Szakdolgozat

Csiszár Attila

Műszaki informatikai szak, hálózati technológiák szakirány, nappali tagozat

Kecskeméti Főiskola

Gépipari és Automatizálási Műszaki Főiskolai Kar

Kecskemét

2008

Tartalomjegyzék

1.Bevezetés....4
2.Webáruházak általános jellemzői....6
2.1.A termékek rendszerezése....7
2.2.A kosár szerepe....8
2.3.Felhasználói szerepkörök....8
3.Rendszer-specifikáció készítése....10
3.1. A termékek rendszere....10
3.2.A kosár....11
3.2.1.A kosarak felhasználókhoz rendelése – munkamenet kezelés alkalmazása....11
3.2.2.A kosár felhasználói esetei....12
3.3.Rendelési folyamatok....12
3.3.1.Rendelések rögzítése....13
3.4.Felhasználók azonosítása és kezelése....13
3.4.1.A jelszavak titkosítása....14
3.4.2.A felhasználók beléptetése....14
3.4.3.Felhasználói jogkörök kiosztása....15
3.4.4.Használati esetek....15
4.Tervezési folyamat - az adattárolás megtervezése....16
4.1.A termékek és kategóriák valamint a kapcsolódó jellemzők tárolása....16
4.2.A rendelésekhez kapcsolódó adatok tárolása....18
4.3. Felhasználói adatok....18
4.3.1.Felhasználói jogkörök....19
5.A felhasználói felületek megtervezése....20
5.1.Navigációs terv....20
5.2.HTML prototípus elkészítése....21
5.3.Tervezési szempontok....23
6.A Ruby on Rails keretrendszer....24
6.1.Mi az a Ruby on Rails?....24
6.2.Mit tud a Ruby nyelv?[4][5]....24
6.2.1.A Ruby építőkövei....25
6.2.2.A Ruby magasabb szintjei....29
6.2.3.Ruby segédprogramok....32
6.3.A Rails az MVC szemszögéből[8]....33
6.3.1. Az ActiveRecord modul - a Rails 'M'-je[9]....35
6.3.2. Az ActionPack modul....39
6.4.A Rails további moduljai....43
7.Alkalmazás áttervezése Rails környezetre[3]....44
7.1.Modellek....44
7.2.Útvonalak és vezérlők megtervezése....44
8.Programfejlesztés Railsben....46
8.1.Kezdő lépések az alkalmazás létrehozásában....46
8.2.Modellek létrehozása....47
8.2.1.Adatbázissémák migrációkkal....47
8.2.2.Kapcsolatok kialakítása a modellek között....49
8.2.3.A felhasználók modell virtuális attribútumai....51
8.2.4.Kosár implementálása....51

8.2.5.Ellenőrzések....52
8.2.6.Az irányítószámokat ellenőrző plugin....54
8.2.7.Modellek gyors ellenőrzése....55
8.3.Vezérlők és nézetek használata....56
8.3.1.Nézetek meghívása és használata....57
8.3.2.Helperek....59
8.4.Űrlapok előállítása és kezelése....60
8.4.1.Egyedi FormBuilder létrehozása....60
8.4.2.Az űrlap paramétereinek kezelése....61
8.5.A Rails munkamenet kezelése....62
8.5.1.Flash - a Rails egy speciális munkamenet eljárása....63
8.6.Felhasználók kezelése....64
8.7.Adminisztrációs felület leválasztása....65
8.8.Az alkalmazás lokalizációja magyar nyelvre....65
9.Tesztelés[11]....68
9.1.Tesztadatok bevitele....69
9.2.Modellek tesztelése egység tesztekkel....70
9.3.Vezérlők tesztelése....73
9.4.Tesztek futtatása....73
9.5.Összefoglalva....74
10.Összefoglalás....75

1. Bevezetés

„A web a változások hatására egyre inkább középpontba kerül. Már ma is vannak olyan területek, ahol úgy végezhetünk munkát, hogy egy böngészőn kívül más alkalmazás nem is fut gépünkön. Például egy újságíró, szerkesztő utánanéz a neten a híreknek (esetleg egy webes program, vagy egy kiterjesztés segítségével folyamatosan értesül róluk), a webes levelezőben kapcsolatot tart a külvilággal, a csevegőszobákban társalog, feltölt és letölt fotókat a webes albumokból, s cikket publikál a webes felületű szerkesztőségi rendszeren keresztül.”[1, A jövő a webes alkamazásoké]

Az internet egyre bővülő felhasználói tábora legnagyobb hatását a web fejlődésére tette. A kezdetben statikus, technikai leírásokat és dokumentációkat tartalmazó World-Wide-Web mára a feledés homályába veszett, felváltotta a dinamikus tartalmú, felhasználó-központú weboldalak tömege. Az új médium széleskörűbb felhasználást hozott magával: olyan web-oldalak jelentek meg – kép - és videómegosztók, közösségi oldalak, webáruházak – amelyeket összetett funkcióik miatt mára web alapú alkalmazásoknak nevezünk.

A fejlődés természetesen nagyban hatott az őket működtető rendszerekre is. Ezek kezdetben jól átlátható, egyszerű megoldások voltak, de a komplexebb igények kielégítése túl időigényessé vált velük. Ennek hatására jöttek létre - a hagyományos asztali programfejlesztésből átvéve - a speciálisan webes alkalmazások fejlesztését megkönnyítő web alapú keretrendszerek.

Ezen keretrendszerek közül emelkedik ki a Ruby on Rails, mely sok tekintetben hozott alapvető szemléletváltozást illetve megközelítést a webalkalmazások fejlesztésébe.

Szakdolgozatommal ezért azt szeretném demonstrálni, hogyan lehet egy web alapúra tervezett alkalmazást - ez a szakdolgozat címében szereplő webáruház - Ruby on Rails keretrendszerben megvalósítani.

Miért pont webáruház? - teheti fel a kérdést az olvasó. Valóban manapság sok informatikus hallgató készíti hasonló témából szakdolgozatát, ugyanakkor tény, hogy egyértelműen ez fedi le leginkább a webalkalmazások fejlesztése és tervezése során előforduló összes problémakört; kezdve az adattárolás megszervezését egy adatbázisrendszerben, a felhasználói felületek és folyamatok megtervezését, a felhasználói-feladatkörök kezelését, illetve a biztonságos és helyes működést biztosító tesztelési fázist.

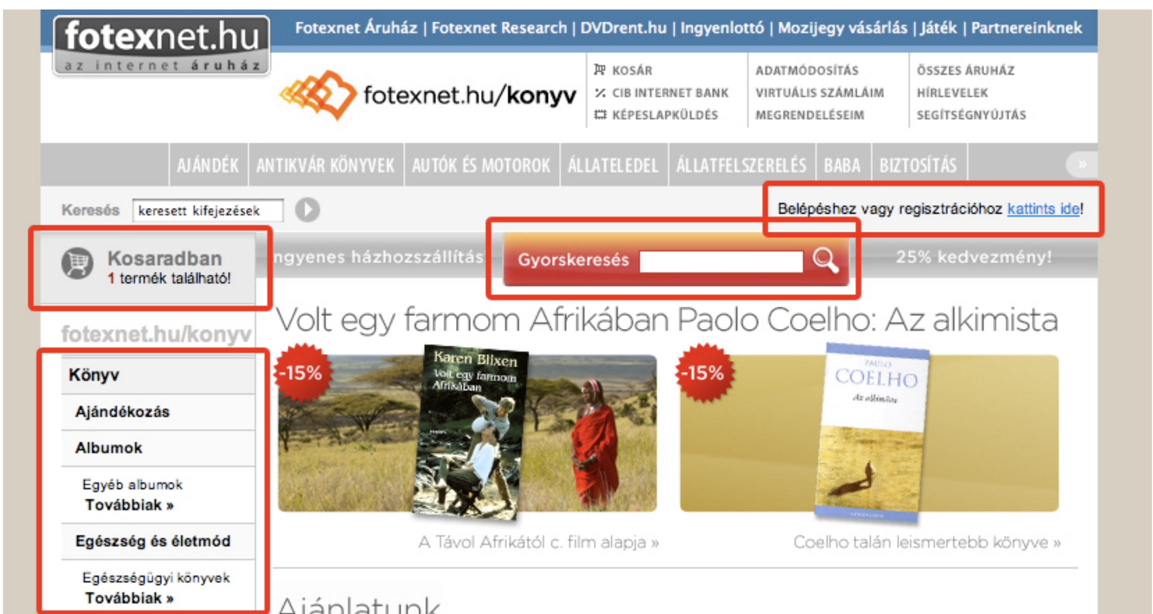
A szakdolgozatom első részében a webáruházakkal kapcsolatos általános jellemzőkre keresem majd a választ, néhány konkrét példával illusztrálva, amelyekből megpróbálom leszűrni mik azok a főbb követelmények amelyeket megfogalmazhatunk egy webáruházzal kapcsolatban.

A meghatározott jellemzőkből következhet a rendszer-specifikáció elkészítése valamint a tervezési folyamatok elvégzése: a szükséges adatbázisséma(háttér) megtervezése, a felhasználó felületek és folyamatmenetek illusztrálása. Ebben az egységben a hangsúlyt a rendszer-függetlenségre fektetem: fontosnak tartom, hogy az alkalmazás tervei akár más keretrendszerben is nyugodtan megvalósíthatóak legyenek.

A kész tervek alapján pedig következhet a webáruház megvalósítása Ruby on Rails környezetben, ahol egy-egy kulcsrészen keresztül szeretném bemutatni, milyen megoldásokat ajánl a keretrendszer a webalkalmazások fejlesztése közben előforduló gyakori problémákra. A fejlesztést végül a tesztelési fázis zárja majd, ahol az elkészült alkalmazás hibamentes működésének biztosításán lesz a hangsúly, gyakorlati módszerekkel.

E szakdolgozatnak nem célja és nem is áll módjában, hogy a webáruházak üzleti jelentőségét, vagy gazdasági szerepét részletezze, csupán technikai megvalósítást tartalmaz.

2. Webáruházak általános jellemzői



1. ábra. A fotexnet.hu webáruház egy részlete[1]

Ebben a fejezetben körüljárjuk azokat a főbb tulajdonságokat, amelyek nemcsak jellemeznek, de feltétlenül szükségesek is egy webáruház alapvető működéséhez.

Bár a webáruházaknak milliónyi változata létezik attól függően, hogy épp milyen termékeket árusít, mégis vannak jellemzők, amelyek azonosak. Az 1. ábrán egy webáruház részletét láthatjuk kiemelve rajta azokat a főbb funkciókat, amelyek a webáruházak többségében megtalálhatóak: ilyenek a kosár, termékkategóriák, keresési funkciók, felhasználók beléptetése és regisztrálása.

Egy webáruház működése elméletben nem sokban különbözik fizikai testvéreitől: a főbb szerepeket itt is a termékek, az oldalt böngésző vásárlók, és az azt üzemeltető, karbantartó tulajdonos hármasa határozza meg.

1 Forrás: http://www.fotexnet.hu/fotexnet_aruhaz/page/category/001/konyv

2.1. A termékek rendszerezése

A termékek esetében legfontosabb, hogy valamilyen rendszerbe tudjuk rendezni őket, mint ahogy a boltban is külön sorokban találjuk az eltérő kategóriájú termékeket. A mi esetünkben mindez azt jelenti, hogy amikor egy felhasználó(vásárló) kiválaszt egy kategóriát, a benne található termékek kilistázásra kerülnek. Ehhez a legmegfelelőbb egy **hierarchikus, fa-szerű kategória-rendszer** kialakítása, amelyben fő és alkategóriák rendszerezik a termékeket.

A 2. ábra egy lehetséges megvalósítást mutat be egy már működő webáruház esetében.

A termékek kategóriákba való rendezése megkönnyíti a specifikus keresést, de korlátokat is görgethet a felhasználó elé, hiszen nem biztos, hogy tudja milyen kategóriában keresse a kívánt terméket. Ezért jelentősen megkönnyíti a vásárlók tájékozódását egy megfelelő **keresési funkció** implementálása.

2. ábra. A bookstation.hu webáruház hierarchikus kategória rendszere[2]

Hasznos lehet továbbá, ha a termékeket nemcsak a jellemzőik (pl. gyártó, típus, ár, szerző stb.) határozzák meg, hanem kulcsszavakat is tudunk rendelni hozzájuk. A legújabb webalkalmazásoknál egyre népszerűbb funkciót töltenek be az ezt megvalósító címkézési megoldások, címkefelhők. Így nemcsak a keresést könnyíthetjük meg, de a kereszthivatkozások létrehozását is: például egy, az állatok természetéről szóló könyvet böngésző vásárlónak rögtön ajánlhatunk pár hasonló témájú természetfilmet is.

A termékek rendszerezése a tájékozódást segíti, a tájékoztatást nem. Mivel a vásárlók fizikailag nem tudják megnézni a termékeket, ezért minél részletesebb információval kell majd ellátnunk őket a termékekről. A webáruházak többsége ezt egy különálló felületen oldja meg: a lista nézetekben csak a főbb jellemzőiket (pl. ár, rövid leírás) találjuk meg, ahonnan a vásárló könnyedén kérheti a termék részletes információinak a megjelenítését.

2 Forrás: http://www.bookstation.hu/index.php?op=list_item&category_id=014004

2.2. A kosár szerepe

Abban az esetben, ha egy látogató szeretne megrendelni egy terméket, biztosítanunk kell a választások, és a rendelni kívánt mennyiségek valamilyen formában való tárolását, hiszen rendkívül kényelmetlen lenne, ha a rendeléseket külön-külön egyenként kellene leadnia. Ennek a szerepét tölti be a (**virtuális) kosár**: segítségével több terméket is kiválaszthat a vásárló, rendelési mennyiséget adhat meg hozzájuk, de ugyanígy ki is veheti őket a kosárból (lásd Error: Reference source not found. ábra). Hasonlóan mint a valóságban.

Árucikk	Egységár	Mennyiség	Összesen	Törlés
AZ ÉN TÖRTÉNETEM LEWIS HAMILTON RINGIER KIADÓ KFT Szállítás: 5-7 munkanap	Bolti ár: 2.990 Ft Az Ön egyedi ára: 2.841 Ft (5% kedvezmény)	2 db	5.682 Ft	
AZ INTERNET ÉS LEHETŐSÉGEI BÁRTFAI BARNABÁS BBS-INFO KÖNYVK. ÉS INFORM. KFT. Szállítás: 2 munkanap	Bolti ár: 1.970 Ft Az Ön egyedi ára: 1.872 Ft (5% kedvezmény)	10 db	18.720 Ft	
UNIX - A RENDSZER HASZNÁLATA KETLER IVÁN - SZEBERÉNYI IMRE - SZIGETI SZABOLCS PANEM KFT Szállítás: 2 munkanap	Bolti ár: 2.900 Ft Az Ön egyedi ára: 2.755 Ft (5% kedvezmény)	1 db	2.755 Ft	
Összesen szállítási költség nélkül: 27.157 Ft				

Kosár frissítése | Vásárlás folytatása | Megrendelés véglegesítése

3. ábra. A libri.hu internetes webáruház kosara[3]

2.3. Felhasználói szerepkörök

A vásárlás lezárásaként a felhasználó véglegesíti rendelési tételeit, megadja személyi adatait, házhoz szállítás esetén lakcímét és elérhetőségeit, amelyeket mind el kell majd tárolnunk. Visszatérő vásárlók esetében, hogy ne kelljen minden vásárlás alkalmával adataikat megadniuk, vagy, hogy például különféle kedvezményekben részesüljenek a webáruházak a vásárlók adatait **felhasználói profilok**hoz kötik, amelyhez szorosan kapcsolódik egy **regisztrációs és beléptető rendszer** kialakítása.

Gondolnunk kell arra is, hogy a webáruházunk üzemeltetője - eladója - szeretné időnként megváltoztatni a termékkínálatot, esetleg a kategória-rendszert, ellenőrizné a leadott rendeléseket vagy akciós termékárakat, termékcsomagokat alakítana ki, egyszóval biztosítanunk kell majd számára egy ehhez szükséges interfészt. A hasonló feladatok ellátására a webalkalmazások többsége egy, az alkalmazás vagy weboldal eredeti funkcióitól különálló **adminisztrációs felület**et biztosít, ahol az erre jogosult felhasználó egyszerűen végezheti el feladatait.

3 Forrás: http://www.libri.hu/hu/shop_cart/

Mint a fentiekből is kiderült a webáruház működésében két fő szereplő vesz majd részt, a vásárló és a működtető, így meg kell majd oldanunk a **felhasználók azonosítását**, és biztosítanunk kell a **jogosultságok** megfelelő **kezelését**.

Most, hogy sikerült meghatároznunk milyen fő funkciókat kell majd a webáruháznak mindenféleképpen megvalósítania következhet az alkalmazás megtervezése. Az terveknek elsődlegesen az implementációt kell segíteniük, hogy lássuk hogyan is kell az egyes egységeknek működnie, másodlagosan pedig dokumentációs célokat is szolgálnak.

3. Rendszer-specifikáció készítése

Programfejlesztőként egy alkalmazás leírásakor, a tervezés előtt rendszer-specifikációt kell készítenünk. Hogy mit tartalmaz egy jó rendszer-specifikáció arról megoszlanak a vélemények, egységes elfogadott szabvány helyett a józan eszünkre hallgassunk: egy rendszer-specifikáció akkor jó, ha tartalmaz minden olyan kulcselemet, amely az alkalmazás megtervezéséhez szükséges, és nemcsak mi, hanem más fejlesztők is kölcsönösen megértik - különösen igaz ez egy többfős projekt esetében.

A rendszer-specifikáció elkészítéséhez az előző fejezetben megállapított főbb jellemzőkből indulok ki. Meghatározom milyen használati (use-cases) esetek fordulhatnak elő velük kapcsolatban, azaz hogy az egyes felhasználók milyen funkciókat érhetnek el, milyen módon használják majd a rendszerünket. A lehetséges eseteket két szereplőre vetítjük le: a vásárlóra és az eladóra. A használati esetek leírása megkönnyíti majd a tervezés második lépését, amikor a felhasználói felületek tervezését végezzük, hiszen pontosan tudjuk majd milyen funkciókat kell az alkalmazás használóinak elérniük.

Ezenfelül a már meghatározott fő funkciók megvalósítását és a hozzájuk kapcsolódó problémákat tárgyalom ebben a fejezetben ki.

3.1. A termékek rendszere

A vásárlóknak biztosítanunk kell a termékek böngészési lehetőségeit, ez nyilvánvalóan az alkalmazásunk legfőbb feladata lesz. Meg kell oldanunk a termékek listaszerű megjelenítését különböző szűrési és rendezési lehetőségekkel - például kategória, név, ár szerint - egy külön nézetettel, ahol a vásárló részletes információhoz juthat a termékről, megnézheti a hozzá kapcsolt képeket, elolvashatja a jellemzőit.

Az eladó számára ugyanakkor a következőket kell implementálnunk a termékek-rendszerével kapcsolatban:

- Kategóriák létrehozása, szerkesztése és törlése

  Ide kapcsolódik majd a rendszerezés: alkategóriák és főkategóriák megadása

- Termékek hozzáadása, szerkesztése és törlése

  Az alapvető adatok - elnevezés, ár, leírás, egy kép csatolása valamint jellemzők hozzárendelése.

Mint a fentiekből is látszik, és ahogy arról már előzőekben szó esett, egyszerűbb, ha ezeket a funkciókat a webáruház egy elkülönített szekciójában valósítjuk meg, ezért terveznünk kell egy adminisztrációs felületet. Ez további feladatokat követel meg: biztosítanunk kell, hogy azt csak az arra jogosultak használhassák. Ez utóbbiról részletesebben a felhasználók azonosítása című részben lesz szó.

3.2. A kosár

A kosár fogja biztosítani, hogy a vásárló által, a böngészés során megrendelésre kiválasztott termékeket elraktározzuk és megőrizzük azokat a rendelés lezárásáig, hogy a rendeléshez mennyiséget is tudjon hozzárendelni, visszalépés esetén pedig egyszerűen törölhesse a korábban kiválasztott terméket.

A kosarak teljes tartalma a rendelési tételekből épül fel. A tételen keresztül hivatkozunk a termékre, ez rögzíti a rendelési mennyiséget valamint az árakat – erre azért van szükség, mert előfordulhat, hogy a megrendelés után változik a termék ára, nekünk viszont a vásárlási árat kell biztosítanunk.

Azt, hogy a kosarak tartalmát miként tároljuk el programozási szinten nagyban befolyásolja, hogy hogyan rendeljük azokat az egyes felhasználókhoz.

