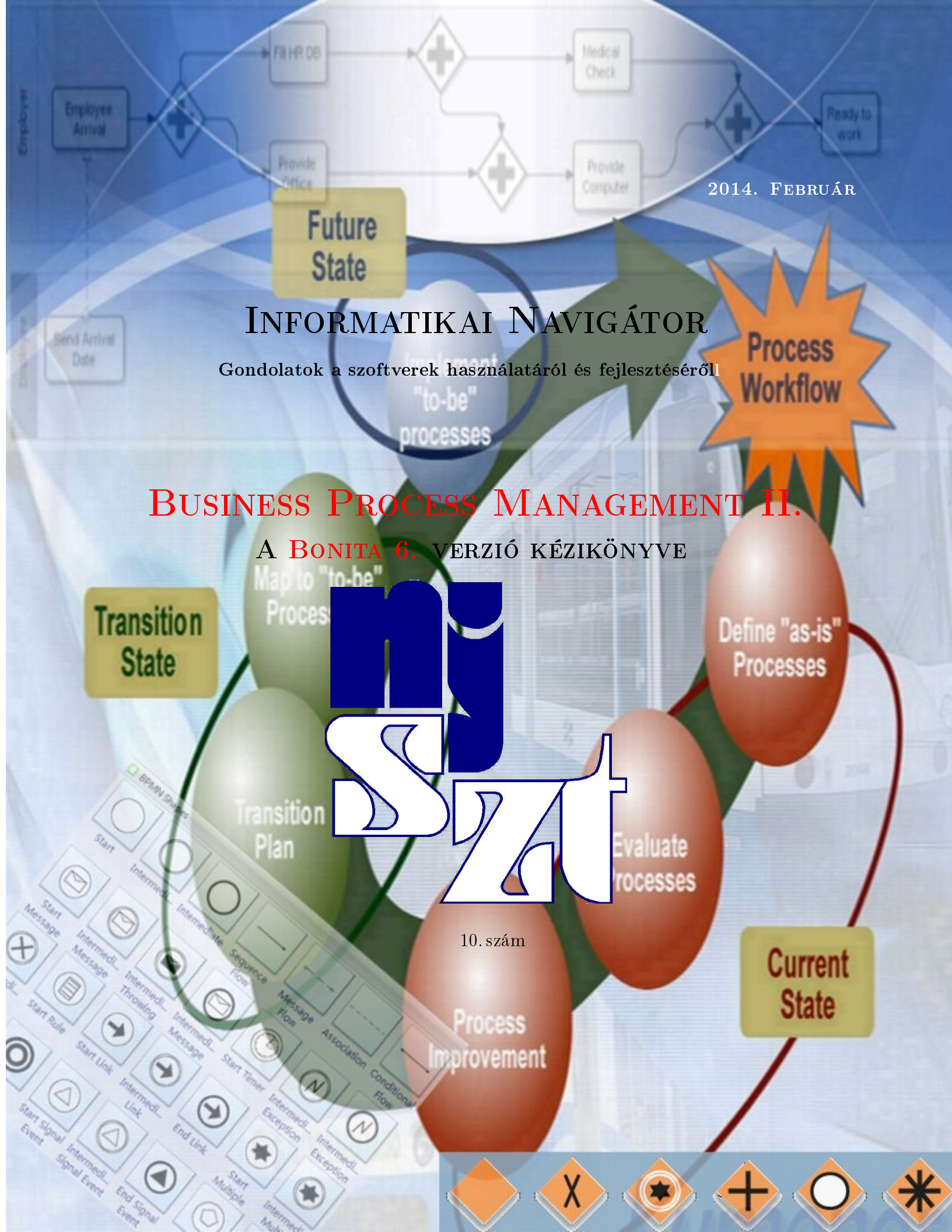
2014. Február

# Informatikai Navigátor

Gondolatok a szoftverek használatáról és fejlesztéséről

## Business Process Management II.

### A Bonita 6. verzió kézikönyve




10. szám

# Tartalomjegyzék

1. Bonita 6. workflow alkalmazások tervezési lehetőségei 3
2. Task és Actor 16
3. A Form designer eszközeinek használata 36
4. A Bonita 6. portál áttekintése 78
5. Alkalmazás fejlesztés a Bonita Stúdió 6. használatával 93
6. Az események használata a BPMN modellekben 113
7. A Bonita 6. API használata 130
8. Bonita 6. konnektorok áttekintése 154
9. Saját Bonita konnektor készítése 172
10. Bonita 6. Actor Filter készítés 178
11. A Bonita 6. Web Restful API 183
12. A Bonita 6. produktív környezet telepítése 198
13. A Bonita Business Activity Monitoring 212
14. Bonita 6. - Tippek, Trükkök és néhány apróság 221

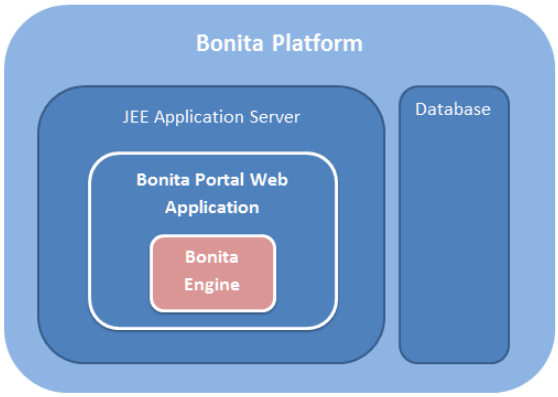
Főszerkesztő: Nyiri Imre (imre.nyiri@gmail.com)

# 1. Bonita 6. workflow alkalmazások tervezési lehetőségei

Az Informatikai Navigátor 5. száma a Business Process Management és a Bonita 5.x verzió részletes bemutatását kínálja nekünk. Ebben a fejezetben helyenként megismételjük az ott írtakat is, főleg azzal a céllal, hogy a Bonita 6.x új vagy megváltozott lehetőségeit jobban elmagyarázhassuk. Emiatt ezen kiadvány az 5. számmal együtt olvasva ad egy teljes és részletes képet a Bonita 6.x alkalmazásáról, a vele végezhető fejlesztési lehetőségekről.

## Rövid áttekintés a változásokról

A Bonita 6.x jó néhány újdonságot hozott, ezért ezen fejezet célja ezeket bemutatni, ahol a kifejtés hosszabb lélegzetű részt igényel, azokat a következő fejezetekben találhatjuk meg részletesen. A Bonita 2 fő szoftver komponensből áll: Stúdió és Platform. Az előző a fejlesztést teljeskörűen támogató, Eclipse alapú környezet.




1.1. ábra. A Bonita 6 motor felépítése

A Bonita Platform felépítését a 1.1. ábra mutatja. A környezet 2 infrastruktúra komponensből áll:

- Egy JEE (Java Enterprise Edition) szerver
- Egy adatbázis szerver

A Bonita portál és a workflow motor is a JEE szerverbe van telepítve, az adatbázisban pedig az objektumokat perzisztálja a rendszer.

### Context helyett Parameters

Az 5.x *Context* egy olyan elem volt, ami segítette a futó WF alkalmazás környezeti információihoz való elérését (pl. adatbázis név, egy webhely neve). Ennek hasonló korlátozásai vannak, mint az Actor Selectornak, így a 6.x verziótól a *Parameters*-t használjuk. A *Parameter* egy konfigurációs információs csomag, ami képes kapcsolódni a process-hez. A *Parameter* egy-egy *Environment*-hez van konfigurálva és értékeit futási idő (runtime) alatt lehet update-elni. Az *Environment* egy új elem, ami az 5.x verziókban nem létezett, célja, hogy minden processre egybefogja ezeket az információkat:

- Actor Mapping
- Actor Filter
- Parameters
- Connector implementációk

- A külső *jar* fájl függőséges (process-re és alkalmazásra)

Amikor a Bonita Stúdióban konfigurálunk egy process-t, akkor paraméter értékeket rendelhetünk hozzá. Ezek olyan értékek, amik deployment-enként változhatnak. A *Performance Edition* azt is lehetővé teszi, hogy ezeket az értékeket telepítés után is megváltoztassuk. Ilyen paramétereket 2 lépésben hozhatunk létre a stúdió segítségével:

1. Egy process paraméter definíció létrehozása
2. A paraméter konfigurálása

Egy *Pool* (medence) vagy *Lane* (sáv) számára lehet ilyen paramétereket meghatározni és ilyen típusú lehet: Text, Integer, Double, Boolean. Attól függően, hogy Pool vagy Lane szinten akarunk-e definiálni egy paramétert, jelöljük ki azt, majd válasszuk ki a *General* → *Parameters* fület. A már megadott paraméterek táblája ilyenkor láthatóvá válik. Nyomjuk meg az *Add* gombot, majd töltsük ki a definíciós adatokat:

- A paraméter neve
- Típusa (egy drop-down list vezérlőből választható)
- A paraméter rövid leírása, itt a célját érdemes megadni

Ezeket az értékeket a későbbiekben változtathatjuk, a paramétert akár törölhetjük is. Ez volt eddig a process paraméter definíció létrehozása. Nézzük meg a paraméter konfigurációját is! Egy paraméter értékét ezekkel a lépésekkel lehet megadni, miután megnyitottuk a process diagramot:

1. A Cool bar *Configure* gombját nyomjuk meg és válasszunk egy környezetet
2. Válasszuk ki a *Parameters* menüpontot, amire megjelenik egy táblázat, ahova a megfelelő értéket beírhatjuk, majd nyomjuk meg a *Finish* gombot.

A paraméter *Java* vagy *Groovy* kóddal így érhetjük el:

```
Object o = getInputParameter(String paramName)
```

A másik fontos fogalom az *Environments*. Egy környezet arra szolgál, hogy a projekthez kötődő beállításokat eltároljuk. Egy process-t mindig összeköthetünk valamelyik környezettel a fejlesztés során. Egy process több környezettel is képes futni, amikor azt teszteljük. Tekintettel arra, hogy egy environment a process-hez kötődő fogalom, így annak megadása így lehetséges:

1. Válasszuk ki a pool-t, ami a processt adja meg
2. A CoolBar *Configure*, *Run* vagy *Debug* kiválasztásával adhatunk meg egy új környezetet

A konfigurálást egy varázsló fogja végigvinni. Amikor futtatjuk (vagy debug-oljuk) a process-t, akkor a megfelelő környezetet ki kell választanunk.

### Workspace API

Ez egy új funkcionalitás, a régebbi Bonitákban nem volt. A fejlesztők munkaterületeken dolgoznak és ezeket a fájlformátumokat tudják hordozni ezen API támogatása révén:

- .bar fájl → a szerverre való telepítés formátuma
- .bos fájl → 2 Bonita Stúdió közötti fájl formátum. Szerver telepítésre nem alkalmas.

Az API-hoz jár egy script, aminek a neve *BonitaStudioBuilder* és képes *bar* fájl építésre a Bonita BPM repository-ból. Mindehhez nem kell futnia a Bonita Stúdiónak, de telepítve kell lennie. Maga a script ilyen néven érhető el: *BonitaStudioBuilder.sh* és a Studio *workspace_api_scripts* könyvtárában található meg (itt Windows-os *bat* fájl is van). A cél az, hogy az *SVN* repository tartalma alapján a build folyamatot automatizálni lehessen úgy, hogy ahhoz nem kell a stúdió sem. A script lehetséges paraméterei a következőek:

- *repoPath*: A SVN munkakönyvtár szülőkönyvtára, ami tipikusan a *trunk*.
- *outputFolder*: Ebbe a könyvtárba kerül az elkészített bar fájl.
- *processes*: A processek listája (vesszővel elválasztva), amikből bar-t szeretnénk építeni. Ez az általános formátuma: process,version;process,version...
  Példa: -process=TravelRequest,1.5;Expenses;LeaveRequest,1.0
- *buildAll*: Amennyiben *true* az értéke, úgy a *bar* fájlok mindig a legutolsó process alapján fognak épülni. Alapértelmezés: *false*.
- *withoutUI*: Alapértelmezett érték a *false*, de *true* esetén Window Manager nélkül is építhetjük a *bar* fájlokat.
- *environment*: Egy olyan környezet, amit a Stúdióból ismerünk.

A script a *workspace/.metadata/.log.* helyre naplózza a futását. Példa a script meghívására:

```
./BonitaStudioBuilder.sh -repoPath=/home/myBonitaRepoCheckedOutOfSVN -withoutUI=true -➥
    outputFolder=/home/bonita/myBARs -buildAll=true -environment=Qualification
```

Előtte bármilyen SVN klienssel szedjük ki a repository-ból a projektet és töröljük ki a *workspace* könyvtár (ez is a telepített Stúdió egyik könyvtára) tartalmát. A lehetséges környezetek: Productive, Qualification, Development.

### BPMN2 – A szabvánnyal való kompatibilitás frissítése

A parallel gateway lett a szabványhoz igazítva, azaz most már így működik:

- Minden beérkező TOKEN-t (azok bármilyen sorrendben jöhetnek) kötelező bevárni, addig nem léphet tovább a munkafolyamat a következő TASK-ra.
- Minden output TOKEN-t egyidejűleg és feltétel nélkül ad tovább.

### Változások a Bonita Stúdió-ban

- A *Tomcat* lett a beépített fejlesztői web container
- Az *Expression Editor* át lett tervezve (szerepe: itt kell például a Groovy scripteket is megadni)
- A TASK-ok beállításánál megjelent az *Iteration* fül. Itt lehet beállítani, hogy a TASK loooping vagy Multi Instancos legyen-e (ezen belül parallel vagy soros). Itt már nem kell kiegészítő Java osztályokat készíteni, hogy a működés a szabványnak megfelelő legyen. Ehelyett elég a workflow valamelyik belső, *Collection* típusú változóját adhatjuk meg.
- Jobb lett a WF belső változók létrehozásának támogatása
- A definíció és implementáció a konnektorok esetén is különvált
- Az *Exclusive Gateway* átmeneteinek sorrendje jobban kezelhető
- Egy új *Validation View* megoldás használható, ami sokat segít a process tesztelésben is
- Megjelent az *Organization* fogalma
- A névtelen (*Anonymous*) user használható, amikor az szükséges

A Bonita 6.x stúdióban a beépített tesztkörnyezet Tomcat-re épül. Ennek az az előnye, hogy ez jól hasonlít egy későbbi produktív környezethez. Gyorsabb lett ezzel a stúdió elindítása is, mert a Tomcat attól függetlenül futva indul el.

### Változások a Bonita Portal-ban

A Bonita 5.x User Experience helyett a 6.x verzió egy teljesen új, átdolgozott *Portal*-t vezetett be. Ennek van Mobile Web Portal verziója is, ami úgy lett tervezve, hogy a mobil telefon böngészőjével lehessen ergonomikusan használni. Ezenkívül ezek a nagyobb újdonságok kerültek be a 6.x verzióba:

- Javított TASK menedzsment: A felhasználók képesek más user-hez rendelni vagy visszakérni egy TASK-ot. Lehet kihagyni egy feladatot és ezt visszavonni. Megjegyzés fűzhető hozzá. Hiba esetén ismételten végrehajtható (replayed a failed task).
- *SUBTASK*: Bármelyik felhasználó létrehozhat ilyet és hozzárendelheti bárkihez (magához is).
- Dynamic Reconfiguration
- A TASK vagy konnektor ismétlése (mert például hiba volt)
- Business Data Search
- Auto Login (Anonymous user)

### A Bonita workflow engine javítása

Az engine teljesen át lett írva, egy új szolgáltatás alapú architektúra lett kialakítva. Az engine elérését biztosító API-n javítottak.

## A process tervezési metodológia rövid átismétlése

### A célok kitűzése

A legfontosabb talán mindig az, hogy tudjuk mit szeretnénk elérni, miért és mit tartunk fontosnak az üzleti folyamataink optimalizálásakor. Másfelől ezek megvalósítása során gondolnunk kell arra, hogy olyan mérhető pontokat tegyünk a folyamatba (KPI[1]), amik visszajelzik, hogy a folyamat céljai megvalósulnak-e vagy még további javításokat kell tennünk. A célkitűzések lehetnek mennyiségi és/vagy minőségi jellegűek. A Bonita BPM természetesen nem tud segíteni a célok kitűzésében, de a KPI megvalósításában és ellenőrzésében igen.

### A process scope pontos meghatározása

A folyamat hatályának meghatározása a következő pontok rögzítését jelenti:

1. Az *emberek szempontjából* nézve a fontos kérdések: Kié a folyamat? Ki használja a folyamatot? Kik a kulcsfelhasználók? Kinek kell ismernie, hogy ez a folyamat létezik? Mikor és hogyan kell használni a folyamatot? A folyamat gyakran használt? Van valamilyen ütemezése? Mit kell tenni vészhelyzet esetén? Egy vagy több ember használja ugyanabban az időben? A folyamat teljesen automatikus vagy azért van humán felhasználója?

2. Az *információ szempontjából* nézve a fontos kérdések: Milyen információkat kezel a process és ebből melyeket kell továbbítani külső rendszereknek vagy embereknek? Mik a folyamat elvárt eredményei?

3. Az *biztonság szempontjából* nézve a fontos kérdések: A folyamat milyen ellenőrzéseket végezzen? A hitelesítési és jogosultságkezelési megoldás milyen legyen?

4. Az *kapcsolatok szempontjából* nézve a fontos kérdések: Milyen kapcsolatok szükségesek a külső rendszerekhez? Küldjön-e e-mail valamely feltétel bekövetkeztekor? A külső rendszerek milyen integrációs technikákkal érhetőek el?

### A process diagram megrajzolása

A Bonita BPM segítségével a folyamatot gyorsan fel lehet rajzolni, meg lehet határozni a kiindulási, döntési, a tevékenység és befejeződési pontokat. Ezen szakasz feladata meghatározni a folyamat tulajdonosait és kulcs felhasználóit. Amíg ez nincs meg, addig nem szabad a megoldásban továbblépni. Mindig gondoljunk arra, hogy az üzleti folyamatoknak van egy vagy több kezdete és vége. Amennyiben a process rendelkezik párhuzamosan végzett ágakkal, úgy azok elejét és végét jelöljük egy gateway node-dal.

[1] KPI=key performance indicator (fő teljesítmény mutató)

### A process részleteinek meghatározása

Ezt a szakaszt implementációnak is nevezzük és a következő feladatok teljesítését jelenti:

- *Adatok meghatározása.* Minden step-re meg kell határozni az input és output adatokat. Fel kell állítani a workflow adatmodelljét, ami az adatok körét, típusát és egymással való kapcsolatának rögzítését jelenti. Itt szükséges specifikálni, hogy melyik adatnak mi a forrása, honnan kell azt megszerezni, illetve hova kell majd továbbítani és milyen módon szeretnénk tárolni.

- *A Step-ek részletezése.* Ismét át kell tekinteni a már lerajzolt ábrát abból a szempontból, hogy minden step megfelelő típusú-e és kifejező névvel rendelkezik.

- *A munkafolyamat lefolyása (transition és flow elemzés).* Minden átmenet (*transition*) tartalmazzon egy beszédes címkét. Le kell ellenőrizni, hogy minden ágon létezik-e egy alapértelmezett út (*default path*). Amennyiben a folyamatban van hurok, úgy meg kell nézni, hogy ezen iterációk rendesen be tudnak-e fejeződni (például a maximális iteráció szám beállításával).

- *Konnektorok.* A konnektorokat igénylő step-ekhez konfiguráljuk be azokat. Mindig érdemes meggondolni, hogy a konnektorokat egy humán taszkhoz kötjük-e vagy egy külön step-et veszünk fel neki.

