DIPLOMAMUNKA

Himer Sándor

Debrecen

2009

Debreceni Egyetem

Informatika Kar

Webes alkalmazásfejlesztés a Spring Framework és az Adobe Flex szemszögéből

Témavezető:
Dr. Kuki Attila
egyetemi adjunktus

Készítette:
Himer Sándor
programtervező matematikus

Debrecen

2009

Tartalomjegyzék

1 Bevezetés

A diplomamunkám célja egy olyan webes alkalmazás architektúra bemutatása, ami megfelel a mai modern kor által támasztott követelményeknek. Nemcsak bemutatásra kerül az architektúra, hanem ábrákkal is illusztrálva lesz az egyes elemek működése és kapcsolata. Az építőköveinek használatára, és összekötésére is láthatunk kisebb példákat.

Ezen architektúra kialakításánál két szempontot vettem figyelembe. Az egyik a nyílt forráskódú komponensek használata, a másik a gyors alkalmazásfejlesztési lehetőség. Utóbbi szempont nem csak az architektúrától függ, hanem hogy hogyan is fejlesztjük az alkalmazásunkat. Ezért olyan építőköveket használtam, ami könnyen tudunk valamilyen tervezési mintát alkalmazni, vagy maga is egy tervezési mintára épül.

A nyílt forráskódúság a következő előnyöket rejti:

- Költségtakarékos, hiszen nem kell licence díjat fizetni
- Stabilitás, megbízhatóság. Ez abból adódik, hogy a fejlesztésben és a tesztelésben nem csak egy kis csoport vesz részt. A tesztelési lehetőség bárki számára elérhető, így az esetleges hibák gyorsabban kiderülnek.
- Fejlődés. Bárki adhat ötletet egy új funkciók fejlesztésére, vagy egy meglévő optimalizálására, továbbfejlesztésére. Tehát a komponens folyamatosan fejlődik, ami nekünk csak jót jelent.
- Nem függ egyetlen gyártótól sem. Ez nagyban megkönnyíti a hordozhatóságot, és ha valami különleges igényünk van, azt jó eséllyel megkaphatjuk egy következő verzióval.
- Nincsenek korlátok a felhasználhatóság szempontjából.
- Támogatottság, hiszen minden nyílt forráskódú programhoz tartozik minimum egy fórum, ahol lehet kérdezni és a fejlesztők válaszolnak.

1.1 Gazdag internet alkalmazás

A Rich Internet Application (RIA), vagyis a gazdag internet alkalmazás kifejezés és mozaikszó egyre jobban kezd elterjedni. A RIA egy webes alkalmazás, amely rendelkezik az asztali alkalmazások legtöbb jellemzőivel, és mégis egy böngészőn keresztül jut el hozzánk. Vagy valamilyen plugint vagy virtuális gépet használnak működésükhöz. Ezt a kifejezést az akkori

Macromedia (most már az Adobe része) vezette be 2002-ben. A RIA a következő előnyöket biztosítja számunkra:

- A szerver teljesítményének növelése: A RIA általában a „nincs oldalfrissítés” modellre épül. Ez annyit tesz, hogy az alkalmazás betöltődik a kliens memóriájába és addig nem is kell az oldalt frissíteni, amíg a felhasználó az alkalmazást használja. Ez több kliens oldali feldolgozást tesz lehetővé, mivel az alkalmazásunk megőrzi az állapotát.
- A felhasználói élmény fokozása: Nagyon sok olyan dolgot meg tudunk a RIA alkalmazásban csinálni, amit egy egyszerű HTML oldal nem tud. Ilyen például az asztali alkalmazásokból jól ismert „fogd és vidd” módszer.
- Fejlesztői hatékonyság növelése: A RIA fejlesztés az újrahasználható komponensek módszertanára épül, ezáltal a már egyszer jól megírt komponenseinket újra fel tudjuk használni (pl.: egy bejelentkező oldalt). A RIA által nagyon sok tervezési mintát fel tudunk használni az alkalmazásunkban, ami ugyancsak növeli a fejlesztés gyorsaságát.

Ha egy létező alkalmazásunkat RIA alkalmazássá szeretnénk alakítani, akkor el kell rugaszkodnunk a tradicionális kliens-szerver architektúrától. El kell jutnunk a vékony klienstől a vastag kliensig. A vékony kliens a kliens-szerver architektúrából a kliensre utal, és azért vékony, mert mindent a szerver csinált, a kliens csak a HTML oldalakat jelenített meg. Ráadásul a szerver kezelte a kliens állapotát is. A vastag kliens ezzel szemben sokkal többet számol, a kliens állapotát is a kliens oldalon tárolja, és általában csak akkor fordul a szerverhez, ha adathoz szeretne hozzáférni, vagy adatot szeretne tárolni.

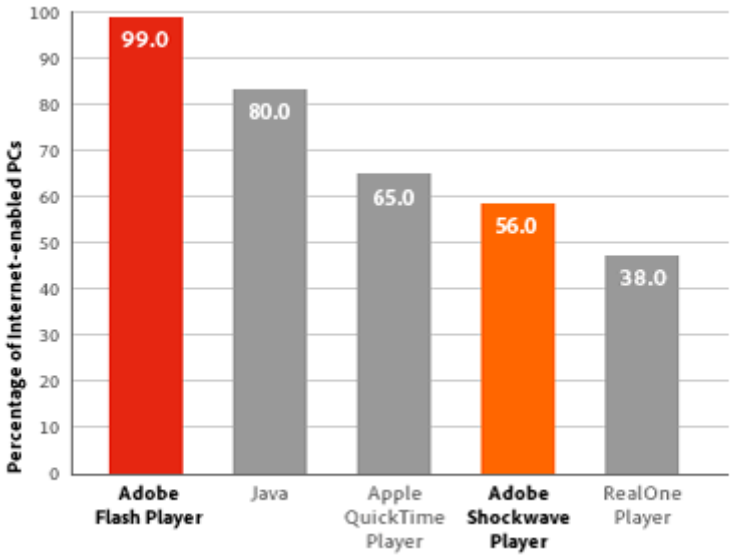
A RIA a szerverrel csak aszinkron módon kommunikál. Mivel nincs szükség az oldal újratöltésére, ezért csak olyan adatokat fog küldeni a szerver, amire tényleg szükség van. Ezáltal csökken a hálózati forgalom is.

Mivel már nem a szerver van a középpontban, ezért a kliensen is el lehet végezni olyan dolgokat, mint az adatrendezés, adatszűrés, adatkonvertálás vagy éppen adat caching.

Néhány példa a RIA keretrendszerekre:

- Adobe Flex
- Google Web Toolkit (GWT)
- OpenLaszlo
- JavaFX
- Microsoft Silverlight

Azért az Adobe Flex szeretném bemutatni ezek közül, mert egy 2009. szeptemberi felmérés szerint az internetre csatlakozott számítógépek 99%-án megtalálható az Adobe Flash Player.



1. ábra: Flash Player statisztika[1]

1.2 Tervezési minták

A tervezési minták használatával is tudjuk gyorsítani az alkalmazásunk fejlesztését, mivel ezek kipróbált fejlesztési paradigmákat kínálnak. A tervezési minta egy általános, újrafelhasználható megoldás a szoftverfejlesztés sűrűn előforduló problémáira. Ez nem egy kész kód, ami azonnal fel tudunk használni a saját kódunkban. Inkább ez egy leírás vagy sablon, hogy hogyan tudjuk könnyen és gyorsan megoldani az adott problémát. Az objektumorientált tervezési minták általában osztályok közötti kapcsolatokat és interakciókat írnak le.

A webes alkalmazásfejlesztésben az egyik legtöbbet használt architektúrális tervezési minta az MVC (Modell-View-Controller, magyarul Modell-Nézet-Vezérlő). Ez a minta szétválasztja a kliens oldali logikát az adattól és a megjelenítéstől. Ezáltal gyorsítható a fejlesztés, hiszen az egyik programozó csinálhatja a vezérlő logikát, addig egy másik fejlesztheti a felületi megjelenítést.

Egy másik minta, amit sűrűn használnak, az a szingleton tervezési minta. Ezzel a mintával azt tudjuk elérni, hogy egy osztályból csak egy példány legyen az egész alkalmazásban. Mivel

[1] Forrás: http://www.adobe.com/products/player_census/flashplayer/

csak egy példány lehet az adott objektumból, ezért biztosítani kell egy globális hozzáférési pontot.

A Business Delegate tervezési minta arra jó, hogy csökkentse a kliens és a távoli üzleti metódus közötti szoros kapcsolatot. A Business Delegate elrejti a mögöttes üzleti szolgáltatás implementáció részleteit, mint például, hogy hogyan érhetünk el egy távoli szolgáltatást. Nekünk így csak a megfelelő metódus meghívásával kell törődnünk, a többit elvégzi a Delegate.

A FrontController tervezési mintát akkor használjuk, ha akarunk egy olyan osztályt, ami az kezdeti kapcsolattartó pontja a kérések kezelésének. A vezérlő kezeli a felhasználóktól jövő kéréseket, és elirányítja azt a megfelelő helyekre. Így minden felhasználói interakció kezelés egy helyen lesz kezelve. Nem futunk abba a hibába, hogy egy ugyanolyan kérést több helyen is kezelünk, és ha esetleg megváltozna, akkor minden helyen változatni kellene.

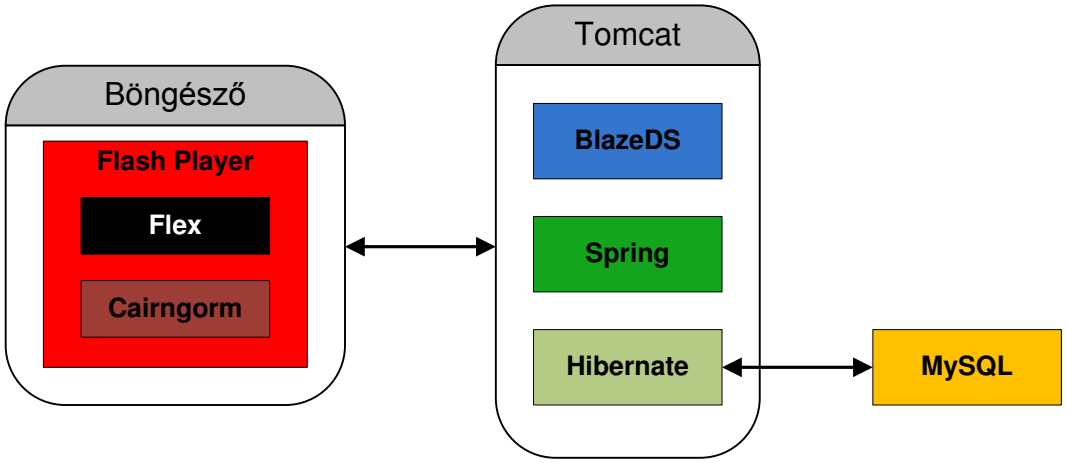
A Command tervezési minta arra szolgál, hogy egy egységbe fogja egy metódus meghívásához szükséges adatokat. Ilyen adatok az osztály, amin meg akarjuk hívni a metódust, maga a metódus és a metódus paraméterei.

A Data Access Object (DAO), vagyis az adat hozzáférés objektum tervezési mintát, arra használhatjuk, hogy egységbe zárja, és elrejti az adatforráshoz kapcsolódást és adatkezelést. A DAO kezeli az adatforráshoz kapcsolódást, és az adatok kikérését vagy tárolását. Adatkikérésnél felelős az értékobjektumok létrehozásáért.

A Value Object (VO), azaz az értékobjektum tervezési minta, arra szolgál, hogy az üzleti adatokat egységbe zárja. A metódusok ezzel kommunikálnak egymással. Nagy előnye, hogy csökkenti az adatbázishoz fordulás kérések számát, hiszen egy entitásról egy kéréssel általában megkapjuk az összes információt.

2 Egy webes alkalmazás lehetséges architektúrájának áttekintése

Én egy RIA alkalmazásnak a következő architektúrát képzelem el.



2. ábra: Egy lehetséges RIA architektúra

A kliens oldalon lenne az Adobe Flex és az Adobe Cairngorm. A Flex egy nyílt forráskódú keretrendszer, ami az Adobe Flash Plyerben fut. A Flex szolgáltatja a HTML-ből már ismert elemeket (gomb beviteli mező, táblázat, stb). A Cairngorm egy mikro architektúra a Flex alkalmazások számára. Ezen architektúra használatával lényegében az MVC tervezési mintát tudjuk egyszerűen használni.

Alkalmazásszerver helyett egy Apache Tomcat web konténert használnék. A Tomcatet egy Java 6-os virtuális gépre (JVM) telepíteném. Ebben futna a BlazeDS, a Spring és a Hibernate. A BlazeDS lényegében egy szerver, ami biztosítja a kapcsolatot a kliens és a szerver oldal között, általában az Adobe saját AMF protokollján keresztül. Azért kell a Tomcat, mert a BlazeDS szervleteket is tartalmaz. A Spring is egy nyílt forráskódú keretrendszer, amivel a szolgáltatásokat implementáljuk a kliens oldal számára. A Hibernate egy objektum-reláció leképező keretrendszer. Segít az Java osztályokat adatbázis rekordokká alakítani.

Adatbázis kezelő rendszernek a MySQL-t a választanám.

Integrált fejlesztői környezetnek az Eclipe-et használnám, hiszen ekkor egy IDE-n keresztül tudjuk a Java és a Flex kódunkat fejleszteni. Azért tudjuk ezt megtenni, mert az Adobe a Flexhez kínál Eclipse plugint.

Ezt az architektúrát egy 4-rétegű architektúraként tudom elképzelni. A réteges architektúrára az a jellemző, hogy minden réteg egy-egy jól definiált feladatot lát el, és a

rétegek csak a közvetlen szomszéd rétegekkel tud kommunikálni. A következő rétegek lennének fentről lefelé haladva:

- Kliens réteg: Ez a réteg tartalmazza a Flexet és a Cairngormot. Feladata a felhasználói felület megjelenítése és módosítása. A felhasználói interakciók hatására meghívni a megfelelő szolgáltatásokat, és a szolgáltatások eredményét figyelembe véve frissíteni a felületen lévő elemeket.
- Szolgáltatás réteg: Ez a réteg tartalmazza a BlazeDS-t. Feladata, hogy kezelje a klienstől jövő kéréseket. Átalakítsa azokat a Java nyelvnek megfelelően, és meghívja az üzleti metódusokat. Az eredményt pedig továbbítsa megfelelő formátumban a kliensnek.
- Üzleti logika réteg: Ez a réteg tartalmazza a Springet és a Hibernate-et. Feladata a megfelelő funkcionalitás biztosítása. Ez a réteg biztosít adatokat az adatrétegnek és kér ki adatot az adatrétegből. Feladata még a különböző háttérfolyamatok elvégzése.
- Adatréteg: Ez tartalmazza a MySQL-t. Feladata az adatok perzisztálása, és az azokon végezhető elemi műveletek biztosítása.

Az üzleti logika és a szolgáltatás réteget akár össze is lehet vonni, mivel együtt futnak egy Tomcat web konténerben. Azért alakítottam ki még egy réteget, hogy jól látszódjon, hogy a kliens csak a szolgáltatásokon keresztül tud kommunikálni a szerverrel.

3 Flex

A Flex kombinálja az asztali és a web alkalmazások előnyeit. A Flex, mint keretrendszer kiterjeszti a Flash API-t, hogy biztosítsa a Flash komponensek előnyeit a Flex keretrendszerrel integrálva. A Flex nagyon sok előre gyártott komponenssel rendelkezik, és ezek nagyon finoman hangolhatók a saját elképzeléseinknek megfelelően. Egyszerűen létrehozhatunk saját komponenst, úgy hogy kiterjesztünk egy meglévőt. A Flex biztosítja, hogy a komponensen használni tudjuk a Flash-ből jól ismert áttűnéseket és animációkat.

Egy Flex alkalmazást futásidőben kiváltott események vezérlik. Ilyen esemény lehet, ha a felhasználó rákattint egy gombra, vagy kiválaszt valamit egy legördülő listából. Ezen eseményekhez pedig hozzá tudjuk rendelni a végrehajtandó utasításokat, ami ActionScript kód.

A Flex programozása lényegében MXML és ActionScript programozás. A Flex alkalmazás Flash bájtkóddá fordul, ami egy SWF (Small Web Format) fájl. Az MXML egy XML alapú, a Flexes alkalmazásunk vizuális kinézetének programozására való nyelv. Az MXML-t arra is használhatjuk, hogy nem vizuális elemeket definiáljunk. Ilyen elem lehet például egy szerveroldali adatok elérésre használható komponens vagy egy animálási effektus. Az ActionScript, hasonlóan a JavaScript-hez az ECMAScript szabványnak megfelelő nyelv. Ezzel tudjuk az alkalmazás kliensoldali viselkedését leírni. Ezek a viselkedések széleskörű funkcionalitásokat foglalnak magukba. Ilyen funkcionalitás például egy esemény kiváltása, vagy egy dialógusablak megjelenítése.

Mivel az MXML fájlok valójában átlagos XML fájlok, ezért bármilyen szövegszerkesztő vagy XML szerkesztő programot használhatunk. Az Adobe azonban biztosít számunkra egy fejlesztőeszközt, aminek a neve Adobe Flex Builder. Igaz ez nem ingyenes, de 60 napig szabadon használhatjuk. A Flex keretrendszer azonban ingyenes, és parancssorból Apache Ant segítségével könnyen tudjuk fordítani a megírt Flex alkalmazásokat.

