olyan tevékenységek, amik szorosan egymáshoz tartoznak. Ennek tükrében egy logikai egységbe sorolhatóak. Az elkészült termékeket el is kell jutatni az üzletekhez vagy a partner cégekhez. Ezt az adminisztrációs feladatot foglalja magába a szállítólevél készítés.

Az értékesítés során már csak kész termékekkel foglalkozunk. Ezen terméket a gyártól kapjuk, melyet az üzletekben készletezzünk. A készletből derül ki, hogy az adott üzletben milyen termékek szerepelnek. Fontos, hogy csakis készleten lévő terméket tudunk kiszámlázni a vevőnek. A számlákat az adatbázisban tartjuk nyilván, és elsősorban csak id-kat tartalmaz. Azonban lehetőséget kell biztosítani a számlák nyomtatására is. A vásárlókról csak azokat az információkat tartjuk nyilván, amik az értékesítés szempontjából fontosak, mint például azt, hogy az egyes vásárlók rendelkeznek-e kedvezménnyel, és ha igen, akkor mekkora kedvezménnyel.

Az üzletek száma nagy valószínűséggel változhat a szoftver használata közben, ezért ezt a fajta változást kezelni kell az alkalmazásnak.

A követelmények áttekintése során megállapítottam, hogy hasznos lenne, ha a gyártás és értékesítés során a termékekhez tartozó mértékegységeket (például: db), és áfa kulcsokat egy külön adattáblában tárolnám. Ezzel csökkenne a redundancia mértéke, ugyanis azokban a táblákban, ahol ezekre az értékekre van szükség, csak a megfelelő id-t kell letárolni. További előnye ennek az elképzelésnek, hogy az adatbevitelt kényelmesebb, gyorsabb és egységes lenne, ha egy legördülő listából ki lehetne választani a megfelelő értéket.

Tervezés

A tervezés során arra adok választ, hogy az eddig meghatározott követelményeket milyen módon fogom implementálni. Számos olyan megszorítás van, ami véleményem szerint napjainkban elengedhetetlen, ahhoz hogy egy olyan alkalmazást kapjunk eredményül, ami képes igazodni a folyamatos változásokhoz.

Nézzünk néhány ilyen megszorítást:

- rövid legyen a fejlesztési idő.
- legyen prototípus készítési lehetőség, aminek segítségével a megrendelőnek már látható eredményt adhatunk, továbbá támpontot is ad, hogy a fejlesztés milyen irányban haladjon tovább.
- fejlesztési költségek alacsonyak legyenek.
- könnyen lehessen új funkciókat bevezetni a már működő rendszerbe, illetve a változó üzleti logikából származó módosításokat is rövid időn belül teljesíteni lehessen.
- egységes konvenciók a fejlesztés során. Ez elsősorban akkor hasznos, ha új fejlesztő érkezik a csapatba, illetve ha a mások által készített kódot kell nekünk átnézni debugolás, vagy módosítás miatt. Továbbá az egységes fejlesztési konvenciók egyfajta időmegtakarítás is jelent, ami további költségcsökkentést eredményezhet.
- a későbbiekben bekövetkező informatikai változások miatt lehetőleg adatbázistól és operációs rendszertől függetlenül is lehessen használni a rendszerünket.
- Nagyon fontos továbbá, hogy a bekövetkező adatbázis módosítások ne okozzanak adatveszteséget a működő rendszerben.

Symfony

A célom az, hogy ezeknek a megszorításoknak az elkészülő alkalmazás mindenképp eleget tegyen. Ehhez a Symfony keretrendszert hívom segítségségül. Hogy miért pont ezt a keretrendszert választottam, annak több oka is van, mint például:

- Symfony egy komplett keretrendszer, amit webalkalmazások fejlesztésének optimalizálásához terveztek
- Számos eszközt és osztályt tartalmaz, amik megrövidítik egy bonyolult webalkalmazás fejlesztési idejét.

- Különféle projektekben alaposan letesztelték és jelenleg is használják számos országban magas volumenű üzleti weboldalak üzemeltetésnél.
- Lehetőséget biztosít a már elkészült modulok tesztelésére.
- A Symfony keretrendszernek része a Doctrine, ami véleményem szerint az egyik legjobb adatbázis elérési réteg. A Doctrine által a Symfony adatbázis-motor független, vagyis nem igényel idő és költségigényes beavatkozást, ha az adatbázis rendszert kicseréljük az alkalmazás alatt.
- Elég stabil hosszú távú projektek alkalmazásához
- A kódgenerálási eszközök nagyon jók prototípuskészítéshez
- A naplózási funkciók pontos adatokat adnak az adminisztrátoroknak az alkalmazás tevékenységeiről
- Beépülő modulok magas szintű bővíthetőséget tesznek lehetővé

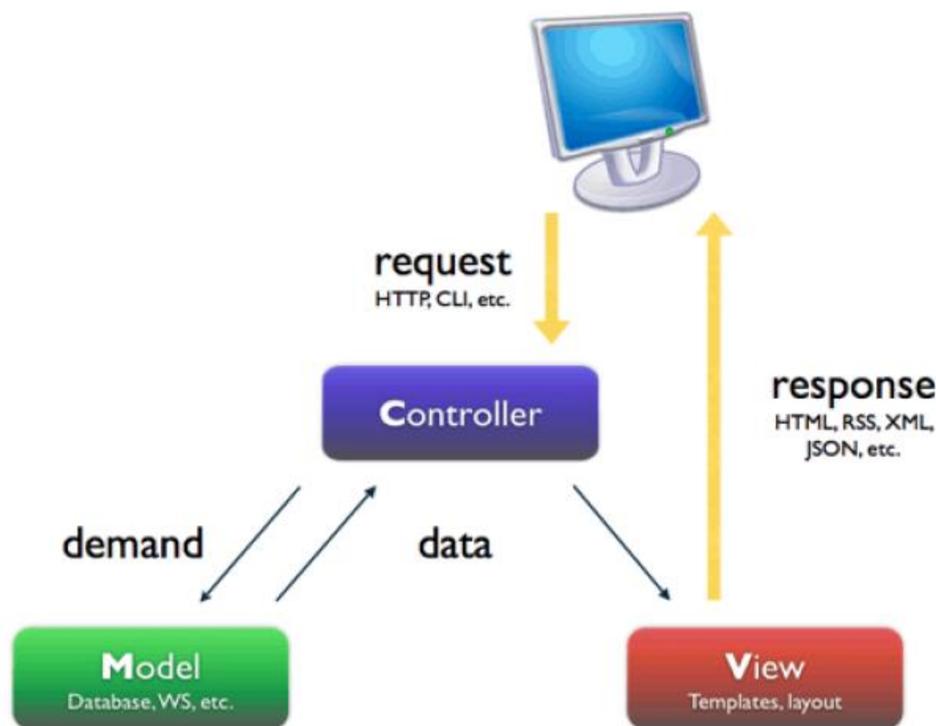
A lista még folytatható lenne, hogy még mi mindenre ad lehetősége a Symfony a fejlesztés megkönnyítésére, most viszont néhány mondatban bemutatom az MVC modellt, és a keretrendszernek azon elemeit, amiket az implementálás során minden iterációban használni fogok.

MVC pattern

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

- Model: a model részben valósítjuk meg az üzleti logikát. A model feladata definiálni a program által használt adatstruktúrákat, illetve azokat a szabályokat, amely alapján hozzáférünk ezekhez, illetve módosíthatjuk őket.
- View: a view-k feladata a model tartalmának bemutatása, a program kimenetének generálása. Az adatokhoz csak a model objektumain keresztül férhetnek hozzá. Könnyen előfordulhat, hogy rendszerünket többféle interfészen keresztül (böngésző, mobil telefon, webszolgáltatás, stb.) használják, így többféle kimenetre (HTML, WML, XML) lehet szükség. Itt jelentkezik az MVC használatának előnye, ha egy új interfész típust kell támogatnunk, akkor csak egy új view-t kell készítenünk, ami a már meglévő modelt használja.

- Controller: ez az alkalmazás „lelke”. Ez köti össze a model-t és a view-kat. A felhasználó által eszközölt akciót (form elküldése, kattintás egy linkre, stb.) lefordítja egy a model által végrehajtandó akcióra. Ezután szintén a controller dolga, hogy az akció lefutásának eredményétől függően megjelenítse a szükséges view-t.



Az adatbázis elérésre minden iteráció során szükségünk lesz, ezért ezt a témakört kiemeltem a tervezés fázisába. A most következő részben bemutatom miért is jó nekünk az ORM és az azt megvalósító Doctrine.

Doctrine - egy PHP-s ORM:

A hagyományos adatbázis elérésnek számos hátránya van. Néhány ilyen hátrány:

- Nem hordozható kód, adatbázisonként eltérő utasítások
- Adatbázis szinten kell gondolkodni, ismerni kell a teljes adatbázis felépítését
- Ismerni kell az adott adatbázis szintaktikáját
- Nincsenek újrafelhasználható kódok, sokat kell gépelni
- Hibakezelést, tesztelést csak alapszinten támogatja
- Nem MVC barát

A hagyományos adatbázis elérés hátrányaira megoldást ad nekünk az ORM (object relational mapper). Az ORM számos előnnyel rendelkezik, mint például:

- A programozó elől elfedi az adatbázis szerkezetet
- A táblákat és a mezőket csak metódusok segítségével lehet elérni
- Relációs adatbázisokat kezelhetünk objektumrelációs adatbázisként
- Gyorsabb fejlesztés
- Tesztelés egyszerűsítése

A Doctrine egy PHP-s ORM. Az ORM az adatbázis táblákat objektumokra képzi le. Minden táblához tartozik két osztály. Egy tábla szintű és egy sorszintű osztály. A sor osztály tartalmazza a tábla oszlopait, ezek típusait, megszorításokat, táblák közötti kapcsolatot, get és set metódusokat. A tábla osztály tartalmazza sor osztályokkal visszatérő metódusokat, mint például a find() és findAll() metódus.

A Doctrine-t nem csak új adatbázis, új alkalmazás készítésénél használhatom, hanem akkor is ha már egy meglévő adatbázisba akarjuk beépíteni. Minden esetben szükségünk lesz egy modellre, amely tartalmazza az adatbázis osztályokat

A Doctrine-nal migráló kódokat is tudunk készíteni. A migrálás az olyan problémákra ad megoldást, mint például:

- Több adatbázist kell párhuzamosan változtatni
- Az adatbázis változásokat nyomon kell követni

A migrálás során minden adatbázisban létrejön egy migration_version tábla. Ebben tárolja a Doctrine az adatbázis aktuális verzióját. Minden migráló kódnak van egy sorszáma. Ez a sorszám adja meg, hogy hányadik verzióra migrálja az adatbázist. Attól függően, hogy lefele vagy felfele migrálunk végrehajtódnak az up() és a down() metódusok. Az up() metódus például létrehoz egy adattáblát, a down() metódus pedig töröl egy adattáblát.

A keretrendszer és a Doctrine további tulajdonságairól a soron következő implementálás fázisban még lesz leírás .

Implementálás

Az implementálást négy ciklusban végzem el. Minden ciklus végén egy - az alkalmazás számára fontos - egység kerül megvalósításra. Elsőként a felhasználók jogosultság kezelésével, authentikációjával foglalkozom. Ezt követőn azokat a modulokat készítem el, amikre a gyártás során lesz szükség. Következő iterációba megvalósítom az értékesítéshez szükséges modulokat. Zárásként az elkészült prototípus felületéről írok néhány gondolatot. Az elkészült egységek együttesen valósítják meg azt a célt, amit a követelmények feltárása során megfogalmaztam. Az egyes iterációs ciklusok egységesen a szükséges adattáblákat írja le, első lépésként. Megjeleníti az adattáblák közötti kapcsolatokat. Ezt követően egy rövid jellemzést is adok az elkészült modellről.