- *Actorok.* Definiáljuk az egyes humán step-ekhez az *Actor*-okat, gondoljuk át, hogy szükség lesz-e *Actor Filtert* használni.

- *Monitorozás.* Meg kell határoznunk a mérésre és gyűjtésre szánt fő KPI mutatókat, amik lehetővé teszik a BAM[2] használatát.

- *Kivételkezelés.* Kell egy terv arra vonatkozóan is, hogy egy hiba vagy váratlan esemény bekövetkeztekor azt milyen módon akarjuk kezelni. Meg kell fontolni, hogy ilyenkor állítsuk-e le a process-t vagy azt egy másik ágon folytassuk ilyenkor. Sokszor egy esemény (*event*) által kezdeményezett exception handling subprocess használata is megfontolandó.

- *A folyamat dokumentálása.* Le kell ellenőrizni, hogy a BPMN terv tartalmazza-e az összes releváns dokumentációs célú bejegyzést, ami a későbbi dokumentáció generáláshoz is hasznos lesz.

### A process alkalmazás elkészítése

A process részletes elkészítése után ez a következő lépés. A feladat az, hogy a workflow humán taszkjaihoz elkészítsük a felhasználói felületeket. A Bonita Stúdióban (lásd a 3. fejezetet) ezt gyorsan el lehet végezni, de szükség esetén külső alkalmazásokat is használhatunk.

[2]BAM=Business Activity Monitoring

### Tesztelés és telepítés

Ezek a lépések mindegyik, így a BPM fejlesztések során is kötelező elemek. A Bonita Stúdió beépített eszközöket ad a unit tesztek elvégzéséhez.

### Az Organization specifikálása

Ezen lépés részleteit a 2. fejezet írja le. Itt az a fő feladat, hogy a process lépéseit valódi emberekhez tudjuk rendelni. Ehhez egy segédeszköz a szervezet kialakítása.

### Monitorozás és folyamatos tökéletesítés

Az üzemeltetés során rendszeresen érdemes elemezni a folyamatba épített KPI mutatókat, ezek segíthetnek a process állandó tökéletesítésében, illetve annak megállapításában, hogy az mennyire érte el a tervezéskori célját.

## A TASK iterációja

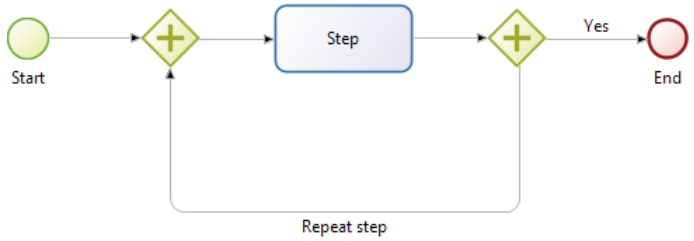
Az iteráció a modell komplexitását csökkenti, ugyanis egy gyorsított eszközt ad arra, hogy tipikus esetekben a STEP milyen módon legyen leképezhető több TASK-ra. Az iterációnak 2 fajtája van:

- *loop* (ciklikusan ismételt TASK)
- *multi-instantiation* (több példányban létrehozott TASK)

A létrejött TASK-ok számosságára egy értéket adhatunk meg vagy egy kollekció típussal szabályozhatjuk azt. Egy fontos filozófiai különbség van a STEP és a TASK fogalma között. A szóhasználatban ez a 2 fogalom néha helyettesíti egymást, de valójában csak bizonyos esetben jelentik ugyanazt. A STEP a legáltalánosabb fogalom, azt jelenti, hogy a folyamatnak egy lépése. Ez lehet egy egész alfolyamat is, de ahogy az iteráció is mutatja, 1 darab STEP konfigurálható úgy, hogy sok TASK-ot hozzon létre. Erre mindig gondoljunk, ha úgy érezzük, hogy elvesztünk a process tervezésének részleteiben.

### A TASK ismétlése (Looping)

Amikor egy TASK-ot néhányszor ismételni kell, akkor először a legtermészetesebb gondolat a 1.2. ábrán mutatott konstrukció. A 2. gateway-ből egy kilépési feltétellel szabályozzuk, hogy akarunk-e ismételni (*Repeat step* ág).

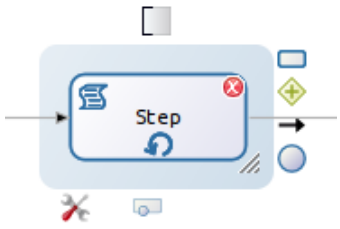



1.2. ábra. Egy task ismétlése hagyományos módon, kapukkal

Erre a gyakori feladatra találta ki a BPMN a looping task fogalmát. Álljunk a *Step* taskra és a *General* → *Iteration* fülön látni fogjuk az *is a standard loop* kiválasztás lehetőségét (1.3. ábra).


1.3. ábra. A looping TASK konfigurációja

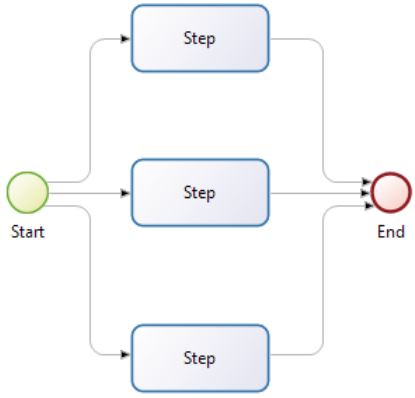
A *Loop while* egy feltételt ad meg, aminek igaz volta esetén egy új ismétlés lesz. Mindez úgy működhet, hogy a hurok szükségességének vizsgálata a TASK-ba lépés előtt (*Test before*) vagy onnan való kilépéskor (*Test after*) történjen. A *Maximum loop* rendelkezik arról, hogy egy TASK ennél több alkalommal ne hajtódjon végre. Azonban ez sem egy merev konstans, hanem lehet egy script kifejezés eredménye is.




1.4. ábra.
A looping task-ot egy 3/4 körös, nyilas ikon jelöli.

### Több TASK példány létrehozása (Multi-instantiation)

A BPMN szemantikája alapján a 1.5. ábra egy olyan szituációt mutat, amikor 3 db TASK párhuzamosan hajtódik végre.



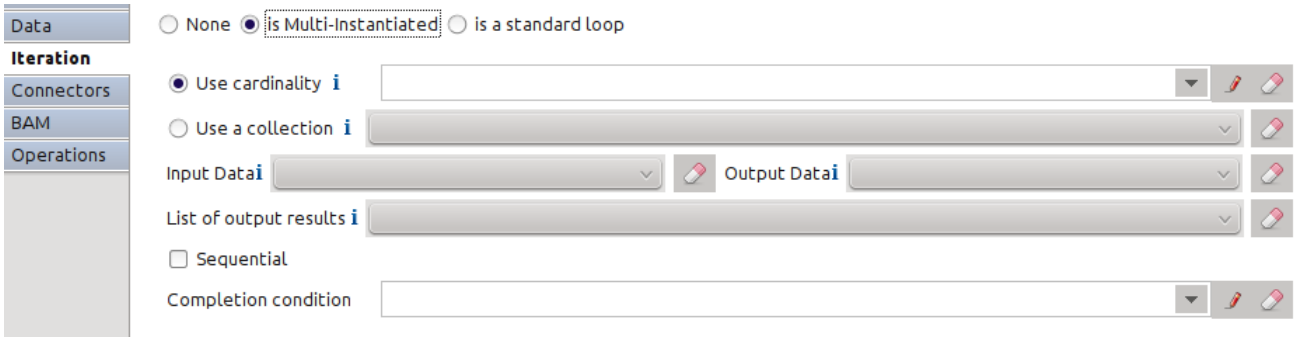

1.5. ábra. Párhuzamosan létező TASK-ok

A BPMN ennél általánosabban fogalmaz és azt teszi lehetővé, hogy egy STEP több példányban (ebben az értelemben a STEP egy-egy példánya a TASK) jöhessen létre. Mindezt kétféleképpen kérhetjük, aminek a 1.5. ábra csak az egyik esete:

- Valóban párhuzamosan jöjjenek létre a dinamikusan generált TASK-ok
- Szekvenciálisan jöjjenek létre a dinamikusan generált TASK-ok

Ezt a 2 esetet Bonitában úgy kell bekonfigurálni, hogy a 1.6. ábra képernyőjén az *is Multi-instantiated* lehetőséget választjuk ki. A továbbiakban azt nézzük meg, hogyan lehet dinamikusan, futás közben meghatározni azt, hogy mennyi TASK példány jöjjön létre. Erre 2 fő lehetőség van:

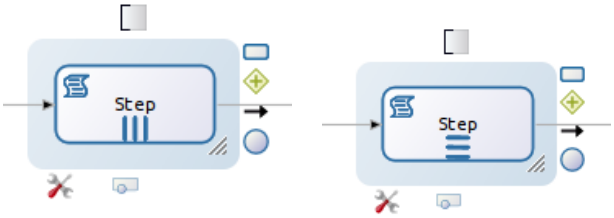
- *Use cardinality*: pontosan annyiszor, amennyiszer a kifejezés számértéke előírja
- *Use a collection*: Egy kollekció adattípus minden elemére jöjjön létre egy külön TASK




1.6. ábra. A Multi-instantiation konfigurációs lehetőségei

Még a folytatás előtt szeretnénk kiemelni a *Sequential* checkbox szerepét, ugyanis ezzel tudjuk megmondani, hogy párhuzamos (jele az 1.7/a. ábrán) vagy soros (jele az 1.7/b. ábrán) végrehajtást (más szavakkal: TASK generálást) kérünk-e.

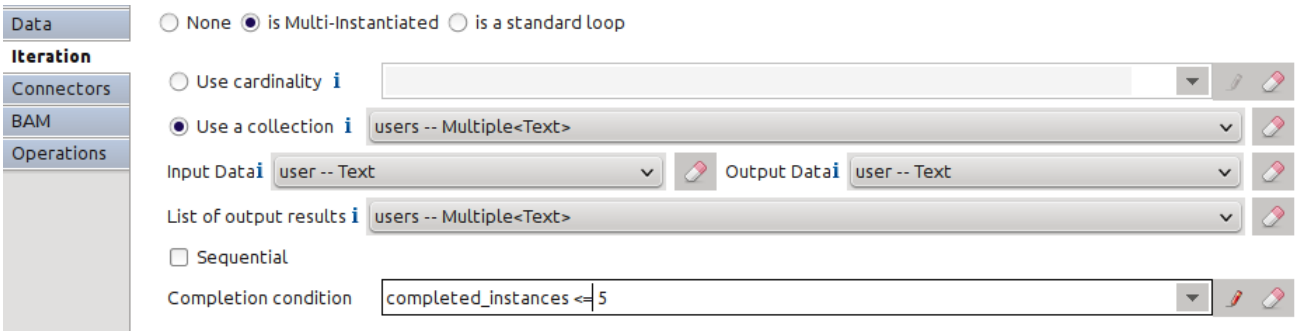
Amikor egy kollekciót adunk meg az automatikus TASK generálás vezérlésére, néhány dolgot át kell gondolnunk (lásd a 1.6. ábrát ismét).




(a) Valódi párhuzamosság (b) Szekvenciálisan hajtja végre a példányokat

1.7. ábra. A Multi-instantiation esetei.

- Mi lesz maga a kollekció (melyik worklfow változó)
- *Input Data*: Az adat a kollekció i. eleméből ide fog bemásolódni a konkrét TASK példány létrehozásakor. Ebben az értelemben ez az adatrész az i. TASK-hoz tartozik csak, azaz az ő munkájához szükséges körülményeket adhatjuk át neki (olyan, mint az input paraméter átadás).
- *Output Data*: Az az adat amit a TASK példányból vissza kell tennünk majd a kollekcióba (olyan, mint az output paraméter).
- *List of output results*: A TASK-ok ebbe a kollekcióba rakják össze az eredményét a munkájuknak




1.8. ábra. Egy bekonfigurált több példányos step

Az 1.8. ábrán egy kollekciót használó step-et látunk konfiguráció után. A TASK rendelkezik egy Text típusú *user* lokális változóval, a pool szintjén pedig létre van hozva egy *users* változó, ami egy Text kollekció (azaz *List<Text>* a Java-ban), mert bepipáltuk az is *multiple* checkbox-ot rá. Ezután a step működését ezekkel a lépésekkel lehet elképzelni:

1. Elérkezik a folyamat a step-hez
2. A workflow motor látja, hogy a *users* kollekció elemei alapján kell létrehozni a TASK-okat
3. Minden kollekció elemből hozzárendel egy-egy értéket a TASK lokális *user* nevű változójához.

4. A TASK kompletté válik, munkája során esetleg változtatta a *user* (lehetett volna más is) változó értékét. A példában a Bonita motor kiveszi, használja, majd visszarakja a megadott *List of output results* kollekció (most *users* változó) megfelelő i. helyére az értéket.

Miért jó mindez? Mert a TASK-ok lefutása után lesz egy olyan közös kollekció, ahol összegyűjtöttük azok részeredményeit, ha belegondolunk eredetileg is ez volt a célunk. Az 1.8. ábrán még látunk egy *Completion condition* megadási lehetőséget is. Itt egy olyan feltételt adhatunk meg, ami azt szabályozza, hogy a több TASK példányos step mikor tekinthető komplettnek. Ennek is sokszor van haszna, amit egy egyszerű példán keresztül szeretnénk érzékeltetni.

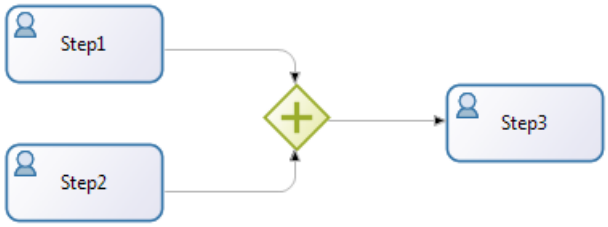
*Példa*: Egy biztosítási társaságnál egy olyan step van a biztosítás megítélési folyamatban, ahol szavazni lehet. Tegyük fel, hogy 10 vezető szokott szavazni, de amennyiben 6 már igent mondott, akkor vége a szavazási lépésnek. Ekkor elindul párhuzamosan 10 TASK példány (átadva mondjuk egy 2 mezős objektumot, mint kollekció elemet: user, szavazat). Ezek fokozatosan kompletté válnak, ezalatt az engine figyeli, hogy az a feltétel teljesült-e már, hogy 6 db IGEN született.

## A BPMN Kapu (Gateway)

A Bonita 6.x a gateway-eket a legújabb BPMN szabványhoz igazította, ezért tekintsük át röviden. A gateway arra szolgál, hogy a munkafolyamat lépéseinek folyamatát tudjuk jól definiált módon vezérelni.

### Parallel Gateway (AND Gateway)

Az 1.9. ábra egy Parallel gateway-t mutat (rombuszban egy + jel).

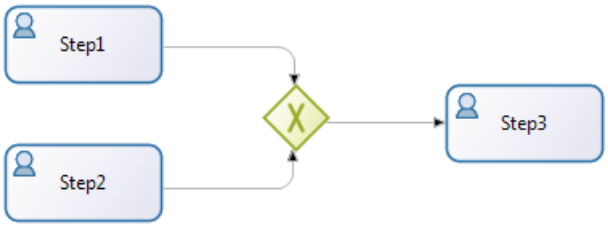



1.9. ábra. Parallel Gateway

Az AND kapu működése megköveteli, hogy a Step1 és Step2 TASK is kompletté váljon, addig a kapu várakoztat, utána továbbléphetünk a Step3-ra. Amennyiben több nyíl megy ki az AND gateway-ből, úgy azok párhuzamosan induló TASK-okat fognak eredményezni. Azt is érdemes észrevenni, hogy a várakoztatás miatt az AND kapu egy szinkronizációs feladatot is el tud látni. Aki ismeri az UML Activity diagram Fork és Join működését, annak ez a kapu ismerős, mert pont azt valósítja meg.

### Exclusive Gateway (XOR Gateway)

Az 1.10. ábra egy Exclusive gateway-t mutat (rombuszban egy X jel).

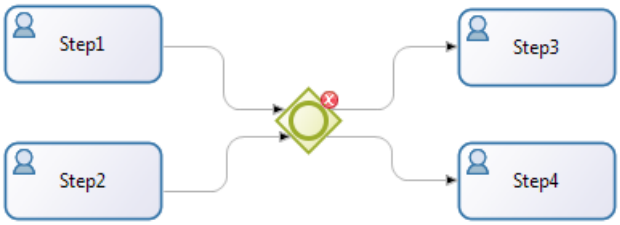



1.10. ábra. Exclusive Gateway

A workflow design-nak biztosítania kell, hogy pontosan 1 db input fusson be, ami azt jelenti, hogy a befutó nyilakon lévő logikai feltételeknek egymást kizárónak kell lennie. Amennyiben több nyíl megy ki a kapuból, úgy ez igaz arra is, mindig csak 1 irányba mehet tovább a munkafolyamat.

### Inclusive Gateway

Az 1.11. ábra egy Inclusive gateway-t mutat (rombuszban egy O jel).




1.11. ábra. Inclusive Gateway

Egy Inclusive gateway bevárja az összes olyan útról a TOKEN-t, amely aktív (az aktívságot a workflow engine figyelemmel kíséri). Kifelé itt is párhuzamosan mehet a vezérlés, de itt őrző feltételeket kell megadni és csak azokba az irányokba megy, amelyek igazak lettek a kiértékelés során. Az egyik ágnak default-nak kell lennie, azaz amennyiben az összes többi hamis, úgy ezen megy tovább a folyamat. Nem szükséges őrző feltétel, ha csak 1 kimenő nyíl van.

## Keresési indexek (search index) létrehozása

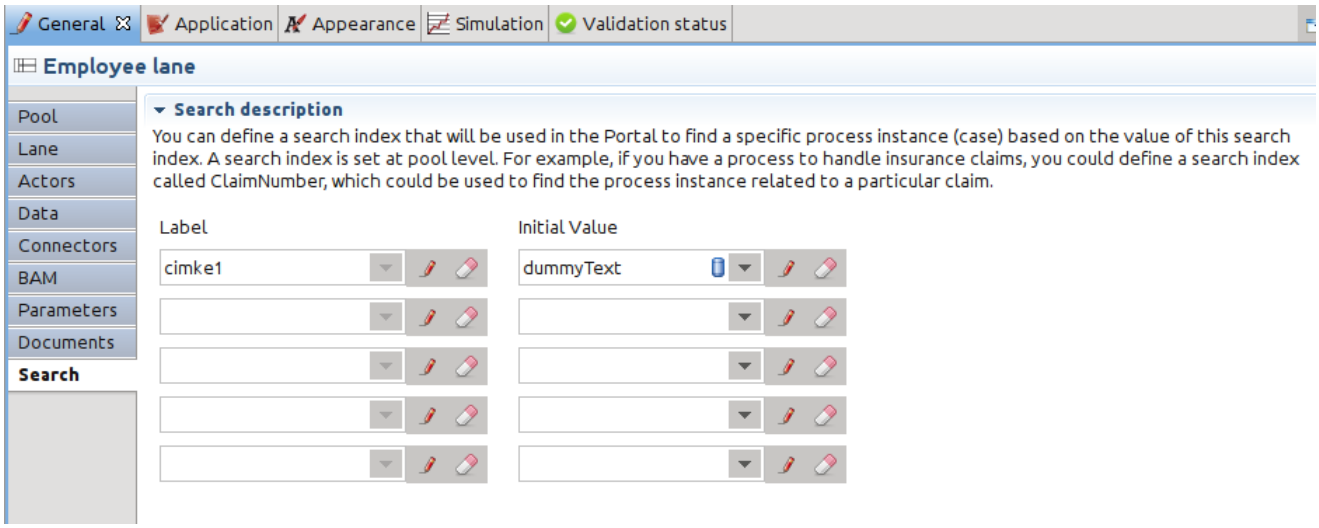
A Bonita minden process esetére legfeljebb 5 keresési index létrehozását teszi lehetővé, aminek az a célja, hogy a Bonita Portálon gyorsan megtaláljuk egy-egy process-re nézve azt, amit szeretnénk.