3.2.1. A kosarak felhasználókhoz rendelése – munkamenet kezelés alkalmazása

> ❞ Amikor egy asztali (desktop) alkalmazással dolgozunk, megnyitjuk, változtatunk valamit, majd bezárjuk. A számítógép tudja, hogy ki végzi a műveletet. Tudja, amikor elindítjuk az alkalmazást, és amikor bezárjuk. De az interneten ez nem így van. A webszerver nem tudja, hogy kik vagyunk és mit csinálunk, mert a HTTP protokoll nem tartja meg a kérések közt az állapotát.[2, 8.2.6. fejezet]

Figyelembe kell vennünk, hogy a HTTP protokoll állapotátmeneti protokoll, nem képes a kérések között kapcsolatot teremteni, ennek leküzdésére többféle megoldást is kitaláltak, amelyeket egységesen munkamenet kezelésként definiálunk.

A legegyszerűbb megoldás, ha a kérések URL-jében helyezünk el valamilyen azonosítót. Ez viszont több szempontból sem szerencsés. Egyrészt nem túl felhasználóbarát: zavaró, adott esetben túl hosszú URL-eket eredményez, amelyek ráadásul nem könyvjelzőzhetőek. Másrészt rossz indulatú támadások célpontja is lehet, tehát nem túl biztonságos. Ezért bár az egyedi azonosítót megtartjuk, azt más módszerekkel adjuk át két kérés között.

A böngészők többsége ma már támogatja a sütiket (cookies), amelyek lehetővé teszik, hogy információt helyezzünk el a látogató számítógépén, majd azt a következő kéréssel kiolvassuk. A webáruház esetében ez utóbbit fogjuk alkalmazni.

Épp ezért az egyes kosarakat nem adatbázis szinten fogjuk tárolni, hanem a munkameneteken keresztül, végső tárolásukra csak a rendelés feladásakor kerül majd sor - az már más kérdés, hogy végső soron a munkamenetek tartalmát is adatbázisban tároljuk majd, a felhasználó sütije csak a hozzá tartozó munkamenet azonosítóját fogja tartalmazni.

3.2.2. A kosár felhasználói esetei

A kosár esetében a felhasználók számára a legfontosabb, hogy belehelyezhessék a rendelésre kiválasztott termékeket - ezt, web alapú felület révén a termékek melletti linkeken keresztül érhetik majd el. Ezenfelül a kosarak tartalmát is meg kell majd jelenítenünk: a felhasználónak itt nyílik majd lehetősége a mennyiségek módosítására, illetve hogy töröljön egy rendelési tételt - egy kiválasztott terméket kivegyen - a kosarából.

3.3. Rendelési folyamatok

Ha a vásárló befejezte a böngészést, és betette kosarába a rendelni kívánt termékeket eljutunk a vásárlási folyamat végéhez, amely a rendelés lezárását jelenti.

A lezárást a felhasználó bejelentkeztetésével vagy regisztrálásával kezdjük, amit a következő fejezet részletesen bemutat. Ami fontosabb, hogy a rendelés végeztével még egyszer összesítjük a kosár tartalmát átvizsgálásra, majd a felhasználótól bekérjük a rendelési adatokat.

A következő adatokat kell kötelezően megadni, alattuk szerepel a érvényesség feltétele:

- Név
- Lakcím - irányítószám, település, valamint utca és házszám mezőkből áll

  A formátumok megfeleljenek az általános formázási elveknek, ezenfelül létező településnevet adott-e meg, az irányítószám és település egyezik-e (ez utóbbi két feladathoz valamilyen külső adatforrásra lesz majd szükségünk).

- Telefonszám

  Megfelel-e az általános formázási elveknek

- E-mail cím

  Meg kell felelnie a következő formának: azonosito@host.tld

- Fizetési és szállítási mód

  Egy kiválasztása kötelező lesz.

Ennél a lépésnél ellenőriznünk kell az adatok helyességét, csak ezután zárhatjuk le a rendelést, és menthetjük el az adatbázisban. Ha minden adatot helyesnek találtunk, és rögzítettük a rendelést, küldünk egy értesítő üzenetet a vásárlónak, hogy a rendszerben mentésre került a megrendelése.

3.3.1. Rendelések rögzítése

Az alkalmazás szintjén a rendelés rögzítésekor nemcsak a vásárló adatait kell elmentenünk, hanem a kosarak tartalmát is, mivel azokat eddig a munkamenetben tároltuk.

Bár a felhasználók címadatait adatbázis szinten is eltároljuk a rendeléseknél erre nem támaszkodhatunk - mivel előfordulhat, hogy a felhasználó megváltoztatja adatait két rendelés között vagy csak egyszerűen a regisztrálttól eltérő címre szeretne rendelni. Ezért minden megrendelés esetén a rendelési címet is rögzítenünk kell.

A rendelések ezenfelül különböző státuszokat vehetnek fel - erre a rendszerezésük miatt van szükség az eladók számára - lehetnek teljesítettek, függőben lévőek, vagy hibásak. Ezeket az adminisztrációs felületen keresztül állíthatják majd be az eladók, ahol megtekinthetik majd a leadott rendeléseket is.

3.4. Felhasználók azonosítása és kezelése

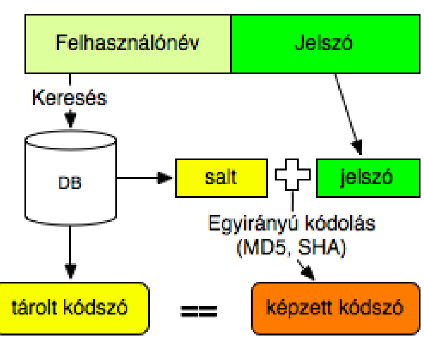
A felhasználók azonosítására mind a vásárlók beléptetésénél mind az adminisztrációs felületen szükségünk lesz. A felhasználóknak azonosítaniuk kell majd magukat, hogy jogosultságot szerezzenek az adott funkció eléréséhez.

Az autentikáció során két adat megadására kötelezzük őket: a felhasználónevüket és a hozzá tartozó jelszót kell majd megadniuk - amit összevetünk az adatbázisban tároltakkal. Mindezek azonban bizonyos biztonsági lépéseket is megkövetelnek, a külső támadások kivédése miatt.

3.4.1. A jelszavak titkosítása

A jelszókat a következőképp tároljuk: a karakterlánchoz hozzáadunk egy másik véletlenszerűen generált karakterláncot[4], amit egyirányú kódolással elmentünk az adatbázisba.

Az egyirányú kódolás olyan kódszót képez, amiből nem tudjuk közvetlenül az eredeti információt visszanyerni. Így beléptetéskor a felhasználó által megadott jelszóból ugyanezzel az eljárással mindig kódszót kell képezni és összevetni azt az eltárolttal(lásd 4. ábra). Ebből következik, hogy a jelszóhoz hozzáadott karakterláncot is el kell majd tárolnunk, hiszen az összevetés során szükségünk lesz rá.



4. ábra. A jelszó ellenőrzési folyamat sémája

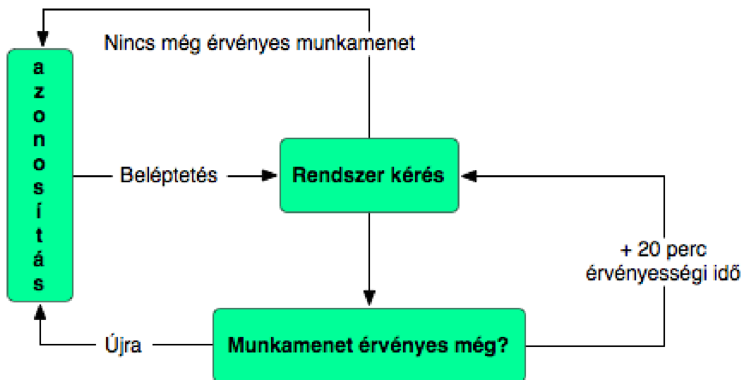
3.4.2. A felhasználók beléptetése

A kosár feladatköreinél már megbeszéltük, hogy a HTTP protokoll nem képes a kéréseket egyedi felhasználókhoz kötni, ezért munkamenet-kezeléssel fogjuk hozzárendelni az egyes kosarakat a megfelelő vásárlóhoz. Valami hasonló megoldást kell alkalmaznunk a felhasználók azonosítása után is, hogy ne kelljen minden oldalkéréskor jogosultságaikat megadniuk.

Az autentikáció tehát megköveteli egy beléptető felület létrehozását, ahol azonosíthatják magukat a felhasználók. Sikeres azonosítás után a munkamenetben elhelyezzük és minden válasszal elküldjük a felhasználói azonosítójukat, amit minden kéréskor ellenőriznünk kell.

Itt bevezetünk egy újabb biztonsági intézkedést, a munkamenetek csak adott időintervallumig lesznek érvényesek - például 20 percenként lejárnak - , az érvényességi időt minden kérés-válaszkor ennyi időegységgel növeljük meg (lásd az 5. számú ábrát).

4 A kriptográfiában ezt a tetszőlegesen választott karakterláncot saltnak, hozzáadott sónak nevezik.



5. ábra. A munkamenet-kezelés folyamata

3.4.3. Felhasználói jogkörök kiosztása

A funkciók meghatározásakor már említettem, hogy alkalmazásunk kétszereplős lesz: vásárlókra és eladókra(adminisztrátorokra) oszlik (természetesen lesz egy harmadik szereplőnk is az egyszerű látogató, de ő a mostani szempontunkból lényegtelen). Ami fontos, hogy a belépett felhasználókat meg tudjuk különböztetni, jogköröket tudjunk hozzájuk rendelni - például a vásárlók ne érhessék el az adminisztrációs funkciókat - amit szintén adatbázis szinten kell majd megoldani.

A felhasználók kezelését részletesen kidolgoztuk, de nem ejtettünk még szót milyen használati esetek fordulhatnak elő alkalmazásunk felhasználói moduljában.

3.4.4. Használati esetek

Ezek többsége kimerül a felhasználók saját adatainak megváltoztatásában: amit mindkét szereplőnk - vásárlók és eladók - esetében biztosítanunk kell. Az eladók esetében ugyanakkor az adminisztrációs felületen implementálunk kell, hogy hozzáadhassanak új, adminisztrációs jogokkal rendelkező felhasználókat, módosíthassák meglévő felhasználók adatait vagy akár törölhessék hozzáférésüket a rendszerből.

Most, hogy meghatároztuk az alkalmazásunk egyes részeinek miként kell működnie, a felhasználók pedig milyen funkciókat érhetnek majd el, következhet az alkalmazás részletes megtervezése.

4. Tervezési folyamat - az adattárolás megtervezése

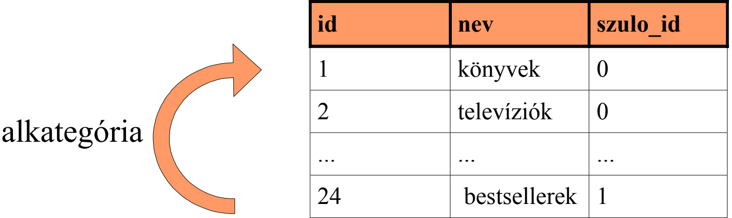
Az alkalmazás megtervezését az adattárolás mikéntjét megválaszoló kérdésekkel kezdjük: mivel egy adatbázisrendszer használatára lesz szükségünk meg kell majd terveznünk annak struktúráját - a különálló és egymáshoz kapcsolódó táblákat, a kapcsolati rendszereket, az adatok tárolási formáját.

Elsőként kezdjük a webáruház alapját képező termékekkel és a hozzájuk kapcsolódó adatok tárolásának a megvalósításával.

4.1. A termékek és kategóriák valamint a kapcsolódó jellemzők tárolása

Az előző fejezetekben szó esett arról, hogy a termékeket kategóriákba kell majd sorolni: a vásárlókat elsődlegesen ez segíti abban, hogy a termékek nagy információhalmazából kiszűrjék a számukra legfontosabb elemeket.

Az nyilvánvaló, hogy külön táblára[5] kell bontanunk a kategóriákat és a hozzájuk tartozó termékeket. Közös tábla esetében ugyanis redundancia alakulna ki - egy-egy kategória információja többszörösen is megjelenne az adatbázisban. További feladatunk, hogy a kategóriák között kapcsolatot teremtsünk, fő és alkategóriák hálózatát hozhassuk létre. Ehhez minden kategória esetében bevezetünk egy plusz mezőt, a szülő azonosítót, amely egy már létező kategóriára azonosítójára mutat és így meghatározza mely kategória gyermeke az adott elem.



id	nev	szulo_id
1	könyvek	0
2	televíziók	0
...	...	...
24	bestsellerek	1

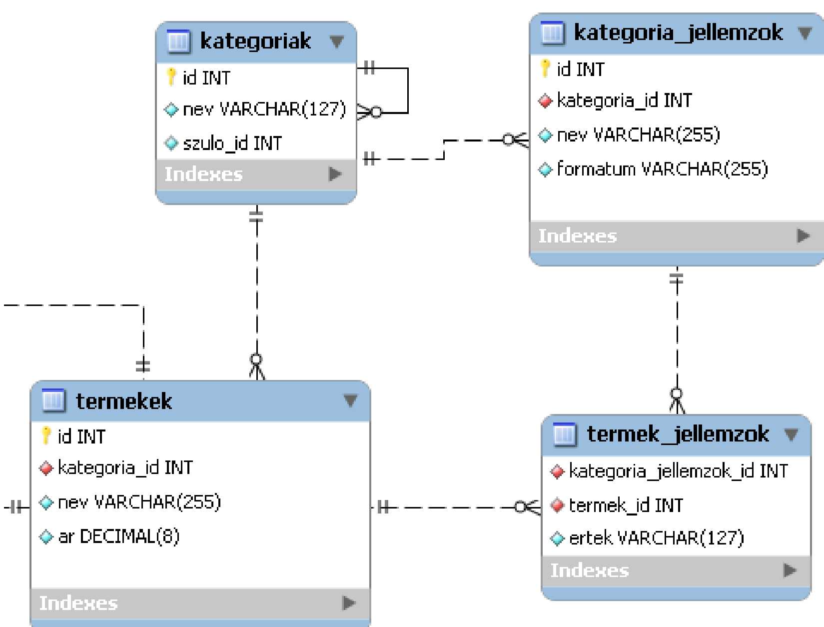
6. ábra. Kategóriák és alkategóriák ábrázolása az adatbázisban

A 6. ábra ezt szemlélteti: az olyan elemeknél, amelyek nem tartoznak felsőbb kategóriába - nincs szülő elemük - a megkülönböztetés végett, a szülő azonosító értéke a 0 lesz.

5 Nem került említésre, de az adatbázis tervezésén értelemszerűen relációs adatbázisban való tervezést értek.

Ezenfelül biztosítanunk kell, hogy a termékekhez tulajdonságokat is rendelhessünk. Mivel a hasonló kategóriájú termékekhez hasonló jellemzők tartoznak - például a televíziók esetében ilyen lehet a képátmérő és a gyártó, a könyvek esetében a szerző és a kiadási év - az egyes kategóriákhoz rendeljük a termékjellemzők csoportját.

A 7. ábra mutatja meg ennek az adatbázis szintű megvalósítását: míg a kategória jellemzők táblázat a kategóriákhoz tartozó jellemzőket tárolja, addig azt, hogy az egyes termékek esetében ezek milyen értékeket vesznek majd fel egy kapcsoló táblázat tartalmazza.



7. ábra. A kategóriák, a termékek és a hozzájuk kapcsolódó jellemzők kapcsolata az adatbázisban

A címkézési rendszer megvalósításához ehhez hasonlóan két további táblára lesz szükségünk. Egy tárolja majd a címkéket, egy másik - szintén kapcsoló táblaként, hogy mely termékhez milyen címkék tartoznak. A címkék átmenetet jelentenek majd, biztosítják a kapcsolatot két teljesen eltérő kategóriájú termék között, így a termékajánlók szempontjából lesznek majd hasznosak.

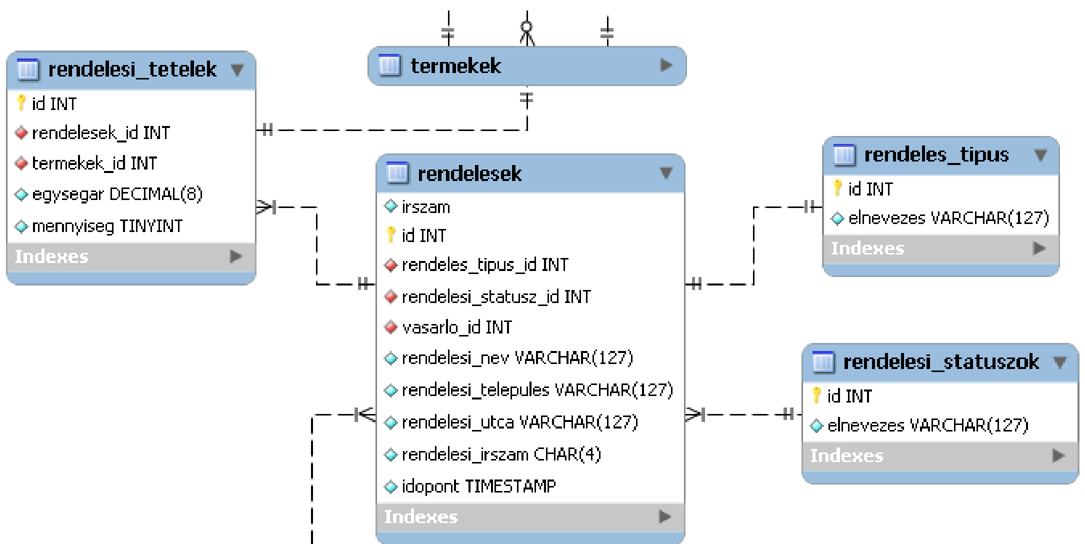
4.2. A rendelésekhez kapcsolódó adatok tárolása

A 8. ábra mutatja meg milyen formában történik a rendelések rögzítése adatbázis szinten. A rendelési tételek táblába kerülnek a kosár egyes tételei: ezen keresztül érhetjük el, hogy melyik termékre hivatkozik, itt rögzítjük, hogy a termék milyen árral szerepelt a rendeléskor, valamint a rendelési mennyiséget is itt tároljuk el.

A rendelési adatok - ilyenek a címadatok, és a megrendelő adatai - is itt kerülnek rögzítésre.

A fizetési és rendelési módokat egy különálló táblában, mint rendelési típusokat rögzítjük, a rendelések innen hivatkoznak rájuk, így a rendszer továbbfejlesztésekor vagy módosításakor könnyen lehet majd új feltételeket megadni.

A rendelési státuszok is hasonló elv miatt kerülnek egy különálló táblába.



8. ábra. A rendelések, rendelési tételek és a termékek közötti kapcsolat az adatbázisban

4.3. Felhasználói adatok

A felhasználók - tehát mind a regisztrált vásárlók, mind az adminisztrátorok - autentikációs adatait – ilyenek a jelszavak és felhasználónevek – elegendő egy közös táblában eltárolni, nincs értelme különálló táblákra bontani.

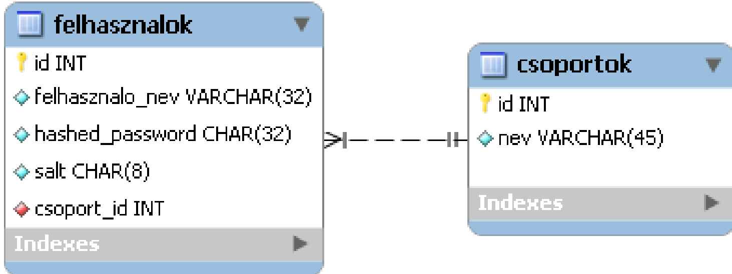
Azonban, mivel az adminisztrátorok esetében nincs szükség a címadatok tárolására, a vásárlói adatokat egy különálló táblában kell rögzítenünk, itt a vásárló felhasználói azonosítója lesz a kulcs.

4.3.1. Felhasználói jogkörök

A jogkörök jelölésekor megelégedhetnénk akár azzal is, ha a felhasználók táblájában elhelyeznénk egy, az adminisztrátori funkciót jelölő igaz/hamis értékű mezőt. Azonban nem árt, ha előre is gondolkozunk és felkészülünk arra, hogy a jövőben akár több eltérő jogkörű felhasználót is hozzá kell adnunk a rendszerhez, ezért egy különálló táblát (lásd a 9. számú ábrát) hozunk létre a felhasználói jogkörök jelölésére.

Ennek alapján a webáruház indulásakor két csoportot különböztetünk meg:

- a vásárlókét és
- az adminisztrátorokét.