Fontos, hogy az implementálás során nem egy teljesen kész, minden felhasználói igénynek eleget tevő alkalmazás készül, hiszen ezen dolgozat magára a fejlesztésre, mint tevékenységre helyezi a hangsúlyt, amit természetesen igyekezni fogok minél jobban illusztrálni. Továbbá szó lesz a felhasznált eszközök hatékonyságáról és előnyeiről is.

Az implementálás során számos modul készül el. Minden modul csak egy feladatkört fed le. Ennek a megvalósítási módnak nagy előnye, hogy nem okoz különösebben problémát az, ha a megrendelő a prototípus bemutatása során, nem elégedett bizonyos feladat megvalósításának a módjával. Rövid időn belül el tudunk készíteni egy másik modult, mindezt úgy, hogy illeszkedjen a már elkészült egységekhez. Továbbá, közelebb is kerülünk a felhasználói igényekhez, pontosabb követelményrendszert tudunk készíteni, hiszen most már lesz viszonyítási alapom. Nem utolsó szempont az sem, hogy ebben az esetben a fejlesztési költség is alacsonyabbak lesznek.

Adatbázis séma készítésének lehetőségei

Mielőtt még elkezdeném tárgyalni az egyes iterációs ciklusokat, előtte bemutatom, hogy a Symfony milyen támogatás nyújt az adatbázis sémájának a elkészítéséhez. A sémák leírására a Symfony a YAML formátumot használja.

A YAML szerkezeti felépítésében sorbehúzásokat használ. Nagyon egyszerű nyelv és gyorsan olvasható. Az XML-hez hasonló módon ír le adatokat, de sokkal egyszerűbb szintaktikát használ. A YAML fájlok készítése során van néhány olyan konvenció, amit 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, valamint 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.
- Megjegyzéseket # megadása után írhatunk a sor végéig.

A lista még folytatható lenne, viszont a fejlesztése során elsősorban ezekre a konvenciókra volt szükségem.

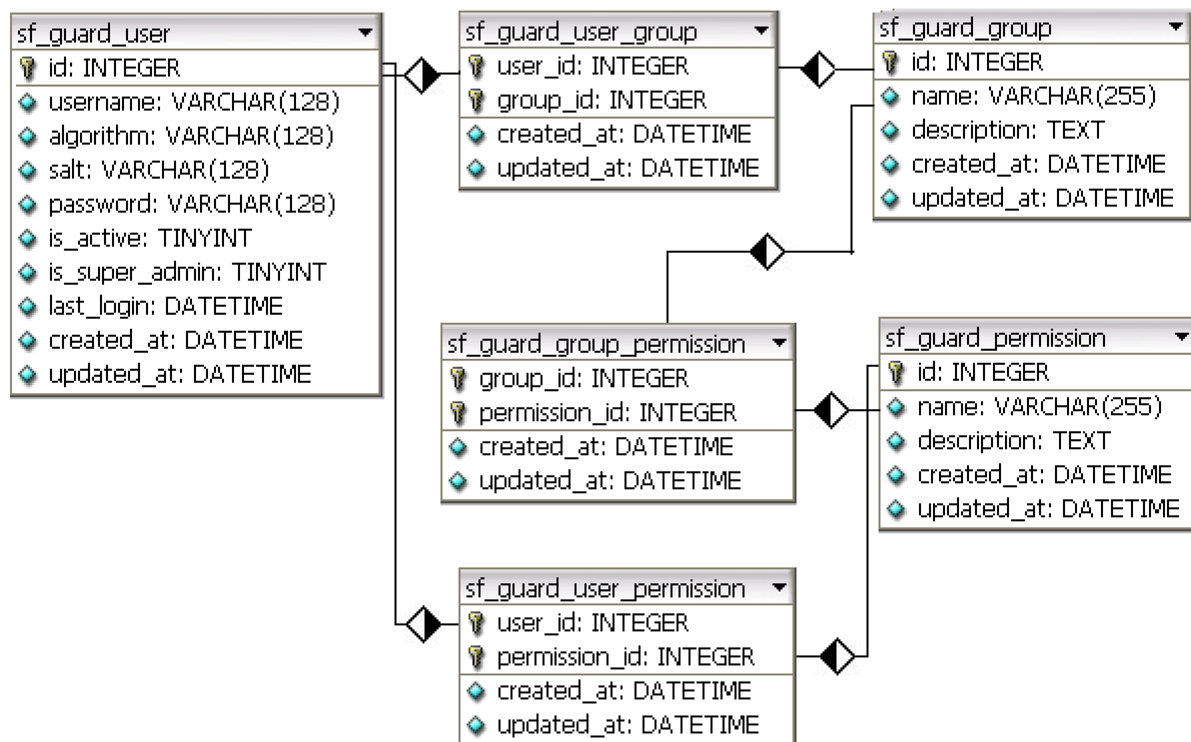
Az adatbázisom sémája a schema.yml fájl tartalmazza, mely projektem_neve/config/doctrine/ könyvtárban található. A Symfony nem kötelezi a fejlesztőt arra, hogy csak ezen fájl szerkesztésével tudja módosítani az adatbázist. Lehetőség van még migráló kódok használatára is, vagy elkészíthetjük az adatbázisunk tábláit egy grafikus felületű kliens programmal, majd egy parancssorból futtatható PHP kód segítségével az elkészült adatbázis sémáját kigenerálhatjuk a schema.yml fájlba. Ez már kényelmesebb, de ebben az esetben kompromisszumot kell kötnünk, ugyanis olyan tartalmat kapunk, ami a fejlesztő szemszögéből felesleges sorokat tartalmaz. Továbbá, mivel az adatbázis nem tárolja a táblák közötti kapcsolatokat, ezért ezeket nekünk kell kézzel beállítani. Szerencsére a YAML fájlok könnyen érthetőek, egyszerűen szerkeszthetőek. Minden táblában a relation után megadható azoknak a tábláknak a neve, melyekkel kapcsolatban áll.

Jogosultság kezelés

A felhasználók kezelésére egy a keretrendszerhez telepíthető sfDoctrineGuardPlugin nevezetű plugin-t fogok használni. Ez a plugin lényegében a Symfony keretrendszer belső authentikáló osztályaira épül. Az sfDoctrineGuardPlugin használatával a felhasználók kezeléséhez egy komplett jogosultságkezelő modult kapunk, amit nagyon egyszerűen az igényeinkhez tudunk igazítani. A szükséges konfigurációs beállításokat a generate.yml fájlban tudjuk elvégezni. Az sfDoctrineGuardPluginak saját schema.yml fájlja van, amit a plugin config/doctrine mappában találhatunk meg. Ebben van leírva azoknak az adattábláknak

sémája melyekre a jogosultság kezelés során szükségünk van, továbbá tartalmazza ezen táblák között lévő kapcsolatok.

Jogosultság kezelés adattáblái



A fenti adattáblák sémáját és a táblák közötti kapcsolatokat a projektmappa\plugins\sfDoctrineGuardPlugin\config\doctrine\schema.yml fájlban található meg. Ahogy a fenti ábra is mutatja az egyes táblák között N:M kapcsolat van. Ez azért van, mert egy felhasználó több csoporthoz is tartozhat, mint pl: manager, illetve egy felhasználó természetesen több joggal is rendelkezhet. A plugin a felhasználók kezeléséhez négy modult készít, melyek rendre a következők: sfGuardAuth, sfGuardGroup, sfGuardPermission, sfGuardUser. Az sfGuardAuth modul felelős a felhasználó bejelentkeztetésért, és kiléptetésért. Az sfGuardGroup modul végzi a felhasználói csoportok kezelését, az sfGuardPermission segítségével tudjuk a felhasználói jogokat kezelni, és végül az sfGuardUser modul az, ami a felhasználók karbantartását végzi.

Jelszó védelme

Ahogy a fenti ábrán is látható az sf_guard_user táblában van 3 mező, ami megmozgathatja a fantáziánkat. Az algorithm mező azt tárolja, hogy az adott felhasználó jelszavát, milyen kriptográfiai algoritmussal tároltuk el. Az alapértelmezett tárolási mód az

sha1 titkosítás. Természetesen ezt meglehet változtatni, sőt akár saját titkosítást is használhatunk. Ezekhez a változtatásokhoz mindössze csak annyit kell tennünk, hogy az ’ app.yml’ fájlban megadom hogy milyen algoritmus szeretnénk használni.

pl:

```
all:
      sf_guard_plugin:
      algorithm_callable: [MyCryptoClass, MyCryptoMethod]
     VAGY
all:
      sf_guard_plugin:
        algorithm_callable: md5
```

Esetleg felmerülhet bennünk a kérdés, hogy miért tároljuk minden felhasználó esetén, a titkosítási algoritmus? Erre az a válasz, hogy ha későbbiekben meggondolnák magunkat és másik titkosítást akarunk bevezetni, de a rendszerben már akár több száz felhasználó szerepel, akkor nem kell minden user esetén újra generálni a jelszót.

Az elkészült alkalmazásban az alap beállítások hagytam meg, mivel a jelenlegi igényeket ez is kielégíti. Az sha-1 ’Secure Hash Algorithm’ algoritmus egy meglehetősen gyors és kriptográfiai értelemben kellően erős lenyomatkészítő eljárás, ezért széles körben használják. Az sha-1 egy iterációs struktúrát követ, melynek végén előáll a lenyomat. Az sha-1 titkosítás használatával a felhasználók jelszavait nagyobb biztonságban tudhatjuk, mintha a „hagyományosnak” tekinthető md5 kódolás helyett. Ma már léteznek olyan md5 visszakereső adatbázisok, aminek köszönhetően egy egyszerű jelszó md5 megfelelője könnyen visszafejthető. Ha ebből a szempontból vizsgáljuk, akkor az sh1 kódolás lesz az első védelmi vonaluk, hogy megakadályozzuk a rendszer feltörését. A második védelmi vonalunk, az úgynevezett „salting - sózás”, ami lényegében azt takarja, hogy egy karakterlánccal megfűszerezzük a jelszavakat, mivel így - a használt karakterlánc ismeretének hiányában - a támadó még mindig nem tud mit kezdeni a megszerzett visszafejtett jelszavakkal. Ezt a karakterláncot az sfDoctrineGuardPlugin véletlenszerűen állítja elő, ami tovább erősíti a rendszer védelmét.

A plugin a jelszavak kezelését az abstract PluginsfGuardUser osztályba tárolja. Ezt az osztályt a BasesfGuardUser osztályból származik. A BasesfGuardUser osztályt a doctrine generálja a schema.yml fájlból. Ez az osztály írja le a sf_guard_user táblát. Ebben az osztályban a setTableDefinition() metódus írja le, hogy milyen mezői vannak a táblának. Mi a mezők neve, mi a típusa, milyen megszorítással rendelkezik stb. Ez az osztály rendelkezik

még egy setUp() metódussal is, mely megmondja, hogy melyik táblával milyen kapcsolatban áll a model. A PluginsfGuardUser osztály a project\plugins\sfDoctrineGuardPlugin\lib\model\doctrine\... elérési úton található. Ennek az osztálynak számos metódusa van, de a következőket mindenképp kiemelném:

```
/**
* ez a metódus állítja be egy adott felhasználó jelszavát.
*
* paraméterként egy stringet vár.
*/
public function setPassword($password) {
     …
     //salt meghatározása

     if (!$salt = $this->getSalt())
         {
         $salt = md5(rand(100000, 999999).
           $this->getUsername());
           $this->setSalt($salt);
         }
     …
     //az ellenőrzést és beállítást követően lementjük az
     adatbázisba a jelszót

     parent::_set('password', call_user_func_array(
                                  $algorithm,
                             array($salt.$password))
                             );
}

/**
* ez a metódus végzi a felhasználó jelszavának az
ellenőrzését.
*
* paraméterként egy stringet vár.
* visszatérési értéke boolean
*/
public function checkPasswordByGuard($password) {

     //a szükséges beállításokat és ellenőrzéseket követően

     return $this->getPassword() == call_user_func_array(
          $algorithm, array($this->getSalt().$password));
}
```

Authentikáció

A Symfonynak a hozzáférési beállításokat a következő elérési úton megtalálható security.yml fájlban adhatjuk meg.

```
# apps/app_name/config/security.yml
     default:
     is_secure: false
```

Ha az is_secure értékét true/on –ra állítom, akkor az alkalmazásom megkövetelni az oldal látogatójától a hitelesítést. A keretrendszer lehetőséget biztosít arra, hogy akár action szinten is megköveteljünk bizonyos jogokat a felhasználótól. Például, ha egy felhasználó nem rendelkezik a törlés jogával, akkor tájékoztatja a felhasználót, hogy úgynevezett credential –el kell rendelkeznie. A keretrendszer lehetőséget biztosít különböző metódusokon keresztül, hogy ezeket a credentials –et be lehessen állítani.