Egy új search index létrehozása a Bonita Stúdióban a következő lépésekből áll:

1. A process diagramon álljunk rá a megfelelő folyamatra (pool-ra) és válasszuk ki a Details panel *General* → *Search* fület (1.12. ábra).

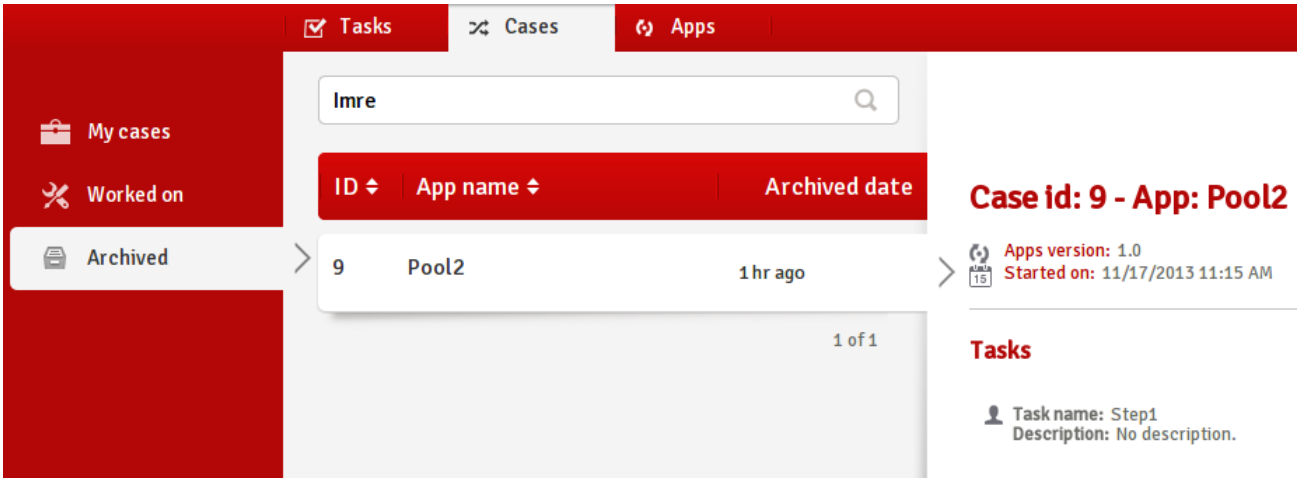
2. A *Label* egy index név, ami ezt a keresési index szempontot azonosítja a Bonita motorban. Nincs megjelenítve a Portál felületen.

3. Az *Initial Value* adja meg az index értékét




1.12. ábra. Keresési index megadása a Bonita Stúdióban

A keresési értékek akkor számítódnak ki, amikor a workflow instance létrejön. Természetesen nem mindig ismerjük a keresési kifejezés véglegesen használni kívánt értékét, ezért egy Task Operations segítségével annak értékét a workflow futása során is időszerűsíthetjük. Ekkor egy olyan actions készíthetünk, aminek a targetje az a címke, amit a keresési feltétel megadásánál adtunk meg, azaz a lehetséges target értékek között a search filter label-ek is megjelennek. A használatot az 1.13. ábra mutatja.




1.13. ábra. A keresés használata a Bonita Portálon

# 2. Task és Actor

A most következő fejezet a humán TASK és ACTOR fogalmát próbálja elmélyíteni, bemutatva azok kapcsolatát. A Bonita 6. hozott néhány fontos újdonságot ebben a témában, így fontosnak gondoljuk az itt kínálkozó lehetőségek részletes bemutatását. Eközben megismerkedünk az *Organization* eszközzel is.

A Bonita 5.x verziókban az *Actor Selector* volt az az eszköz, ahogy egy TASK esetén megadhattuk, létrehozhattuk azon a USER halmazt, akik azt kezelhették, láthatták. Ez egy olyan speciális konnektor volt, ami mindig egy *List<String>* értéket adott vissza, más szavakkal ez a user nevek listája volt. Emiatt ennek a konnektornak csak az input paramétereit kellett megadnunk és azt, hogy ezek hogyan rendelődnek (mapping) össze a workflow belső változóival. Az Actor Selector így egy ilyen függvény volt (*Actor Selector Function*):

```
ASF(par1, par2, ...) —> user nevek String sorozata
```

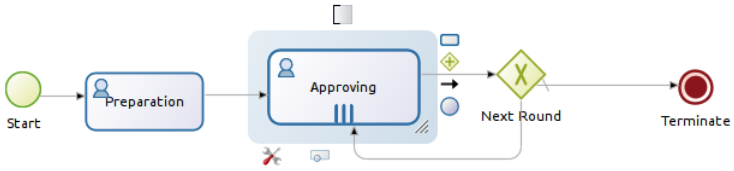
Amikor egy *ASF* futott egy következő TASK inicializálása részeként és előállította a USER halmazt, akkor ehhez tetszőlegesen sok input paramétert kaphatott, lényegében ez biztosította a dinamizmust és rugalmasságot. Mindig az adott szituációra illesztett USER-ek listája tudott legenerálódni. Sok előre elkészített ASF függvény volt, általában ezek sok mindenre elegek is voltak. Ki kellett választani egyet *Pool*, *Lane* vagy *Task* szinten, a hierarchiában lejjebb lévő örökölte a szülő Actor Selector-t, amennyiben nem volt neki sajátja. Könnyen lehetett saját ASF függvényt (és annak a varázslóját is) készíteni, csak implementálni kellett a megfelelő Java interface-t. Minderről részletesebben az *Informatikai Navigátor 5.* számából olvashatunk. A Bonita 6.x-ben az Actor Selector mechanizmust az *Actor Mapping* megoldás váltotta fel. Ez azt jelenti, hogy minden egyes actor egy

- *GROUP*-hoz (csoporthoz),
- *ROLE*-hoz (szerepkörhöz) és
- *MEMBERSHIP*-hez (tagsághoz) vagy *USER*-hez egy *Organization*-on belül

lehet rendelve (mapping). Ezt a megoldást az Actor Filter segítségével lehet annyira finomítani, ahogy az az 5.x verziók Actor Selectorában is volt.

## A Step és a Task

A munkafolyamat STEP-ekből áll, egy lépés az, ami reprezentálja a folyamatban az előrehaladást. A 2.1. ábra 2 step-et tartalmaz: *Preparation* és *Approving*. Ezek a lépések több TASK-ot is jelenthetnek, azaz a STEP és TASK között 1:N reláció van. Az ábra alapján az *Approving* step-ből ilyen okok miatt jöhet létre több taszk:




2.1. ábra. Egy példa process 2 TASK-kal

- A párhuzamosságot jelentő 3 vonal önmaga azt jelenti, hogy a kiértékelt feltételtől függően a step-ből több taszk jöhet létre. Ez más szavakkal azt jelenti, hogy ez a lépés több, párhuzamos elfogadási feladat megvalósulása után lesz csak komplett.

- A kilépés utáni XOR kapu kiértékelődhet úgy, hogy a *Next Round* ágon haladva ismét új *Approving* taszkok keletkeznek meg.

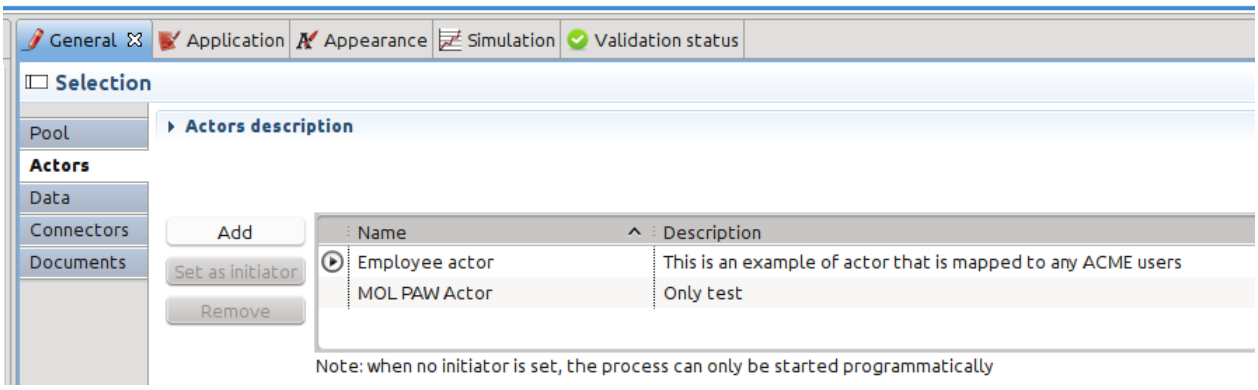
## Az Actor fogalma és használata

Egy ACTOR a folyamat tervezésénél egy névvel ellátott *placeholder* (helyőrző), ami a tervezés során képviseli azt a fogalmat, hogy „Ők fogják végrehajtani". Olyasmi, mint egy változó, ami még nem kapott konkrét értékhalmazt, de azt tudjuk, hogy majd ezt a hiányt fel kell oldanunk. A tervezés tehát 2 „fokozatból" áll:

1. A folyamat designer (elemző, üzleti képviselő vagy aki még ilyet csinál) első lépésben csak ACTOR neveket vesz fel és ezeket rendeli hozzá a TASK-okhoz. Ezt néha implicit módon teszi, mert a POOL és LANE is rendelkezhet ACTOR hozzárendeléssel, így amennyiben egy TASK nem rendelkezik ilyennel, úgy örökli azt a fölötte lévő úszósávtól, esetleg a medencétől.

2. A 2. fokozat pedig a logikai ACTOR nevek egy tényleges USER halmazra való leképzése. Amikor egy process-t telepítünk és futtatni akarunk, akkor az ACTOR nevek leképződnek (ACTOR Mapping) egy szervezet valódi felhasználóira.

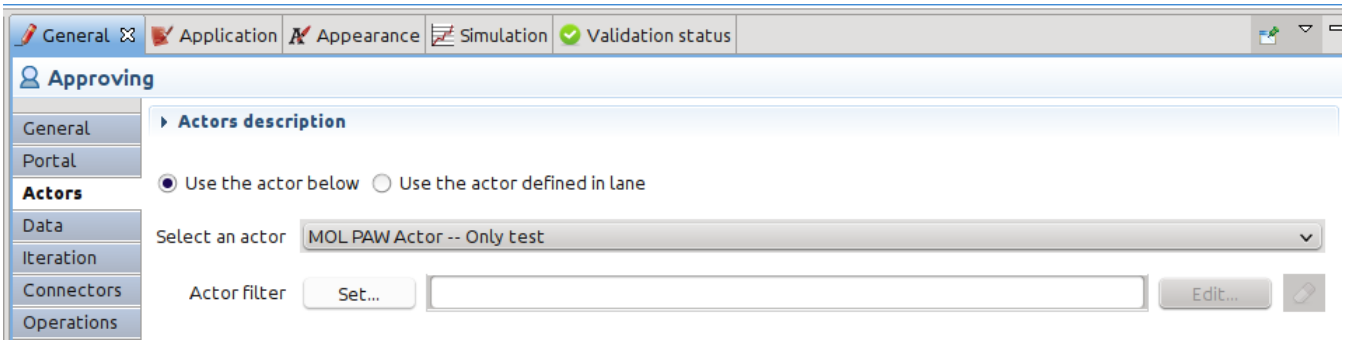
Egy új ACTOR-t létrehozni úgy lehet, hogy a POOL szintjére állunk (oda kattintunk vagy a *Tree View* ablakban kiválasztjuk) és a *General* → *Actors* fület választjuk ki. A számunkra érdekes lehetőségeket az 2.2. ábra mutatja. Felvehetünk és törölhetünk ACTOR neveket. A *Set as initiator* nagyon lényeges, mert amennyiben egy ACTOR-ra ez nincs engedélyezve, úgy azzal nem tudunk egy formon keresztül (például Bonita Portálból) új process instance-t (case-t) létrehozni.




2.2. ábra. Új Actor létrehozása

Amikor az ACTOR-t használni akarjuk, akkor azt a LANE vagy TASK (STEP) szintjén az *Actors* fülön tehetjük meg, egy dropdown vezérlő használatával, ami persze már tartalmazni fogja

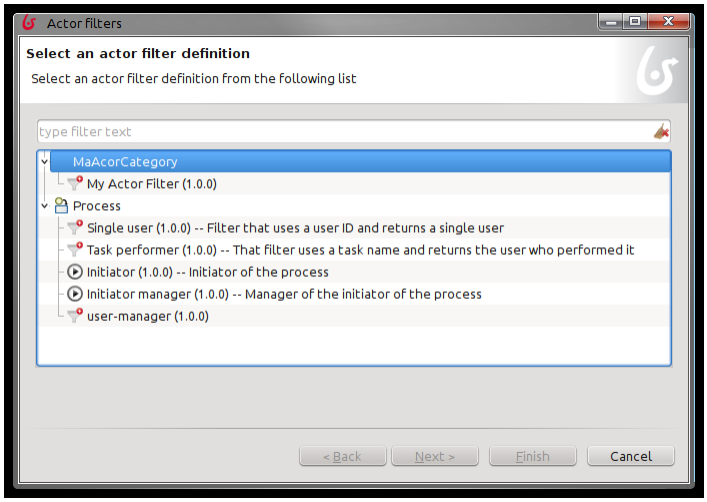
az eddig létrehozott összes ACTOR nevet. A 2.3. ábra azt mutatja meg, hogy egy ACTOR-t (*MOL PAW Actor*) hogyan rendelünk hozzá egy *Approving* nevű TASK-hoz.




2.3. ábra. Az Approving TASK és a MOL PAW Actor összerendelése

Amennyiben nem akarunk saját ACTOR-t rendelni ehhez a TASK-hoz, úgy használhatjuk a LANE ACTOR-át is, ekkor a 2.3. ábra *Use the actor defined in lane* rádiógombját kell választanunk.

A 2.3. ábrán látható esetben a *Use the actor below* gomb van kiválasztva, azaz ehhez a TASK-hoz egy saját, a már említett ACTOR-t adtuk meg. A *Set...* megnyomására a 2.4. ábra ablaka jön fel. Ezzel elérkeztünk oda, hogy megértsük mi is az az *Actor Filter*, hiszen itt majd egy olyat fogunk választani. A *MyActorFilter* saját készítésű teszt filter, az nem része a gyári filtereknek, de majd még erre is visszatérünk, amikor bemutatjuk a saját készítésű filter létrehozását. A *Process* kategória alatt láthatóak a gyári Bonita ACTOR filterek. Nézzük meg őket röviden!



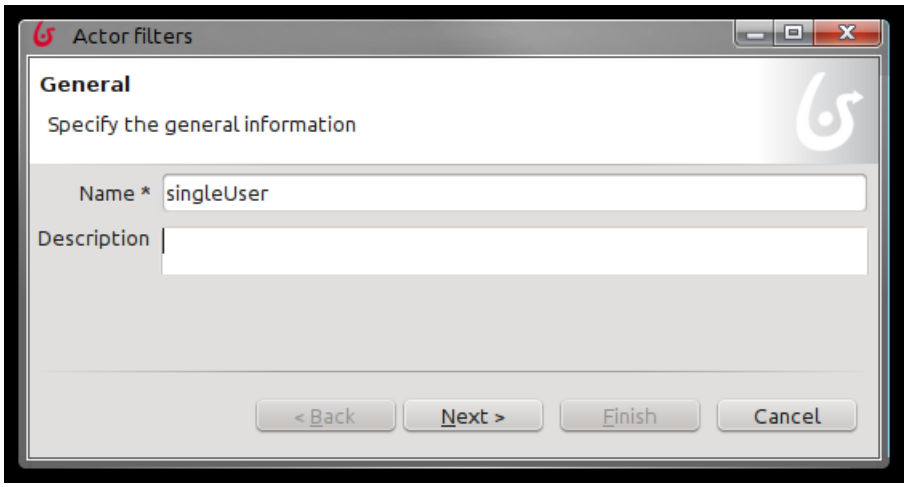

2.4. ábra. A *Set...* hatására feljövő dialógus ablak

**Single user**

Ezzel a filterrel egy megadott user-t rendelhetünk a taszkhoz. Nézzük a használatát! Az első dialógus ablakban adunk egy nevet a konkrét filter példánynak (2.5. ábra).

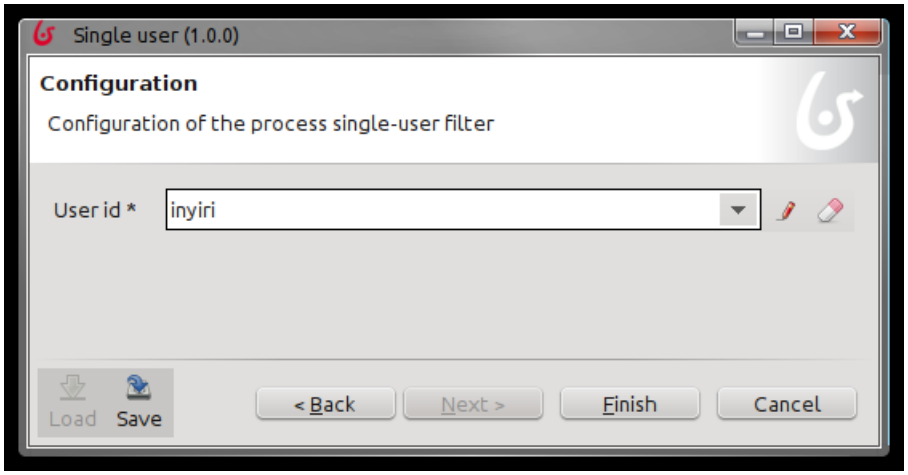
*Konkrét Filter példány:* Mielőtt továbblépünk gondolkodjunk el azon, hogy ez mi? A filter egy Java class, ami egy USER halmaz előállítására képes módszert képes megadni. A példány ennek egy objektuma, ami már a használat körülményeit is ismeri, az input paraméterek segítségével konstruálja meg a szűrést (azaz a USER halmaz generálását) végző objektumot.

□



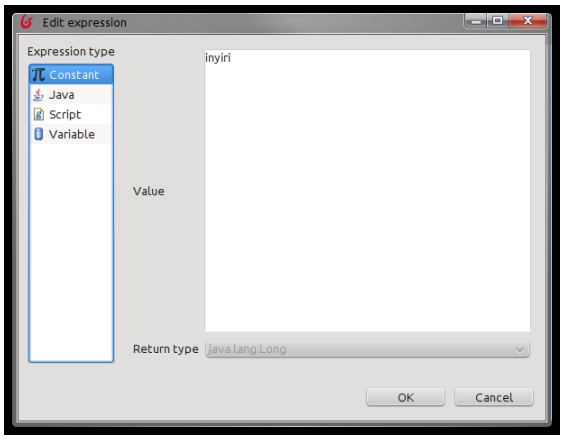

2.5. ábra. Single user Actor Filter specifikálása - 1. lépés

Ezután adjuk meg a konkrét user-t (2.6. ábra), miután a 2.3. ábra *Actor filter* mezőjének ez lesz majd az értéke: *singleUser – Single user*. Mindez azt jelenti, hogy esetünkben az *Approving* TASK-hoz érve a Bonita generál egy single user típusú filter objektumot, ami paraméterül az *inyiri* értéket kapja. Az objektum filterezést végző metódusa lefut és a működési modelljéből következően ezt a USER halmazt állítja elő a taszkhoz: $\{inyiri\,ID\}$.