3.1 Egy Flex alkalmazás felépítése

Azt, hogy hogyan is néz ki egy Flex kód egy példával a legegyszerűbb szemléltetni.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >
       <mx:Panel title="Az első alkamazásom"
         paddingTop="10" paddingBottom="10"
         paddingLeft="10" paddingRight="10" >

         <mx:Label text="Helló Flex!" fontWeight="bold" fontSize="24"/>
```

```
    </mx:Panel>
</mx:Application>
```

Flex példaalkalmazás

3. ábra: Az első Flex alkalmazásunk eredménye

Mivel a forrás ténylegesen egy XML, ezért az első sor tartalmazza az XML verziószámát és a fájl karakterkódolását. A legjobb az UFT-8 karakterkódolást használni, mert ez biztosítja a legnagyobb kompatibilitást a platformok között.

Az <mx:Application> tag az alkalmazásunk fő XML eleme. Egy tárolót reprezentál. A tároló egy felhasználó felület komponens, ami megmondja, hogy a benne lévő elemek hogyan helyezkedjenek el. Alapértelmezetten függőlegesen fentről lefele kerülnek elrendezésre. Nem csak az <mx:Application> tag egyetlen konténer. Ezen belül lehet, akárhány tároló lehet, mint ahogyan a példában is látszik az <mx:Panel> egy másik tároló.

Az MXML tagok tulajdonságai, mint az <mx:Label> komponens text, fontWeight, fontSize tulajdonságokkal a kezdeti értékeket állíthatjuk be. Ezt azonban futási időben akárhogy változtathatjuk az <mx:Script> tag segítségével, ami ActionScript kódot tartalmaz.

Tehát egy MXML által létrehozott felhasználói felület két összetevőből tevődik össze. Az egyik a vezérlő, ami form elemnek felel meg, mit például egy gomb, egy szöveges mező vagy egy rádiógomb csoport. A másik a tároló, egy téglalap alakú terület a képernyőn, ami tartalmazhat más tárolókat és vezérlőket. Beépített elrendezési szabálya van a gyerek komponensek pozícionálására. A tárolókat a felhasználó navigálásra is használjuk.

3.1.1 Kapcsolat az MXML tagok és az ActionScript osztályok között

Az Adobe a Flexet ActionScript osztálykönyvtárként implementálta. Ez komponenseket (tárolók és vezérélők), menedzser osztályokat, adatszolgáltatás osztályokat és egyéb osztályokat tartalmaznak. Alkalmazásfejlesztéskor az MXML-t és az ActionScriptet használjuk ezen osztálykönyvtárral.

Az MXML tag megfelel egy ActionScript osztálynak. A Flex feldolgozza az MXML tagokat és lefordítja az SWF fájlt, ami tartalmazza a megfelelő ActionScript objektumot. Például az ActionScript biztosít egy Label osztályt, ami egy Flexes Label vezérlőt definiál. Az MXML, mint

ahogy a példában is látszik a következőképpen hozzunk létre: <mx:Label text="Helló Flex!"/>. Így ugyanazt érjük el, mintha az ActionScriptben példányosítottuk volna a Label osztályt, és utána a text tulajdonságát beállítottuk volna a „Helló Flex” karaktersorozatra.

3.1.2 Alkalmazásstruktúra

Egy MXML alkalmazást megírhatunk egyetlen fájlként, vagy felbonthatjuk több fájlra is. Az alkalmazás fő fájlja az, amelyik tartalmazza az <mx:Application> tagot. Ebből a fájlból hivatkozhatunk több MXML, ActionScript vagy ezek kombinációját tartalmazó fájlokra.
Az Adobe azt ajánlja, hogy az alkalmazásunkat osszuk fel funkcionális egységekre vagy modulokra, ahol minden modul egy diszkrét feladatot lát el. Ennek a következő előnyei vannak:

- Könnyíti és gyorsítsa a fejlesztést: Az egyes modulokat a programozók párhuzamosan tudják fejleszteni, és egyszerűsíti a hibakeresést is.
- Újrahasználhatóság: Az egyes modulokat újra tudjuk használni más alkalmazásokban, így is gyorsítva a fejlesztés és a tesztelési időt.
- Karbantarthatóság: Könnyebb egy modult tesztelni, hogy nem az egész alkalmazást kellene igénybe venni annak hibájának felderítésére.

A modulok egymásba ágyazhatóság szempontjából nincs megkötés a Flex által. Akármilyen mélységig hivatkozhatnak komponensek komponensekre.

3.2 Flex komponensek

A Flex komponensek hasonlóak a HTML elemekhez. A Flex azonban sokkal többet kínál, mint egy HTML. Ha be szeretném mutatni az összes komponens, akkor elégé terjedelmes diplomamunka lenne ebből. Inkább egy webes alkalmazás használata során nagyon is fontos adatvalidálás, adatformázás és stílus használatát mutatnám be. Az összes Flex 3-as komponens megtekinthető az examples.adobe.com/flex3/componentexplorer/explorer.html címen használati példákkal együtt.

3.3 Adatkötés és egyedi azonosító

Mielőtt rátérnék az előbb felsorolt dolgokra, két dolgot még meg kell említenem. Az egyik az adatkötés, a másik az MXML tagok id attribútuma.

Az adatkötés egy folyamat, mikor a Flex objektum számunkra fontos adatait hozzákötjük egy adatobjektumhoz. Ugyanez igaz fordítva is. Futásidőben automatikusan másolódnak az értékek az objektumok tulajdonságai kötött. Szükség van tehát egy forrás objektumra és egy cél objektumra. Ha az adatot szolgáltató objektum adott tulajdonsága frissül, akkor frissül az ehhez kötött objektum adott tulajdonságának az adata is. A Flex a következő adatkötéseket biztosítja számunkra:

- kapcsos zárójel
- MXML-ben az <mx:Binding> tag
- ActionScript kifejezések: Hasonló a kapcsos zárójeles kötéshez, csak ActionScript osztályban használatosak.
- ActionScript kötések: BindingUtils.bindProperty() metódus segítségével tudunk adatokat kötni egymáshoz. Paraméternek át kell adni a cél/tulajdonság és a forrás/objektum értékeket.

Az első kettőre majd láthatunk példát a következő részekben.

Az MXML tagoknak lehet id tulajdonságuk, és ha van, akkor egyedinek kell lennie az egész alkalmazásban. Ezzel az id-val tudunk hivatkozni az adott objektumra az ActionScriptben. Az MXML fordító ilyen esetben generál egy id nevű publikus változót, ami az adott objektumra hivatkozik. Ezt fel tudjuk használni ActionScript osztályokban vagy az <mx:Script> blokkokban. Ezáltal tudjuk változtatni az adott MXML objektum tulajdonságait, vagy akár meghívni a metódusait.

3.4 Adatvalidálás

Mint már említettem az adatvalidálás nagyon fontos egy alkalmazás zavartalan működéséhez. A Flex a nagyon sok előre gyártott validációs osztályokat kínál nekünk. Ilyenek a bankkártya szám, a pénznem, az e-mail cím, a dátum, a szám, a szöveg, irányítószám és a személyigazolvány szám validátorok. A Flex biztosít olyan validátort is, amelynek reguláris kifejezéseket adhatunk meg, ezáltal egy adott mintára is tudunk ellenőrzéseket végezni. Nem mutatom be mindegyiket, csak azt, hogy hogyan lehet használni őket. Ha ezek nem felelnek meg az elvárásainknak, létrehozhatunk saját validátor osztályt is. A Flex alapértelmezetten angol hibaüzeneteket jelenít meg, de ezt tudjuk módosítani a megfelelő tulajdonság értékének a megváltoztatásával.

A validációt azért jó a kliens oldalon elvégezni, mert így csökken a hálózati forgalom. Persze nem tudunk mindent a kliens oldalon validálni, van olyan adat, amit csak szerver oldalon lehet.

3.4.1 Validálás előidézése ActionScripten keresztül

A Validator osztály és leszármazott osztályai rendelkeznek egy validate() metódussal, amivel elérhetjük, hogy az ellenőrzés megtörténjen. A Validator listener attribútumának azt az objektumot kell megadni, amin majd megjelenik a validálás vizuális változása. Ezt kötelező megadni, hogy a Flex tudja, hol kell jelezni a hibás adatbevitelt.

```
public function validateDate(): void {
	var validator:DateValidator = new DateValidator();
	validator.listener=birthday;
	validator.required=true;
	validator.inputFormat="YYYY.MM.DD"
	validator.validate(birthday.text);
}
```

Egy dátumvalidátor

Vannak még egyéb tulajdonságai is, de ezeket tartom a legfontosabbaknak. Ezt előidézni úgy tudjuk, ha meghívjuk a validateDate() metódust, például egy gomb kattintás eseményéhez hozzárendelve. Alapértelmezetten angol hibaüzeneteket jelenít meg, de ezt felül tudjuk definiálni. Például, ha a required értéke igazra van állítva, és nincs semmi az ellenőrizendő mezőben, akkor az adott validátor requiredFieldError üzenet fog megjelenni. Felüldefiniálni a következőképpen lehet:

```
validator.requiredFieldError="A születési dátum megadása kötelező."
```

A listener tulajdonságnak beállított birthday érték egy DateField objektum. Az inputFormat mondja meg, hogy milyen formátumú lehet a dátum. Jelen esetben a négy számjegyű év egy ponttal, utána szóköz nélkül egy kétszámjegyű hónap egy ponttal, majd végül ismét szóköz nélkül a két számjegyű nap. Végül meg kell hívni a validátor validate() metódusát a vizsgálandó szöveggel. A végeredmény a következő:

4. ábra: Dátumvalidátor működés közben

A hiba szövege csak akkor jelenik meg, ha az adott objektum felé pozícionáljuk az egérmutatót. Amíg ezt nem tesszük meg, addig csak a hibás beviteli mező piros kerete tájékoztat minket, hogy valami nincs rendben a bevitt adattal.

A Validator osztálynak van egy statikus validateAll() metódusa. Ennek azokat a validátorokat lehet átadni egy tömbben, amelyeknek le akarjuk futtatni a validate() metódusát. Ilyenkor azonban a validáló objektumnak be kell állítani a source és property tulajdonságait, mert a validate() metódus paraméter nélkül hívódik meg.

3.4.2 Adatkötésen keresztüli validáció

Használhatunk olyan validációt is, ami adatkötéshez kapcsolódik. A lentebbi példában az alapértelmezett valueCommit esemény van használva a validációra. Ez azt jelenti, hogy akkor lesz adatellenőrzés, ha a felhasználó átlép egy következő mezőre, tehát az adott objektum elveszti a fókuszt.

```
<mx:EmailValidator id="emailValidator"
        source="{email}"
        property="text"
/>
```

A source tulajdonság mondja meg, hogy melyik objektumon, a property pedig hogy ezen objektum melyik tulajdonságán kell az ellenőrzést végrehajtani. Jelen esetben az email objektum egy <mx:TextInput> komponens. A felhasználó által beírt e-mail cím az <mx:TextInput> text tulajdonságában tárolódik, ezért a property értékének a text szöveget adjuk meg. Mivel itt nem adtuk meg a listener tulajdonságot, ezért a Flex a soruce objektumhoz fogja kötni a validációs hibákat.
A source tulajdonság támogatja a ponttal elválasztott karaktersorozatot, hogy egymásba ágyazott komponensekre is lehessen hivatkozni.

3.4.3 Validálás eseményekhez kötve

Eseményekhez úgy tudjuk kötni az adatellenőrzést, hogy a validátor trigger és triggerEvent tulajdonságát beállítjuk. A trigger adja meg azt a komponenst vagy objektumot, ami generálja az eseményt. Ha ez hiányozna, akkor a Flex a source tulajdonságban megadott értéket használja fel. A triggerEvent pedig azt mondja meg, hogy mely - a triggerben megadott komponens által kiváltott - eseményre kell lefutnia a validációnak. Ez alapértelmezetten a valueCommit. Ha nem szeretnénk egy eseményre sem adatellenőrzést kiváltani, akkor egy üres karaktersorozatot kell beállítani.

```
<mx:EmailValidator id="emailValidator"
        source="{email}"
        property="text"
        trigger="{email}"
        triggerEvent="change" />
```

Ilyenkor, ha az email objektum kiváltja a change eseményt, akkor bekövetkezik a validáció. Jelen esetben minden egyes beírt karakter után lefut az adatellenőrzés.

3.5 Adatformázás

Egy másik hasznos lehetőség, amit a Flex biztosít számunkra az az adatformázás. Ennek akkor van jelentősége, ha az adatbázisban máshogy tároljuk az adatot, mint ahogy a felületen megjelenítjük. A Flex a következő adatformázó komponenseket kínálja nekünk: pénzformázót, dátumformázót, számformázót, telefonszám-formázót és irányítószám-formázót. A formázandó nyers adat először szöveggé konvertálódik, majd csak ezután meg végbe rajta a megadott formázás. A felsorolt formázók mindegyikének az ősosztálya a Formatter osztály, aminek van egy format() metódusa, aminek át kell adni a formázandó szöveget. Menete a következő:

1. Létrehozunk egy MXML formázó tagot, aminek beállítjuk a formátumot.
2. Meghívjuk a létrehozott objektum format() metódusát kapcsos zárójelben, a formázandó értéket átadva neki paraméterül.

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
	<mx:Script>
		<![CDATA[
			[Bindable]
			private var date:Date = new Date();
		]]>
	</mx:Script>
	<mx:DateFormatter id="dateFrm"
			formatString="YYYY MMMM DD"/>

	<mx:Panel label="Aktuális dátum">
		<mx:Label text="{dateFrm.format(date)}" />
	</mx:Panel>
</mx:Application>
```

Példa a formázó használatára

Ebben az egyszerű példában az aktuális dátum kerül kiírásra, mégpedig úgy, hogy az évet követően a hónap betűvel lesz kiírva.

3.6 Stílusok

Stílusokat CSS (Cascading Style Sheets, stílusleíró nyelv) segítségével adhatunk a komponensekhez. Megváltoztathatjuk a betűk, a színek és a kinézet tulajdonságait. Ezek a tulajdonságok megváltoztathatóak globális szinten vagy egyénileg egy-egy komponensen. A CSS stílusok az SWF fájlba fordulnak bele, tehát nem futási időben dolgozza fel a web

böngésző, mint a HTML-nél. A komponensek x, y, width, height nem stílus tulajdonságok, ezért ez CSS-sel nem tudjuk állítani. Következőképpen tudjuk a komponenseknek a stílusát beállítani:

- Külső CSS fájl használatával
- <mx:Style> tag segítségével
- setStyle() metódus segítségével
- A komponens deklarációjánál a kódba égetve (Inline)

Nem fontos a CSS stílusneveket használni, használhatjuk egy komponens tulajdonságnevét is egy stílus definiálásánál. A Flex megfelelően fogja értelmezni. Gondolok itt olyanra, hogy a CSS-ben a háttérszín megadása background-color megadásával történik, a Flex tulajdonság neve pedig backgroundColor.

3.6.1 Külső CSS fájl használata

Ez a legegyszerűbb módja, hogy stílust alkalmazzunk. Ilyenkor nem kell az ActionScriptet igénybe venni. A stílusok definiálhatnak globális stílusokat, amiket minden vezérlő örököl, vagy a stílusok egy osztályát, amit csak bizonyos vezérlők használnak.
Ehhez fel kell venni a következő sor fő MXML fájlba közvetlenül az <mx:Application> tag után:

```
<mx:Style source="hu/housing/asserts/style.css" />
```

Ezzel megmondjuk a Flexnek, hogy hol található a stílusfájl, amit használni szeretnénk. Mivel a source tulajdonságot megadtuk, ezért az <mx:Style> tagnak üresnek kell lennie. A stílusfájl tartalmazhat típus- és osztályszelektorokat is.
A Flex tartalmaz egy globális stíluslapot, ami a framework.swc fájlban található default.css stílusfájl. Ez tartalmazza az szinte összes Flex komponens globális osztály- és típusszelektorát. A Flex ezt mindig használja az komponensekre, ha nincs felüldefiniálva a valami a saját CSS fájlunkban az adott komponensre nézve.

3.6.2 Az <mx:Style> tag

Akkor használjuk ezt, ha az aktuális MXML fájlra és gyermekeire akarunk stílust alkalmazni, vagy ha egyéni kinézetet akarunk adni egy komponensnek. E tagon belül akármilyen CSS szintaktikájú stílust megadhatunk. Ugyanazokat tartalmazhatja, mint egy külső CSS fájl.

```
<mx:Style>
        .reserverFormHeadingStyle {
                fontSize: 15;
            color: #9933FF;
             }
</mx:Style>
```

Itt egy egyéni szelektort adtunk meg ezért, meg kell mondani, hogy mely komponensre akarjuk ezt alkalmazni. Ezt a komponens styleName tulajdonságával tudjuk beállítani.

```
<mx:FormHeading label="Foglaló regisztráció"
    styleName="reserverFormHeadingStyle" />
```

3.6.3 A setStyle() metódus

E metódust minden Flex komponens tartalmazza. Segítségével futásidőben tudjuk a stílusokat változtatni az egyes komponenseken. Viszont ez nagyobb munkát igényel a kliens géptől, tehát csak óvatosan használjuk.