Dinamikus menü

Az első iteráció a dinamikus menü elkészítésével zárul le. A dinamikus menü mindig a bejelentkezett felhasználótól függően készít egy egyszerű listát, amit majd egy CSS fájl segítségével animálunk, hogy ezzel is növeljünk az alkalmazásunk megjelenésének a színvonalán. Minden szükséges információ a menü elkészítéséhez, a már meglévő adattáblákból kinyerhető. Azzal, hogy nem készítettem újabb adtatáblákat helyet takarítottam meg, viszont a felhasználók kezeléséhez szabályokat kellett bevezetnem:

- a felhasználói csoportok nevei lesznek a főmenü pontok.
- a felhasználó csoportokhoz tartozó jogok nevei lesznek az almenüpontok, a jogok leírása mezője tartalmazza a szükséges url-t

Ezeknek a szabályoknak betartásával egyszerűen tudunk az egyes felhasználókhoz menüt készíteni, ami pontosan csak azokat menüpontokat jeleníti, amihez a felhasználónak joga van.

A menü elkészítése a legjobb módszert igyekeztem felkutatni, amit végül is úgy oldottam meg, hogy készítettem egy menu modult, aminek az action osztálya az sfComponents osztályból származtattam. Ennek eredményeként egy úgynevezett komponenst kaptam, ami kinézetre megegyezik egy hagyományos action-el. A menu komponenst a layoutban jelenítem meg, attól függően, hogy van-e bejelentkezett felhasználó:

```
<?php if ($sf_user->isAuthenticated()): ?>
     <?php include_component('menu', 'index')?>
<?php endif; ?>
```

Az index action gyakorlatilag egy Doctrine utasításból áll, ami visszaad nekünk egy Doctrine Collection-t. Mivel a menu modul action osztálya egy komponens osztály, ezért nem a szokásos névadási konvenció jellemező a index action-höz tartozó template fájlra:

```
/**
*_index.php a fájl neve szemben a hagyományos
*indexSuccess.php névadási konvencióval
*/
<?php if (!$sf_user->isSuperAdmin()): ?>
     <ul>
          <li><h2><?php echo link_to('Adataim', $adataim)?>
          </h2></li>
     </ul>
<?php endif; ?>
<?php foreach ($group as $rekord): ?>
<ul>
     <li><h2><?php echo $rekord->name?></h2>
     <ul>

     <?php foreach ($rekord->permissions as $rekord2): ?>
          <li><?php echo link_to(
               $rekord2->name, $rekord2->description)?>
          </li>
     <?php endforeach; ?>
     </ul>
</ul>
<?php endforeach; ?>
<ul>
    <li>
          <h2><?php echo link_to(
               'Kilépés', '@sf_guard_signout')?></h2>
     </li>
</ul>
```

A komplett menü elkészítéséhez tehát mindössze erre a néhány sorra van szükség. Ez az egyszerűség köszönhető a keretrendszernek, az MVC pattern-ek és a Doctrine-nak. Mivel ez az első alkalom, hogy egy Doctrine-t is tartalmazó kódot illesztettem be, ezért egy rövid magyarázatot is adok a fenti kód értelmezéséhez

Az első sorban van egy feltétel, ami azt hivatott ellenőrizni, hogy a belépett felhasználó szuper adminisztrátor-e. A szuper adminisztrátor egy olyan felhasználó, akinek a egész rendszerhez van hozzáférése. Ha egy olyan felhasználó lép be, aki nem rendelkezik ezzel a tulajdonsággal, annak megjelenik a menüsorban az „Adataim” menüpont, ahol a belépett felhasználó a saját adatait tudja módosítani.

Ezt követi a menü további részének az elkészítése. A szükséges adatokat adatbázisból olvassuk ki. A Doctrine egy úgynevezett Doctrine collectiont ad vissza, aminek a bejárása egy foreach segítségével könnyen elvégezhető az ORP miatt. A $group változó értéket a komponensben határoztam meg, ami a felhasználóhoz tartozó csoportokat tartalmazza. A $rekord->permissions az adott csoporthoz tartozó jogokat tartalmazza. Ez a kifejezés egy objektumnak egy attribútumát jelöli. Ez az attribútum viszont ebben az esetben egy kapcsolat lesz, mégpedig az a kapcsolat, amit az sf_guard_group és az sf_guard_permissions között adtunk meg a setUp() metódusban. A permissions ebben a kapcsolatban egy alias, a kapcsolat neve. A háttérben a Doctrine elkészíti a két tábla összekapcsolásához szükséges SQL utasítást, ezzel is segítve a programozó munkáját.

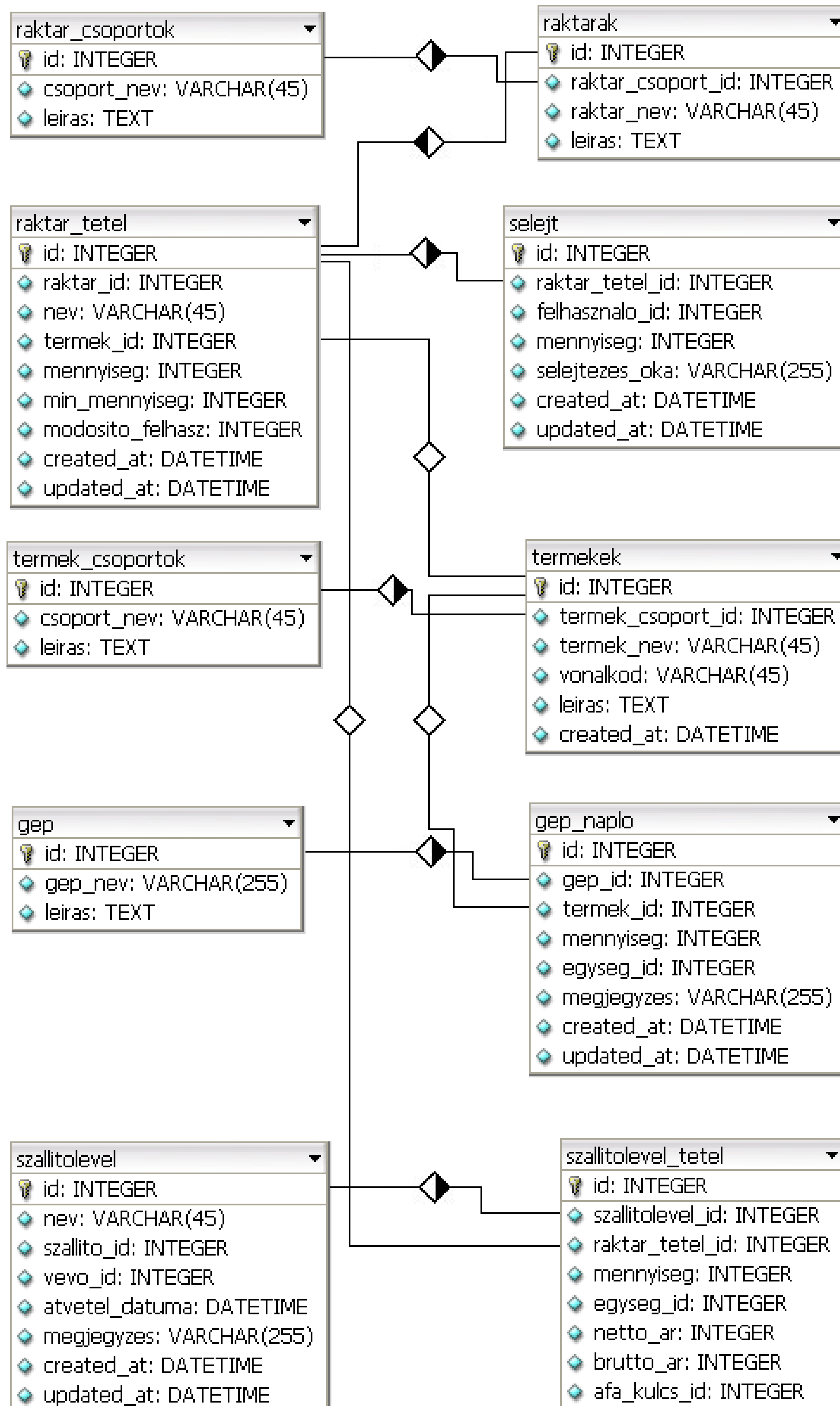
Ezt követően már csak a kilépéshez szükséges menüpont kell a menüsor végére illeszteni.

Az így kigenerált HTML listát ezt követően egy dropdown.css fájl segítségével formázom. A további CSS fájlokról valamint az alkalmazás kinézetéről az utolsó iterációban fogok részletes leírást adni.

Gyártás

A követelmény feltárás során már szóba került, hogy a megrendelő rendelkezik egy gyárral, amihez tartozik egy központi raktár és számos üzlet. Az igények között szerepelt, hogy minden üzlet kapcsolatban álljon a központi raktárral, és a raktár is az üzletekkel. A megrendelő távlati tervei között szerepel a vállalkozásának a bővítése, ami a gyár kibővítését, egy újabb raktár és több üzlet kiépítését takarja. Kérései között szerepel, hogy az elkészülő program az ezekhez hasonló változásokat zökkenőmentesen kövesse. Habár az üzleti logika még nincs teljes egészében megfogalmazva, viszont rendelkezésünkre áll annyi információ hogy egy prototípus elkészülhessen. Ebben az iterációban készítem el a gyártáshoz szükséges adattáblákat, modulokat, az alapvető action metódusokat, és a hozzájuk tartozó template fájlokat.

Gyártás adattáblái:



Az adattáblák összeállítása során arra törekedtem, hogy az egyes táblák a lehető legkevesebb mezőt tartalmazzák, és csak azokat az adatokat tárolja, ami logikailag az adott táblához tartozik. Nagy figyelmet fordítottam a csoportosításra is, melynek eredményeként egy struktúrát kaptunk, amiben a szűrés rendkívül meggyorsítja az egyes adtaelemek megkeresését. A szűrők használata egy igen fontos funkcionalitás egy olyan környezetben, ahol sok adattal kell dolgozni. Gondoljuk végig, hogy egy gyár esetén milyen sokféle terméket kell nyilvántartani. Az is előfordulhat, hogy nem csak azokat a cikkeket szeretnénk adatbázisban tárolni, melyek közvetlenül a gyártáshoz szükségesek, mint például alapanyagok vagy késztermékek, hanem olyanokat is ami az irodában dolgozó koordinációs feladatokat ellátó személyeknek szükséges.

Az elkészült modell természetesen tovább bővíthető újabb táblákkal, illetve az egyes adattáblák további mezőkkel. A bővíthetőség nagyon fontos tulajdonság adatmodell szinten. Egy új termék gyártása esetén előfordulhat, hogy szükség van arra, hogy specifikus információkat is tároljunk. Ennek következtében a termék táblát bővíteni kell. Szerencsére a Doctrine fel van készítve az ilyen szituációkra. Migráló kódok használatával, bővíthetjük az adattáblát anélkül, hogy adatveszteség következne be.