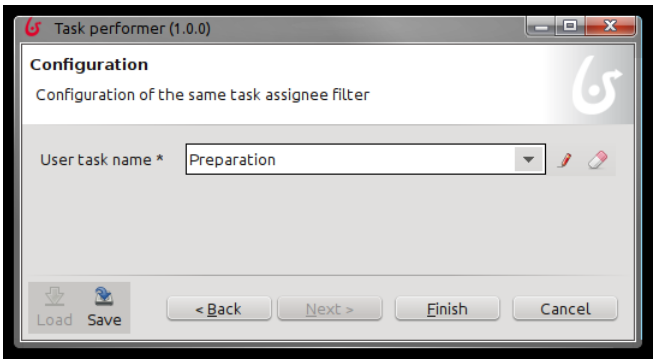



2.6. ábra. Single user Actor Filter specifikálása - 2. lépés

Az *Edit...* gombbal bármikor változtathatjuk ezt a konfigurációt, a törlő ikonnal ki is törölhetjük. Dinamikus megoldásra ad lehetőséget az, ha az 2.6. ábra ceruza ikonját nyomjuk meg, amire a 2.7. ábra ablaka jön be. Látható, hogy az elsőnek látott konstans username (itt: inyiri) helyett sokféle más módon is adhatunk meg értéket. Például a *Script* azt jelenti, hogy a workflow belső változói felett futó *Groovy* script számít ki egy String-et és annak az értéke lehet a felhasználó.




2.7. ábra. Actor kiválasztása a kifejezés szerkesztővel




2.8. ábra. Task Performer konfiguráció az Approving step-re

### Task Performer

Ez egy olyan filter, ami azt a user-t adja vissza, aki egy megadott nevű TASK-ot csinált meg az aktuális process példányban. Pillantsunk rá ismét a 2.1. ábrára, ami egy 2 TASK-os process-t mutatott. A 2.8. ábra mutatja, hogy az *Approving* TASK esetén kiválasztottuk azt a user-t, aki a *Preparation* TASK-ot is elvégezte. Ebben az esetben a 2.3. ábra *Actor filter* mezőjébe ez az érték fog kerülni: *taskPerformer – Task performer*. A 2.8. ábra ceruza ikonja itt is dinamizmusra ad lehetőséget, de ekkor ez most a TASK nevének dinamikus előállítását jelenti.

### Initiator

Ez egy nagyon egyszerű eset. Azt adhatjuk meg vele, hogy az legyen a kiválasztott konkrét user, aki a munkafolyamatot kezdeményezte, azaz például az indító formot kitöltötte.

### Initiator manager

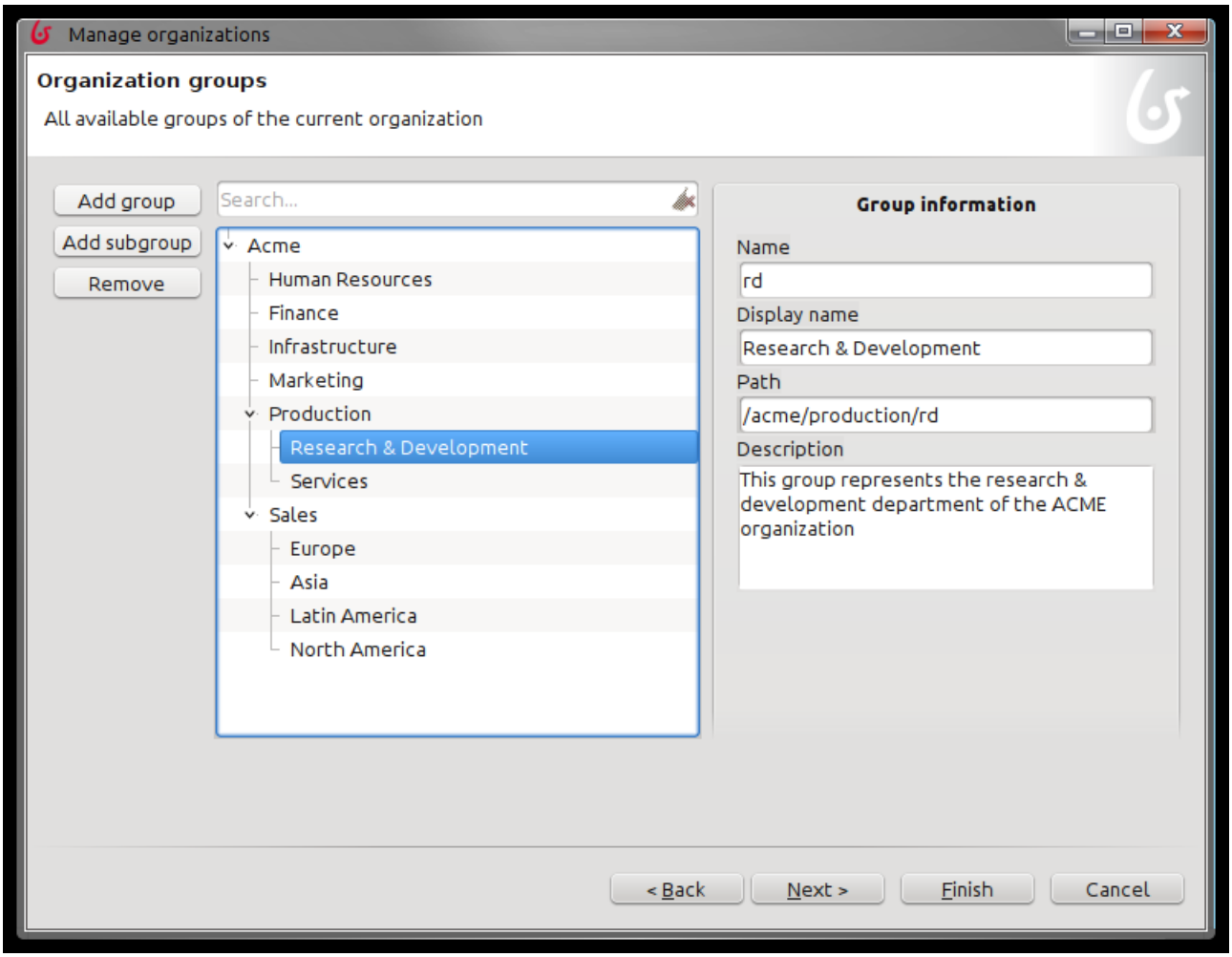
Ez a 2.4. ábrán látható következő filter, ami nem a munkafolyamat indítóját, hanem annak a főnökét adja vissza. Ennek használata sem igényel semmilyen input paramétert, így a varázslója is nagyon egyszerű.

### User manager

A megadott user menedzserét rendeli hozzá a TASK-hoz.

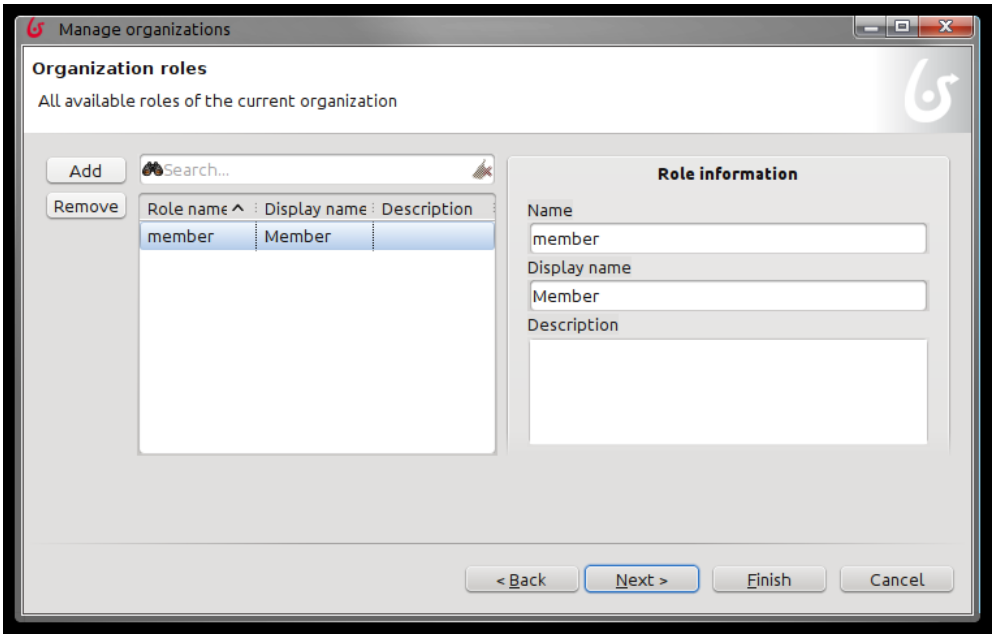
## Actor Mapping - A USER halmaz hozzárendelése a TASK-hoz

Az Actor Filter mellett a Bonita 6.0 bevezette az *Actor Mapping* mechanizmust, ami deklaratív módon igyekszik az ACTOR→USER halmaz leképzést megvalósítani.

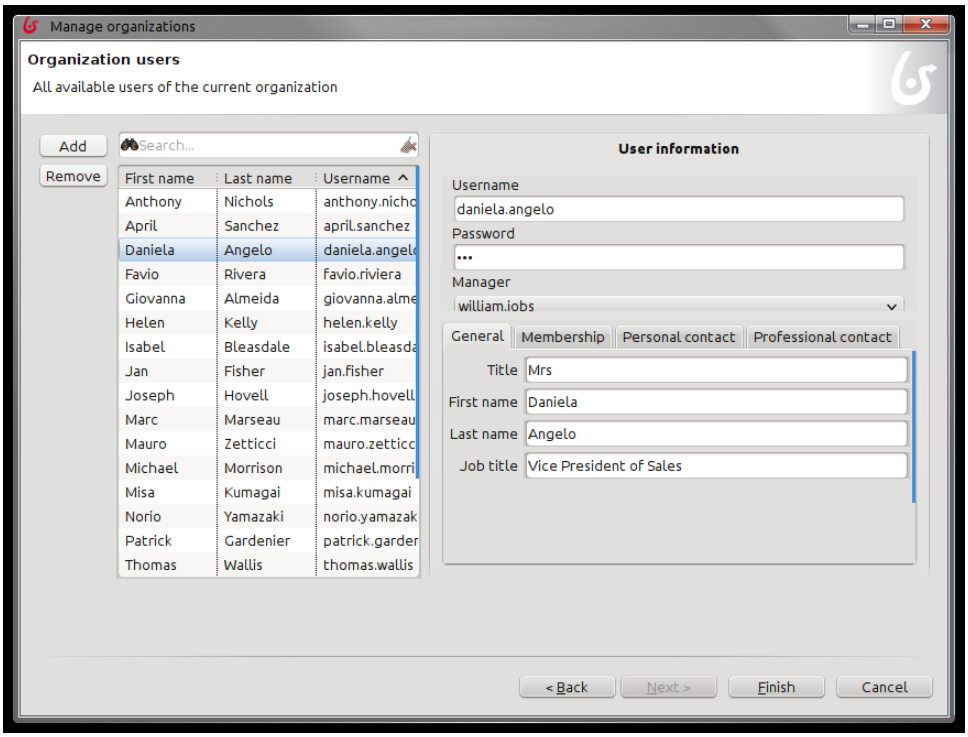



2.9. ábra. Organization - 1: GROUP

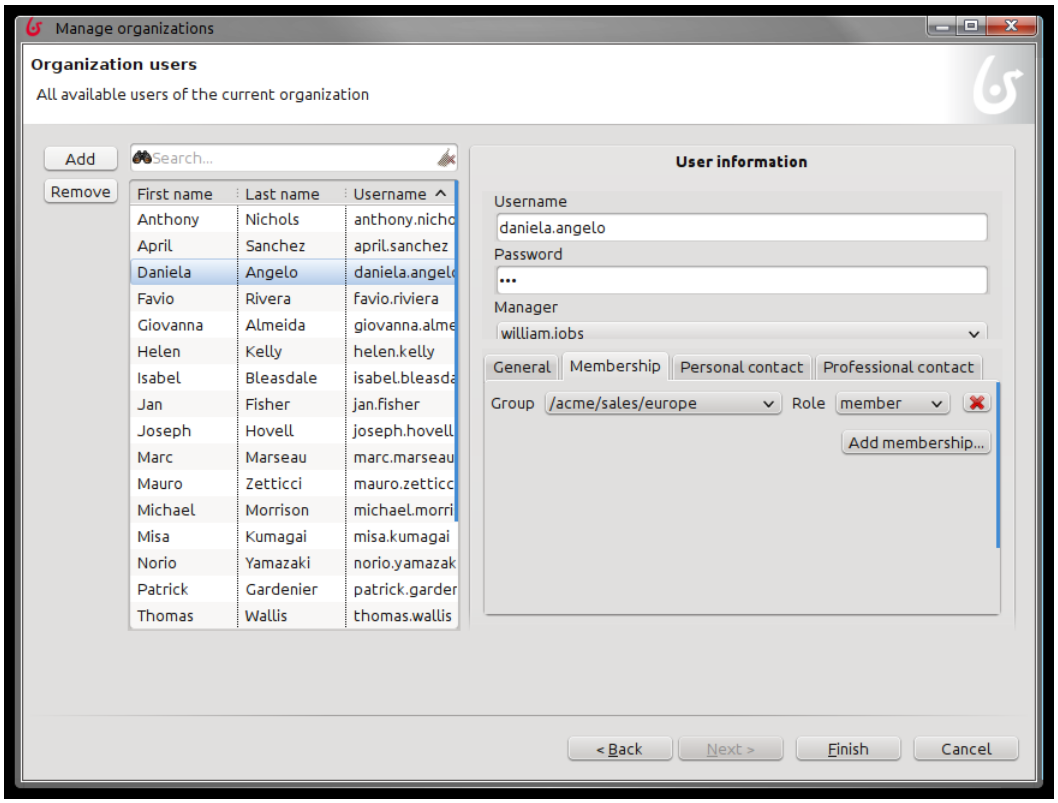
Amennyiben ilyen megoldást is használunk, úgy a process-t ACTOR-ait mappingelni kell a USER-ekhez, amik az ORGANISATION-ban találhatóak. Milyen egy *Organization*? Ez is egy új fogalom a Bonita 6.x verzióban. A Bonita Stúdió tartalmaz egy példa szervezetet (a neve: ACME), de mi is létrehozhatunk tetszőleges számban újakat. Az *Organization* főmenűponton keresztül juthatunk el a 2.9. ábra képernyőjéhez, ami most az ACME szervezetet, annak a csoportjait mutatja. A fejlesztés során egyszerre csak egy organization lehet aktív, ami a valóságban azt jelenti, hogy a stúdió beépített Bonita engine-jébe a *Publish* menüpont segítségével kell kiválasztani és telepíteni az éppen használni kívántat. A következő ablak (2.10. ábra) a *Next* gombra jön fel és a ROLE-okat tudjuk benne definiálni. Ez azt jelenti, hogy egy szervezeti egységben (azaz GROUP-ban) többféle ROLE-ban dolgozó ember lehet, amely szervezeti szerepköröket itt adhatjuk meg. A 2.11. ábrán a USER-ek megadását látjuk, akik a szervezetben dolgoznak. A *General* fül az általános user adatokat tartalmazza. A 2.12. ábra a tagságot mutatja, azaz *Daniela* a megadott csoport, megadott szerepkörében van. A *Personal contact* és *Professional contact* fülek néhány leíró adatot tartalmaznak a felhasználóról (e-mail, telefon, ...).




2.10. ábra.
Organization - 2: ROLE. A ROLE egy névvel és display névvel rendelkezik. Ez egy String, ami egy elképzelt szerepkör neve. A member ROLE a tagság szerepkört jelenti. Ehhez hasonló szerepköröket érdemes kitalálni, mert hatékonnyá teszi a későbbi actor összerendelést. Ilyen lehet például az approval, ami azt jelenti, hogy ebben a szerepkörben lévő ember elfogadhat.




2.11. ábra.
Organization - 3: USER - General. A szervezeten belüli felhasználókat itt felveheti vagy módosíthatja a fejlesztő. A cél az, hogy az egyes felhasználók legfontosabb adatait rögzíteni tudjuk. Éles környezetben ez az adatbázis a cég különféle user adatbázisaiból (AD, SAP, ...) szinkronizálódik ide. A Personal és Professional személyes információk között például megtaláljuk, hogy ki az adott felhasználó menedzsere.




2.12. ábra. Organization - 4: USER - Membership

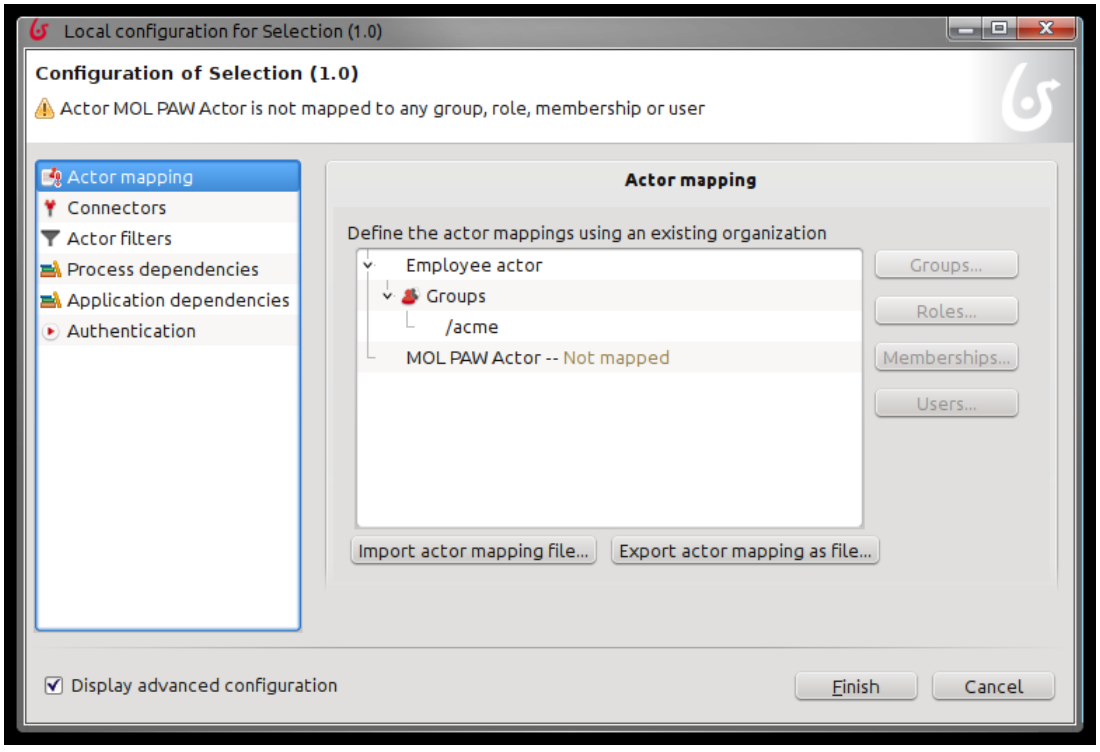
A 2.12. ábra még mindig a felhasználókat mutatja, ahol mindenkire be tudjuk állítani, hogy milyen tagságai (*Membership*) vannak. Az *Organization*-okat exportálni lehet XML fájlba, ahogy azt az ACME szervezet esetén a 2-1. Programlista mutatja. Ezeket az XML-eket a szerverre lehet telepíteni. Lehet őket importálni is a Bonita Stúdióba.