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script><![CDATA[
        public function changeLabel1Style():void {
            label1.setStyle("fontSize",15);
            label1.setStyle("color",0x9933FF);
        }
    ]]></mx:Script>

    <mx:Label id="label1" text="Ez fog megváltozni!"/>
    <mx:Button id="button1" label="Változtass!" click="changeLabel1Style()"/>

</mx:Application>
```

Példa a setStyle() metódus használatára

3.6.4 Inline CSS

Ezt csak akkor használjuk, ha tényleg csak egy adott komponenst kell egyedi stílussal ellátnunk. Ez a leggyorsabb, ha stílust akarunk alkalmazni egy komponensre, hiszen itt nem kell ActionScript blokkot használni vagy metódusokat hívogatni. A következőképpen néz ki:

```
    <mx:Button id="myButton" color="0x9933FF" fontSize="15" label="Kattints!"/>
```

4 Cairngorm

A Cairngorm az Adobe Flex vagy AIR alkalmazások számára egy nyílt forráskódú mikro architektúra. Az MVC tervezési mintán alapul. Lényege, hogy a modellt a nézetet és a vezérlőt elkülönítse egymástól. Ennek előnye, hogy csapatban könnyebb lesz dolgozni, mivel mindenki a saját feladatára tud koncentrálni. Másik nagy előnye a kód karbantarthatósága, a későbbi fejlesztése esetén nem lesz az alkalmazás átláthatatlan.

A modell feladata, hogy tárolja az adatokat, amiket a nézet és a vezérlő használ. Soha nem hivatkozhat e két elemre, hogy független maradjon tőlük. A nézet feladata a modellben lévő adatok megjelenítése, és felhasználói események generálása. Tehát a nézet függ a modelltől. A vezérlő feladata a nézet által generált események alapján műveletek végrehajtása. A vezérlő frissítheti a modellben lévő adatokat. Tehát a vezérlő tud a modellről, de nem tudhat semmit a nézetről.

A Cairngorm MVC által felhasznált tervezési minták a value object, a front controller, a command és a delegate minták.

4.1 A Cairngorm részei

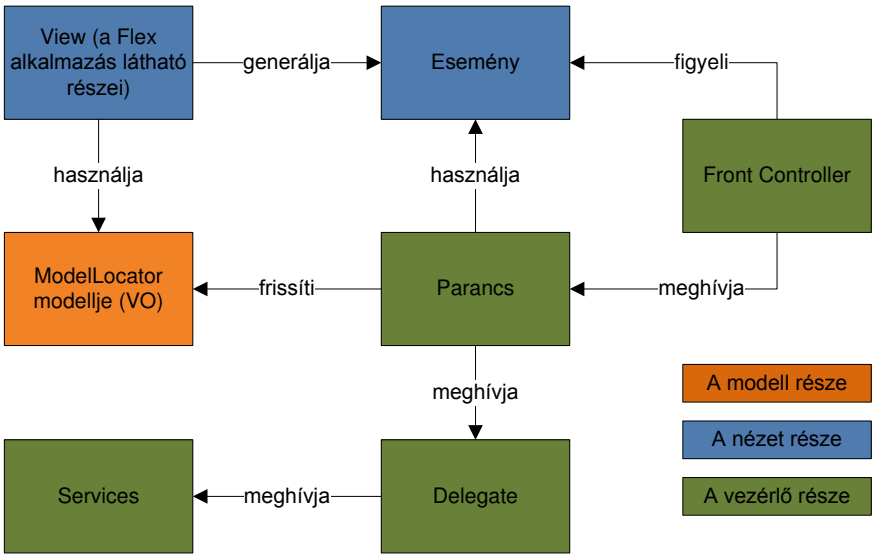
- *IValueObject interfész:* Egy jelölő interfész, ami segíti a kód olvashatóságát. Beazonosítja azokat az osztályokat, amiket VO-ként szeretnénk használni, hogy az alkalmazás rétegei között adatokat továbbítsunk. Tehát ennek az interfésznek nincs implementálandó metódusai.
- *IModelLocator interfész:* Ez is egy jelölő interfész. Az saját „modell felderítőket" jelölhetjük vele. Az MVC architektúrában a modellt implementálhatjuk ezzel az interfésszel. Az alkalmazásban ez egy szingletont jelent. A kliens oldali érték objektumokat tárolására használjuk. Így a kliens oldalon minden VO-ból globálisan csak egy darab lesz, ezt használjuk az egész alkalmazásban.
- *CairngormEvent osztály:* Ezt az osztályt arra használjuk, hogy megkülönböztessük a Cairngorm eseményeket a Flex keretrendszer által kiváltott eseményektől. Ez kötelező a Cairngorm események szétküldéséhez. Az esemény tipikusan a felhasználó cselekedeteinek eredményeként létrejövő üzenetszórás. Ilyen felhasználói interakció lehet egy kattintás, egy menünek a kiválasztása, egy billentyű lenyomása, stb.

- *CairngormEventDispatcher osztály:* Ezt az osztályt használhatják a fejlesztők, hogy az eseményeket szétterjessze az alkalmazásban. Ezek az események a felhasználók által kiváltott eseménynek. Ez az osztály is egy szingleton, ami biztosítja, hogy egy és csak egy osztály legyen felelős az üzenetek szétszórására.
- *FrontController osztály:* Ezen osztály segítségével tudjuk a felhasználók által kiváltott eseményeket továbbítani a megfelelő parancs osztályok felé. Tehát a Cairngorm események és a parancsok összerendelését tartalmazza. Mikor egy eseményt kiváltunk, akkor az eseményhez kapcsolt parancs fog végrehajtódni.
- *ICommand interfész:* Ez az interfész biztosítja a kapcsolatot a FrontController és a parancs osztályok között. A FrontController figyeli a számára érdekes eseményeket (amiket beregisztráltunk az addCommand segítségével), és ha bekövetkezik egy, akkor a vezérlést átadja annak a parancs osztálynak, amelyik az eseményhez van rendelve. Ekkor a parancs osztály execute() metódusa fog lefutni, ami az ICommand interfész implementálandó metódusa. Ez a belépési pontja a parancsnak.
- *ServiceLocator osztály:* Ezt az osztályt kell kiterjeszteni egy MXML komponensen keresztül. Ebben vannak definiálva a távoli adatszolgáltatások.

4.2 Egyéb használt interfészek és osztályok

- *IResponder interfész:* Implementálnia kell a parancs osztályoknak, hogy kezelni tudják a távoli vagy aszinkron hívások válaszait. Két metódust kell implementálni, az egyik az eredményt kezeli, a másik a hibát.
- *Delegate osztály:* Ez egy saját osztály, ami a Business Delegate tervezési minta alapján kell kialakítanunk. Itt kerülnek meghívásra a távoli vagy aszinkron hívások a ServiceLocator segítségével.

4.3 Cairngorm a gyakorlatban



5. ábra Cairngorm vázlat

4.3.1 Az értékobjektum, azaz a ValueObject

ActionScript osztályok, implementálja az IValueObject interfészt. Hasonló a JavaBean-hez, tehát csak értéket tárolnak, nincs nekik üzleti, csak mezőhozzáférő metódusaik. Van neki egy alap, paraméter nélküli konstruktora. A Flex alkalmazások VO-i inkább publikus mezőket részesítik előnyben. Ilyenkor a get és a set metódusok implicit módon jönnek létre.

```
package hu.housing.model.vo
{
	import mx.collections.ArrayCollection;

	[Bindable]
	[RemoteClass(alias="hu.housing.domain.Reserver")]
	public class ReserverVO
	{
		public function ReserverVO() {}

		public var id:Number;
		public var loginName:String = "";
		public var password:String = "";
		public var fullName:String = "";
		public var email:String = "";
		public var homeAddressLine:String = "";
		public var billAddressLine:String = "";
		public var birthday:Date;
		public var registrationDate:Date;
		public var lastLogin:Date;
		public var phoneNumbers:ArrayCollection = new ArrayCollection();
	}
}
```

Egy értékobjektum példa

A RemoteClass meta adattag az ActionScript osztályokat képezi le szerver oldali Java osztályokra. Az alias paraméter segítségével megmondhatjuk, hogy melyik osztály felel meg a VO-nak. A Java osztályoknak ugyanilyen nevű adattagokkal kell rendelkezniük. Ha valamelyik adattag hiányzik, akkor figyelmeztet minket a Flex, igaz csak futás időben. Kivétel nem váltódik ki.

A Bindable meta adattag beazonosítja azokat a tulajdonságokat, amelyeket forrásként használhatunk egy „kötő” kifejezésben a View rétegben. A kötés azt jelenti, hogy ha a forrás objektum megváltozik, akkor a cél objektum értéke is megváltozik, tehát a Flex keretrendszer automatikusan átmásolja a megváltozott értéket. Ha ezt egy osztálydefiníció előtt adjuk meg, akkor az osztály összes adattagja használható lesz „kötő” kifejezésekben. A Flexben egy lehetséges „kötő” kifejezés a kapcsos zárójelek között megadott adattag.

```
<mx:FormItem label="Bejelentkezési név:" required="true">
	<mx:TextInput id="loginName" text="{model.reserver.loginName}"/>
</mx:FormItem>
```

A példában, ha megváltozik a ModelLocator reserverVO, akkor a felületen is látszani fognak ezek a változások. Ha viszont a felületen változtatom meg a foglaló bejelentkezési nevét, akkor nem fog megváltozni a VO-ban az értéke. Erre találták ki az mx:Binding tagot.

```
<mx:Binding source="loginName.text" destination="model.reserver.loginName" />
```

4.3.2 A modellek tárolója, azaz a ModelLocator

```
package hu.housing.model {
	import com.adobe.cairngorm.model.IModelLocator;
	import hu.housing.model.vo.PhoneVO;
	import hu.housing.model.vo.ReserverVO;

	[Bindable]
	public class ModelLocator implements IModelLocator {
		private static var modelLocator : ModelLocator;
		public function ModelLocator()
		{
			if ( modelLocator != null )
				throw new Error( "Only one ModelLocator instance
                                    should be instantiated" );
		}
		public static function getInstance() : ModelLocator {
			if ( modelLocator == null )
				modelLocator = new ModelLocator();
			return modelLocator;
		}
		public var reserver : ReserverVO = new ReserverVO();
		public var tempPhoneVO:PhoneVO;
	}
}
```

A globális adatok tárolója a ModelLocator

A saját modell lokátorunkból a kliens oldalon mindig egynek kell lennie. Ezért szingletenként kell kialakítani az osztályt. Ha példányosítani akarja egy fejlesztő, és már van egy példánya, akkor kivétel váltódjon ki. Legyen egy olyan statikus metódusa, ahol ezt az egy példányt el tudjuk érni. Ezen keresztül használjuk a Viewban az MVC architektúrában. Ennek az osztálynak az a szerepe, hogy a globális adatokat egy helyen tároljuk megkönnyítve ezzel a fejlesztők munkáját.

Használata:

```
[Bindable]
public var model:ModelLocator = ModelLocator.getInstance();
```

4.3.3 Az esemény, azaz a CairngormEvent

```
package hu.housing.control.events
{
	import com.adobe.cairngorm.control.CairngormEvent;
	import hu.housing.model.vo.ReserverVO;

	public class ReserverEventCreateOrSave extends CairngormEvent
	{
		public static var EVENT_RESERVER_CREATE_OR_SAVE:String =
			"event-reserver-createOrSave"
		public var reserverVO:ReserverVO;

		public function ReserverEventCreateOrSave(reserverVO:ReserverVO)
		{
			super(EVENT_RESERVER_CREATE_OR_SAVE);
			this.reserverVO = reserverVO;
		}
	}
}
```

A foglaló létrehozása esemény

Az események tartalmazzák azt az objektumot, amit fel akarunk dolgozni, mikor szétküldésre kerül. Például egy táblázatnál azt a sort, amelyikre rákattintottak, vagy éppen az újonnan létrehozandó VO-t mikor a mentés gombra kattintunk. A szétküldött eseményeket a kontrollerek figyelik, és a feliratkozott eseményekhez végrehatják a hozzárendelt parancsot. Mit láthatjuk, az eseménynek adni kell egy azonosítót, aminek az egész alkalmazásban egyedinek kell lennie. Ezzel az azonosítóval tudjuk hivatkozni a kontroller osztályban. Az ősosztály konstruktorát meg kell hívni, az egyedi azonosítónkkal, hogy létrehozzuk az eseményt a megadott névvel, ami egyben a típusa lesz a saját eseményünknek. Egy eseményt a következő módon küldhetünk szét:

```
private function createOrSaveReserver():void {
	var reserverEventCreateOrSavene:ReserverEventCreateOrSave =
		new ReserverEventCreateOrSave(model.reserver);
	reserverEventCreateOrSavene.dispatch();
}
```

Tehát az adott esemény dispath() metódusát kell használni az adott esemény szétküldéséhez. Vagy használhatnánk a CairngormEventDispatcher osztályt is.

4.3.4 Az irányító, azaz a FrontController

Ez tartalmazza a felhasználó által kiváltott események és a parancsok összerendelését. Használni úgy tudjuk, hogy kiterjesszük a Cairngorm FrontController osztályát, és az addCommand metódusával összerendeljük az eseményeket az egyedi azonosítjuk és a parancsokat az osztálynevük alapján. Az összerendelés azzal is jár, hogy feliratkoztatjuk a FrontController osztályunkat az esemény bekövetkezésének a figyelésére. Ha egy esemény bekövetkezik, akkor a parancs execute() metódusa fog lefutni. A parancsok a kontroller osztályhoz gyenge referenciával vannak hozzákapcsolva, ami azt jelenti, ha a parancsot törölte a szemétgyűjtögető, akkor a kontrollerbeli hivatkozás is törölve lesz.

```
package hu.housing.control
{
	import com.adobe.cairngorm.control.FrontController;
	import hu.housing.control.commands.ReserverCreateOrSave;
	import hu.housing.control.events.ReserverEventCreateOrSave;

	public class MainController extends FrontController
	{
		public function MainController()
		{
		addCommand(ReserverEventCreateOrSave.EVENT_RESERVER_CREATE_OR_SAVE,
			ReserverCreateOrSave);
		}
	}
}
```

A saját FrontController osztályunk

Ezt példányosítani úgy tudjuk, hogy az alkalmazásunk fő komponensében – ami tartalmazza az <mx:Application> tagot – a következőt írjuk:

```
	<control:MainController id="mainController" />
```

Ezt csak akkor fog működni, ha már előtte - hasonlóan a Java importhoz – létrehoztuk a következő XML dokumentumokban is használatos névteret:

```
<mx:Application
	xmlns:mx="http://www.adobe.com/2006/mxml"
	xmlns:control="hu.housing.control.*">
```

4.3.5 A parancs, azaz a Command

A parancs felelős a belső vagy külső metódushívások elindításáért a business delegate-en keresztül. A saját parancsainknak implementálnia kell a Cairngorm ICommand interfészét. Ezzel egy metódust kell megvalósítanunk, aminek a neve execute(). Ez fog lefutni, mikor a parancshoz rendelt esemény bekövetkezik. A parancs osztályunkat a FrontController fogja példányosítani. Az execute() metódus paraméterként megkapja a kiváltó eseményt. A parancsok kezelik általában a távoli hívások eredményeit. Ezért implementálnunk kell az IResponder interfészt. Ennek két megvalósítandó metódusa van. Az egyik a result(), amikor a távoli eljárás eredménnyel tér vissza, akkor ez fog meghívódni. A másik a fault(). Ez akkor hívódik meg, ha valamilyen gond volt a szolgáltatással, például egy kivétel váltódott ki. A paraméterei ezeknek a metódusoknak legtöbbször eseménynek (ResultEvent és FaultEvent), amelyek tartalmazzák az eredményt illetve a hiba okát. Ha eredménnyel tért vissza a távoli hívásunk, akkor általában a ModelLocator adott elemét frissítjük, vagy beállítjuk. Hiba eseték pedig egy figyelmeztető ablakban megjelenítjük a hiba okát.

```
package hu.housing.control.commands
{
	import com.adobe.cairngorm.commands.ICommand;
	import com.adobe.cairngorm.control.CairngormEvent;
	import hu.housing.control.delegates.ReserverDelegate;
	import hu.housing.control.events.ReserverEventCreateOrSave;
	import hu.housing.model.ModelLocator;
	import hu.housing.model.vo.ReserverVO;
	import mx.controls.Alert;
	import mx.rpc.IResponder;
	import mx.rpc.events.FaultEvent;
	import mx.rpc.events.ResultEvent;

	public class ReserverCreateOrSave implements ICommand, IResponder {
	public function ReserverCreateOrSave() {}

	public function execute(event:CairngormEvent):void {
		var delegate : ReserverDelegate = new ReserverDelegate( this );
		var reserverEventCreateOrSave : ReserverEventCreateOrSave =
			event as ReserverEventCreateOrSave;
		delegate.createOrSaveReserver( reserverEventCreateOrSave.reserverVO );
	}

	public function result(data:Object):void
	{
		var event:ResultEvent = data as ResultEvent;
		ModelLocator.getInstance().reserver = event.result as ReserverVO;
	}

	public function fault(info:Object):void
	{
		var faultEvent:FaultEvent = info as FaultEvent;
		Alert.show(faultEvent.toString(), "Hiba");
	}
	}
}
```

A foglaló létrehozó vagy módosító parancs

4.3.6 Az üzleti metódus delegáló, azaz a Business Delegate

A szerver oldali üzleti metódusainkat nem egyből a parancs osztályból hívjuk meg, hanem egy delegáló osztályon keresztül. Be kell állítani neki egy válaszkezelő osztályt, ami a parancs osztályunk, hiszen implementálja az IResponder interfészt. Ezt a példányosításnál adjuk át paraméterként. Példányosítani a parancs osztályban kell. Ekkor állítódik be a szolgáltatás is, amivel meghívjuk az adott távoli metódusunkat.

```
package hu.housing.control.delegates
{
	import com.adobe.cairngorm.business.ServiceLocator;
	import hu.housing.model.vo.ReserverVO;
	import mx.rpc.AbstractService;
	import mx.rpc.AsyncToken;
	import mx.rpc.IResponder;

	public class ReserverDelegate
	{
		private var responder : IResponder;
		private var service : AbstractService;

		public function ReserverDelegate( responder : IResponder )
		{
			this.service = ServiceLocator.getInstance().
				getRemoteObject( "reserverService" );
			this.responder = responder;
		}

		public function createOrSaveReserver(reserverVO:ReserverVO):void {
			var call:AsyncToken = service.createOrSaveReserver(reserverVO);
			call.addResponder(responder);
			service.logout();
		}
	}
}
```

A Business Delegate osztály

Az adott delegate metódus meghívásakor a saját szolgáltatásunk egy AsyncToken objektum lesz a visszatérési értéke. Ennek kell beállítani a válaszkezelő osztályunkat. Ez az AsyncToken osztály fogja meghívni a válaszkezelő osztály – jelen esetben a parancs osztályunk - result vagy fault metódusát, attól függően, hogy volt-e hiba vagy nem. Ha végeztünk, akkor a szolgáltatásból ki kell jelentkeznünk, hogy ne foglaljuk feleslegesen az erőforrásokat.

4.3.7 A szolgáltatás tárolója, a ServiceLocator

Ez is egy szinglenton osztály a távoli szolgáltatás komponensek tárolására. Ezt az osztályt egy MXML komponensként kell megadni, amely tartalmazza a távoli szolgáltatásokat. Ennek az osztálynak ki kell terjesztenie a Cairngorm ServiceLocator osztályát. Így könnyen és gyorsan tudunk szolgáltatásokat létrehozni, és elérni.

