Általános Informatika Tanszék

KÜLSŐ MODULOK BEKAPCSOLÁSA JAVA 9 ÉS ÚJABB ALKALMAZÁSBA FUNKCIONÁLIS PROGRAMOZÁSI TECHNIKÁVAL

Szakdolgozat

Készítette: Bozzay Ádám
Neptun-kód: D898C0
Cím: Miskolc, Szántó Kovács János utca 29.

Tartalomjegyzék

1. BEVEZETŐ

Ezen dolgozatnak két fő célja van. Egyrészről annak a lehetőségét vizsgálom, hogyan lehet Java alkalmazásba külső modulokat bekapcsolni. Külső modulok alatt a programlogika egy-egy egységét értem, amelyek nem tartoznak a program fordítási egységébe, a program nyelvétől különböző nyelven íródhatnak, és akár futási időben kicserélhetőek. A külső modulok szerepe a cserélhetőség, annak a lehetőségnek a biztosítása, hogy az alkalmazás az újabb és újabb igényekhez könnyen igazodni tudjon.

A témaválasztás ezen részére sok tényező jelentett motivációt, melyek közül néhányat itt megemlítek. Az egyik tényező a korábbi bekezdésből már adódik: ne kelljen minden specifikus felhasználási módra felkészíteni az alkalmazást, a felmerülő igényeket utólag egy másik fejlesztőcsapat vagy akár maga az ügyfél is elkészítheti. Egy másik motiváló tényező, hogy szeretnék a programlogika egyes részeire, egyes algoritmusokra hasonlóan tekinteni, mint az adatokra. Ha konfigurációs adatot és üzleti adatot lehet kívülről megadni az alkalmazásnak, akkor adódik, hogy logikát is lehessen. Erről bővebben a „2. Koncepció” fejezetben írok. A munkahelyemen megbizonyosodhattam róla, hogy erre valós igény van. Végül, a dolgozat írása közben szembesültem azzal, hogy a Java nyelvhez jelenleg nem létezik kielégítő keretrendszer külső modulok használatára. Amelyek elérhetőek, azokról a „3.1.3 Jogosultságkezelés Javán belül” fejezetben írok, de ezek valódi megoldást nem jelentenek. Így olyan témán dolgozhatok, amelyre még nincs kész megoldás.

A dolgozatban egy példaalkalmazás segítségével bemutatom, hogy funkcionális programozási technika segítségével hogyan lehet egy intuitív, egyszerűen cserélhető és bővíthető rendszerstruktúrát kidolgozni. Külön hangsúlyt fektetek egy ilyen, külső logikát használó rendszer biztonságtechnikai kérdéseire. Fontos kitérni arra is, hogy a logika cserélhetőségével a felhasználó felület rugalmassága is együtt jár. A dolgozat egy másodlagos céljának tekintem, hogy egy olyan struktúrát dolgozzak ki, amelyben a felhasználói felület elemei a logika változásaival együtt változzanak. A példaalkalmazásban egy külső modul egyaránt tartalmaz algoritmust és UI leíró kódot is.

A dolgozat másik célja a Java nyelv 9-es verziójában bevezetett modulrendszer tanulmányozása, bemutatása. A modulrendszer a csomagoknál magasabb szeparációs szintet ad a Java alkalmazásokhoz. Többek között szeretném bemutatni azt, hogy a modulrendszer hogyan működik, hogyan tud más technológiákkal együttműködni és bevezetése milyen kihívásokat hozhat magával a fejlesztők számára. Ezen felül vizsgálom a Java modulrendszer felhasználhatóságát a külső modulok kezelésre.

A „modul” szót a dolgozatban kétféle értelemben használom. Szeretném tisztázni, hogy a „Java modulrendszer” az alkalmazás feldarabolását célozza meg belső egységekre. A „külső modul” alatt egy az alkalmazáson kívüli üzleti logikai egységet értek, mely jelentésében a „plugin” szóhoz áll közelebb. A két „modul”-nak azon felül, hogy „logikai egység” jelentéssel bírnak, nincs közük egymáshoz.

A modulrendszer vizsgálatára, bemutatására szintén több tényező jelentett motivációt. Egyrészt a modulrendszert már 2008 óta tervezik bevezetni a Java nyelvbe.[1] Kíváncsi voltam, hogy egy ilyen régóta tervezett és a Java nyelvet ilyen jelentősen érintő funkció hogyan néz ki, mikor elkészül. Másrészt, mivel egy általánosságban véve is fontos funkcióról beszélünk, ez alatt az idő alatt más megoldások is elterjedtek a modulrendszer funkciójának betöltésére (pl. *Maven*, *Gradle*). Kíváncsi voltam, hogy a „hivatalos” és „nem hivatalos” megoldások hogyan férnek össze, illetve hogy a Java új rendszere kiválthatja-e a már elterjedt rendszereket. Harmadrészt úgy tűnik, a Java modulrendszer a mai napig nem terjedt el igazán, így kevés publikáció született róla. Ez lehet az első publikáció, amely magyar nyelven, példákon keresztül és gyakorlati szemszögből bemutatja a Java modulrendszert. Végül motivációt jelentett önmagában az is, hogy a Java új funkciókkal bővült. A Java 8 óta követem a nyelv újdonságait és felhasználási lehetőségeit. A Java 9 2017 szeptemberében jelent meg, én ekkortól érdeklődöm a technológia iránt. Ez remek alkalom arra, hogy a Java 9 adta új lehetőségeket másoknak is bemutassam.

A dolgozat keretében tehát példákkal illusztrálva mutatom be a Java 9 használatát. Az 1. Melléklet egy komplett, de egyszerű példa a Java 9, a Spring és a Maven együtt történő használatára, illetve maga a dolgozat példaalkalmazása is ezekre a technológiákra épül.

2. KONCEPCIÓ

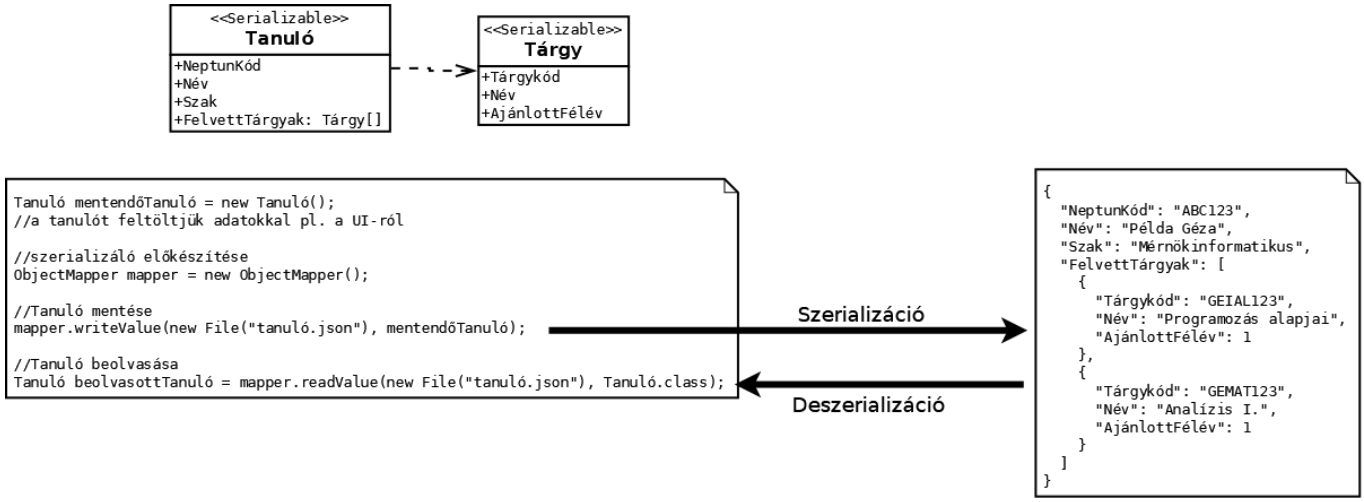
A bevezetőben említettem, hogy a logika egyes egységeire, egyes algoritmusokra szeretnék hasonlóan tekinteni, mint az adatra. Ahhoz, hogy bemutassam, mit értek ezalatt, és hogy mi jelentette a motivációt, kissé távolabbról indulok.

Minden alkalmazás dolgozik adattal, ezt az adatot valahonnan be kell olvasni, fel kell dolgozni és el kell tárolni. Objektumorientált programozási nyelvek használata esetén az alkalmazás életciklusa alatt valahol, valamikor az adat objektumok formájában jelenik meg, hiszen az objektumorientált szemlélet szerint a különböző adatokat különböző objektumokkal modellezzük. Ha tehát adatot szeretnénk az alkalmazásba beolvasni vagy kiírni, akkor azt objektumokba kell beolvasnunk, illetve objektumokból kiírnunk.

Bízhatjuk a fejlesztő fantáziájára is, hogyan transzformálja át az adatot az objektumból a forrásba és vissza. Ilyen eset például, amikor az objektumorientált kódba SQL utasításokat írunk, és mi magunk döntjük el, hogy az objektumot hogyan „mappoljuk” rá az adatbázisra. Egy másik példa, amikor az adatot CSV formátumba mentjük el és olvassuk ki.

De mivel az adat az objektumorientált programban már eleve egy struktúrába van szervezve – objektumba –, adja magát az ötlet, hogy ezt a struktúrát felhasználva mentsük ki, illetve olvassuk be az adatot. Az SQL-es példánál maradva ez lenne az *ORM*. De ilyenformán az objektumokat nem csak adatbázisba, hanem XML, JSON, bináris, stb. formátumokba is könnyedén le tudjuk menteni (szerializáció), illetve ugyanolyan könnyedén vissza is tudjuk olvasni (deszerializáció). A folyamat egyszerűségét az 1. ábra szemlélteti.

A példából jól látszik, hogy ez egy rendkívül egyszerű, intuitív folyamat, éppen ezért hasonló eljárást széles körben használnak. Az adat tárolása és beolvasása tehát kényelmesen megoldható. Azonban egy program nem csupán adatokból áll, hanem adatokból és a rajtuk műveletet végző logikából.



***1. ábra**: Példa szerializációra és deszerializációra*

„A számítógépes program azon utasításoknak a sorozata, amelyeket a számítógépnek egymás után végre kell hajtania valamely feladat elvégzése céljából, jellemző módon azt, hogy az adatokkal milyen műveleteket végezzen. A számítástechnikai és számítógéptudományi zsargonban a „programokat" és az „adatokat" általában különböző fogalomként kezeljük, habár az informatikai alapismeretek szerint mindkétféle bithalmaz, tágabb értelemben, adatnak számít."[2]

Adódik a kérdés, hogy lehetséges-e a programlogika egy részét lementeni, illetve betölteni, hasonlóan ahhoz, ahogy az adatot lementjük, illetve betöltjük. Lehetséges-e a logikát szerializálni? Ha igen, hogyan néz ki ez a gyakorlatban? Hogyan lehet biztosítani az egyszerűséget, az intuitivitást, a bővíthetőséget, a cserélhetőséget – akár futás közben is, a biztonságot, stb. Ezekre a kérdésekre keresem a választ e dolgozatban.

A példaalkalmazás, melyben mindezeket szemléltetni fogom egy adatrögzítő- és menedzselő rendszer, mely megmérettetéseket (versenyeket) rögzít és menedzsel. A rendszer jellemzői:

Moduláris, mert az egyes felhasználói igényekhez különálló logikai és megjelenítési egységek készíthetőek, így a rendszer egyaránt használható pl. sportversenyekhez, e-sport megmérettetésekhez, oktatási versenyekhez, ill. bármilyen kompetitív tevékenység levezényléséhez. Rögzítő rendszer, minthogy elsődleges feladata a különböző pontszámok,

részeredmények, eredmények elektronikus lejegyzése. És végül, menedzselő rendszer, mert a pontszámokat, részeredményeket, eredményeket feldolgozza, aggregálja, transzformálja.

Egy modul tartalmazza az adott megmérettetéshez tartozó pontszámaggregáló és -transzformáló logikát, valamint a pontos felhasználói felületet (amin keresztül a pontokat rögzíthetjük és közben a részeredményeket láthatjuk). Ilyen modult írhat bárki, az ehhez szükséges API nyílt lesz, de nyilvánvalóan jómagam is biztosítok néhányat. A felhasználó letöltheti a szükséges modulokat az internetről, majd a megmérettetés létrehozásakor kiválaszthatja a megfelelőt. A program kinézete ennek megfelelően alakul.

A rendszer egyszerű, de valószerű példaalkalmazásként lesz lefejlesztve, a teljes értékű funkcionalitás elkészítése (pl. felhasználókezelés, publikus webes felület, stb.) ezen dolgozatnak nem témája. A dolgozat célja a fenti működés implementálása, valamint néhány modul elkészítése.

3. A MODULARITÁSRÓL

Modularitás alatt számos dolgot érthetünk. A különböző forrásokban említett jellemzők alapján a következő definíció adódik:

A (program-)modul egy olyan szoftver egység, amely egységbe zárja a teljes program egy-egy funkcióját, lecserélhető, jól definiált interfésszel kommunikál a külvilággal, valamint önálló – önállóan fordítható, tesztelhető.[3][4]

A példaalkalmazásban egy modul egy megmérettetésnek felel meg, tartalmaz mindent, ami ahhoz szükséges, hogy a szoftver ezt a megmérettetést kezelni tudja. Tartalmazza a szükséges logikát és a felhasználói felületet is definiálja. Mivel a szoftvernek szigorúan véve nem a része, felmerülhet, hogy ez inkább plugin, mint modul.

A plugin *„Egy adott szoftverbe vagy hardverbe opcionálisan beépíthető, annak képességeit bővítő vagy módosító kiegészítő modul.”*[5] Ezen definíció nem felel meg teljesen a fentebb ismertetett modulnak, ugyanis az nem opcionális, legalább egyre szükség van a szoftver működéséhez, valamint a szoftver működését nem kiegészíti, hanem megadja, pontosítja. Amit ezen dolgozatban modulnak nevezek tehát a klasszikus értelemben véve nem plugin. Mindazonáltal a modularitás megvalósításakor számos technológiai megoldást a pluginok megoldásai közül használok fel, így a továbbiakban nem teszek különbséget pluginra és modulra vonatkozó forrás között.

A továbbiakban a „fő programra” – melybe a modulok beépülnek – „keretprogram” néven fogok hivatkozni. A kijelentés, hogy egy alkalmazásba kívülről logikát szeretnék beépíteni, némi kifejtést igényel. Mit értek az alatt, hogy kívülről, hogy logikát, illetve hogy beépíteni.

Az alatt, hogy „kívülről” azt értem, hogy a keretprogramtól függetlenül. Vagyis akár független fejlesztőktől, függetlenül a keretprogram fejlesztési idejétől, ütemétől, más technológiai megoldásokkal készülhet el egy modul.

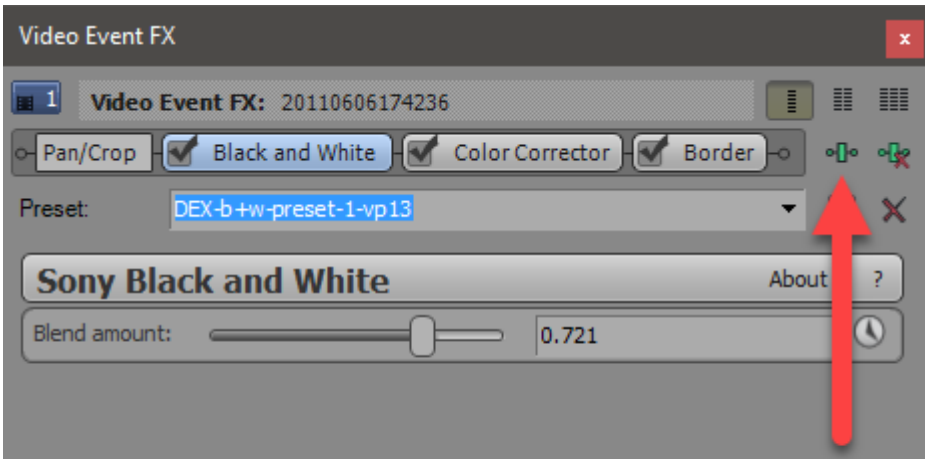
Az alatt, hogy „logika” egyszerűbb, vagy komplexebb algoritmust értek. Az egyes logikai elemek leválasztása a keretprogramról azt a célt szolgálja, hogy ezeket a logikai

elemeket könnyen ki lehessen cserélni. A kívülről bevitt logika nem a keretprogram magasabb szintű működését befolyásolja. A logikai elemek függvények, vagyis van bemenetük és kimenetük, a feladatuk pedig az, hogy a bemenetet áttranszformálják a kimenetté. Elsősorban ilyen függvényekre gondolok, mint aggregáció (bemenet egy lista, kimenet egy skalár érték), szelekció (skalár értéket másik skalár értékké transzformál), szűrés (listát szűkebb listává transzformál), stb.

A „beépítés” alatt azt értem, hogy a keretprogramban ki vannak alakítva kapcsolódási pontok, és ezekhez független logikát kapcsolok. Ez azt jelenti, hogy van egy skalár vagy lista értékem, amit paraméterként átadok a külső függvénynek, és egy skalárt vagy listát várok vissza. A keretprogram innentől kezdve ezzel az új értékkel dolgozik tovább. A külső logika tehát a program folyását nem változtatja meg, csak az adatot, amivel a program dolgozik.

A fentebb leírt működés alapvetően különbözik attól, amit plugin alatt szokás érteni. A pluginok kiegészítik a keretprogram működését, ezt sokszor úgy érik el, hogy befolyásolják a keretprogram futását, pl. meghívják a keretprogram egyes rutinjait. Az általam prezentált modulok nem így viselkednek, soha nem hívnak át a keretprogramba, hanem a keretprogram hív át beléjük (*Hollywood elv*).

A modulok működése egyes audio-, és videoszerkesztő alkalmazások szűrőpluginjaihoz hasonlítható (2. ábra). Ezek láncba összefűzhetőek, mindegyik kap egy bemenetet, azt valahogyan áttranszformálja, majd továbbadja. Ezek a szűrőpluginok nem befolyásolják a program működését, csupán függvények, amelyek adatot (itt képkockákat) transzformálnak. Ezért hasonlatosak az általam használt modulokhoz.



2. ábra: Ez a kép a Sony Vegas videóvágó program egy pluginláncát mutatja. A pluginok különböző effektekért felelősek, az első effekt megkapja bemenetként a nyers videót, transzformálja majd továbbadja a következőnek, az szintén transzformálja, és így tovább.
Forrás: https://www.moviestudiozen.com/doctor-zen-faq/586-how-to-save-a-chain-of-fx-presets-in-vegas

3.1 Technológiai lehetőségek

A modularitás megvalósítása kapcsán a következő igényeknek szeretnék megfelelni:

- futásidejű cserélhetőség (hot-swap),
- specifikus programozási ismeretek szükségességének minimalizálása – bárki készíthessen, személyre szabhasson modulokat,
- platformfüggetlenség és egyetlen futtatható állomány – az alkalmazás maga képes legyen az igényeket teljesíteni mindennemű külső program nélkül,
- törekvés a biztonságra.

Ezeket az igényeket az utolsó pontból kiindulva vizsgáltam, úgy gondolom modularitás kapcsán alapvető probléma az egységbezárás, ill. információrejtés elégséges megvalósítása, vagyis a biztonságos kapcsolat megteremtése a keretprogram és a modulok között. Ha a modulokban bármilyen Java forráskód használata megengedett volna, akkor elkerülhetetlen volna, hogy a modulok rosszindulatból vagy nemtörődömségből a keretprogram olyan területeihez férjenek hozzá, amelyekhez nem volna joguk. Kutatásaim során több potenciális módszert találtam a modularitás biztonságának megteremtéséhez.

3.1.1 Szkriptnyelvek

Ha biztonsági problémát jelent a Java nyelv használata a modulokban, akkor kézenfekvő megoldás, hogy ne Java nyelvet használjunk. Számos szkriptnyelv áll

rendelkezésre, amelyekkel külső logikát lehet beépíteni a Java programokba. Ehhez egységes felületet nyújt a Java Scripting API.

A Java Scripting API egy szkriptnyelv-független keretrendszer szkriptnyelvek integrálásához a Java programba. Az egységes felületnek köszönhetően a program írásakor nem kell rögzíteni a szkriptnyelvet, a szkript írója azt megválaszthatja. Gondoljunk csak bele, mekkora jelentőségű ez. Aki a keretprogramhoz modult készít, tetszőleges, a számára legszimpatikusabb nyelven megírhatja azt. Nem hallottam még olyan programról, amelyhez a pluginokat tetszőleges nyelven meg lehetne írni, ez mindenképp újszerű gondolat.

Néhány konkrét Java-kompatibilis szkriptnyelv a teljesség igénye nélkül: Nashorn (JavaScript), luaJ, Jython, Jruby, Scala, Groovy. Ezek közül jelenleg a legnépszerűbb Java szkriptelési célra a Groovy és a Nashorn. (De a Nashorn a Java 11 megjelenésétől kezdve deprecated-nek lett nyilvánítva).

Azonban azért merült fel eredendően a szkriptnyelvek használata, hogy a külső modul ne férhessen hozzá a program belső osztályaihoz. A fentebb említett nyelvek azonban mind bájtkódra fordulnak és a belső programmal egyazon JVM-en futnak, így semmiféle garancia nincs rá, hogy ne tudnának hozzáférni a program belső osztályaihoz, így a létjogosultságuk biztonsági szempontból megkérdőjelezhető.

Az, hogy a szkriptnyelvek nem jelentenek valódi biztonságot, nem biztosítanak homokozó (sandbox) környezetet, felvet néhány kérdést. Mi értelme van így a szkriptelésnek, valóban nem biztonságosak-e, illetve nem lehet-e őket biztonságossá tenni valamilyen módon? Mindegyik kérdést feltették már mások is előttem, és választ is kaptak rá.

1. A szkriptelés értelme: A Java Scripting API használható akkor, ha más nyelvre írt kód könyvtárat szeretnék használni, ha a felhasználónak lehetőséget akarunk adni arra, hogy snippeteket (független kódtöredékeket, pl. szűrűfeltételeket) adjon meg az alkalmazásnak, ha komplex logikát szeretnénk megadni egy konfigurációs fájlban, illetve ha olyan nyelvi elemeket szeretnénk használni, amelyek a keretprogram nyelvében nem elérhetőek[6].
2. A szkriptelés biztonsági kérdése: Több módja is van annak, hogy egy szkript kitörjön a neki szánt környezetből. Használhatja a Class.forName() metódust. Használhatja az egyes Script Engine-ekbe beépített utasításokat, pl. Nashorn esetén

a quit() függvény használata.[7] Használhatja a számára mindig elérhető „engine” adattagot, amin keresztül bővebb jogosultsági kört szerezhet magának akkor is, ha egyébként a jogosultsági körét előzőleg valahogyan leszűkítették.[8] Ez csupán néhány a számtalan biztonsági kockázat közül.