Űrlapok

A webes alkalmazásfejlesztés elképzelhetetlen formok használata nélkül. Az elkészül alkalmazás gyakorlatilag jól elkészített formok összessége. A Symfony készítői ezt pontosan tudják, és egy nagyon jó keretrendszert alkottak meg a programozók számára, melynek köszönhetően hatékonyan tudunk űrlapokat készíteni. A formok fontossága nem vitatott, ezért most röviden bemutatom, hogy hogyan lehet őket elkészíteni a Symfony keretrendszerben.

Konvenció szerint a form osztályok a lib/form/ könyvtárban találhatók. Igazság szerint az űrlapok bárhol tárolhatók, ahol a Symfony autoloading mechanizmusa megtalálja azokat, de később látni fogjuk, hogy a rendszer a lib/form/ könyvtárba hozza létre a modell objektumokból generált űrlapokat.

Egy form vagy más néven űrlap meghatározott mezők (rejtett, beviteli, szövegdoboz, lenyíló list, ...) összessége, melyeket a felhasználó tölt ki adatokkal. A Symfonyban az űrlap egy objektum, mely az sfForm osztályból származik. Az sfForm osztály minden űrlap ősosztálya, mely lehetővé teszi azok egyszerű kezelését és konfigurálást.

Widgetek

Egy widget egy űrlap mezőt ábrázol.

```
$this->setWidgets(array(
      'id'           => new sfWidgetFormInputHidden(),
      'termek_nev'   => new sfWidgetFormInputText(),
      'vonalkod'     => new sfWidgetFormInputText(),
      'leiras'       => new sfWidgetFormTextarea()
));
```

A setWidgets() metódussal lehet az űrlapon használt widgeteket definiálni. A setWidgets() metódus egy asszociatív tömböt vár, ahol a kulcsok jelölik az egyes mezők nevét, az értékek pedig a widget objektumokat. Minden widget egy objektum, melynek az sfWidget osztály az őse. Példánkban három típusú widgetet használunk:

- sfWidgetFormInputHidden: ez a widget egy rejtett mezőt ábrázol
- sfWidgetFormInputText: ez a widget egy szöveges input mezőt ábrázol
- sfWidgetFormTextarea: ez a widget egy szövegdobozt (textarea) ábrázol

További hasznos widgetek:

- sfWidgetFormInputPassword: ez a widget az sfWidgetFormInput osztályból származik. Jelszavakat tárolunk benne, melyet a böngészőnk fog elfedni a kíváncsi szemek elől.
- sfWidgetFormDoctrineChoice: ez a widget az sfWidgetFormChoice osztályból származik. Nagyon hasznos widget. Segítségével egy olyan select HTML kódot kapunk melynek option tagjai egy tábla rekordjait jeleníti meg paraméterezett formában. Ennek a widgetnek a konstruktora számos beállítási lehetőséget nyújt, melynek köszönhetően nagyon rugalmasan lehet az igényekhez igazítani. Néhány főbb paramétere:
    - multiple: értéke ha true, akkor egy multiple select HTML taget készítünk, ami annyiban különbözik a select tagtől, hogy több értéket is felvehet. Alapértelmezetten a multiple opció a false értéket veszi fel.
    - model: itt adom azt a modell osztályt, aminek a rekordjait akarom megjeleníteni (megadása kötelező)
    - add_empty: alapértelmezetten ennek az opciónak false az érétke. Viszont nem csak boolean értéket vehet fel, hanem text-et is, ebben az esetben a megadott

szöveg lesz alapértelmezetten kiválasztva a listában. Hasznos paraméter, ha a lista tartalmáról szeretnék információt adni a felhasználónak.

- method: megadható, hogy melyik metódust használjuk az objektumok értékeinek a megjelenítésére. Alapértelmezett metódus a `__toString()`. Az elkészült programban a szükséges model osztályokban felüldefiniáltam a `__toString()` metódust, mivel az alapértelmezetten a rekord id-át jelenítettem meg. Az én konvencióm, hogy a rekordhoz tartozó nev mezőket jelenítse meg. Ez alól kivételt képez az Afa model osztály `__toString()` metódusa, mert ott az áfa neve mellett feltüntetem az adott áfa kulcsot is.

Az elkészült alkalmazásban igen gyakran használtam ezt a típusú widgetet. Például a számla nyomtatásnál:

```
/**
*a method opciót megadása nem szükséges, mert az
*alapértelmezetten a __toString()
*/
$this->setWidgets(
     array('szamla' =>
      new sfWidgetFormDoctrineChoice(array(
     'model' => 'Szamla',
     'add_empty' => 'Válasszon egy számlát'))
     )
);

/**
*minden felhasználói csoporthoz több felhasználó és *több
jogosultság is tartozhat. A felületen ezt egy *multiple
select-tel tudjuk a legjobban kezelni
*/
$this->setWidgets(array(
     …
     'users_list'      =>
          new sfWidgetFormDoctrineChoice(array(
          'multiple' => true,
```

```
                'model' => 'sfGuardUser')),
        'permissions_list' =>
                new sfWidgetFormDoctrineChoice(array(
                'multiple' => true,
                'model' => 'sfGuardPermission'))
        )
);
```

Cimkék

Minden mezőhöz automatikusan létrejön egy címke. Alapbeállításként a címke nevet a mező nevéből képzi a rendszer a következő szabályok alapján:

- az első betű, nagy betű
- az aláhúzások cseréje szóközökkel.

Például: a termek_nev mezőhöz a Termek nev címkét képzi a Symfony. Habár az automatikus címke generálás nagyon hasznos, a keretrendszer megengedi a címkék testre szabását is a setLabels() metóduson keresztül:

```
$this->widgetSchema->setLabels(array(
        'termek_nev'    => 'Termek név',
        'vonalkod '     => 'Vonalkód',
        'leiras'        => 'Leírás',
));
```

Form megjelenítése

Habár az űrlap alapvetően HTML táblázatként jelenik meg, a layout megváltoztatható. A különböző layout formák osztályokban definiálhatók, melyek az sfWidgetFormSchemaFormatter osztály leszármazottai. Az űrlapok a táblázat formát (table) használják alapértemlezettként, ami az sfWidgetFormSchemaFormatterTable osztályban van meghatározva. Használhatunk akár lista formát (list) is:

```
$this->widgetSchema->setFormFormatterName('list');
```

Űrlapunk használatra készen áll. Már csak létre kell hozni egy modult, hogy megjelenítsük azt. Mikor létrehozzuk az űrlapot, a korábban definiált metódus automatikusan lefut. Már csak egy template-re van szükségünk, hogy megjelenjen űrlapunk.

A `<?php echo $form ?>` használata rendkívül hasznos űrlap prototípusok készítésénél. Lehetővé teszi, hogy a fejlesztő az üzleti logikára koncentráljon anélkül, hogy a megjelenés miatt keljen aggódnia. Mikor a `<?php echo $form ?>` formát használjuk megjelenítésre, a PHP a `$form` objektum szöveges alakját jeleníti meg. Az objektum szöveggé konvertálásakor a PHP a `__toString()` magic metódust próbálja meg futtatni. Minden widget megvalósítja ezt, hogy az objektum HTML alakját vissza tudja adni. A `<?php echo $form ?>` meghívása egyenértékű a `<?php echo $form->__toString() ?>` hívással.

XSS védelem (Cross-Site Scripting)

Amikor a Netscape elsőként bevezette a JavaScript nyelvet, már ők is sejtették, hogy ha a web szerver végrehajtható kódot küld át a böngészőnek, akkor az biztonsági kockázatot rejt magában. Az XSS a számítógépes sebezhetőség egy fajtája, amely tipikusan web alkalmazásoknál fordul elő: egy rosszindulatú web-felhasználó olyan kódot illeszt egy weblapra, amit más felhasználó is lát. Például ilyen kód lehet a HTML kód vagy egy kliens oldali script. Ha egy támadó egy XSS sebezhetőséget felfedez, azt - többek között - felhasználhatja arra, hogy a hozzáférési ellenőrzést kikerülje.

Mikor a widgetekhez attribútumokat állítunk be vagy alapértelmezett értékeket határozunk meg, az sfForm osztály automatikus védelmet biztosít az XSS támadás ellen a HTML kód generálásakor. Ez a védelem nem függ az escaping_strategy konfigurációs beállítástól (settings.yml). Ha a tartalom már le van védve valamely más eljárással, ez itt nem történik meg újra. Ugyancsak levédi a ' és " karaktereket, amelyek hibás generált HTML kódot eredményezhetnek.

Lássunk egy példát erre a védelemre:

```
$hello_word = new sfWidgetFormInput(array(), array(
      'value' => 'Hello "World!"',
      'class' => '<script>alert("XSS")</script>',
));

// Generált HTML
<input
      value="Hello &quot;World!&quot;"
      class="&lt;script&gt;alert(&quot;xss&quot;)&lt;/script
&gt;"
      type="text" name="contact[email]" id="contact_email"
```

```
/>
```

Űrlap érvényesítés – Validátorok

Egy Symfony űrlap mezők összessége. Minden egyes mezőt egyedi név azonosít, ahogy azt az fentebb láthattuk. Minden mezőhöz egy widgetet kapcsoltunk a megjelenésük sorrendjében. Most pedig lássuk, hogyan érvényesíthetjük a mezők tartalmát.

Minden egyes mező érvényesítéséért objektumok felelnek, amelyek az sfValidatorBase osztály leszármazottai. Azért, hogy érvényesíthessük egy űrlap adatait, minden egyes mezőhöz meg kell adnunk a megfelelő validátor objektumokat. Ezt az űrlap osztályban a setValidators() metódus segítségével tehetjük meg.

```
/**
* A termék raktár mezőinek a validálása
*/

$this->setValidators(array(
     'raktar_id'         => new sfValidatorDoctrineChoice(array(
          'model' => $this->getRelatedModelName('Raktarak'))),
     'nev'               => new sfValidatorString(array(
          'max_length' => 45)),
     'termek_id'         => new sfValidatorDoctrineChoice(array(
          'model' => $this->getRelatedModelName('Termekek'))),
     'mennyiseg'         => new sfValidatorInteger(),
     'min_mennyiseg'     => new sfValidatorInteger(),
     'modosito_felhasz' => new sfValidatorDoctrineChoice(array(
          'model' =>
                    $this->getRelatedModelName('sfGuardUser'))),
     'created_at'        => new sfValidatorDateTime(),
     'updated_at'        => new sfValidatorDateTime(),
));
```

Minden validátor az első paraméterben az opciók listáját fogadja. A widgetekhez hasonlóan bizonyos opciók kötelezőek, mások opcionálisak. Minden validátornak megadható még a required és a trim opció, amelyek az sfValidatorBase osztályban vannak definiálva:

- required:
    - alapértelmezett értéke: true
    - jelentése: a mező kötelezően kitöltendő
- trim:

    - alapértelmezett értéke: false
    - jelentése: automatikusan eltávolítja a whitespace karaktereket a sztring elejéről és végéről az érvényesítés előtt

Ebben a példában négy különböző validátort használunk:

- sfValidatorDoctrineChoice:
    - Felmerülhet a kérdés, hogy miért szükséges ellenőrizni a raktar_id, és termek_id mezőket? A select tag előre definiált értékekhez köti a felhasználót. Egy átlagos felhasználó valóban csak a listában szereplő értékek közül tud választani, ám más érték is elküldhető olyan eszközzel, mint például a Firefox Developer Toolbar, vagy tetszőleges kérést lehet szimulálni olyan eszközökkel, mint a curl vagy a wget.
- sfValidatorString:
    - Egy sztringet érvényesít. Ahogy a példában is látható ennek a validátornak megadhatunk kiegészítő opciót, ami jelen esetben egy max_length. A max_length segítségével tudom ellenőrizni, hogy a felhasználó által megadott nevet csonkítás nélkül le tudom menteni az adatbázisba. További kiegészítő opció még min_length. Használatával lehetőségem van rákényszeríteni a felhasználót, hogy ne adjon meg túl rövid nevet.
- sfValidatorInteger:
    - Egy integer érvényesítést végzi. Kiegészítő opciói a min és a max, melyek használatával egy intervallumot tudok megadni
- sfValidatorDateTime:
    - Dátumokat érvényesít. A validátor többféle bemenetet fogadhat (egy időbélyeget, egy regexp-en alapuló formátumot, ...). A bemeneti érték egyszerű visszaadása helyett azt a Y-m-d H:i:s formátumra alakítja át. Ez garantálja a fejlesztő számára az egységes formátumot, függetlenül az eredeti érték formájától. A rendszer ezáltal a felhasználó számára nagyfokú rugalmasságot tesz lehetővé, míg a fejlesztő számára biztosítja a konzisztens adatokat.