```
<cairngorm:ServiceLocator
	xmlns:mx="http://www.adobe.com/2006/mxml"
	xmlns:cairngorm="com.adobe.cairngorm.business.*">
	<mx:Script>
		<![CDATA[
		[Bindable]
		private var endpoint:String =
			"http://localhost:8080/HousingBusiness/messagebroker/amf";
		]]>
	</mx:Script>

	<mx:RemoteObject id="reserverService"
		destination="reserverService"
		endpoint="{endpoint}"
		showBusyCursor="true"
		 />
</cairngorm:ServiceLocator>
```

A szolgáltatás osztály

Itt a példában egy távoli eljáráshívás látható éppen. Az mx:RemoteObject tagról a későbbiekben lesz szó. Példányosítani az alkalmazás fő komponensében kell, a FrontController osztályhoz hasonlóan:

```
	<service:Services id="services"/>
```

ServiceLocator.getInstance().getRemoteObject() metódussal lehet kikérni a távoli eljáráshívás szolgáltatást. Itt lehet még megadni WebService, HTTPService és saját távoli adatszolgáltatásokat.

5 BlazeDS

A BlazeDS egy nyílt forráskódú, szerver alapú technológia, amely Java távoli eljáráshívás (RPC) és Java üzenetküldő szolgáltatás (JMS) technológiákat használ. A BlazeDS segít integrálni a Flex-et és a Java-t. Egy üzenet alapú architektúrára épül, amely üzenetcsatornákat használ, hogy kommunikáljon a kliensekkel.

Használatának az előnyei:

- Java metódusokat hívhatunk Flex alkalmazásokból.
- Segít leképezni a Java osztályokat AS3 osztályokra, és visszafele.
- Valós idejű üzenettovábbítás.
- Kezeli a Flex és a Java között kommunikációs csatornát.

A BlazeDS egy Java web alkalmazás. Tehát elég neki egy szervlet konténer, amit bármelyik alkalmazásszerverben megtalálható. Népszerű nyílt forráskódú alkalmazásszerverek: Tomcat, JBoss AS, GlassFish, Geronimo.

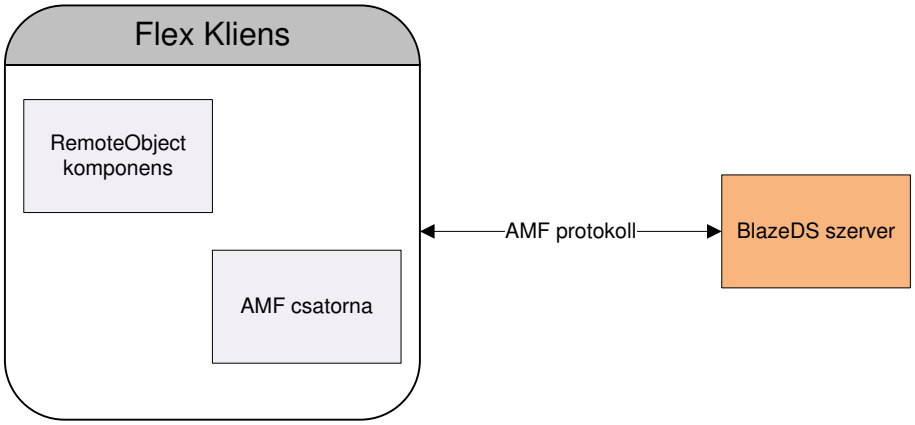
5.1 A BlazeDS architektúrája

A kliens, hogy üzenetet küldjön a hálózaton keresztül, csatornát használ. Az üzenetcsatornákat üzenetcsatorna halmazokká fogja össze, attól függően, hogy milyen csatornákat használunk (pl.: AMF, HTTP). A csatorna egységbe zárja az üzenetformátumot, a hálózati protokollt és a hálózat viselkedését, hogy elkülönüljön a szolgáltatástól és az alkalmazáskódtól. A csatorna formázza és átalakítja az üzenetet hálózat specifikus formátummá, és kézbesíti a szerver megadott végpontjának. A csatornák Java végpontokkal kommunikálnak. A végpont kibontja az üzenetet a protokollspecifikációnak megfelelően, és átadja Java formában az üzenetbrókernek. Az üzenetbróker megfejti, hogy melyik szolgáltatásnak kell küldeni az üzenetet, és elküldi a megfelelőnek.

A kommunikáció az Adobe AMF (Action Message Format) protokollján alapul az RPC (Remote Procedure Call, távoli eljáráshívás) esetén. Ez egy alkalmazás szintű protokoll, amely a HTTP protokollt használja. Az RPC szolgáltatás a kérés-válasz mintát használja az üzenetek kicserélésére. Működése a következő: A kliens elküldi az üzenetet a szervernek feldolgozásra, a szerver visszaválaszol a kliensnek a feldolgozás eredményével.

A BlazeDS nem csak RPC szolgáltatásokat tud kezelni, de én csak ezt jellemezném, mert az üzleti alkalmazás elvárásainak szerintem ez felel meg a legjobban, ez biztosítja a leggyorsabb választ. A BlazeDS a következő szolgáltatásokat kezeli még: HTTPService, WebService.

5.2 A BlazeDS kliensoldali architektúrája az RPC szemszögéből



6. ábra: A BlazeDS kliensoldali architektúra vázlata

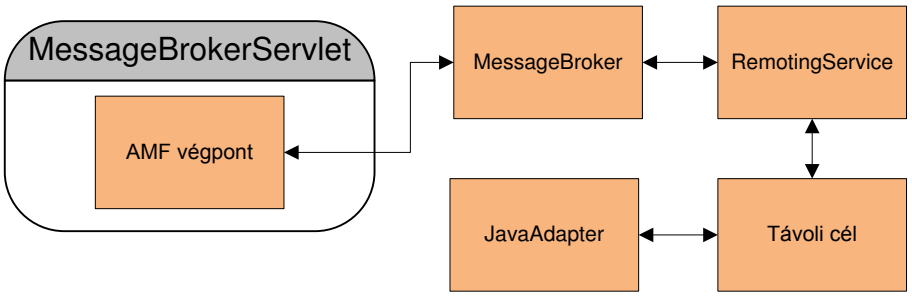
A RemoteObject komponens a Flex SDK része. A csatorna feladata, hogy lebonyolítsa a kommunikációt a BlazeDS szerver adott végpontjával.

A RemoteObject objektum ad hozzáférést a távoli Java osztály metódusaihoz az AMF csatornán keresztül. Ahhoz, hogy hozzáférjünk, példányosítani kell, és a következő paramétereket kell neki megadni:

- *endpoint*: A végpontnak az elérési útját határozzuk meg ezzel a paraméterrel.
- *id*: Egy azonosítót kell neki megadni, hogy a kódban tudjunk rá hivatkozni, mikor kikérjük szolgáltatásként.
- *destination*: A szerveroldali célszolgáltatásunk azonosítója. A kikért szolgáltatás ugyanolyan metódusokkal fog rendelkezni, mint a szerveroldali szolgáltatás Java osztály publikus metódusai.

A használatára példa „A szolgáltatás tárolója, a ServiceLocator” és „Az üzleti metódus delegáló, azaz a Business Delegate” részben.

5.3 A BlazeDS szerveroldali architektúrája az RPC szemszögéből



7. ábra: A BlazeDS szerveroldali architektúra vázlata

Az AMF szervlet alapú végpont, ami a Java alkalmazásszerver web konténerében van. Ez azt jelenti, hogy a szervlet kezeli az I/O-t és a HTTP sessionokat a végpont számára. A végpont MessageBrokerServlet által kerülnek példányosításra. A kliens és a végpont csatornákon keresztül kommunikál. A kliensnek és a végpontnak ugyanazt az üzenetformátumot kell használnia ebben a csatornában. A végpont feladata hogy, kicsomagolja a HTTP kérésből az üzenetet és átadja a MessageBrokernek. A végpontokat a services-config.xml fájlban lehet konfigurálni.

Miután megérkezett az üzenet a végponttól, a MessageBroker kideríti, hogy kinek szól az üzenet, azaz a meghatározza a célt, és átadja annak a szolgáltatásnak, amelyik az adott célt kezeli. Jelen esetben ez a RemotingService. Ezt a remoting-config.xml állományban lehet beállítani.

A távoli célokra úgy kell tekinteni, mint a RemotingService által menedzselt célok példányaira. A Flex kliensbeli RemoteObject komponens a RemotingService objektummal kommunikál. A RemoteObject komponensben meg kell adni egy célnak az azonosítóját, ami az adott azonosítóval ellátott távoli célra hivatkozik. A távoli cél nem más, mint egy Java osztály, amin a metódust meg akarjuk hívni. Ezek lesz a Springes beanek, tehát ezt az applicationContext.xml fájlban lehet konfigurálni.

A Java adapter az utolsó láncszem az üzenet feldolgozásban. Mikor az üzenet megérkezik a megfelelő célhoz, az átadódik az adapternek, és teljesíti a kérést. A Java adapter fogja meghívni a metódust ténylegesen az adott Java objektumon. Alapértelmezetten állapot nélküli, azaz minden egyes kérésnél újra példányosításra kerül.

5.4 A services-config.xml

Ez a BlazeDS fő konfigurációs állománya. Ez tartalmazza a szolgáltatások definícióját, biztonsági beállításokat, a csatornák leírását és a szolgáltatások naplózási beállítását.

```
<services-config>

<services>
	<service-include file-path="remoting-config.xml" />
</services>

<security>
	<login-command
		class="flex.messaging.security.TomcatLoginCommand"
		server="Tomcat"/>
</security>

<channels>
<channel-definition id="my-amf" class="mx.messaging.channels.AMFChannel">
	<endpoint
	url="http://{server.name}:{server.port}/{context.root}/messagebroker/amf"
	class="flex.messaging.endpoints.AMFEndpoint"/>

	<properties>
		<polling-enabled>false</polling-enabled>
	</properties>
</channel-definition>
</channels>

<logging>
	<target class="flex.messaging.log.ConsoleTarget" level="Error">
		.
		.
		.
	</target>
</logging>

<system>
	<redeploy>
		<enabled>false</enabled>
	</redeploy>
</system>

</services-config>
```

A services-config.xml fájlból részlet

A szolgáltatások másik fájlból vannak felsorolva, és itt van jelezve, hogy mely fájlok azok. A távoli szolgáltatásokat a remoting-config.xml fájlban adjuk meg. A HTTP szolgáltatásokat a proxy-config.xml és az üzenetszolgáltatásokat a message-config.xml fájlban lehet definiálni.

A biztonság résznél kell megadni, hogy melyik osztályt szeretnénk használni a hitelesítésre és azonosításra. Az alkalmazásszervert is meg kell adni, amin végrehajtjuk ezeket. Ezt a Flex implementálja nekünk, de írhatunk saját biztonságkezelő osztály is.

A következő résznél kell megadni az üzenetcsatorna definíciókat. Mindegyik csatornának rendelkezni kell egy azonosítóval, amivel majd később hivatkozhatjuk. Meg kell adni a végpontnak az elérési útját, és azt a csatornaosztályt, amelyet a kliens használ. A

tulajdonságok függnek a használt csatornától. A polling kikapcsolása azt jelenti, hogy a csatorna nem gyorsítótárazható, tehát nem lehet újra felhasználni, hanem minding új jön létre.

Azt is meg lehet adni, hogy hova naplózzanak a szerveren a szolgáltatások, és milyen hibák kerüljenek naplózásra. Beállítható, hogy mi kerüljön a naplóbejegyzésbe (dátum, a hiba szintje, a hiba kategóriája, milyen prefix-szel). Szűröket is megadhatunk, hogy csak adott osztályok hibái legyenek naplózva.

A rendszer résznél azt lehet megmondani, hogy ha valamelyik XML fájlunk megváltozik, akkor az alkalmazásunk újratelepüljön-e. Meg lehet adni, hogy milyen időközönként és mely állományokat figyelje a BlazeDS.

5.5 A remoting-config.xml

Itt adható meg a szolgáltatás. A szolgáltatásnak kell adni, hogy hivatkozhassunk a célszolgáltatásunkban, és meg kell adni a szolgáltatáskezelő osztályt is.

```
<service id="remoting-service" class="flex.messaging.services.RemotingService">
<adapters>
	<adapter-definition id="java-object"
		class="flex.messaging.services.remoting.adapters.JavaAdapter"
		default="true"/>
</adapters>

<default-channels>
	<channel ref="my-amf"/>
</default-channels>

</service>
```

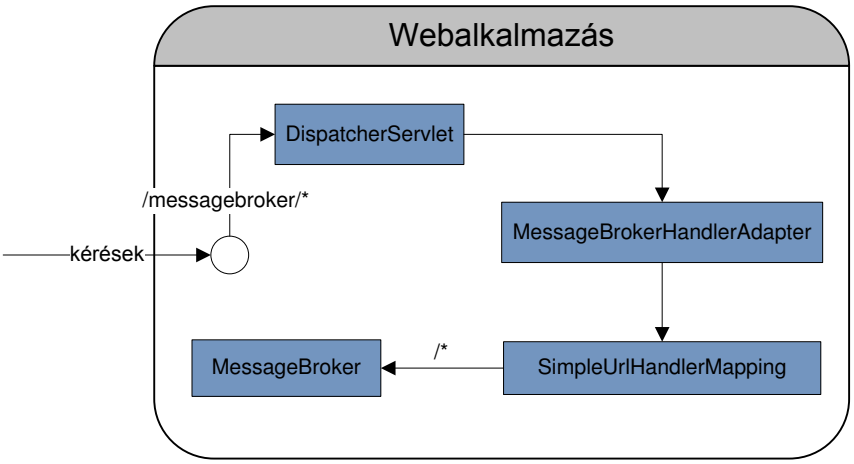
A remoting-config.xml

Az adapter-definition résznél adható meg egy adapter, amit használni szeretnénk. Ha a default tulajdonság igaz, akkor ezt fogja használnia a dinamikusan létrehozott célszolgáltatás (destination service). A Spring BlazeDS integrációja a célszolgáltatásokat ilyen elven hozza létre. Cél definíciók ezért nem szerepelnek itt.

A default-channels taggal alapértelmezett csatornákat adhatunk meg. Ha egy szolgáltatásnak nincs megadva explicit csatorna, akkor a felsorolás sorrendjében próbál egyet használni. Az alapértelmezett csatorna megadása kötelező, ha nem XML fájlból, hanem dinamikusan akarunk célszolgáltatást létrehozni.

5.6 Képbe kerül a Spring

Mivel az üzleti metódusokat Spring segítségével lesznek implementálva és a Spring is kínál BlazeDS integrációt, ezért nem az alap konfigurációt fogom bemutatni, hanem a Spring által megvalósítottat. A Spring BlazeDS integráció a MessageBroker-t a Spring által menedzselt objektummá teszi.



8. ábra: Spring BlazeDS integráció vázlat

A HTTP kérések a Flex klienstől, ami illeszkedik a /messagebroker/* mintára a Springes DispatcherServlet-hez kerül. Ezután a kérés a MessageBrokerHandlerAdapter osztályhoz jut. Ez eldönti a SimpleUrlHandlerMapping segítségével, hogy melyik MessageBroker példánynak kell eljuttatni a kérést. Jelen esetben minden kérést egy MessageBroker-nek továbbítunk. A MessageBroker válasza is hasonló úton halad, csak visszafele a SimpleUrlHandlerMapping kihagyásával.

5.6.1 A DispatcherServlet konfigurációja

```
<servlet>
	<servlet-name>Spring BlazeDS Dispatcher Servlet</servlet-name>
	<servlet-class>
		org.springframework.web.servlet.DispatcherServlet
	</servlet-class>
	<init-param>
		<param-name>contextConfigLocation</param-name>
		<param-value>/WEB-INF/applicationContext.xml</param-value>
	</init-param>
	<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
	<servlet-name>Spring BlazeDS Dispatcher Servlet</servlet-name>
	<url-pattern>/messagebroker/*</url-pattern>
</servlet-mapping>
```

A web.xml-ből egy részlet

A Spring DispatcherServlet-t használjuk a Flex klienstől jövő üzenetek továbbítására a szolgáltatások felé. Ezt a web.xml fájlban kell definiálnunk. Megadjuk neki az alkalmazásunk környezetének a beállításait, amit az applicationContext.xml tartalmaz. Ez a Springes alkalmazásunk lelke. Ez a szervlet hozza létre a Springes konténert, amiben az applicationContext.xml fájlban szereplő beanek lesznek kezelve.

A szervlethez hozzárendeljük a klienstől jövő összes /messagebroker/* szövegre illeszkedő URL-t. Ezek a Flex kliens által küldött üzenetek, hiszen a végpont webcíme is illeszkedik erre a mintára. A szervletet indításkor betöltjük az alkalmazásszerverünk web konténerébe, hogy egyből kezelni tudja a kéréseket. Ez a szervlet az üzenetet eljuttatja, minden, a HandlerAdapter interfészt implementáló osztálynak.

5.6.2 A MessageBrokerHandlerAdapter inicializálása

A MessageBrokerHandlerAdapter osztályt a Springes környezetdefiniáló XML fájlban kell példányosítani a következőképen:

```
<bean class="org.springframework.flex.servlet.MessageBrokerHandlerAdapter"/>
```

Feladata, hogy a SimpleUrlHandlerMapping segítségével, kiválassza, egy adott kérést melyik MessageBroker példányhoz kell eljuttatni.

5.6.3 A SimpleUrlHandlerMapping konfigurációja

Ez az osztály rendeli össze a kéréseket a MessageBrokerekkel. Konfigurálni a következőképpen kell:

```
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
	<property name="mappings">
		<value>/*=messageBroker</value>
	</property>
</bean>
```

A SimpleUrlHandlerMapping konfigurációja

Itt minden a DispatcherServlet-től jövő üzenetet, a következő részben példányosított messageBroker bean-hez juttatunk.

5.6.4 A MessageBroker konfigurációja

A MessageBrokert is az applicationContext.xml fájlban definiálunk, abban, amit a szervletnek is megadtunk paraméternek.