3. A biztonság megteremtése: A 2. pont értelmében a biztonság igenis megkérdőjelezhető, sőt kijelenthetjük, hogy alapesetben a szkriptnyelvek használata nem biztonságos. Hogyan lehet mégis biztonságossá tenni a használatukat? Mivel a szkriptek a keretprogrammal egyazon JVM-en futnak, biztonsági szempontból a Java kódhoz nagyon hasonlóak. A következő alfejezetekben megvizsgálom a logika kívülről történő bevitelének további megoldási lehetőségeit, ezek a szkriptekre is vonatkoznak.

3.1.2 Külön JVM minden modulnak

A szkriptnyelvek tehát nem jelentenek valódi biztonságot, mert a keretprogrammal azonos JVM-en futnak. Kézenfekvő a következtetés, hogy minden modult futtassunk külön JVM-en. Ekkor használhatnánk akár szkriptnyelvet, akár Javát, a modul teljesen el lenne szigetelve a keretprogramtól. Ez azonban rendkívül erőforrás igényes, és elvesztenénk vele a platformfüggetlenséget. Közvetlenül programból körülményes elindítani egy újabb JVM példányt, ehhez operációs rendszer hívásra van szükség, ez pedig függ az adott operációs rendszertől.[9]

A dolgozat írása közben kaptam egy tanácsot, miszerint érdemes lenne mindössze két JVM-et futtatni. Ha az egyik JVM-ben fut a keretprogram, a másikban pedig az összes modul együtt, akkor a modulok és a keretprogram hermetikus elszigetelése megvalósul. Ez önmagában nem ad megoldást az összes biztonsági kérdésre, és továbbra is operációs rendszer függő, de számos problémát leegyszerűsít. Ha ez a megoldás hamarabb eszembe jut, talán inkább ezen az úton indultam volna el. Néhány szót mindenképp megérdemel ez a megközelítés is.

Ha a modulok külön JVM-en futnak, akkor reflection segítségével nem férnek hozzá a keretprogramhoz. Nem tudják a keretprogram állapotát sem megváltoztatni, pl. váratlanul leállítani, végtelen ciklussal leterhelni, stack túlcsordulással összeomlasztani. De egy modul továbbra is befolyásolni tudja a többi modul működését, törölhet fájlokat, módosíthat környezeti változókat, stb. Ezek elkerülése végett használhatunk pl. jogosultságkezelést (lásd a továbbiakban).

Sok fantáziát láttam az -Xbootclasspath parancssori paraméter használatában a java program indításához. Ez utóbbi arra ad lehetőséget, hogy gyári JRE osztályokat, pl. a java.lang csomagban lévőket lecseréljük. Tehát egyszerűen kivehetem azokat a metódusokat, osztályokat, csomagokat, amelyeket nem tartok biztonságosnak. Ennek az ötletnek a gyakorlatban történő megvalósítása azonban valószínűleg komoly nehézségekbe ütközne. Érdemes észben tartani a következőket is: Oracle JDK-t és JRE-t használva a gyári osztályok átalakítása, megkerülése sértheti a licenszfeltételeket[10], illetve ez egy „nem szabványos” paraméter, amit bármikor kivehetnek a platformból, vagy átalakíthatják, illetve lényeges különbségek lehetnek az Oracle és más Java implementációk között.

Az a megoldás tehát, hogy a keretprogramtól függetlenül a modulokat egy külön, közös JVM-ben futtassuk, sok pozitívummal rendelkezik. De ezzel mindenképpen veszítünk a platformfüggetlenségből, a két külön java program indítására platformspecifikus, alaposan tesztelt implementáció szükséges. Emellett a felmerült védelmek kialakítása sem egyszerű feladat. A két JVM példány közötti kommunikáció is hordoz magában kihívást, de erre vannak bevett módszerek, mint pl. *RMI*, socket-ek localhost-on, *JMX*, kifejezetten erre készült könyvtárak, és hagyományos IPC mechanizmusok.

A fenti negatívumok miatt, és azért, mert későn merült fel ez a lehetséges megoldás, nem ezt fogom alkalmazni a dolgozatban. Azonban ha valaki hasonló problémával áll szemben – sandbox környezetet szeretne kialakítani Java-ban – annak ez egy hasznos gondolat lehet.

3.1.3 Jogosultságkezelés Javán belül

A fentiek alapján eldöntött kérdés, hogy a főprogramnak és a moduloknak egyazon JVM-en kell futniuk. Ebben a szituációban kell a biztonságot garantálni. Van lehetőség Javán belül megvalósítani a jogosultságkezelést, erre szolgál a SecurityManager[11]. Ez az osztály lehetővé teszi, hogy az alkalmazások biztonsági policy-t határozzanak meg, ellenőrizzék az egyes műveleteket, és engedélyezzék vagy megtagadják azokat. Emellett célszerű saját *ClassLoader*-t is implementálni, hogy a külső osztály betöltésekor ellenőrizhetőek legyenek a felhasznált függőségek. Ez a legkorrektebb megoldás, azonban

irreálisan körülményes ezt a megoldást implementálni, és rendkívül sok tesztelést igényel, hogy az összes kiskapu befoltozásra kerüljön.

Esetleg részleges védelemre, második védvonalként lehet hasznos. Bizonyos műveletek könnyen letilthatóak vele, pl. fájlműveletek vagy az alkalmazásból való kilépés. De számos hátránya van. Egyrészt nehéz róla információkat találni. Nem csoda, ez egy elég speciális eleme a Java nyelvnek. Másrészt a SecurityManager célja globális jogosultságok beállítása. Ezt az eszközt még a Java 2-vel, 1997 körül készítették, ekkor a nyelvnek még más céljai voltak, mint ma. A leglényegesebb ok, amiért a SecurityManager-t bevezették, a Java Appletek világa volt. Az Appleteket weboldalakon használták, és a kliens gépen található JVM futtatta őket, így rendkívül fontos volt, hogy alapesetben ne férjenek hozzá a kliens fájlrendszeréhez. Az Appleteket azonban már évek óta nem használják (jelentős részben biztonsági okokból), végül 2018 szeptemberében távolították el a platformból[12]. Bár az Appletek esete csak egy példa volt, annak idején sok más felhasználási módja volt a Javának, ahol a globális jogosultságkezelés hasznos lehetett, de ezek manapság már kevéssé relevánsak.

A fentiek miatt tehát a SecurityManager-t globális, teljes JVM példányra megadott jogosultságkezelésre szánták. Nincs lehetőség külön jogosultságokat megadni egyes csomagokra, osztályokra vagy metódusokra. Kikerülő megoldások vannak, de azok csak tovább bonyolítják a SecurityManager használatát[13]. Ilyen módszerek képzelhetőek el, mint a call stack-ről kikeresni a hívó metódust vagy megvalósítani egy szálra nézve lokális SecurityManager-t. Összességében a körülményessége miatt tehát ez az út nem megfelelő.

Léteznek kész megoldások is a moduláris szoftver felépítéséhez, ezek közül többet kifejezett alkalmasnak neveznek külső modulok, plugin-ok beépítésére. A következő keretrendszereknek néztem utána: OSGi (Open Services Gateway initiative), JPF (Java Plugin Framework), PF4J (Plugin Framework for Java). Ezen keretrendszerek közül azonban egyik sem állítja, hogy megvalósítható vele a modulok elszigetelése. A JPF és PF4J (különösen az utóbbi) kényelmes lehetőséget biztosít a modulok dinamikus betöltéséhez, ami nagy előny, azonban ezen túlmenően nincs több előnye pl. a szkriptnyelves módszerhez képest. Sőt hátrány, hogy ezen keretrendszerek már lefordított forráskódból tudnak dolgozni, tehát a modulok készítőjének külön le kell fordítania és

össze kell csomagolnia a kódot. Nézzük meg röviden az említett 3 technológia tulajdonságait.

Az OSGi inkább egy architektúra specifikáció, mint egy keretrendszer. A használata rendkívül körülményes, és nagyon kevés érthető és aktuális dokumentum található hozzá. Elvileg meg lehet benne valósítani class fájlok dinamikus betöltését biztonságosan. Inkább nagy enterprise alkalmazások tervezésekor jöhet szóba az alkalmazása, esetünkben nem megfelelő.

A JPF inkább azért került a listába, mert több helyen is említik. De a fejlesztését még 2007-ben leállították, és a dokumentációjában sehol sem említik meg a biztonság kérdését. Elvileg alkalmas osztályok dinamikus betöltésére, de az említett okok miatt nem jöhet szóba.

A PF4J a legígéretesebb a három közül, ezt még ma is fejlesztik. Bár plugin framework-ként emlegetik, valójában inkább az alkalmazás strukturálására lehet használni. Futásidőben nem lehetséges pluginok beépítése. Inkább belső bővíthetőségre való, nem kívülről érkező kód beépítésére. A dokumentációjában nem foglalkoznak a biztonság kérdésével. De könnyen bővíthetőre van tervezve a keretrendszer, és minden plugint külön ClassLoaderrel tölt be, így valószínűleg meg lehetne valósítani a pluginok elszeparálását, de semmivel sem lennénk előrébb, mint a SecurityManager-rel.

Van egy negyedik keretrendszer is, a TWMAS. Ez egy kísérleti projekt volt, ami hasonló kérdésre kereste a választ, mint ez a szakdolgozat. Hogyan lehet Javában, platformfüggetlenül kialakítani egy sandbox környezetet, hogy az alkalmazásba kívülről biztonságosan be lehessen építeni kódot. Miután ez a projekt csak egy prototípus, és nem egy kész megoldás, illetve a SecurityManager-t használja, amelynek számos hátránya van, nem kívánok erre a projektre építeni. Ugyanakkor készült hozzá egy dokumentáció, amiből sokat lehet tanulni. Ezt a dokumentációt a 2. Mellékletben csatoltam, kiegészítve néhány kapcsolódó információval.

A fentiek értelmében a Javán belüli jogosultságkezelés önmagában nem felel meg a célnak. Egyéb megoldások mellett második védvonalként a SecurityManager hasznos lehet. Már létező megoldások között alkalmasat nem találtam.

3.1.4 Java 9 modulok

Az eddigieket tekintve meglehetősen szerencsétlen helyzetben van az, aki külső modulokat szeretne Java alkalmazásba kapcsolni biztonságosan. Az összes lehetséges megoldást számba véve egyik sem felel meg a célnak. Azonban 2017. szeptember 21-én megjelent a Java 9[14], amely új lehetőségeket nyithat.

A Java 9-ben bevezetésre kerültek a JVM-szintű modulok. Ezek egységbe zárnak több package-et, bennük több osztállyal. Lehetőségünk van meghatározni egy module-info.java speciális fájlban, hogy az egyes modulok mely package-eket exportálnak a külvilág számára (`exports` kulcsszó), illetve hogy mely más modulokra nézve van függőségük (`requires` kulcsszó). Mivel a modulrendszer JVM-szintű, megvalósul a teljes elszigeteltség – habár később láthatjuk, hogy ez így ebben a formában nem teljesen igaz.

Adódik a gondolat, hogy a külső modulokat feleltessük meg egy-egy Java modulnak. A Java 9-es modulrendszert e célra használva két hiányossága azonnal szembeötlik: nem támogatja a hot-swap-ot, és előfordított kódra van hozzá szükség. (Az implementáció során számos egyéb nehézségbe is ütköztem, ezeket a megfelelő fejezetekben tárgyalom.)

A hot-swap hiányossága megoldható egy launcher segítségével. Esetünkben nincs feltétlenül szükség hot-swap-ra, elegendő, lehet, ha az alkalmazás indításakor megadjuk, hogy milyen modulokat töltsön be. Először elindul a launcher, ez betölti a szükséges modulokat, majd ezután indul el ténylegesen az alkalmazás. Ez így azonban platformfüggő kódot igényel, a launcher-ből nem lehet teljesen platformfüggetlenül elindítani a Java alkalmazást, ahogyan ez a „3.1.2 Külön JVM minden modulnak” fejezetben is ki lett tárgyalva. Alternatív lehetőségként maga az alkalmazás is betöltheti a modulokat, és egy újraindítás után ugyanúgy láthatóvá válnak, mint a launcher esetében. Ez már garantálja a platformfüggetlenséget, de a használata kényelmetlen lehet, hiszen minden modul-konfiguráció módosításkor újra kell indítani az alkalmazást. Ráadásul ahhoz, hogy ezt automatizálni lehessen szintén platformfüggő kód szükséges, ellenkező esetben kézi indítást igényelne.

A nagyobb kihívást az előfordított kód igénye jelenti. A modulok hagyományos Java osztályokból állnak, vagyis a program futásakor lefordítva, bájtkódként rendelkezésre kell

állniuk. Kétféle megoldás jöhet szóba. Vagy a modul készítője fordítja le a kódot, vagy a felhasználónál automatikusan lefordul. Utóbbi lehetőséggel az a probléma, hogy valószínűleg feltétlenül szükséges hozzá a JDK telepítése minden kliens gépre. Ez irreális elvárás, és alternatív lehetőséget nem találtam. Az előbbi lehetőség azonban nem nagy elvárás, hiszen a modul írója feltehetően eleve valamilyen IDE-t használ a modul készítéséhez, így nem nagy lépés a kód lefordítása és jar fájlba tömörítése. De ez mindenképp egy kényelmetlenebb, és félreérthetőbb megoldás. Emlékezzünk vissza a fő fejezet legelején (3. A modularitásról) tárgyalt fogalmakra. A modul a keretprogram futásának menetét nem befolyásolja, hanem „fekete dobozokat” tartalmaz, a bemeneteket kimenetekké transzformálja. Ha a fejlesztők a modulokat Java nyelven írják, és azt lefordítva adják a programhoz, az arra sarkallhatja a fejlesztőket, hogy egyszerű függvények helyett komplexebb megoldásban gondolkodjanak. Ha ugyanezt JavaScript-ben teszik, az feltehetően más hozzáállást eredményez.

Azonban lehetőség van egy meglehetősen érdekes hibrid módszer alkalmazására is. A külső modulokat megírhatjuk valamilyen szkriptnyelv segítségével, a szkripteket pedig egy állandó Java 9 modul tölti be. Így megkapjuk a szkriptnyelvek összes előnyét a Java 9 modulok biztonságával.

A Java modulrendszerét alapvetően nem olyan célra találták ki, amelyre én használni szeretném – a pontos célokról és a működésről a következő fejezetben fogok írni. De ahogy azt a teljes „3.1 Technológiai lehetőségek” fejezetben tárgyaltam, könnyen használható, biztonságos, kész megoldás nincs erre a problémára. Önmagában a Java modulok sem jelentenek megoldást, de kiegészítve a szkriptnyelvekkel, illetve esetleg a SecurityManager-rel és egyedi ClassLoaderrel, elérhetjük a szükséges tulajdonságokat. Ezek közül a Java modulok a következőket garantálják nekünk:

- Azokat a csomagokat, amelyeknél ez expliciten nincs megengedve, nem lehet reflektíven elérni. Azonban különböző forrásokat olvasva arra eszméletem, hogy ez így nem teljesen igaz. A legtöbb forrás homályosan fogalmaz arról, hogy mit lehet pontosan elérni reflection segítségével, ha nincs expliciten megnyitva egy csomag. Ezt hiteles forrással sajnos nem tudom alátámasztani, én a következő definíciót fogalmazom meg: Ha egy csomag nincs expliciten megnyitva reflektív hozzáférésre, akkor annak a csomagnak csak és kizárólag a publikus tagjait lehet elérni reflektíven.[15]

- Csak és kizárólag azokat az osztályokat érheti el egy modul, amelyek olyan csomagban szerepelnek, melyek a tartalmazó modulban `exports` kulcsszóval, az igénylő modulban pedig `requires` kulcsszóval szerepelnek. Esetünkben az utóbbi a fontos. Ha a szkripteket tartalmazó modul nem fér hozzá pl. a `java.instrumentation` modulhoz, akkor az ebben található `java.lang.instrumentation.Instrumentation` osztály számára nem létezik, azt semmilyen módon nem tudja elérni és használni. Ezt a védelmet a következő fejezetben részletesebben is kifejtem.

Ezzel a két garantált védelemmel le tudjuk szűkíteni a biztonsági kockázatot. A saját moduljaink felett teljes befolyással bírunk, a projektünkön belül pontosan meg tudjuk határozni, hogy a szkriptek mikhez férhessenek hozzá, és mikhez ne. Azonban a `java.base` modulhoz kivétel nélkül minden más modul is hozzáfér (implicit módon). És miután ez a modul számos biztonsági kockázatot rejt magában (pl. kilépés az alkalmazásból, hálózati hozzáférés, fájlhozzáférés), sajnos a Java modul rendszere önmagában messze nem jelent elegendő védelmet.

3.2 A Java 9 modulrendszerről

Alapvetően a Java modul rendszer[16] egyszerűen használható, gyorsan tanulható. Azonban különböző okokból ezt az új technológiát nagyon lassan kezdték a fejlesztők használatba venni. Nem találtam erre vonatkozó adatokat, de az a gyanúm, hogy a Java modul rendszert még ma is kevesen használják – a Java felhasználási köréhez képest. A lehetséges okokról a következőkben a saját elméletemet vázolom.

Az egyik ok az IDE támogatás lassú kialakulása. Amint kiadásra került a Java 9 – 2017 szeptemberében – én elkezdtem foglalkozni vele. Érdekelt, hogy ez a technológia, amely alapjaiban változtathatja meg a Java használatát, hogyan néz ki a gyakorlatban. De azzal szembesültem, hogy a három nagy IDE közül a Netbeans egyáltalán nem támogatta, az Eclipse akkortájt kezdte a fejlesztését, az IntelliJ IDEA pedig részlegesen támogatta. A három IDE közül csak az IDEA volt az, amivel bele lehetett kezdeni a modulrendszer használatába.

Azonban nem az IDE támogatottság hiányossága okozta az egyetlen gondot. Azoknak a keretrendszereknek, amelyek az egyes alkalmazások fejlesztéséhez szükségesek, szintén későn kezdték meg a támogatás fejlesztését. Én nevezetesen a Maven-t és a Spring-et használtam – ezekről a későbbi fejezetekben a felhasznált technológiák kapcsán írok –,

egyik sem fért össze problémamentesen a Java modulrendszerrel. És mit csinál az ember, ha a fejlesztés során falakba ütközik? Rákeres a problémára interneten.

És ezzel el is érkeztünk a harmadik problémakörhöz. Mivel a technológia új volt, nem volt IDE támogatás és a keretrendszerek sem fértek könnyedén össze vele, kevesen vették a fáradságot, hogy a Java modulrendszerben fejlesszenek. Ez pedig azt eredményezte, hogy sok kérdésre nem lehetett a választ megtalálni az interneten, illetve a keretrendszerek, libek fejlesztőihez is kevesebb visszajelzés jutott el.

Ezzel ki is alakult az ördögi kör. Nehéz volt benne fejleszteni, ezért kevesen fejlesztettek benne, ennek hatására pedig nehéz is maradt benne fejleszteni. Ahhoz, hogy kipróbáljam, a gyakorlatban hogyan férnek össze ezek a technológiák, elkészítettem egy példa projektet. Többszöri próbálkozásra sikerült elkészítenem egy példát, amelyben a Spring, Maven és a Java modulrendszer együtt tud működni. Ezt az 1. Melléklet-ben mellékelem, illetve a következőkben ezzel fogom szemléltetni a Java modulrendszer működését.

3.2.1 A Java modulok működése

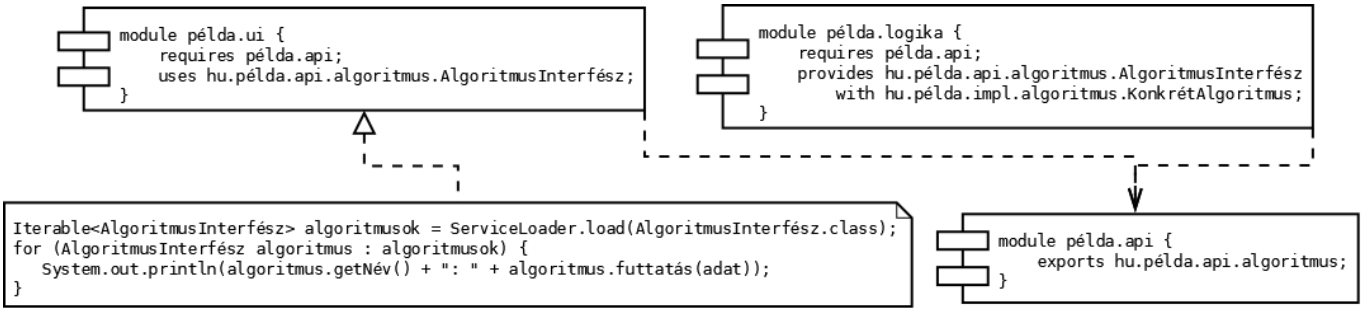
Tehát hogyan is működnek a Java modulok? A Java modularitás egy új, magasabb csoportosítási szintet ad a csomagok (package-ek) fölé. A modul egy egyedi elnevezésű, újrafelhasználható egysége a csomagoknak. Minden modulhoz tartozik egy modul leíró (deszkriptor), amely tartalmazza a modul nevét, függőségeit (mely más modulokra van szüksége), exportjait (mely csomagokat biztosítja a külvilág felé), reflektív nyitottságát (mely más modulok számára engedélyezi a reflektív hozzáférést), illetve a nyújtott és kért szolgáltatásokat.

A szolgáltatás-orientált megoldást e dolgozatban nem fogom alkalmazni. Egyrészt azért, mert ezt a célt a Spring már ellátja, ráadásul kényelmesebben. Másrészt azért, mert a Java modulrendszert eleve kevesen használják, és kevés róla az információ, ez pedig a szolgáltatás rendszerű megoldásra hatványozottan igaz. Mindazonáltal a teljesség kedvéért az alábbiakban ezt is bemutatom.

Egy modul szolgáltatásai összességén gyakorlatilag a modul API-ját értjük. Ez definiálja, hogy hogyan lehet ezzel a modullal kommunikálni, milyen formátumú üzenetet fogad, és milyen formátumban válaszol ezekre. Ezt a viselkedést hagyományosan interfészekkel valósítjuk meg. Ha nem alkalmazunk magasabb absztrakciós szintet, akkor ez a következőképpen néz ki:

```
PéldaInterfész példaInterfész = new KonkrétPéldaOsztály();
```

Ennek a módszernek számos hátránya van, ezekkel minden fejlesztő tisztában van. A módszer néhány gyengeségét, pl. az erős kapcsolatot az osztályok között, a *Factory design pattern* segítségével ki lehet küszöbölni. De a Factory és a gyártott osztályai között továbbra is megmarad az erős kapcsolat. Itt jön képbe a Java modulrendszer. A szolgáltatást igénylő modul csak az interfészt látja, a konkrét implementációkat nem. Az interfész és az implementációk között a kapcsolatot a modulrendszerrel együttműködő `ServiceLoader` osztály teremti meg.