Validátorok testreszabása

Abban a formában, ahogy fentebb látható, a termék raktár űrlapnak a hibaüzenetei nem túl hasznosak. Lássuk, hogyan tehetjük őket még intuitívabbá.

Minden validátor hibákat adhat az űrlaphoz. Egy hiba egy hibakódból és egy hibaüzenetből áll. Minden validátor rendelkezik legalább a required és az invalid hibával, amelyek az sfValidatorBaseben vannak definiálva. A hibaüzenetek a validátor objektumoknak átadott második argumentumban adhatók meg. A raktár tétel példánál maradva:

```
$this->setValidators(array(
      //megelőző validátorok
      'nev' => new sfValidatorString(array(
          'min_length' => 4), array(
          'required'   => 'A termék raktárnak nevet kell adni',
          'min_length' => 'A név: "%value%" túl rövid.
               Minimum %min_length% hosszúságúnak kell lennie.'
      )),
     //ezt követő validátorok
    ));
```

A fenti kódrészletben két hibakódot is találhatunk. Az első hibakód a ’required’, ami azt hivatott figyelni, hogy az adott mező kötelezően kitöltendő és az értéke üres. A Symfony lehetőséget nyújt a fejlesztő számára, hogy dinamikus értékeket használjon a hibaüzenetekben. A ’min_length’ hibakódhoz tartozó hibaüzenet különbözik attól, amit már eddig bemutattam, mivel ez két dinamikus értéket is tartalmaz:

- a felhasználó által megadott nevet:
- a mezőhöz meghatározott minimális karakterszámot (4).

Minden hibaüzenetben használhatók dinamikus értékek, ehhez az érték nevét kell megadni százalékjelek (%) közé zárva. Az elérhető értékek általában a felhasználói adat (value) és a hibával kapcsolatosan a validátornak megadott opció neve.

Alapértelmezetten egy űrlap csak akkor érvényes, ha minden elküldött mezőhöz van hozzárendelt validátor. Ez biztosítja, hogy minden mezőhöz van érvényesítő szabály és nem lehet olyan mezőkhöz értéket megadni, amik nincsenek előre meghatározva az űrlapon. Ez a megközelítés egyfajta biztonságot generál, hiszen ha mindenféle védelem nélkül a felhasználó elküld egy űrlapot, akkor a kódunk sebezhető volna.

Űrlapok küldése

Miután elkészült az űrlapunk nincs más teendő, mint feldolgozás céljából elküldjük azt egy action metódusnak. A feldolgozást végző metódust az űrlap megjelenését szolgáló template fájlban az url_for() helper segítségével könnyen megadható. Több módszer is létezik, hogy miként is dolgozzunk fel egy elküldött űrlapot. Tapasztalatom szerint a legjobb megoldás erre a feladatra, ha ugyanabban a metódusba végzem a feldolgozást, mint amivel az űrlapot megjelenítem, ugyanis miután az űrlap megjelenítéshez GET kérést, az elküldéshez POST kérést alkalmaztunk, ezért egy action-ben problémamentesen le lehet kezelni a megjelenítést és a feldolgozást is. Egy ilyen metódus működését tekintve a következő egységekre bontható:

- GET kérés esetén az űrlapot inicializáljuk, majd átadjuk a template-nek a megjelenítéshez. Ekkor az űrlap kezdő állapotban van (initial state)
- Mikor a felhasználó elküldi az adatokat egy POST kéréssel, a bind() metódus köti össze az elküldött adatokat az űrlappal és elindítja az adatok érvényesítését. Az űrlap ekkor kötött állapotba kerül (bound state).
- Mikor az űrlap kötött állapotba került lehetőségünk van az űrlap érvényességének ellenőrzésére az isValid() metódussal
  - Ha a visszatérési értéke true, akkor az űrlap érvényes és a felhasználó átirányítható a megfelelő oldalra
  - Ha a visszatérési értéke false, akkor az űrlap érvénytelen. Ebben az esetben a hibás mezőknél megjelennek azok a hibaüzenetek, amiket a validátoroknál adtunk meg.
  - Mikor az űrlap kezdő állapotban van, az isValid() metódus mindig false értékkel tér vissza, és a getValues() metódus üres tömböt fog visszaadni

Minden validátor két feladatot lát el:

- egy érvényesítő feladatot (validation task)
- és egy tisztító feladatot (cleaning task).

A getValues() metódus az érvényesített és tisztított adatokkal tér vissza, ezért használata erősen javaslott minden űrlap feldolgozása során.

A tisztító folyamat két fő tevékenységből áll:

- a bejövő adatok normalizálása
- és konvertálása.

Az adat normalizálást a trim opció segítségével eltávolítja a whitespace karaktereket a sztring elejéről és végéről az érvényesítés előtt. A normalizálás ennél persze sokkal fontosabb például dátumok esetén. A normalizálás garantálja a fejlesztő számára az egységes formátumot, függetlenül az eredeti érték formájától. A rendszer ezáltal a felhasználó számára nagyfokú rugalmasságot tesz lehetővé, míg a fejlesztő számára biztosítja a konzisztens adatokat. A getValues() metódus egy tömbbel tér vissza, ami az érvényesített és tisztított adatokat tartalmazza. Ha csak egy mező értékére van szüksége érdemes a getValue() metódust használni

Javaslom, hogy a felhasználót POST kérés után mindig továbbirányítjuk:

- Ezzel megakadályozzuk az űrlap újra küldését, ha a felhasználó frissítené az oldalt.
- A felhasználó tudja használni a Vissza gombot anélkül, hogy egy felugró ablakot kapna, hogy küldje el az űrlapot újra.

Összegezve a fentebb leírtakat belátható, hogy egy jól elkészített űrlap milyen sok időt képes felemészteni a fejlesztőtől. Szerencsére a Symfony keretrendszert egy jól felépített produktum. Lehetőséget nyújt a fejlesztőknek, hogy automatizálja az űrlap készítésének a mechanizmusát. Parancssorból kiadva a következő utasítást:

```
symfony doctrine:build-forms
```

A keretrendszer minden táblához készít egy olyan form osztályt, amely tartalmazz a szükséges widgeteket és validátorokat. A generálás figyelembe veszi az egyes táblák közötti kapcsolatokat is.

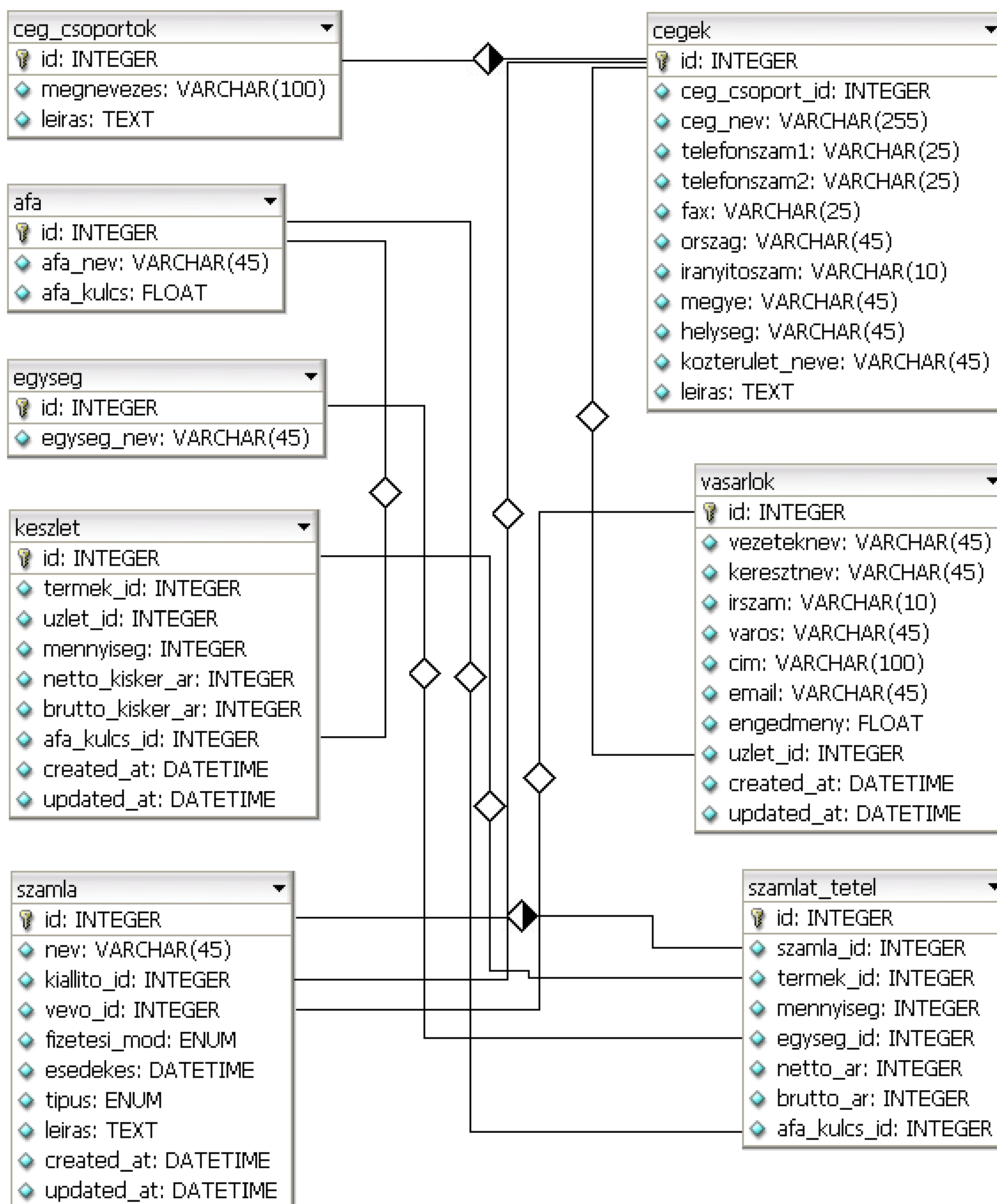
Értékesítés

Ebben az iterációban a legyártott termékek értékesítéséhez szükséges modulokat készítem el. Az értékesítés iteráció megtervezése során fontos szempont volt a rugalmasság, és a nyomon követhetőség. A nyomon követhetőség elsősorban a készlet szempontjából volt fontos. Számos alkalommal gondoltam át, hogy milyen adattáblában, milyen módon lehessen egy adott üzlethez tartozó készletet a legjobban kezelni. Az egyik fő kérdés az volt, hogy ha

egy termék már szerepel a készletben, és egy újabb mennyiséget szeretnénk készletre felvinni, ugyanabból a termékből, akkor a meglévő készlet mennyiségét növeljem, vagy egy új rekordod készítsek a készlet táblába. Ha a meglévő készlet mennyiséget növelem, akkor a készlet adattábla kevesebb rekordot fog tartalmazni, ezáltal valószínűleg az adatok elérési sebessége csökken. Viszont ennek a megoldásnak van egy komoly hátránya, mégpedig az, hogy így nem fogom tudni nyomon követni a készlet áramlását. Ez pedig a későbbiekben a szoftver életciklusa során jelentős problémát okozhat. Az előnyök és hátrányok mérlegelését követően úgy döntöttem, hogy a második módszert implementálom le.