```
	<bean id="messageBroker"
		class="org.springframework.flex.core.MessageBrokerFactoryBean" >
		<property name="servicesConfigPath"
```

```
                value="WEB-INF/flex/services-config.xml" />
    </bean>
```

A MessageBroker konfigurációja

Létrehozunk egy Beant, ami a MessageBrokerFactoryBean osztály, ez fogja példányosítani a MessageBroker osztályt. Ezzel könnyen tudunk Springes szolgáltatásokat elérhetővé tenni a Flex kliensek számára. Paraméterként meg kell adni a BlazeDS szolgáltatások konfigurációjának az XML állományát.

5.6.5 Spring szolgáltatás BlazeDS szolgáltatásként

Ezt nagyon egyszerűen véghez tudjuk vinni. Nem kell mást tennünk, mint hogy az applcationContext.xml állományba felvesszük a következő sort:

```
<flex:remoting-destination ref="reserverService" message-broker="messageBroker" />
```

Ezzel a hivatkozott Springes beanünk (amiről a későbbiekben szó lesz) elérhetővé válik a Flex kliensek számára. A message-broker attribútummal tudjuk megadni, hogy mely MessageBroker-t akarjuk használni a szolgáltatásnál.

6 Spring

A Spring egy nyílt forrású keretrendszer. A Spring segítségével egyszerű POJO (plain-old Java object) használatával tudjuk az üzleti logikánkat megvalósítani, tehát egy alternatívát nyújt az EJB-vel (Enterprise JavaBeans) szemben. A Spring egy könnyűsúlyú függőség injektáló és aspektusorientált konténer és keretrendszer.

A könnyűsúly a méretére és a használhatóságára utal. A spring.jar mérete kb. 2,8 MB, igaz ebben nincs benne a Spring MVC (spring-webmvc.jar), ami tartalmazza például a DispatcherServlet-et is. A könnyűsúlyú használat arra utal, hogy nem kellenek más gyártók jar állományai, ha használni akarjuk a Springet (ha teljes mértékben ki akarjuk használni, akkor viszont kellenek).

A Spring biztosít egy lazán csatoló technikát, amit függőség injektálásnak (Dependency Injection) nevezünk. Ha a DI-t használjuk, akkor az adott objektumunk független a függőségeiktől, tehát nem az objektum állítja be vagy példányosítja magának a függő objektumokat, hanem a Spring konténer hajtja végre mindezeket, mikor kikérjük az objektumot a konténerből (vagyis mikor példányosítani akarjuk).

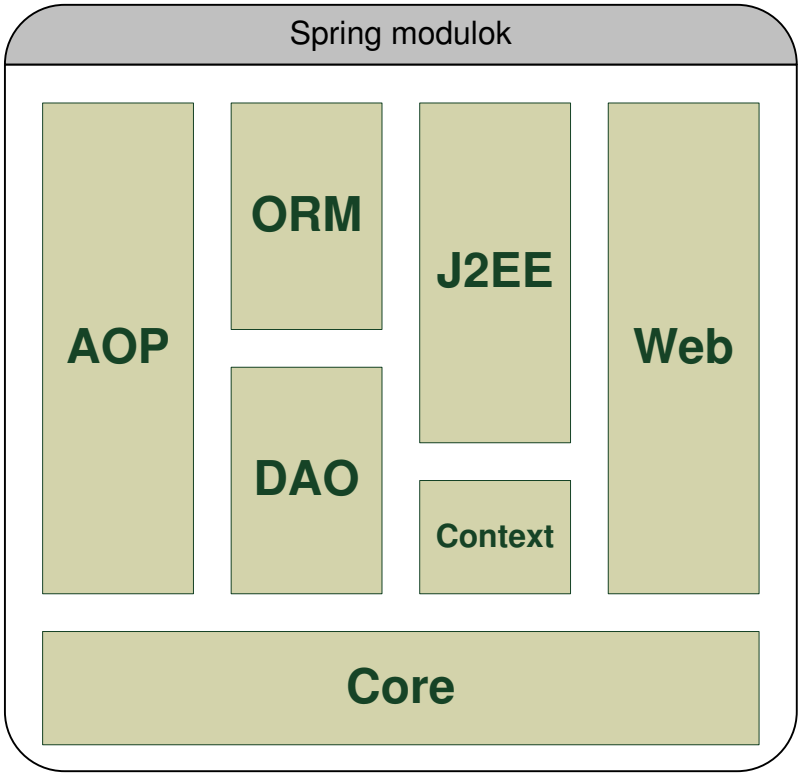
A Spring aspektusorientált programozás (AOP) támogatása elég gazdag. Az AOP segítségével el tudjuk különíteni az üzleti logikánkat a rendszerszolgáltatásoktól. Ilyen szolgáltatások a naplózás vagy a tranzakció-kezelés. Tehát az adott osztályunk csak az üzleti logikánkat tartalmazza, minden mást az AOP tesz köré.

A Spring konténer menedzseli az alkalmazásunk objektumainak az életciklusát és a konfigurálását. Megadhatjuk minden egyes objektumra nézve, hogy hogyan jöjjenek létre, hogyan legyenek beállítva és hogyan kapcsolódjanak egymáshoz.

A Spring keretrendszer lehetővé teszi, hogy egy összetett alkalmazást kisebb komponensekből állítsuk össze. A Springben az alkalmazás objektumait deklaratívan állítjuk össze, tipikusan egy XML fájlban. A Spring keretrendszer sok funkcionalitást biztosít, hogy nekünk csak az alkalmazás üzleti logikájának megírásával kelljen törődni. Ilyen funkcionalitások például a tranzakció és biztonság menedzselése vagy a valamilyen ORM keretrendszer integrálásának a támogatása.

Tehát a legfőbb ok, hogy Springet használjunk az alkalmazásainkban, hogy lazán csatolt kódokat tudjunk írni. Ez segít a tesztelések és a további fejlesztések során. Növeli a kód karbantarthatóságot.

6.1 Spring modulok vázlatosan



9. ábra: Spring modulok vázlatosan

Néhány szót a modulokról:

- Core: A Spring keretrendszer legalapvetőbb része. Ez biztosítja a DI tulajdonságot. Ez szinte minden modulnak az alapja.
- AOP: A Spring az AOP „szövetségnek" megfelelő aspektusorientált programozás implementációt biztosít, ezzel is növelve a kódunk lazán csatoltságát.
- DAO: Ez a modul a JDBC absztrakciója. Segítségével a kódunkból eltávolíthatjuk a feleslegesen ismétlődő sorokat. Ilyenek a JDBC kapcsolat kikérés, a gyártóspecifikus hibák kezelése, az eredményhalmaz feldolgozása, a lefoglalt erőforrások felszabadítása, stb.
- ORM: Ez a modul segít integrálni a népszerű ORM keretrendszereket. Ilyen keretrendszerek például a Hibernate, a JPA és az iBatis.
- Context: Ez a modul a Core modult egészíti ki olyanokkal, mint az I18N támogatás, az alkalmazás életciklus események terjesztése és az erőforrások betöltése.
- J2EE: Ez a rész támogatja a JNDI elérést, az EJB integrációt, az e-mail küldést, és sok más Java Enterprise szolgáltatást.

- Web: Ez a modul biztosítja a Web integráció lehetőségét. Ilyenek például a több részből álló fájl feltöltése, a konténer inicializálását szervleten keresztül és a JSF, WebWork és Structs keretrendszerek integrálását. E modul MVC része egy Model-View-Controller implementációt is biztosít a webes alkalmazáskészítés számára.

Ezen modulok használatára a későbbiekben láthatunk majd példákat is. A Web modult, pedig csak, arra használjuk, hogy a konténerünket inicializáljuk, és a klienstől jövő kéréseket elirányítsuk a megfelelő MessageBroker fellé, mint ahogy azt az előző fejezetben láthattuk.

6.2 A Core modul

A Spring konténerben beanek vannak. A bean nem más, mint egy objektum, amit példányosítva lett és a Spring konténer kezel. Tehát nincs semmi különleges egy beanben. Ezek a beanek, és a köztük lévő függőségeket a konténer, a konfigurációs meta-adat felhasználásával állapítja meg. A Spring keretrendszer BeanFactory interfész reprezentálja az egyszerűbb konténert. Ez a következő feladatokat látja el: példányosítja vagy erőforrással látja el az alkalmazás objektumokat, konfigurál ilyen objektumokat és összeállítja a függőségeket ezen objektumok között. Sok implementációja létezik ennek az interfésznek, de a legtöbbet használt az XMLBeanFactory osztály. Ez XML konfigurációs meta-adatokat használ, hogy a rendszerünket teljesen felkonfigurálja.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

  <bean id="..." class="...">
  </bean>

  <bean id="..."
      factory-bean="..."
      factory-method="..."/>
   .
   .
   .
</beans>
```

Minimális bean konfiguráció

A legfelső szinten a <beans> elem áll. Ezt tartalmazza a felhasználni kívánt névtereknek a definícióját. A Spring konfigurációs állományának legalább egy bean definíciót tartalmaznia kell. A beaneknek nem fontos tartalmaznia ID-t, ekkor a konténer generál neki egy egyedi azonosítót. Akkor nem kell például ID, ha az adott beant nem akarjuk máshol hivatkozni, csak példányosítani szeretnénk. A beaneknek lennie kell egy class attribútumának, ami a

példányosítani kívánt osztály nevét tartalmazza a teljes csomagnévvel együtt. Vagy lennie kell egy factory-bean és egy factory-method attribútumának. Az factory-bean egy másik bean ID-ját tartalmazza, amin a megadott metódus fog meghívódni, hogy létrehozza az a definiált beant.

A függőségeket kétféleképpen injektálhatjuk be egy adott beanbe:

- Konstruktoron keresztül. Ezt a <constructor-arg> elem használatával érhetjük el, a bean definíció belsejében. Elsősorban az argumentum típusa dönti el, hogy melyik argumentum melyik paraméterhez kötődik. Ha egyszerű típust használunk, akkor a Spring nem tudja eldönteni, hogy mit-mihez kell kötni. Ilyenkor segíteni kell neki, hogy megadjuk az argumentum típusát, vagy pedig indexeket használunk.
- Setter metóduson keresztül. Ezt a <property> tag használatával érhetjük el. Úgy történik, hogy elsőnek az adott osztály paraméter nélküli konstruktora, majd a megadott setter metódusa hívódik meg. Általában ezt szokták használni, mivel ha sok függőségeket kell beinjektálnunk az adott osztályba, ráadásul úgy, hogy null értékűek is lehetnek, akkor sokféle paraméterű konstruktort kellene létrehozni.

Másik beanre hivatkozni a <ref> elemmel lehet. Ha a beanünk ugyanabban az XML fájlban van, mint ahol hivatkozzuk, akkor megadatjuk a bean attribútum helyett a locale attribútummal is a hivatkozni kívánt beant. Ilyenkor az XML ellenőrző ugyanabban a fájlban keresi a beant. Ez azért jó, mert hamarabb ki lehet deríteni az elírásokat.

A depends-on tulajdonsággal, megmondhatjuk, hogy mely beanek legyenek előbb inicializálva. Ha többet adunk meg, akkor vesszővel, szóközzel vagy pontosvesszővel választhatjuk el.

A lazy-init attribútummal megadhatjuk, hogy e tulajdonsággal rendelkező bean csak akkor legyen példányosítva, ha hivatkozás történik rá. Alapértelmezetten a konténerünk, minden szingleton beant már a konténer inicializálása után egyből példányosít. Ennek hátránya, hogy az esetleges hibák csak a beanre való hivatkozás után jönnek elő. Ha a beans attribútum default-lazy-init tulajdonságát igazra állítjuk, akkor mindem szingleton bean csak akkor jön létre, ha hivatkozás történik rá, kivéve ha ezt nem definiáljuk felül az egyes beanekben.

Lehetőségünk van a beaneket automatikusan is összekapcsolni, ezzel sorokat megspórolva az XML állományban. Ezt a bean autowire attribútumával tudjuk megadni. 5-féle mód közül választhatunk:

- nincs (no): Ilyenkor a <ref> elemmel kell hivatkozni a függő beanre.

- név szerint (byName): Az összekapcsolás attribútum név egyeztetéssel történik. Ha a beanünk tulajdonságának a neve megegyezik egy másik bean azonosítójával, akkor a konténer automatikusan beállítja neki azt.
- típus szerint (byType): a tulajdonság beállítódik típus szerint, ha pontosan egy ilyen típusú beanünk van a konténerben. Ha több ilyen típusú is van, akkor a konténer nem tudja, hogy melyiket használja, ezért egy kivétel váltódik ki. Ha nincs ilyen típusú, akkor nem történik semmi.
- konstruktor szerint (constructor): ez ugyanaz, amit a típus szerint, csak konstruktorra alkalmazva.
- automatikus (autodetect): a konstruktor vagy a típus szerintit választja a konténer, attól függően, hogy a bean, hogy van definiálva. Ha alapértelmezett konstruktort talál a konténer, akkor csakis a típus szerint fog eljárni.

Az explicit megadott tulajdonságokat az automatikus összekapcsolás mindig felülbírálja. A primitív típusokra nem működik ez a módszer. Ez egy lehetőség, de én ezt nem ajánlom, mert szerintem csak nehézségeket okoz a programkód nagyobbodásával.

Azt, hogy egy bean az alkalmazás futása során hányszor legyen példányosítva a bean scope tulajdonságával állíthatjuk be. A két lényeges értéke:

- singleton: Ilyenkor csak egy példány jön létre az adott beanből a konténerben, és ezt hivatkozza minden egyes bean. Ez az alapértelmezett beállítás.
- prototype: Ha ilyen beanre hivatkozunk, akkor minden egyes hivatkozáskor egy új példányt hoz létre a konténer.

A következő tulajdonságokat lehet még beállítani: request, session és global-session. Ezeket viszont csak akkor tudjuk használni, ha Springgel fejlesztjük az alkalmazásunk webes részét is, mivel ezek HTTP kérésekhez vagy munkamenetekhez vannak rendelve.

6.3 Az AOP modul

6.3.1 Az AOP-ról általában

Egy alkalmazásban sok tevékenység általános. Ilyenek a naplózás, a biztonság vagy a tranzakció kezelés. Ezek benne vannak az üzleti logika kódjában, melyek nehezítik az olvashatóságot és a karbantarthatóságot. Itt lép a színre az aspektus-orientált programozás (AOP). Az AOP segít elkülöníteni ezeket a funkcionalitásokat az üzleti logikánktól.

AOP kifejezések:

- Tanács (Advice): A tanács lényegében az aspektus által elvégzendő feladat. a Tanács az aspektusnak a következő részeit definiálja: mit és mikor. A mikor kérdések például: A tanácsot a metódushívás előtt alkalmazzuk? Vagy előtte és utána is? Vagy csak akkor, ha kivétel váltódott ki?
- Kapcsolódási pont (Joinpoint): Egy alkalmazásban rengeteg lehetőség van, hogy tanácsokat illesszünk be a végrehajtásába. Ezeket a lehetőségéket nevezzük kapcsolódási pontoknak. Ilyen pontok lehetnek egy metódus meghívása vagy egy kivétel dobódása vagy esetleg egy osztály mezőjének az értékének a megváltozása.
- Szűrő (Pointcut): Ha a tanács a mikor és a mit válaszokat mondja meg, akkor a szűrő azt válaszolja meg, hogy hol. Azért neveztem el szűrőnek, mert a kapcsolódási pontok közül válogatja ki a számunkra érdekeseket. Ennek segítségével tudjuk megmondani, hogy egy tanács hol szövődjön be a végrehajtási folyamat során. Ez lehet egy osztálynév, metódusnév vagy reguláris kifejezés, ami az osztálynevekre vagy a metódusnevekre illeszkedik.
- Aspektus (Aspect): Az aspektus a tanács és a szűrő összekapcsolva. Tehát egy aspektus tudja, hogy mikor, hol és mit kell csinálnia.
- Cél (Target): Az az objektum ahova a tanácsot be akarjuk szőni.
- Helyettes (Proxy): A tanács beillesztődik a cél objektumba. Ekkor létrejön egy új objektum, amit helyettesnek nevezünk. Ezt az AOP keretrendszer hozza létre. Ekkor ez hívódik a célobjektum helyett.
- Szövés (Weaving): Az a folyamat mikor a tanács beillesztésre kerül a célobjektumba és létrejön az új, helyettes objektum.

Különféle tanácsokat hozatunk létre, attól függően, hogy mikor akarjuk azt végrehajtani. Lássuk ezeket:

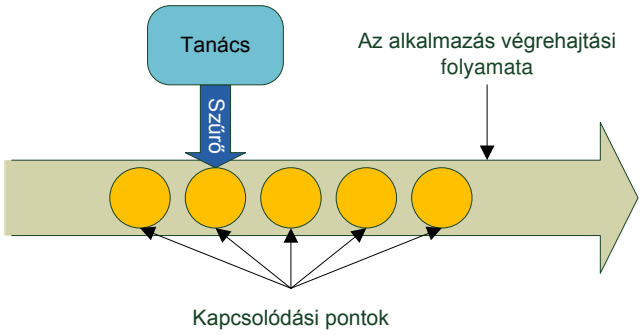
- Előtte (Before advice): Ezen típusú tanácsok a kapcsolódási pont lefutása előtt hajtódnak végre.
- Visszatérés után (After returning): E típusú tanácsok akkor fognak lefutni, ha a kapcsolódási pont normálisan befejezte a végrehajtását, azaz nem dobott kivételt.

- Kivételkor (After throwing): Ez a tanács, akkor hajtódik végre, ha a kapcsolódási pont kivételt dobott.
- Utána (After [finally] advice): Ez a tanács a kapcsolódási pont lefutása után mindig lefut, nem számít, hogy volt-e kivétel.
- Körül (Around advice): Ez a típusú tanács, az előtte és az utána tanács egybegyúrása.

Nem csak a tanácsoknak vannak típusai, hanem annak is, hogy az AOP keretrendszer mikor szövi be a tanácsot a cél objektumba. Ezen műveletnek a következő típusai vannak:

- Fordítási időben: mikor a célobjektum leforduld, akkor szövődik be az aspektus. Ehhez speciális fordító szükséges.
- Osztálybetöltődéskor: Az aspektus akkor kerül be a célobjektumba, mikor a JVM betölti azt. Ezen folyamathoz speciális osztálybetöltő szükséges.
- Futási időben: Az aspektus futási időben szövődik be a célobjektumba. A Spring AOP modulja is így működik.

6.3.2 Az AOP működése vázlatosan



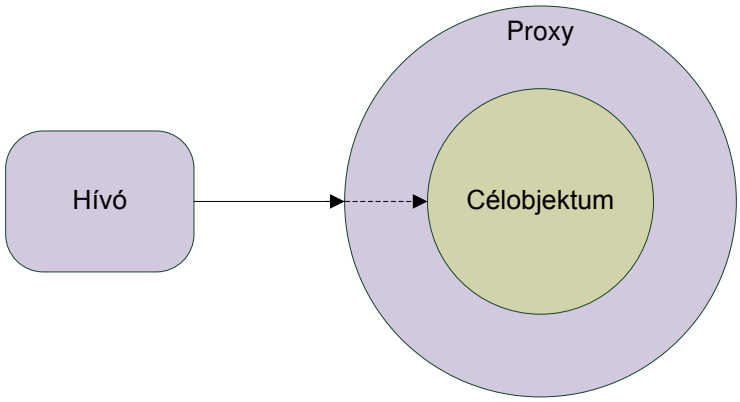
10. ábra: Az AOP vázlatosan

A programban nagyon sok kapcsolódási pont van. Az aspektus - ami lényegében a tanács és a szűrő együttvéve - megmondja, hogy mely kapcsolódási pontos esetén kell az adott tanácsot beszőni a program végrehajtása során. A beillesztés történhet futás időben, fordításkor vagy a cél osztály betöltődésekor. A kapcsolódási pontokhoz viszonyítva is többféle beillesztési időpontot választhatunk. A legáltalánosabb az, mikor a körül típusút választjuk. Ez viszont csak, akkor ajánlatos, ha pl. tranzakciót akarunk kezelni. A legmegfelelőbb tanácstípust választva egyszerűsödik a programozási modell, ami maga után vonja a kevesebb hibázási lehetőséget.