3. ábra: Egyszerű példa a Java modulok kapcsolatára, felhasználására, ideértve a Java modulrendszer szolgáltatás-orientált megoldását is

Az ábrán látható, hogy a `példa.api` modul exportál egy csomagot a `hu.példa.api.algoritmus`-t, ez tartalmazza az `AlgoritmusInterfész`-t. A `példa.logika` modul függőségként megjelöli a `példa.api` modult, így számára elérhetővé válik az `AlgoritmusInterfész`, és erre az interfészre ad egy implementációt, mint szolgáltatást. A `példa.ui` modul szintén megjelöli függőségként a `példa.api` modult, hogy számára is elérhetővé váljon az `AlgoritmusInterfész`, és ezt mint szolgáltatást használatba is veszi. A használat az ábrán látható módon történik. A `ServiceLoader` (Java bépített) osztálytól lekérjük az összes elérhető `AlgoritmusInterfész` implementációt, majd meghívunk rajtuk metódusokat.

Nos, ennyit a Java modulrendszer szolgáltatás-orientált megoldásáról, a továbbiakban ezzel nem foglalkozom, helyette a Spring által nyújtott dependency injection (DI) megoldást fogom alkalmazni. További információk pl. ezen a címen találhatók: https://www.oreilly.com/library/view/java-9-modularity/9781491954157/ch04.html. De a fenti ábra a szolgáltatásokon túlmenően is hasznos. Figyeljük meg a következőket:

- A `példa.ui` modul a `requires` kulcsszóval jelzi a függőségét.
- A `példa.api` modul az `exports` kulcsszóval jelzi, hogy mely csomagot teszi láthatóvá a külvilág számára.
- Ennél az ábránál igyekeztem egyszerű és jól érthető neveket adni a moduloknak, de a gyakorlatban a modulokra általában hasonló elnevezési konvenciókat használunk, mint a csomagokra, ezért ezeket könnyű összekeverni. Mégis így teszünk, több okból is:
  - a hivatalos álláspont szerint ez erősen ajánlott, vagyis a modul neve egyezzen meg a csomagok gyökerével,
  - a modulrendszer fejlesztői úgy döntöttek, hogy a csomagnevekhez hasonlóan a modulneveknek is célszerű globálisan egyedinek lenni,
  - a modul csomagok gyűjteménye, tehát logikus, hogy a csomagokhoz kapcsolódó nevet válasszunk,
  - egy modul egy csomagnévért, és az azalatt lévő csomagnevekért felelős (ez így nem teljes és pontos, a „4.4.1.1 Split Package problémakör” fejezetben erre még visszatérek).
- Figyeljük meg azt is, hogy a `példa.ui` modulnak látszólag egyetlen függősége van, ez pedig a `példa.api`. Mégis a `példa.ui` modulban található kódban több osztályhoz is hozzáférünk, mint: `Iterable`, `ServiceLoader`, `System`. Ezek az osztályok nem csak az éterből jönnek, ezek a `java.base` modulban találhatóak. A `java.base` modult pedig minden modul implicit módon megkapja függőségként.

Mielőtt belevágnánk egy életszerűbb példába, ismerkedjünk meg a Java modulok fajtáival. Háromféle modul van. Az egyikkel már találkoztunk fentebb is, ezek a named, vagyis nevesített modulok. Onnan lehet őket felismerni, hogy van module-info.java fájl hozzájuk rendelve.

A másik fajta modul az unnamed, vagyis nem nevesített modul. Ez a Java 9 előtt írt java alkalmazások és könyvtárak kódjait tartalmazza. Azokhoz a csomagokhoz, osztályokhoz, amelyek a Java 9 előtt készültek természetesen nem kapcsolódnak module-info fájlok, így alapvetően nincs nevük. A nem nevesített modul úgy áll össze, hogy a Java fordító elemzi, mely más modulokra van szükség, és az összes ilyet automatikusan függőségnek jelöli meg. Valamint a nem nevesített modul az összes csomagját automatikusan exportálja. Ez a megoldás garantálja azt, hogy a Java 9 előtt készített

alkalmazások a Java 9 és újabb verziókkal is fordíthatók, futtathatók. Ugyanakkor egy nevesített modul a nem nevesített modult nem hivatkozhatja meg függőségként. Nincs lehetőség ilyen kifejezést használni, mint pl. „`requires unnamed`”, így egy nevesített modul a névtelen modult nem tudja használatba venni. Tehát a két világ együtt dolgozni nem tud.

Megoldást a harmadik típusú modul jelent, ez az automatikus modul. A moduláris alkalmazásokat *class path* helyett *module path*-ban indítjuk el. Ha a module path-ba olyan jar-okat is behúzunk, amelyek nem tartalmaznak module-info fájlt, akkor automatikus modul keletkezik minden ilyen jar-hoz. Az automatikus modul neve a jar nevéből lesz generálva (nagy vonalakban) úgy, hogy a kötőjelek ponttá változnak, a verzió szám pedig eltűnik. A példa-library-1.2.1-SNAPSHOT.jar fájlból a `példa.library` automatikus modulnév adódik. Alternatívaként a jar manifest fájljában manuálisan is meg lehet adni ezt a nevet. A manuális megadás azt a célt szolgálja, hogy egy libet minimális munkával elegánsabban be lehessen illeszteni a modulrendszerbe. Egy automatikus modul minden más nevesített és automatikus modult megjelöl függőségként, illetve minden csomagját exportálja. A gyakorlatban az automatikus modulokat úgy vesszük igénybe, hogy legalább egy nevesített modulban meghivatkozzuk a generált nevét.[17][18]

A fenti egyszerű példa után nézzünk meg egy valósághűbbet. Ezt a példa modult az 1. Melléklet-ből emeletem ki.

```
module hu.kleatech.jigsaw {
    requires spring.boot;
    requires spring.boot.autoconfigure;
    requires spring.beans;
    requires java.sql;
    exports hu.kleatech.jigsaw to spring.core, spring.beans, spring.context;
    opens hu.kleatech.jigsaw to spring.core;

    requires hu.kleatech.jigsaw.api;
}
```

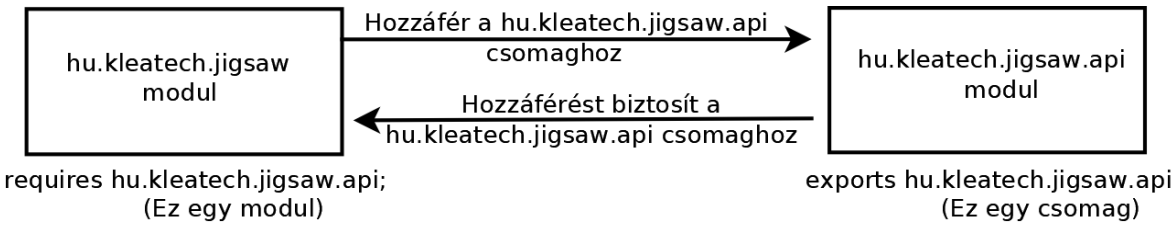
Az első sorban a modul neve látható. A modul neve ebben az esetben megegyezik a modulban található csomag nevével, ahogy azt fentebb már jeleztem. A következő négy sorban a Spring függőségek láthatóak. Ennek a modulnak szüksége van a `spring.boot`, `spring.boot.autoconfigure`, és a `spring.beans` modulra. Ez a három automatikus modul. A `java.sql` modulra közvetlenül nincs szükség, azt valamely Spring modul használja; ezt normál esetben nem kellene beleírni a modul deszkriptorba, de ahogy már korábban

említettem, a Spring és a Java 9 nehezen férnek össze, és ez például így nyilvánulhat meg. Annak pontos oka, hogy erre a modulra miért van szükség, számomra ismeretlen. Egy lehetséges – bár kevésbé valószínű – magyarázat, hogy a Spring Boot, melyhez speciális fordítóeszköz is tartozik, a fordításkor olyan kódot illeszt be a forrásfájlba, amely már függ a `java.sql` modultól.

Az `exports` kulcsszó sorában látható, hogy ez a modul a `hu.kleatech.jigsaw` package-hez hozzáférést biztosít a Spring számára. Normál esetben nem szokás kijelölni, hogy egy modul csak egy-egy bizonyos másik modul számára adjon hozzáférést, erre is a Spring keretrendszer sajátosságai miatt van szükség, egyébként ezt a modult nem kívánjuk publikálni, ez csupán összefogja az alkalmazást.

Az `opens` kulcsszó sorában látható, hogy a modul a `hu.kleatech.jigsaw` package-et megnyitja a `spring.core` modul számára. Ezzel a modul reflektív hozzáférést biztosít a Spring számára, erre a dependency injection miatt van szükség.

És végül az utolsó sorban látható egy szokványos függőség: ennek a modulnak szüksége van a `hu.kleatech.jigsaw.api` modulra. Ez a modul csak akkor fordul le, ha a `hu.kleatech.jigsaw.api` modulban van legalább egy nyilvános `exports` kulcsszó.



4. ábra: Függőségi reláció a Java modulok között

A java modulrendszer működésének ismertetését ezzel le is zárom. A modulrendszer sajátosságaira és e dolgozat céljaira történő felhasználására a következő fejezetben lesz szó, az implementáció kapcsán.

4. IMPLEMENTÁCIÓ

Ebben a fejezetben a dolgozat címét adó problémakör gyakorlati, tényleges megvalósításáról, illetve a példaalkalmazás elkészítését érintő dolgokról fogok írni. Ide értve többek között:

- a külső logika bekapcsolásának helyét, módját,
- a frontend kívülről történő bekapcsolásának helyét, módját,
- a különböző célokra felhasznált technológiákat,
- a példaalkalmazás célját, felépítését, működését,
- az implementáció közben adódó kihívásokat, akadályokat,
- és a biztonság gyakorlati megvalósításának lehetőségeit.

4.1 Felhasznált technológiák

Az alábbi alfejezetekben összefoglalom, hogy a példaalkalmazáshoz milyen technológiákat használtam fel, melyek lettek volna az alternatívák, és miért az adott technológiára esett a választás. A dolgozat terjedelme nem teszi lehetővé az egyes technológiák ismertetését, így itt csak említés szintjén fognak szerepelni. Minden technológiához biztosítok forrásokat, amelyek a megfelelő választásban segítettek, illetve az olvasó érdeklődését betölthetik. A forrásokat a Szójegyzékben mellékelem.

4.1.1 IDE

A felhasznált technológiák kapcsán először a választott fejlesztőkörnyezetről, vagyis az IDE-ről szeretnék írni. A teljesség kedvéért szerepeljen itt az összes szóba jöhető IDE: IntelliJ IDEA, Visual Studio Code, Eclipse, Atom IDE, Netbeans.

A felsoroltak közül a VS Code és az Atom IDE ún. lightweight IDE-k. Ezek jelentősen kevesebb támogatást nyújtanak társaiknál. Ilyen feladatra pedig, mint a java modulrendszer használatba vétele, minden támogatásra szükség van. Ezért ezen két IDE használata valójában fel sem merült.

A három nagy IDE közül e sorok írásakor mindegyik támogatja valamilyen szinten a java modulrendszert. Azonban amellett sem mehetünk el szó nélkül, hogy amikor az

Oracle kiadta a Java 9-et, 2017 szeptemberében, ekkor milyen támogatottság volt, hiszen ekkor kezdtem el a modulrendszerrel foglalkozni. Ahogy azt korábban említettem, a Netbeans ekkor egyáltalán nem támogatta a modulrendszert, a Eclipse pedig leginkább csak tervezgette az implementálását. Így az IntelliJ IDEA-t választottam, amelyet tanulói licensszel ingyenesen igénybe vehettem.

Az IntelliJ IDEA támogatása 2017-ben még gyerekcipőben járt, de használható volt, és folyamatosan javítottak rajta. De 2018 végén még mindig nem voltam vele teljesen megelégedve. Úgy éreztem, az IDEA által használt projektstruktúrába a Java modulrendszer nem illeszthető bele természetesen, a fejlesztők bele erőszakolták, hogy az IDE ezt is támogassa. Az IDEA a Java modulrendszerrel együtt már három különböző modulrendszert használ egymással párhuzamosan, egy projekten belül. A Maven-féle modulokat, a Java modulokat és az IDEA saját moduljait. Ennek a hármasnak mind a létrehozása, mind a karbantartása körülményes.

A fentiek okán nagyon megörültem, amikor megtudtam, hogy a – most már Apache kézben lévő – Netbeans következő verziója, az évek óta várt 9-es kiadás elkészült 2018 júliusában, év végén pedig a Netbeans 10-et is kiadták. Számomra mindig is a Netbeans volt a legszimpatikusabb Java IDE.

Az Eclipse-ről számos negatív tapasztalatot hallottam, és személyes negatív tapasztalataim is voltak vele, így nálam csak a Netbeans és az IDEA verseng a legjobb IDE címért. Az IDEA-nak – fizetős szoftver lévén – jobb a támogatottsága, egyesek szerint kényelmesebben használható és praktikusabb, mint a Netbeans. Ez azonban azt is jelenti, hogy az IDEA kínált funkciói sokkal gyorsabban bővülnek, az IDE gyorsan fejlődik.

A gyors fejlődés, bővítés pedig sok esetben alacsonyabb megbízhatóságot, elhamarkodott döntéseket von maga után. Erre példa a fent említett háromféle modulrendszer, amelyeket egyszerre kell használni. További negatívum számomra, hogy az IDEA-t „túl kényelmessé” szándékozták tenni, ezért néhány olyan felhasználói felülettel kapcsolatos döntés született, amely sokaknak okozott igen kellemetlen élményeket. Példaként a CTRL+Y billentyűkombináció esetét tudom felhozni. Az operációs

rendszerekben ez a billentyűkombináció általában a „redo” vagyis újbóli végrehajtás, vagy a visszavonás visszavonása műveletet jelenti. Az IDEA-ban ez a billentyűkombináció az adott sor törlését jelenti. Ez pedig azt jelenti, hogy a CTRL-Y tévedésből történő lenyomása az undo stack törlésével jár, vagyis elég jelentős mennyiségű munka mehet kárba miatta. Ez olyan szinten feldühített néhány felhasználót, hogy kizárólag emiatt a dizájn döntés miatt nem használták tovább az IDEA-t.[19]

A Netbeans-t az IDEA-val szemben lassabb ciklusban fejlesztik, valamivel kevesebb funkciót tartalmaz, és egyszerűbb felületével néhányan kevésbé felhasználóbarátnak tarthatják. De ezek a tulajdonságok éppen azt eredményezik, hogy egy könnyen, természetesen használható, nagyon megbízható IDE-t kapunk. Nem mindenhez ad támogatást, de amihez igen, az jól működik. Az eddig tapasztaltak alapján a Java modulrendszer támogatása is remekül működik. Annyit azonban érdemes megemlíteni, hogy a Netbeans az Apache Software Foundation-nél még „incubating” állapotban van. Azok a szoftverek kerülnek ilyen állapotba, amelyek még nem teljes támogatást élvező Apache szoftverek, de jó úton járnak afelé, hogy azok legyenek. Úgy gondolom, hogy a Netbeans nemsokára teljes értékű Apache szoftverré fog válni.

A választott IDE: Netbeans.

4.1.2 Nyelv, platform

Az a dolgozat korábbi részéből már nyilvánvalóvá vált, hogy a témához és a példaalkalmazáshoz Java nyelvet és a Java platformot fogom használni. Ebben a részben ezt szeretném megindokolni.

A Java nyelvet és platformot azért választottam, mert számomra fontos a platformfüggetlenség. Egyrészt azért fontos, mert megakadályozza az egyes felhasználói körök diszkriminációját, másrészt támogatja a szabad versenyt és az igazságosabb, kiegyenlítettebb piaci helyzetet. Ez utóbbi a fő ok, amiért feltétlenül platformfüggetlenségben gondolkodom, úgy gondolom, hogy a nagy cégek egyeduralma, monopóliuma súlyos következményekkel fog járni a jövőben, és már ma is rendkívül igazságtalan.

Azonban a platformfüggetlen nyelvek között is van választék. Jól bevált, jó pár éve létező nyelvet szerettem volna választani, amelynek nagy a felhasználói tábora és támogatottsága. Modern szemléletű nyelvet akartam, amely támogatja az objektumorientált programozást és a funkcionális programozási paradigmát. Olyan nyelvet kerestem, amelyet enterprise rendszerek fejlesztésére is használnak, és komoly, több mindent átfogó, jól átgondolt keretrendszerek tartoznak hozzá. Ezen feltételeknek a következő nyelvek felelnek meg: Java, Python, C#, JavaScript.

A C#-ot kizárhatjuk a platformfüggősége miatt. Ugyan a .NET Core platform már több operációs rendszeren is használható, ez még meglehetősen korlátozott, és a támogatottságával sem vagyok megelégedve.

A JavaScriptet manapság már enterprise alkalmazások készítésére is használják, és megfelelő keretrendszerek is léteznek hozzá. Ugyanakkor az IDE támogatottsága (részben a dinamikus típusossága miatt és a prototípus alapú osztályrendszere miatt) számomra nem megfelelő, a kódkiegészítés egyik IDE-ben sem kielégítő. Ezen kívül a JavaScriptet nem tartom igazán alkalmasnak enterprise szoftverek backend komponenseinek fejlesztésére, véleményem szerint kevésbé átláthatóbb, több rejtett hibát tartalmazó kódot eredményez, és a performanciát is csökkenti. Így a JavaScript az igényeimnek nem felel meg.

A Python-t potenciálisan alkalmasnak találom a céljaimra. Ugyanakkor Python-ban semmiféle tapasztalatom nincs, így ehhez a szakdolgozathoz nem jöhetett szóba. De látok rá lehetőséget, hogy e dolgozat témáját Python-ban is meg lehet valósítani.

Marad tehát a Java nyelv. A platform tekintetében a Standard Edition-t fogom használni. Azon belül Oracle JDK-ban kezdtem el a fejlesztést Windows 10 operációs rendszer alatt, de technikai okokból át kellett térnem Linuxra, és egyúttal váltottam OpenJDK-ra. Az Oracle nemrégiben történt licenszfeltétel módosításai után nem kizárt, hogy idővel ebből az okból is váltani kellett volna. Ettől függetlenül is végig szem előtt tartom, hogy amit leírok és felhasználok, az működhessen tetszőleges JDK-n ill. JRE-n is.

A Javán belül a szakdolgozat témájához fontos a funkcionális programozás (Java 8) és a modularitás (Java 9). Így a Java nyelv és platform verzióját tekintve 9-es vagy magasabb verziót használok.

A választott nyelv: Java. A választott platform: Java SE, Oracle JDK és OpenJDK.

4.1.3 Keretrendszerek

Ebben az alfejezetben azokat a lényeges keretrendszereket foglalom össze, amelyekre a példaalkalmazás épül. Ezen dolgozat célja az is, hogy bemutassa, ezek a keretrendszerek hogyan tudnak együtt dolgozni a Java modulrendszerrel, illetve egymással a modulrendszerben. Illetve célom azt is bemutatni, hogy a keretrendszerek hogyan támogatják a külső kód beillesztését az alkalmazásba, akár futás közben. Ahogy az a dolgozat későbbi részében látszódni fog, a programlogika esetén ez a keretrendszerek szempontjából triviális, de a felhasználói felület darabjainak futás közben történő lecserélése már izgalmasabb feladat.

4.1.3.1 Maven

Ezen projekt szempontjából a Maven fordítási folyamat automatizálási funkciója a lényeges, melyet leegyszerűsít és egységesít. A Spring Boot esetében ez azt jelenti, hogy egy néhány soros Maven pom.xml fájlt szükséges csak megírni, és a projekt egyetlen egyszerű paranccsal indítható is. A pom.xml elkészítése után az alkalmazás az `mvn install` paranccsal fordítható, és az `mvn spring-boot:run` paranccsal indítható. Semmi több nem szükséges hozzá. Alternatíva a Gradle lehetne, de abban nincs tapasztalatom, és az általános vélemény szerint kevésbé vagy körülményesebben támogatja a Java modulrendszert, mint a Maven.

4.1.3.2 Spring Boot (Spring Data, MVC, stb.)

A Spring keretrendszerből számos technológiát felhasználok, melyekből itt a lényegesebbek említem meg. A Spring DI konténerét használom a függőségek (komponensek) példányosítására. Automatikus komponens szkennelést használok az `@Autowired` annotációval. A letisztultsága miatt field injection-t használok. Spring Boot-ot

használok, hogy az alkalmazás konfigurációja (AutoConfiguration) és karbantartása egyszerűbb legyen. A Spring Boot beépített webszerverét használom, mert a példaalkalmazás egyik elsődleges célja az, hogy egyetlen futtatható állományból álljon.

Ugyanezen okból kifolyólag beépített adatbázis kezelőt használok. Három beépíthető és a Spring Boot által közvetlenül támogatott DBMS-t találtam: H2, HSQL, Derby. Mindhárom megfelel a célnak, de a H2-t használtam eddig, és ez a legújabb, legfrissebb, így ezt választottam.

Az adatkezelésre Spring Data-t használok. A Spring Data rendkívüli módon leegyszerűsít az adatkezelést. Adattárolásra relációs adatbázist használok. A relációk leírását a JPA szabvány keretein belül annotációkkal végzem. Alternatívaként az XML alapú konfiguráció jöhetne szóba, de az annotáció alapú véleményem szerint sokkal kényelmesebb és átláthatóbb. A Spring Data a háttérben használ valamilyen ORM keretrendszert, ezek közül a Hibernate az alapértelmezett. A Hibernate széles körben elterjedt keretrendszer, nem láttam okát lecserélni, illetve ezen kívül más ORM keretrendszert még nem használtam. Mindazonáltal alternatívák vannak, pl. Eclipse Link. A Spring Data az alatta lévő ORM keretrendszert elfedi, így az egyszerű esetekben valódi jelentősége nincs, hogy melyiket használom. A lekérdezéseket többféle módon is meg lehet adni, pl. `@Query` annotációval, egyedi Repository-val, és Query by example-el, de én az egyszerűség kedvéért a Spring Data saját lekérdezőnyelvét fogom használni, amely metódusnevek alapján generálja a lekérdezéseket. Ennek a használata igényli a legkevesebb előkészületet, ugyanakkor korlátozott. A példaalkalmazásban egy-egy esetben kénytelen voltam a `@Query` annotációt, illetve az egyedi Repository-t is használni.