9. ábra. A csoportok tábla rögzíti a felhasználók jögköreit

Az eddigiek során specifikáltuk, hogy az alkalmazásunknak milyen feladatokat, miként kell majd ellátnia, és megterveztük a mögötte működő adatbázis háttért. Ezek ugyan segítenek elképzelni a működést, de vizuális megjelenítés nélkül olyan mintha a sötétben tapogatóznánk. Szükségünk van arra, hogy képileg is el tudjuk képzelni, minként fog majd az alkalmazás egy egységben működni, azaz meg kell terveznünk a felhasználói felületét.

5. A felhasználói felületek megtervezése

A felhasználói felületek tervezése nemcsak azt jelenti, hogy megtervezzük például, hogyan nézzen ki az webalkalmazásunk nyitóoldala, hanem sokkal inkább segítenie kell elképzelni az alkalmazás különböző működési folyamatait - például, ha egy felhasználó betesz a kosarába egy terméket mi történjen -, segítenie kell a működés közben felmerülő feladatok és problémák előrevetítésében.

A felhasználói felületek tervezésével olyan problémákat, feladatokat is felfedezhetünk, amelyek egyébként csak az implementálás - programozás - folyamán bukkannának elő.

A tervezés során felskiccelhetünk vázlatképeket, használhatunk segédprogramokat is, a szakdolgozatom esetében azonban az egyik legkézenfekvőbb megoldást választottam: HTML alapon terveztem meg az egyes felületek prototípusát.

A HTML alapú prototípus gyártásnak több előnye is van, ezek:

- Egyszerű

  Ahelyett, hogy grafikai és felülettervező eszközök elsajátításával bajlódnánk egyértelműen tudunk tervezni.

- Kipróbálható

  Mivel HTML alapon készül, az elkészült prototípus működés közben egy böngésző segítségével kipróbálható. Így a fejlesztőcsapat, vagy maga a megrendelő pontos képet kaphat a végső alkalmazásról még az implementálás előtt. Megjegyzéseivel pontosíthatja az elképzeléseket, amivel fejlesztési időt spórolunk meg, mert nem a végső alkalmazást kell átszabnunk új igények esetén.

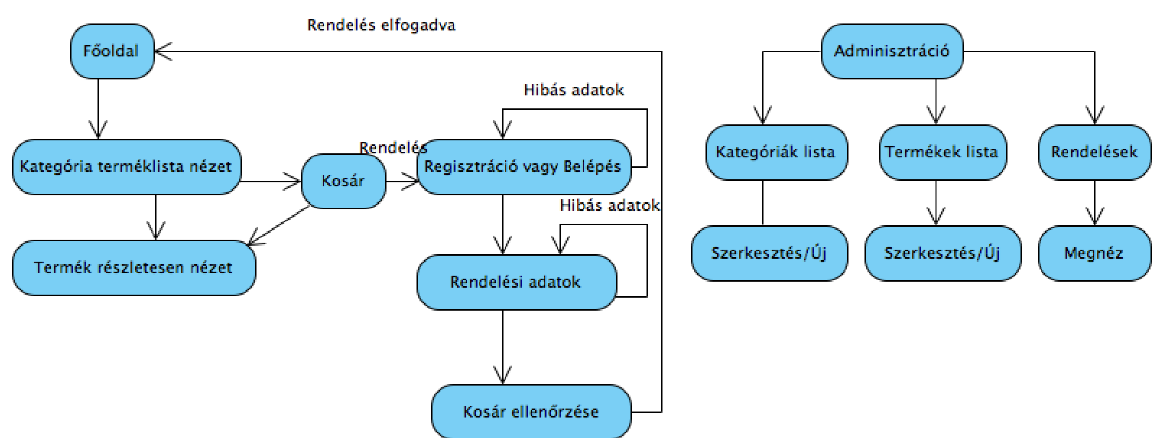
- Hatékony

  A létrehozott prototípusok felhasználhatóak lesznek a fejlesztés későbbi szakaszában, a HTML kódok beépíthetőek lesznek a végső alkalmazásunkba is.

5.1. Navigációs terv

Mielőtt hozzálátnánk a prototípus elkészítéséhez nem árt, ha készítünk egy kezdetleges navigációs tervet: azaz vázoljuk, hogy a felhasználók milyen útvonalakon érhetik el majd a különböző funkciókat.

Hogy milyen funkciókat kell figyelembe vennünk? A második fejezetben meghatároztuk a használati eseteket, ezeket felhasználva a 10. ábrán látható milyen nézeteket kell majd elkészíteni a prototípusban.



10. ábra. Az alkalmazás lehetséges nézetei

A prototípus ezek alapján a következőképpen készül: minden nézethez egy-egy oldalt rendelünk. A nézetek közötti átjárást a linkek biztosítják, csakúgy, mint a végső alkalmazásban.

A valós működést ugyanakkor csak szimuláljuk: amikor egy felhasználó például elhelyez a kosarában egy terméket egy link segítségével, akkor a következő nézetben úgy állítjuk be, mintha már lenne termék a kosarában - valós adat nincs mögötte. Amikor kiválaszt egy kategóriát a következő nézetben egy fiktív kategória fiktív termékeit tekintheti meg – megint csak anélkül, hogy valós adatokat vennénk a háttérből, mondjuk egy adatbázis segítségével. Ha a felhasználó kitölt egy űrlapot(formot), és az elküldésre kattint a következő nézetben azt szimuláljuk majd, hogyan jelenjenek meg a hibás kitöltésből eredő hibaüzenetek, függetlenül attól milyen adatokat adott meg előzőleg.

5.2. HTML prototípus elkészítése

A prototípus gyártását felgyorsíthatjuk különböző eszközökkel, például olyan sablon nyelvekkel amelyek HTML kimenetet állítanak elő. Én erre a Haml-t[6] használtam kombinálva egy statikus HTML-t előállító rendszerrel, a Webby[7]-vel.

6 http://haml.hamptoncatlin.com/
7 http://webby.rubyforge.org/

A Webby egy Ruby alapú mikro-cms: a szöveges fájlokat - amelyek tartalmazhatnak beágyazott Ruby kódokat (Erb) és tetszőleges HTML kimenetű sablon nyelvekkel (Textile, Markup, Haml) is készülhetnek – HTML oldalakká alakítja.

Mivel támogatja a sablonok szétbontását - használhatunk egységes oldalvázakat (layout), és al-sablonokat is - így nem kell a közös tartalmi elemek - például fejlécek, menük stb. - minden oldalon való elhelyezésével bajlódni.

A webby használatát itt most nem szeretném részletezni, mindösszesen csak amennyit érdemes tudnunk.

Új Webby projekt létrehozásához a **webby** parancsot - amivel létrejön a projekt könyvtárváza - új sablon létrehozásához a **webby create:page oldal_elnevezése** parancsot kell használnunk.

A sablonfájlok elején metaadatokat találunk (lásd 1. lista), amelyek fontos információt nyújtanak a Webby számára miképp alakítsa át a sablont HTML-é.

1. lista Metaadatok a prototípus egy fájljában (webshop_proto/content/index.txt)

```
---
title:  Nyito oldal
created_at:  2008-08-02 14:06:40.000000 -06:00
filter:
  - erb
  - haml
---
```

Itt adhatjuk meg, milyen szűrőket (sablonnyelveket) használunk az oldal leírására valamint azt is, hogy melyik oldalvázba illessze be az oldal tartalmát - az oldalvázakat a **layouts** könyvtárban találjuk. A webáruház prototípusa két oldalvázat használ: alapértelmezettként egy **default** elnevezésűt és egy **admin** nevűt az adminisztrációs oldalakhoz.

A webbyben lehetőség van helperek(mankók vagy segítők) használatára is. Ezek olyan pár soros kódok amelyeket gyakran hívunk meg, akár több különböző oldalon is. Mivel a webbyben a helperek működése megegyezik a Ruby on Rails keretrendszerével, így érdemes minél többször használni őket, mert könnyen átültethetőek, felhasználhatóak.

Én például olyan oldalak esetében használtam, amikor több terméket kellett megjeleníteni: a helper visszaadta a termékek egy tömbjét, iterációs ciklussal végigjárva pedig egyenként megjelenítettem őket a szükséges HTML kóddal. Így ha a HTML kimenet változott kellett nem kellett n-nyi esetben változtatni a sablonokban.

Az elkészített oldalak a **webby** parancs kiadása után az output könyvtárban HTML-é alakítva jelennek meg. Hogy ez automatikusan, minden módosítás után lefusson a **webby autobuild** parancsot kell kiadni.

5.3. Tervezési szempontok

A felületetek tervezésekor elsősorban funkcionális szempontok vezéreltek.

Minden fontos információ, és a gyakori műveletek egy nézeten belül(egy kattintással) elérhetőek legyenek: így került a felső sávba a 'Bejelentkezési vagy Regisztrációs', valamint a részletes vásárlási információkat hordozó menüpontok - a felhasználók itt nyerhetnek bővebb felvilágosítást a rendelés menetéről pl. fizetési és szállítási módok, garanciális feltételek stb.

Hasonló szempontból került a Keresési funkció a fejléc alá, a Kategóriák menüpont pedig jobb oldalra. Itt az éppen kiválasztott, azaz aktuális kategória kiemelve, alkategóriái közvetlenül alá kerülnek, a gyorsabb navigáció érdekében ekkor a főkategóriák ugyanúgy szerepelnek a listában. Abban az esetben, ha a felhasználó elhelyez egy terméket a kosarában a bal oldali menüsávban megjelenítjük annak tartalmát: a termékek mellett a mennyiségeket és az árakat. A mindig látszódó kosárral így könnyedén számon tarthatja a vásárló a rendelési tételeket.

A fejlesztőmunka nagy részét elvégeztük: tisztában vagyunk vele, hogy adatbázis szinten miként tároljuk majd az adatokat, milyen funkciókat érhetnek el a felhasználók sőt működő prototípusunk is van, hogyan nézzen ki a végső alkalmazás, mielőtt azonban elkezdenénk átültetni a terveket Ruby on Rails keretrendszerbe, nem árt ha átnézzük, mit is kell tudnunk róla.

6. A Ruby on Rails keretrendszer

6.1. Mi az a Ruby on Rails?

„Rails is a web-application framework that includes everything needed to create database-backed web applications according to the Model-View-Control pattern.”[3]

A Ruby on Rails - továbbiakban csak Rails - egy Ruby nyelven írt, MVC mintára épülő webalkalmazások fejlesztésére szolgáló keretrendszer, publikus verziója 2004 óta érhető el. Nyílt forráskódú lévén bárki ingyenesen letöltheti, használhatja és szabadon bővítheti, így első megjelenése óta számtalan programozó járult ötleteivel a keretrendszer fejlődéséhez. A projekt vezetését egy többtagú, úgynevezett core-team végzi, amelynek vezetője a Rails megalkotója, David Heinemeier Hansson dán programozó. A szakdolgozat készítésekor elérhető legfrissebb verzió a 2.2-es verziószámot viseli, a dolgozat példája is ezen verzióval készült.

A Rails sok újszerű meglátást, olyan programozástechnikai elveket és módszereket hozott a webfejlesztésbe - DRY[8], Agilis programfejlesztés, Egység Tesztelés stb. - , amelyeket korábban még nem, vagy csak nagyvállalati környezetben alkalmaztak ezen a területen. Sikere több programozót is arra ösztönzött, hogy más programnyelvekbe is átültessék a keretrendszer koncepcióját.

A Ruby sokat köszönhet a Railsnek de ez fordítva is igaz, Ruby nélkül nem lenne Rails.

6.2. Mit tud a Ruby nyelv?[4][5]

„Ruby is simple in appearance, but is very complex inside, just like our human body”[6]

A Ruby nyelvet megalkotója Matsumoto Yukihiro kedvenc programozási nyelvei (Perl, Smalltalk, Eiffel, Ada, és a Lisp) ötvözéseként 1995-ben hozza létre. Szülőhazájában népszerű, a világ többi részein kevésbé ismert. A Railsnek köszönhetően változik a helyzet: elismert nyelvvé válik. A Railshez hasonlóan nyílt forráskódú, szabad szoftver.

A Ruby jól átgondolt, alapos tervezés eredménye, most megpróbálom kiemelni a nyelv kulcselemeit, amelyek feltétlenül szükségesek a Rails keretrendszer és a szakdolgozat példájának a megértéséhez.

8 Dont Repeat Yourself – kerüld az ismétléseket elv

6.2.1. A Ruby építőkövei

A Ruby interpretált script nyelv, működéséhez a Ruby értelmezőre (fordítóra) van szükség, amelynek jelen pillanatban elérhető, legfrissebb stabilan működő változata az 1.8.7-es verziószámot viseli.

A Ruby szintaktikája (lásd 2. lista) jelentősen eltér a C-s gyökerű programnyelvektől, leginkább a Perl, Python, Pascal vonalat képviseli: a kapcsos zárójelek helyett kulcsszavakat használ.

2. lista A hello world Ruby nyelven

```
class A
  def hello_world()
    puts 'Hello világ'
  end
end
A.hello_world() => 'Hello világ'[9]
```

Változódeklarálás nincs, sőt Rubyban nincsenek változótípusok sem, helyettük alaposztályok léteznek, lásd az 1. táblázatot.

1. táblázat. A Ruby nyelv alaposztályai

Típus	Osztály	Jelölés
Szám	Fixnum v. Bignum	42 vagy 1_000_000
Szöveg	String	'ez egy ruby string'
Szimbólum	Symbol	:szimbolum_nev
Tömb	Array	[1, 'szoveg', :szimbolum]
Szótár, Hash	Hash	{ :kulcs => ertek }
Sorozat, Tartomány	Range	('a'..'z'), (1..42)
Reguláris kifejezés	RegExp	/reguláris kifejezés/

A feltételes elágazások hasonlóak a többi nyelvben megszokottakhoz: **if** és párja az **else**, amelyet egy **end** kulcsszónak kell lezárnia. A **case** a többes kiértékeléshez, míg feltételes ciklusokhoz a **while** használható.

Két érdekességet kiemelnék: egyrészt használhatjuk az **if** tagadásaként az **unless** - ha nem jelentésű - kulcsszót, amely az **if not feltétel** kifejezéssel egyenértékű. Ciklusok esetében hasonló funkcióval bír az **until** - amíg nem - kulcsszó.

---

9 Egy kifejezés eredményének a jelzésére a => jel használata terjedt el, a szakdolgozat esetében is ezt a jelölést alkalmaztam.

Ezenfelül egysoros kiértékelés is lehetséges:

a = 5 if a < 4 ,

ahol előbb a feltételt értékeli ki (**a** kisebb, mint négy), és csak annak függvényében hajtja végre az előtte álló utasítást.

Az operátorok megegyeznek a C++ -éval - amelyek felsorolását itt mellőzném - továbbá:

- =~ hasonlító operátor a reguláris kifejezésekhez,
- !~ az előbbi tagadása,
- << elem hozzáadása stringhez, tömbhöz,
- <=> objektumok összehasonlítása.

Az operátorok többsége valójában metódusnak tekinthető, csak az érhetőbb, könnyebb használat érdekében rendelkeznek az operátorok működéséhez hasonlatos szintaktikával[10].

Például a fordító a következő kifejezést:

a + 100 ,

a következőképp értelmezi:

a.+(100) .

Mivel az operátorok metódusoknak számítanak működésük osztályainkban felülírható - a C++-hoz hasonlóan. Ezzel is magyarázható, hogy a Ruby nyelvben nincsenek eggyel növelő(++) és csökkentő operátorok(--), helyettük a + (plusz) és - (mínusz) metódusokat használhatjuk. Például:

a += 1 és a -= 1 ,

amit a fordító megint csak a.+(1) illetve a.-(1) -ként értelmez.

Mindezeken felül lehetséges az úgynevezett párhuzamos értékadás is:

a,b = b,a ,

ahol megcseréli egymással a két elemet.

10 A programozástechnikában ezt syntactic sugar-nak nevezik.

Változók és konstansok

A változótípusok jelölését a 2. számú táblázat tartalmazza.

2. táblázat. Változótípusok jelölése a Ruby nyelvben

$valtozo_nev	globális változó
@@valtozo_nev	osztály változó
@valtozo_nev	példány változó
KONSTANS	egyedül a nagy kezdőbetű kötelező, de a konvenciók szerint végig nagy betűvel írjuk
ez_egy_hosszu_valtozo	informálisan az összetett szavaknál változóknál alul vonás jelet használunk elválasztásként

Érdekesség, hogy maguk a konstansok is változók, sőt értékük futásidőben megváltoztatható(!), a fordító csak hibaüzenetet szolgáltat a helytelen használatról. Ezenfelül az osztályok nevei is konstansoknak számítanak, épp ezért nagy kezdőbetűvel kell írnunk minden osztály nevét.

Metódusok és osztályok

A metódusnevekkel kapcsolatban nem árt ha betartunk pár íratlan szabályt:

- végig kisbetűvel írjuk, a szótöredékeket alul vonás karakter választja el;
- az igaz vagy hamis értékkel visszatérő metódusok neveit ? jel;
- az objektum állapotát megváltoztatóakat ! jellel zárjuk.

A metódusok visszatérési értéke mindig az utolsó kiértékelt kódsor eredménye, a **return** kulcsszó használata így nem kötelező, csak ha szükséges.

Feltétlenül érdemes tudnunk, hogy a konstruktorok meghívása a **new** metódussal történik, definiálásuk az osztályon belül az **initialize** metódussal lehetséges. Öröklődést, származtatást a < jellel definiálhatunk.

Az osztályok és a példányok változói – amelyeket szakszerűbben inkább attribútumoknak nevezünk – az osztályokon kívülről nem, csak az úgynevezett olvasó/író metódusok definiálása után érhetőek el. Mivel ezek a metódusok a leggyakrabban csak egysoros metódusok, hiszen egyszerűen visszaadják, vagy felülírják az attribútum értékét, rövidítésükhöz a Ruby biztosít három segédmetódust[11], lásd 3. lista.

3. lista Attribútumok olvasása és írása

```
class A
  # Magunk definiált író és olvasó metódusok

  def nev
    @nev
  end

  def nev=(uj_nev)
    nev=uj_nev
  end

  # A nyelv nyújtotta író és olvasó metódusok

  attr_reader :eletkor       # olvasható attribútum
  attr_writer :eletkor       # írható attribútum
  attr_accessor :eletkor     # olvasható és írható attribútum
end
```

Természetesen szabályozhatjuk a metódusok hozzáférését - a C++-hoz hasonlóan - a **public**, **protected**, **private** kulcsszavakkal. Arra azonban ügyeljünk, hogy ezeket nem metódusonként kell megadnunk: a kulcsszó megadását követően az összes metódus felveszi a hatókört, és csak egy újabb kulcsszó megadásával tudjuk felülbírálni a korábbit.

Osztálymetódusok - amelyek példányosítás nélkül is meghívhatóak - definiálásához a metódus neve előtt jeleznünk kell, hogy az osztályunkhoz tartozik. Ezt vagy az **OsztalyNev.metodus_nev** vagy a **self.metodus_nev** formában tehetjük meg, meghívásuk pedig a **OsztalyNev.metodus** formával lehetséges.

A dokumentációkban is ez utóbbi jelöléssel találkozhatunk, a példány-metódusokat ezzel szemben a **Osztaly#metodus** formával jelölik. A szakdolgozat esetében is ezeket a jelöléseket igyekeztem követni.

Modulok

A modulok az osztályokhoz közelálló, egységbezáró jelölések. Tartalmazhatják több osztály definícióját, konstansokat, metódusokat, de nem példányosíthatóak.

11 Ezek az úgynevezett setter és getter metódusok.

A moduloknak van egy nagyon fontos tulajdonságuk, betölthetik őket más osztályok[12], így terjesztve ki saját működésüket. Például a Ruby Comparable modulja olyan metódusokat tartalmaz, amelyekkel két objektum összehasonlítását végezhetjük el (lásd 4. lista). A modulok kontextusában az egymás alatt lévő osztályok, és változók bejárására a :: jel használható.

4. lista Az Enumerable modul használata

```
class Person
  attr_reader :age

  def initialize(name,age)
    @name,@age = name,age
  end

  include Comparable

  # A '<=>' definiálásával elérhetvek lesznek
  # az összehasonlító operátorok
  def <=>(masik_obj)
    self.age <=> masik_obj.age
  end
end

ati,peti = Person.new("Attila",22), Person.new("Péter",26)
ati < peti  # => true
ati > peti  # => false
ati == peti # => false
```

6.2.2. A Ruby magasabb szintjei

Szimbólumok

A szimbólumok - jelölésüket lásd az 1. táblázatban - speciális string típusok és szorosan kapcsolódnak a Ruby belső működéséhez, tulajdonképpen hivatkozásokat jelentenek. A **Hash** típus esetében pl. a kulcsok megadására, és hivatkozására szolgál. De szimbólumként megadhatóak az osztályok és a metódusok nevei is.