6.3.3 A Spring AOP képességei és céljai

A Spring AOP tiszta Java nyelven van implementálva. Nincs szükség arra, hogy az osztálybetöltőt kontrolláljuk, emiatt használhatjuk alkalmazásszerverekkel együtt is. A Spring AOP csak a metódus futását tudja csatlakozási pontként biztosítani. Nincs lehetőség például egy osztály mezőjének a változását felhasználni csatlakozási pontként. A Spring AOP nem egy teljes AOP implementáció, inkább közelebbi integrációt biztosít az AOP és a Spring konténer között, hogy könnyen tudjuk általános problémákat megoldani az alkalmazásunkban. Ha egy teljesen implementált aspektus-orientált programozási nyelvre van szükségünk, mint például az AspectJ, akkor a Spring ennek az integrációjára is biztosít lehetőséget. A Spring AOP alapértelmezetten a JavaSE dinamikus proxyját használja AOP proxyként. Ilyenkor a java.lang.reflect.Proxy osztályt hívja segítségül. Ez lehetővé teszi a Spring számára, hogy dinamikusan generáljon új osztályt, ami implementálja a szükséges interfészeket, beszövik a tanácsot és bármely hívást ezen interfészeken elirányít a cél osztályhoz. Ezt akkor alkalmazza a Spring, ha az a metódus, amit tanáccsal szeretnénk ellátni, az egy interfész által implementált metódus. A Spring AOP a CGLIB proxyt is ismeri. Ezt akkor használja a Spring, ha az üzleti objektumunk nem implementál egyetlen interfészt sem, vagy olyan metódust akarunk tanáccsal ellátni, ami nem interfész által implementált. Fontos megérteni azt a tényt, hogy a Spring AOP proxy alapú.

6.3.4 A Spring AOP proxy működése



11. ábra: A Spring AOP proxy működési elve

A Spring AOP-ban az aspektusok futásidőben szövődnek be a célobjektumba, amely be lesz csomagolva egy proxy objektumba. A proxy objektum azt színleli, hogy a célobjektum. Elkapja a tanáccsal ellátott metódushívásokat, és továbbítja azt a célobjektumnak. Az

elkapás és a továbbítás között pedig végrehajtja a megfelelő tanácsot. Ha a metódus sikeresen lefut vagy kivételt dob, akkor ugyancsak a proxy objektumon keresztül kerül vissza a vezérlés a hívóhoz, így ilyenkor is végbemennek az ide illő tanácsok. A Spring nem generálja ki addig a proxyt, amíg nincs rá szükség.

6.3.5 Autoproxy

Az autoproxy azt jelenti, ha a Spring egy beanről megállapítja, hogy egy vagy több aspektussal fel van vértezve, akkor automatikusan létrehozza köré a proxy osztályt, ami elkapja az adott metódushívást, és biztosítja, hogy a tanács lefusson, amikor szükséges. Az autoproxy engedélyezéséhez nem kell mást tennünk, mint a Springes DefaultAdvisorAutoProxyCreator osztályt példányosítani a következőképpen:

```
<bean
class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"
/>
```

Az automatikus proxy engedélyezése

6.3.6 Annotáció vagy XML

A Spring kétfele lehetőséget biztosít, hogy az AOP-ot használni tudjuk. Az egyik lehetőség, hogy @AspectJ annotációkat használunk, a másik, hogy maradunk a jó öreg XML állománynál. Ha esetleg nem Java 5 vagy jobbat használnánk, akkor egyértelmű, hogy az XML-t kell használnunk. Akkor is az XML-t kell választanunk, ha az AOP-ot csak eszközként használjuk, hogy az üzleti szolgáltatásokat konfiguráljuk. Az XML stílussal tisztábban láthatjuk, hogy az alkalmazásunkban mennyi és milyen aspektusok vannak. Az XML-nek két hátránya van. Az első az, hogy két helyen kell az aspektusokat konfigurálnunk (az XML fájlban és a tanácsosztályban). A másik, hogy az XML fájlban nem tudjuk kihasználni azt, amit az @AspectJ annotációval (pl. az aspektusok kombinálása egymással).

6.4 A Spring tranzakció kezelése

6.4.1 Fogalmak

A tranzakció nagyon fontos a valós világ alkalmazásaiban. Biztosítja, hogy az adatok és az erőforrások ne kerüljenek inkonzisztens állapotba. A tranzakció néhány művelet egy egységbe fogását jelenti, ami ha nem történt semmi hiba, akkor végbemennek. Ha viszont csak egy hiba is volt, akkor olyan mintha meg sem történt volna. Tehát a tranzakció a

mindent vagy semmit elvet vallja. A tranzakció jellemezésére az ACID (magyarul KATI) mozaikszót szokták megemlíteni. Nézzük, mint is jelentenek a betűk:

- Atomicitás (Atomic): Ez azt jelenti, hogy a tranzakció vagy sikeres, és akkor minden benne lévő művelet végrehajtódik, vagy sikertelen és egyetlen utasítás sem lesz végrehajtva. Tehát a tranzakcióban szereplő utasítások először csak feltételesen hajtódnak végre, és ha az utolsó is sikeresen lefutott, akkor véglegesítődnek.
- Konzisztencia (Consistent): sikertelen és sikeres tranzakció esetén a konzisztenciának meg kell maradni. A tranzakció végrehajtása folyamán ideiglenesen felléphet az inkonzisztens állapot, de mire véget ér a tranzakció, addigra helyre kell állnia.
- Izoláció (Isolated): A konkurens rendszerben, azaz amikor több tranzakció időben egymást átfedve hajtódnak végre, egy tranzakció nem láthatja egy másik tranzakció kezdeményezett, de még be nem fejezett műveleteinek a hatásait.
- Tartósság (Durable): Egy sikeresen lezárt tranzakció által végrehajtott módosítások hálózati hiba vagy hardver meghibásodás esetén sem veszhet el.

Az izoláció és a konzisztencia egymást gyengítő dolgok. Ha az adatbázis-kezelő szekvenciálisan hajtja végre a konkurens tranzakciókat, akkor a programunk teljesítménye jelentősen csökken sok konkurens tranzakció esetén. Viszont a konzisztencia biztosan megmarad. Ha viszont valahogyan párhuzamosan próbálja végrehajtani a tranzakciókat, akkor nő a programunk teljesítménye, cserébe többet kell dolgoznia a programozónak és könnyebben kerülhet inkonzisztens állapotba az adatbázis. Három anomáliát különböztetünk meg:

- Piszkos olvasás (Dirty Read): Az a jelenség, mikor egy tranzakció olyan adatot olvasott ki az adatbázisból, amelyet egy másik tranzakció módosított, de még nem véglegesített.
- Megismételhetetlen olvasás (Nonrepeatable Read): Ez akkor lép fel, ha egy tranzakció kiolvas egy adatot az adatbázisból, majd újra ki akarja olvasni, de közben egy másik tranzakció megváltoztatja azt (véglegesítődik a tranzakció). Ekkor a tranzakció nem ugyanazt az eredményt kapja, mint az első olvasáskor.
- Fantomsor olvasás (Phantom Read): Ez akkor történik meg, ha egy tranzakción belül kétszer is végrehajtja ugyanazt a lekérdezést, de a második végrehajtásnál újabb sorok is bekerülnek az eredménybe. Ez akkor lehetséges, ha egy másik tranzakció

véglegesítődik, és az ez által létrehozott sorok megfelelnek az szóban forgó tranzakció keresési feltételének.

E három anomáliát négy izolációs szinttel tudjuk befolyásolni, amit a „Spring a gyakorlatban” részben mutatok be.

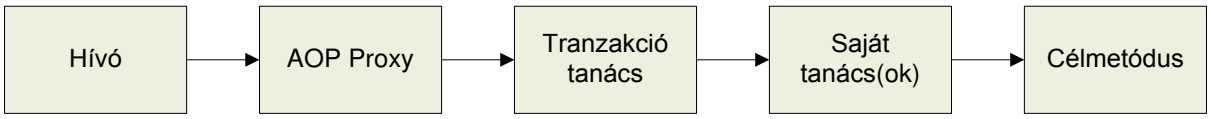
6.4.2 Spring vs. EBJ CMT

A Spring tranzakció kezelése lehet deklaratív és programozott. Csak a deklaratívat mutatnám be, hiszen a legtöbb esetben elég ezt használni. A programozottra csak nagyon speciális esetben lehet szükség. Ilyenkor mi kezeljük a tranzakció indítását, véglegesítését vagy visszagörgetését a kódban. A deklaratív tranzakció kezelés hasonló, mint az EJB (Enterprise Java Bean) konténermenedzselt tranzakció kezelése (Container Managed Transaction), röviden EJB CMT.

Nézzük a különbségeket az EJB CMT-hez képest:

- Az EJB CMT JTA-t (Java Transaction API) használ, míg a Spring keretrendszer deklaratív tranzakció kezelése bármely környezetben használható úgy, mint JDBC vagy Hibernate.
- A Spring segítségével bármely osztályban használhatunk deklaratív tranzakció kezelést, nem csak EJB-ben.
- Az EJB környezettel ellentétben, a Spring biztosítja a setRollbackOnly() metódus meghívását deklaratív tranzakció kezelés esetén is. Ezzel megjelöljük a tranzakciót, hogy csakis visszagörgetésre kerülhet.
- Az AOP segítségével még a tranzakció viselkedését is befolyásolhatjuk. Ha például pluszban el kell végezni valamit egy tranzakció során, akkor az AOP tanácsokkal könnyedén megtehetjük ezt.
- A Spring viszont nem támogatja a tranzakciók továbbterjedését távoli hívásokon keresztül, azaz az elosztott tranzakció kezelést. Ha erre lenne szükségünk, akkor EJB-t kell használnunk.
- A Spring segítségével nem csak ellenőrzött kivételek esetén lesz automatikus tranzakció visszagörgetés, hanem a nem ellenőrzött kivételek esetén is, ha azt beállítjuk. Sőt azt is beállíthatjuk, hogy mely ellenőrzött kivétel esetén nem akarunk visszagörgetést.

6.4.3 A Spring tranzakció működési váza

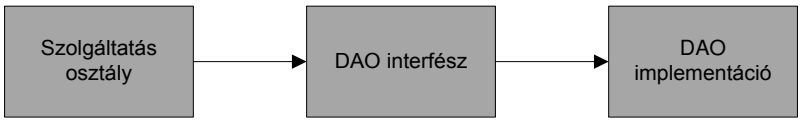


12. ábra: A Spring tranzakció működésének a vázlata

Mint ahogy láthatjuk a hívó nem közvetlenül a célmetódust hívja meg, hanem a metódus köré az AOP által kialakított proxy osztály metódusát. A hívási lánc a Tranzakció tanáccsal folytatódik. Ez a Spring saját tranzakció kezelő tanácsa. A híváskor létrejön a tranzakció, a visszatéréskor pedig az eredménytől függően vagy elfogadásra kerül vagy visszagörgetődik a tranzakció. Csak ez a két eset lehetséges. Saját tanácsokat is belerakhatunk a tranzakció kezelésébe. Mind a célmetódus hívás előtti és utáni tanács megengedett.

6.5 Spring adatelérés elmélete

A DAO (Data Access Object) felelős, hogy adatot az adatbázisba írjunk, és az adatbázisból olvassunk. Ezt a funkcionalitást interfészen keresztül teszi elérhetővé. Ezen interfészt használjuk a szolgáltatás osztályunkban az adatok elérésére és tárolására. A szolgáltatás is rendelkezhet interfész implementációval, mivel ez a kulcsa, hogy lazán csatolt kódokat írjunk.



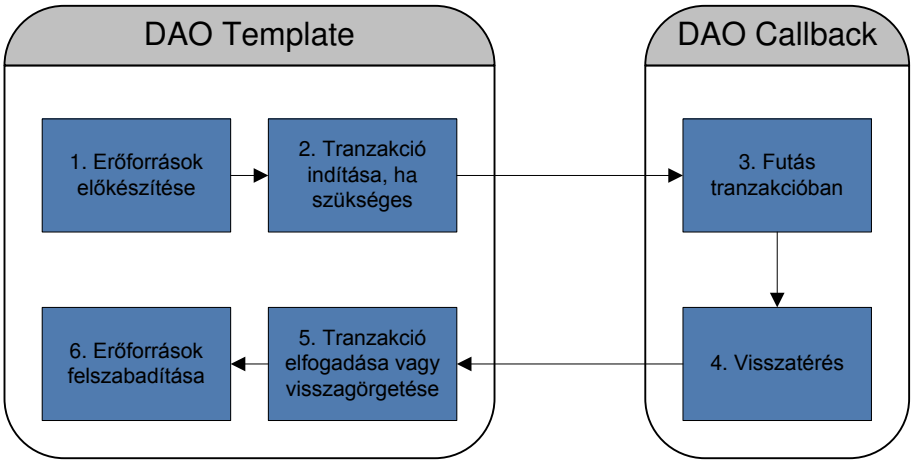
13. ábra: A DAO szerkezete

Az interfész használatából két előnyünk származik. Az egyik az, hogy a szolgáltatás osztályt könnyen tudjuk tesztelni, hiszen nem függ az adott adatelérés implementációtól. Például a DAO implementációt a tesztelési folyamat idejéig nem csatlakoztatjuk az adatbázishoz, így felgyorsíthatjuk az egységteszteket. A másik az, hogy csak a DAO implementáció függ a választott ORM keretrendszertől, így a DAO átírásával az egész alkalmazásunk alatt le tudjuk cserélni ezt a keretrendszert.

Ha már adatelérésről esett szó, akkor itt kell a kivételkezelést is megemlíteni. A JDBC mondhatni, csak egyfajta kivételt dob, ráadásul az ellenőrzött is. Ez a kivétel az SQLException. Így rá vagyunk kényszerülve a kivételek elkapására, a try-catch blokkok írására. A Hibernate esetén jobb a helyzet, mivel kb. két tucat különböző kivételt dob, így

eldönthetjük, hogy melyeket akarjuk elkapni. Ez azonban gyártó specifikus. A Spring ezért a kivételeket becsomagolja, hogy a megírt kódunk ne függjön az adatelérés biztosító keretrendszertől. A Spring adatbázis kivételek ősosztálya a DataAccessException. Ez egy nem ellenőrzött kivétel, tehát nem kötelező elkapni. A Spring a nem ellenőrzött kivételek híve, tehát a programozóra bízza, hogy mely kivételeket akarja megfogni, és melyeket elengedni.

Ha adatbázissal akarunk dolgozni, akkor mindig kell egy adatbázis kapcsolat, hogy elérjük a szükséges adatot, és ha végeztünk, akkor mindig fel kell szabadítani a lefoglalt erőforrásokat. Ezek az állandó részei az adatelérésnek. Azonban minden egyes adatelérésnél, más és más műveleteket hajtunk végre. Például, nem csak egy táblából kérdezünk le, egy frissítést is különböző módon csinálunk minden egyes objektumon. Ezt nevezzük az adatelérés változó részeinek. A Spring az adatelérés e két részét két osztályba különíti el. Az állandó részeket a Template osztályokba, a változó részeket – amiben a saját adatelérési kódunk van kezelve - a Callback osztályokba.

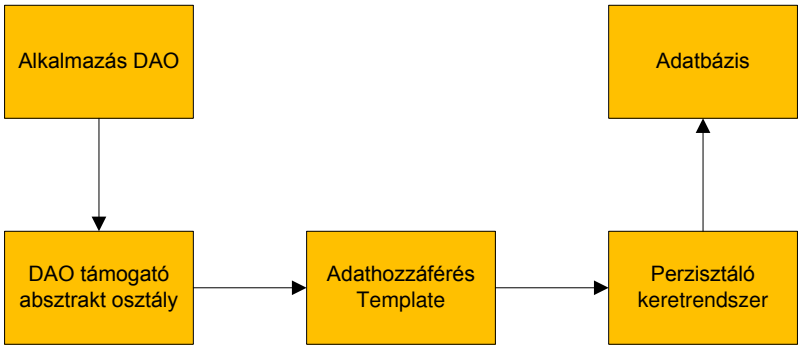


14. ábra: A DAO Template és Callback

Mint ahogy azt az ábrán is látjuk a Template vezérli a tranzakciót, menedzseli az erőforrásokat és kezeli a kivételeket. A Callback létrehozza az SQL utasítást, beköti a paramétereket és feldolgozza az eredményhalmazt.

A Spring mindegyik támogatott perzisztens platformhoz biztosít DAO Template-et. Ezeket kétféleképpen tudjuk használni. Az egyik lehetőség, hogy példányosítjuk egy Spring beanként, és utána a DAO osztályunkba bekötjük a konfigurációs fájlon keresztül. A másik, és ez a jobbik megoldás szerintem, hogy a mi DAO osztályunk kiterjeszti a választott DAO Template támogató osztályát.

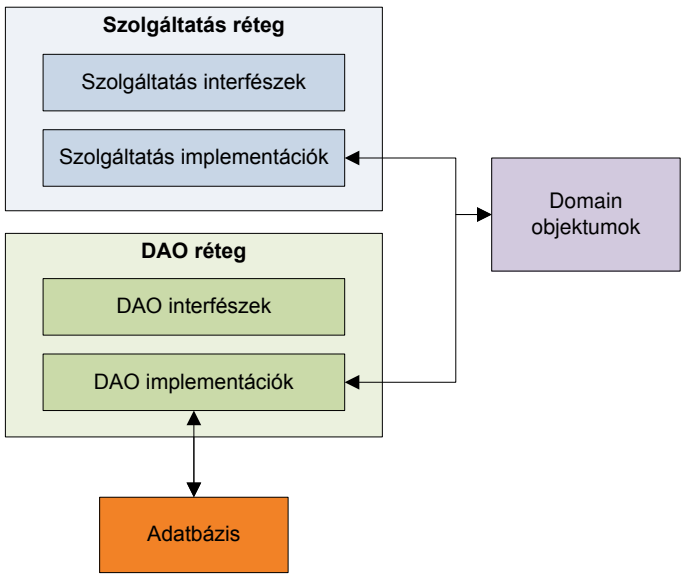
A Spring adathozzáférés keretrendszere nem csak a Template-ekből áll. Mindegyik Template kényelmesen használható metódust biztosít, ami úgy egyszerűsíti az adathozzáférést, hogy nem kell saját Callback implementációt létrehozni, hanem a Spring biztosít számunkra egyet.