2-1. Programlista: Az ACME Organization XML reprezentációja - részlet

```
<?xml version="1.0" encoding="UTF-8"?>
<organization:Organization xmlns:organization="http://www.bonitasoft.org/ns/organization/6.0.0-beta-016">
  <users>
    <user userName="william.jobs">
      <firstName>William</firstName>
      <lastName>Jobs</lastName>
      <title>Mr</title>
      <jobTitle>Chief Executive Officer</jobTitle>
      <professionalData>
        <email>william.jobs@acme.com</email>
        <phoneNumber>484-302-5516</phoneNumber>
        <faxNumber>484-302-0516</faxNumber>
        <building>70</building>
        <address>Renwick Drive</address>
        <zipCode>19108</zipCode>
        <city>Philadelphia</city>
        <state>PA</state>
        <country>United States</country>
      </professionalData>
```

```
20        <metaDatas>
21          <metaData name="Skype_ID"/>
22          <metaData name="Twitter"/>
23          <metaData name="Facebook"/>
24        </metaDatas>
25        <password encrypted="false">bpm</password>
26      </user>
27      <user userName="april.sanchez">
28        <firstName>April</firstName>
29        <lastName>Sanchez</lastName>
30        <title>Mrs</title>
31        <jobTitle>Compensation specialist</jobTitle>
32        <manager>helen.kelly</manager>
33        <professionalData>
34          <email>april.sanchez@acme.com</email>
35          <phoneNumber>484-302-5965</phoneNumber>
36          <faxNumber>484-302-0965</faxNumber>
37          <building>70</building>
38          <address>Renwick Drive</address>
39          <zipCode>19108</zipCode>
40          <city>Philadelphia</city>
41          <state>PA</state>
42          <country>United States</country>
43        </professionalData>
44        <metaDatas>
45          <metaData name="Skype_ID"/>
46          <metaData name="Twitter"/>
47          <metaData name="Facebook"/>
48        </metaDatas>
49  ...
50       <password encrypted="false">bpm</password>
51      </user>
52    </users>
53    <roles>
54      <role name="member">
55        <displayName>Member</displayName>
56        <description></description>
57      </role>
58    </roles>
59    <groups>
60      <group name="acme">
61        <displayName>Acme</displayName>
62        <description>This group represents the acme department of the ACME organization</➥
              description>
63      </group>
64      <group name="hr" parentPath="/acme">
65        <displayName>Human Resources</displayName>
66        <description>This group represents the human resources department of the ACME organization➥
              </description>
67      </group>
68      <group name="finance" parentPath="/acme">
69        <displayName>Finance</displayName>
70        <description>This group represents the finance department of the ACME organization</➥
              description>
71      </group>
72      <group name="it" parentPath="/acme">
73        <displayName>Infrastructure</displayName>
74        <description>This group represents the infrastructure department of the ACME organization➥
              </description>
75      </group>
76  ...
77       <description>This group represents the services department of the ACME organization</➥
            description>
78      </group>
79    </groups>
```

```
80    <memberships>
81      <membership>
82        <userName>william.jobs</userName>
83        <roleName>member</roleName>
84        <groupName>acme</groupName>
85      </membership>
86      <membership>
87        <userName>april.sanchez</userName>
88        <roleName>member</roleName>
89        <groupName>hr</groupName>
90        <groupParentPath>/acme</groupParentPath>
91      </membership>
92      <membership>
93        <userName>helen.kelly</userName>
94        <roleName>member</roleName>
95        <groupName>hr</groupName>
96        <groupParentPath>/acme</groupParentPath>
97      </membership>
98      <membership>
99        <userName>walter.bates</userName>
100       <roleName>member</roleName>
101       <groupName>hr</groupName>
102       <groupParentPath>/acme</groupParentPath>
103     </membership>
104     <membership>
105 ...
106     </membership>
107   </memberships>
108 </organization:Organization>
```

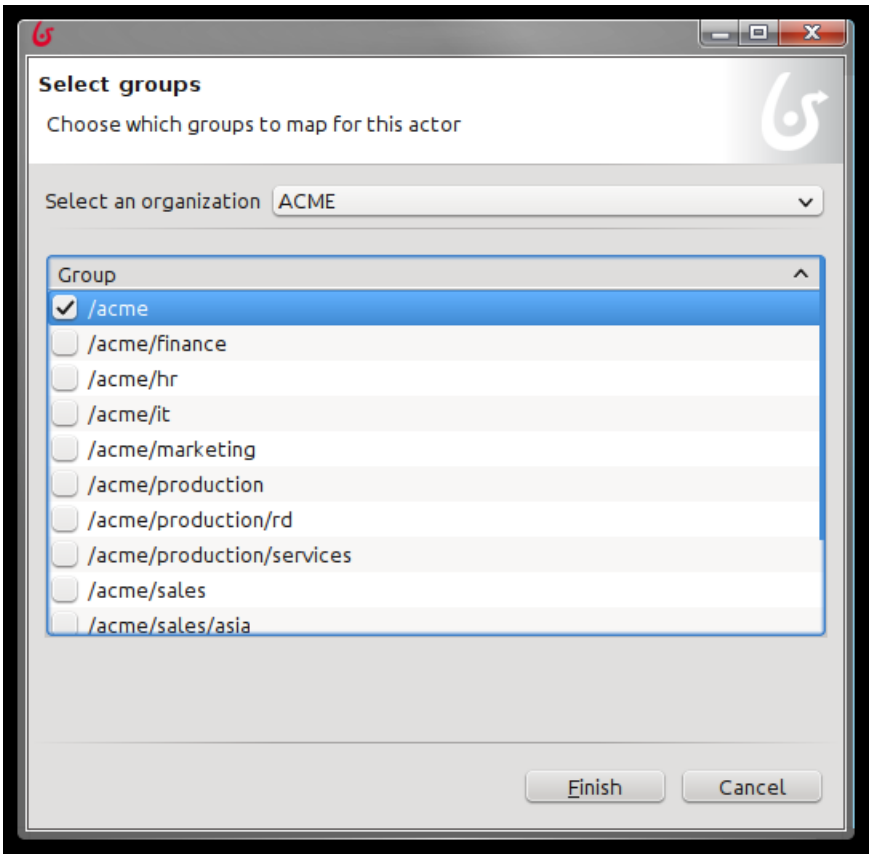



2.13. ábra. Actor Mapping ablak

Ennyi előzetes után itt az ideje az *Actor Mapping* bemutatásának. Nyissuk meg a kívánt process diagramot, amelyiken dolgozni szeretnénk, majd nyomjuk meg a *Cool Bar Configure* gombot! A megjelenő (2.13. ábra) dialógus ablakban az első konfigurálható sor éppen az *Actor Mapping*. Látható, hogy jelenleg 2 ACTOR-unk van:

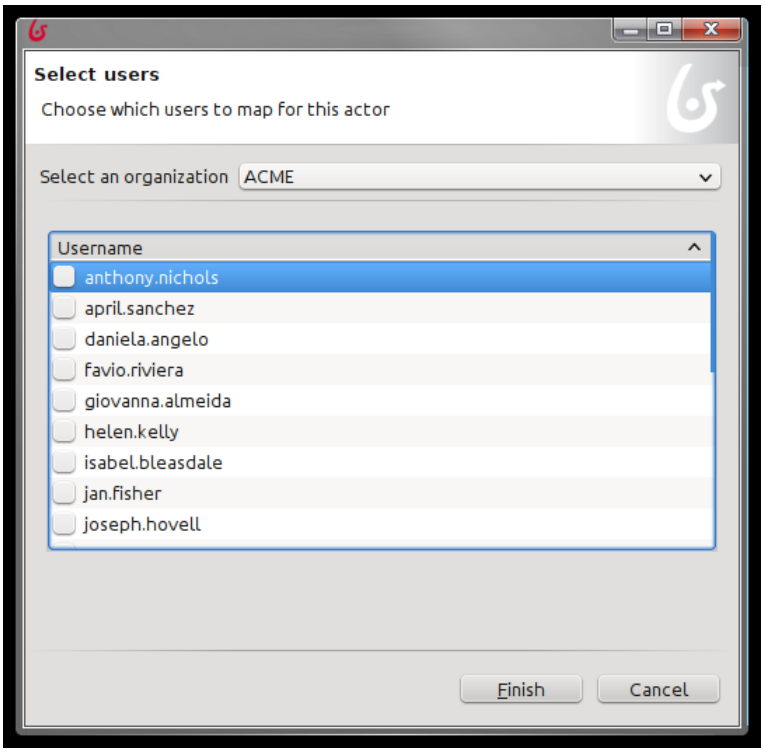
- Employee actor és
- MOL PAW Actor

Az *Employee actor* mappinget nézzük át! A példában az */acme* csoportba tartozást pipáltuk ki, ez azt jelentik, hogy az *Employee actor* feloldásakor keletkező USER halmaz ezzel a csoporttal fog megegyezni, ha nem mondunk még mást is. A *Groups...* gomb megnyomása után (akkor lesz megnyomható, ha a *Groups* ágra kattintunk) a 2.14. ábra ablaka jön be, azaz látható, hogy több csoport uniója is jelentheti a generált USER halmazt.




2.14. ábra. Actor Mapping ablak - Groups

Ha a *Roles...* gombot nyomjuk meg, akkor ezzel szűkítést adunk meg arra nézve, hogy az ACTOR ezekre a szerepkörökre korlátozott. Amennyiben szervezeti (vagy csoporton belüli) szerepkört akarunk adni, akkor a *Membership...* gomb megnyomása után egy (GROUP, ROLE) párt adhatunk meg. A tagság nem egyszerűen egy csoportba vagy szerepkörbe tartozást jelent, hanem azt, hogy az adott csoportban van olyan szerepkörben.




2.15. ábra. Mely userek tartoznak ebbe az ACTOR-ba

A 2.15. ábra ablaka a 2.13. ábrán lévő *Users...* gombra bukkan fel. Azt adhatjuk meg, hogy a konfigurálás alatt lévő ACTOR (most éppen: *Employee actor*) milyen további, ily módon konkrétan megadott USER-ekhez van MAP-pelve. Az actor mapping-et XML fájlba exportálhatjuk (onnan importálhatjuk is a Bonita Stúdióba), erre a 2.16. ábra mutat példát.

```
<?xml version="1.0" encoding="UTF-8"?>
<actormapping:actorMappings xmlns:actormapping="http:/
  <actorMapping name="Employee actor">
    <users/>
    <groups>
      <group>/acme</group>
    </groups>
    <roles/>
    <memberships/>
  </actorMapping>
  <actorMapping name="MOL PAW Actor">
    <users/>
    <groups/>
    <roles/>
    <memberships/>
  </actorMapping>
</actormapping:actorMappings>
```

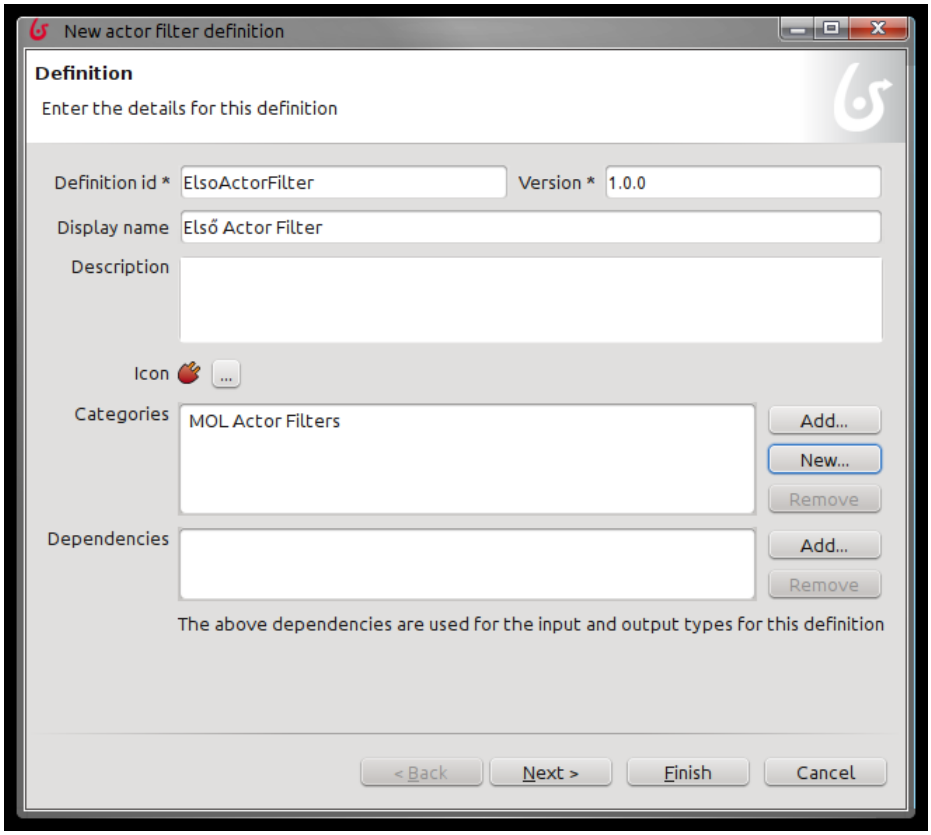
2.16. ábra. Az Actor mapping XML exportja

## Saját készítésű Actor Filter

A 2.4. ábrán láttuk és utána röviden ismertettük is a gyári filtereket. Sok esetben ezek mellett saját, az egyéni igényeinket is kielégítő filter szükséges, aminek készítését és használatát mutatjuk be ebben a pontban. Egy új Java *Actor Filter* írása 2 részből áll:

1. elkészíteni vagy meghatározni a filter definícióját (*Development* → *Actor filters* menüpont)
2. implementálni a kiválasztott filter definíciót (*Development* → *Actor filters* menüpont)

Amikor egy új filter definíciót készítünk, akkor meg kell adni az input paramétereket is, hiszen egy ilyen USER halmazt generáló kiválasztó az éppen fennálló külső körülményeket is figyelembe kell vegye. A következőkben nézzük meg, hogy milyen módon lehet egy új filter felületet létrehozni.

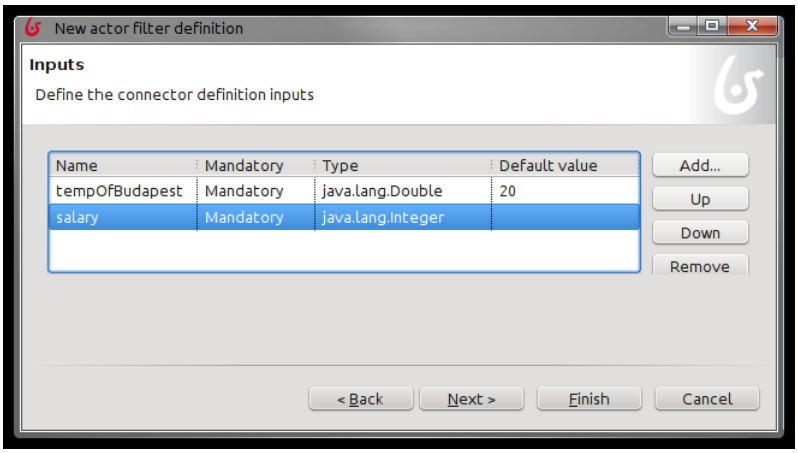



2.17. ábra. Saját készítésű Actor Filter - Interface - 1

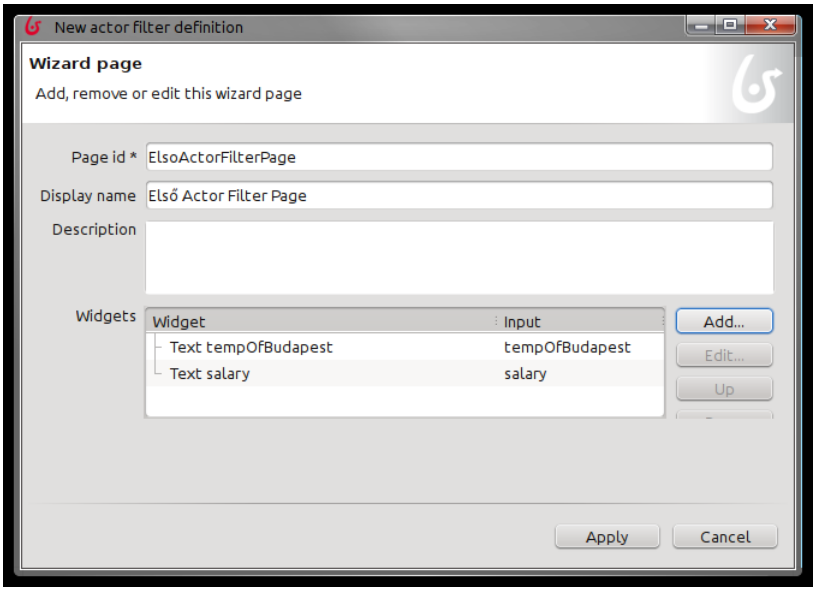
A 2.17. ábra mutatja a varázsló első lépését, ahol meg kell adnunk a filter definició nevét. Ez esetünkben most *ElsoActorFilter* lesz. Tekintettel arra, hogy itt még a varázsló működését is meg kell adjuk, ezért célszerű szépen kitölteni ezeket a mezőket is:

- *Display name*: amikor használjuk a wizzard-ot, ezt fogjuk látni
- Egy szép ikont is rendelhetünk a varázslóhoz

- A *Description* rendes kitöltése (ami itt elmaradt) elemi érdekünk, hiszen bizonyos számú filter után már jól jön az ide beírt kis help, azaz mit is csinál a filter
- A *Categories* beszédes és találó megadása is nagyon lényeges. Mi is vehetünk fel újat, de használhatunk meglévőt is. Ez egy olyan név, ami valamilyen szempontból összefogja az összetartozó filter definíciókat, más szerepe nincs, de ez nagyon hasznos
- Végül a *Dependencies* az opcionálisan megadható külső Java könyvtárak (jar fájlok) listája, ugyanis ilyenekre természetesem szükség lehet (például egy adatbázisból is kell majd olvasni, akkor annak a JDBC drivere biztos kell).