A Rails keretrendszer intenzíven használja a paraméterek hash-ként való átadását metódusoknak, így a különböző opciókat legtöbbször szimbólumok jelölik.

Blokkok

A Ruby fontos nyelvi elemét jelentik a blokkok, amelyek tulajdonképpen a metódusoknak átadott kódrészletek. Segítségével egyszerűen tudunk iterációs, ciklusos feladatokat megvalósítani.

12 Gyakran használják erre a 'mixin' kifejezést.

Blokkokat a paraméterlista után a **do end** kulcsszavak, vagy a **{ }** -ek közé szúrva adhatunk meg. A blokkoknak is vannak paraméterei ezeket a pipe(cső) | | jelek közé illesztjük, értékeiket mindig az őket meghívó metódustól kapják. Példa egy blokkra:

[1, 2, 3].each **do |n| puts n+10 end** .

Szorosan a blokkokhoz tartozik a **yield** kulcsszó, amely a vezérlés és a paraméterek átadására szolgál a metódusból az átadott blokknak (lásd 5. lista).

5. lista A yield kulcsszó használata

```
class A
  def each(szam_egy,szam_ketto,szam_harom)
    yield(szam_egy)
    yield(szam_ketto)
    yield(szam_harom)
  end
end

A.each(4,1,6) do |szam| puts szam end
=> 4 1 6
```

Kivételkezelés

Kivételeket a **raise** kulcsszó segítségével hívhatunk meg. Elkapásukhoz a **begin** és **end** definíciós blokk közé kell tett tennünk a forráskódunkat, lekezelésükhöz a **rescue** kulcsszó használható (lásd 6. lista).

6. lista Kivételkezelés

```
begin
  1 / 0
  rescue ZeroDivisionError=>e
    puts 'A nullával való osztás nem értelmezett!'
    raise e #Tovább küldés
  end
end
```

A Java nyelvhez hasonlóan a kivételek itt is osztályok, az **Exception** osztály leszármazottjai.

Lambda és Proc[7]

A **Lambda** és a **Proc** ha egyszerűen akarnánk megfogalmazni olyan kódrészletek, amelyeket paraméterként átadhatunk és meghívhatunk metódusokban.

Ami igazán érdekessé teszi őket, hogy a létrehozásuk kontextusában lévő változókat használják fel meghívásukkor, nem az őket meghívó metódusét. Mintha becsomagolnánk egy kódrészletet az aktuális változóinkkal, majd kicsomagolnánk egy másikban. A funkcionális programozásban ezeket closure-öknek nevezik.

7. lista A Proc és lambda használata

```
x = 5
p = Proc.new { puts x }
l = lambda { puts x }
def x(p,l)
  x=10
  p.call
  l.call
end
x(p,l)
=>5 5
```

A 7. lista alapján a lambda és a Proc használata és működése között látszólag nincs különbség, ami megkülönbözteti őket az az argumentumok használata. Míg előbbi megköveteli, hogy pontosan annyi argumentumot adjunk át neki ahányat a funkciónk vár - ellenkező esetben ArgumentError kivételt dob - , utóbbi esetében viszont nincs ilyen megkötés.

Ha megfigyeljük a closure-ök szintaktikája és működése nagyon hasonlít a blokkokéhoz és ez nem véletlen. A blokkok is a lambda elvén működnek (lásd 8. lista).

8. lista Blokkok használata lambdaként

```
def add(k,l,m, &blk )
  blk.call(k+m+l)
end

add(1,2,3, &lambda {|x| puts x})    #A & jellel adhatunk át blokkot
                                    # paraméterként metódusnak
=> 6
```

A procedúrális programozáshoz szokottak számára a closure-ök nehezen értelmezhetőek, ugyanakkor ahogy a Ruby, úgy a Rails is előszeretettel használja ezeket bizonyos esetekben, így működésük megértése nélkülözhetetlen.

A Ruby dinamizmusa

A Ruby dinamikus nyelvnek számít, mert futás közben az osztályok bármikor kibővíthetőek, metódusaik felülírhatóak. A nyelv számtalan olyan módszert felajánl, amellyel objektumaink és osztályaink működését futásidőben befolyásolhatjuk. Ezek felsorolása és jellemzése meghaladja e dolgozat kereteit, de annyit mindenképp érdemes megjegyezni, hogy a Rails előszeretettel alkalmazza ezeket.

Minden objektum

Az, hogy egy nyelv objektum orientált, és benne objektumokkal dolgozunk elcsépelt szó. A Rubyban viszont valóban minden objektum: így az igaz(true), hamis(false), és nil érték az őket reprezentáló osztály egyetlen példányai, valamint minden osztály önmaga is egy objektum - a **Class** osztály egy példánya.

Osztályok kontextusán kívül megadott definícióknak is van egy szülőosztálya, a **Kernel** osztály. A Rubyban épp ezért objektum orientált szemszögből nézve nincsenek függvények csak metódusok.

6.2.3. Ruby segédprogramok

irb – Interatív Ruby Shell

Az irb használatával gyorsan és egyszerűen próbálhatunk ki ruby kódokat, anélkül, hogy fájlokat kéne írnunk és futtatnunk rajtuk a ruby interpretert (lásd 9. lista). Meghívása a parancssorból az irb paranccsal történik, ahol minden soremelés leütése után elvégzi a kód kiértékelését, eredményét pedig a standard kimenetre írja.

9. lista Az irb demonstrálása

```
$ irb
irb(main):001:0> 5.class
=> Fixnum
irb(main):002:0>
```

Rake[13]

A Rake létrehozásával alkotója a GNU Make[14]-t - a Unixos környezetben jól ismert, programok forráskódjának kezelésére használt segédprogramot - szerette volna átültetni Ruby nyelvre. Mivel ruby programok írásakor nincs szükség futtatható állományok generálására, így a raket inkább azok telepítésére valamint gyakori feladatok rögzítéséhez és futtatásához használjuk.

A Rake segítségével tulajdonképpen rubyban írt kódokat futtathatunk, ezeket rake taszkoknak hívjuk. A taszkokat lehetőség van névterekbe ágyazni, így csoportosítva az egymással összefüggőeket. A Rails a rake taszkokat adatbázissémák karbantartására, adatok feltöltéséhez valamint tesztek futtatásához használja.

13 http://rake.rubyforge.org/
14 http://www.gnu.org/software/make/

Rubygems[15]

A Rubygems a Ruby saját csomagkezelője: egységes felületet nyújt függvénykönyvtárak valamint programok telepítésére, törlésére és létrehozására. A csomagkezelő figyelembe veszi egy csomag függőségeit, és automatikusan előállítja a hozzá tartozó dokumentációt is. Többek között a Rails is a Rubygems-szen keresztül telepíthető a legegyszerűbben:

```
$ gem install rails
```

Ezzel kanyarodjunk is vissza a Ruby on Rails keretrendszerhez.

6.3. A Rails az MVC szemszögéből[8]

Bár az MVC szemlélet már az 1980-as évektől létezik, mégis csak az utóbbi években kezdték el programozók szélesebb körben alkalmazni. Az MVC elnevezés három komponense kezdőbetűiből tevődik össze:

M - Model - Modell

A modell feladata az alkalmazás állapotának megőrzése, ami lehet, hogy csak addig a pár pillanatig tart amíg a felhasználóval kommunikálunk, de az is lehet, hogy tartósabb és egy adatbázisban való adattárolást jelent. A modell ugyanakkor nem csupán puszta adathalmaz; tartalmazza mindazon logikákat, amelyek hozzá társulhatnak - például, hogy milyen értékeket vehetnek fel az adatelemei, azokat hogyan formázhatjuk.

V - View - Nézet

A nézet felelős a felhasználói interfész generálásáért, az alkalmazással való kommunikáció megteremtéséért. A nézet legtöbbször a modelltől vett adatokkal dolgozik és jeleníti meg azokat a felhasználó számára olvasható formában, vagy épp lehetőséget teremt új adatok bevitelére. Látszólag a nézet tölti be a legkisebb feladatot a három komponens közül, de elválasztása a többi egységtől lehetőséget teremt arra, hogy eltérő formátumú kimeneteket hozzunk létre anélkül, hogy az alkalmazás többi részén jelentősen változtatnunk kellene. (Például a legfrissebb termékeket nem csak a weboldalunkon, hanem egy hírcsatornán keresztül is közzé tehetjük).

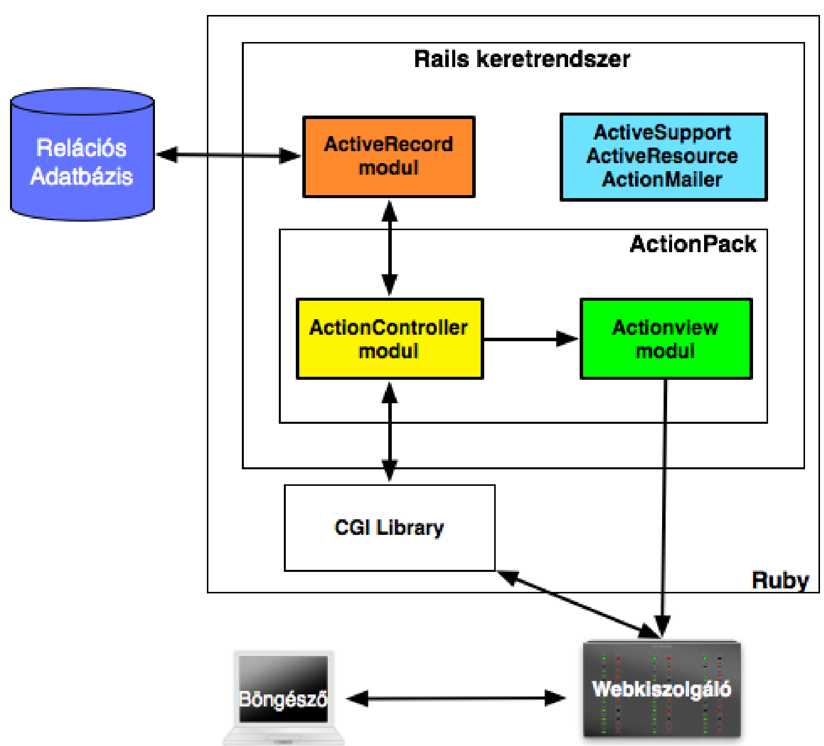
15 http://www.rubygems.org/

C - Controller - Vezérlő

A vezérlő felel az alkalmazás működéséért: kéréseket dolgoz fel, amelyek leggyakrabban a felhasználói felülettől tehát a nézeteinktől származnak, és adatszerzést végez, tehát egy modellel kommunikál. A kérés végeztével pedig válaszként egy újabb nézetet jelenít meg. A vezérlő ily módon összefogó szerepet tölt be a másik két MVC modul között.

Az MVC minta komponensei remekül illeszkednek a webalkalmazások fejlesztéséhez, hiszen a HTTP protokoll miatt mindig nézeteket (legtöbbször HTML oldalakat) kell előállítani, a beérkezett kéréseknek megfelelően (Vezérlő), amelyekhez leggyakrabban valamilyen adatbázisban tárolt adat (Modell) kapcsolódik. Az MVC előnyeit több más webprogramozó is felismerte, így sorra születnek meg a különböző nyelveken a különféle implementációk (Pl. PHP - CakePHP, Python - Django, Ruby - Rails és Merb).

A Rails megvalósításában szintén az MVC minta moduláris felépítését követi, olyannyira, hogy ezeket különálló (Ruby) modulokban implementálták: az ActiveRecord felelős az adatbázis alapú modellért, az ActionView a nézetek megjelenítését, az ActionController pedig a vezérlő rész feladatait látja el - mivel ez utóbbi kettő szorosan összefügg egymással az ActionPack nevű modul fogja össze őket - lásd a 11. ábrát.



11. ábra. A Ruby on Rails keretrendszer moduljai

6.3.1. Az ActiveRecord modul - a Rails 'M'-je[9]

A webalkalmazások többsége az adatok tárolására valamilyen relációs adatbázist használ, az adatok kezelését pedig SQL nyelven keresztül végzik. Ez utóbbiak legtöbbször egymáshoz nagyon hasonló parancsok megírását jelentik, ezért az adatbázissal való kommunikáció leegyszerűsítésére már a kezdetektől fogva léteznek interfészek. Ezek egyik, objektumorientált megvalósítása az ORM[16]-nek nevezett elméleti megközelítés.

Az ORM elve egyszerűen fogalmazva egy-egy osztályra képezi le az adatbázis táblákat, a tábla mezőit pedig az osztály attribútumainak felelteti meg. Az osztályokból létrehozott példányok így a tábla egy-egy rekordjának felelnek meg. Ezt a koncepciót terjeszti ki az Active Record[17], mint elv azzal, hogy lehetővé teszi az osztályok(táblák) közötti kapcsolatok egyértelműbb létrehozását, valamint kihasználja az öröklődést, és az osztályok feladataként definiálja a rekordokhoz tartozó logikákat - ellenőrzés, megjelenítés, keresési funkciók. Ily módon egységesen modellezhetjük adatbázisunk szerkezetét.

Mindezen koncepciók Ruby alapú megvalósítása, a Rails Modell részének alapértelmezett adatbázis alapú interfésze az ActiveRecord modul[18].

Az ActiveRecord adaptereken keresztül kommunikál az adatbázissal, az egyes adatbázis kezelőkhöz (MySQL, PostgreSQL, SQLite, Oracle) más más adapter[19] tartozik.

Megfeleltetések

Minden ActiveRecord típusú modell egy egy ruby osztály, az ActiveRecord::Base osztály leszármazottja lesz. Használatukhoz az adatbázis kapcsolat megadásán kívül további konfigurációra nincs szükség, hiszen tábláinkat automatikusan leképezi a megfeleltetései[20] alapján:

- Az modellek osztálynevei mindig az angol nyelv egyes, a tábláké a többes szám szabályait követik. Ebből tudja az **Product**(Termék) osztály, hogy hozzá a **products**(termékek) tábla tartozik.
- Elsődleges kulcsnak az **id** mező felel meg.

16 Object Relational Mapping - Objektum Relációs Leképezés
17 Az Active Record eredeti koncepciója Martin Fowler-től származik, lásd http://www.martinfowler.com/eaaCatalog/activeRecord.html
18 http://ar.rubyonrails.org/
19 A Rails 2-es verziójától ezeket az adaptereket különállóan, mint rubygemek kell telepítenünk, nem részei az ActiveRecord modulnak.
20 A Rails keretrendszer fontos elve, hogy konfigurációk helyett konvenciókat alkalmaz.

- Idegen kulcsok elnevezése az **osztálynév_id** formát követi. A fenti példánál maradva a **Product** modellekhez kapcsolódó kategóriákat (**Category** modell és osztály) a **category_id** mezőben lévő idegen kulcs értékével fogja keresni.
- A mezőnevek nem tartalmazhatnak nagybetűket, a szókapcsolatokat lehetőleg alulvonással jelöljük, ahogy tennénk azt rubyban is. Például így: egyseg_ar. A mezőnevek ugyanis példány-attribútumokká válnak.

A megfeleltetések természetesen felülbírálhatóak, akár teljesen el is térhetünk tőlük, de az ActiveRecord által helyesnek gondolt működés betartásával minden alkalmazás esetén koherens modelleket kapunk.

Az ActiveRecord::Base osztály szerepe

Mivel az ActiveRecord szemlélete eleve igyekszik minimalizálni az SQL lekérdezések használatát, ezért az osztályok és objektumaik - a leszármaztatásnak köszönhetően - számtalan metódussal egészülnek ki.

Rekordok lekérdezésére a **find** metódus használható, amely igen széles körben paraméterezhető:

- ha egy konkrét termékre vagyunk kíváncsiak, egy azonosítót adunk meg:

  Product.find(10) .
- ha egy vagy több terméket szeretnénk megkeresni, adott feltételekkel:

  Product.find(:first | all, :conditions => {:name=>'Rails könyv'}) .

Mivel a feltétel alapú lekérdezésekre gyakran van szükségünk az ActiveRecord definiál néhány kereső metódust az attribútumainkkal is. Ezeket dinamikus, attribútum alapú kereséseknek nevezzük: a **find_by** és **find_all** metódusok[21]). Például a fenti példával egyenértékű a **Product.find_by_name('Rails könyv')** használata.

Akármelyik kereső metódust is használjuk, az a keresés rekordjait mindig az adott osztály objektumaként vagy objektumok tömbjeként adja vissza.

21 Valójában ezek a metódusok nincsenek előre definiálva csak - kihasználva a Ruby képességeit - az ActiveRecord megpróbálja a meghívott metódust értelmezni, innen a dinamikus jelző.

Az objektumok a hozzájuk tartozó rekord mezőit mint attribútumok tárolják, ezek - mezőtípusuknak megfelelően - ruby alaptípusokat vesznek fel. A szöveges típusok (text, char) **String**ek, a számok(int, float, decimal stb.) **Fixnum** típusúvá válnak. Az attribútumok kiolvasásához és beállításához - tehát a rekord módosításához - az ActiveRecord a mező nevével hoz létre író-olvasó metódusokat, így például a

Product.find(1).name=('Ruby könyv')

metódus beállítja, a

Product.find(1).name

pedig visszaadja a name elnevezésű mező tartalmát.

Az ActiveRecord asszociációi

Fentebb már említettem, hogy az Active Record koncepciója eredetileg az ORM kapcsolat-kezelési hiányosságának kiküszöbölésére született, azaz, hogy az adatbázis táblái között lévő kapcsolatok megjelenhessenek a modellek (osztályok) között is.

Az ActiveRecord modul ennek megoldására asszociációs metódusokat definiál, amelyeket makrószerűen, a modell osztályának törzsében kell meghívni, jelezve neki, hogy milyen más osztályra (modellre) hivatkozunk.

Az asszociációs metódusokat a kapcsolattípusoknak megfelelően definiálja.

1-1 kapcsolat

Egy-egy kapcsolatot kétféleképpen adhatunk meg, a **belongs_to** (tartozik … -hoz) vagy a **has_one** (van neki egy ...) metódusokkal. Azt, hogy épp melyiket használjuk az idegen kulcs helye határozza meg: ha a modellünk táblájában található a hivatkozás a másik rekordra akkor a **belongs_to** metódust, ellenkező esetben a **has_one**-t kell használnunk.

1-N kapcsolat

Egy-több kapcsolat esetében az átjáráshoz egy **has_many** (van neki több ...) asszociációt a hivatkozó, egy **belongs_to**-t a hivatkozott modellben kell megadnunk - hiszen ez utóbbiban található a hivatkozóra mutató idegen kulcs.

N-N kapcsolat

Több-több kapcsolatot kétféleképpen határozhatunk meg. Egyrészt használhatjuk a **has_and_belongs_to_many** asszociációt, ahol egy kapcsoló táblára lesz szükségünk - ennek elnevezése az osztálynév_osztálynév formát kell, hogy kövesse és nem tartalmazhat elsődleges kulcsot, valamint nem tartozhat egy modellhez sem.

Viszont abban az esetben, ha szükségünk van a köztes táblára, mint modell - például, hogy ellenőrzéseket társítsunk a rekordokhoz mentés előtt - a **has_many ... :through => …** (van neki több … -va/ve … -n keresztül) asszociációt kell használunk. Itt a :through opcióval közvetetten kapcsolódunk a köztes, úgynevezett kapcsolat (join) modellhez, amely összeköti a másik két hozzá kapcsolódó modellt.

Az asszociációs metódusok után szimbólumként kell megadnunk a hivatkozott osztály (modell) nevét, itt is figyelve néhány megfeleltetésre: ha csak egy objektumra (rekordra) hivatkozunk egyesszámot, ha több objektumot érünk el többeszámot kell használnunk.

A kapcsolatok megadása után a Rails dinamikusan hozzáad a modell osztályához metódusokat - a kapcsolat típusától függően -, amelyekkel megadhatjuk és elérhetjük a hivatkozott modelleket.

Validációk

Az MVC minta értelmében adataik helyességéről a modelleknek mindig önmaguknak kell gondoskodniuk, így az alkalmazásba bevitt adatok ellenőrzéséért elsődlegesen a modellek a felelősek.

Az ActiveRecord modellek esetén az adatok ellenőrzéséhez kétféle megoldást is használhatunk.