15. ábra: Kapcsolat a DAO és az adatbázis között

6.5.1 Spring szolgáltatások

Minden egyes bean biztosítja a gyakran szükséges funkcionalitások egy darabját. A szolgáltatás segítségével össze tudjuk fogni ezen funkcionalitásokat. A Spring azt tanácsolja, hogy az interfészt és az implementációt különítsük el egymástól. Minden szolgáltatás a Spring konténerében kerül példányosításra, hiszen be kell injektálni a megfelelő DAO-kat, hogy használni tudjuk. Nézzük, hogyan is kell elképzelni a szolgáltatások, a DAO-k és a domain objektumok kapcsolaltát.



16. ábra: A Spring szolgáltatások és DAO-k kapcsolata

Az ilyen szolgáltatások lesznek elérhetők a kliens oldalon a Flex által.

7 Spring a gyakorlatban

A Spring lelke az alkalmazás környezetét beállító XML állomány. A következőkben ezt szeretném bemutatni példán keresztül.

Először meg kell mondani, hogy mely Springes modulokat vagy egyéb segítséget nyújtó szolgáltatásokat szeretnénk használni a fájlban. Ezt a fájl elején lévő <beans> tagban kell megmondani egy XML névtér formájában. Ha például az AOP modult szeretnénk konfigurálni, akkor a következő névteret kell megadni:

```
xmlns:aop="http://www.springframework.org/schema/aop"
```

Ha Eclipse-szel fejlesztünk, és használjuk a Spring IDE-t, akkor hasznos lehet az XML séma helyének a megadása. Ezt ugyancsak a <beans> tagon belül az xsi:schemaLocation attribútumba fel kell venni a következő sort:

```
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
```

7.1 Az adatforrás beállítása

Kétféle adatforrást mutatok be. Az első egy egyszerű, mindig új kapcsolatot létrehozó adatforrás definíció, amit a Spring kínál számunkra. Ennek az osztálynak a neve DriverManagerDataSource. Használata a következő:

```
<bean id="dataSource"
            class="org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver" />
      <property name="url"
      value="jdbc:mysql://localhost:3306/housing_reservation?autoReconnect=true" />
      <property name="username" value="housing" />
      <property name="password" value="housingpass" />
</bean>
```

Egyszerű adatforrás

Először is példányosítjuk a DriverManagerDataSource osztályt, majd setter metódusok segítségével injektáljuk be a tulajdonságokat. A driverClassName tulajdonsággal kell megadni a JDBC meghajtónk osztályát. Az url tulajdonság mondja meg, hogy hol található meg az adatbázisunk. A username és a password az adatbázisba való belépés felhasználói nevét és jelszavát definiálja. Még egyéb tulajdonságokat is meg lehet adni, de most ezeket nem részletezném.

Ez az adatforrás típus nagyon jó teszteléshez, mivel könnyen tudunk csatlakozni az adatbázishoz, hogy a DOA implementációnkon egy egységtesztet futtassunk le. Hátránya

viszont, hogy mivel minden egyes adatbázis hozzáféréskor új kapcsolatot hoz létre, nagyban lelassíthatja a rendszer működését.

A másik adatforrás, amit be szeretnék mutatni az a JNDI adatforrás. Az elkészített alkalmazásunk úgy is egy alkalmazásszerverbe lesz telepítve, ezért ajánlott a JNDI adatforrás használata. Ehhez az alkalmazásszerverben be kell állítani egy adatforrást. Minden egyes szerverben más és más a konfigurálás. Most csak a Tomcat web konténer általi beállítást mutatnám be. Ehhez nem kell mást tennünk, mint a Tomcat könyvtárában lévő conf/context.xml fájlba felvenni a következő sorokat:

```
<Resource name="jdbc/HousingDB" auth="Container" type="javax.sql.DataSource"
      username="housing" password="housingpass"
      driverClassName="com.mysql.jdbc.Driver"
      url="jdbc:mysql://localhost:3306/housing_reservation?autoReconnect=true"/>
```

A Tomcat adatforrás beállítása

Mint ahogy látjuk, egy erőforrást kell definiálnunk, aminek meg kell adni a nevét és a típusát. A többi paraméter megegyezik az előbb leírtakkal.

Ennek az adatforrásnak az előnye, hogy nem függ az adatforrástól, hiszen az alkalmazásunk a nevével fog hivatkozni rá. A másik nagy előnye, hogy az alkalmazásszerver által kezelt adatforrás nagyobb teljesítményt biztosít a számunkra.

A <jee:jndi-lookup> elem segítségével objektumokat kérhetünk ki a JNDI-ből valahogy így:

```
<jee:jndi-lookup id="dataSource"
      jndi-name="/jdbc/HousingDB"
      resource-ref="true" />
```

JNDI adatforrás

A jndi-name attribútummal hivatkozhatunk az előbb konfigurált adatforrásra. Ha a resource-ref értéke igaz, akkor a JNDI név eleje a „java:comp/env” karakterlánccal egészül ki. Erre azért van szükség, hogy tudassunk az alkalmazásszerverünkkel, hogy egy Java erőforrást szeretnénk kikérni a JNDI könyvtárából. Tehát az aktuális név, amit használni fog a Spring a java:comp/env/jdbc/HousingDB. Persze, hogy használni tudjuk ezt, a jee névteret fel kell vennünk az XML állomány elején.

7.2 Hibernate Template használata

A Spring ORM keretrendszer támogatásán kívül a következő szolgáltatásokat is nyújtja:

- A Spring deklaratív tranzakciójának integrált támogatása
- Átlátható kivételkezelés
- Szál-biztos egyszerű sablonosztályok

- DAO támogató osztályok
- Erőforrás menedzselés

Az org.hibernate.Session a fő interfész ahol kapcsolatba léphetünk a Hibernate keretrendszerrel. Ez az interfész biztosítja az alap adathozzáférés funkcionalitását, azaz a mentés, frissítés, törlés és objektum betöltése az adatbázisból. Az alkalmazásunk DAO-ja a Hibernate Session interfészen keresztül biztosít mindent, ami a perzisztáláshoz szükséges. Referenciát szerezhetünk erre a Session objektumra a Hibernate SessionFactory interfész implementációján keresztül. A SessionFactory egyebek közt felelős ezen Session objektumok nyitásáért, zárásáért és a menedzseléséért. Tehát a SessionFactory egyszerűsíti a Sessionok nyitását és zárását, valamint a Hibernate-specifikus kivételeket átalakítja Springes ORM kivételekké.

7.2.1 A SessionFactoryBean beállítása

Mielőtt létrehoznánk egy sessionFactory objektumot tudnunk kell, hogy annotációt vagy XML fájlt alkalmazunk a Hibernate használatához, mivel különböző Springes osztályok tartoznak hozzájuk. Én az annotációval való konfigurálást választottam.

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
	<property name="dataSource" ref="dataSource" />
	<property name="annotatedClasses">
		<list>
			<value>hu.housing.domain.Reserver</value>
			<value>hu.housing.domain.Phone</value>
		</list>
	</property>
	<property name="hibernateProperties">
		<props>
		<prop key="hibernate.dialect">${hibernate.dialect}</prop>
		<prop key="hibernate.show_sql">${hibernate.show_sql}</prop>
		<prop key="hibernate.format_sql">${hibernate.format_sql}</prop>
		<prop key="hibernate.hbm2ddl.auto">${hibernate.hbm2ddl.auto}</prop>
		</props>
	</property>
</bean>
```

A sessionFactory bean és beállításai

Amint látható a sessionFactory objektumnak be kell injektálni az előbb létrehozott dataSource példányt. Az adatforrásnak implementálnia kell a javax.sql.DataSource interfészt. Mivel Hibernate annotációkat használunk, ezért a sessionFactory objektumnak be kell állítani az annotatedClasses tulajdonságát. Ez egy lista, amiben felsoroljuk a Hibernate annotációkkal ellátott osztályokat. A hibernateProperties tulajdonság segítségével Hibernate Session beállításokat tudunk érvényesíteni. A dialect tulajdonság megadása kötelező, hiszen

ekkor a Hibernate nem tudná, hogy milyen nyelven kommunikáljon az adatbázissal. A ${} helykitöltő változókat jelölnek.

7.2.2 A helykitöltők használata

A helykitöltők használatát azért vezette be a Spring, hogy az alkalmazás telepítéséhez kapcsolódó dolgokat külön tároljuk az alkalmazás környezetét beállító XML-től. Ilyen például az adatbázis specifikus dolgok, mivel teszteléshez és majd az éles futáshoz úgyis külön adatbázist kell használnunk, így csak e fájlban kell módosítani. Erre találta ki a Spring a PropertyPlaceholderConfigurer osztályt. A Hibernate Session beállításait is tárolhatjuk property fájlban.

```
<bean id="propertyPlaceholder"
    class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
       <property name="location" value="WEB-INF/hibernate.properties"/>
</bean>
```

A propertyPlaceholder konfigurálása

```
hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
hibernate.show_sql=true
hibernate.format_sql=true
hibernate.hbm2ddl.auto=create
```

A hibernate.properties tartalma

A PropertyPlaceholderConfigurer osztálynak meg kell mondani, hogy hol találja meg a konfigurációs fájlt. Erre a location tulajdonságot használjuk. Ha több fájl is akarunk használni, akkor nem ezt kell használni, hanem a locations tulajdonságot és ahol egy listában felsoroljuk a fájlokat. Mikor a Spring konténer létrehozza a beaneket, akkor, ha talál egy helykitöltő változót, megvizsgálja, hogy a megadott fájlban szerepel-e az adott nevű helykitöltő. Ha szerepel, akkor helyettesíti a megtalált tulajdonság értékével.

A dialect egy osztályt definiál, ami meghatározza, hogy milyen nyelven kell kommunikálni az adatbázissal. Ha a show_sql értéke igaz, akkor a Hibernate a konzolra írja minden SQL utasítását. Ez nagyon jól jön a fejlesztési fázisban, vagy ha még nem ismerjük annyira a Hibernate-et, akkor láthatjuk, hogy mit is csinál a háttérben. Ha a format_sql értéke igaz, akkor a Hibernate megformázza az SQL utasításokat. Ha a hbm2ddl.auto értéke create, akkor a Hibernate létrehoz mindent (lefuttatja az annotált osztályhoz létrehozott DDL utasításokat), tehát törli a meglévő táblákat és létrehozza újra üresen. Ez is csak akkor jó, ha a fejlesztési fázis elején tartunk, és sokat változnak az alkalmazásunk entitás osztályai. Egyéb Session beállítások is vannak, de a fejlesztés megkezdéséhez szerintem ezek a beállítások bőven elegendőek.

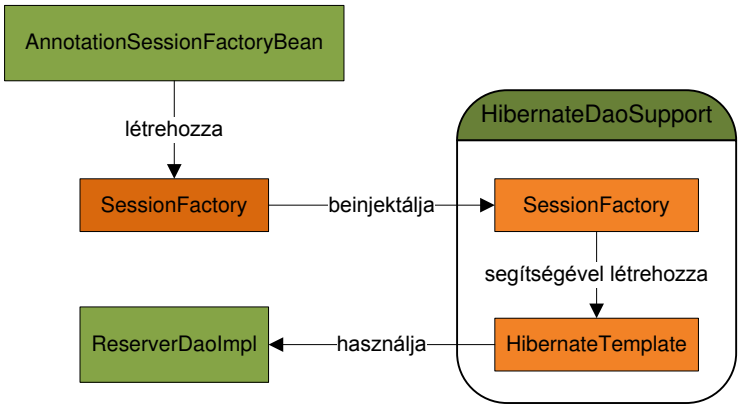
7.2.3 A HibernateDaoSupport absztrakt osztály

Miután már rendelkezünk dataSource-szal és sessionFactory-val létrehozhatunk egy DAO osztályt. Először is definiálni kell egy interfészt, majd az implementáló osztálynak ki kell terjesztenie a HibernateDaoSupport absztrakt osztályt.

```
public class ReserverDaoImpl extends HibernateDaoSupport implements ReserverDao {
	...
	@Override
	public Reserver createOrSaveReserver(Reserver reserver) {
		return (Reserver)getHibernateTemplate().merge(reserver);
	}
	...
}
```

DAO implementáció részlet

A háttérben az történik, hogy a HibernateDaoSupport absztrakt osztály létrehozza a HibernateTemplate-et, majd a DAO ezt tudja használni a getHibernateTemplate() metóduson keresztül.



17. ábra: DAO és SessionFactory kapcsolat

A getHibernateTemplate() merge metódusa, azt jelenti, ha a paraméterben kapott domain objektumnak az Id értéke 0 vagy null, akkor egy SQL insert fog lefutni, ha pedig ettől különböző, akkor egy SQL update az adott Id felhasználva azonosításra. Természetesen rengeteg metódust biztosít a HibernateTemplate, hogy az adatbázis adatainkat karbantartsuk. Minden egyes beszúró, törlő, létrehozó és kereső metódusból tucatnyi paraméterű létezik, ezzel is megkönnyítve a programozó munkáját.

Már csak egy dolgunk van, hogy használni tudjuk a DAO osztályunkat. Ez pedig az, hogy be kell injektálni a SessionFactory objektumot. Így néz ki az alkalmazás környezetleíró XML fájlban:

```
	<bean id="reserverDao" class="hu.housing.dao.impl.ReserverDaoImpl">
		<property name="sessionFactory" ref="sessionFactory"/>
	</bean>
```

7.2.4 Kontextuális session használata

Egy másik lehetőség, hogy a Hibernate sessiont használni tudjuk.

Az egyik előnye a HibernateTemplate használatának, hogy kezeli a Hibernate sessionjét. Ez azt jelenti, hogy automatikusan nyitja vagy zárja a sessiont, és biztosít arról, hogy tranzakciónként csak egy session lesz. A HibernateTemplate használatával viszont a DOA osztályunk függni fog a Spring API-tól, amit nagyon sok fejlesztő inkább kerül, hiszen a Spring a lazán csatoltság elvét vallja. Itt jön jól a Hibernate 3 által bevezetett kontextuális session. Ez a fajta session képes a Spring segítsége nélkül is biztosítani az egy session per tranzakció elvet. Használata a következő:

```
public class ReserverDaoImpl implements ReserverDao {

	private SessionFactory sessionFactory;
	public void setSessionFactory(SessionFactory sessionFactory) {
		this.sessionFactory = sessionFactory;
	}

	@Override
	public Reserver createOrSaveReserver(Reserver reserver) {
		return (Reserver)sessionFactory.getCurrentSession().merge(reserver);
	}

}
```

A DAO átalakítva kontextuális session használatára

Injektálni a sessionFactory objektumot ugyanúgy kell, mint az előbb. Ami változott, hogy most már nem terjesszük ki a HibernateDaoSupport osztályt, hogy létrehoztunk egy új attribútumot, és hogy ezt az attribútumot használjuk az adatbázis műveletekre. Így a DAO osztályunk nem fog függni egyetlen Springes objektumtól sem, hiszen a SessionFactory-t a Hibernate biztosítja számunkra.

7.3 Tranzakció-kezelés a gyakorlatban

A Spring deklaratív tranzakció kezelését a következőképpen tudjuk munkára fogni:

1. Az alkalmazásunk környezetét beállító XML állományban felvesszük a <tx:annotation-driven/> sort.
2. @Transactional annotációval látjuk el azokat az osztályokat vagy metódusokat, amelyekre alkalmazni akarjuk a tranzakció szabályait.

7.3.1 Konfiguráció az XML állományban

Először is létre kell hozni egy tranzakciót menedzselő beant, és be kell neki injektálni az előbbiekben létrehozott sessionFactory objektumunkat valahogy így:

```
<bean id="transactionManager"
      class="org.springframework.orm.hibernate3.HibernateTransactionManager">
      <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

A HibernateTransactionManager delegálja a tranzakció menedzselésének a felelősségét a Hibernate-es Transaction objektumnak, amit a sessionFactory bocsát rendelkezésre. Ha a tranzakció sikeresen lefutott, akkor a HibernateTransactionManager a Transaction osztály commit() metódusát hívja meg. Sikertelen tranzakció esetén pedig a rollback() metódust.

A Springgel tudatni kell, hogy a tranzakció kezelésére a @Transactional annotációt használjuk. Ezért az alábbi sort kell még felvenni az XML állományba:

```
<tx:annotation-driven transaction-manager="transactionManager" />
```

Ezzel megmondjuk a Spring konténernek, hogy minden a konténerben előforduló beanben keresse a @Transactional annotációt. Ezáltal minden ilyen bean AOP tranzakciós tanáccsal lesz ellátva.

7.3.2 @Transactional annotáció

A Springben a deklaratív tranzakciót tranzakciós attribútumokkal definiálhatjuk. Ezt megadhatjuk az XML fájlban, de használhatjuk a Java 5-ben bevezetett annotációkat is. Az utóbbit mutatnám be. A @Transactional annotáció vonatkozhat osztályra és metódusra is egyaránt. Ha osztályra vonatkozik, akkor az osztály minden metódusa ezzel a tranzakciós tulajdonsággal fog rendelkezni. Ha azt szeretnénk, hogy egy metódus más tranzakciós tulajdonságú legyen, akkor a metódus elé kell tenni ezt az annotációt a megfelelő paraméterekkel. Ilyen paraméterek a terjedés, az izolációs szint, az írhatóság, az időtúllépés és a visszagörgetési szabály befolyásolása.