2.18. ábra. Saját készítésű Actor Filter - Interface - 2



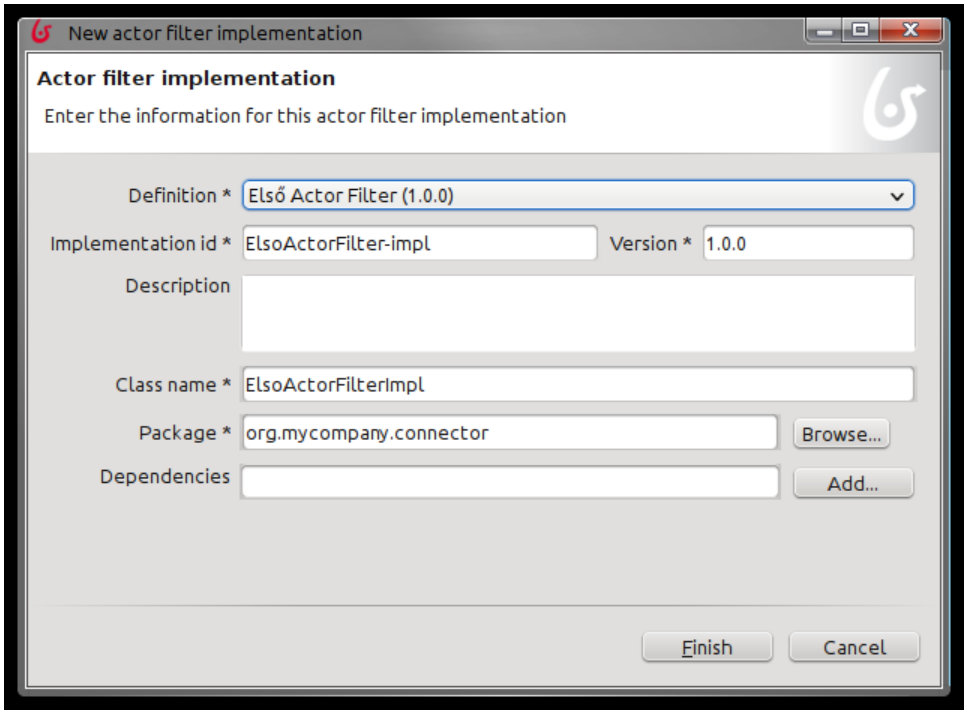

2.19. ábra. Saját készítésű Actor Filter - Interface - 3

A *Next* gomb megnyomása után a 2.18. ábra dialógus ablaka tárul elénk. Itt megadhatjuk, hogy az *ElsoActorFilter* filterünk milyen input paramétereket fogad és használ, amikor az eredmény USER halmazt legyártja. A 2.18. ábrán minden mélyebb értelem nélkül 2 paramétert vettünk fel az *Add...* gomb segítségével:

- *tempOfBudapest* → amikor fut, akkor éppen ekkor Budapest hőmérséklete
- *salary* → aki ennél kevesebbet keres, az ne kerüljön be az eredmény USER halmazba

Ismét nyomjuk meg a *Next* gombot, hogy a 2.19. ábra Page Wizard képernyőjéhez jussunk. Itt megadunk egy lap azonosítót, amire 2 Widget-et (grafikus form vezérlő) tettünk fel és összerendeltük a megfelelő input paraméterrel. A kapott összerendelést a 2.19. ábra alsó részében figyelhetjük meg. Amikor a varázslót majd használjuk, mindez még érthetőbb lesz a későbbiekben.

Eddig elvégeztük az új filter definíciós feladatait, most az implementáció van hátra, azaz meg kell mondanunk, hogy ilyen felülettel szemben milyen módon működjön a filterünk.




2.20. ábra. Saját készítésű Actor Filter - Implementáció - 1

A 2.20. ábra dialógus ablakát úgy kaptuk, hogy a *Development* → *Actor filters* → *New Implementation* menüpontot választottuk ki és kértük az *Első Actor Filter* implementációját. Természetesen itt egy Java class készül, ezért annak meg kell adnunk az

- osztálynevét (*ElsoActorFilterImpl*)
- csomagját (*org.mycompany.connector*) és

- külső *jar* Java könyvtár függőségeit

A *Finish* gombra végül eljutunk egy kódszerkesztőbe, ahova az implementáció előre legenerált váza kerül (2.21. ábra). Amennyiben a kódra tekintünk, akkor azt kell leghamarabb megértenünk, hogy a *filter()* metódus által visszaadott *List<Long>* halmaz jelenti a USER halmazt, ugyanis a user-ek belső azonosítói *Long* típusúak. A *filter()* feladata tehát az, hogy egy *Long* elemekből álló listát adjon vissza, amit a célnak megfelelő algoritmussal állít elő. A példakód azt mutatja, hogy milyen szellemben lehet itt implementációt írni. A továbbiakban bemutatjuk az itt használt API néhány tipikus részét, de ezek csak példák, a teljesség igénye nélkül.

Így férhetünk hozzá a most éppen használatos process példány azonosítójához:

```
processInstanceId = getExecutionContext().getProcessInstanceId();
```

Ebből azt is megtanulhatjuk, hogy mindegyik process példány (instance, case) szintén egyedi azonosítót kap, amik *Long* típusúak és kulcsként funkcionálva a process azonosítói az adatbázisban. A 2.21. ábra kódja a process indítójának a főnökét adja vissza, illetve annak user azonosítóját. Ez ugyanaz a filter, ami a gyári filterek között az utolsó volt, azaz a User manager.

A következő kód a process-t indító felhasználót adja vissza:

```
long processInitiator = getAPIAccessor().getProcessAPI().getProcessInstance(processInstanceId).➥
    getStartedBy();
```

Amennyiben egy user nevét tudjuk, úgy annak *User* objektumához így férhetünk hozzá:

```
User user = getAPIAccessor().getIdentityAPI().getUserByUserName( actorName );
```

Ezután a *ezzel a* hívással már USER ID-ja is megszerezhető.

```
user.getId();
```

Ezek után már az *Initiator manager* gyári filtert is el tudnánk készíteni. Amennyiben a paraméterekhez is hozzá szeretnénk jutni, akkor ezt az ősben automatikusan generált getter metódussal tehetjük meg:

```
double temp = getTempOfBudapest();
int salary = getSalary();
```

Innentől kezdve a USER halmaz paraméterezett legyártásához is hozzákezdhetnénk. Például írhatnánk egy olyan filtert, ami úgy működik, hogy amennyiben 20 foknál melegebb van, úgy a hierarchiában felfelé azt az első főnököt adja vissza, aki megfelelően sokat keres. Lehet, hogy ez csak a főnök, főnökének főnöke lesz... A példa azt érzékelteti, hogy a paramétereknek milyen fontos szerepük van a dinamikusan kiszámított USER halmaz legenerálásában. Ugyanakkor azt is vegyük észre, hogy ez a mechanizmus végtelenül rugalmas, lényegében bármilyen összetett szempont szerint megvalósítható vele, hogy kik legyenek az ACTOR mögött a munkafolyamat futása során.

Végezetül kiemeljük, hogy a filter is egy újrahasznosítható komponens, ezért azok a *Development* menüpont használatával exportálhatók/importálhatóak.

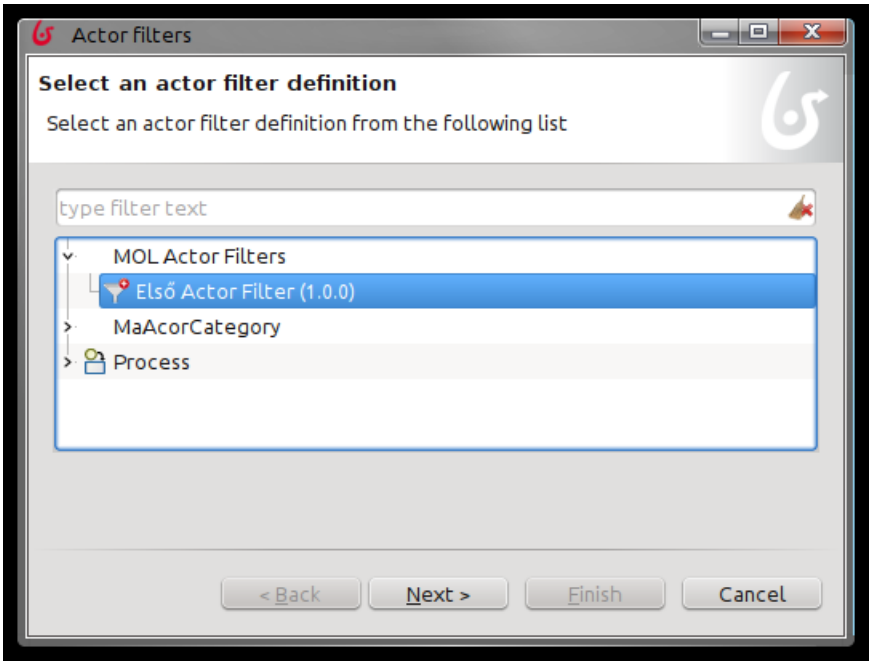
```java
package org.mycompany.connector;
import java.util.Arrays;
import java.util.List;
import org.bonitasoft.engine.connector.ConnectorValidationException;
import org.bonitasoft.engine.filter.UserFilterException;
import org.bonitasoft.engine.identity.User;
/**
 *The actor filter execution will follow the steps
 * 1 - setInputParameters() --> the actor filter receives input parameters values
 * 2 - validateInputParameters() --> the actor filter can validate input parameters values
 * 3 - filter(final String actorName) --> execute the user filter
 * 4 - shouldAutoAssignTaskIfSingleResult() --> auto-assign the task if filter returns a single result
 */
public class ElsoActorFilterImpl extends AbstractElsoActorFilterImpl {

    @Override
    public void validateInputParameters() throws ConnectorValidationException {
        //TODO validate input parameters here
    }

    @Override
    public List<Long> filter(final String actorName) throws UserFilterException {
        //TODO execute the user filter here
        //The method must return a list of actor id's
        //you can use getApiAccessor() and getExecutionContext()
        final long processInstanceId = getExecutionContext().getProcessInstanceId();
        try {
            long processInitiator = getAPIAccessor().getProcessAPI().getProcessInstance(processInstanceId).getStartedBy();
            long l = getAPIAccessor().getIdentityAPI().getUser(processInitiator).getManagerUserId();
            // actorName vagy bármelyik userName (pl: inyiri)
            User user = getAPIAccessor().getIdentityAPI().getUserByUserName( actorName );
            user.getId();
            // Így jutunk hozzá az átadott paraméterekhez
            double temp = getTempOfBudapest();
            int salary = getSalary();
            return Arrays.asList( l );
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return null;
    }

    @Override
    public boolean shouldAutoAssignTaskIfSingleResult() {
        // If this method returns true, the step will be assigned to
        //the user if there is only one result returned by the filter method
        return super.shouldAutoAssignTaskIfSingleResult();
    }
}
```

2.21. ábra. A filtert implementáló Java kód megírása

## A saját készítésű Actor Filter használata

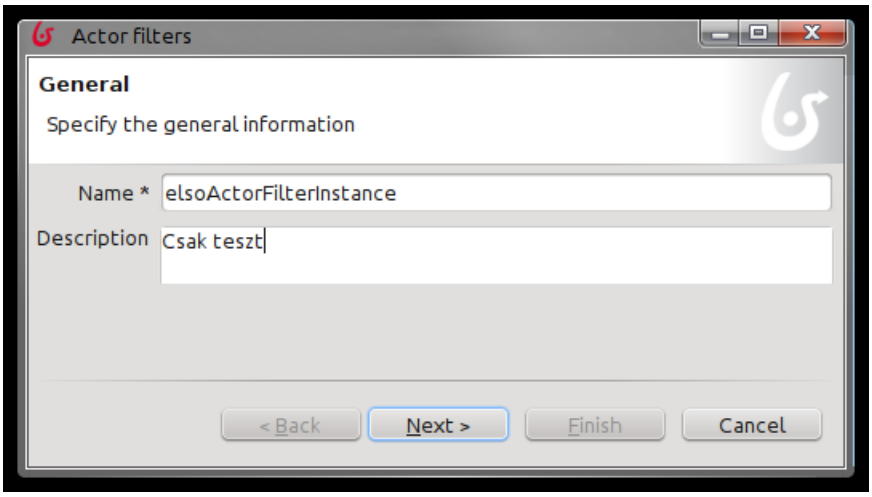
Példaképpen tekintsük a 2.1. ábra egyszerű process diagramját és kattintsunk az *Approving* TASK-ra, amihez a USER halmazt a most elkészített filterrel szeretnénk hozzárendelni. Ha a *General* → *Actors* fülre megyünk, akkor ismét a 2.3. ábra formját kapjuk, ahol nyomjuk meg a *Set...* gombot. Emlékezzünk rá, hogy ettől függetlenül korábban már beállítottuk ennek a placeholder feladatot ellátó ACTOR nevét, azaz a *MOL PAW Actor*-t. A *Set...* gombra a 2.22. ábra dialógus ablaka

jelenik meg.

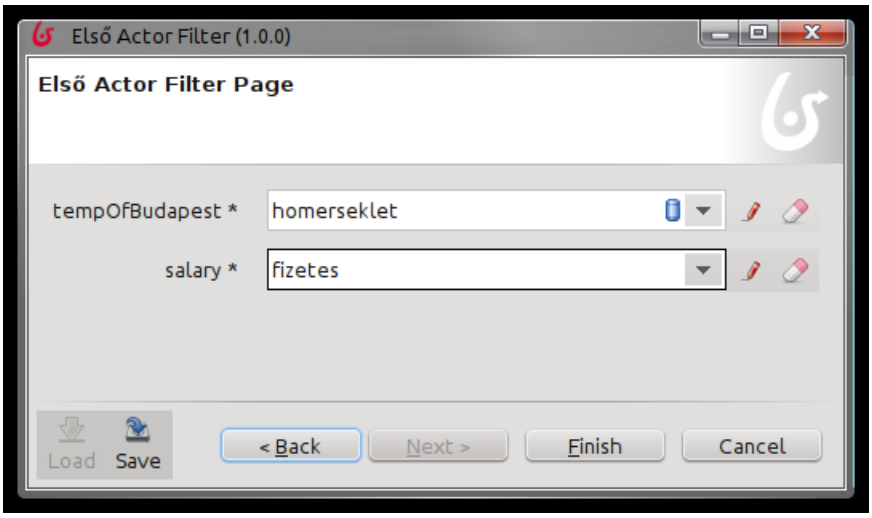



2.22. ábra. Saját készítésű Actor Filter - Használat - 1

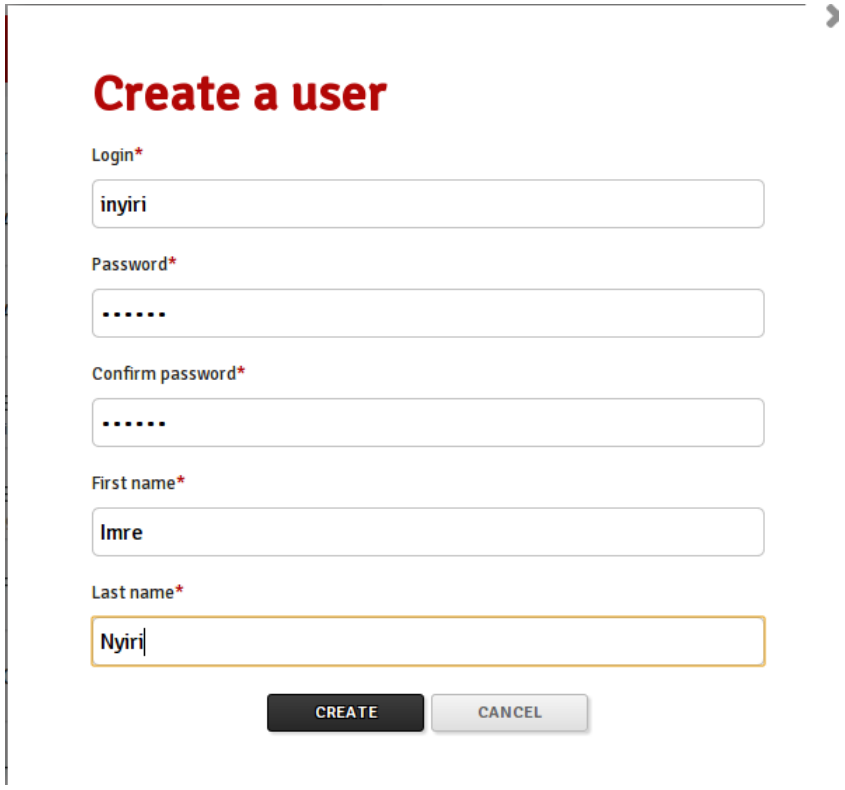
Látható, hogy a kategóriák között ott van a mi *MOL Actor Filters* kategóriánk is, illetve alatta a most elkészített egyetlen, saját filterünk is. Válasszuk ki és nyomjuk meg a *Next* gombot! Megjelenik a 2.23. ábra dialógus ablaka, ahol elkezdjük elkészíteni a konkrét actor filter példányt erre a konkrét (TASK és ACTOR) esetre.




2.23. ábra. Saját készítésű Actor Filter - Használat - 2




2.24. ábra. Saját készítésű Actor Filter - Használat - 3




2.25. ábra. Új user felvétele a portálon

Ezután újra *Next*, majd a filter 2 input paramérét rendeljük össze (2.24. ábra) a workflow korábban létrehozott *homerseklet* és *fizetes* változóihoz. Amikor a filter elkezd működni, ezekből fogja dinamikusan kivenni az értékeket. A *Finish* gomb után a 2.3. ábra Actor Filter mezőjének most ez lesz az értéke: *elsoActorFilterInstance – Első Actor Filter*.

## Honnan jönnek a user-ek?

A Bonita saját user adatbázissal rendelkezik, ezt a Portálon keresztül lehet menedzselni admin módban (2.25. ábra). Biztosítani lehet a kapcsolatot más user adatbázisokkal úgy, hogy ide szinkronizálunk. Nem mindent, de akkor a külső adatbázishoz való rekordkulccsal rendelkezni kell (példa: domain/user info AD esetén).

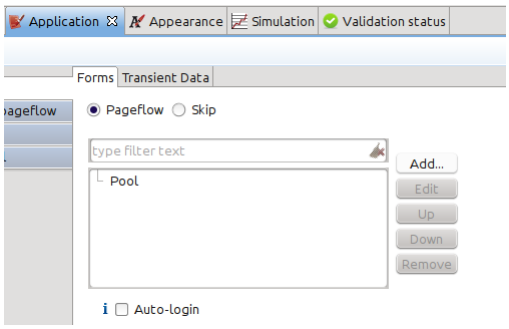
Amennyiben Bonita Stúdióba akarunk fejlesztés közben új user-t felvenni, akkor az egyik lehetőség az actor mapping XML használata (2.26. ábra), ahol az XML fájlba betesszük az új *<user>* tag-eket, majd az így keletkezett XML-t importáljuk ott, ahol a kezdeti XML-t exportáltuk. Az eredmény *Actor Mapping* dialógus ablakban megjelenik.

```
<?xml version="1.0" encoding="UTF-8"?>
<actormapping:actorMappings xmlns:actormapping=
  <actorMapping name="Employee actor">

    <users>
        <user>inyiri</user>
    </users>

    <groups>
      <group>/acme</group>
    </groups>
    <roles/>
    <memberships/>
  </actorMapping>
  <actorMapping name="MOL PAW Actor">
    <users/>
    <groups/>
    <roles/>
    <memberships/>
  </actorMapping>
</actormapping:actorMappings>
```

2.26. ábra. Actor Mapping

## Anonymous User

Olyankor jön jól ez az új lehetőség, amikor olyan workflow-t készítünk, mely esetben nem hitelesített user is használhatná a form-jainkat. Például egy web áruháznál egy könyvrendelés, ahol elég később is tudni, hogy ki a user. Ezt a lehetőséget a Bonita Studió form tervezésekor adhatjuk meg, ahogy azt a 2.27. ábra mutatja.




2.27. ábra. Anonymous User

# 3. A Form designer eszközeinek használata

Ebben a fejezetben a Bonita Form fontosabb építő eszközeit tekintjük át. Amikor humán taszkunk van, akkor ez mindig egy FORM-ot is jelent, hiszen az embernek szüksége van egy olyan felületre, ahol elvégezheti a feladatát vagy annak adminisztrációját. Egy új munkafolyamat is gyakran úgy indul, hogy valaki kitölt egy eForm-ot és ezzel egy új case-t indít az útjára.

A Bonita rendszer folyamataihoz 4 féle form építő megoldást tudunk alkalmazni:

1. A Bonita Stúdió Form builderét és annak beépített vezérlőit használjuk. Ezen eszközzel rendkívül gyorsan össze lehet állítani egy-egy formot, a minősége is megfelelő, de nagyon kifinomult igények esetén elérkezhetünk a határához.