Először is felüldefiniálhatjuk a modell osztályának **validate** metódusát, ahol magunknak kell elhelyeznünk mindazon logikákat, amelyek ellenőrzik, hogy a modell attribútumai megfelelő értékűek. Ezeket alacsony szintű ellenőrzéseknek nevezzük.

A gyakori ellenőrzési formákhoz - ilyen lehet például, hogy egy adott attribútum értéke nem üres-e, vagy megfelel-e adott formátumnak - a Rails definiál magasabb szintű ellenőrzéseket, amelyeket az asszociációkhoz hasonlóan makrószerűen tudunk meghívni. Ezek dokumentációját megtaláljuk a **ActiveRecord::Validations::ClassMethods** modulban, a legfontosabb magasabb szintű ellenőrző metódusok:

- validates_presence_of :attributum, …

  adott attribútum(okat) kötelezően meg kell adni létrehozáskor.
- validates_numericality_of :attributum, (opciók)

  szám típusú attribútumok vizsgálata.
- validates_uniqueness_of :attributum, :scope => …

  az attribútum nem vehet fel a táblában már létező értéket, a scope opcióval több attribútumot is figyelembe vehetünk.
- validates_inclusion_of :attributum, :in=>(felsorolas)

  ellenőrzi, hogy az attribútum értéke szerepel-e a felsorolásban.
- validates_exclusion_of :attributum, :in=>(felsorolas)

  az előző ellentéte.
- validates_format_of :attributum, :with=>/reguláris kifejezés/

  összeveti az attribútumot a reguláris kifejezéssel.

Az ActiveRecord minden tulajdonságának felsorolása meghaladja e szakdolgozat kereteit, de mint az a fentiekből talán kiderült, ha adatbázistámogatásra van szükségünk Rails alapú alkalmazásunkban akkor az ActiveRecord kiforrott megoldást kínál. Ugyanakkor szó sincs arról, hogy minden modell esetében ragaszkodnunk kellene a használatához, hiszen előfordulhat, hogy nincs is szükségünk adatbázis támogatásra bizonyos modellek esetében.

6.3.2. Az ActionPack modul

Az ActionPack modul feladata a beérkező kérések átadása egy vezérlőnek feldolgozásra, majd az előállított nézetek megjelenítése válaszként.

Vezérlők

A vezérlők a modellekhez hasonlóan ugyancsak osztályok - az **ActionController::Base** osztály leszármazottjai, amely sok háttérfeladatot elvégez helyettünk: többek között ruby objektumokká alakítja a kérésnek átadott paramétereket és munkameneteket, valamint gondoskodik az adatok átadásáról a nézetek számára.

A vezérlőket általában a modellekhez társítjuk, például külön egyet a termékek kezeléséhez és külön a felhasználókéhoz. A különböző feladatok szétbontása a metódusok definiálásával történik.

Mivel ezek leggyakrabban az adott modellen végzett négy „alapműveletet” jelentik, elterjed rájuk a CRUD[22] kifejezés, és a metódusok uniformalizált elnevezése: index - összes elem lekérdezése, show - egy elem lekérdezése, update - elem frissítése, delete - egy elem törlése.

Egy vezérlő metódusának meghívását és a kapcsolódó nézet megjelenítését eseménynek hívjuk, meghívásuk az URL-ek alapján történik. Azt, hogy egy URL melyik vezérlő, melyik metódusát hívja meg azt az ActionPack az előre definiált útvonalak alapján dönti el - a Railsben az útvonalak létrehozását a **config/routes.rb** fájlban tehetjük meg.

Az útvonalak megadásakor egy sztringgel mintát illesztünk az URL-re - ahol a paramétereket a kettőspont jellel különböztetjük meg, mivel ezek változóként elérhetőek lesznek - majd megadjuk a meghívandó eseményt. A bejövő kérések útvonal keresése mindig az első találatig tart, ezért fontos a megfelelő sorrend betartása.

A Rails a kéréseket alapértelmezettként a vezerlo/akcio/:id formában értelmezi, így a **http://pelda.hu/products/show/2** kérés a ProductsController osztály show metódusát hívja meg, a paraméterek között pedig az id kettes értékkel fog szerepelni.

Lehetőségünk van továbbá, hogy az útvonalakhoz egyedi elnevezést is rendeljünk - ennek a nézetekben az útvonalak létrehozásakor van jelentősége.

Szorosan az események meghívásához kapcsolódik, hogy a Rails keretrendszer a 2-es verziójától inkább a REST[23] elvű alkalmazások létrehozását támogatja a hagyományos CRUD elvű események mellett.

Rails és a REST[10]

A REST elvei szerint a webalkalmazásunk egyes állapotait és funkcióit különböző erőforrások írják le, amelyeket egyedi azonosítókkal érhetünk el - a HTTP protokoll esetében ezt a szerepet az URL tölti be. A REST elve ugyanakkor a HTTP metódusok - **GET**, **POST**, **PUT**, **DELETE** - szigorúbb használatával leegyszerűsíti az eléréseket.

22 CRUD - Create Read Update Delete
23 A REST (Representational State Transfer) elvét Roy Fielding fogalmazta meg doktori disszertációjában.

A hagyományos elgondolás esetében míg az URL-ek magukban hordozzák a vezérlőt, az eseményt, és valamilyen paramétereket, (például egy rekord azonosítóját) és legjobb esetben is csak a GET és a POST metódust használják ki, addig a REST elvben az erőforrásainkat a HTTP metódusok és URL-ek kombinációjaként érhetjük el. Az 3. táblázat összefoglalja a hagyományos és a REST elvű URL-ek különbségeit: jól látható, hogy a HTTP metódusok helyes használatával nemcsak kevesebb webcímre van szükségünk, de a metódus neve magában hordozza a művelet célját is.

3. táblázat. A REST elvű és a hagyományos webcímek különbsége

REST		**Művelet**	**Hagyományos**	
Http metódus	**URL**		**Http Metódus**	**URL**
GET	/products/	Listázás	GET	/products/index
GET	/products/1	Lekérdezés	GET	/products/show/1
POST	/products	Létrehozás	POST	/products/create
PUT	/products/1	Felülírás	POST	/products/edit/1
DELETE	/products/1	Törlés	GET	/products/delete/1

Fontos megértenünk, hogy a PUT és DELETE metódusok mindig is a HTTP protokoll részét képezték és eredetileg is a REST által definiált funkciót rendelték hozzájuk, de a webböngészők és a webalkalmazások programozói elfeledkeztek ezek használatáról.

A Rails keretrendszerben az erőforrások egy modell és vezérlő kombinációjának felelnek meg, ezért használatukat, mint útvonalak kell definiálni, a keretrendszer ugyanis ezek és a kérés HTTP metódusa[24] alapján hívja meg a megfelelő CRUD eseményt, ezt a 4. táblázat mutatja meg.

A Rails ezenfelül használ két segédmetódust is erőforrások esetén az adatok bevitelére szolgáló HTML nézetek előállításához: a GET HTTP metódussal elérhető **new**t és **edit**et.

24 Mivel a böngészők (egyelőre) nem támogatják a PUT és DELETE metódusokat a Rails előbbit egy rejtett űrlapmező használatával, utóbbit pedig JavaScript segítségével oldja meg.

4. táblázat. Az események meghívása az erőforrások címei alapján

Http metódus	URL	Meghívott vezérlő	Meghívott metódus	Paraméterek
GET	/products/	Products	index	-
GET	/products/1	Products	show	id=1
POST	/products	Products	create	küldött paraméterek
PUT	/products/1	Products	update	küldött param. és id=1
DELETE	/products/1	Products	destroy	id=1

Bár a Rails igyekszik minél inkább a REST elv használatára ösztönözni, sok esetben felesleges, és bonyolult megoldás az erőforrások használata, ezért érdemes mindig jól átgondolni, hogy szükségünk van-e rá.

Nézetek előállítása

Az eseményekhez tartozó nézetek előállítását az ActionPack modulon belül az ActionView modul végzi, amely a HTTP kérésnek megfelelően (Accept fejléc mező) generálja a választ: nekünk csak elég megadni, hogy adott esemény milyen válaszokat állíthat elő, a többit a Rails automatikusan elvégzi. Ez lehetővé teszi különböző formátumok - HTML mellett például XML és JavaScript válaszok - előállítását ugyanazon eseményhez.

A HTML nézetek előre megírt sablonokból állnak, ezek alapértelmezettként HTMLt tartalmaznak, a dinamikus tartalmakat pedig Erb[25]-be ágyazott ruby kód állítja elő: az Erb **<% %>** jele kiértékeli, a <%= %> pedig kiírja a kifejezés visszatérési értékét. A nézetekben csak a vezérlő által létrehozott példányváltozókat (@valtozo) tudjuk használni.

Az azonos tartalmak létrehozásához, és a kódismétlés elkerüléséhez az ActionView lehetővé teszi, hogy a nézeteinket szétbontsuk és alsablonokat hozzunk létre. Az oldalvázakkal (layoutok) ugyanakkor egységes sablonokat hozhatunk létre több esemény között: tipikusan ilyen probléma amikor az oldal egy adott szegmensébe mindig különböző tartalmat töltünk be a többit változatlanul hagyjuk.

25 http://www.ruby-doc.org/stdlib/libdoc/erb/rdoc/

Az ActionView modul ezenfelül több olyan osztályt és metódust is tartalmaz, amellyel gyorsabbá és kényelmesebbé válik a HTML (és XML) nézetek előállítása: ilyenek az űrlapelemek, linkek, képek stb. HTML tag-jeit előállító és formázó metódusok.

6.4. A Rails további moduljai

Az ActiveRecordon és az ActionPackken kívül több kisebb modul is kapcsolódik a keretrendszerhez: az ActiveResource például a REST elv egyes részeinek megvalósításért felelős, az ActionMailer emailküldési megoldásokat tartalmaz, az ActiveSupport pedig segédmetódusokkal egészíti ki a Ruby alaposztályait. Ezeket a modulokat a Rails külön csomagokban, mint rubygemek telepíti.

Mindez miért is hasznos? Azzal, hogy modulok a másik megléte nélkül is működőképesek különállóan is használhatjuk őket. Ez például az ActiveRecord esetében azt jelenti, hogy rubys programjainkban, mint komplett ORM megoldást használhatjuk[26].

A Rails tehát tulajdonképpen a különálló modulok összefogása: a keretrendszer minden szükséges feladatot elvégez helyettünk - a kérések átirányítását, adott események meghívását stb. - így fejlesztőként csak az alkalmazás felépítésére kell koncentrálnunk.

26 A szintén Ruby nyelven írt MVC keretrendszer, a Merb is ezt teszi.

7. Alkalmazás áttervezése Rails környezetre[3]

Az előző fejezetben láthattuk, hogy milyen megfeleltetéseket használ a Rails. Ennek alapján az eredeti terveken pontosítani kell, ez elsősorban az adatbázis sémáját érinti. Hogy ne kelljen a felülbírálásokkal foglalkozni a táblázatok és mezőnevek elnevezésekor az angol megfelelőiket fogom használni, előbbieknél természetesen a többesszámú alakot.

7.1. Modellek

Minden táblához egy-egy modell tartozik majd, kivéve a termékeket a címkékkel párosító kapcsoló táblánál - ez nem igényel külön modellt. A kategóriajellemzőket a termékjellemzőkkel összekapcsoló táblázat ugyanakkor igen, az értékek megjelenítése miatt. A rendelések és a felhasználók címeit ugyanakkor egy közös táblára bontjuk, így egyszerűbb lesz a címekhez hozzárendelni egy közös modellt ami elvégzi az adatok ellenőrzését. Ráadásul a későbbiekben így egyszerűbb lesz bővíteni a rendszert is új címmezők megadásakor.

7.2. Útvonalak és vezérlők megtervezése

A kategóriákat a **CategoryController**, a termékeket a **ProductsController** vezérlő fogja kezelni, ezek REST elvű erőforrásokat fognak használni. Az előbbi feladata csupán a kategória termékeinek a kilistázása, utóbbié pedig a termék részletes adatainak a megjelenítése lesz.

A kosarak kezelését a **CartController** vezérlő fogja végezni, itt erőforrások helyett a hagyományos címeket fogjuk használni. Az általános forma a cart/(add/remove/destroy)/:id lesz, ahol az **:id** paraméter a kosárba helyezendő vagy már ott lévő termék azonosítója.

A **UserController** vezérlő a felhasználók központi kezeléséhez kell majd. Mivel itt sincs szükség erőforrások használatára, így a címek a következőképpen alakulnak: a user/login a vásárlók beléptetéséhez, az admin/login az adminisztrációs jogokkal rendelkező felhasználókhoz, a user/logout esemény pedig a felhasználók kiléptetését fogja szolgálni.

A rendelések háromlépcsős folyamatát - rendelési cím megadása, rendelés elfogadása, rendelés rögzítése - az **OrderController** vezérlő irányítja majd, ennek alapján a három fő útvonal az **order/address**, **order/confirm**, és az **order/create** lesz.

Mivel lehetőségünk van, hogy a vezérlőket modulokba soroljuk az adminisztrációs feladatokat különválasztva egy Admin nevű modulban fogjuk össze, itt az egyszerűbb kezelés miatt ismét csak a REST elvű erőforrásokat és vezérlőket használjuk fel.

8. Programfejlesztés Railsben

8.1. Kezdő lépések az alkalmazás létrehozásában

Ahhoz, hogy létrehozzuk alkalmazásunkat a következő parancsot kell kiadnunk:

```
$ rails --database=mysql webshop
```

amellyel létrejön az alkalmazás könyvtárváza. Az **app** könyvtárba kerülnek - könyvtáránkénti bontásban - az MVC komponensek, a pluszként itt szereplő helper könyvtárban a nézetekben használható segédmetódusokat helyezhetjük el. A **config** könyvtár tartalmazza a konfigurációs beállításokat, a **db** az adatbázis sémákat, a **test** a tesztelést végző kódok, a **vendor** pedig külső kódok - például kiegészítők - telepítési helye.

Az alkalmazás létrehozása után meg kell adnunk az adatbázis elérési adatait a config/database.yml[27] fájlban, hogy alkalmazásunk csatlakozni tudjon az adatbázishoz. Itt érhet az első meglepetés minket ugyanis alapértelmezettként minden Rails alkalmazás három adatbázist használ, három különböző környezetének megfelelően: a **development** a fejlesztéshez, új ötletek kipróbálásához, a **test** a tesztek futtatásához, a **production** pedig az alkalmazás éles futtatásához használható. A környezetek lehetővé teszik, hogy különböző feladatköröket határozzunk meg és szétválasszuk az alkalmazás egyes fejlesztési, működési fázisait.

A Rails ennek megfelelően alkalmazásnév_környezet adatbázis elnevezéseket használ (a mi esetünkben mindez **webshop_development**, **webshop_test**, **webshop_production** elnevezéseket jelent).

A beállítások ellenőrzéséhez hozzuk létre az adatbázisokat:

```
$ rake db:create:all
```

Az alkalmazás megvalósítását a modellekkel kezdjük.

27 A .yml kiterjesztés a YAML(Yet Another Markup Language) formátumra utal - lásd http://www.yaml.org/. A Rails alkalmazások tipikusan ezt használják konfigurációk tárolásához.

8.2. Modellek létrehozása

A Rails könyvtárszerkezeténél már említettem, hogy alkalmazásunk MVC komponensei az app könyvtárba kerülnek: a modelleket az app/models könyvtárban kell elhelyezni. A manuális létrehozás helyett azonban tanácsosabb a Rails beépített generátorai támaszkodni, mivel ezek egyúttal létrehozzák a szükséges adatbázis sémákat és a modellek teszteléséhez szükséges fájlokat is.

Modellek létrehozásához a **script/generate** parancs model nevű generátorát kell meghívni.

```
$ script/generate model product name:string price:decimal
```

A modell neve után megadhatjuk milyen attribútumokat szeretnénk hozzá társítani - név:sql_típus formában - erre a modellekhez kapcsolódó táblák létrehozásánál van szükség.

8.2.1. Adatbázissémák migrációkkal

Mivel az ActiveRecord megvalósítása arra törekszik, hogy a fejlesztőnek csak a legvégső esetben kelljen SQL parancsokat használnia, ezért az adatbázissémák létrehozása és módosítása a Railsben migrációkkal történik.

A migrációk ruby nyelven írt parancsok, segítségükkel adatbázisfüggetlenül tudjuk leírni a sémáinkat. A migrációkat bármikor alkalmazhatjuk, és bármelyik táblát valamint mezőt megváltoztathatják, de lehetőségünk van visszavonni is őket, így amolyan verziókezelőként működnek az adatbázissémákhoz.

A migrációs fájlokat a **db/migrate** könyvtárban találjuk, elnevezésük a **szamsor_migracio_neve.rb** formát követi, ahol a számsor rész a migráció létrehozásának időbélyege[28].

A 10. lista mutatja, hogy a migrációk is osztályok, az ActiveRecord modul Migration osztájának leszármazottai és két osztálymetódussal rendelkeznek: az **up** a migráció végrehajtásához, a **down** a visszavonásához szükséges.

28 Az időbélyegek használatával mindig egyedi azonosítók jönnek létre. A rendszer ezért tudja a migrációkat sorrendben alkalmazni, csoportos munkavégzéskor pedig nem kavarodnak össze az eltérő forrásból származó migrációk.

10. lista A termékek táblát létrehozó migráció

```
class CreateProducts < ActiveRecord::Migration
  def self.up
    create_table :products do |t|
     t.string :name
     t.column :price, :decimal, :precision => 8
     t.references :category
    end
  end

  def self.down
    drop_table :products
  end
end
```

Az up metódusban a sémát érintő változtatásokat írjuk le, a down metódusban viszont ennek épp az ellenkezőjét, hogyan tudjuk visszavonni a migrációt: ez tábla létrehozásakor törlést, egy mező törlésekor annak létrehozását, egy mezőnév megváltoztatásakor az eredeti visszaállítását jelenti. A teljes adatbázisséma egy adott pontra való visszaállításakor ugyanis így nem szükséges minden migrációt újra alkalmazni a legelsőtől kezdve, hanem akár vissza is lehet görgeti őket a legutolsótól indítva.

Az up és down metódusokon belül helyezkednek el az úgynevezett transzformációs metódusok, ezek segítségével tudjuk bővíteni vagy megváltoztatni az adatbázis sémáját. A transzformációs metódusok meghatározott paraméterekkel hívhatóak meg - a tábla és mezőnevekre ismét csak szimbólumokkal kell hivatkozni.

Mezőtípusok megadásakor arra kell figyelni, hogy nem az adatbázis-kezelő beépített típusait kell használnunk, hanem az ActiveRecord által definiáltakat! Az elérhető típusok :string, :text, :integer, :float, :decimal, :time, :datetime, :timestamp, :boolean. Így például a **char** és **varchar** típusok helyett a :stringet kell használni.

Tábla létrehozásakor elsődleges kulcsokat nem kell megadnunk, az ActiveRecord migrációi automatikusan hozzáadják azt **id** néven - ezt természetesen felülbírálhatjuk. A migrációik emellett tartalmaznak pár kisegítő transzformációt is: a **timestamps** csatolja a rekord létrehozási és a legutóbbi módosítási idejét tároló mezőket, a **references** transzformáció pedig az idegen kulcsok létrehozására használható.

A migrációk alkalmazásához a Rails beépített rake taszkjai közül a db névtérben lévőket kell használnunk. A legfrissebbre mindig a **db:migrate** parancs hozza a táblákat, a **db:migrate:up**-pal lehetőségünk van előrelépni, a **db:migrate:down**-nal pedig visszalépni egy korábbi sémaverzióra.

A Rails a már alkalmazott sémák számon tartására az alkalmazás adatbázisában egy schema_migrations elnevezésű táblát használ. Az adatbázis aktuális, teljes séma definíció-ját ugyanakkor megtaláljuk a **db/schema.rb** fájlban is, ezt a **db:migrate:load** paranccsal betölthetjük egy üres adatbázisba, így az alkalmazás portolásakor nem kell az összes mig-rációt lefuttatni egyenként.

8.2.2. Kapcsolatok kialakítása a modellek között

Az ActiveRecord leírásánál már bemutattam az asszociációs metódusokat, ezek alkal-mazásával könnyedén kapcsolatot teremthettem a modellek között. A megfeleltéseknek kö-szönhetően konfigurációra nem, a megfeleltetések felülbírálására is csak mindössze né-hány esetben volt szükség.

A kategóriarendszeren belüli kapcsolat

A kategóriák rendszerében a fa-struktúra megvalósításához a Rails beépített támogatá-sát, az **acts_as_tree**(viselkedj úgy, mint egy fa) plugint[29] alkalmaztam, segítségével nem kellett megírni a fát bejáró metódusokat.