A Spring Web MVC keretrendszert használom a HTTP *kérések* feldolgozására. A Web MVC-t elsősorban a HTTP kérések paramétereinek feldolgozására, a view-k példányosítására (*view resolving*), illetve az MVC model és a session kezelésére használom.

4.1.3.3 Thymeleaf, Bootstrap, Jquery

A példaalkalmazáshoz egyszerű frontendes technológiákat szerettem volna választani, olyanokat, amelyeket jelenleg is sokan használnak, modernek és jó a támogatottságuk. Az is fontos szempont, hogy a többi technológiával könnyen együtt tudjanak dolgozni. A felhasználó felület megvalósítására három lehetőséget tudtam elképzelni: asztali alkalmazás, REST kliens és template alapú (vagyis szerveroldalon generált weboldal).

A REST kliens egy plusz komplexitást vinne az alkalmazásba, amit szerettem volna elkerülni, így ezt kizártam – mint később kiderült, talán elhamarkodottan. Sokan úgy gondolják, hogy az asztali alkalmazások kihalófélben vannak, és a webalkalmazások teljesen átveszik a helyüket. Én ezzel nem értek egyet, úgy gondolom, az asztali alkalmazásoknak mindig is meglesz a helyük. Ugyanakkor ehhez az alkalmazáshoz megfelelőbbnek találtam a template alapú megoldást. Elsődlegesen azért tartom ezt előnyösnek, mert a felhasználói felület egyes elemeit is kívülről kell az alkalmazásba beemelni, ráadásul futás közben. Amennyiben ezt úgy valósítom meg, hogy HTML kódot cserélgetek, azzal jelentősen le tudom egyszerűsíteni a problémát, nem szükséges egyedi megoldást kitalálni hozzá.

A template-k kezelésére a Thymeleaf-et választottam, mert modern, jó a támogatottsága és tökéletesen együttműködik a Spring keretrendszerrel, ez nagyban megkönnyíti a használatát esetünkben. Alternatíva lehetett volna pl. valamilyen JSP-re épülő technológia, mint a JSF, de abban kevesebb tapasztalatot szereztem és – ugyan a mai napig fejlesztik – régebbi technológia a Thymeleaf-nél. A Thymeleaf több template leíró nyelvet támogat, én az XHTML-t választottam. Ez gyakorlati szemszögből annyit jelent, a template leíró részleteket „th” namespace alias-al kezdődő attribútumok formájában adom meg. Ezen túl a Thymeleaf elég megengedő, a kigenerált weboldal tartalmazhat HTML5 attribútumokat is. A példaalkalmazásban betöltött szerepe miatt külön kiemelném a fragment-eket. Ezek weboldal-töredékek, maguk is template-ek. A fragment-ek lehetnek külön fájlokban, és ezeket a Thymeleaf a template feldolgozása során összerakja. Ez egy remek lehetőség arra, hogy a frontend-et modulárissá tegyem, a fragmentek ugyanis tetszőleges helyről betölthetők.

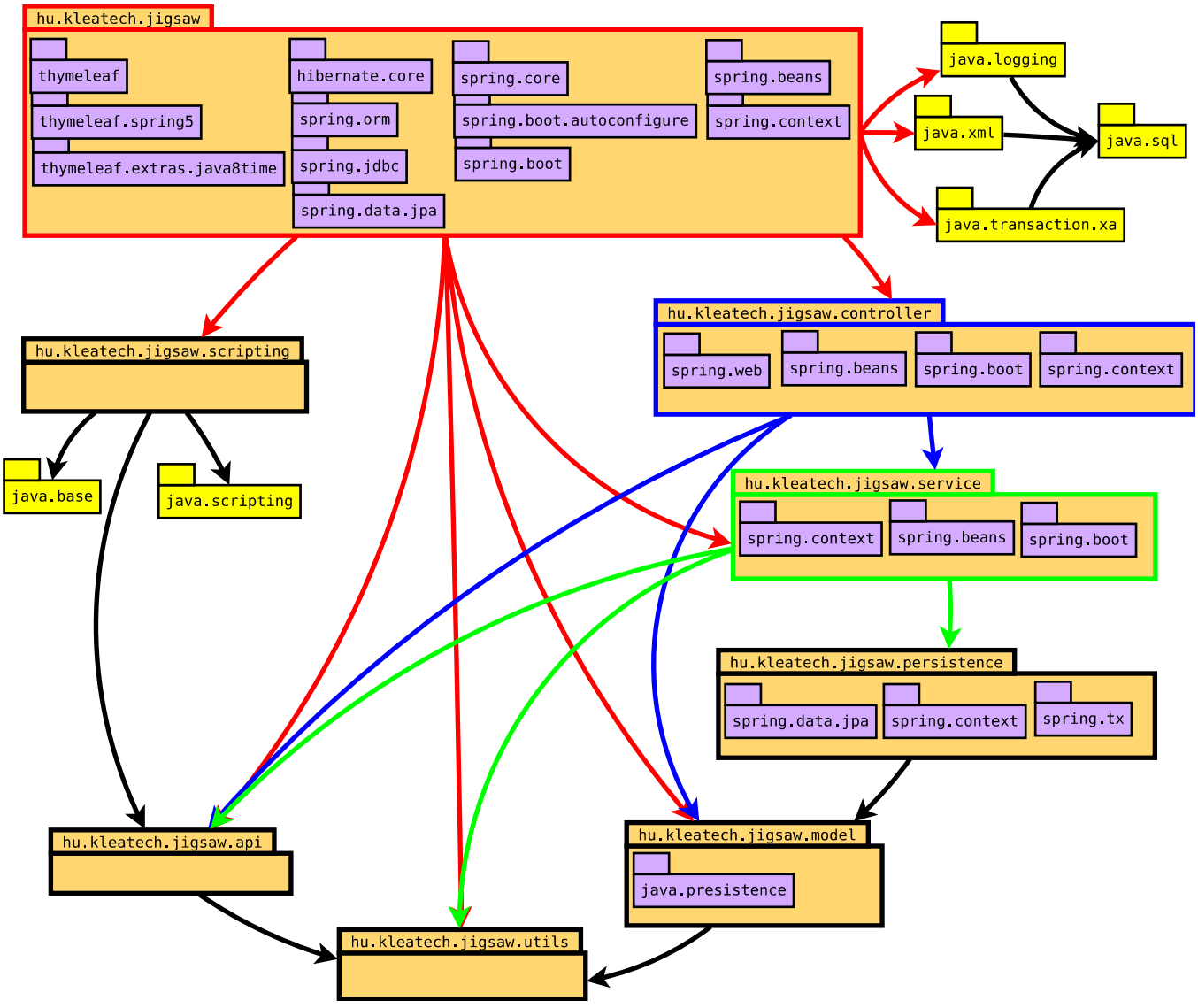
A Bootstrap a legnépszerűbb CSS könyvtár reszponzív weboldalak készítésére, az egyszerűsége miatt választottam. Az elemek elrendezésére a Bootstrap Grid rendszerét használom. Használhatnék Flexbox-ot is, az egyébként a Bootstrap-pel is együtt tud működni, de nincs benne tapasztalatom, és a Grid rendszer mellett nem éreztem szükségét. Ugyanakkor az implementáció során találkoztam olyan szituációkkal, amelyekben hasznos lehetett volna, így megfontolom a legközelebbi alkalmazásomban a használatát. Bootstrap-ből az e sorok írásakor létező legfrissebb verziót használom, a 4.3.1-et. A friss verzió a *Toast* funkció használata miatt vált szükségessé.

A DOM struktúra bejárására és a HTTP kérések kiadására a jQuery-t választottam. A jQuery rendkívül népszerű, egyszerű a használata, és minimális overhead-et jelent, ezért esett rá a választásom. Nincs tudomásom arról, hogy a jQuery-nek lenne valódi alternatívája.

4.2 Struktúra, kapcsolódási pontok

Ebben a fejezetben az alkalmazás felépítését és működését fogom bemutatni technikai szemszögből.

4.2.1 Modul struktúra



5. ábra: Modul diagram

Az 5. ábrán látható az alkalmazás teljes modul függőségi gráfja a Java modulok szemszögéből. Az *A* modulból *B* modulba mutató él azt jelöli, hogy *B* modul az *A*

modulnak függősége. A modul függőségi gráfnak tilos körutat tartalmaznia, ennél az alkalmazásnál ezt nehéz volt megvalósítani, ennek az okát és feloldását később ismertetem.

A citromsárgával jelöltek a Java beépített moduljai. A `java.base` modul implicit módon minden másik modulnak függősége, ez az ábrán nem látszik. De a `hu.kleatech.jigsaw.scripting` modul esetén ezt külön jelöltem, mert így tisztán látható, hogy a szkripteket futtató modul csak és kizárólag a `java.scripting` és a `java.base` modulhoz fér hozzá, illetve a `hu.kleatech.jigsaw.api`-hoz, máshoz semmilyen módon sem.

A lila színnel jelölt modulok automatikus modulok. Az automatikus modulokat korábban már ismertettem, ezek a .jar fájlok neveiből generált nevű, nem moduláris libek. Az automatikus modulok nem definiálnak függőségeket, így az átláthatóság kedvéért ezeket a függőségi gráfon kívül, az egyes modulokhoz rendelten ábrázoltam.

A barna színnel jelölt modulok a saját modulok. Ezeknek a szerepéről és viszonyáról a következő alfejezetben írok. A saját modulok neveit a következőkben az olvashatóság kedvéért rövidítem, csak az utolsó tagjukat írom le, pl. `hu.kleatech.jigsaw.scripting` helyett `scripting`; `hu.kleatech.jigsaw` helyett `jigsaw`.

4.2.1.1 Saját modulok szerepe

A hierarchiában legfelül a `jigsaw` modul szerepel, ennek az a feladata, hogy összefogja és elindítsa az alkalmazást. Ebben található a main metódus. Ez a Spring Boot belépőpontja, így itt található a program konfigurációjának nagy része. A konfiguráció egyrészt áll az application.properties-ből, ebben van beállítva a H2 adatbázis és részben a Thymeleaf. Másrészt itt található a `@Configuration` annotációval ellátott osztály is, amelyben további konfiguráció adható meg. A konfigurációs osztályban beállítok az alapértelmezett mellett egy második Thymeleaf template resolvert, amely fájlból dolgozik. Így az egyik resolver a weboldal keretét és közös részeit adja, ezek a classpath-on találhatóak, a másik resolver pedig a weboldal dinamikusan változó részét adja, amely kívülről töltődik be. A gyorsítótárazás ki lett kapcsolva, ezzel meg is valósul a dinamikus cserélhetőség. A `jigsaw` modulnak tehát olyan függőségei vannak, amelyek a konfigurációért felelősek, vagy abban érintettek. Ezen kívül ide kapcsolódik a `spring.core` és a `spring.boot`, hiszen ez a Spring

Boot kiindulópontja, illetve a `spring.context` és a `spring.beans`, ezek a Spring-es annotációkért felelősek, szinte minden Spring-es modulban használatosak. Ide kapcsolódik a `service`, `persistence` és `model` is, de ezek csak tesztelési célból, a kiadott alkalmazásban erre nem lesz szükség. Ide kapcsolódik továbbá a `scripting` és az `api` is, ennek okát majd külön tárgyalom. És végül a `controller`, amely a `jigsaw`-tól átveszi a vezérlést.

A `controller` modul a HTTP kérések fogadásáért és feldolgozásáért felel. Egy-egy kéréshez egy-egy metódus tartozik, ami azt feldolgozza, és vagy egy view-t küld vissza, vagy valamilyen válaszüzenetet. A `controller` modul kéri le a `scripting` modultól a szkripteket (az `api` modulon keresztül), ennek okáról a következő fejezetben írok. Illetve tartalmaz még a frontend-hez kapcsolódó utility osztályokat, metódusokat. A `controller` használja még a `spring.web` modult nyilvánvaló okokból. Az előfeldolgozott kéréseket pedig a `service` modul felé továbbítja.

A `service` modul tartalmazza a backend logikáját. Elsősorban adatkezeléssel kapcsolatos kéréseket kap a `controller`-től, ezekkel műveleteket végez, majd továbbítja a kérést a `presistence` felé. A `service` modult használom arra, hogy a külső szkriptek felé támogatást – segédosztályokat, metódusokat biztosítsak, habár ez csak bemutató célt szolgál, a példaalkalmazás ezt a lehetőséget nem használja ki.

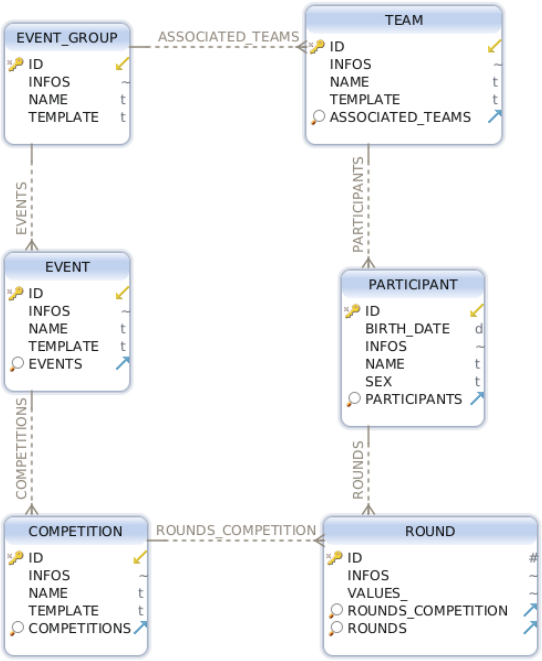
A `persistence` modul képviseli az adatelérési réteget. Ebben a Spring Data keretrendszerbe illesztett interfészek találhatóak, a logikát a Spring Data generálja le. Illetve az alkalmazáshoz szükség volt egy egyedi kiegészítésre is az egyik generált interfészhez, ez is itt található. A `persistence` modul függősége a `spring.tx` modul, amely a tranzakciókezelést végzi.

A logikai hierarchia alján a `model` modul található, amely a JPA szabványnak megfelelő entity (egyed) osztályokat tartalmazza, ehhez szüksége van a `java.persistence` automatikus modulra. Ez azért automatikus modul, mert nem a Java SE, hanem a Java EE része. Ezen osztályok a bennük lévő adattagokkal és az azokon lévő annotációkkal egyetemben megadják az ER modellt. A diagramot lásd a későbbi fejezetben.

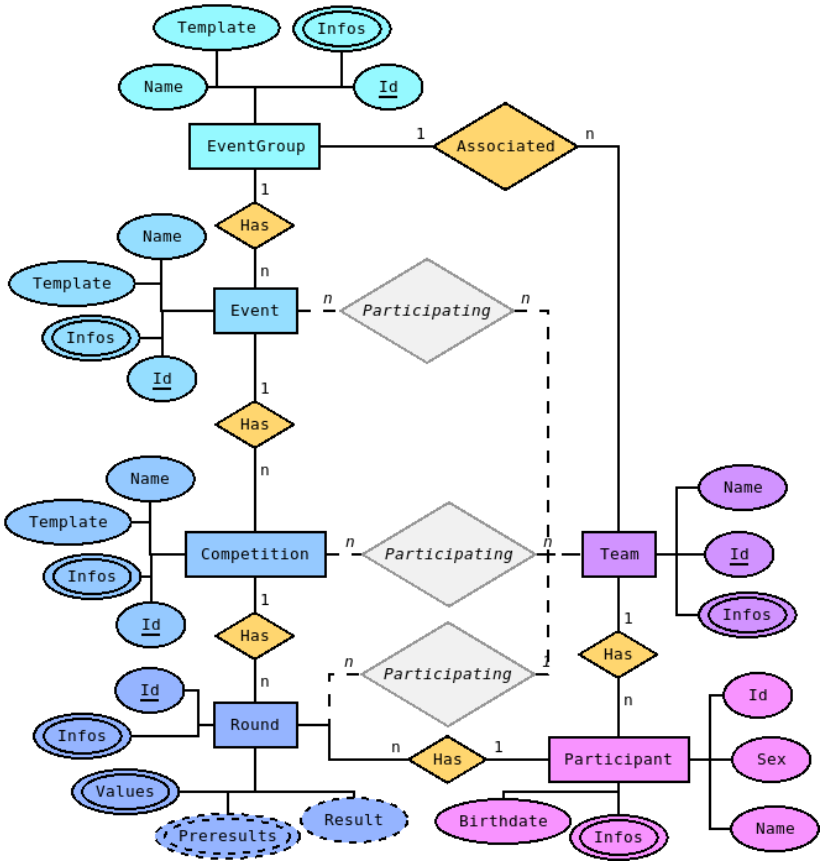
Ténylegesen a hierarchia legalján a `utils` modul szerepel, amely segédosztályokat, segédmetódusokat tartalmaz, melyeket a `model` is használ. A `utils` modulnak más függősége nincs, de ne felejtsük el, hogy implicit módon a `java.base` modul ennek is függősége.

A hierarchia mintegy mellékágán helyezkedik el a `scripting` és az `api` modul. Ezekről bővebben a következő fejezetben írok.

4.2.2 Adatszerkezet



6. ábra: Relációs diagram



7. ábra: ER diagram

A 6. és a 7. ábrán látható a példaalkalmazás adatbázisának struktúrája. A struktúra egy megmérettetésnek (sportverseny, tudományos verseny, stb.) felépítését valósítja meg a lehető legáltalánosabban, hiszen a dolgozat koncepciója az, hogy az alkalmazással akármilyen megmérettetést le lehet jegyezni.

Az `EventGroup` a megmérettetés egészének felel meg, pl. Diákolimpia. Az `Event` a megmérettetés főbb részegységeinek felel meg, amelyek több versenyszámot is tartalmaznak, pl. Diákolimpián a fiú és lány kategóriák. A `Competition` egy konkrét

versenyszámot takar, pl. távolugrás. A `Round` egy versenyző nekifutásait jelenti az adott versenyszámon, pl. Béla a távolugrás versenyszámon 1.97, 2.32, 2.14 métereket ugrott (`Values`), ez a három ugrás felel meg egy `Round`-nak. A `Round`-ok értékeiből részeredményeket (`Preresults`) és eredményt (`Result`) számol az alkalmazás, de ezeket külön nem tárolja le. A `Round`-okat résztvevőkhöz (`Participant`) társítjuk, a fenti példában ez lenne Béla. A résztvevők pedig csapatokba rendezettek (`Team`).

Az egyes csapatokat `EventGroup`-okhoz rendeljük, ez jelenti a regisztrációt, hogy egy csapat egy versenyen részt vesz. Ezt a kapcsolatot asszociációs kapcsolatnak neveztem. Az, hogy egy csapat regisztrált egy megmérettetésre, nem jelenti, hogy minden versenyszámon részt vesz, így ez a reláció nem váltja ki a többi hasonlót.

Mindegyik egyedhez tartozik egy több értéket támogató `Infos` attribútum. Ez egy saját adatbázis tervezési séma, amelyet már több projektemben is alkalmaztam. Azt a célt szolgálja, hogy a táblákban olyan értékeket is el lehessen tárolni, amelyekre nincs mező definiálva. Ugyanis bármikor igény merülhet fel arra, hogy egy új attribútumot kellene tárolni az adatbázisban, de ehhez módosítani kellene az adatbázist és az alkalmazás backend-jét is. Ehelyett az adatbázis tartalmazzon egy binárisként (BLOB/LOB) tárolt Java `Properties` objektumot, és a backend is ezt az objektumot vezesse át a metódus paraméterekben, illetve a HTTP kérésekben is ezt fogadja. Ezzel elérjük azt, hogy amennyiben a jövőben új attribútumot kell letárolni, akkor ahhoz csakis a kliens kódját kell megváltoztatni (kis mértékben), a backend és az adatbázis érintetlen maradhat. Csupán a `Properties` objektumba kell egy új értéket felvenni. Ez az adatbázis tervezési séma a dolgozatban prezentált alkalmazás esetében különösen hasznos lehet, hiszen minden megmérettetés más és más adatokat igényelhet.

Az eddig leírtak látszanak a relációs modellen. Az ER modellen ezen felül van még három Participating kapcsolat. Ezek nincsenek átvezetve az adatbázisba, hanem az érintett egyedek lekérdezésekor ezeket külön lekérdezi az alkalmazás (tranziensek). A szerepük a következő. A 6. és 7. ábrán látható, hogy az adatbázis struktúra két részre tagolódik:

- `EventGroup → Event → Competition → Round`
- `Team → Participant → Round`

A két ág között a `Round` jelenti a kapcsolatot. Ahhoz, hogy pl. egy `Event`-ről meg tudjuk mondani, hogy milyen `Team`-ek vesznek részt rajta, az asszociációnak le kell menni a `Round`-ig, és onnan fel a `Team`-ig. Azt megelőzendő, hogy az asszociációkat minden ilyen lekérésnél elő kelljen állítani, utólag közvetlen kapcsolatot alakítok ki az `Event` és a `Team` között. Ez a Participating reláció. A gyakorlatban ez így néz ki:

```
@PostLoad
private void onLoad() {
    this.teams = getCompetitions().stream().flatMap(c → c.getTeams().stream())
                                           .collect(Collectors.toList()); }
```

Alább látható a `Competition` egyedhez készült osztály kódjának az adatstruktúra szempontjából releváns részlete.

```
@Entity
public class Competition
  implements Serializable {

  @ManyToOne
  @JoinColumn(name = "competitions")
  private Event event;

  @Column
  private String name;

  @OneToMany(mappedBy = "competition",
             cascade = CascadeType.ALL)
  @Column(name = "rounds_competition")
  private List<Round> rounds;

  @Transient
  private List<Team> teams;

  @Column
  private String template;

  @Column
  @Lob
  private Properties infos;

  @Id
  @GeneratedValue
  private Long id;

  @PostLoad
  private void onLoad(){
  //teams feltöltése adatokkal
  }
}
```

4.2.3 Külső logika kapcsolódása

Ebben a fejezetben arról fogok írni, hogyan valósul meg a szkriptek formájában kívülről az alkalmazásba bekapcsolt logika átvétele, használata. Talán alulról érdemes megközelíteni a kérdést. A kapcsolódási pontot funkcionális programozási módszerek segítségével oldottam meg.

A funkcionális programozás egy deklaratív programozási paradigma, miszerint a feladatokat függvények kiértékelésére vezetjük vissza, szem előtt tartva az állapotmentességet, és elkerülve az adatmutációt. A függvényeknek nincs mellékhatása. A függvények definíciója arra utal, hogy mit szeretnénk megcsinálni, nem arra, hogy hogyan. [20] A funkcionális paradigma szerint programozni szinte minden programozási nyelven

lehet, de számos nyelv kifejezetten támogatja is ezt. A Java a 8-as verziótól kezdve támogatja.