Egy másik kulcsfontosságú kérdéssel is meg kellett birkóznom még mielőtt a elkezdtem volna controllereket gyártani, ez pedig az értékesítéshez kapcsolódó modell felépítése. A megrendelő több üzlettel is rendelkezik. A feladat egy olyan alkalmazást kell készíteni, amit minden üzlet tud használni. A jelenlegi követelmények alapján egy üzlethez megközelítőleg 6-7 adattáblára van szükség. Felvetődik a kérdés, hogy ezt a néhány táblát külön adatbázisban tároljam-e le vagy minden üzlet ugyanazt a táblát használja. Ebben a kérdésben véleményem szerint jobb megoldás a külön adatbázis használta. Az egyes üzletek adatai nagyobb biztonságban vannak, ha csak ők manipulálják, másfelől az üzleti logikai is egyszerűen kivitelezhető, hiszen nem kell folyamatos kontrollt fenntartani, az adatok megjelenítése és módosítása esetén. A dolgozatban viszont nem ezt a megoldását kódoltam le. Ennek oka nagyon egyszerűen magyarázható. A dolgozat keretében elkészült webes alkalmazás egy prototípus, ami éles környezetben soha nem fogunk használni, viszont arra kiválóan alkalmas, hogy a bemutatása után pontosabb, részletesebb követelményrendszert lehessen felállítani, és támpontként szolgáljon az első kiadás elkészítéséhez.

Értékesítés adattáblái:



Az elkészült adattáblákat böngészve talán egy kicsit kaotikusnak tűnik a táblák közötti kapcsolatok. Ha egy kis idődő rászánunk a fenti ábra értelmezésére, akkor véleményem szerint egy jól átgondolt és letisztult sémát látunk. Az adatbázis készítés során arra törekedtem, hogy az adat modellemet a gyakorlatban is elvárt 3NF-ra hozzam.

Az értékesítés folyamata igen összetett, és nehéz olyan üzleti logikát megvalósítani, ami átlátható, és könnyen bővíthető. Az MVC minta nekem sokat segített az értékesítéshez tartozó modell tervezése során, hiszen egy időpillanatban csak egy problémával kellett foglalkoznom.

A megrendelő a követelmények feltárása során kihangsúlyozta, hogy az értékesítés nem csak a saját üzletekben magánszemélyek felé történik, hanem közvetlenül a gyártól is rendelhetnek úgynevezett partner cégek. Ezek a cégek beszerzői vagy nagyker áron jutnak a megrendelt termékekre. Ennek tükrében készítettem két táblát is. A ceg_csoportok tábla nagy segítséget nyújt a cégek szűréséhez. Ahogy az fenti ábrán is látható, minden cég egy csoporthoz tartozik, és egy csoportba több cég is besorolható.

Kulcsfontosságú a készlet tábla ebben az iterációban. Az üzleti logika megköveteli a szoftver felhasználójától, hogy csak a készleten lévő termékeket értékesítheti. Ennek a megszorításnak köszönhetően számlát is csak azokról a termékekről szabad kiállítani, ami szerepel az adott üzlethez tartozó készletek között.

Számla készítése

Ebben az iterációban a hangsúlyt a hatékony számlakészítésre helyeztem. A hatékony szó alatt a gyorsaságot és az adatok érvényességének a fontosságát értem. Ennek legjobb módja, ha a lehető legtöbb adatot az adatbázisból olvasom ki, és az űrlap elküldést megelőzően egy validátoron engedem keresztül. Mindezt felületen legördülő listákkal oldottam meg, melynek készítéséről az előző iterációban olvashattak. A kitűzött cél megvalósítása azonban modell szintem nagyobb feladat volt, mint a megjelenítés. A modell készítése során arra törekedtem, hogy minden ismétlődő adatot külön táblába emeljem ki, és a számla illetve a számlákhoz tartozó tételek letárolása során csak a megfelelő id-at tároljam le ezekben a táblákban. Ennek tükrében készítettem külön táblát az áfának illetve az egységeknek. Ezek olyan ismétlődő adatok, amiket mindenképp érdemes külön táblában tárolni. Mindezek után jött a gondolat, hogy a számlához tartozó tételek felvételét tovább lehet gyorsítani, ha készítek egy olyan submit gombot, ami a mentést követően lehetőséget ad újabb tétel felvitelére.

A kitűzött cél nem kis munka árán végül sikerül elérnem:

Új számla tétel felvétele

Számla	Első számla
Termék	Nive szék - pácolt bükk - krém
Mennyiség	
Egység	db
Nettó ár	
Bruttó ár	
Áfa kulcs	Kisker áfa (25.00%)

Vissza a számla tételek listához Mentés Ment és felvesz

Számla nyomtatása

A felhasználói követelmények között szerepelt, hogy az elkészülő alkalmazás lehetőséget biztosítson az elkészült számla nyomtatására. Habár a jelenlegi követelményrendszer csak a számla nyomtatást foglalja magában, az alkalmazás készítése alatt arra törekedtem, hogy ha később újabb nyomtatási igény merül fel, akkor azt könnyen lehessen orvosolni. Ezt úgy értem el, hogy egy külön modult szántam erre a problémára. Minden nyomtatási igényt egy-egy action metódus fog kielégíteni. Jelen esetben csak egy ilyen action van, ami a NyomtatForm.class.php fájlban definiált űrlap egy példányát állítja elő.

TCPDF

A TCPDF egy PHP nyelven íródott nyílt forráskódú pdf fájl generátor, mely könnyen integrálható a Symfony keretrendszerbe. Ez a plugin OOP felépítésű. Rengeteg hasznos eszközt nyújt a könnyű, gyors és egyszerű használhatósághoz. Nyelvi támogatottsága teljes. Támogatja a jobbról balra való írásmódot is, valamint könnyedén illeszthetünk dokumentumainkba képeket és linkeket. Továbbá egységes felületet ad a testre szabható fejléc és lábléc. Kényelmi funkciói közzé tartozik az automatikus oldalszámozás és többféle szabvány szerinti vonalkód készítés is. A dokumentum elkészítésének több módja is

támogatott. Kezelhetjük a dokumentumot, mint vásznat, ebben az esetben koordináta rendszerként kell tekinteni a dokumentumra, de kényelmi funkció a HTML tartalom pdf-be konvertálása. Munkámban az utóbbi módszert választottam, vagyis HTML táblázatok segítségével rendeztem a tartalmat. A TCPDF plugin azonban széleskörű CSS támogatást is nyújt, abban az esetben, ha a HTML tartalomba inline CSS-ket használunk. A plugin-nak ez a támogatottsága nem teljes körű. Az alkalmazás fejlesztés során szerzett tapasztalataim szerint azonban a CSS 1.0 szabványt támogatja. További kényelmi funkció, hogy a generált fájlok neveit is megválaszthatjuk, ezzel is megkönnyítve az alkalmazásom használóinak a generált számlák rendszerezését. Bár ez a plugin nagyon hasznos és kényelmes a használata, de tapasztalataim szerint a hibakezelés sok esetben „szűkszavú". Ha a HTML tartalom hibás, ami a fejlesztés során könnyen előfordulhat, akkor nem ad hibaüzenetet csak üres oldalt generál. További negatívum, hogy csak a FreeSerif betűtípus az, ami támogatja a magyar abc összes betűjét. Mivel nem ez a betűtípus az alapértelmezett sok időt vett igényben ennek a kiderítése. Pozitívuma ennek a plugin-nak a rendszerben nem lévő betűtípusok integrálhatósága. Ha letöltünk egy új *.ttf fájlt, akkor egy parancssorból futatható php programkód automatikusan beintegrálja azt a TCPDF plugin-ba. Ezzel biztosítva a dokumentum egyedi megjelenését. Az egyedi megjelenést nem csak egyedi betűtípussal, hanem képekkel is hangsúlyozhatjuk. Lehetőségünk van a pdf dokumentum fejrészébe illetve a dokumentum törzsbe is képeket illeszteni, mivel támogatja a weben legnépszerűbb képformátumokat, mint png, gif, jpg.

PDF generálás

Mivel a TCPDF integrálva van a keretrendszerbe így a Symfony konvenciói szerint action-ként futtathatjuk a executeSzamla(sfWebRequest $request) metódusunkat. Ez a metódus BasesfTCPDFActions osztályban implementáltam, ami az sfActions ősosztályból származik. Mikor meg hívom az executeSzamla(sfWebRequest $request) metódust, akkor paraméterként megkapja a NyomtatForm-ban kiválasztott számla id értékét. Összesen erre az egy értékre van szükség, hogy az adatbázisból az adott számlához tartozó összes információ kinyerhető legyen. Az adatok adatbázisból történő kiolvasásához szükséges metódusok egy részét a Doctrine nyújtja nekünk, viszont mint ebben az esetben, szükség lehet általunk készített, speciális is igényeket kielégítő metódusokra is. A Doctrine minden tábla szintű osztályhoz nyújt két metódust a find() és finAll() metódusokat. A find() metódus egy adott id-val rendelkező rekord objektumot ad vissza, míg a findAll() egy tábla összes rekordját

tartalmazó Doctrine Collection-el tér vissza. A find() metódus kitűnően alkalmazható, hogy a paraméterként megkapott id-hoz tartozó számlát az adatbázisból kinyerjük:

```
$szamla = Doctrine::getTable('Szamla')->find($request-
>getParameter('szamla'));
```

Az ehhez a számlához tartozó számla tételek lekérdezésére a Doctrine már nem nyújt kész metódust, de lehetőség van saját metódus elhelyezésére a SzamlaTetelTable.class.php osztályban:

```
public function findSzamlaTetelBySzamlaId ($id) {

     $szamla_tetel = Doctrine_Query::create()
           ->select('s.*')
           ->from('SzamlaTetel s')
           ->where('s.szamla_id = ?', $id))
           ->execute();

}
```

Ez a metódus a paraméterében megadott id-hoz tartozó összes számla tételt adja vissza egy Doctrine Collection-ben. A kapott collection-t egy foreach segítségével könnyen bejárható. Ilyen foreach ciklust a dinamikus menü bemutatása során elemeztem.