Használatához a modellben meg kell hívni az acts_as_tree elnevezésű metódust - az asszociációkhoz hasonlóan - valamint egy parent_id elnevezésű, visszahivatkozó kulcsot kell elhelyezni a modell táblájában.

Az acts_as_tree olyan metódusokat definiál a modellünkhoz, amelyekkel faként járható-ak be az egymáshoz kapcsolódó rekordok, a mi esetünkben ezek a következőket jelentik:

- children - visszaadja egy kategória összes alkategóriáját;
- parent - visszaadja egy alkategória szülő kategóriáját;
- siblings - visszaadja egy kategória szülőkategóriájába tartozó elemeket - magyarul a vele egy szinten lévő kategóriákat;
- root - egy alkategóriából a felette álló legfelsőbb kategóriát érhetjük el.

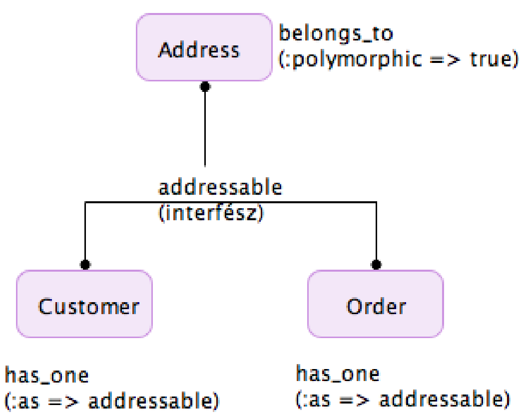
29 A fa struktúra mellett több úgynevezett acts_as megoldás is létezik. Mint ahogy a nevük is sejteti segítségükkel különféle viselkedések imitálhatóak kollekciók esetében.

Címek modellekhez rendelése polimorf kapcsolatként

A rendszer tervezésekor szándékosan különálló táblára bontottuk a címeket - leválasztottuk azokat a rendelések és a vásárlók tábláiról. Ami elsőre jó döntésnek tűnt, most gondba állít minket. Az ActiveRecord ORM megvalósítása miatt ehhez a modellünkhöz (**Address**) két további (**Order** és **Customer**) tartozik **belongs_to** kapcsolattal, ami lehetetlenné teszi egy használható kétirányú kapcsolat létrehozását.

Ezt a problémát kétféleképpen tudjuk feloldani. Egyrészt kapcsolatonként létrehozhatunk egy, a két modellt összekapcsoló köztes táblát valamint egy hozzá tartozó modellt, ami működne is, de további modellek kapcsolódása esetén a címhez, könnyen rémálommá válna a kezelésük. Ezért inkább az ActiveRecord polimorfikus kapcsolatait kell használnunk.

Polimorfikus kapcsolat létrehozásához elsőként a hivatkozott közös modellben kell egy virtuális interfészt deklarálnunk: az Address modellben elhelyezzük a **belongs_to :addressable, :polymorphic => true** kifejezést. Az így létrehozott interfészt (**addressable**) használhatják kapcsolódásra a Order és a Customer modellek, méghozzá a **has_one :address, :as => :addressable** sorral - lásd a 12. ábrát.



12. ábra. Polimorfikus kapcsolatok

A kapcsolatok helyes használatához két plusz mezőre van szüksége a polimorf kapcsolatnak. Egy típus mezőben a kapcsolódó osztály nevét, egy azonosító mezőben pedig a rekord kulcsát tároljuk el. Mindkét mezőnév prefixumaként az interfész neve fog szerepelni, így esetünkben **addressable_type** és **addressable_id**. Erre azért van szükség mert elképzelhető, hogy a két kapcsolódó modell egy egy rekordja azonos kulccsal rendelkezik, a tí-

pus mező használatával viszont egyértelműen meghatározható a keresett rekord.

Az egyes, Address típusú modellekhez kapcsolódó objektumokat az addressable interfészen keresztül érhetjük el:

Address.find(2).addressable => #<Order id: 1 … > ,

ahol a típus mező mondja meg ismétcsak milyen osztálypéldányt kell visszaadnia, míg a másik oldalról a rendelésekhez tartozó címeket az address asszociáción keresztül: **Order.first.address => #<Address id: 2 …** > .

8.2.3. A felhasználók modell virtuális attribútumai

Virtuális attribútumoknak nevezzük az olyan attribútumokat, amelyeket nem tárolunk el az adatbázisban ezeket általában az adatok egyszerűbb megadásához használjuk.

A felhasználókat kezelő User modellnél például létrehoztam egy virtuális attribútumot (password), a jelszavak egyszerűbb megadásához. A **User#password=** metódus így az átadott jelszóra automatikusan meghívja a titkosító eljárást, a hash-elt jelszót és a hozzáadott sót pedig beállítja.

8.2.4. Kosár implementálása

A kosár megvalósításához két újabb modellre volt szükség: a **Cart** nevű reprezentálja a teljes kosarat, a **CartItem** nevű pedig a kosár egyes elemeit. Mivel a modellek adatainak tárolásához nincs feltétlenül szükség adatbázisháttérre, hiszen a munkamenetekben is eltárolhatjuk azokat, az egyszerűség kedvéért úgy döntöttem a modellek egyszerű ruby osztályok lesznek az ActiveRecord típus helyett.

A **Cart** modell kezeli a kosarat, ezen keresztül helyezzük el és töröljük a termékeket, valamint növeljük és csökkentjük a mennyiségeket. A kosárban lévő minden egyes termékhez egy **CartItem** modell tartozik: ez rögzíti a terméket - product attribútum - és a rendelési mennyiséget - quantity attribútum. A Cart modellhez tartozó összes CartItem modellt az **items** tömb tárolja.

Mivel a Cart modell nem ActiveRecord típusú a mennyiségek változtatásakor és a termékek kivételekor nekünk kell megvalósítani a keresési műveleteket, ami tulajdonképpen az items tömb kezelését jelenti.

A kereséshez a paraméterként kapott termék azonosítót használja - összehasonlítja azt minden egyes, a hozzá tartozó CartItem modellekben lévő termék azonosítójával.

A Cart modell négy metódussal rendelkezik:

- A Cart#add_product hozzáadja a terméket a kosárhoz - egy új CartItem létrehozásával, de ügyel arra is, hogy a kosárban lévő termék újbóli kosárba helyezésekor ne új rendelési tétel keletkezzék, hanem csak az előző mennyisége növekedjen egy egységgel, valamint nem létező terméket ne lehessen hozzáadni;
- A Cart#remove_product egy egységgel csökkenti a kosárban már benne lévő termék mennyiségét – egy egységnél a csökkentés a termék kivételét jelenti;
- A Cart#destroy item kiveszi az adott terméket a kosárból;
- A Cart#total_price pedig összesíti a kosár árát - a termékek árát szorozza a rendelési mennyiséggel.

8.2.5. Ellenőrzések

Az ActiveRecord bemutatásánál említettem, hogy a modellek ellenőrzését alacsony és magasabb szinten is elvégezhetjük. A példaalkalmazás elkészítése alatt többségükben elegendő volt a magasabb szintű validációkat használni, ami jelentősen könnyítette a fejlesztést.

Ezek közül kettőt emelnék ki, amelyek különösen hasznosak: egyrészt a kötelező attribútomok megadására szolgáló **validates_presence_of** , valamint a formátumok ellenőrzését elvégző **validates_format_of** metódust, ahol a reguláris kifejezések megadásával gyorsan és egyszerűen lehetett az összehasonlításokat elvégezni.

Először a kategóriák egyediségének az ellenőrzésére - ne fordulhasson elő, hogy két alkategóriát ugyanazzal a névvel hozzanak létre - saját validációs metódust írtam, de mint kiderült a **validates_uniqueness_of** metódusnak hatókört is megadhatuk. Így mindösszesen elég volt csak megadni, hogy a szülőkategória azonosítóját is vegye figyelembe a kategórianevek egyediség vizsgálatakor, amivel az alacsonyabb szintű ellenőrzés feleslegessé vált.

Új felhasználó vagy jelszó hozzáadásakor a jelszavak megerősítéséhez pedig elég volt csak a **validates_confirmation_of :password** kifejezést használni – ekkor az attribútumok között egy :password_confirmation elnevezésűt keres és ellenőriz.

A magasabb szintű validációkkal a numerikus értékek ellenőrzését is minden gond nélkül meg lehetett valósítani, például, hogy az irányítószámoknak négy jegyűeknek kell lenniük:

validates_numericality_of :zipcode, :greater_than => 999, :less_than => 10000 ,

vagy, hogy a termékekhez nem adhatnak meg az eladók negatív árat:

validates_numericality_of :price, :greater_than_or_equal_to => 1 .

Néhány esetben, amikor a magasabb szintű validációkkal nem volt megoldható az ellenőrzés az alacsonyabb szintű validációkat kellett alkalmazni. A **validate** metódus definiálásakor sincs azonban túl nehéz dolgunk, a hibák jelzése pedig egységes módon történik.

A hibák jelzéséhez ugyanis minden modell példányához tartozik egy **Errors** objektum (az ActiveRecord modul Errors osztályának egy példánya) - amit az errors metódussal érhetünk el. Az Errors objektum tulajdonképpen egy tároló, ahol az egyes attribútumokhoz külön-külön adhatunk meg hibaüzeneteket. Hibaüzenet hozzáadásához az errors.add(:attribútum, hibaüzenet), hibaüzenetek kiolvasására adott attribútum esetében az errors.on(:attribútum) metódust kell használnunk.

A hibák hozzáadása pedig egyértelműen jelzi az ActiveRecord számára, hogy a modell nem menthető az adatbázisba.

Hibaüzenetek egységesítése

Azért, hogy a hibaüzenetek is egységes formában jelenjenek meg a Rails alapértelmezett hibaüzeneteket használ, méghozzá hibatípusonként - pl. üres attribútum, hibás formátum esetén. A magasabb szintű ellenőrzéseknél, ha nem csatolunk egyedit, a Rails automatikusan a hibatípushoz tartozó alapértelmezett hibaüzenetet rendeli az attribútumokhoz.

Mivel ezek alapértelmezettként angol nyelvűek és a lokalizációs modulhoz tartoznak, használatukról részletesebben az alkalmazás lokalizációjánál ejtek majd szót.

8.2.6. Az irányítószámokat ellenőrző plugin

A címeknél előírtuk, hogy ellenőrizzük megfelel-e egymásnak az irányítószám és a település, amit külön kiterjesztésként(plugin) implementáltam.

Az irányítószámellenőrzés kiterjesztésként való megvalósításával azt szerettem volna demonstrálni, hogyan képezhetünk hordozható kódot, amit bármikor egyszerűen átültethetünk egy másik alkalmazásba, másrészt hogyan terjeszthetjük ki a Rails működését.

Az irányítószámok ellenőrzésekor a legfontosabb, hogy hiteles adatforrással rendelkezzünk. A Magyar Posta weboldaláról[30] letölhető az összes Magyarországi irányítószám és a hozzá tartozó cím XLS formátumban. Az adatok adatbázisban való tárolásához ezt előbb CSV formátumba konvertáltam, majd az adatbázisba töltéskor - a kinyert adatforrás több, számunkra szükségtelen adatot is tartalmaz - csak az irányítószám és a hozzá tartozó település került mentésre. Ez egyben azt is jelenti, hogy a budapesti irányítószámok esetében minden irányítószám mellett Budapest szerepel mint város, a kerületeket pedig nem veszszük figyelembe. Az ellenőrző modul implementálásakor így azt is figyelembe vettem, hogy egy településhez akár több irányítószám is tartozhat.

Az egyszerűbb használat érdekében a kiterjesztés az adatbázist egy Zipcode nevű ActiveRecord típusú modellen keresztül éri el - értelemszerűen két attribútummal rendelkezik: **zipcode** (irányítószám), **city** (település). A modell használatához így egy migrációval előbb létre kell hozni a szükséges **zipcodes** nevű táblát.

A irányítószámok adatbázisba töltését a **ZipcodeMatch.import_from_csv** metódus végzi el. Ahhoz, hogy ezt ne kelljen mindig manuálisan meghívni, valamint az adatokat is törölni lehessen az adatbázisból, két rake taszkot is létrehoztam. A db és zipcode névtéren belül a load (db:zipcodes:load) betölti, a delete (db:zipcodes:delete) pedig kitörli az adatokat az adatbázis táblából.

Az irányítószámok kezelését és összehasonlítását a ZipcodeMatch osztály végzi, csak osztálymetódusokat definiál, ezeket lásd a 5. táblázatban. A legfontosabb közülük a **ZipcodeMatch.match?(település,irányítószám)** , amely eldönti, hogy a megadott település és irányítószám összetartozik-e figyelembe véve, hogy egyes városok több irányítószámmal is rendelkezhetnek.

30 http://www.posta.hu/object.a4c06249-c686-4d95-b333-08b467959979.ivy

Használatát az Address modellben a 11. lista mutatja (látható, hogy alacsony szintű ellenőrzésként kell alkalmaznunk).

11. lista A ZipcodeMatch modul használata irányítószámok ellenőrzésére

```
def validate
  unless ZipcodeMatch::match?(self.city, self.zipcode)
    errors.add_to_base …
  end
end
```

5. táblázat. A ZipcodeMatch modul metódusai

ZipcodeMatch osztálymetódus	Mire használható?
match?(település,irányítószám)	Annak eldöntése, hogy a település és irányítószám egyezik-e
city_exist?(település)	Annak eldöntésére, hogy a megadott település létezik-e
zipcode_exist?(irányítószám)	Mint az előző csak irányítószám esetében
city_with_zipcode(irányítószám)	Visszaadja adott irányítószámhoz tartozó települést
zipcodes_for_city(település)	Megkeresi az adott településhez tartozó irányítószámo(ka)t

8.2.7. Modellek gyors ellenőrzése

A modellek kapcsolatát és működését nagyon egyszerűen ellenőrizhetjük, hiszen a Rails biztosít egy, a ruby interaktív shelljéhez hasonló megoldást, amit a **script/console** paranccsal érhetünk el. Ez tulajdonképpen egy irb shell megnyitását jelenti, csakhogy vele az alkalmazásunk Ruby on Rails környezete is betöltődik.

Innen kényelmesen el tudtam érni a modellek osztályait, új rekordokat hozhattam létre, törölhettem a meglévőeket, gyorsan ellenőrizhettem a modellek közötti kapcsolatokat, az adatbevitelt valamint a hibaüzeneteket. Raádásul az alkalmazás változtatásakor elég volt csak egy **reload!** parancsot kiadni, amivel a környezet újratöltődött.

Miután a szükséges modellek többségét létrehoztam és gondoskodtam arról, hogy csak a helyesnek tartott adatokat fogadják el, az alkalmazás implementálását a nézetek és az őket irányító vezérlők létrehozásával folytattam, mivel ezek szorosan összefüggnek egymással.

8.3. Vezérlők és nézetek használata

A vezérlőket a modellekhez hasonlóan legegyszerűbben a script/generate paranccsal hozhatjuk létre, paramétereként ezúttal az akciókat kell megadnunk.

```
$ script/generate controller products index show update delete
```

A parancs létrehozza a vezérlőt, a vezérlőhöz tartozó helper és teszt fájljait.

A modellek, vezérlők és nézetek együttes létrehozására a scaffold (gyorsváz) generátort is használhattuk volna, de ez inkább gyors ötletek kipróbálására használható, és mivel kész tervekkel rendelkezünk nem sok előnye lett volna, ráadásul alapvetően a REST erőforrá-sok létrehozására használható.

Gyökérútvonal beállítása

A nyitóoldal megjelenítését érdemes külön vezérlőhöz rendelni, ezért létrehoztam egy MainController elnevezésűt. Ahhoz, hogy a keretrendszer ezt alapértelmezettként hívja meg az útvonalak között, a root elnevezésűhöz kellett a vezérlő index eseményét megadni.

Központi vezérlő használata

Mivel az egyes vezérlők nem hívhatják meg egymás metódusait, a Rails definiál egy központi vezérlőt, **ApplicationController** néven. A többi vezérlőt ebből leszármaztatva az ott megadott metódusok közösekké válnak.

Az ApplicationControllert így olyan feladatok definiálására használtam, amelyeket több vezérlőben is meg kellett hívni.

Szűrők

A szűrők más metódusok meghatározott pillanatban való meghívását teszik lehetővé események futtatásakor. Ennek megfelelően három típusuk van: **before_filter** az esemény előtt, **around_filter** az esemény közben, az **after_filter** az esemény után fut le. A beállított szűrők alapértelmezettként a vezérlő minden metódusára vonatkoznak, korlátozni az **:only**, és **:except** opciókkal lehet.

A központi vezérlő metódusait a szűrőkkel kombinálva olyan közös feladatok állandó meghívására használtam fel, mint a felhasználói jogkörök ellenőrzése események előtt vagy a nézetek közös elemeinek előállítása. Például a webáruház kategória listáját minden nézetben elő kell állítani, ehhez az ApplicationController osztályban kellett egy metódust előszűrőként megadni, amely lekérdezi a kategóriákat.

A vezérlők létrehozásakor igyekeztem az MVC szemléletet követni, hogy ne bonyolítsák túl a működésüket. Feladatuk általában kimerül abban, hogy a kapott paraméterekkel létrehoznak egy modellt - a modellek osztálya így elvégzi az adatok ellenőrzését - beállítanak néhány objektumváltozót, amit a nézetek automatikusan elérhetnek, végül választ generálnak, amely egy nézet előállítását jelenti.

8.3.1. Nézetek meghívása és használata

Egy adott eseményhez tartozó nézetet a vezérlő megfeleltetéseként mindig a vezérlő/esemény.formátum.erb fájlként keres, amelytől a **render** metódussal térhetünk el. A render képes meghívni egy másik esemény nézetét is, de ekkor magát az eseményt nem futtatja, arra a **redirect_to** átirányító metódust kell használni.

A nézetekben az adatokat a vezérlő által beállított objektumváltozókból érhetjük el, lehetőség van ugyan minden modell metódusának a meghívására is, ezt azonban elvi okokból inkább kerülni igyekeztem - az MVC szemlélete szerint a vezérlőnek kell minden adatról gondoskodnia a nézetekben.

Nézetek egységesítése

Ahhoz, hogy az eltérő események nézetei közös tartalommal rendelkezzenek az oldalvázakat (layouts) használtam fel, ezek az alkalmazás **views/layouts** könyvtárában találhatóak vezérlőkénti bontásban. A webáruház két oldalvázat használ: az application.html.erb az áruház, az admin.html.erb az adminisztrációs felület egységes elemeit írja le - kihasználtam, hogyha a vezérlőhöz nem tartozik layout mindig az application elnevezésűt hívja meg.

Az eseményekhez tartozó nézetek betöltéséhez az oldalváz egy adott pontján a **yield** kulcsszót kellett használni.

A több nézet által is használt, ismétlődő kódrészleteket igyekeztem mindig egy közös helyen (fájlokban) elhelyezni, hogy csökkenjen a redundancia, és a módosítások elvégzése is egyszerűbb legyen. Ehhez az alsablonokat használtam fel, viszont sok esetben a nézetek közötti azonos tartalom szétbontásához a partial-ok használta egyszerűbbnek bizonyult.

Partial fájlok

A partial fájlok olyan rövid, paraméterezhető sablonok, amelyeket egy-egy objektum leírására használunk - például egy termék hogyan jelenjen meg listanézetekben - és amelyeket különböző vezérlőjű nézetek is felhasználhatnak, valamint AJAX funkciók megvalósítása során a válaszként küldött üzenetek előállítására is használhatjuk. Ez utóbbiakat az alkalmazás ugyan nem használ, de igyekeztem gondolni a későbbi fejlesztésekre is, ezért sok esetben előre szétbontottam a nézetek bizonyos sablonelemeit.

A partial-ok fájlneve megkülönböztetésként alulvonás jellel kell, hogy kezdődjön, meghívásukhoz a nézetekben a render metódust kell használni, a :partial opcióval megadjuk a partial fájl elérését és elnevezését, de alulvonás nélkül.

A partialoknak a változókat mindig át kell adnunk, ezt a **:locals** paraméternek megadott hash-el tehetjük meg, ahol a kulcs lesz a változó neve a sablonon belül. A partialok rendelkeznek egy hasznos **:collection** opcióval, amelynek egy tömböt átadva a partialt annak minden tagjára meghívja. Például a következő kódrészlet

```
<%= render :partial => 'products/mini', :collection => @products, :as => :product %>
```

meghívja a kategória minden termékére a rövid, képes leírását, így nem kell a nézetekben for ciklusokat használni.