Java-ban a népszerű függvények definícióját a `java.util.function` csomag tartalmazza, de írhatunk sajátot is. Java-ban a függvény definíciója egy olyan interfész, amelynek pontosan egy metódusa van (és az egyértelműség kedvéért `@FunctionalInterface` annotációval láttuk el). Ennek az interfésznek az implementálása a függvény deklarálása.

Az alkalmazásban a külső logika helyét ilyen függvény definíciókkal jelöltem ki. A függvények definícióját a `Round` modell osztály tartalmazza:

```
@Entity
public class Round {  //Nem releváns adattagok, metódusok kihagyva

@Column(name = "values_")
@Lob                  //Used ArrayList instead of List because of serialization
private ArrayList<Double> values = new ArrayList<>(3);

public List<Double> preresults(Function<List<Double>, List<Double>> logic) {
    return logic.apply(values);
  }

public Double result(Function<List<Double>, Double> logic) {
    return logic.apply(values);
  }

}
```

Két kapcsolódási pontot emeltem ki: a részeredmények számítását végző `preresults` metódust és az eredményt számító `result` metódust. A `preresults` metódus paraméternek egy olyan függvényt kap, amely egy valós vektort valós vektorrá transzformál. A `result` metódus egy olyan függvényt vár paraméternek, amely egy valós vektort skaláris valós számmá transzformál. Mind a `preresults`, mind a `results` metódus a paraméterben kapott függvényt fogja meghívni a `values` adattaggal. De honnan jönnek ezek a függvények?

A tényleges implementációt a külső szkript fájlok tartalmazzák. Ezek a szkript fájlok az `EventGroup`-ok szerint (pl. Diákolimpia) vannak csoportosítva. Ezen belül a neveik a `Competition`-ök `template` adattagjai alapján kötöttek, ezt az elnevezési konvenciót alkalmazva azonosíthatóak be. A `template`-ekről később lesz szó, de elöljáróban annyit érdemes itt is megjegyezni, hogy egy `template`-hez (vagyis egyféle megjelenéshez) egy

logika kapcsolható, így a szkriptek egyértelműen azonosíthatóak. Alább egy JavaScript fájl teljes tartalma látható, amely a távolugrás részeredményét számítja.

```
function apply(doubleList) {
  var array = [(doubleList[0] + doubleList[1] + doubleList[2])/3.0,
              (doubleList[3] + doubleList[4] + doubleList[5])/3.0];
  return Java.to(array, "java.util.List");
}
```

A függvényeket az `api` modul `Dispatcher` osztályától lehet lekérni. A `Dispatcher` osztály bevezetésére a körkörös referencia elkerülése miatt volt szükség, amelyről a megfelelő fejezetben lesz szó. Az `api` egy `EngineProvider`, azon belül egy `Engine` interfészt definiál. Az `Engine`-t implementáló osztály fogja a szkripteket futtatni, az `EngineProvider` pedig példányosítja az `Engine`-t a megfelelő elérési úttal. A szkriptek elérési útja az aktuális mappán belül a „modules” almappa, azon belül pedig az `EventGroup` neve, pl. Diákolimpia.

```
public interface EngineProvider {
    Engine getEngine(Path scriptPath);
    static interface Engine {
        Function<List<Double>, List<Double>> preresults(String filename)
          throws FileNotFoundException, MyScriptException, ClassCastException;
        Function<List<Double>, Double> result(String filename)
          throws FileNotFoundException, MyScriptException, ClassCastException;
}}
```

A `FileNotFoundException` akkor dobódik, ha a megadott szkript nem található, a `ClassCastException` pedig akkor, ha szkript visszatérési értéke nem megfelelő formátumú. Ha a szkript futtatása közben történik valami hiba, akkor egy `ScriptException` dobódik, de ez a kivétel a `java.scripting` modulban található. Nem szerettem volna, ha a keretprogramhoz csak egy kivétel kedvéért hozzá kell adni az egész `java.scripting` modult, így inkább ezt az exception-t becsomagolom egy sajátba, amit az `api` modul tartalmaz. Az `api` tartalmaz még egy `Statistics` interfészt is, amellyel azt mutatom be, hogyan lehet a keretprogramból a külső modulok felé adatot vinni vagy segédmetódusokat biztosítani.

```
public interface Statistics {
    Function<double[], Double> getMean();
    Function<double[], Double> getMedian();
}
```

Az `Engine` interfészt és implementációját tekinthetjük egy `service`-nek, hiszen jól definiált üzleti logikai egységet alkot. Így nem éreztem szükségét, hogy egy külön `service`-t hozzak létre ennek elfedésére. A `controller` vagy bármelyik `service` hozzáférhet közvetlenül az `Engine` interfészhez, ha szüksége van rá. E sorok írásakor az `Engine`-t csak a `controller` használja, de lehetne pl. egy rangsort készítő, vagy pdf-et generáló `service`, ami szintén használja. A használata a következőképpen néz ki:

```
model.addAttribute("prefunc", TryOrNull(() ->
    engineProvider.getEngine(scriptPath(compSelected))
        .preresults(scriptName(compSelected, ResultType.PRERESULT))));
```

Ezen a kódrészleten egy utasítás látható, amely egy megmérettetéshez rendelt részeredmény-függvényt helyez el az MVC szerinti modellbe. Alulról indulva:

1. Visszafejtjük a szkript nevét az aktuális megmérettetés (`compSelected`) template-jéből.
2. Visszafejtjük a szkript elérési útját az aktuális megmérettetés `EventGroup`-jának nevéből.
3. Egy `EngineProvider` segítségével példányosítunk egy `Engine`-t az elérési úttal.
4. Az `Engine`-től lekérjük a megadott nevű részeredmény-függvényt.
5. Ha a lekérés közben dobódott kivétel, akkor `null`-al térünk vissza, egyébként a függvénnyel.
6. A függvényt elhelyezzük az MVC struktúra szerinti modell objektumban.

Ezek után a modellben már elérhető lesz a függvény, azt a template-ben fel lehet használni a következő módon:

```
<td th:text="${#numbers.formatDecimal(round.preresults(prefunc).get(0),
0, 'COMMA', 2, 'POINT')}"></td>
```

Ez a kódrészlet egy Thymeleaf template-ből származik. A Thymeleaf-specifikus dolgokkal „4.2.5 A UI formálása kívülről” fejezetben foglalkozom, itt azt szeretném bemutatni, hogy az előbbiekben a modellbe helyezett függvényt hogyan lehet használni. Egyszerűen át kell adni a külső függvényt a korábban bemutatott Round osztály preresults metódusának, ez egy valós számokból álló listával tér vissza, amelynek itt vesszük a 0. elemét.

Ezzel bezárult a kör. Megvan a függvény helye (a `Round` osztályban), megvan a szkript (külső JavaScript fájlban), megvan az `Engine`, ami betölti a szkriptet, és megvan az osztály, ami használja (ebben a példában az MVC View).

4.2.4 Az adat és vezérlés útja

Ebben a fejezetben azt fogom bemutatni, hogy egy HTTP kérés hogyan halad át a modulokon. Amint egy kliens megnyitja az alkalmazást a böngészőből, betöltődik az index oldal. Az alkalmazás valójában csak ezt az egy oldalt használja, erről el nem navigál (single page). Az index oldal tartalmazza a navigációs sávot és az `EventGroup` választót. A háttérben ez tartalmazza továbbá az összes általános JavaScript és CSS kódot, hogy ezeket elegendő legyen egyszer betölteni a kliens élete során.

- Az `EventGroup` kiválasztásakor meghívódik a `retrieveEventGroup(id)` JavaScript függvény.
- Ez egy HTTP GET kéréssel meghívja a `/getEventGroupFragment/{eventGroupId}` útvonalra beállított metódust.
- A metódus betölti az `EventGroup`-hoz tartozó template-eket, és az MVC model-ben elhelyezi a kiválasztott `EventGroup`-ot feltöltött adattokkal, végül visszatér az `EventGroup`-hoz társított template fragment-tel.
- A template-ből html generálódik.
- A kliens betölti a fragment-et a megfelelő helyre.
- Ezek után a kliens az `EventGroup`-hoz hasonlóan kiválasztja az `Event`-et, azon belül a `Team`-et, azon belül a `Competition`-t, ezek az előző lépések szerint betöltődnek.
- A `Competition` kiválasztásakor a modelbe bekerül az aktuális `Team` (benne a `Participant`-okkal), az aktuális `Round`-ok, a részeredmény- és eredményfüggvény, illetve egy *POJO* objektum, amelyet a kliens feltölthet adatokkal, ha új `Round`-ot szeretne hozzáadni.
  - A részeremény- és eredményfüggvények a megfelelő külső szkriptből töltődnek be, a folyamat a „4.2.3 Külső logika kapcsolódása” fejezetben olvasható.
- A kliens a `Round`-okat törölheti, felülírhatja és felvihet újakat. Habár az adatbázis és a backend ezt megengedi, a példaalkalmazás nem támogatja azt, hogy egy `Participant` egy `Competition`-ben több `Round`-dal rendelkezzen – úgy logikus, hogy egy részvevő egy versenyszámon csak egyszer vehet részt.
- Egy `Participant`-hoz a `Round` felvitele a `send(compId, partId, json)` JavaScript függvénnyel történik, amely POST kéréssel áthív az `addRound()` metódusba, mely paraméterként a kitöltött POJO objektumot kapja meg, illetve a `Competition` és a `Participant` azonosítóját.
  - Az `addRound()` metódus a `CompetitionService` és a `ParticipantService` segítségével lekéri a megfelelő egyedeket, majd a `RoundService` segítségével lementi a felvitt

`Round`-ot az adatbázisba. A `RoundService` hozzáadáskor továbbítja a kérést a `RoundRepository`-nak.
  - Meglévő `Round` módosításakor a kérés a `RoundService` `replace()` metódusába érkezik, amely a `Round` osztály `overwrite()` metódusát használja.

```
public Round replace(Round old, Round nevv) {
    old.overwrite(nevv);
    roundRepository.flush();
    return roundRepository.save(old); }
```

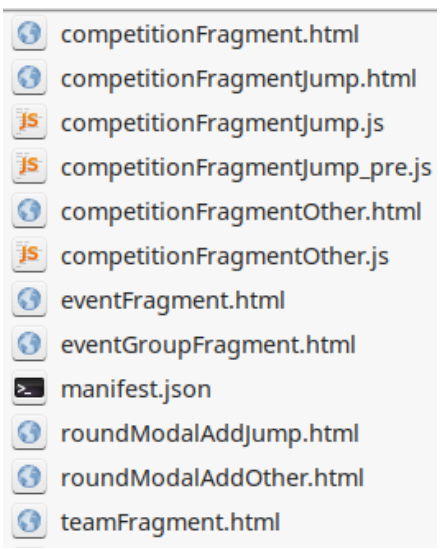
Az `overwrite()` metódus minden egyedben megtalálható és a szerepe az, hogy a régi egyedet felülírja az új egyeddel úgy, hogy az adatbázisban is használt azonosítóját és a nem frissíthető adattagjait (pl. template név) érintetlenül hagyja. Az egyedek egyébként immutable-ök, csak ezen az egy metóduson keresztül módosíthatóak.

Az `EventGroup` → `Event` → (`Team`) → `Competition` → `Round` ághoz hasonlóan működik az (`EventGroup`) → `Team` → `Participant` ág kezelése is.

- A navigációs sávon a Teams menüpontot választva először ki kell választani az `EventGroup`-ot.
- Ez meghívja a `retrieveTeams(id)` JavaScript metódust, mely GET kéréssel áthív a `TeamController`-be, a `getTeamsFragment()` metódusba.
- A metódus paraméterként megkapja a kiválasztott `EventGroup` azonosítóját és visszatér a hozzá társított `Team`-ekkel.
  - Új `Team` hozzáadásához nem szükséges POJO objektum, mert a példaalkalmazásban elegendő a `Team` nevét megadni.
  - Új `Participant` hozzáadásához azonban már szükséges a POJO objektum, a `getTeamsFragment()` metódus így egy `Participant`-ra szabott POJO osztállyal is visszatér.
- A korábban ismertetetthez hasonló módon történik az új egyedek lekérdezése, felvitele és törlése, itt a szerkesztés nem támogatott.
  - `Participant` felvitelénél a kliens kitölti a POJO objektumot a névvel, születési idővel és nemmel, majd POST kéréssel elküldi ezt a `TeamController`-nek, az `addParticipant()` metódushoz.
  - A `TeamController` átkonvertálja egyeddé, majd hozzáadja a `ParticipantService` segítségével az adatbázishoz.

4.2.5 A UI formálása kívülről

A 8. ábrán egy külső modulban található fájlok láthatóak. Háromféle fájl figyelhető meg. A .js-re végződőek a szkript



8. ábra: Fájlok egy külső modulban

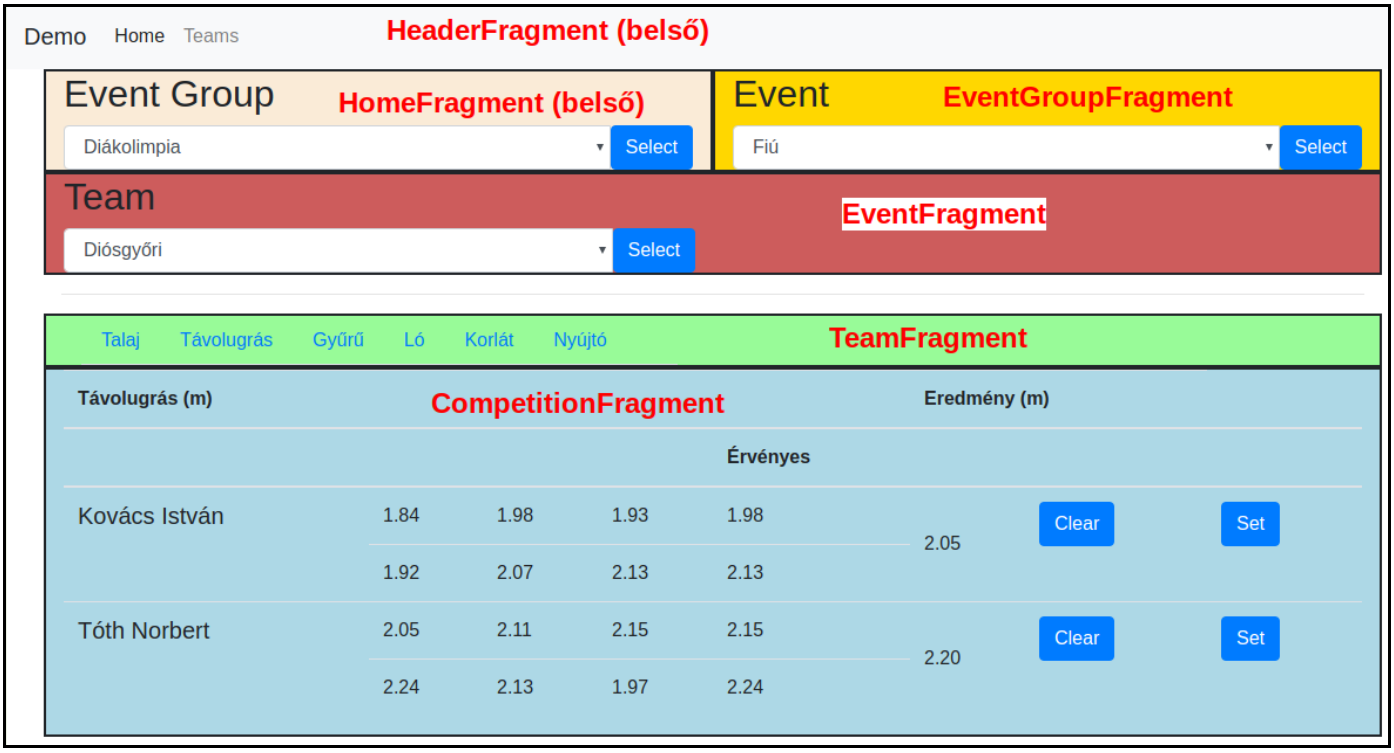
fájlok, ebben az esetben JavaScript nyelven. Ezekről a „4.2.3 Külső logika kapcsolódása” fejezetben volt szó. A .html-re végződő fájlok a fragmentek, ezek tartalmazzák a felhasználói felület egyes elemeinek leírását. A „4.1.3.3 Thymeleaf, Bootstrap, Jquery” fejezetben leírtak szerint ezek XHTML fájlok, a Thymeleaf megengedő a formátumukkal kapcsolatban, nem kötelező .xhtml kiterjesztést adni nekik. A manifest.json fájl pedig az egyes egyedekhez társítja a fragmenteket. A tartalma a Diákolimpiás példánál a következő:

```
{ "eventGroup": "Diákolimpia",
  "template": "eventGroupFragment",
  "events": [
    { "name": "Fiú",
      "template": "eventFragment",
      "competitions": [
        { "name": "Talaj",
          "template": "competitionFragmentOther" },
        { "name": "Távolugrás",
          "template": "competitionFragmentJump"  },
        { "name": "Gyűrű",
          "template": "competitionFragmentOther" },
        { "name": "Ló",
          "template": "competitionFragmentOther" },
        { "name": "Korlát",
          "template": "competitionFragmentOther" },
        { "name": "Nyújtó",
          "template": "competitionFragmentOther" }
      ]},
    { "name": "Lány",
      //A többi rész kihagyva
    }
  ]
}
```

A manifest.json segítségével vihető fel az adatbázisba egy `EventGroup`, a hozzá tartozó fragment és a hozzá tartozó `Event`-ek. Továbbá itt vihetőek fel az `Event`-ek és a `Competition`-ök a hozzájuk tartozó fragmentekkel. Ezekkel ellentétben a `Team`-ek, `Participant`-ok, illetve `Round`-ok csak az alkalmazáson belül vihetőek fel. Ennek az indoklása a következő. Az egyes modulok az egyes megmérettetésekhez készülnek, ennek a megmérettetésnek a logikáját és megjelenését definiálják. A moduloknak nem feladata a csapatok és résztvevők felvitele, ezeket a megmérettetést megelőzően az alkalmazáson belül lehet felvinni, hiszen a résztvevők nem képezik a logika vagy a megjelenés részét, és az időbeni vonatkozásuk is más.

A továbbiakban a fragmentek felépítését vázolom nagy vonalakban. A fragmentek elnevezési konvenciója szerint egy adott nevű fragment a hierarchiában alatta lévő

elemeket tartalmazza, önmagát nem. Elsőként a kliens kiválasztja az `EventGroup`-ot, az `EventGroup` választót a belső Home fragment tartalmazza. Az `EventGroup` kiválasztásakor az EventGroupFragment fog betöltődni, amely tartalmazza az `Event` választót. Az `Event` kiválasztásakor az EventFragment fog betöltődni, amely tartalmazza a `Team` választót. És így tovább.



9. ábra: A főoldal fragmentjei. A különböző színek különböző fragment területét jelölik.

De természetesen a fragmentek neveit a manifest.json tartalmazza, így tetszőleges elnevezési konvenciót lehet alkalmazni. Egy megkötés azonban van: a fragment nevéből következnie kell a szkript nevének. `Competition` esetén az eredményfüggvényt tartalmazó szkript neve ugyanaz, mint a fragment neve, csupán a kiterjesztésben térnek el (ez a 8. ábrán jól látható). A részeredmény-függvény neve megegyezik a fragment nevével, megtoldva a „_pre” kiegészítéssel, és a szkript kiterjesztésével. A fragmentek tartalmát a terjedelmük okán itt nem idézem. Két példa fragment látható a 4. Mellékletben.

4.3 Use-Case-ek, rendszer tesztek

Ez a fejezet két célt szolgál. Bemutatja a példaalkalmazás funkcióinak egy részét Use-Case-ek formájában. Másrészről a Use-Case-eket úgy fogalmazom meg, hogy azok rendszer tesztelési célra is megfeleljenek. E sorok írásakor az alkalmazáshoz egységtesztek és integrációs tesztek nem készültek. Mivel ez egy egyszerű példaalkalmazás, törekedtem arra, hogy a logikája is a lehető legegyszerűbb legyen, így viszonylag kevés egységteszt készülhetett volna hozzá, és azok is kevéssé lettek volna relevánsak a hibák kiszűrésében (pl. a legtöbb service metódus egyetlen sorból áll). A példaalkalmazásban az integrációs teszteknek is kevés jelentőséget tulajdonítok. Ezek például azt vizsgálnák, hogy egy teszt modulban a Thymeleaf változói az elvárt értékeket tartalmazzák-e. Miután több példamodult is tervezek az alkalmazáshoz mellékelni, ez a típusú vizsgálat szabad teszteléssel eleve adott. Ezek helyett rögzítek egy modult, a Diákolimpiát, és lejegyzem, hogy ennek pontosan hogyan kell viselkednie – rendszerteszt formájában.

1. Use-Case: Csapatok és résztvevők hozzáadása

	Teszt lépés	**Elvárt viselkedés**
1.	Indítsd el a szervert, nyisd meg a címét egy böngészőből. (Alapesetben localhost:8081)	Az oldal megnyílik, az `EventGroup` választó látható.
2.	A navigációs sávon kattints a Teams menüpontra.	A „Teams” felirat és az `EventGroup` választó látható
3.	Válaszd ki a Diákolimpiát.	Az „Add new team” gomb megjelenik
4.	Kattints az „Add new team” gombra.	Megjelenik egy *Modal*, ahol az új csapat nevét lehet megadni.
5.	Írj be egy nevet, kattints a „Save” gombra.	Az új csapat megjelenik.
6.	Kattints az „Add new team” gombra.	Megjelenik a Modal.
7.	Kattints a „Cancel” gombra.	Bezáródik a modal. Semmi más nem történik.
8.	Ismételd meg a 6-7. lépés ezúttal az „×” gombra kattintva a „Cancel” helyett.	Az eredmény ugyanaz.
9.	Kattints az „Add new participant” gombra.	Megjelenik a Modal. Megadható a résztvevő neve, születési ideje és neme.
10.	Töltsd ki az adatokat. Kattints a „Save” gombra.	A Modal bezáródik. Az új résztvevő megjelenik. A neve előtt a nemét jelző ikon és a neve után az életkora megfelelő.

11.	A 4-5. lépés ismétlésével adj még hozzá néhány csapatot, a 9-10. lépés ismétlésével pedig résztvevőket az egyes csapatokhoz. Különböző nemeket és születési időket használj.	A hozzáadott csapatok és bennük a résztvevők láthatóak. A nemüket jelző ikon és az életkoruk megfelel a felvitt adatoknak.
12.	Törölj ki egy csapatot a neve melletti kuka gombra kattintva.	A csapat és benne a résztvevők törlődnek.
13.	Törölj ki egy résztvevőt egy csapatból a neve melletti kuka gombra kattintva	A résztvevő törlődik, minden más érintetlen marad.