Az adatok beolvasását követően a következő lépés a konfigurációs fájlok betöltése, ami a következő sor végzi a metódusban:

```
$config = sfTCPDFPluginConfigHandler::loadConfig();
```

A konfigurációs beállítások a pdf_configs.yml fájlban találhatóak. A fenti kód segítségével automatikusan betöltjük ennek a tartalmát a $config változóba. Ezzel viszont még nem tudjuk használni a fenti fájlban lévő beállításokat. Ezeket a TCDPF osztály példányosítása után különféle set metódusokkal tudjuk alkalmazni, mit például a következőek:

```
$pdf->SetFont("FreeSerif", "", 12);
$pdf->SetMargins(
     PDF_MARGIN_LEFT, PDF_MARGIN_TOP, PDF_MARGIN_RIGHT
     );
$pdf->SetHeaderData(
     PDF_HEADER_LOGO, PDF_HEADER_LOGO_WIDTH
     );
```

A beállításokat követően az AddPage() metódussal tudunk egy üres pdf dokumentumot létre hozni. Ezt követően tudjuk feltölteni a kívánt tartalommal. Korábban említettem, hogy kezelhetjük a dokumentumot úgy, mint vásznat, vagy hozzáadhatunk egy HTML tag-eket tartalmazó szöveget is, és ezt konvertáljuk pdf-be. Munkámban az utóbbi módszert választottam, vagyis HTML táblázatok segítségével rendeztem a tartalmat.

```
// számla adatok (esedékesség, fizetési mód, stb...)
    $htmlcontent='
        <table>
            <thead>
                <tr>
                    <th><b>Fizetési mód</b></th>
                    <th><b>Teljesítés időpontja</b></th>
                    <th><b>Számla kelte</b></th>
                    <th><b>Esedékesség</b></th>
                </tr>
            </thead>
            <tbody>
                <tr>
                    <td>'.$fizetesi_mod.'</td>
                    <td>'.$teljesites_idopontja.'</td>
                    <td>'.$szamla_kelte.'</td>
                    <td>'.$esedekesseg.'</td>
                </tr>
            </tbody>
        <table>
        <hr />
    ';
    $pdf->writeHTML($htmlcontent , true, 0);
```

A generált pdf fájl több HTML táblázatból tevődik össze. Az egyes táblázatokat a $htmlcontent változóban konkatenálom össze a dinamikusan változó adatokkal. Minden elkészült táblázat után a meghívom $pdf objektum writeHTML() metódusát. Ez a metódus végzi az tartalom hozzáadását a $pdf objektumhoz. Ez a metódus három kötelező paramétert vár:

- az első a HTML tartalom
- a második egy boolean, ami true esetén sortörést helyez el a pdf dokumentumban
- a harmadik paraméter pedig a háttérszínért felelős

Miután minden kívánt tartalmat hozzáadtuk a $pdf objektumhoz a következő sor elküldi a böngészőnek a kész pdf dokumentumot a beállított néven és módon.

```
$pdf->Output('szamla.pdf', 'I');
```

A fenti metódus paraméterezésére érdemes egy kis figyelmet fordítani:

- az első paraméter a fájl neve kiterjesztéssel együtt.
- a második paraméter mondja meg, hogy milyen módon adjuk át a fájlt a böngészőnek. Négy lehetséges mód áll a rendelkezésünkre 'I', 'D', 'F', 'S' Ezek közül az alábbiakat az 'I' és a 'D' módokat javaslom.
- Az 'I' közvetlenül a böngészőben jeleníti meg a pdf dokumentumot, amennyiben az rendelkezik pdf olvasó pluguin-nal.
- A 'D' felkínálja olvasásra illetve mentésre a dokumentumot.

Az executeSzamla(sfWebRequest $request) metódus miután sikeresen lefutott a következő szamla.pdf dokumentumot állítja elő:

WEBRAKTÁR
RAKTÁRKEZELÉS AZ INTERNETEN

Számla

Kiállító	**Vevő**
Kerepes úti üztelünk	**Faragó Attila**
1119, Budapest	4028, Debrecen
Kerepes út 23.	Geresi u. 13

Fizetési mód	**Teljesítés időpontja**	**Számla kelte**	**Esedékesség**
átutalás	2010-04-26 12:16:00	2010-04-26 12:16:00	2010-04-30 10:00:00

Megnevezés	**Mennyiség**	**M. egység**	**Egységár**	**Nettó**	**Áfa %**	**Áfaérték**	**Bruttó**
DIN 912 ISO 4762 PA 6.6	50	db	20	1000	25.00	250	1250
Nive szék - pácolt bükk - krém	5	db	20000	100000	25.00	25000	125000
Lugano dohanyzóasztal	1	db	40000	40000	20.00	8000	48000

Összesen	**Nettó**	**Áfaérték**	**Bruttó**
	141000	33250	174250

Fizetendő végösszeg: 174250

1/1

Felület

A Symfony 1.4 keretrendszer kitűnő lehetőséget ad a modern MVC minta megvalósítására. Ez a rendező elv azonban már a korábbi verziókban is megfigyelhető volt. Az 1.0-ás keretrendszerben is nagy hangsúlyt kapott a fenti MVC pattern.

A fentiekből tehát következik, hogy a rendszer a lehető legteljesebb mértékben igyekszik elválasztani egymástól a PHP, HTML, CSS, és JavaScript kódokat.
Megvalósítása a következő:

- Az adott action-ben meghívjuk a setTemplate() metódust, amely paraméterként egy template fájl nevét várja, például: setTemplate('index'). A template file-ok elnevezésénél konvenció, hogy mindig Succes.php-ra kell végződniük, tehát az index action-höz tartozó template nevének így kell kinéznie: indexSucces.php. Még egy megkötés, hogy a template neve kisbetűs, kivéve a Succes szót kell nagybetűvel kezdenünk.
- A template file-t php nyelven írjuk. A Symfony-nak nincs saját template nyelve, mint például a Smarty template kezelőnek. Az egyes változókhoz csak az a template fájl férhet hozzá, amelyik action meghívta azt, ezzel is minimalizálva a változó nevek ütközésének lehetőségét. Az egyes változókra a template file-ban a következőképpen hivatkozhatunk: $user. Természetesen feltéte, hogy a $user változó létezzen és lehetőleg értéket is kapjon az action-ben, mely meghívta a template-et.

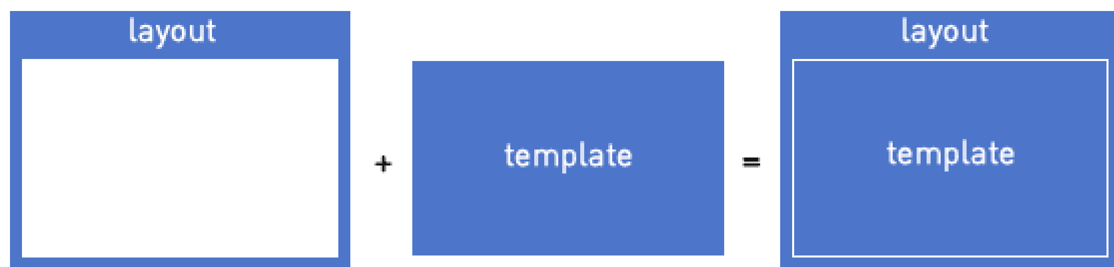
A Symfony keretrendszer lehetőséget ad arra is, hogy minden egyes template file-hoz rendelhessünk saját CSS és JavaScript file-t is. Használata igen egyszerű és logikus is, hiszen egy felhasználónak nem szükséges letöltenie az adminisztrátor felületéhez tartozó CSS vagy Javascript fájlokat. Ezzel is csökkentve a szerver terhelését, ezáltal gyorsabban töltődik le a lap a kliens gépére

Layout

A megjelenítés többféleképpen megvalósítható. Kevésbé elegáns módszer az, hogy a HTML-t logikailag összetartozó részek mentén „szétvágjuk”, és ezeket include-juk egy kész template-té. Előnye ugyan a kód duplikáció elkerülése, de nagy hátránya, hogy nem validak az egyes template részek, mivel hiányoznak a megfelelő HTML tag-ek, így nagy odafigyelést

igényel, hogy megfelelő sorrendben következzenek az egyes template file-ok a HTML oldal összeállításában.

Ez az ábra szemlélteti a fenti elképzelést, amely azonban már elavult. A keretrendszer egy elegánsabb megoldást kínál a probléma áthidalására. A Symfony megoldása az, hogy egy globális template-et készítünk, amit layout-nak nevezünk. Ezt a layout-ot alkalmazás szintjén is el tudjuk készíteni, amit az apps/alkalmazás/templates mappában találunk. Ez a könyvtár tartalmazza az összes globális template-et az alkalmazásunk számára.



A fenti ábra mutatja a hagyományos megoldás hátrányait kiküszöbölő megoldást, melyet a keretrendszer alkalmaz. Lényege, hogy a HTML oldal statikus részeit a layout template tartalmazza. A megjelenítés mechanizmusa, hogy először a dinamikus tartalmat illeszti a layout-hoz, és az így kapott valid HTML oldalt küldi el a böngészőnek. Az alkalmazásom layout-ja az alábbi főbb részeket emeltem ki:

- logo megjelenítése
- menüsáv megjelenítése
- dinamikus tartalom megjelenítése:

```
<div id="content"><!-- content -->
    <?php echo $sf_content ?>
</div><!-- content vege -->
```

- lábléc megjelenítése

CSS és JavaScript használata

A CSS és JS file-ok használata a következő módon történik: Az egyes template file-okban meghívjuk a következő metódusokat, melyek paraméterként a használni kívánt CSS, vagy js file nevét várják:

```
use_stylesheet();
use_javascript();
```

Ezzel a két metódussal jelezzük, hogy a paraméterükben szereplő file-okat include-olni akarjuk a HTML tartalomba, amelyért a következő két metódus felel:

```
include_stylesheets();
include_javascripts();
```

Az alkalmazáshoz két css file-t készítettem. A style.css-t és dropdown.css-t. A style.css az alapvető beállításokat tartalmaz, melyek a lap szerkezetét írják le. Az egyes sávokat, mint a tartalom és az oldalsáv szélességét, valamint az egész lapra érvényes design elemeket írja le. Fontosnak tartottam a böngésző független kivitelezést. Ezért több böngésző alatt is teszteltem a megjelenést például: Internet Explorer 7 és 8, Firefox 2 és 3, Opera. Mivel az Internet Explorer 6 egy, már a Microsoft által sem támogatott böngésző, így én sem készítettem fel az alkalmazásomat az ezzel való megjelenítésre. Alapvetően használható a lap ezzel a böngészővel is, de a legördülő menü a CSS támogatottsága miatt nem megfelelően működik. Ennek kijavítása IE6 alatt legkönnyebben JavaScript felhasználásával lehetséges, de a fenti ok miatt ez nem lett megvalósítva. Az egységes megjelenítés érdekében a style.css file első néhány sora általam választott értékekre állítja a böngésző beépített stílusait, többek között a margókat, színeket, betűtípusokat, behúzásokat stb. Ennek célja az egyes böngészők eltérő alapbeállításaiból adódó kellemetlenségek kiküszöbölése.

Tesztelés

A mai alkalmazásfejlesztés szerves részét képzi a különböző tesztek megírása. Ennek ellenére sajnos nagyon sok olyan webfejlesztő van, aki a megírt kódhoz nem készít futtatható teszteket. A tesztek elkészítése nem mondható triviális feladatnak. Ezt a tényt minden fejlesztő megállapítja, mikor úgy dönt, hogy teszteket kezd írni. Tényként kezelhető, hogy a fejlesztők nem szeretnek sem teszteket, sem pedig dokumentációt készíteni. Azonban minden olyan informatikus, aki fejlesztőként szeretne dolgozni, idővel belátja majd, hogy ezen eszközök szerves részét képzik fejlesztésnek.

A tesztek készítésére szánt idő, a szoftver életciklusa alatt kifizetődik. Gondoljuk végig az alábbi szituációt. Egy meglévő rendszerbe kell egy új apró változtatást elvégeznünk. Rövid időn belül el is készülünk és kiadjuk a változást a felhasználóknak. Ezt követően érkeznek a visszajelzések a felhasználóktól, hogy ilyen meg olyan problémájuk van, amiket megvizsgálva ráébredünk, hogy a nemrégiben elkészült változtatás bizony kihat a szoftver egyéb, eddig megfelelően működő részeire is. Lázas debugolásba kezdünk, hogy ezt a problémát mielőbb kijavítsuk, hiszen nem szeretnénk rossz hírnevet szerezni, vagy netán elveszíteni az ügyfeleinket. Az ezekhez hasonló kellemetlenségeket tesztek írásával elkerülhetjük, továbbá biztonságérzetet is ad, hogy az elkészült szoftver megfelel a felhasználói igényeknek.

Azok a fejlesztők, akik eddig nem úgy készítették termékeiket, hogy nem írtak hozzájuk teszteket, azoknak sincs minden veszve, hiszen ma már több olyan keretrendszer van, ami kényelmes felületet nyújt tesztek készítésére, mint például a PHPUnit, vagy a Selenium. Érdemes elsőnek a meglévő bugokra teszteket írni. Ezt követően, ha minden új funkcionalitás elkészítése után vagy akár előtte megírjuk a hozzá tartozó teszteket, akkor a kódunk egyre nagyobb részét fedjük le tesztekkel. Idővel hozzászokunk, hogy teszteket kell írni, melynek eredményeként színvonalasabb programot fogunk kapni.