Erb-től eltérő sablonrendszer használata

Sok esetben a Erb túl bonyolultnak bizonyul, sok esetben más HTML-t leíró sablonnyelvek használata is felmerül. A mi esetünkben a prototípus egyes részei Haml leírónyelven készültek, és importálni szeretnénk az ott készült Haml sablonokat. A Rails keretrendszer szerencsére ebben is rugalmasnak bizonyul és nagyon egyszerűvé teszi az alapértelmezettként használt Erb-től való eltérést.

A fájlok elnevezése ugyanis nemcsak a típust, hanem a használt leírónyelvet is magában hordozza, amit a harmadik tag jelöl - az index.html.erb esetében tehát tudjuk, hogy erb-ben íródott -, így nincs más dolgunk, minthogy minden Haml sablon esetében a haml kifejezést adjuk meg a név és típus után (pl. index.html.haml).

Az oldalak értelmezéséhez természetesen telepíteni kellett a Haml plugint az alkalmazásba.

8.3.2. Helperek

A helpereket bonyolultabb, elsősorban ciklusos HTML kódok előállítására használtam fel, hogy egyszerűsítsem a nézetek működését vagy, hogy közös elemeket adjak vissza több nézet számára - a kategórialista például ezen elv miatt került helperként implementálásra. A nézetekben ezeket mint segédmetódusokat hívhatjuk meg, és az egyes, vezérlőkhöz tartozó helpers modulban kell elhelyeznünk őket - de vezérlőtől függetlenül ezek bármelyik nézetben meghívhatóak. Az ActionView modul is hasonlóképpen definiál saját helpereket, ezekkel gyakorlatilag bármilyen HTML tag kiírása leegyszerűsíthető.

Útvonalak (linkek) elhelyezése

A linkek egyszerű elhelyezésére használható a **link_to** helper, amely a megadott :controller, :action opciókból előállítja egy esemény URL-jét. Én azonban kihasználtam, hogy az útvonalakhoz egyedi neveket rendelhetünk, és ezeket megadhatjuk a linkeknek is **útvonal_név_path** formában.

Így nemcsak egyszerűbbé válik a hivatkozások létrehozása, de változtatáskor elég csak a routes.rb fájlban megváltoztatni az útvonalakat, nem kell azt minden nézet esetében megtenni. A REST alapú erőforrásokhoz tartozó útvonalak esetében pedig a Rails által automatikusan definiált útvonal-előállító metódusokat (az erőforrás neve a _path utótaggal ellátva) kellett használni.

8.4. Űrlapok előállítása és kezelése

HTML nézetek esetén a felhasználótól kizárólag űrlapokon (formokon) keresztül kérhetünk be adatokat, a Rails ezért változatos opciókat nyújt a formok előállítására és kezelésére.

Űrlap elemek létrehozásához az ActionView helperjei között találunk egy **FormHelper** osztályt, metódusainak mindösszesen két opcióra van szükségük: egy modell objektumra és a megjelenítendő attribútumra. A helper metódus ebből előállítja a szükséges HTML form tagot, beállítva hozzá a megfelelő name attribútumot, a value attribútumba pedig automatikusan beilleszti a modell hivatkozott attribútumának értékét, ha van neki.

Az űrlapok létrehozását azonban a **form_for**-t jelentősen megkönnyítheti, mivel egyedi hatókört hoz létre a megadott objektum körül, így az átadott blokkban úgy definiálhatjuk a form tag egyes elemeit, hogy csak a modell attribútumára kell hivatkoznunk, magára a modell objektumra nem.

A **fields_for** a form_for-hoz hasonlóan egy egyedi hatókört hoz létre a megadott modell objektum körül, de azzal ellentétben a form elemet nem hozza létre vele. Ezt egymásba ágyazott modellek esetén, mint például a rendelések és a rendelési címek, egy űrlapon való megadásához használtam.

8.4.1. Egyedi FormBuilder létrehozása

Az alkalmazás prototípusában minden űrlapbeviteli mezőt egységes HTML szemantikával láttunk el a stílusok megjelenítése miatt: körülvettük egy div elemmel, és rendeltünk hozzá egy címkét - label tagot. Hibás beviteli adat esetén a befogó div elem pedig felvette a field-with-error nevű CSS osztályt.

Ezen formázások megadása minden űrlapelem esetében nem túl előnyös: hosszan tart és változások esetén minden elemet újra át kell írnunk. A redundancia csökkentésére egyedi **FormBuilder**t írtam és hozzárendeltem azt az űrlapokhoz.

Ehhez létrehoztam egy új osztályt a helpereink között és a Rails alapértelmezett builderjéből, az **ActionView::Helpers::FormBuilder** osztályból származtattam le - így nem kell teljesen egészében megírni minden FormBuilder metódust elég csak felüldefiniálni őket.

Megint csak kerülve a kódismétlést és a Ruby dinamizmusát kihasználva a define_method dinamikusan létrehozza a szükséges metódusokat, mivel azok törzse mindig azonos: „becsomagolják” a mező tartalmát a szükséges elemekbe, majd a super kulcsszó segítségével egész egyszerűen meghívják az ősosztály hasonló metódusát. A hibás adatbevitel jelzéséhez pedig előbb ellenőrzik tartozik-e a megadott ActiveRecord objektumhoz hibaüzenetet, ha igen, hozzáadják a becsomagoló div elemhez a szükséges CSS osztályt.

A hibaüzenetek egységes formai megjelenítését a nézetekben szintén ebben az osztályban oldottam meg: a **custom_error_messages** metódus előállítja és megfelelően megformázza a modellekhez tartozó hibaüzeneteket.

Mindezeken felül szükséges volt, hogy kiiktassam a Rails alapértelmezett hibakezelését, ami minden hibás mezőt és a hozzá tartozó címkét is egy egy div elemmel vesz körbe: ehhez felül kellett írni az **ActionView::Base.field_error_proc** processzt (ehhez lásd a config/initializers/field_error_proc.rb fájlt).

8.4.2. Az űrlap paramétereinek kezelése

A vezérlőkben az űrlapok által átadott paramétereket a **params** hash-el érhetjük el kulcs/érték formában. Lehetőségünk van ugyanakkor, hogy egymásba ágyazott paramétereket hozzunk létre, a [] jellel. Ha például a beviteli mező name attribútumában user[name] szerepelt, azt a {:user => {:name => ertek}} formában kapjuk vissza. Így rendkívül kényelmesen tudunk egymással összefüggő ActiveRecord modelleket egy formon belül létrehozni, és mivel a vezérlőben elég csak a paramétereket tartalmazó hash-t átadni a modell **new** metódusának, az abból automatikusan létrehozza a kapcsolódó modelleket is.

Ezt kihasználhatjuk az űrlap paramétereinek ellenőrzéséhez is, mivel elég mindig egy új objektumot létrehozni a kapott paraméterekkel a vezérlőben: mentéskor úgyis előbb lefutnak az ActiveRecord ellenőrzései. Ha nem tudjuk elmenteni, mert a modell hibás, akkor újra megjelenítjük vele az űrlap nézetét: a nézet pedig ezt az objektumot fogja felhasználni, így megjeleníti az űrlapon a hibás adatbeviteleket és a hibaüzeneteket is.

8.5. A Rails munkamenet kezelése

A munkameneteket a kosarak vásárlókhoz való rendelésénél és a felhasználók azonosításánál kellett használni.

A Rails sok más keretrendszerhez hasonlóan egy központosított változót[31] használ az adatok elhelyezésére és kivételére a munkamenetek között: a session nevű hasht.

A session változót bárhol használhatjuk az alkalmazásunkban úgy, mintha csak hash-ekkel dolgoznánk, lásd a 6. táblázatot.

6. táblázat. A session hash használata

Adat elhelyezése	session[:user_id] = user.id
Adat lekérdezése	user_id = session[:user_id]
Törlés	session[:user_id] = nil

Ennél több dolgunk nincs is, hiszen a Rails minden kéréskor automatikusan előállítja, minden válaszkor pedig eltárolja a munkamenethez tartozó adatokat.

A keretrendszerben alapértelmezettként a munkamenet-adatok tárolása és kérésekhez rendelése sütiken keresztül történik, így az alkalmazás helyett a felhasználó böngészője tárolja az adatokat, és azok a sütikben vándorolnak a kérések között. Természetesen biztonsági szempontból a munkamenetek erősen titkosítva vannak.[32]

A sütik titkosítása egy szintig elegendő biztonságot nyújt: a felhasználók nem tudják értelmezni, a munkamenetek feltörése és átírása a titkosítás miatt pedig szinte lehetetlen. Ugyanakkor nyilvánvaló hátrányokkal is rendelkezik: ha az alkalmazásban olyan változtatásokat végzünk, amelyek érintik a munkameneteket is, a felhasználóknál lévő elavult tartalmak miatt könnyen érvénytelen adatokat kaphatunk vissza, és a korábbiakat csak különböző trükkök bevetésével tudjuk megváltoztatni. Ezt figyelembe véve én inkább a munkamenetek adatbázisban történő tárolása mellett döntöttem.

31 A PHP-ban például hasonló szerepet tölt be a $_SESSION nevű szuper globális tömb.
32 SHA512 titkosítási algoritmust használ az alkalmazáshoz tartozó egyedi kulccsal.

Ehhez először létre kell hoznunk a munkamenet adatok tárolására szolgáló táblát: a **db:sessions:create** rake taszk elvégzi helyettünk a szükséges séma előállítását, így csak futtatnunk kell a migrációt. Majd megadjuk, hogy a keretrendszer az adatbázison keresztül kezelje a munkameneteket: a környezet konfigurációjában (config/enviroment.rb) a **config.action_controller.session_store** beállításnál **:active_record_store** paramétert kell megadnunk.

Ekkor a sütikben csak a munkamenet azonosítók fognak utazni és az adatbázisból rendeli hozzájuk az adatokat.

Kérdés, hogy mi történik akkor, ha a felhasználó böngészőjében tiltva vannak a sütik? A keretrendszer[33] ekkor automatikusan minden válasszal generál egy rejtett form elemet is, amely visszaküldi adatait a kérésekkel a szervernek.

Az ActionController::InvalidAuthenticityToken kivétel kezelése

Közvetlenül az adatbázisos munkamenet tárolásra való átállás után botlottam a fenti kivételbe. Az elsőre érthetetlennek tűnő probléma gyorsan orvosolható: az **app/controllers/application.rb** fájlban ki kell venni a komment jelet a **protect_from_forgery** metódus után álló **secret** szimbólum elől.

A Rails ugyanis az ActiveRecordos munkamenet kezelésre való átállás után minden űrlapban elhelyez egy titkosított kódszót(Authenticity Token), majd a kérések fogadásakor ellenőrzi azokat, ezzel megelőzve, hogy - a GET metódusos kérések kivételével - kéréseket lehessen küldeni az alkalmazáson kívülről is - így védekezik a CSRF (Cross-Site Request Forgery)[34] támadások ellen.

8.5.1. Flash - a Rails egy speciális munkamenet eljárása

Szorosan a Rails munkameneteihez tartozik a flash hash használata. Bizonyos esetekben ugyanis szükségünk van arra, hogy üzeneteket tudjunk átadni két átirányított esemény között, például hogy hibaüzeneteket vagy információkat közöljünk a felhasználóval.

33 Valójában ezt a Ruby CGI moduljának Session osztálya végzi, mivel a Rails munkamenet kezelése erre épül.
34 Bővebb információhoz juthatunk az API ActionController::RequestForgeryProtection::ClassMethods részében magáról a problémáról és a támadás leírásáról.

Azonban mivel minden átirányítás során a Rails új vezérlő objektumokat hoz létre és így az előzőleg létrehozott példányváltozók elvesznek, ezeket kénytelenek vagyunk a munkamenetben elhelyezni, majd minden lehíváskor törölni az eredeti üzenetet. A Rails ezt a folyamatot teszi szükségtelenné a flash hash használatával.

A **flash** változót ugyanúgy használhatjuk, mint a **session** változót, azzal a különbséggel, hogy a benne tárolt adat mindösszesen egy válaszig őrzi meg tartalmát, utána automatikusan törlődik.

A flash rendelkezik továbbá két segéd metódussal: **flash#new[]** és **flash#keep()**. Az előbbi a flash tartalmát csak az éppen futó eseményig tárolja el, nem adja át azt egy következőnek a munkamenetben. Míg utóbbi megőrzi a már létező flash tartalmat a következő átirányításig is.

8.6. Felhasználók kezelése

A felhasználókhoz kapcsolódó minden folyamatot egy központi vezérlő, a UserController végez el. Itt történik az új vásárlók regisztrálása (a registration esemény a paraméterként kapott adatokkal létrehoz egy új felhasználót, csoportként pedig beállítja a vásárlókét) valamint a regisztrált felhasználók beléptetése - mind a vásárlók, mind az eladók esetében a paraméterként érkezett adatokkal meghívja a User modell autentikációs metódusát, majd beállítja a munkamenetben a belépett felhasználó azonosítóját (**session[:user_id]**).

A logout esemény ezzel ellentétben kilépteti a felhasználót azzal, hogy kitörli a munkamenetet.

A belépett felhasználók jogosultságának ellenőrizéséhez az ApplicationControllerben két metódust definiáltam: az **authorize_as_customer** metódus a vásárlók, az authorize_as_admin az eladók (adminisztrátorok) jogosultságát ellenőrzi. Mindkettő lényege, hogy a munkamenetben beállított felhasználói azonosítót ellenőrzi először, hiszen ez azt jelenti, hogy sikeres volt a belépés, majd második lépésként ellenőrzi a felhasználóhoz tartozó csoportot. Mindkét ellenőrzés kudarca esetén a belépéshez irányít át.

Az ellenőrző metódusokat előszűrőként (before_filter) kell meghívni a szükséges metódusok előtt.

8.7. Adminisztrációs felület leválasztása

Az adminisztrációs felület leválasztásához az alkalmazás többi részétől létrehoztam egy különálló mappát a vezérlők között (app/controllers/admin). Ahhoz, hogy a Rails is tudja ezeket használni a benne lévő osztályokat modulokként kell definiálni, így rendre Admin::ProductsController, Admin::OrdersController stb. lesz az elnevezésük. A közös metódusok elhelyezéséhez itt is létrehoztam egy központi vezérlőt (AdminController - app/controllers/admin), a többi adminisztrációs felületet kezelő vezérlőt ebből származtattam le - ez mindig ellenőrzi a jogosultságokat azzal, hogy előszűrőként meghívja az authorize_as_admin metódust.

Az adminisztrációs felületen a modellek kezelését REST alapú erőforrásokkal oldottam meg, mivel ez egyszerűbb megvalósítást tett lehetővé. Ehhez azonban előbb az útvonalaknál egy névteret kellett létrehozni az adminisztrációs erőforrásokhoz is - a vezérlők modulokba csoportosítása miatt.

8.8. Az alkalmazás lokalizációja magyar nyelvre

Az alkalmazások lokalizációja sokáig a Rails leggyengébb láncszemének számított. A pluginekkel (kiegészítőkkel) való küszködést oldotta fel a Rails 2.2-es változatában debütáló I18n[35] modul.

Az I18n modul egységes interfészt (API) nyújt a lokalizációk készítéséhez, miközben igyekszik elkerülni a felesleges konfigurálást.

Alapértelmezettként angol lokalizációs beállításokat használ, saját - például magyar nyelvű - definiálásához a következőkre van szükségünk:

- Jelezni az I18n számára a lokalizációs fájljaink helyét és használatát

  Az **I18n.load_path** metódussal hozzáadhatjuk, a **I18n.default_locale** metódussal pedig aktivizálhatjuk az adott nyelv lokalizációs fájljainak használatát - ezeket a beállításokat a **lib/initializers/locales.rb** fájlban adtam meg.

- Elhelyezni a lokalizációs fájlokban az alapértelmezett üzeneteket

35 Az I18n az internacionalizáció szó rövidítése

Ehhez én a http://rails-i18n.org/wiki/pages/translations-available-in-rails címen elérhető oldal tartalmát használtam fel: itt találjuk a dátum- és időformátumokat, valamint az ActiveRecord alapértelmezett hibaüzeneteit YAML formátumban. (rubyban írt formátumot is használhatunk, ahol lehetőségünk van a futásidőben történő döntésekre - tipikusan ilyen lehet egyes nyelvekben az egyes- és többesszám jelölése mennyiségek után.)

Ez alapján a lokalizációs fájlok helyeként az **app/locales/** könyvtárt állítottam be. A Rails alapértelmezett, magyar nyelvre lefordított üzeneteit a **hu-HU.yml**, az alkalmazás egyedi üzeneteit a **messages.yml**, a modellek attribútumainak fordítását a **models.yml** fájl tartalmazza.

Az üzenetek megkeresésére és elhelyezésére az **I18n.translate** metódust használhatjuk, azonban a legtöbb lokalizációs beállítást az I18n modul automatikusan megkeres: ilyenek az ActiveRecordos hibaüzenetek, a modellek és attribútumaik lokalizált változatai, valamint a dátum és időformátumok.

ActiveRecord hibaüzenetek

A modellek validációjánál már említettem, hogy a különböző ellenőrzésekhez a Rails egységes hibaüzeneteket használ. Az I18n modul követi ezt, de kiterjeszti is azt azzal, hogy mind a modellek mind a hozzájuk tartozó attribútumok esetében adhatunk meg egyedi üzeneteket. Ezek eléréséhez konvenciókat alkalmaz meghatározott sorrendben.

Példaként nézzük meg, hogy a **User** modell **username** attribútumának formai ellenőrzése esetén ez hogyan néz ki.

Az attribútum formai hibája esetén az alapértelmezett hibaüzenetnek az **invalid** elnevezésű fog megfelelni. Az I18n tehát ezt fogja keresni, elsőként a modell attribútumához tartozó invalid üzenetként, az **activerecord.errors.models.user.attributes.username.invalid** útvonalon - az alkalmazásunk esetében ez megtalálható, így ezt fogja alkalmazni. Ellenkező esetben a modell invalid üzenetére ugrik az **activerecord.errors.models.user.invalid** úttal. Ha ez sem található meg az alapértelmezett ActiveRecord hibaüzenetet alkalmazza, amelyet az **activerecord.errors.messages.invalid** úton ér el.

Modell attribútumok

A modell attribútumok magyar nyelvre fordítására az űrlapoknál van szükség: enélkül a hibaüzenetekben a mezőnevek angol megfelelői jelennének meg. Az I18n modul itt is követi a fenti automatizmust, és meghatározott útvonalakon keresi a lokalizált neveket: a modelleket az activerecord.models.modelnév, az attribútumokat az activerecord.attributes.modelnév.attributumnév útvonalon.

Dátum és időformátumok

A dátumokat a Rails az adatbázisban mindig UTC[36] formátumban tárolja, ami időzóna független, egységes dátumokat jelent. Ahhoz, hogy ezek a magyarországi időzónának megfelelően jelenjenek meg a config.time_zone beállításnál a budapesti időzónát (+1 óra) kellett megadni.

A dátumok és időpontok formázásához egyszerűen azok **to_s** metódusát kell meghívni, paraméterként átadva a formátumstringet - az strftime függvénynek megfelelő formázási stringek használhatóak. Egyszerűbb azonban, ha a formátumstringeket a lokalizációs fájljainkban adjuk meg, mert ekkor elég csak szimbólumként megadni a formátum nevét a to_s metódusnak - és így a változtatásokat megint csak egy helyen kell elvégezni.

36 Coordinated Universal Time - Egyezményes koordinált világidő

9. Tesztelés[11]

Minden alkalmazás esetében elengedhetetlen a megfelelő működés biztosítása, amelyet megfelelően megválasztott tesztekkel valósíthatunk meg. A legegyszerűbb megoldás, ha működés közben előre megtervezett szcenáriókat próbálgatunk végig, ami belátható, hogy nem túl előnyös: nehezen reprodukálható és meglehetősen kétes végeredményt ad.

Ennél hatékonyabb, ha előre megírt teszteket futtatunk a rendszeren, ahol a rendszerbe bevitt adatokra előre megadjuk, hogy szerintünk milyen állapotokat kellene felvennie a teszt egyes pillanatában - mi lenne a helyes működésük - és ha ezt nem teljesítik akkor a teszt megbukott. A tesztek effajta definiálása azért is előnyös, mert előremutatóak, ha változtatunk az alkalmazásunkon újra futtathatjuk a őket, és kiértékelhetjük a kapott eredményt.

Szerencsére a Rails keretrendszer is támogatja ezeket a megoldásokat: az egységes tesztkörnyezetet beépítettként kapjuk, így fejlesztőként csak a tesztek megírásával és tesztadatok biztosításával kell törődnünk minden mást biztosít a keretrendszer számunkra.