Terjedés (Propagation) viselkedés befolyásolása azt jelenti, hogy ha egy metódus meghívásakor már életben van egy korábban indult tranzakció, akkor mi történjen. A következő értékeket tudjuk beállítani:

- MANDATORY: A metódusnak kötelező tranzakcióban futnia. Ha meghívásakor nincs aktív tranzakció, akkor kivel keletkezik. Ha van, akkor csatlakozik ahhoz a tranzakcióhoz.
- NESTED: A metódusnak beágyazott tranzakcióként kell futnia, ha a meghíváskor van létező tranzakció. A beágyazott tranzakció a beágyazó tranzakciótól függetlenül tud véglegesíteni vagy visszavonni. Ha nem volt aktív tranzakció, akkor úgy viselkedik, mint a REQUIRED. Ez az attribútum gyártó függő, tehát nem mindig támogatott.
- NEVER: A metódus nem futhat tranzakcióban. Ha a meghívásakor van aktív tranzakció, akkor kivétel váltódik ki.
- NOT_SUPPORTED: Az ilyen metódus nem támogatja a tranzakciókat. Tehát tranzakción kívül kell futnia. Ha van élő tranzakció, akkor az a metódus futásának idejére felfüggesztődik.
- REQUIRED: Ezen metódusnak kötelező tranzakcióban futnia. Ha van aktív tranzakció, akkor csatlakozik ahhoz. Ha nincs, akkor egy új tranzakció fog indulni.
- REQUIRES_NEW: Mindenféleképpen új tranzakcióban fut az ilyen metódus. Ha még nincs aktív tranzakció, akkor indítva lesz egy. Ha már létezett, akkor az felfüggesztődik, és egy új tranzakció indul a metódus futásának idejére.
- SUPPORTS: Ezen metódus meghívásakor, ha van aktív tranzakció, akkor kapcsolódik hozzá. Ha nincs, akkor tranzakció nélkül fog lefutni a metódus.

Az alapértelmezett érték a REQUIRED.

Az anomáliákat az izolációs szintekkel lehet kiküszöbölni. Egy metódusnak a következő izolációs (Isolation) szinteket lehet megadni:

- DEFAULT: Az adatbázis alapértelmezett izolációs szintjét használja
- READ_UNCOMMITTED: Egyetlen anomáliát sem tudjuk ezzel a szinttel kiiktatni, tehát egy tranzakció által jóvá nem hagyott módosítás is olvasható lesz a konkurens tranzakciók számára. Ez a szint csak akkor gondoskodik, hogy a módosítások olyan sorrendben lesznek olvashatók, amilyen sorrendben végrehajtódtak.
- READ_COMMITTED: Csak a jóváhagyott módosítást tudják olvasni a konkurens tranzakciók. Tehát kiküszöböli a piszkos olvasás anomáliát. A legtöbb adatbázis-kezelő rendszernél ez az alapértelmezett.

- REPEATABLE_READ: Ez a szint biztosítja, hogy egy mező többszöri olvasása mindig ugyanazt az eredményt adja. Tehát ezzel a piszkos olvasás mellett a megismételhetetlen olvasás anomáliáját is kiküszöböli.
- SERIALIZABLE: Ezzel a szinttel azt érjük el, hogy a konkurens tranzakciók, hiába konkurensek, olyan mintha egymás után futnának. Így azonban mind a három anomáliát megszüntetjük.

Az alapértelmezett érték a DEFAULT.

A tranzakció írhatóságnak befolyásolására két lehetőség közül választhatunk: igen, nem. A @Transactional elemei közül a readOnly-val tudjuk beállítani. Ha a tranzakciót csak olvashatóra állítjuk, akkor az adatbázis különböző optimalizációkat hajthat végre, hogy növelje programunk teljesítményét. A csak olvasható tulajdonság csak akkor lesz alkalmazva, ha tranzakció kezdődik, ezért csak azoknál a terjedési attribútumoknál van értelme használni, ami új tranzakciót hoz létre. Az alapértelmezett érték, hogy nem csak olvasható a tranzakció.

A tranzakció időtúllépést @Transactional timeout elemével lehet megadni. Másodperben kell megadni. Ha letelik az adott másodperc, akkor a tranzakció automatikusan visszagörgetésre kerül. Mivel ez is csak akkor fog beállítódni, amikor új tranzakció kezdődik, ezért csak a REQUIRED, REQUIRES_NEW és a NESTED tranzakció tulajdonságnál van értelme használni. Az alapértelmezett érték, hogy nincs időtúllépés.

A visszagörgetés viselkedését a @Transactional rollbackFor elemével lehet befolyásolni. Itt azokat a kivételosztályokat kell felsorolni, ami ha kiváltódik, akkor automatikus visszagörgetés történjen. Alapértelmezetten a tranzakció csak akkor görgetődik vissza, ha RuntimeException, tehát nem ellenőrzött kivétel váltódik ki. A noRollbackFor segítségével pedig azt tudjuk megadni, hogy mely kivételek esetén ne legyen visszagörgetés.

Lássunk egy példát a használatára:

```
public interface ReserverDao {

	@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
	public Reserver createOrSaveReserver(Reserver reserver);
}
```

Azért jó interfészt annotálni, mert így minden implementációja számára biztosítva van a tranzakciós viselkedés, és nem kell mindegyiknél külön-külön beállítani.

7.4 Az AOP a gyakorlatban

Az AOP gyakorlati használatára egy naplózási példát mutatnék be. A valós alkalmazásokban nagyon fontos a kivételek naplózása, hogy gyorsan tudjunk javítást eszközölni a hibára. A bemutatott példában is csak a kivételek lesznek naplózva. Ehhez nem kell mást tennünk, mit beállítani a Spring autoproxy szolgáltatását (lásd az Autoproxy fejezetben), létrehozni a kivételnaplózó tanácsot, majd az alkalmazás környezetbeállító XML-ben bekonfigurálni.

7.4.1 Kivételnaplózó osztály

```
public class ExceptionLoggerAdvice implements ThrowsAdvice {

	public ExceptionLoggerAdvice() {}

	public void afterThrowing(Method method, Object[] args, Object target,
			Throwable throwable) {
		StringBuffer sb = new StringBuffer();
		Logger logger =
			Logger.getLogger(method.getDeclaringClass().getName());
		sb.append("Exception at:\n\t" +
			method.getDeclaringClass().getName() + "." +
			method.getName() + "(..)\n");
		sb.append("\tArguments:\n");
		for (Object o : args)
			sb.append("\t\t(" + o.getClass().getName() + ") -- " +o+ "\n");
		sb.append("\tException info:\n");
		sb.append("\t\tMessage: " + throwable.getMessage() + "\n");
		try {
			sb.append("\t\tCause: " +
				throwable.getCause().getCause().getMessage() + "\n");
		}
		catch (Throwable th) {}
		sb.append("\t\tStackTrace:\n");
		for (StackTraceElement e: throwable.getStackTrace()) {
			sb.append("\t\t" + e + "\n");
		}
		logger.error(sb.toString());
	}
}
```

Az ExceptionLoggerAdvice osztály

Elég bonyolultra sikerült az osztály, de így benne van minden információ, ami a hibakereséshez szükséges. A kivétel, a kivétel okozója, ha van, az aktuális argumentumok és a kivétel StackTrace-e. Minden információt összegyűjtünk egy StringBuffer objektumba és ezt naplózzuk ki a kiváltó osztály naplózási beállításainak megfelelően.

Az osztályunk implementálja a ThrowsAdvice interfészt. Ezzel tudatjuk a Springgel, hogy ez egy kivétel bekövetkezésekor alkalmazandó tanács. Lennie kell egy afterThrowing metódusának, ami akkor hívódik meg, ha a paraméterben megadott kivétel váltódik ki.

7.4.2 XML konfigurálás

Ahhoz, hogy az előző osztály működjön is még az alábbi sorokat kell felvenni az XML fájlba:

```
<bean id="exceptionLoggerAdvice" class="hu.housing.advice.ExceptionLoggerAdvice"/>
<bean class="org.springframework.aop.aspectj.AspectJExpressionPointcutAdvisor">
              <property name="advice" ref="exceptionLoggerAdvice"/>
              <property name="expression"
                     value="execution(* hu.housing.service.*.*(..))"/>
</bean>
```

Először is példányosítani kell az előbb létrehozott osztályunkat, majd egy AspectJExpressionPointcutAdvisor objektumnak beállítjuk az advice paramétereként. Azért ennek az osztálynak kell átadni, mert ez AspectJ szintaktikájú kifejezéseket kezel, amivel legkülönfélébb szűrési feltételeket állíthatunk be. A szűrési feltételeket – tehát, hogy mikor fusson le a tanács – az expression paraméterének kell átadni. Itt akkor fog a tanács lefutni, ha az teljes osztálynév elején szerepel a hu.housing.service karaktersorozat, bármilyen metódusnévvel tetszőleges visszatérési érték és paraméter mellett.

8 Hibernate

A legtöbb alkalmazás kapcsolatot tart valamilyen adatbázissal. Azonban elég nehézkes egy alkalmazás adatbázis kezelése. Az adatbázis tábla sorait Java objektummá kell alakítani, és egy Java objektumot adatbázis sorrá. A Java biztosítsa erre a JDBC (Java Database Connectivity) API-t, amivel nagyon körülményes és időigényes ezen konverziók elvégzése. Itt lép a színre a Hibernate. A Hibernate fő feladata a Java objektumok leképezése adatbázis sorokká, és az adatbázis sorokból Java objektumok készítése. A Hibernate tehát elrejti előlünk a fizikai adatbázist. Ha a Hibernate 3 entitás osztályainkat annotációkkal kezeljük, akkor nagyon hasonlítanak a JPA (Java Persistence API) perzisztens objektumaira. A Hibernate-tal nagyon egyszerűen tudjuk a kapcsolatok kezelni az entitások között. A következőken az entitások konfigurálását mutatnám.

```
@Entity
@Table(name="RESERVER")
public class Reserver {
	public Reserver() {}

	@GeneratedValue(strategy=GenerationType.AUTO)
	@Id @Column(name="ID")
	private Long id;
	@Column(name="LOGIN_NAME", unique=true)
	private String loginName;
	@Column(name="FULL_NAME")
	private String fullName;
	@Column(name="EMAIL", unique=true)
	private String email;
	@Column(name="PASSWORD")
	private String password;
	.
	.
	.
	@OneToMany(cascade=CascadeType.ALL)
	@JoinColumn(name="RESERVER_ID")
	private List<Phone> phoneNumbers = new LinkedList<Phone>();
	// getter és setter metódusok
}
```

Részlet a Reserver osztályból

Hogy egy osztály entitás legyen, az alábbi dolgoknak kell teljesülnie:

- Lennie kell egy paraméter nélküli konstruktorának
- A mezőknek privátnak kell lenniük, és rendelkezniük kell getter és setter metódusokkal
- El kell látni két annotációval: @Entity és @id. Az Entity annotációt az osztálydefiníció elé kell írni, az Id-t pedig azon mező elé, amelyiket elsődleges kulcsként szeretnénk használni.

- Opcionálisan felüldefiniálhatjuk az Object osztálytól örökölt toString(), equals(Object o) és hashCode() függvények is.

Mivel ennyi is elég, hogy egy entitás osztályt létrehozzunk, ezért ilyenkor a Hibernate az adatbázisban táblanévnek az osztály nevét állítja be, a mezőnevek is az osztályban megadottak lesznek. Ha ettől el szeretnénk térni, akkor az osztálydefiníciónál megadhatjuk a @Table annotációt, aminek a name elemével beállíthatjuk az osztályunk adatbázisbeli nevét. Az attribútumokra a @Column annotációt használhatjuk. Ennek is a name elemével tudjuk megmondani, hogy az adott attribútumnak mi legyen az adatbázisbeli mezőnév megfelelője. Ha azt akarjuk, hogy egy osztály attribútuma nem vegyen részt a perzisztálásban, akkor a @Transient annotációval kell ellátni.

Az elsődleges kulcsnak megjelölt attribútumnál használhatjuk a @GeneratedValue annotációt. Ezzel a Hibernate automatikusan generál elsődleges kulcsokat az objektumainkhoz. A @GeneratedValue annotáció strategy értékével megadhatjuk, hogy hogyan generálódjon az elsődleges kulcs. A GenerationType enumerátor következő értékei közül az egyiket lehet neki beállítani:

- AUTO: az alábbiak közül azt fogja választani, ami a legjobban megfelel az alatta lévő adatbázisnak.
- IDENTITY: Ebben az esetben egy speciális oszloptípust használ a Hibernate, ami szinte mindegyik adatbázis kezelő implementációban megtalálható.
- SEQUENCE: Ezt csak néhány adatbázis kezelő ismeri (ilyen pl. az Oracle). Ilyenkor a beépített szekvencia generátort használja a Hibernate. Ennek részleteit a @SequenceGenerator annotációval tudjuk beállítani.
- TABLE: Ilyenkor a felhasználó által definiált tábla egy adott sorában és oszlopában lévő érték lesz a következő azonosító. Ezt konfigurálni a @TableGenerator annotációval tudjuk.

Az entitások összekapcsolására is használhatunk annotációt. Összesen hétféle összekapcsolási lehetőséget tudunk használni. Ez úgy jön ki, hogy számosságból négyféle típus van (egy-egy, egy-több, több-egy, több-több), és mindegyik lehet egyirányú és kétirányú. Az egyirányúság azt jelenti, hogy csak az egyik entitásból lehet a másikat elérni, fordítva nem. Azért csak hét darab, mert az egy-több és a több-egy kétirányú kapcsolat ugyanazt jelenti. A kapcsolattípusokhoz tartozó annotációk a következők: @OneToOne, @OneToMany, @ManyToOne és @ManyToMany. Többes kapcsolat esetén a

reprezentálandó objektumnak a Collection interfészt implementáló objektumnak kell lennie.
A kapcsolat tulajdonosainak meghatározása:

- Egy-egy kapcsolat esetén az idegen kulcsot tartalmazó oldal a tulajdonos.
- Egy-több kapcsolat esetén a több oldal a tulajdonos, vagyis amelyik oldalt a @ManyToOne annotációval láttuk el.
- Több-több kapcsolatban bármelyik oldal lehet tulajdonos.

A tulajdonosi viszony meghatározása azért kötelező, mert kétirányú kapcsolat esetén az inverz oldalnak hivatkoznia kell a tulajdonosra egy mappedBy elemmel amit, számosságot jelölő annotációban kell megadni.
Mint azt jól tudjuk, a kapcsolatokat adatbázis táblákra az idegen kulcs segítségével lehet leképezni. Ennek helye a következők szerint alakul:

- Egy-egy kapcsolat esetén bármelyik oldalra tehető az idegen kulcs.
- Egy-több kapcsolat esetén mindig a több oldalon van az idegen kulcs.
- A több-több kapcsolat esetén pedig mindig szükség van egy kapcsolótáblára, amiben az idegen kulcsot tároljuk.

Az idegen kulcs oszlopának a nevét a @JoinColumn annotációval lehet megadni. A több-több kapcsolat esetén a kapcsolótábla nevét a @JoinTable annotációval beállítani, ezen belül az oszlopok nevét ugyanúgy a @JoinColumn határozza meg.
Még egy fontos dolog van, ez pedig a kaszkádolás. Ez annyit jelent, hogy ha az entitáson elvégzünk valamilyen műveletet, akkor az a művelet végrehajtódjon-e az adott entitással kapcsolatban lévő entitásra is. Ezt a tulajdonságot a kapcsolatban tulajdonosként részvevő kapcsolatleíró annotációban kell megadni a cascade elemnek. Ezt a CascadeType enumeráció alábbi elemeivel lehet beállítani:

- ALL: minden művelet,
- MERGE: csak a merge (beszúrás vagy frissítés) típusú művelet,
- PERSIST: csak a beszúrás művelet,
- REFRESH: csak a frissítés művelet,
- REMOVE: csak a törlő művelet

hajtódik végre a kapcsolatban részvevő entitáson is. Alapértelmezetten nincs semmilyen kaszkádolás.

9 Összegfoglalás

Célom az volt, hogy bemutassak egy architektúrát, ami minden tekintetben megfelel egy gazdag internet alkalmazás elkészítéséhez. Remélem ezt sikerült is elérnem, aki elolvassa ezt a diplomamunkát, az betekintést nyer az egyes komponensek alapvető működésébe. Nem csak leírást akartam adni ezzel a munkával, hanem példákat is szolgáltatni a használatukra. A Flex komponenseit bemutatni megfelelő keretek között lehetetlen lett volna, hiszen nagyon sok van belőlük. Inkább az üzleti alkalmazások számára lényeges adatvalidálást és adatformázást, és a komponensek testre szabásának lehetőségeit mutattam be. A Cairngorm egy kiváló mikro-architektúra, hogy a kliensoldali logikát átlátható módon kezeljük. A BlazeDS szerverrel egyszerűen tudunk kapcsolatot kialakítani a szerver és a kliens között. Nagyban megkönnyíti az átjárhatóságot a Java és az ActionScript között. A Spring egy nagyon kiváló eszköznek ismertem meg. Segítségével többek között tranzakció kezelést és a kliens által elérhető távoli célokat tudjuk Java nyelven implementálni. Az üzleti életben fontos tranzakció kezelés aspektus orientált programozás segítségével történik, ezért az AOP-ot egy picit részletesebben mutattam be. Ez egy nagyszerű dolog, hiszen így teljesen csak az üzleti logikának a programozására kell koncentrálnunk. Az AOP-ot saját céljainkra is fel lehet használni, ilyen például az alkalmazás naplózása. Spring segítségévek DAO-kat írni is nagyon egyszerű, mivel a Spring kínál hozzájuk Hibernate támogatást. Végül a Hibernate entitáskezelését mutattam be. Ez is egy remek dolog, mert leképezi helyettünk az osztályok adatbázis rekordokká, oda-vissza irányban, így nekünk ezzel nem kell foglalkozni. Ezek csupa olyan dolgok, amik a programozó munkáját könnyítik, mind fejlesztés, mind karbantarthatóság tekintetében.

10 Felhasznált irodalom

Chafic Kazoun, Joey Lott - Programming Flex 3 (O'Reilly, 2008)

Craig Walls - Spring in Action, Second Edition (Manning, 2008)

Chris Giametta - Pro Flex on Spring (Apress, 2009)

Nyékiné – J2EE útikalauz Java programozóknak (ELTE, 2002)

Imre Gábor – Szoftverfejlesztés Java EE platformon (Szak, 2007)

http://livedocs.adobe.com/flex/3/html/help.html

http://help.adobe.com/en_US/ActionScript/3.0_ProgrammingAS3_Flex/index.html

http://www.adobe.com/devnet/flex/articles/introducing_cairngorm.html

http://livedocs.adobe.com/blazeds/1/blazeds_devguide/index.html

http://static.springsource.org/spring-flex/docs/1.0.x/reference/html/index.html

http://static.springsource.org/spring/docs/2.5.x/reference/index.html

http://static.springsource.org/spring/docs/2.5.x/api/index.html

http://docs.jboss.org/hibernate/stable/core/reference/en/html/

http://docs.jboss.org/hibernate/stable/annotations/reference/en/html/