2. Továbbra is a Form buildert használva, azt kiegészítjük sok egyedi elemmel és Javascript (például *JQuery*) részlettel.

3. Lehetőség van arra, hogy a Redirect URL opció választásával egy Form-ot ne a Bonita Stúdió eszközeivel építsünk össze, hanem egy Java, .NET vagy PHP technológiával, amelyek külső web alkalmazásként tudják ezeket a kéréseket fogadni.

4. A workflow-hoz kötődő teljes alkalmazást a 3. pontban említett valamely külső eszközzel készítjük el, ami a Bonita API-n keresztül éri el a Bonita Engine-t.

A pontok között lefelé haladva egy több lehetőségünk van a felhasználói felület testreszabására, azonban a hozzá szükséges szaktudás igény is egyre nagyobb lesz. Ebben a cikkben leginkább az 1. pont beépített lehetőségeit igyekszünk áttekinteni.

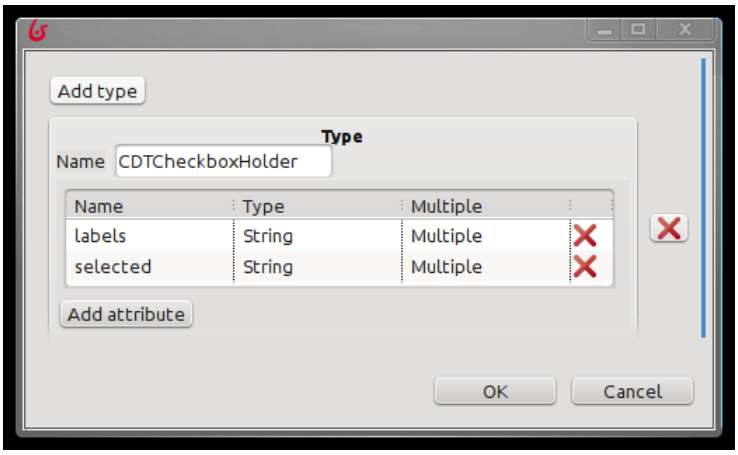
## Checkbox lista Widget

A checkbox lista vezérlő (*widget*) feladata, hogy felkínáljon egy választéklistát, amiből több elemet is választhatunk. Ezt akár úgyis felfoghatjuk, hogy a kiválasztott elemek Yes (true) értékűek, míg az összes elemből visszamaradt, nem kiválasztott tagok No (false) értékűek. Ez a vezérlő a következő értékekkel és típusokkal dolgozik:

- Az *Available values* mező: egy *List<String>* típusú érték (vagy Java *Map*), azaz *String*-ek listája. Ez tartalmazza az összes elemét a checkbox listának, függetlenül attól, hogy az ki van-e pipálva.

- A *Selected value* mező: egy *List<String>* típusú érték, ami az *Available values* mezőnél lévő értékek egy részlistája. Azt mondja meg, hogy mely listaelemek legyenek kipipálva.

Az ad-hoc típusok és megoldások helyett érdemes egy olyan osztályt elkészíteni, ami képes visszaadni ezt a 2 *List<String>* értéket. A *Development→Data types* menüpontnál hozzunk létre egy új osztályt (és jar-t természetesen), ahogy azt az 3.1.ábra mutatja.

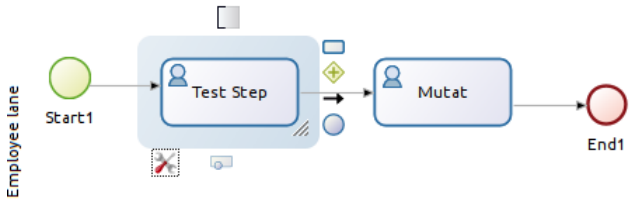



3.1. ábra. A *CDTCheckboxHolder* osztály

A *CDTCheckboxHolder* class (CDT=Common Data Types) úgy valósítja meg a 2 *List<String>* tárolását, hogy mindkettőt tárolja:

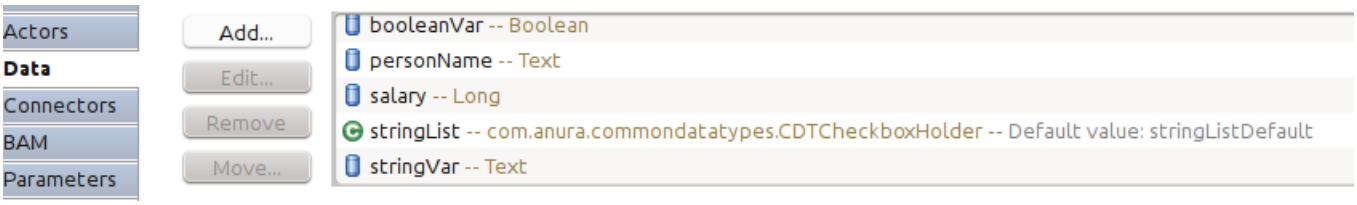
- *labels*: a feliratok (és ezzel az összes érték) tárolása,
- *selected*: a kiválasztott értékek tárolására

Lehetne 1 db *List<String>* mező is, mellette egy *List<Boolean>*, ami adminisztrálja a mindenkori rész *List<String>*-et. Komolyabb hozzáállással egy külső, rendes fejlesztőeszközzel is készíthetünk ilyen class-okat, felszerelve sok hasznos metódussal.



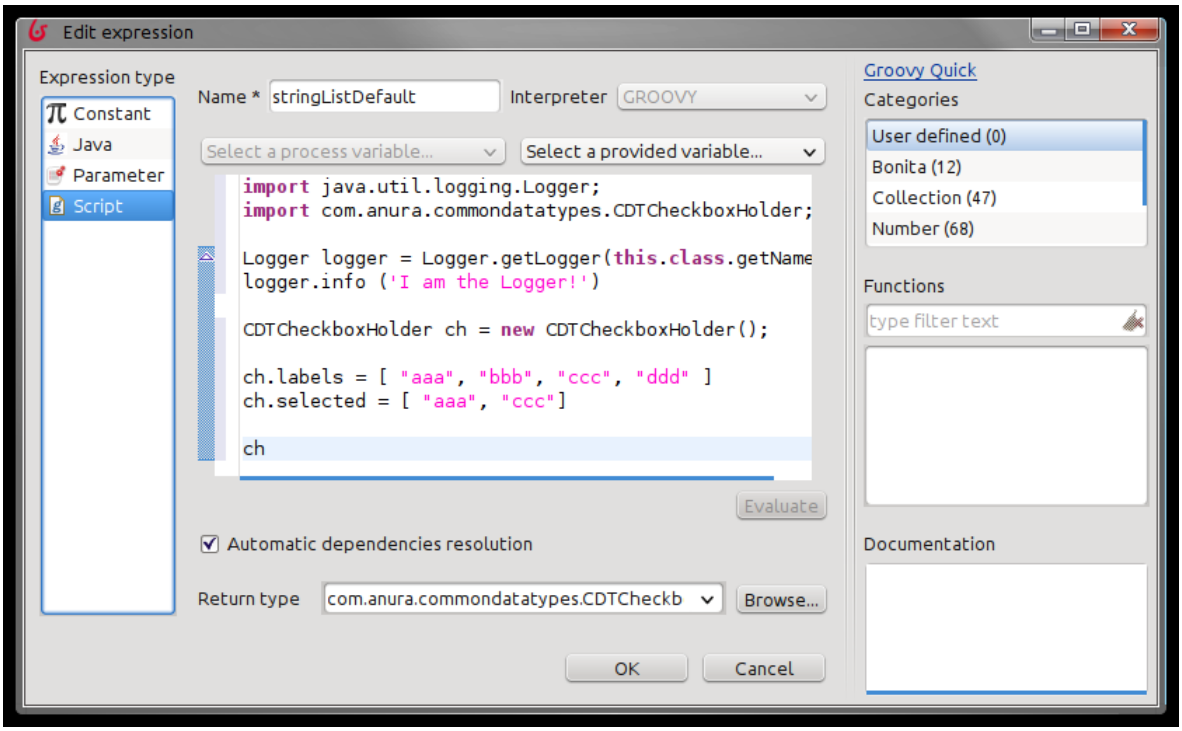

3.2. ábra. Egyszerű példa workflow

Tekintsük tovább a példánkat, amit a 3.2. ábra workflow-ján keresztül fogunk bemutatni és a példa készítésekor ilyen belső változói voltak (3.3. ábra):

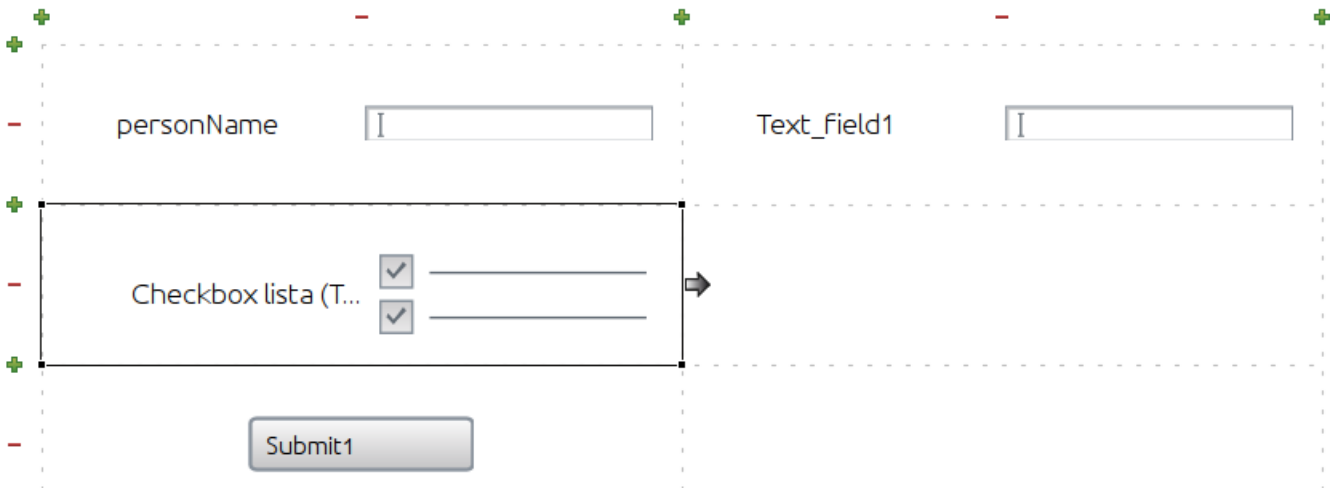



3.3. ábra. A 3.2. ábra workflow-jának belső változói

Vegyük észre, hogy felvettünk egy *stringList* változót is, aminek a korábban kialakított *CDTCheckboxHolder* lett a típusa. Adtunk neki kezdőértéket is, amit a 3.4. ábrán látható script valósít meg, elsősorban a tesztelés kedvéért szükséges, hogy ezen változóknak legyen valamilyen induló értékük.

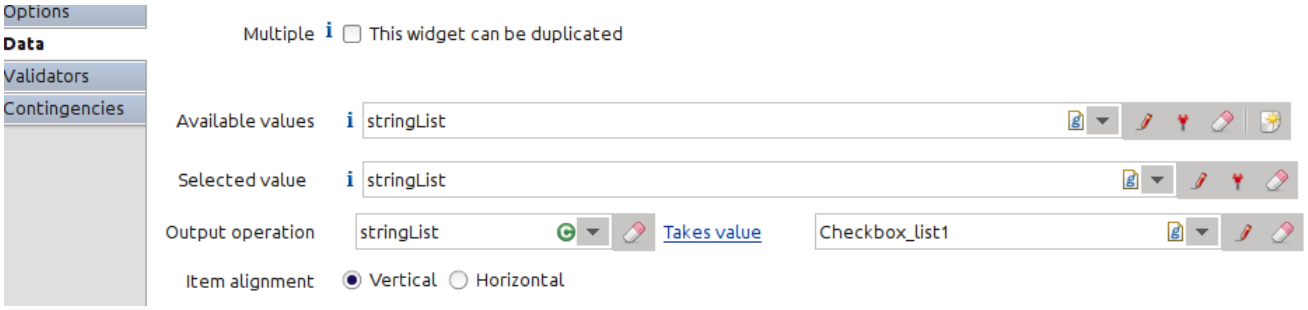



3.4. ábra. A *stringList* változó ezzel veszi fel a kezdőértéket

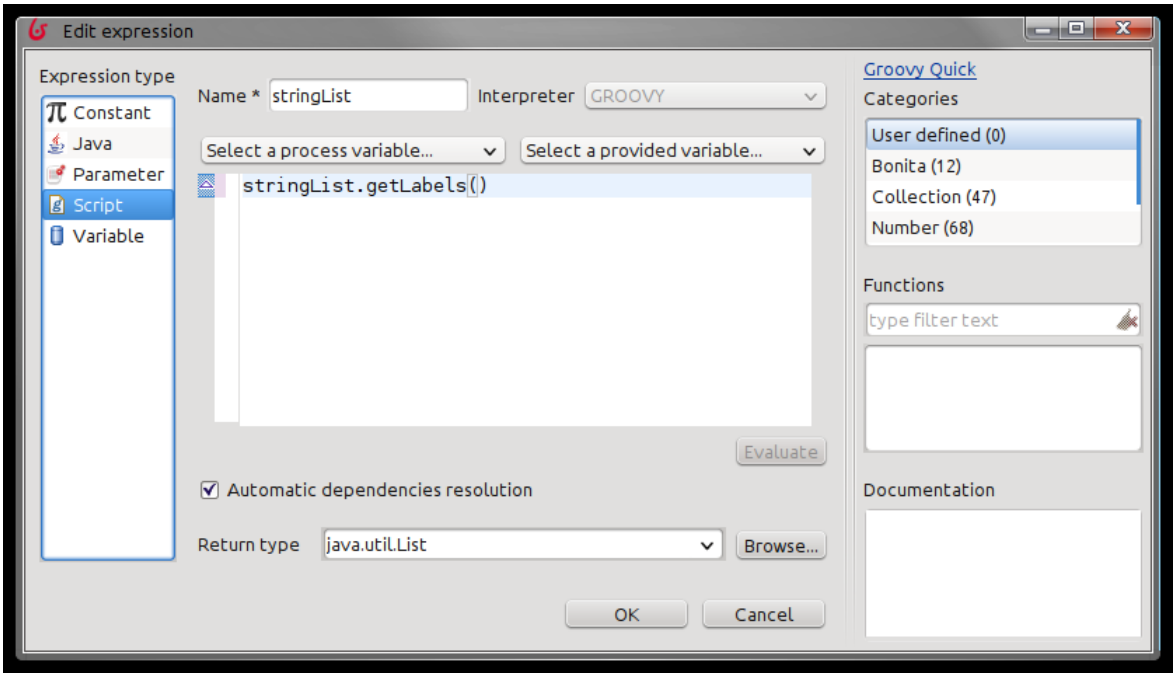



3.5. ábra. A *Test Step* TASK Form-ja

A 3.4. ábrára tekintve felhívjuk a figyelmet a *Return type* helyes megadására is, ez természetesen a mi osztályunk.




3.6. ábra. A checkbox lista beállításai




3.7. ábra. A *stringList* script, ami a nevét a *stringList* változóról kapta

Az első *Test Step* nevű task-ra generáljunk le egy képernyőt, amin csak a *personName* van, majd ehhez tegyünk még 2 új widget-et manuálisan fel a Form-ra, ahogy azt az 3.5. ábra mutatja. A 2 új vezérlő szerepe ez lesz:

- Egy checkbox lista, ami mutatja a *stringList* változó alakulását.
- Egy Text field, ami szintén a *stringList* változó alakulását mutatja, a *toString()* használatával.

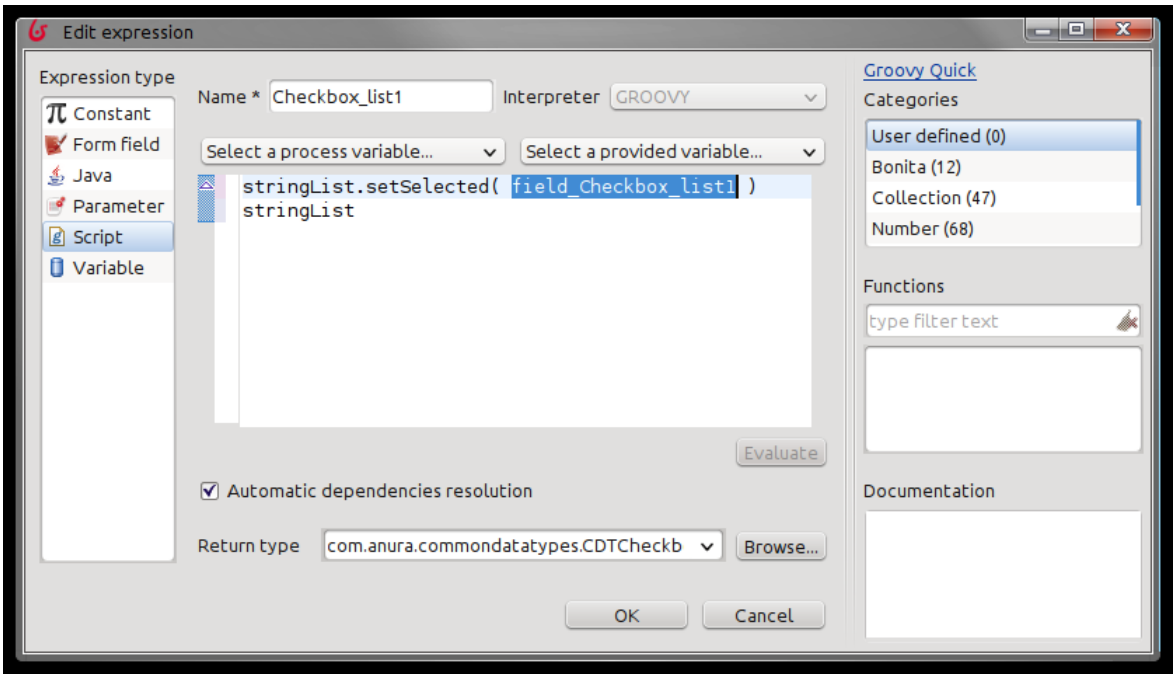
Nézzük meg előbb a checkbox lista beállításait (3.6. ábra)!

Itt fontos néhány apró részletet megérteni, ezért most jól figyeljünk! Az *Available values* mező egy *List<String>*, ahogy azt a korábbiakból tanultuk és a checkbox összes elemét szolgáltatja. A *stringList* név most nem a workflow változóra utal, hanem egy script nevére, amit ide kötöttünk (3.7. ábra) és az implementációja 1 sor, ami visszaadja a megfelelő *List<String>* értéket, a *Return type* is erre van beállítva. Amikor egy mező mellett jobbra egy kis *g* betűt látunk, az mindig azt jelenti, hogy mögötte egy *Groovy* script található.