1. Táblázat: Use-Case – Csapatok és résztvevők hozzáadása

2. Use-Case: Pontszámok lejegyzése

	Teszt lépés	**Elvárt viselkedés**
1.	Indítsd el a szervert, nyisd meg a címét egy böngészőből. (Alapesetben localhost:8081)	Az oldal megnyílik, az `EventGroup` választó látható.
2.	Válaszd ki a Diákolimpiát.	Megjelenik az `Event` választó.
3.	Válaszd ki a Fiút.	Megjelenik a `Team` választó.
4.	Ha a `Team` választó üres, az 1. Use-Case szerint vigyél fel csapatokat és résztvevőket.	A `Team` választóban az előzetesen felvitt csapatok láthatóak.
5.	Válassz ki egy csapatot, amelyben több résztvevő van.	Megjelenik a `Competition` választó.
6.	Válaszd ki a Talajt.	Megjelenik a `Round`-ok felvitelére alkalmas felület, benne a csapat tagjaival.
7.	Az első résztvevő neve mellett kattints a „Set” gombra.	Megjelenik a pontszámok felvitelére alkalmas Modal, benne 3 mezővel.
8.	Írj be pontszámokat. Tizedes számokat is használj. Kattints a „Save” gombra.	Megjelennek a felvitt pontszámok és mellettük a számított eredmény. A Talaj versenyszámon az eredmény a pontszámok átlaga. A résztvevő sorában megjelenik a „Clear” gomb.
9.	A 7-8. lépés ismétlésével a többi résztvevőhöz is vigyél fel pontszámokat.	Minden versenyzőnél megjelennek a pontszámok, az eredmények a pontszámok átlagát adják.
10.	Egy résztvevő sorában kattints a „Set” gombra.	Megjelenik a pontszámok felvitelére alkalmas Modal, benne 3 üres mezővel.
11.	Írj be pontszámokat, kattints a „Save” gombra.	A résztvevő sorában az új pontszámok jelennek meg, helyes eredménnyel.
12.	Egy résztvevő sorában kattints a „Clear” gombra.	A résztvevőhöz felvitt pontszámok törlődnek.
13.	Válts át a Távolugrás fülre.	Megjelenik a `Round`-ok felvitelére alkalmas felület, benne a csapat ugyanazon tagjaival. A résztvevőknél nem szerepel pontszám.

14.	Ismételd meg a 7-12. lépést a Távolugrás versenyszám esetében is.	Az alkalmazás hasonlóan viselkedik a következő kivételekkel. A pontszámok felvitelére alkalmas Modal ezúttal 2×3 mezőből áll. A pontszámokból részeredmény is számítódik, mely a hármas csoportok legnagyobb értéke. Az eredmény ezen két érték átlaga.
15.	Válassz másik csapatot a `Team` választóban, majd válaszd a Távolugrást.	Az új csapat résztvevői láthatóak, mellettük pontszám nem szerepel.

2. Táblázat: Use-Case – Pontszámok lejegyzése

4.4 Kihívások, akadályok

A dolgozat készítése közben számos akadályba ütköztem. Minden akadály tanulságos volt a maga nemében. Két kategóriába tudom őket besorolni.

4.4.1 Megoldott akadályok

Az egyik kategória a megoldott akadályok halmaza. Ezekből megtanultam, az adott problémát hogyan kell megoldani. Ezzel a tapasztalattal a hasonló jövőbeli akadályokat vagy elkerülni vagy gyorsan megoldani képes leszek. A teljesség igénye nélkül néhány ilyen eset:

```
public Event add(EventGroup eventGroup,
                 String name, String template, Properties infos) {

  return eventRepository.findAll().stream().filter(e ->
    e.getName().equals(name) && e.getEventGroup().equals(eventGroup))
                                          .findFirst().orElseGet(() ->
    eventRepository.save(new Event(eventGroup, name, template, infos))); }
```

Az itt látható metódus megírásával sokat küzdöttem két különböző okból is. Ez a metódus a következőt csinálja:

1. Lekéri az összes `Event`-et az adatbázisból.
2. Megnézi, van-e már az adatbázisban a megadott `EventGroup` alatt megadott nevű `Event`. (Az `Event`-ek nevei a tartalmazó `EventGroup`-ra nézve egyediek.)
3. Visszaad egy `Optional<Event>`-et, amely vagy tartalmazza a meglévő `Event`-et, vagy ha nincs ilyen az adatbázisban, akkor üres.
4. Ha az `Optional`-ben van érték visszatér vele, ha nincs, akkor elment egy új `Event`-et az adatbázisba és visszatér azzal.

A 4. lépésben követtem el egy hibát. Az `Optional`-nek van `orElse(T)` és `orElseGet(Supplier<T>)` metódusa is. Ha az `eventRepository.save(Event)` metódust az

`orElse(T)` paramétereként hívom meg, akkor a `save(Event)` mindkét ágon kiértékelődik. Akkor is, ha van az `Optional`-ben érték és akkor is, ha nincs. (Ugyan az előbbi esetben nem ezzel tér vissza). Így az adatbázisban az egyedek többször tárolódtak el. A megoldás a *deferred execution* jelentőségének észben tartása.

A másik hibaforrás az `equals()` metódusokban rejlik, amelyeket az `Event`-ek nevei és tartalmazó `EventGroup`-ok összehasonlítására használok. Általában az `equals()` metódus egyszerűség, performancia és a limitált scope miatt közvetlenül adattagokat hasonlít össze, vagyis megvizsgálja, hogy az adott objektum releváns adattagjai egyenlőek-e a másik objektum adattagjaival. Ez általában működik. Azonban mivel a Hibernate nem az egyedek objektumaival tér vissza, hanem azokból leszármazott *proxy*-val, már más a helyzet. A proxy objektumok lusták, vagyis az adattagokban mindaddig `null` van, amíg a hozzájuk tartozó gettert legalább egyszer meg nem hívják. Az `equals()` metódus tehát valós adattagokat hasonlított össze `null`-al. Ez `NullPointerException`-t vagy hamis negatív egyenlőségvizsgálatot eredményezett. A megoldás, hogy gettereket kell használni az `equals()` metódusban is.

A fenti két eset inkább programozástechnikai hiba, figyelmetlenség. A továbbiakban inkább a nagyobb jelentőségű, mások számára is tanulságos kihívásokat emelem ki.

4.4.1.1 Split Package problémakör

A 9-es verziótól kezdve a Java nem támogatja az ún. split package-eket. A split package[21] azt jelenti, hogy ugyanaz a csomagnév két különböző .jar fájlban szerepel. Kompatibilitási okokból a class path-ban split package használata még megengedett, de a module path-on már nem. Mivel a „3.2.1 A Java modulok működése” fejezet szerint a modulnevek konvenció szerint globálisan egyediek, és egy csomag csak egy modulban található meg, egy modul válik a kizárólagos felelősévé egy csomagnak. Ez a nyilvánvaló egységbezárási előnyök mellett jobb performanciát is eredményez, hiszen egy osztály keresésekor a JVM-nek nem szükséges az összes modult, ill. jar fájlt átnézni. Emellett a split package-ek különböző rejtett hibákat is okoztak, kiváltképp RMI keretrendszerekkel. A problémák sokszor abban gyökereztek, hogy ilyen módon hozzá lehetett férni package private tagokhoz kívülről. A hozzáférési korlátozások megszegése számos negatív

következménnyel járt, ezért is döntöttek úgy a Java fejlesztői, hogy ezt a továbbiakban nem támogatják.

De mit jelent mindez a gyakorlatban, milyen problémákat okozott a dolgozat elkészítésében? Bizonyos régóta létező libek a Java 9 megjelenésekor is tartalmaztak split package-eket. Az alkalmazásomban pedig használtam olyan libeket, amelyek a háttérben ezeket a problémás libeket használták. Ez önmagában még nem okoz feltétlenül gondot a class path és a module path közötti hídnak köszönhetően. Ez a híd az automatikus modulokra utal. A névtelen modul (amely a class path-on található) kompatibilitási okokból tartalmazhat split package-eket, a nevesített modulok azonban nem hivatkozhatják meg a névtelen modult. Itt jönnek képbe az automatikus modulok, amelyek ugyan nem tartalmazhatnak split package-et, de implicit módon meghivatkozzák a névtelen modult, ami viszont igen. Így lehetőség van arra, hogy a split package-ek a névtelen modulban legyenek, ezeket egy automatikus modul meghivatkozza, ezt az automatikus modult pedig már egy nevesített modul is meghivatkozhatja.

A probléma ott jön elő, hogy egy automatikus modul nem szivárogtathat ki tagokat a névtelen modulból. Tehát ilyen pl. egy nevesített modulban nem szerepelhet: `automatikusModulMetódus(NévtelenModulTípus obj)`. Ebben az esetben automatikus modullá kellene tenni a `NévtelenModulTípust` tartalmazó jar fájlt, ez viszont már split package-et hozhatna be.

A példaalkalmazásban történt konkrét split package konfliktusokat nem tudok felsorolni. Ahogy a „1. Bevezető” fejezetben említettem, a Java 9 megjelenése óta foglalkozom a modulrendszerrel. Így a felmerülő split package problémákat már azelőtt megoldottam, hogy ezt a dolgozatot elkezdtem volna írni. Az én esetemben – emlékeim szerint – a megoldást az jelentette, hogy Maven segítségével kihagytam a duplikált csomagok azon felét, amelyre nem volt szükség. De akadtak olyan split package-ek, amelyeknek mindkét felére szükség volt. Erre csak az jelentett megoldást, hogy meg kellett várni, amíg az adott lib fejlesztőcsapata javítja ez a hibát. Miután a példaalkalmazás aktuális verziójában már csomagok kihagyására nincs szükség, úgy tűnik, idővel fejlesztik ezeket a libeket, és megoldódnak a problémák.

A következő javítási módszereket találtam a split package-ek feloldására:

- Átmozgatás másik csomagba: ha split package-et olyan lib hozza be, amely saját fejlesztésű, akkor át kell nevezni a csomagot és feloldani az esetleges package private hozzáférés megsértését.[22]
- Frissítés: külső lib esetén jelezni kell a fejlesztőknek a hibát, meg kell várni, míg kijavítják, majd frissíteni a függőségeket.[22]
- Egyesítés: Ha más lehetőség nincs, de ennek a lehetősége adott, meg lehet próbálni a két jar fájlt egyesíteni, a két csomagot összevonni.[22]
- Kihagyás: Ha a split package tagjaiból csak az egyikre van szükség, és erre van lehetőség, akkor a másik tagot ki lehet hagyni a fordítási egységből.
- Elfedés: Mivel az automatikus modulok hozzáférnek a névtelen modulhoz, amely tartalmazhat split package-et, megtehetjük, hogy készítünk egy automatikus modult csak azért, hogy elfedjük a problémás csomagokat. Alternatívaként azt is megtehetjük, hogy ideiglenesen a saját modulunkat átalakítjuk automatikus modullá, ekkor maga is hozzá fog férni a névtelen modulhoz.[21]
- Patch: Használhatjuk a --patch-module fordítói parancssori argumentumot, mely arra szolgál, hogy egy jar fájl tartalmát beleömlesszük egy meglévő modulba.[23]

4.4.2 Áthidalt akadályok

Számos olyan apróbb akadályba futottam bele az alkalmazás fejlesztése során, amelyek visszavették a fejlesztés tempóját. Ezek egy része az adatbázis kezelésre, másik része template kezelésre vezethető vissza, illetve ide sorolom a körkörös referencia problémát is.

Az adatbázis kezeléssel kapcsolatos problémák mögött a Hibernate lusta inicializációja és az entitások közötti kötött reláció áll. Előbbire a „4.4.1 Megoldott akadályok” fejezet elején ismertetett `equals()`-os eset hozható fel, amelyet ugyan sikerült megoldani, de több hasonló is akadályozta az alkalmazás fejlesztését. Utóbbira pedig egy példa a `Round`-ok törlése. A Hibernate ismeretlen okból nem képes a `Round`-okat kitörölni. Hasonló hibajelenségbe mások is belefutottak már, de a javasolt megoldások az én esetemben nem segítettek. A javaslatok között szerepelt egy brute-force módszer is, mely szerint ha a Hibernate helyett saját SQL utasítással próbálom törölni az egyedet, akkor ki fog törlődni. Ez a módszer valóban bevált, és bizonyítja, hogy technikailag semmi akadálya nem volt az egyedet kitörölni, csupán a Hibernate-nek okoz ez gondot valamiért.

Az adatbázis kezeléssel kapcsolatban azt a tanulságot tudom levonni, hogy vagy a Hibernate-et kell mélyebben megismernem, vagy könnyebben érthető alternatívára kell váltanom.

A template-ekkel kapcsolatosan két problémakörrel találkoztam. Egyrészt szembesültem azzal, hogy a statikus template-ek rendkívül korlátozottak, relatíve nagy mennyiségű JavaScript kóddal szükséges azt kiegészíteni ahhoz, hogy igényes, gyors és modern weboldalt kapjunk. A legfájóbb pont ebben, hogy a Thymeleaf template-ek a JavaScripttel együttműködni csak komplikált módon tudnak. Néhány esetben szükséges volt bevezetni globális változókat, mert nem tudtam másképpen láthatóvá tenni a template egy változóját a JavaScript kód számára. Emellett a szintaxissal voltak problémák, sok esetben a Thymeleaf és a JavaScript szintaxisa ütközött egymással. Ezt elkerülendő készítettem egy saját segédosztályt, amely az egymásba ágyazott szintaxist linearizálja:

```
js.queryById(„példaForm”).func(„css”).argStr(„background”)
                                     .argObj(„rgb(50,100,0)”)

==> $(„#példaForm”).css(„background”, rgb(50,100,0))
```

Másrészt a példaalkalmazásban a template-ek betöltése úgy működik, hogy amikor a kliens az `EventGroup` választón kiválaszt egy `EventGroup`-ot, akkor az ahhoz tartozó template-ek átmásolódnak egy dedikált mappába, és a Thymeleaf ezeket dolgozza fel. Ezzel az a probléma, hogy egy felhasználóssá teszi az alkalmazást. Legalábbis egyszerre csak egy `EventGroup` lehet kiválasztva. Habár ez sok esetben nem jelent valódi problémát, hiszen egy időben egy klienssel csak egy megmérettetést logikus tartani. Ugyanakkor más alkalmazásban ezt a sémát nem lehetne kivitelezni. Az példaalkalmazásban ezt a korlátozást végül nem oldottam fel, így ebben a formában egyszerre csak egy felhasználó tudja azt használni.

A template-es kérdéskörből azt a tanulságot vonom le, hogy legközelebb REST-esebb megközelítést érdemes választanom. Illetve a felhasználói felület elemeit nem fájlból, hanem memóriából kell betöltenem.

4.4.2.1 Körkörös referencia

A „4.2.1 Modul struktúra” fejezetben látható modul gráfon szemet szúr, hogy a struktúra meglehetősen komplex. Ennek egyik oka az, hogy a gráfon látszódnak a tesztelési célból definiált függőségi relációk is, ezek a kiadott alkalmazásból természetesen

eltávolíthatóak lennének. A lényegesebb ok, amely a komplexitást növeli a körkörös referencia tiltása.

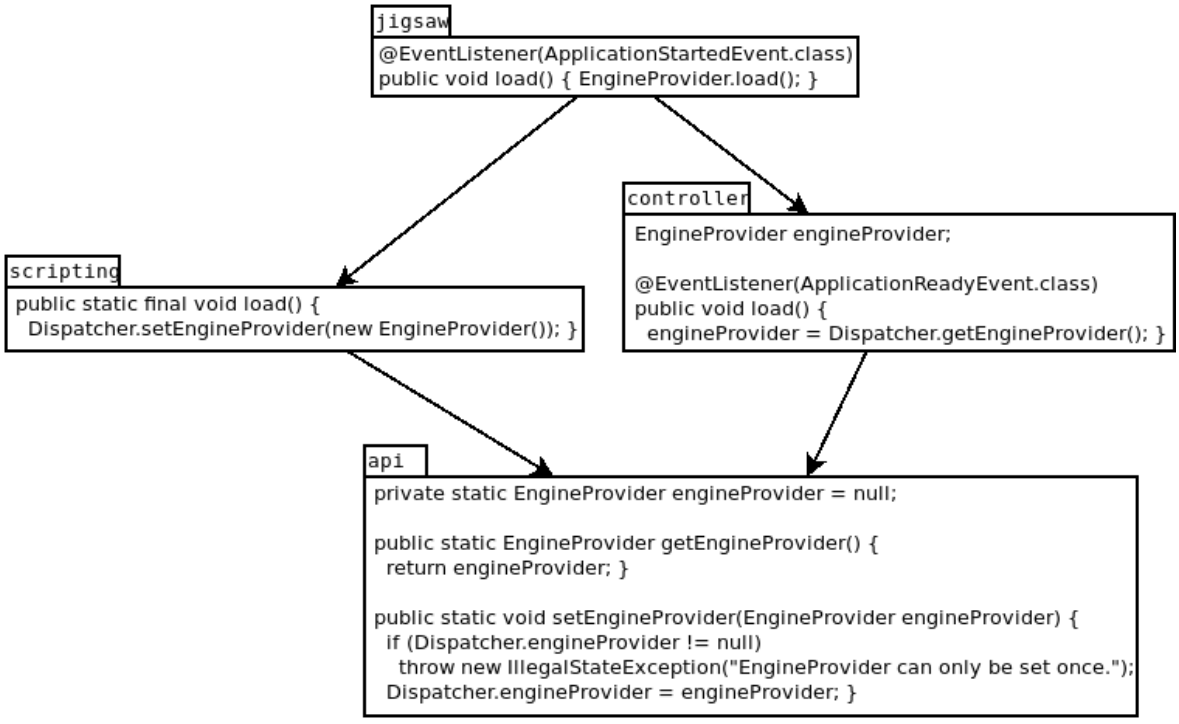
A körkörös referencia (vagy függőség) azt jelenti, hogy két vagy több szoftver komponens direkt vagy indirekt módon függ egymástól. A körkörös referenciát általában anti-pattern-nek tekintik, ugyanakkor valós szoftvereket nézve meglehetősen gyakoriak.[24] Én úgy gondolom, hogy vannak olyan helyzetek, amelyekben a körkörös referencia bevezetése célszerűbb, mint valamilyen kikerülő megoldás alkalmazása, illetve akadhatnak olyan szituációk is, melyeknél nem is lehetséges a körkörös referenciát elkerülni. Mivel a Java modulrendszer és a Maven is tiltja a körkörös referenciát, a példaalkalmazásban kénytelen voltam kikerülő megoldást alkalmazni, amely körülményesebb, mint elvárnám. Véleményem szerint ez a projekt is azok közé sorolható, amelyeknek jobbat tenne, ha elfogadnák a körkörös referenciákatt. A továbbiakban bemutatom, mi okozta a problémát, és hogyan sikerült ezt a kötöttséget áthidalnom.

A körkörös referencia problémát az okozta, hogy a keretprogramnak hozzá kell férnie a `scripting` modulhoz, hogy a logikát lekérdezze, de a `scripting` modulnak is hozzá kellene férnie (közvetetten) a keretprogramhoz, hogy attól információkat és segédmetódusokat kapjon. A `scripting` modul a keretprogramhoz közvetlenül nem férhet hozzá, hiszen ez biztonsági kockázatot jelentene, de egy `api` modulon keresztül közvetetten már megtehetné. Ez azonban körkörös referenciát okoz.

Az első lépés az volt, hogy bevezettem az `api` modult, de nem csak abból a célból, hogy a `scripting` ezen keresztül érje el a keretprogramot, hanem hogy a keretprogram is ezen keresztül érje el a `scripting` modult. Ha a kommunikáció mindkét irányba az `api`-n, mint közvetítőn keresztül folyik, nem alakul ki körkörös referencia.

Ez azonban egy újabb problémát vetett fel. Ahhoz, hogy ne alakuljon ki körkörös referencia az `api` és a `scripting` között, közöttük is csak egyirányú lehet a kommunikáció. Úgy találtam logikusnak, hogy az `api` legyen a függőségi gráfban alul, tehát ennek ne legyenek további függőségei. Ebből következett, hogy az `api`-ban legyen egy `Dispatcher` osztály, amely statikus vagy singleton, benne pedig egy-egy adattag, amelyben a keretprogram és a külső logika kommunikációjához szükséges objektumok találhatóak.

Csakhogy ezeket az adattagokat fel kell tölteni, az egyes komponenseknek magukat be kell regisztrálniuk. Ez a keretprogram esetében triviális, egy Spring `EventListener` az alkalmazás elindítása után beregisztrálja a kifelé küldött szolgáltatásokat. A `scripting` esetében viszont ezt nehezebb megoldani, mert a Spring konténeren kívül helyezkedik el. Írtam egy metódust a `scripting` modulba, amely betölti a szükséges objektumokat a `Dispatcher`-be, de ezt a metódust valaminek kívülről meg kell hívnia. A körkörös referencia elkerülése végett a `scripting` betöltő metódusát olyan modulból kell meghívni, amely a függőségi gráfban magasabban helyezkedik el, mint ahol felhasználják. Mivel a `scripting` modult a `controller`-ben is szeretném használni, és annál magasabban már csak a `jigsaw` modul van, adta magát, hogy ezt válasszam. A `jigsaw` modul tehát az alkalmazás indulásakor meghívja a `scripting` modulban az `EngineProvider.load()` metódust, mely betölti az `EngineProvider`-t a `Dispatcher`-be, és minden további kérés a `Dispatcher`-hez fog érkezni.



10. ábra: Körkörös referencia elkerülése diszpécser osztály segítségével

4.5 A biztonság kérdése

A biztonság kérdéséről a „3.1 Technológiai lehetőségek” fejezetben már írtam. Ott arra a következtetésre jutottam, hogy kielégítő válasz nincs. Részmegoldások vannak, mint a SecurityManager, egyedi ClassLoader vagy script engine specifikus megoldások.

Egyedi ClassLoader-rel, SecurityManager-rel és néhány „hack”-kel viszonylag jó eredményeket lehet elérni a biztonság terén, de biztonsági rések mindig maradnak. Ezzel a témával kapcsolatban a következő oldalt érdemes megnézni. Jó néhány biztonsági rést említenek itt, és különböző trükköket a kivédésükre.

https://stackoverflow.com/questions/502218/sandbox-against-malicious-code-in-a-java-application

Ugyanakkor még ennek a részleges biztonságnak is ára van: olyan mértékű szabadságvesztéssel járhatnak egyes megoldások a szkriptek számára, amely már szembe megy a használhatósággal. Mivel nem valószínű, hogy az összes biztonsági kockázatot jelentő osztályt ki tudjuk szűrni, csak white list alapú megoldásban érdemes gondolkozni. Azonban ez az eredményezné, hogy minden külső modulnak magának kell biztosítani a szükséges segédosztályokat, ez kényelmetlen a modulok fejlesztőinek és nehezíti a karbantartást is. Enyhíteni lehet ezen a problémán azzal, hogy a keretprogram a külső modulok felé expliciten szolgáltatásokat nyújt. A példaalkalmazásban ez a `Statistics` osztály, amelyet az `api`-n keresztül biztosítok kifelé.