A tervezés során is már számos nagyszerű tulajdonságát soroltam fel a Symfony-nak, és szerencsére a keretrendszer a teszteléshez is nagyon sok segítséget ad. A Symfonyban két fajta tesztet különböztetünk meg:

- unit teszt: arra hivatott, hogy letesztelje, hogy minden metódus és függvény működését, eredményét megvizsgálja.

- funkcionális teszt: arra hivatott, hogy az elkészült alkalmazásunk működését tesztelni tudjuk az „elejétől" a „végéig" a kéréstől a válaszig. Gyakorlatilag az alkalmazás minden rétegét tudjuk vele tesztelni, így a felületet is.

Unit teszt

A Symfony keretrendszer a unit teszteket készítésére egy beépített tesztelő keretrendszer nyújt, amit lime-nak nevezünk. Minden unit teszt ebben a keretrendszerben a következő utasítással kezdődik:

```
require_once dirname(__FILE__).'/../bootstrap/unit.php';
```

A fenti programkód behúzza a unit.php fájt, ezzel elvégzi a szükséges inicializálást. Ezt követően példányosítást útján létrejön egy új lime_test objektumunk.

```
$t = new lime_test(1);
```

A konstruktor paraméterében tudjuk megadni, hogy hány tesztet tervezünk futtatni. Ezeknek a testeknek nagyon egyszerű működési elvük van. A teszteléskor meghívjuk az egyes metódusokat egy előre definiált adathalmazon és az erre adott eredményt fogjuk összehasonlítani az elvárt eredménnyel. Az hogy egy teszt „átment" vagy „elbukott" azt ennek a hasonlításnak az eredménye dönti el. A lime keretrendszer beépített metódusokat biztosít adat értékek és adat tömbök különböző szempontok szerinti összehasonlítására.

A Symfony lehetőséget nyújt a modell tesztelésre, ami részben köszönhető a Doctrine-nak is. Az ilyen tesztek elkészítésére javaslott egy új, csak a tesztelésre fenntartott adatbázis elkészítése. Működési elve ezeknek a teszteknek megegyezi a fentebb írtakkal, azzal a lényeges különbséggel, hogy itt egy modell-hez tartozó metódust tesztelünk. Ilyen lehet például a számla nyomtatásnál bemutatott findSzamlaTetelBySzamlaId() metódus. A teszt adatok a /test/fixtures mappába található, melyet a loadData() metódussal tudunk betölteni.

Minden unit tesztek a /test/unit/ mappában hozunk létre. Névadási konvenció a teszt fájloknak a következő: az osztály nevét a 'Test' szó követi, mint pl: SzamlaTetelTest.php. A teszteket parancssorból könnyen tudjuk futatni. A symfony test:unit SzamlaTetelTest a SzamlaTetelTest.php fájlban található teszteket futtatja. A symfony test:unit paranccsal pedig az összes unit tesztet tudjuk futtatni.

A keretrendszer rugalmasságának egy újabb bizonyítéka, hogy a fejlesztő nincs a lime-hoz kötve. Lehetőség biztosított a sokak számára kedvel PHPUnit használatához is.

Funkcionális teszt

A fejlesztés során gyakran kell manuálisan is ellenőrizni az elkészült alkalmazásunkat. Az esetek többségében ilyenkor elindítunk egy böngészőt, és kézzel végig nézzük, hogy minden elem megfelelően működik-e. Belátható, hogy ez az eljárás igen körülményes egy széles problémakört lefedő alkalmazás esetén. Ma már erre a feladatra is léteznek megfelelő eszközök, és az általam is használt Symfony keretrendszerbe is már beépítették egy ilyen eszközt.

A Symfony keretrendszerben a funkcionális tesztek az sfBrowser osztály egy példányában futnak. Ez az osztály egy web böngészőt implementál, melyben minden olyan Symfony objektumot elérhetővé teszt számunkra, amelyre egy kérés előtt és után szükségünk lehet. A sfBrowser osztály rendelkezik olyan metódusokkal, mellyel az implementált böngészőt tudjuk konfigurálni, és olyanokkal is melyekkel egy felhasználó tevékenységét tudjuk szimulálni, mint például: click(), select(), back(), reload(), get(), stb.

A funkcionális teszteket a /test/functional/ mappában tároljuk. Készítésükhöz az sfTestFunctional osztályt használjuk, melynek konstruktorába egy sfBrowser példányt adunk át. A keretrendszer egy igen kényelmes felületet nyújt a programozónak a felület tesztelésére. Lehetőségünk van az sfWebRequest és az sfWebResponse objektumok tesztelésére is. Ezáltal olyan teszteket is készíthetünk, mint például a menü komponens tesztelése, a felületen a szűrési feltételnek eleget tevő adatok jelennek-e meg, vagy épp hogy egy listában a maximálisan megjeleníthető mennyiségű adat jelenik-e meg. Az elkészült funkcionális tesztek futtatása nagyon hasonlít a unit teszteknél írtakhoz. A leglényegesebb különbség, hogy itt a test:functional task-ot futattjuk le a parancssorban.

A dolgozat megszabott határai miatt nem jutott lehetőségem példatesztek bemutatására. Ennek ellenére úgy gondolom, ez a rövid leírás sok hasznos információt ad a Symfony saját tesztelő keretrendszeréről.

Üzembe helyezés

Az elkészült alkalmazás kifejlesztését a saját személyi számítógépemen végeztem miután minden szükséges konfigurációs lépést elvégeztem. Idővel minden alkalmazást sikerül egy olyan színvonalra kifejleszteni, mely után úgy döntünk, hogy egy éles szerverre is feltesszük. Én sajnos nem rendelkezek olyan szerverrel, hogy ezt megvalósíthassam. A Symfony megalkotói azonban azok számára, akik rendelkeznek a szükséges informatikai háttérrel az üzembe helyezés esettőségre is adnak megoldást. Egy elkészült projekt üzembe helyezésében sajnos nincs tapasztalatom, viszont igyekeztem minél több információt szerezni arról, hogy egy ilyen folyamatot milyen módon lehet végrehajtani.

Első lépésként szükség van az éles szerveren egy web szerverre, egy adatbázis szerverre, és természetesen a PHP-re. Miután ezeket az eszközöket telepítettük, jöhet a szerver konfigurálása. A PHP működésének az ellenőrzésére a Symfony biztosít egy PHP script-et (check_configuration.php), melyet ha lefuttatjuk a böngészőben és parancssorban is akkor megtudjuk, hogy a PHP, minden szükséges kiegészítővel egyetemben megfelelően lett-e telepítve. A dokumentáció javasolja a PHP Accelerator telepítését is, mellyel PHP scriptek futásának gyorsítását, és így a futtató szerver terhelésének csökkentését érjük el. Mindezt úgy, hogy a PHP feldolgozó folyamataiba a szerver beépülve a már egyszer lefordított forrásfájlok újbóli lefordításának munkáját spórolja meg az értelmező számára a lefordított változatok egy gyorsító tárban történő gyűjtésével. Az Symfony keretrendszer relatív útvonalakt használ, melynek köszönhetően nem kell, amit aggódnunk, hogy ha a projektet feltelepítjük a szerverre, akkor nem fogja a szükséges osztályokat elérni. Az adatbázis konfigurálásra két lehetőség van, vagy a szerveren parancssorból kiadva végezzük el a szükséges konfigurálást, vagy a database.yml fájl módosításával. Az alkalmazásom fejlesztése során mivel használtam plugin-t ezért le kell futtatni a publish-assets parancssori PHP kódot is. Ezeket követően még számos beállítási lehetőséget javaslott elvégezni, mint például a hibaoldalak személyre szabása, a web root könyvtár beállítása, a cache és a log beállítása, a session beállítások elvégzése (timeout, stb.).

Nem kis munka egy éles szerver megfelelő beállítása. Ami jó hír, hogy miután ezeket a beállításokat elvégeztük a deployment már gyerekjáték. A keretrendszer egy deploy task-ot ad, a fejlesztő számára melynek segítségével telepíthető az adott projekt a szerverre. Ez a task a config/properties.ini fájlban található beállításokat használja. Itt adható meg, hogy melyik szervert milyen host-on, milyen porton, milyen felhasználó névvel, stb érhetünk el. Abban az

esetben, ha szerverhez teljes hozzáférésünk van, akkor a deploy task nem teszi fel az olyan szükségtelen fájlokat, melyek a cache/ a log/ és a web/upload mappában vannak, illetve biztonsági szempontok miatt a fejlesztői scripteket sem tölti fel a szerverre. Miután a deploy deploy task-ot egy - - go opcióval kiegészítve. Ezzel véglegesítjük a telepítést.

Összefoglalás

A dolgozatomból kiderülhet, hogy az alkalmazásfejlesztés egy igen összetett tevékenység. Kulcsfontosságú szerepe van a felhasználói követelmények megértésének, hiszen az alkalmazásunkat úgy kell felépíteni, hogy a követelményekben szereplő igényeket mindenképpen kielégítse. Ezen elképzelés kivitelezése a gyakorlatban nagyon nehéz, mivel a felhasználók nem programozó szemmel tekintenek egy alkalmazásra, hanem laikusként. Ennek kiküszöböléséhez tapasztalatra van szükség, valamit gyakori kommunikációra a fejlesztő és megrendelő között.

A folyamatosan változó igények miatt egy szoftver soha sem tekinthető teljesen befejezettnek. Úgy gondolom, hogy a Symfony keretrendszer konvenciói pontosan ezen a tényen alapszik. Ezt az állításomat alátámasztja az a tény is, hogy az elkészült alkalmazásomban több plugin-t is használtam az alap keretrendszer mellett. Ebből a megközelítésből kiderülhet, hogy a Symfony nagyon rugalmasan kezeli a folyamatos változásokat. A dolgozat keretében elkészült alkalmazás egy prototípus szerepét tölti be, ami nagy segítséggel szolgálhat egy pontosabb követelményrendszer felállításában. Segítségével lehetőség van egy felhasználói igényeket jobban kielégítő, akár kereskedelmi verzió elkészítésére. Ehhez azonban több tapasztalatra volna szükségem a dolgozatom témájaként választott raktározási és értékesítési tevékenységről, amit legjobban a szoftvert használók visszajelzéseiből tudnék megszerezni.

Az alkalmazás készítése közben több ötlet is felmerült, amivel a felhasználói élményt lehetne javítani, illetve újabb modulok beépítésével még hatékonyabb segítséget nyújtana a felhasználóknak. Ezen ötletek közé tartozik egy jQuery naptár plugin beépítése és a formok AJAX-os feldolgozása, illetve több kliens oldali validáló funkció beépítése. A vezetői szerepköröket betöltő felhasználók munkájának segítéséhez egy olyan modult szeretnék elkészíteni, amely kimutatásokat készítene táblázatos és diagramos formában is.

Irodalomjegyzék

- Dave W. Mercer, Allan Kent, Steven D. Nowicki, David Mercer, Dan Squier, Wankyu Choi: PHP5 - Bevezetés a PHP5 programozásába
- http://www.symfony-project.org/jobeet/1_4/Doctrine/en/
- http://www.symfony-project.org/plugins/
- http://www.symfony-project.org/api/1_4/
- http://forum.symfony-project.org/
- http://www.doctrine-project.org/documentation
- http://www.doctrine-project.org/documentation/api/1_2
- http://www.doctrine-project.org/blog/introducing-the-doctrine-forum

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani témavezetőmnek, Dr. Kuki Attila tanár úrnak, hogy lehetőséget biztosított munkám sikeres elvégzéséhez, valamint dolgozatom megírásához nyújtott útmutató tanácsaiért.