A Rails alapvetően négy tesztelési fogalmat és típust különböztet meg, attól függően mire használjuk:

- egység tesztekkel (Unit Test) a modellek tesztelését;
- működési tesztekkel (Functional Test) a vezérlőket;
- integrációs tesztekkel(Integration Test) az alkalmazás egészét érintő felhasználói folyamatokat;
- míg a teljesítménytesztekkel különböző futási méréseket végezhetünk.

A tesztek mindegyike az egységteszt elvére épül: azaz a programkódunktól teljesen különállóan hozunk létre és futtatunk tesztelési eseteket és azokban megvizsgáljuk, hogy a programkód bizonyos részeinek futtatásakor milyen eredményeket kapunk. Az eredmények kiértékelésére állításokat fogalmazunk meg, ezeket az **assert** metódusokkal tehetjük meg. Mivel a Rails keretrendszer a Ruby alapértelmezett egységteszt könyvtárát a **Test::Unit**-ot használja, az elérhető assert-tek többsége megegyezik az ott definiáltakkal, ugyanakkor a Rails is definiál néhány egyedi assert-tet, elsősorban a vezérlők teszteléséhez: ilyenek az átirányításokat és HTTP válaszokat ellenőrző állítások.

Az alkalmazás létrehozásakor már említettem, hogy a keretrendszer tartalmaz egy különálló (test) környezetet a teszteléshez, amely külön beállításokat és adatbázist használ, így még jobban elválaszthatjuk a tesztelési folyamatokat a fejlesztéstől és az éles működési környezettől. A tesztek futtatásakor ezt az adatbázist használjuk, így a tesztadatok felvitele is ide történik.

9.1. Tesztadatok bevitele

Tesztek írásának nem sok értelme van, ha nincsenek adataink amelyek ellen futtathatjuk őket. A tesztadatok bevitelére épp ezért a Rails nyújt egy kényelmes megoldást: a fixtures-eket, magyarul alapadatokat.

Az alapadatokkal tulajdonképpen minden modellhez rekordokat adhatunk meg, amelyeket azután a Rails betölt a teszt adatbázisba. Ezeket a **test/fixtures** könyvtárban találjuk meg - minden modellhez egy-egy ilyen fájl tartozik. Az egyszerűség miatt ezek YAML formátumban tartalmazzák a rekordokat a következő formában: minden rekordhoz egy név tartozik, alatta pedig **kulcs: érték** párokként az értékeik. A rekordok elnevezése a tesztadatok rendszerezése és a modellek kapcsolata miatt fontos.

A neveket kereszthivatkozásként használhatjuk a különböző kapcsolatokhoz, ráadásul a tesztek írásakor név szerint, közvetlenül el is érthetjük őket, így nem kell az azonosítók megjegyzésével bajlódni.

Ha például létrehozok egy könyvek elnevezésű rekordot a kategóriák között, erre, a hozzá tartozó alkategóriákban - a kapcsolat létrehozásához - ugyanezzel a névvel hivatkozhatok (a 12. lista ezt demonstrálja a kategóriákhoz tartozó fixtures fájl egy részletével). Az ilyesfajta hivatkozásokkor egy szabályra kell ügyelni: ne adjunk meg azonosítókat (id mező) a rekordokhoz, a Rails ugyanis automatikusan generálja és keresi meg azokat.

12. lista A test/fixtures/categories.yml fájl egy részlete

```
books:
  name: Könyvek
  parent: 0

classic_books:
  name: Klasszikus irodalom
  parent: books

computer_books:
  name: Számítástechnikai könyvek
  parent: books
```

A nevekkel a rekordok elérése is lehetségessé válik: minden fixtures esetében a keretrendszer metódust hoz létre a modell nevével, amelynek csak meg kell adnunk a rekord nevét szimbólumként. A fenti pontnál maradva a Klasszikus irodalom kategóriát például a **categories(:classic_books)** kóddal érhetjük el a tesztjeinkben.

A fixtures-ekbe ruby kódot is beágyazhatunk, így akár több ezer rekordot is létrehozhatunk egy ciklussal.

A keretrendszer a tesztek futtatása előtt az alapadatokat úgy tölti be a teszt környezet adatbázisába, hogy előtte mindig törli a korábbiakat, így mindig konzisztens adatokat kapunk.

9.2. Modellek tesztelése egység tesztekkel

A modellek legfontosabb feladata az adatok tárolása, ahol kulcsfontosságú szerep jut a hozzájuk kapcsolt vagy bevitt adatok ellenőrzésének. Ezért a modellek tesztelésekor olyan teszteket igyekeztem írni, amelyek elvégzik ezen ellenőrzések felülvizsgálatát.

Minden modellre kiterjedő, általános szempontok a következők voltak:

- Kötelezően megadandó attribútumok hiánya esetén a modellünk valóban hibásnak minősül-e.
- A bevitt adatok formátum-ellenőrzése megfelelő-e: helyesnek gondolt, és rossznak tartott adatokkal is megvizsgáltam, hogy a modellek validitása hogyan alakul.
- A modellekhez társított metódusok megfelelő eredményeket adnak-e vissza.

A következőekben álljon itt pár példa a kiemelt fontosságú modellek tesztjeiből.

Termékek modellje

A termékek esetében fontos, hogy az ár nem lehet negatív vagy nulla, egy egységnek(forint) vagy annál nagyobbnak kell lennie.

A 13. lista pontban látható ennek ellenőrzése: három lehetséges tesztadatot(negatív, nulla, pozitív érték) adunk meg a modellnek, majd ezekkel ellenőrizzük az elvárt kimeneteket: előre megmondjuk a terméknek melyik esetben kell érvényesnek lennie vagy épp ellenkezőleg. Utóbbi esetében a kapcsolódó hibaüzenetet is összehasonlítjuk az elvárttal.

13. lista A termékek ár ellenőrzésének tesztje (test/unit/address_test.rb)

```
test "price is greater than or equal to 1 forint" do
  ...

  # price >= 1
  product.price = 1
  assert product.valid?, product.errors.full_messages

  # price = 0
  product.price = 0
  assert !product.valid?
  assert_equal I18n.t(..., :count=>1), product.errors.on(:price)

  # price < -1
  product.price = -1
  assert !product.valid?
  assert_equal I18n.t(..., :count=>1), product.errors.on(:price)

  ...
end
```

A modellek validációjánál már ejtettem szót az errors metódusról. A tesztelés során ezzel ellenőrizhetjük, hogy hibás érték esetén a várt hibaüzenetet kaptuk-e vissza, ezt az **errors.on(:attributum)** kifejezéssel, míg, hogy egy attributum értéke pusztán elfogadható-e vagy sem az **errors.invalid?(:attributum)** kifejezéssel tehetjük meg.

Kategóriák

A kategóriáknál előírás, hogy nem lehet két egyforma elnevezésű azonos alkategóriában, azaz azonos szinteken minden rekordnak egyedinek kell lennie. Ennek teszteléséhez előbb az alapadatokban elhelyeztem egy kategóriát és létrehoztam hozzá egy alkategóriát, amelyek a teszt futtatásakor az adatbázisba kerülnek, majd az alkategóriát kivéve a nevével létrehozok egy új modellt, aminek így nyilvánvalóan hibásnak kell lennie. Ezt a logikát az egyediségvizsgálatok mindegyikébe át lehet ültetni.

Címek (Address) modell tesztje

A vásárlókhoz és rendelésekhez tartozó címek tesztelésekor jelent meg leginkább a minél széleskörűbb tesztadatok biztosítása. Itt több jó illetve rossz adatot is definiáltam mind a telefonszámok és e-mail címek mind az utcacímek esetében, majd egyenként ellenőriztem, hogy teljesítik-e a validációt vagy sem.

Itt futottam bele egy érdekes működési hibába: az irányítószám hosszának ellenőrzése ugyanis rendre megbukott! A számomra helyesnek gondolt **validates_ length_of** ugyanis stringek hosszának ellenőrzésére használatos, ezt kicserélve a **validates_numericality_of** helyes beállításával a tesztek sikeresen lefutottak.

További problémát jelentett, hogy a tesztek futtatása előtt a Rails kitöröl minden táblát és az alapadatokkal tölti fel a teszt adatbázist, így az alapadatokon kívül nem lehet saját, előre definiált adatokkal feltölteni. Viszont így az Address modell tesztjei rendre megbuk-tak, mert a ZipcodeMatch modul nem találta az adatbázisban a szükséges adatokat.

Ennek feloldására először az irányítószámok fixtures-ként való megadásával próbálkoz-tam - elfogadja a CSV formátumot is - ez azonban jelentősen lelassította a tesztek futtatá-sát, mivel minden teszt előtt a tesztkörnyezet újratöltötte az alapadatokat. Ezért inkább lét-rehoztam egy, a ZipcodeMatch modult imitáló úgynevezett mock osztályt (test/mocks/zip-code_match.rb), ami tulajdonképpen csak felülírja a modul eredeti metódusait, hogy min-dig igaz értékkel térjenek vissza. Ezzel a teszteket függetleníthetjük a ZipcodeMatch mo-dultól, amelynek működési tesztelését amúgy is az alkalmazástól különállóan kell megten-ni.

Kosár

A kosár tesztjeivel elsősorban azt ellenőriztem, hogy a termékek hozzáadódásakor he-lyes végösszegek, mennyiségek jelennek-e meg. Továbbá, ami még fontosabb, hogy a ko-sárban lévő termék újbóli kosárba helyezésekor ne új rendelési tétel keletkezzék, hanem az előző mennyisége növelkedjen egy egységgel. A kosárból bármelyik termék kivehető, azaz törölhető legyen, de ugyanígy ha a mennyiséget egy egységnél csökkentik, akkor a termék kikerüljön a kosárból. Ha pedig a kosárban egy termék szerepel egy rendelési egységgel annak csökkentésekor a kosár megsemmisüljön, törlődjön a munkamenet.

9.3. Vezérlők tesztelése

A vezérlők tesztjeit a következő szempontok szerint írtam meg:

- A kérés sikeresen lefutott-e
- A megfelelő helyre irányítódott át
- Hibás vagy hiányzó paraméterek esetén a megfelelő helyre (esetünkben a főoldalra) irányítódott át a kérés
- A megfelelő objektumok létrejöttek, a munkamenetek pedig a megfelelő adatokat tartalmazzák.

A vezérlőkkel együtt a nézeteket is tesztelhetjük, én azonban ezt feleslegesnek tartottam, mivel kevésbé komplex, egyszerű nézeteket állítunk elő.

A HTTP kérések szimulálásához a **get**, **post**, **put**, **delete metódusok** használhatóak. Négy argumentumot adhatunk meg nekik: a meghívott eseményt, a kérés paramétereit, a munkamenet változókat, valamint a flash változók állapotát hash formájában. Ezzel gyakorlatilag bármilyen kérést szimulálni tudunk.

A kérés elküldése után négy hash áll rendelkezésünkre az ellenőrzésre. Az objektumváltozókat az **assign metódus**, a beállított sütiket a **cookies**, a munkamenet és flash változókat a **session** és **flash hash**eken keresztül érhetjük el. Az átirányítások ellenőrzéséhez az **assert_redirected_to** állítást használhatjuk.

A vezérlők közül a három legfontosabbat teszteltem elsősorban: a Cart elnevezésűt, hogy megfelelően kezelje a kosarakat a munkamenetek között; a User vezérlőt, hogy a felhasználók regisztrálása és belépése megfelelően működik-e – itt is előlép a munkamenetek ellenőrzése, valamint az Order vezérlőt, a rendelések feladása miatt.

9.4. Tesztek futtatása

A tesztek futtatásához a **test** nevű rake taszkot használhatjuk, adott típus, például a modellek tesztjekor a **test:units** taszkot használjuk, egy-egy teszt különálló futtatásához elég a tesztelési fájlt futtatni.

A tesztek lefutása után az eredmények kiíródnak a standard kimenetre, lásd 14. lista.

14. lista Egy teszt eredménye futtatás után.

```
Loaded suite /usr/local/lib/ruby/gems/1.8/gems/rake-
0.8.3/lib/rake/rake_test_loader
Started
FF.....FF.................E.E.EE.F...........
Finished in 0.46407 seconds.
1) Failure:
test_accept_various_telephone_numbers(AddressTest)

...

Failed with: 06/30/622/3456.
<false> is not true.
```

Egy-egy teszteset végeredményéről tulajdonképpen háromféleképpen szerezhetünk tudomást. Ha sikeres volt mindössze annyit, hogy teljesült, ha megbukott (Failed állapot) akkor értesítést kapunk: a teszt metódus nevéről, visszakapjuk a megbukott assert üzenetét, valamint a metódushívási láncot. A harmadik eset akkor fordulhat elő, ha programozási hibát ejtünk (érvénytelen metódust hívunk, vagy szintaktikai hibát vétünk), ekkor Error státuszt kap a teszteset, és a megbukotthoz hasonlóan kiíródik a hiba pontos oka.

9.5. Összefoglalva

Azt hiszem ez a fejezet nyilvánvalóan megmutatta, hogy a tesztek írása szerves részét képezi egy Rails alkalmazás fejlesztésének.

A tesztek írását egyszerre éreztem könnyűnek és hasznosnak, hiszen már a tesztek kigondolásával is sokat javítottam az alkalmazás működésén. Változtatások esetén elég volt a megírt teszteket újra futtatni, ez például a címek ellenőrzésénél volt különösen hasznos.

Sőt sok esetben előre meghatároztam a tesztekkel, hogy milyen végeredményt szeretnék elérni, majd ezután láttam neki a megvalósításnak.

10. Összefoglalás

E szakdolgozat azt hiszem megmutatta, hogy lehet könnyen és élvezetesen is webes alkalmazásokat fejleszteni, a Ruby on Rails környezetnek köszönhetően. Megállapítható, hogy a keretrendszer minden eleme, minden egysége jól megtervezett munka eredménye, az első megjelenés óta eltelt öt év alatt kiforrott rendszer lett.

A szakdolgozat készítése során a Ruby nyelv elsajátítása új szemléletmódokat és megközelítéseket tanított meg a számomra. A rövid és tömör nyelvnek köszönhetően - véleményen szerint a Rails sikere is nagyban ennek tulajdonítható - sokszor éreztem úgy nincs is szükség a programkód dokumentálására.

A Ruby on Rails keretrendszer használatakor jó volt látni, milyen egyszerűen is meg lehet oldalni azokat a gyakori problémákat, amelyekkel a korábbi webfejlesztési tapasztalataim során már találkoztam, de általában sok energiát emésztettek fel ahelyett, hogy az alkalmazás bővítésével, javításával foglalkozhattam volna.

Az adatbázisok használata az ActiveRecordnak köszönhetően szinte gyerekjáték volt, egy sor SQL lekérdezést sem kellet írnom az implementálás során! Bár a megfeleltetések használatát először frusztrálónak éreztem, később azonban rájöttem, hogy ez olyan fejlesztési praktikákat ad, amelyeket más alkalmazások írásakor is igyekszek majd követni.

Az MVC mintának köszönhetően a felhasználói-felületek és a működés szétválasztását sikerült teljes mértékben megoldani, így egy igazán flexibilis és jól bővíthető alkalmazást kaptam, amely úgy gondolom kis továbbfejlesztéssel nyugodtan alkalmazható lenne éles, működő környezetben is.

Ezt elsősorban a megírt tesztekre alapozom, amelyekkel biztos lehetek abban, hogy az alkalmazás valóban helyesen működik. Bevallom a tesztek írása előtt kicsit szkeptikus voltam, vajon megéri-e a befektetett energiát és időt, de a keretrendszer tesztkörnyezete megint csak mindent biztosított a számomra.

Természetesen nem mehetek el szó nélkül az érzett hiányosságok mellett sem, ami elsősorban a keretrendszer dokumentációját érintette. Nem mintha a Rails API-ja nem nyújtana elegendő információt az elérhető osztályokról és metódusokról, de a megfelelő használatához úgy érzem nem árt, ha már van kellő fejlesztési tapasztalatunk akár a Rails keretrendszerben akár egy Ruby alapú program fejlesztésében.

Ezért egy jó könyv kézbevétele és elolvasása szinte elengedhetetlen az elinduláshoz és az alapvető tippek-trükkök elsajátításához.

További probléma, hogy a keretrendszer újításairól sokszor csak a forráskód követésével, vagy a többi fejlesztő által írt blogok olvasásával értesülhetünk - amelyek legtöbbször angol nyelvűek, így nyelvtudás is szükséges a megértésükhöz. Ezeket a problémákat azonban úgy tűnik a Rails közössége is felismerte és igyekeznek orvosolni a közeli jövőben.

Végezetül, ha egy mondatban kellene megfogalmaznom a gondolataimat: a Ruby on Rails nem csupán egy keretrendszer használatát, hanem egy szemléletmód követését is jelenti.

Irodalomjegyzék

[1] Bártházi András: Mi az a Web 2.0? http://wish.hu/cikkek/web20.html, 2003

[2] Nagy Gusztáv: Web programozás jegyzet, 2008., 171. oldal

[3] Ruby on Rails közösség: Rails Framework Documentation http://api.rubyonrails.org/, 2008

[4] Wikipédia szerkesztők: Magyar Ruby szócikk a Wikipédián http://hu.wikipedia.org/wiki/Ruby, 2008

[5] Fábián Gergely: E-learning tananyag a Ruby on Rails keretrendszerhez http://rails.ruby-oktatas.hu/chapters/ruby/basics, 2008

[6] Yukihiro Matsumoto: Elhangzott a Ruby-Talk levelező listán http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/2773, 2000

[7] Raul Parolari: Code Block, lambda, Proc in Ruby http://raulparolari.com/Ruby2/lambda_Proc, 2008

[8] Dave Thomas - David Heinemeier Hansson: Agile Web Development with Rails, Second Edition, 2006., 22-30. oldal

[9] Ruby on Rails wiki szerkesztők: ActiveRecord in Ruby on Rails http://wiki.rubyonrails.org/rails/pages/ActiveRecord, 2008

[10] Wikipédia szerkesztők: A Wikipédia Representational State Transfer szócikke http://en.wikipedia.org/wiki/Representational_State_Transfer, 2008

[11] Akshay Surve - Pratik Naik - Mike Gunderloy: A Guide to Testing Rails Applications http://guides.rubyonrails.com/testing_rails_applications.html, 2008

A CD Melléklet tartalma

- Az alkalmazás forráskódja (webshop könyvtár)
- Az alkalmazás futtatásához szükséges Ruby on Rails keretrendszer 2.2-es változata
- Az alkalmazás futtatásához szükséges kiegészítők
- Az alkalmazás prototípusának forráskódja (webshop_proto könyvtár)

A telepítési információkat lásd lentebb.

Az alkalmazás elérhető és működő verziója

Az alkalmazás működő változata elérhető a http://webshop.csiszarattila.com URL-en keresztül.

Az adminisztrációs felületre a http://webshop.csiszarattila.com/admin/ URL-en keresztül egy admin nevű, admin jelszójú felhasználóval léphetünk be.

Telepítési utasítások

A CD mellékleten található alkalmazás kipróbálásához szükséges:

- a Ruby fordító (legalább 1.8.6-os verzió)
  letölthető a http://www.ruby-lang.org/en/downloads/ címről.
- a Rubygems csomagkezelő (legalább 1.3.1-es verzió)
  telepítéséhez lásd a http://www.rubygems.org/read/chapter/3 címet.
- a Rake gem
  telepítéséhez lásd a http://rake.rubyforge.org/ címet.

Az adatbázis használatához a Ruby megfelelő moduljának telepítése:

- MySQL esetén lásd a http://www.tmtm.org/en/mysql/ruby/ címet. (Megjegyzés: tapasztalataim szerint csak a MySQL 5.0-ás verziójáig működik megbízhatóan.)

Beállítások

Elsőként az adatbázis hozzáférési adatait adjuk meg a config/database.yml fájlban. Az alkalmazás futtatásához elég a development kulcsszó alatti adatok kitöltése.

Adatbázisok létrehozása és feltöltése

(A következő parancsokat az alkalmazás gyökérkönyvtárában kell kiadni.)

Az adatbázis létrehozásához futtassuk előbb a rake db:create, majd a sémák betöltéséhez a rake db:schema:load parancsot. Ezután töltsük fel az alapadatokat a rake db:fixtures:load, majd az irányítószámokat a rake db:zipcodes:load parancssal - ez utóbbi eltarthat egy ideig.

Alkalmazás futtatása

Végül indítsuk el a Rails beépített webszerverét a ruby script/server paranccsal, a futó alkalmazást ekkor a http://localhost:3000-es címen érhetjük el egy webböngésző segítségével. Az adminisztrációs felületet pedig a http://localhost:3000/admin/ címen, egy admin nevű admin jelszójú felhasználóval.