Mindezek ellenére úgy véltem, nem lenne helyénvaló implementálni egy alkalmazást, amely potenciálisan megbízhatatlan forrásból érkező külső logikával dolgozik anélkül, hogy valamiféle biztonsági intézkedést ne valósítanék meg. Így a korábban ismertettek szerint elkészítettem egy alapszintű védelmet nyújtó és további lehetőségeket előrevetítő szkript betöltőt. Ehhez felhasználtam egyedi ClassLoader-t, egyedi SecurityManager-t valamint egyedi Thread-et. A teljes implementáció hét osztályból áll. Mindahány jelentős szerepet tölt be, azonban a terjedelmükből kifolyólag itt csupán nagy vonalakban ismertetem a működésüket, kódrészleteket pedig melléklet formájában biztosítok (4. Melléklet).

A szkriptek betöltését a `SecureEngine` osztály indítja el, amely implementálja az `api`-ban található `Engine` interfészt. Ha a `Dispatcher`-be a `SecureEngine` kerül betöltésre, és a `Dispatcher`-en keresztül ezt egy osztály használatba veszi, hogy betöltse a külső logikát, a következő események történnek (a saját osztályokat dőlttel szedtem):

- A *`SecureEngine`* elindít egy *`IsolatedThread`*-et, átad egy *`RequestType`*-ot és a szükséges paramétereket. A *`RequestType`* enum tartalmazza a kéréshez tartozó metódus nevét (pl. `preresults`) és szignatúráját (pl. `preresults(ScriptEngine, FileReader)`), a

szignatúra határozza meg a szükséges paramétereket, amelyeket az *IsolatedThread*-nek át kell adni.

- Az *IsolatedThread* bekapcsolja a *MySecurityManager*-t. Továbbá inicializál egy *PollingFuture*-t. A *PollingFuture* a `Future` interfészt implementálja, mely egy olyan adatot reprezentál, amely a létrehozásának pillanatában még nem, hanem majd csak a jövőben fog rendelkezésre állni. A *PollingFuture* adatának típusát az *IsolatedThread*-nek átadott generikus típusparaméter határozza meg.
- A *MySecurityManager* inicializál egy `ThreadLocal` adattagot, amelynek segítségével eléri, hogy a biztonsági ellenőrzések csak a hívó szálon (esetünkben az *IsolatedThread*-en) legyenek érvényben.
- Az *IsolatedThread* a *MyClassLoader* segítségével betölti az *EngineInner* osztályt.
- Ugyan az *EngineInner* osztályt szeretnénk elsősorban az egyedi ClassLoader-rel betölteni, de azok az osztályok is ezzel töltődnek be, amelyeket az *EngineInner* használ. Ha az aktuálisan betöltendő osztály Javás belső osztály, akkor kötelezően az alapértelmezett ClassLoader-rel kell betölteni. Ha az aktuálisan betöltendő osztály ugyan nem belső osztály, de megbízhatónak nyilvánítjuk, akkor szintén betölthető ezzel. Ezen a ponton lehetőség van blacklist vizsgálatra is. Ha azonban a betöltendő osztály ismeretlen (mert külső szkriptből generálódott ki), akkor kézzel töltjük be. Ezen a ponton lehetőség van a bájtkódot biztonsági ellenőrzésnek alávetni, mielőtt betöltjük.
- Ha az *EngineInner* betöltődött, vesszük a *RequestType*-nak megfelelő nevű és szignatúrájú metódust, és meghívjuk a megadott paraméterekkel.
- Közben a *MySecurityManager* dolgozik, és ellenőrzi a kritikus műveleteket. Letiltja pl. a fájlok írását és a hálózat használatát.
- Az *IsolatedThread* beállítja a *PollingFuture*-t, belehelyezi a szkript visszatérési értékét.
- Az *IsolatedThread* kikapcsolja a *MySecurityManager*-t.
- Közben a *SecureEngine* a *PollingFuture* `get()` metódusának meghívásával várja, hogy megérkezzen a szkript visszatérési értéke. Ha megérkezik az érték, visszatér vele, ha pedig letelik a megadott idő, akkor sikertelennek tekinti a szkript futását. Utóbbi esetben meghívjuk az *IsolatedThread* `interrupt()` metódusát, amely kikapcsolja a *MySecurityManager*-t és leállítja a szálat.

Ezzel a módszerrel megakadályozhatjuk, hogy a külső logika olyat tegyen, amelyre egy SecurityManager reagálni tud, azonban nem akadályozhatjuk meg, hogy egy szkript saját ClassLoader-t hozzon létre, hiszen a szkripteket betöltő osztálynak ez az üzemszerű működése. Részlegesen és nagy körültekintéssel akadályozhatjuk csak meg fájlok olvasását, mert ez az osztályok üzemszerű betöltéséhez szükséges. Részlegesen akadályozhatjuk csak meg property-k írását-olvasását, mert a szkripteket betöltő logika ezeket implementációfüggő módon használja.

Végül csak részlegesen szűrhetjük a csomagokat, amelyekhez a külső logika hozzáférhet. A SecurityManager ugyan támogatja a csomagok szűrését, de hosszú kísérletezés után erről végül letettem. A csomagok szűrése nagyon érzékeny, könnyen vezet stack túlcsorduláshoz a használata, mert egy csomag jogosultságát ellenőrző logika lefuttatásához is szükséges a csomagok jogosultságának ellenőrzése.

Ehelyett az osztályok betöltésekor a `MyClassLoader`-ben végezhető lenne csomagszűrés, de mivel a szkripteket betöltő logika saját ClassLoader-t hoz létre, ez nem vezetett eredményre. Ugyanezen okból kifolyólag a `MyClassLoader`-rel betöltött osztályok bájtkódjának elemzése sem vezetne eredményre.

Még ha sikerülne is beüzemelni a SecurityManager csomagszűrését, akkor is sok kívánnivalót hagyna maga után a megoldás. Hiszen nem feltétlenül ismert, hogy az egyes szkript betöltők milyen csomagot definiálnak a generált osztályoknak, illetve nem adott a lehetőség a csomagokon belüli szűrésre. Utóbbi problémára megoldást jelenthet egyedi jogosultságok bevezetése.

A SecurityManager csomagszűrésének stabil beüzemelése és egyedi jogosultságok bevezetése megoldást jelenthet a biztonsági kérdésekre. Ezen funkciók körüljárása és megvalósítása azonban túlmutat e dolgozat témáján, ezek egy külön dolgozat témáját képezhetnék a jövőben. Jelen dolgozatban megelégszem azzal, hogy a csomagok és osztályok szűrésétől eltekintve a SecurityManager jó néhány veszélyes műveletet megakadályoz. E téma mélyebb megismeréséhez kiindulásnak ajánlom a TWMAS projektet, amelyet a 2. Mellékletben mutatok be.

5. ÖSSZEFOGLALÁS

Ebben a fejezetben összegzem a dolgozatot. Elsőként ismertettem a dolgozat célját és koncepcióját. Két fő célt határoztam meg:

- bemutatom a Java 9 modulrendszerét, annak működését és használatát gyakorlati szempontból,
- illetve bemutatom, hogyan lehet egy Java alkalmazásba külső logikát bekapcsolni, körüljárva a biztonság kérdéskörét is.

A dolgozat céljai után ismertettem a koncepcióját, melynek fő pontja, hogy szeretnék a logika egyes részeire mint adatra tekinteni. Szeretném, hogy egy alkalmazás csupán keretet adjon a funkcionalitásnak, amelyre eredetileg szánták, és kevés munkával, akár független fejlesztők által alakítható legyen más, hasonló funkciók betöltésére is. Ennek a koncepciónak a megvalósítására és szemléltetésére készítettem egy példaalkalmazást.

A példaalkalmazás belső strukturálására a Java modulrendszerét alkalmaztam, mely a 9-es verziótól, 2017 szeptemberétől elérhető. A modulrendszer egyrészről a külső strukturáltság mellett egy belső struktúra kialakítására is lehetőséget ad. Másrészről a modulrendszer a biztonság terén is új lehetőségeket ad, melyeket meg kívántam vizsgálni.

5.1 Elért eredmények

A célok és koncepciók ismertetése után a modularitás (itt külső modularitás) kérdéskörét vizsgáltam. Választ adtam a következő kérdésekre:

- Milyen módokon lehet a külső logikát bekapcsolni az alkalmazásba?
- Milyen előnyei és hátrányai vannak a különböző módszereknek?
- Hogyan viszonyulnak az egyes módszerek a biztonság kérdéséhez?
- Létezik-e kész megoldás a problémára?
- Melyik módszer felel meg a koncepciónak, melyiket választottam és miért?

A kérdésekre a válaszok a „3.1 Technológiai lehetőségek” fejezetben olvasható. Egyik módszer sem felelt meg a feltételeimnek, így a dolgozat készítése során több módszerből is merítettem. A problémára kész megoldást nem találtam, ezért úgy gondolom a dolgozat valóban hasznos lehet – ugyanakkor a dolgozat elsősorban csupán körbejárja a kérdést, a terjedelme és időkorlátja nem ad lehetőséget arra, hogy kész megoldást nyújtson a problémára.

A biztonság kérdését körbejártam, arra kielégítő választ nem tudtam adni. Nem sikerült egy Java alkalmazásban a feltételeknek megfelelően sandbox környezetet kialakítani. Azonban a tapasztaltak alapján nem tartom a feladatot lehetetlennek. Talán ezen dolgozat és a 2. Mellékletben ismertetett dolgozat segítségével további kutatás árán megvalósítható a feladat.

A külső modulok kérdésköre után rátértem a Java modulokra. Ezek más célt szolgálnak, de kiegészítik a külső modulok által nyújtott előnyöket. Jobb belső strukturáltságot és önmagukban is némi biztonságot eredményez a használatuk. Véleményem szerint a Java nyelv elterjedtségéhez képest a Java modulrendszert rendkívül kevesen használják, kevés forrás érhető el hozzá. Ezen kívül a modulrendszer könnyen demonstrálható, egyszerűnek hathat, ugyanakkor komplexebb és/vagy speciálisabb felhasználási körülmények között már mélyebb ismerete is szükséges. Ezen okokból a Java modulrendszert e dolgozatban bemutattam, kitértem a speciálisabb esetekre, néhány olyan tényt is összegyűjtöttem, amelyet más forrás nem említ.

Mielőtt rátértem volna a példaalkalmazás implementációs elemeire, ismertettem, hogy milyen technológiákat használtam fel, illetve mi motivált az adott technológia választásában. Mivel a példaalkalmazás speciálisabb környezetet biztosít a különböző technológiák számára, mint amelyben azokat általában használják, minden egyes technológia alkalmazása során kutatásra és kísérletezésre volt szükség. Minthogy sok apró akadály jelent meg a fejlesztés különböző szakaszaiban, ezeket a dolgozatban tételesen nem áll módomban ismertetni. Ugyanakkor néhány ilyen akadályt a „4.4 Kihívások, akadályok” fejezetben ismertettem, a rájuk adott megoldással egyetemben. Valamint maga az alkalmazás is bemutatja, hogy a különböző technológiák hogyan tudnak együttműködni ezzel a környezettel. Az alkalmazás annak ellenére, hogy csak bemutatás célját szolgálja, meglehetősen nagy: csak a keretprogram több, mint 90 lényeges forrásfájlból áll, a külső modulok ehhez darabonként 12-15 forrásfájlt adnak hozzá. Ezért a teljes alkalmazást nem, csupán egyes részeit mutattam be, amelyet a következőkben foglalok össze.

Sikeresen bekapcsoltam külső logikát az alkalmazásba. A keretprogram kétféle számítást végez a kívülről érkező logika segítségével. A külső logikát funkcionális programozási technológia segítségével kapcsoltam be, amely azt takarja, hogy kívülről

egy-egy függvény érkezik az alkalmazásba, az alkalmazás pedig ezekkel a függvényekkel, nem pedig csak a kiszámolt eredményekkel dolgozik. Ez a módszer a kapcsolódási pontok megadását triviálissá teszi. A példaalkalmazás három különböző szkriptnyelvet támogat (JavaScript, Groovy, Python), de olyan módszert alkalmazok, hogy minimális munkával tetszőleges szkriptnyelvekre is támogatást lehet biztosítani. További információ a „4.2.3 Külső logika kapcsolódása” fejezetben olvasható.

Az alkalmazás különböző megmérettetéseket támogat. A modulok az egyes megmérettetések logikában való különbözetük mellett tartalmazzák az egyes UI elemek kinézetét és struktúráját is, így az alkalmazás valóban univerzális tud lenni. A külső modulok teljesen dinamikusan cserélhetőek, módosíthatóak. Új modul hozzáadható, meglévő modul törölhető futásidőben. Az alkalmazás a felhasználót értesíti a modulok változásáról. A frontend a modulok változásával dinamikusan módosul.

Példaként az alkalmazásba lehetőség van megmérettetéseket (pl. Diákolimpia), azon belül eseményeket (pl. fiú, lány kategória), azon belül pedig versenyszámokat (pl. távolugrás) beimportálni, ezzel együtt megadva a szükséges logikát és kinézetet is. Lehetőség van csapatokat, azon belül résztvevőket regisztrálni az egyes versenyszámokra. Végül lehetőség van a versenyszámokon az egyes versenyzőket pontozni. A feladat pontszámokból a külső logika alapján részeredmény és eredmény számítódik.

Az alkalmazás mögött adatbázis található. A célomnak megfelelően az adatbázis-kezelő be van építve az alkalmazásba, külön alkalmazás telepítése nem szükséges. Biztosítottam az alkalmazás teljes platformfüggetlenségét: bármelyik operációs rendszeren futtatható, ahol a Java platform elérhető, nem igényel telepítést és egyetlen futtatható állományt tartalmaz, minden más csupán külső erőforrás fájl. A teljes alkalmazás illeszkedik a Java 9 modul struktúrájába, egyúttal példát mutat arra, hogy a felhasznált technológiák hogyan illeszkednek ebbe a struktúrába.

Végül kitértem a biztonsági megoldások gyakorlati alkalmazására is. Mivel nem találtam kielégítő megoldást, erre a kérdésre az implementáció során kisebb hangsúlyt fektettem. Az elért eredmények a „4.5 A biztonság kérdése” fejezetben olvashatók.

5.2 További lehetőségek

A „4.4 Kihívások, akadályok” fejezet szerint számos olyan aspektusa van a dolgozatban ismertetett koncepció gyakorlati megvalósításának, amelyek az implementáció során nem bizonyultak kielégítőnek. Például a template rendszer helyett a kötöttsége miatt érdemes lenne inkább REST alapú megoldásban gondolkodni, illetve legalább megszüntetni az alkalmazás jelenlegi kötöttségét, miszerint egyszerre csak egy felhasználó tudja azt használni. Egy valós üzleti alkalmazásnak több felhasználót és nagyobb rugalmasságot kell támogatnia. Ugyan a célját ellátja, de szükségtelenül körülményesnek tűnik a körkörös referencia elkerülésére biztosított megoldás is, erre érdemes lenne további alternatívákat keresni.

A példaalkalmazásban megvalósított biztonsági intézkedések hiányosak. Jelenleg a külső modulok nem tudják az alkalmazást leállítani, összeomlasztani, fájlokat módosítani vagy hálózati hozzáférést szerezni, illetve az alkalmazás belső osztályaihoz sem tudnak hozzáférést szerezni. Ezzel azonban még nem vagyok megelégedve, további szűrőfeltételeket szeretnék megadni tovább szűkítve a szkriptek hozzáférését. A távlati cél a teljes sandbox környezet biztosítása. Ehhez azonban olyan módszerek szükségesek, amelyek túlmutatnak e dolgozat keretén.

További lehetőségként tekinthetünk a példaalkalmazás újabb potenciális funkcióira is. Számos olyan funkció képzelhető el, amelynek az alkalmazás valós üzleti környezetben hasznát venné. Ilyen lehet a szálbiztosság, hogy több felhasználó egyidejűleg vihessen fel pontszámokat, vagy a gyorsítótárazás, amelyet a dinamikus betöltések miatt nehéz megvalósítani, de a performancia szempontjából elkerülhetetlen. Azonban a példaalkalmazás célja a dolgozatban ismertetett koncepció gyakorlati megvalósítási lehetőségeinek bemutatása, így nem célom az alkalmazást további funkciókkal bővíteni.

Ugyanakkor amennyiben a biztonság távlati célja adott lenne, az alkalmazásban használt módszereket érdemes lenne általánosítani és egy keretrendszer formájában publikálni tekintettel arra, hogy ilyen jellegű keretrendszer jelenleg nem létezik.

IRODALOMJEGYZÉK

[1] Az OpenJDK dokumentációja a Java modulrendszerről (openjdk.java.net/projects/jigsaw)

[2] A Wikipédia definíciója a programra (hu.wikipedia.org/wiki/Program_(informatika))

[3] A Wikipédia definíciója a moduláris programozásra (en.wikipedia.org/wiki/Modular_programming)

[4] Hartyányi Mária (2010): *Programtervezési módszertan*, Centroszet Szakképzés-szervezési Nonprofit Kft., Gödöllő, 6.3. fejezet (centroszet.hu/tananyag/program/63_modulris_programozs.html)

[5] A PCFórum.hu definíciója a pluginra (pcforum.hu/szotar/plugin)

[6] Brian Agnew a ScriptEngine-ről (2009), StackOverflow (stackoverflow.com/a/1017201)

[7] Oracle JavaSE 8 dokumentáció: Nashorn and Shell Scripting (docs.oracle.com/javase/8/docs/technotes/guides/scripting/nashorn/shell.html)

[8] Max Rhode “Unable to block access to NashornScriptEngine” című issue-ja a Java Delight “Nashorn Sandox” projektjéhez, GitHub, 2018 (github.com/javadelight/delight-nashorn-sandbox/issues/73)

[9] Robert Petermeier “Is *this* really the best way to start a second JVM from Java code?” című kérdése, StackOverflow, 2009 (stackoverflow.com/q/1229605/5539917)

[10] Oracle JavaSE 8 dokumentáció: Java (parancssori utasítás) (docs.oracle.com/javase/8/docs/technotes/tools/windows/java.html)

[11] Oracle JavaSE 6 dokumentáció: Security Manager (docs.oracle.com/javase/6/docs/api/java/lang/SecurityManager.html)

[12] Wikipédia cikk a Java Applet-ekről (en.wikipedia.org/wiki/Java_applet)

[13] A “DNA” nevű felhasználó a SecurityManagerről (2012), StackOverflow (stackoverflow.com/a/9482401)

[14] Gunnar Morling (2017): Accessing private state of Java 9 modules (in.relation.to/2017/04/11/accessing-private-state-of-java-9-modules)

[15] Nicolai Parlog a Java modulrendszer reflektív hozzáférési korlátozásairól (2016), StackOverflow (stackoverflow.com/a/36517269)

[16] Paul Deitel (2017): Understanding Java 9 Modules, In Java Magazine, 2017. szeptember/október, p18 (oracle.com/corporate/features/understanding-java-9-modules.html)

[17] Java 9 Modules - The Unnamed Module, LogicBig.com, 2018 (logicbig.com/tutorials/core-java-tutorial/modules/unnamed-modules.html)

[18] Java 9 Modules - Automatic Modules, LogicBig.com, 2018 (logicbig.com/tutorials/core-java-tutorial/modules/automatic-modules.html)

[19] Az “Ubershmekel” nevű felhasználó petíciója az IntelliJ IDEA Ctrl-Y billentyűkombináció viselkedésének megváltoztatásáról (intellij-support.jetbrains.com/hc/en-us/community/posts/115000125250-Petition-to-change-Ctrl-Y-default-binding-to-redo-on-Windows)

[20] Vishal Verma: Functional Programming Paradigm (geeksforgeeks.org/functional-programming-paradigm)

[21] Mark Reinhold (2016): The State of the Module System – Automatic Edition, Oracle, 3.4 fejezet (openjdk.java.net/projects/jigsaw/spec/sotms/#bridges-to-the-class-path)

[22] Moshe Sayag (2017): Split Packages on Java 9 Modules (msayag.github.io/SplitPackage)

[23] Nicolai Parlog (2017): Five Command Line Options To Hack The Java Module System (blog.codefx.org/java/five-command-line-options-hack-java-module-system)

[24] Wikipédia cikk a körkörös referenciáról (en.wikipedia.org/wiki/Circular_dependency)

Linkek utoljára ellenőrizve: *2019. április 26.*

SZÓJEGYZÉK

Bootstrap: A Bootstrap a legnépszerűbb CSS könyvtár, amely reszponzív weboldalak készítését teszi lehetővé (getbootstrap.com)

Class loader: Osztályok, ált. class fájlok betöltését végzi, egy absztrakciós réteg a fájlrendszer felett. Ahhoz, hogy bármilyen forrásfájlból futtatható kód kerüljön a JVM-be, class loader-re van szükség. A JRE alapértelmezetten három class loader-t használ, és igény szerint újabbak készíthetőek.

Class path: A module path megjelenése előtt a Java fordító minden, a fordításkor megadott osztályt egy virtuális gyűjtőbe ömlesztette. Nem volt tekintettel az osztályok származására vagy a csomagnevekre. Amikor egy osztályt be kellett tölteni, a JVM ebből a gyűjtőből kereste ki a megfelelő osztályt.

Deferred execution: Halasztott végrehajtás, a funkcionális programozással kapcsolatban merül fel. Amikor egy metódust meghívunk olyan paraméterrel, amely egy kifejezés kiértékeléséből származik, két lehetőségünk van. Vagy kiértékeljük a kifejezést helyben, és az eredményt adjuk át a metódusnak, vagy magát a kifejezést adjuk át a metódusnak. Utóbbi esetben beszélünk halasztott végrehajtásról, a kifejezés kiértékeléséről már a metódus gondoskodik, akár el is tekinthet tőle.

Derby: Alternatív Java nyelven írt, beépíthető RDBMS (db.apache.org/derby)

Eclipse Link: Alternatív JPA alapú ORM keretrendszer (eclipse.org/eclipselink)

Factory design pattern: A Factory egy absztrakciós szintet húz az objektum példányosítás felé, így az objektumok példányosításakor az implementációs részletek elrejthetőek.

Flexbox: A CSS3 hivatalos megoldása a reszponzív weboldalak készítésére.

Gradle: Alternatív szoftverprojekt menedzselő eszköz (gradle.org)

H2: Java nyelven írt, beépíthető RDBMS (h2database.com)

Hibernate: Egy a JPA-t implementáló ORM keretrendszer (hibernate.org)

Hollywood elv: „Don't call us, we'll call you” – a programozás világában „Inversion of control” (IoC) néven szokás emlegetni. A procedurális programozási paradigma helyett a program elemei, mint komponensek egy keretbe illeszkednek, és ezeket felülről egy erre rendeltetett komponens szólítja meg.

HSQL: Alternatív Java nyelven írt, beépíthető RDBMS (hsqldb.org)

JMX: Java Management Extensions – Java technológia erőforrások menedzselésére.

JPA: Java Persistence API – egy specifikáció a relációs adatmodellek Javában történő menedzseléséhez.

Jquery: A legnépszerűbb JavaScript függvénykönyvtár a DOM hierarchia bejárására, módosítására (jquery.com)

JSF: JavaServer Faces – eredetileg a JSP-re alapuló, azt kiegészítő komponens alapú view leíró (javaserverfaces.org)

JSP: JavaServer Pages – a Java nyelvhez eredetileg készített view leíró (oracle.com/technetwork/java/jsp-138432.html)

Launcher: A launcher egy alkalmazás, amelynek a segítségével egy másik alkalmazás indítható el.

Maven: Szoftverprojekt menedzselő és fordítás automatizáló eszköz (maven.apache.org)

Modal: A modal egy ablak, amely szorosan kötődik egy szülőablakhoz. A szülőablak előtt jelenik meg, és letiltja az azzal való interakciót, amíg a felhasználó a modal-t be nem zárja.

Module path: A module path a megjelenése óta a class path-al párhuzamosan létezik. Ha a fejlesztő a csomagjait modulokba szervezi, vagy egyes jar fájlokat expliciten hozzáad a module path-hoz, akkor a bennük lévő osztályok a module path-ra kerülnek. A JVM minden modult külön kezel, mindegyikhez külön hozzáférési megkötések tartoznak, és egy-egy csomagot csak a megfelelő modulon belül keres.

Objektum proxy: A proxy egy objektum, amely úgy viselkedik, mint egy másik objektum. A másik objektumot elfedheti vagy helyettesítheti. A proxy pl. új funkcionalitást adhat hozzá a meglévő objektumhoz, vagy utólagos validációt biztosíthat hozzá. A dolgozatban a proxy a Hibernate keretrendszerrel kapcsolatban merült fel, ahol a szerepe a lusta inicializálás. A proxy adattagjai csak akkor kerülnek feltöltésre az adatbázisból, amikor azokat elsőként lekérdezik.

ORM: Object-Relational Mapping, technika, amellyel egy adatbázis elemei leképezhetőek az objektum-orientált világba.

POJO: „Plain Old Java Object” – a klasszikus értelemben véve azokat a Java osztályokat értjük alatta, amelyekben csak adattagok és azokhoz setter-ek, getterek vannak, ezen felül semmilyen adattagot vagy metódust nem tartalmaznak. Tágabb értelemben véve az adattárolásra vagy adatmozgatásra használt végtelenül egyszerű osztályok.

RMI: Remote Method Invocation – a távoli parancsvégrehajtás (RPC) egy implementációja, melyben szerializált objektumokkal történik a kommunikáció.

Spring Boot: Eszköz a Spring keretrendszernek és elemeinek egyszerű összeállításához (spring.io/projects/spring-boot)

Spring Data: Eszköz az adatkezelés megkönnyítésére (spring.io/projects/spring-data)

Spring Web MVC: Egy az MVC struktúrát implementáló keretrendszer a HTTP kérések feldolgozására (docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/mvc.html)

Spring: Alkalmazás keretrendszer számos kiegészítéssel a különböző területekhez (spring.io)

Thymeleaf: Modern view leíró (template motor) (thymeleaf.org)

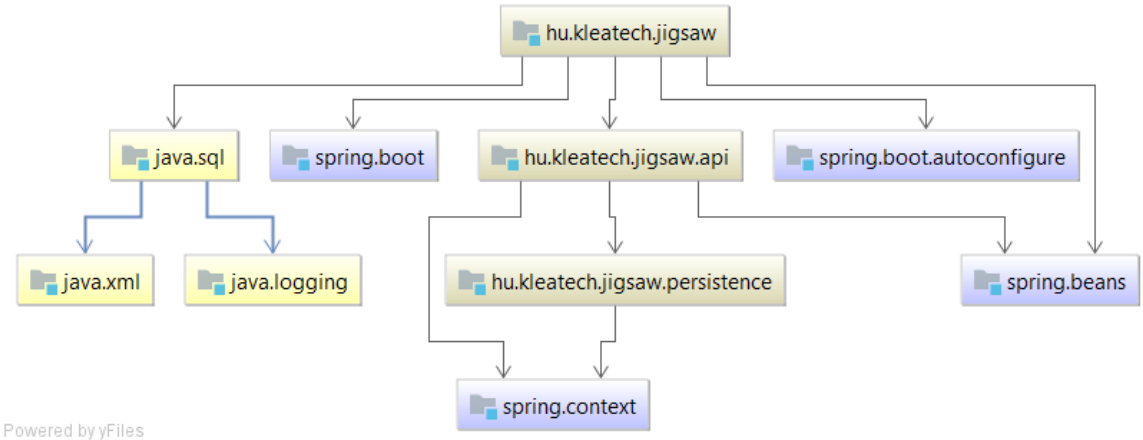
Toast: A Toast egyfajta felugró ablak, amely információs célt szolgál, és nem igényli a felhasználó azonnali beavatkozását. Általában magától eltűnik egy idő után.

View resolving: Az MVC struktúra szerinti Controller feltölti a Model-t adatokkal, a View pedig ezekhez az adatokhoz megjelenítési struktúrát (template-et) definiál. A template nyelve különféle lehet. A view resolving ennek a template-nek a lefordítását jelenti html-re.

1. MELLÉKLET – JAVA MODULRENDSZER PÉLDAPROJEKT

Példaprojekt annak szemléltetésére, hogy a Java 9 modulok hogyan működnek, illetve hogyan működnek együtt a Maven és a Spring technológiákkal. Alább látható a példaprojekt modul diagramja. Ehhez a projekthez egy egyszerű, kevés osztályból és kevés rétegből álló struktúrát készítettem, de ez alapján már kellő bonyolultságú struktúra is kialakítható. A projekt kódja ezen a linken található:

https://gitlab.com/kleavenae/JigsawExample



2. MELLÉKLET – TWMAS

Természetesen nem én vagyok az egyetlen, aki azon gondolkodik, hogyan lehet Javában sandbox környezetet létrehozni. Annak ellenére, hogy ez egy logikus kérdés, nagyon kevés említést találtam róla az interneten. Jelenleg legjobb tudomásom szerint nincs olyan keretrendszer, ami sandbox jelleggel, biztonságosan pluginokat tud Java programba beépíteni, és még aktív fejlesztés alatt áll. Ha ez utóbbi feltételt kivesszük a képletből, akkor egy keretrendszert találhatunk. Ez a *The World's Most Advanced Sandbox™ (TWMAS)*. Ez a keretrendszer platformfüggetlen és biztonságos, habár nem kifejezetten pluginokhoz készült. Sajnos a projektet már 6 éve nem fejlesztik, de a hozzá tartozó dokumentum rendkívül hasznos lehet, ha valaki Java sandox témában gondolkodik. Ezért ez a mellékletet ennek a keretrendszernek, pontosabban a hozzá mellékelt dokumentumnak szentelem.

Weboldal: https://github.com/lihaoyi/6858

Dokumentum: *Li Haoyi, Tim Kaler, Frank Li, Ivan Sergeev: The Word's Most Advanced Sandbox™*

Link a dokumentumhoz: https://docs.google.com/document/d/1-gFHzZR0X8cDG6CWDgktRAs4pcvxHOQKFiUZ9_2mUhE

Abstract: To safely run untrusted Java code, the untrusted code should be sandboxed. Traditional sandboxes rely on non-portable operating system facilities to run the code in separate processes with restricted privileges, which may also incur a high performance overhead. In this report, we introduce TWMAS, a sandbox that does not make use of the operating system. Instead, it uses the Java SecurityManager and instruction-rewriting to safely sandbox untrusted code directly running within a single host Java Virtual Machine.

3. MELLÉKLET – FRAGMENTEK

Ebben a mellékletben két példa fragmentet mutatok be. Ezek lényeges részét képezik a dolgozatnak, de terjedelmük okán mellékletként idézem be őket. További információ a „4.2.5 A UI formálása kívülről” fejezetben található.

```
<!-- RoundModalAddOther.html -->
<div class="modal fade" draggable="true" id="roundModalAddOther" tabindex="-1" role="dialog"
     aria-labelledby="exampleModalLabel" aria-hidden="true" th:fragment="fragment">
  <div class="modal-dialog modal-lg" role="document">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title" id="exampleModalLabel">New round</h5>
        <button type="button" class="close" data-dismiss="modal" aria-label="Close">
          <span aria-hidden="true">&times;</span>
        </button>
      </div>
      <div class="modal-body">
        <form id="formRound" th:competition="${compSelected.id}" addr="" th:object="${pojo}"
              method="post">
          <div class="table-responsive">
            <table class="table table-sm">
              <tr>
                <td><input type="number" th:field="*{first}"/></td>
                <td><input type="number" th:field="*{second}"/></td>
                <td><input type="number" th:field="*{third}"/></td>
              </tr>
            </table>
          </div>
        </form>
        <button onclick='$("#roundModalAddOther").removeClass("fade");
                         $("#roundModalAddOther").modal("hide");
                         $.post($("#formRound").attr("addr"),
                          $("#formRound").serialize(),
                          function(html) {retrieveCompetition($("#formRound").attr("competition"))});'
                class="btn btn-primary">
        Save</button>
      </div>
      <div class="modal-footer">
        <button type="button" class="btn btn-secondary" data-dismiss="modal">Cancel</button>
      </div>
    </div>
  </div>
</div>
```

A RoundModalAddOther.html egy Round adatbeviteli fragmentet definiál. Bootstrap Modal-t használ. A „formRound” id-val rendelkező form-hoz felveszek egy „competition” és egy „addr” attribútumot. A „competition” attribútum az aktuális Competition azonosítóját tárolja, az „addr” attribútum pedig a megfelelő címet, ahová a HTTP kérést küldeni kell (attól függően, hogy adatfelvétel vagy módosítás történik). A form-ban feltöltöm a POJO objektumot, a „Save” gombra kattitva pedig elküldöm. A kód többi része csak a Modal kezeléséhez szükséges.

```
<!-- CompetitionFragmentOther.html -->
<div th:fragment="fragment" class="revert">
  <div th:replace="roundModalAddOther_generated :: fragment"></div>
  <table class="table" th:onload="${'var compSelectedId = ' + compSelected.id + ';'}">
    <thead>
      <tr>
        <th colspan="4" th:text="${compSelected.name}"></th>
        <th colspan="3">Result</th>
      </tr>
    </thead>
    <tbody>
      <th:block th:each="participant : ${teamSelected.participants}">
        <th:block th:with="round=${participant.rounds.size()>0 ? participant.rounds.get(0) : null}">
          <tr th:if="${round==null || round.values.size()>2}">
            <td class="h5" th:text="${participant.name}"></td>
              <th:block th:if="${round!=null}">
                <td th:text="${round.values.get(0)}"></td>
                <td th:text="${round.values.get(1)}"></td>
                <td th:text="${round.values.get(2)}"></td>
                <td th:text="${#numbers.formatDecimal(round.result(func), 0, 'COMMA', 2, 'POINT')}"
                    class="align-middle"></td>
                <td>
                  <button type="button" class="btn btn-primary" th:onclick="${'jQuery.' +
                            js.func('get')
                                .argStr('/deleteRound/' + round.id)
                                .argObj('function(html) {
                                          retrieveCompetition(' + compSelected.id + ')}')}">
                    Clear</button>
                  </td>
                </th:block>
              <td>
                <button type="button" class="btn btn-primary" data-toggle="modal"
                        data-target="#roundModalAddOther" th:onclick="${
                          js.queryById('formRound')
                            .func('attr')
                              .argStr('addr')
                              .argStr(round==null
                                            ? '/addRound/' + compSelected.id + '/' + participant.id
                                            : '/editRound/' + round.id)}">
                  Set</button>
                </td>
              </tr>
          </th:block>
      </th:block>
    </tbody>
  </table>
</div>
```

A CompetitionFragmentOther.html fragment egy Competition-ön belül a Round-ok megjelenését definiálja. A fragment betöltésekor beállítom a „compSelectedId” globális változót, ami az aktuális Competition azonosítóját tartalmazza. Egy-egy Round-ot egy-egy Participant-nak feleltetek meg. Ha a Participant-hoz még nem tartozik Round, akkor a „Set” gomb hozzáadásként működik, és a „Clear” gomb eltűnik. Ha tartozik hozzá Round, akkor a „Set” gomb felülírásként működik és a „Clear” gomb megjelenik, amivel törölni lehet a Round-ot.

Ez a Competition részeredményt nem, csak eredményt használ. Az eredményt a „#numbers” Thymeleaf utility metódussal formázom meg. A JavaScript összeállítását egyedi utility metódusokkal végzem, amelyek megkönnyítik az egymásba ágyazott különleges karakterek (', ”, #, $) kezelését.

4. MELLÉKLET – BIZTONSÁGOS SZKRIPT-BETÖLTŐ

Ebben a mellékletben bemutatom a „4.5 A biztonság kérdése” fejezetben ismertetett megoldáshoz kapcsolódó programkód működését.

```
//SecureEngine.java osztály részlete
public Function<List<Double>, List<Double>> preresults(String filename) throws FileNotFoundException,
                                                                        MyScriptException, ClassCastException
{
    IsolatedThread<Function<List<Double>, List<Double>>> isolatedThread = new IsolatedThread<>(
        RequestType.PRERESULTS, resolveEngine(filename), resolveReader(filename));

        try {
            isolatedThread.start();
            return isolatedThread.result.get(5, TimeUnit.SECONDS);

        } catch (TimeoutException e) {
            isolatedThread.interrupt();
            throw new MyScriptException("Timeout while running external script");
        } catch (InterruptedException | ExecutionException e) {
            throw new MyScriptException(
                        "Interruped while running external script: " + e.getMessage());
        } catch (ClassCastException e) {
            throw new MyScriptException(e.getMessage()); }}
```

Itt az a metódus látható, amelyet a Dispatcher-en keresztül a keretprogram osztályai használhatnak a részeredményfüggvény lekérésére. A SecureEngine létrehoz egy IsolatedThread példányt olyan generikus típusparaméterrel, ami megfelel az adott metódus (itt „preresults”) visszatérési típusának. Átadja neki paraméterként a lekérés típusát (itt „PRERESULTS”), és a lekérés típusában definiált szignatúrának megfelelő paramétereket (itt ScriptEngine és FileReader). Ezután elindítja a szálat, majd az isolatedThread result adattagjának (ami egy PollingFuture) a get metódusával blokkolja az aktuális szálat addig,

```
//IsolatedThread.java osztály részlete
public final PollingFuture<T> result = new PollingFuture<>();

public void run() {
    sm.enable();
    try {
        runUntrustedCode();
    } finally {
        sm.disable(pass);
}}

private void runUntrustedCode() {
    try {
        String className = "hu.kleatech.jigsaw.scripting.inner.EngineInner";
        Object scriptResult = loader.loadClass(className)
                                    .getMethod(requestType.methodName, requestType.signature)
                                    .invoke(null, params);
        result.set((T) scriptResult);

        } catch (Throwable t) {
            t.printStackTrace();
            throw new MyScriptException("Error when running external script: " + t.getMessage());
}}
```

amíg az isolatedThread értéke elérhető nem lesz, de legfeljebb 5 másodpercig. Ha letelik az 5 másodperc, megszakítja a szálat és dob egy kivételt.

Az IsolatedThread bekapcsolja a MySecurityManager-t, lefuttatja a szkriptek betöltéséért felelős kódot, majd kikapcsolja a MySecurityManager-t. A futtatáshoz először betölti az EngineInner osztályt a MyClassLoader-rel. Ezután meghívja rajta a kérés típusából adódó metódust. Végül beállítja a PollingFuture értékét.

```
//MySecurityManager.java osztály részlete
private volatile InheritableThreadLocal<Boolean> enabled = null;
private final Object secret;

MySecurityManager(Object pass) {
    secret = pass;
    initializeEnabled();
}
private void initializeEnabled() {
    enabled = new InheritableThreadLocal<Boolean>() {
        @Override protected Boolean initialValue() { return false; }
        @Override public void set(Boolean enabled) { super.set(enabled); }
};}

void disable(Object pass) { if (pass == secret) enabled = null; }

void enable() {
    if (enabled == null) initializeEnabled();
    if (System.getSecurityManager() != this) System.setSecurityManager(this);
    enabled.set(true);
}

private void deny(String msg, Object... cause) {
    if (enabled != null && enabled.get()) throw new SecurityException(msg);
}

private void allow(Object... cause) {}
```

A MySecurityManager a konstruktorában kap egy jelszót, csak ennek a segítségével lehet kikapcsolni. A MySecurityManager-nek van egy InheritableThreadLocal adattagja, amely azt biztosítja, hogy a SecurityManager csak az adott szálon és annak gyermek száljain működjön. A MySecurityManager-t vezérlő mechanizmus úgy lett megvalósítva, az adott szálról bekapcsolva csak az adott szálra vonatkozzon, azonban globálisan kikapcsolható legyen tetszőleges szálról (de csak a jelszó birtokában). Ezt azért vezettem be, hogy ha egy hibás szkript feltartja az eredeti szálat, akkor egy másik szálról is leállítható legyen a SecurityManager.

A MyClassLoader először megnézi, hogy a betöltendő csomag Java belső csomag-e, ezek a „java.” kezdetű csomagok, ezeket csak az eredeti ClassLoader töltheti be. Ha egy csomag vagy osztály kifejezetten megbízhatónak lett jelölve, azt is betölthetjük a gyári ClassLoaderrel. Ha egy osztály vagy csomag kifejezetten tiltott, annak megtagadhatjuk a betöltését. Ha pedig egy osztály sem az engedélyezett, sem a tiltott listán nem szerepel, akkor manuálisan töltjük be. A loadClassData metódusban lehetne a bájtkód tartamát ellenőrizni tiltott tevékenységet keresve. De ez már nem témája a dolgozatnak.

```
//MyClassLoader.java osztály részlete
@Override public Class<?> loadClass(String name) throws ClassNotFoundException {
    //If the class is precomiled by the JDK and is whitelisted we can load it here
    String pkg = Class.forName(name).getPackageName();
    if (pkg.startsWith("java") ||
        allowedClassNames.contains(name) ||
        allowedPackages.contains(pkg)) {
        return super.loadClass(name);
    } else if (deniedClassNames.contains(name) ||
               deniedPackages.contains(pkg)) {
        throw new SecurityException("Class denied: " + name);
    }
    //If the class is not on the whitelist we load it manually
    return findClass(name);
}
@Override public Class findClass(String name) {
    byte[] b = loadClassData(name);
    return defineClass(name, b, 0, b.length);
}
private byte[] loadClassData(String name) {
    InputStream is = getClass().getClassLoader().getResourceAsStream(name.replace(".", "/")+".class");
    ByteArrayOutputStream byteSt = new ByteArrayOutputStream();
    int len;
    //Checks for illegal actions in the bytecode can be placed here
    try {
        while((len=is.read())!=-1) {
            byteSt.write(len);
        }
    } catch (IOException e) {
        throw new MyScriptException(
                    "Error when converting external script to bytecode: " + e.getMessage());
    }
    return byteSt.toByteArray();
}
```

Alább a RequestType enum részlete látható. Egy-egy RequestType egy-egy külső függvénynek felel meg, tartalmazza annak a nevét és szignatúráját.

```
//RequestType.java enum részlete
PRERESULTS("preresults", ScriptEngine.class, FileReader.class),
RESULT("result", ScriptEngine.class, FileReader.class);

public final String methodName;
public final Class[] signature;

private RequestType(String methodName, Class... signature) {
    this.methodName = methodName;
    this.signature = signature;
}}
```

Alább az EngineInner osztály részlete látható. Ez az osztály csak azt a célt szolgálja, hogy a szkriptek betöltését egybefogja statikus metódusok formájában, hogy könnyebb legyen az egyedi ClassLoader-rel betölteni.

```
//EngineInner.java osztály részlete
public static Function<List<Double>, List<Double>> preresults(ScriptEngine engine, FileReader file)
                                                        throws MyScriptException, ClassCastException {
    try {
        engine.eval(file);
        Invocable inv = (Invocable) engine;
        return inv.getInterface(Function.class);

    } catch (ScriptException e) {
        throw new MyScriptException(
                    e.getMessage(), e.getFileName(), e.getLineNumber(), e.getColumnNumber());
}}
```

A PollingFuture azért került bevezetésre, hogy kényelmesen, flag-ek használata nélkül megvalósulhasson a kommunikáció két szál között. Fentebb a SecureEngine kódjában látható, hogy a SecureEngine létrehoz egy IsolatedThread-et, majd meghívja az IsolatedThread adattagjaként tárolt PollingFuture get metódusát. Ezt megteheti, hiszen a PollingFuture már ekkor is létezik, csak ekkor még „üres”. A get metódus meghívásával megvárja, amíg lesz benne érték, majd visszatér vele.

```
//PollingFuture.java osztály részlete
private volatile T result;

synchronized void set(T result) {
    if (this.result != null) throw new IllegalStateException("Result can only be set once");
    this.result = result;
}
@Override public T get(long l, TimeUnit tu) throws InterruptedException, ExecutionException,
                                                    TimeoutException {
    long timespan = tu.toMillis(l);
    while (result == null) {
        Thread.sleep(pollInterval);
        timespan--;
        if (timespan <= 0) throw new TimeoutException();
    }
    return result;
}
```

A CD MELLÉKLET TARTALMA

A CD melléklet tartalmazza ezen dolgozatot, és a dolgozatban bemutatott példaalkalmazást. A CD melléklet a következő katalógusokat és fájlokat tartalmazza:

- Dolgozat
  - dolgozat.odt
  - dolgozat.pdf
  - kiírás.pdf
  - magyar absztrakt.odt
  - magyar absztrakt.pdf
  - english abstract.odt
  - english abstract.pdf
- Példaalkalmazás forráskód
  - Az alkalmazás Java moduljai egy-egy katalógusban
  - modules katalógus, amely a külső modulokat tartalmazza
  - pom.xml a Mavennel történő fordításhoz
  - readme.txt amely leírja a fordítás és futtatás parancsát
- Példaalkalmazás futtatható
  - modules katalógus, amely a külső modulokat tartalmazza
  - data.sql amely néhány példaadattal tölti fel az adatbázist
  - exec.jar amely a teljes alkalmazást tartalmazza minden függőségével
  - readme.txt amely futtatáshoz szükséges parancsot tartalmazza