A *Selected value* ehhez hasonló, de ennek a script megvalósítása értelemszerűen ez a sor lesz:

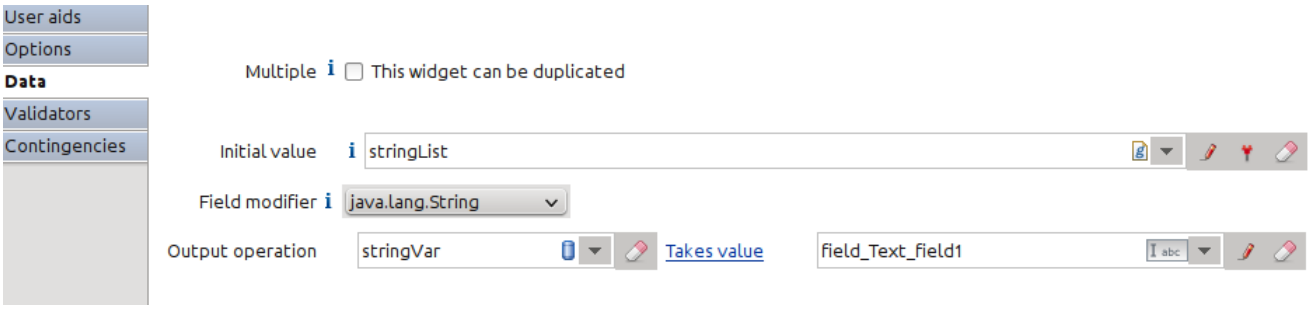
```
stringList.getSelected()
```

Az *Output operation* jelentése az, hogy hova, azaz mely workflow változóba tárolódik a checkbox listát jelképező *field_ Checkbox_ list1* változó tartalma. Az ilyen widget-ekhez kötődő változók mindig automatikusan jönnek létre amikor a vezérlőt feltesszük a white board-ra (a FORM tervező felülete) és csak a FORM szintjén láthatóak. Mi értelemszerűen az erre szolgáló *stringList* változónkba tesszük (3.8. ábra).




3.8. ábra. A *stringList* változóba így írunk vissza a checkboc listből

A 3.8. ábráról látható, hogy a script szépen visszaadja a megfelelő változót, magát a *stringList*-et használva erre, de előtte beállítja a selected adattagját a *field_ checbox_ list1* értékre (setter metódussal), ami mindig a checkbox lista kiválasztott tagjait tartalmazza. Ezzel bekonfiguráltuk a checkbox lista widget-et. Most nézzük meg a text field vezérlőt is! A konfigurációját a 3.9. ábra mutatja.




3.9. ábra. A text field mező konfigurációja

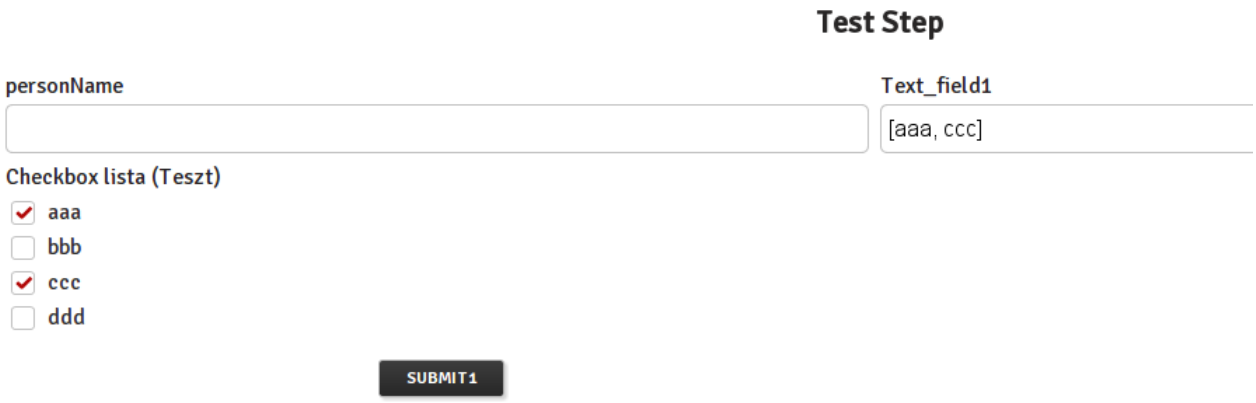
Az *Initial value* mögötti *stringList* nevű script kódja 1 soros és *java.lang.String*-et ad vissza.

```
stringList.getSelected().toString()
```

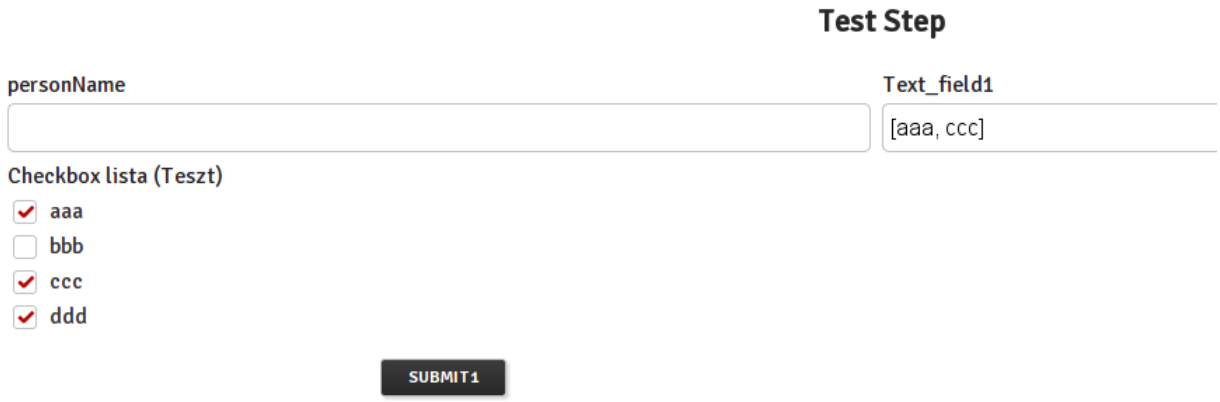
A *stringVar* egy *String* változója a workflow-nak, ez fogja tárolni a háttérben az értéket, ahogy az *Output operation* is mutatja. A *field_Text_field1* ez ezzel kompatibilis mező változó, itt tehát semmi nehézség nincs, egyszerű értékadás történik a mező és workflow változó között, amikor a submit gombra lefutó form action lefut. Ezzel a *Test Step* TASK mögötti formmal végeztünk, most nézzük meg a *Mutat* step formját is. Ez csak azért készült, hogy kattintva a checkbox listán lássuk a változást a következő TASK formján. A 3.5. ábra formjával egyezik ezen TASK formja is, ahhoz képest a következő változtatásokat tettük:

- A checkbox lista címkéje *Mutatva a változást* szöveg lett.
- Nem használtuk az *Output operation* lehetőséget, de mégis szeretnénk kiemelni, hogy eddig mindig a *Takes value* link módszerét használtuk, azaz a jobb oldali mezőváltozó (vagy a script-je) által visszaadott értéket egyszerűen bemásoltuk (azaz kértük ezt a rendszertől) a target workflow változóba. A linkre kattintva annak egy beállító setter metódusa is meghívható lenne.

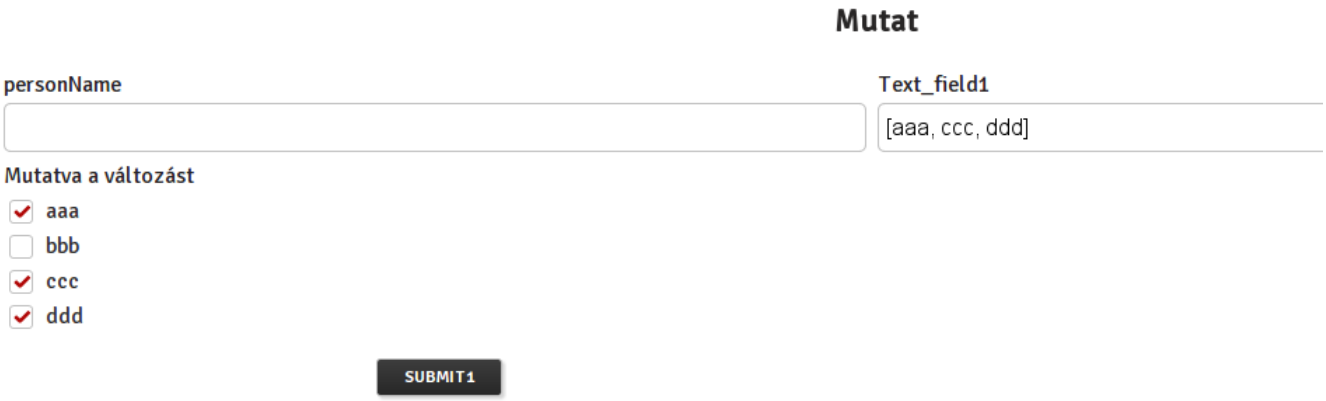
Készen vagyunk a *Mutat* TASK Form-mal is, teszteljük le a működést!




3.10. ábra. Megjelenik a *Test Step* form




3.11. ábra. Bepipáltuk a *ddd* checkbox-ot is

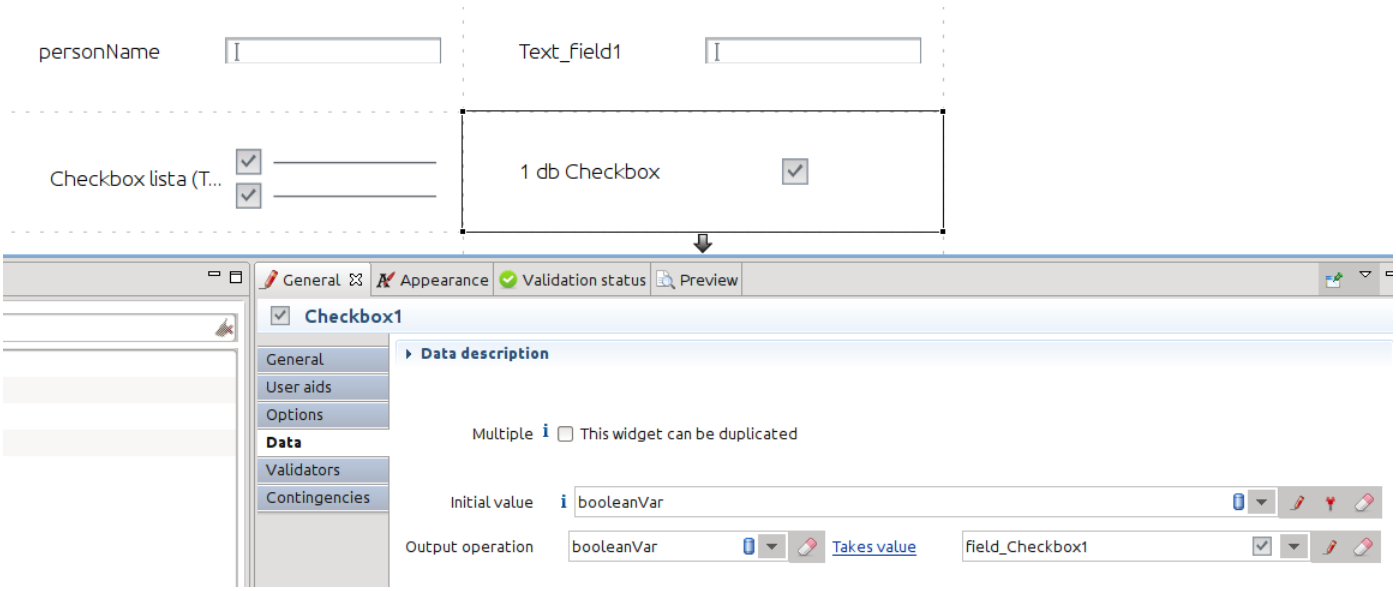



3.12. ábra. A *Mutat* TASK formja

A 3.10. ábra mutatja, hogy valóban a szándékolt, előre inicializált *stringList* értékek, azaz *aaa* és *ccc* lettek kipipálva. Válasszuk ki a *ddd* checkbox-ot is (3.11. ábra). A *SUBMIT1* gomb megnyomása után átkerülünk a *Mutat* TASK formjára, ahol láthatjuk, hogy minden rendben van (3.12. ábra).

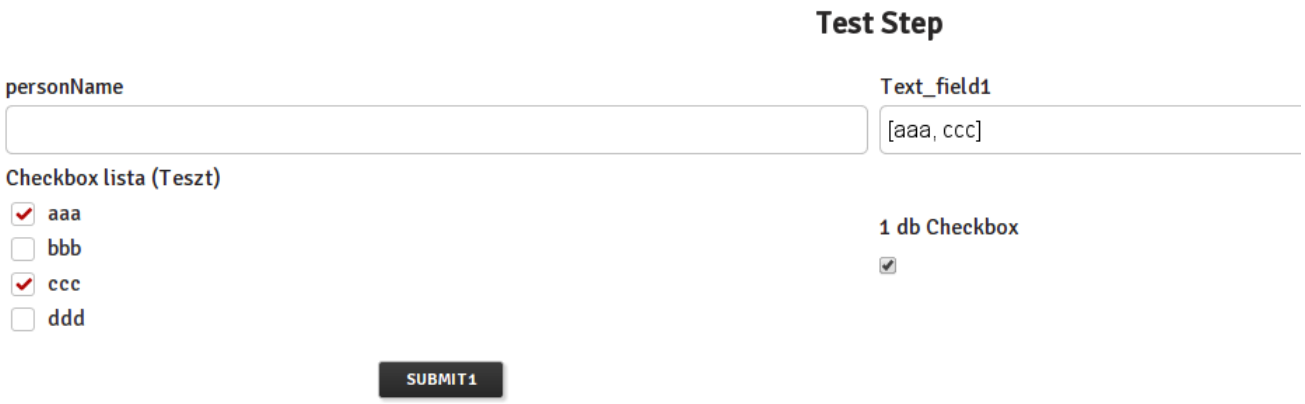
## Checkbox Widget

Az előzőekben bemutatott lista kistestvére a Checkbox field. Ez 1 darab checkbox elhelyezését jelenti, kezelése és konfigurálása is kevesebb műveletet igényel. A 3.3. ábrán láttuk a *booleanVar* workflow változót, erre építve mutatjuk be ezt a „pipás”, logikai Yes vagy No kezelését lehetővé mezőt. A 3.2. ábra *Test Step* és *Mutat* lépéséhez tartozó Form-ra is felteszünk egy olyan checkbox widget-et, ami a *booleanVar* értékét kezeli és mutatja. Jegyezzük meg, hogy ellentétben a listás checkbox-szal, itt a háttér változó mindig egy *java.lang.Boolean* típusú.



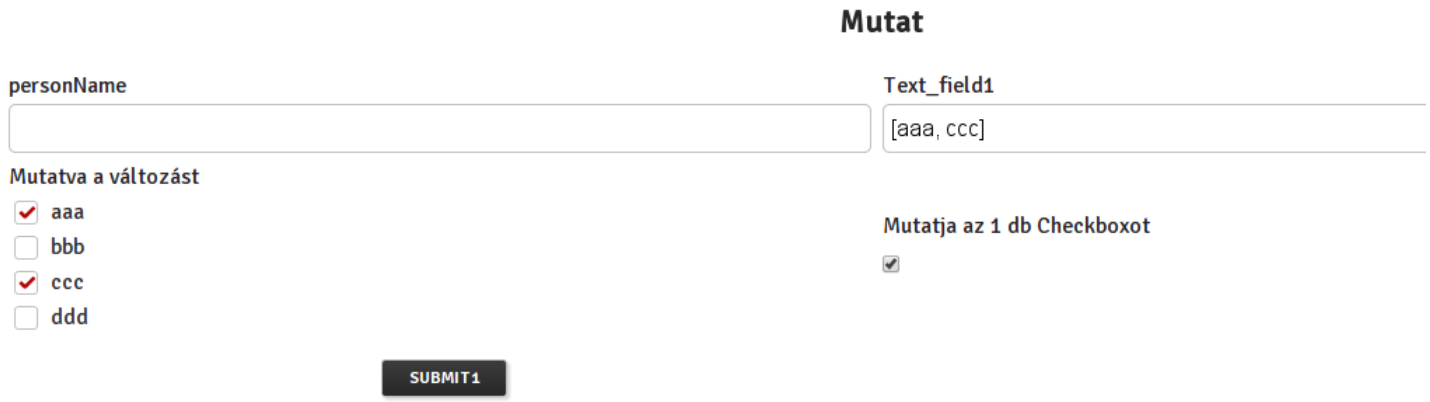

3.13. ábra. A Checkbox és a háttér adatkonfigurációja

A 3.13. ábra mutatja, hogy az *Initial value*, azaz a checkbox induló állapota a *booleanVar* változóból lesz beállítva. Az *Output operation* is ebbe a *booleanVar* változóba teszi a *field_ Checkbox1* mezőváltozó által reprezentált értéket. Aki nem teljesen érti, hogy mik ezek a mezőváltozók, azok gondoljanak arra, hogy böngésző oldalon reprezentált változók, ezek azok, amik közvetlenül változnak, amikor a felhasználók kezelik a Form-okat. Egy submit (vagy más action, azaz szerver oldalra hívás) az, amikor a Bonita képes ezen „böngészős" változókat a workflow változóba is visszatenni. Ezt értjük *Output operation* alatt minden widget esetén. Aki ismeri a Google GWT webes keretrendszert, az gondolhat ezekre a mezőváltozókra úgy, ahogy ott is használjuk.




3.14. ábra. Tesztelés - A checkbox-ot kipipáltuk és megnyomtuk *SUBMIT1*-et

A 3.14. és 3.15. ábrák már a tesztelést mutatják. A *Test Step* TASK form-ján (3.14. ábra) kipipáltuk az „1 db Checkbox"-ot, amit a *Mutat* TASK formja (3.15. ábra) is megjelenít, kipipált állapotban.




3.15. ábra. Tesztelés - A checkbox pipás állapota átment a következő TASK-ra

## Text Field Widget

Egyszerű vezérlő, használatát a Checkbox lista bemutatása során részletesen láthattuk. Szeretnénk kiemelni a *Field modifier* beállítás lehetőségét, ezzel szabályozhatjuk, hogy a mezőváltozó ne az alapértelmezett *String*, hanem valamelyik ismert Java alaptípus (példa: *java.lang.Double*) legyen. Ennek megfelelően persze változik a field változó típusa is, így az *Oputput operation* célváltozó típusának is megfelelőnek kell majd lennie.

## Password Field Widget

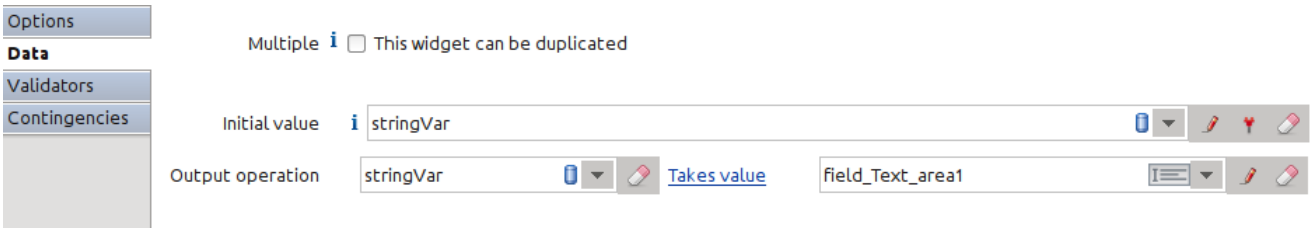
Ez a vezérlő egy olyan Text Field, ahol * karakterek jelennek meg a beírt vagy megjelenített tartalom helyett. Ez a mező azonban csak szöveges típust (*String*) fogad el, így ne is keressük a a Field modifier beállítás lehetőségét.

## Read Only Text Field Widget

A Text feliratú widget egy csak olvasható, azaz egy megjelenítő komponens. Ennek megfelelően az adat kapcsolata a workflow változóval rendkívül egyszerű, csak az *Initial value*-t kell megadni, nincs értelme az *Output operation*-nak hiszen nincs is lehetőség a változtatásra.

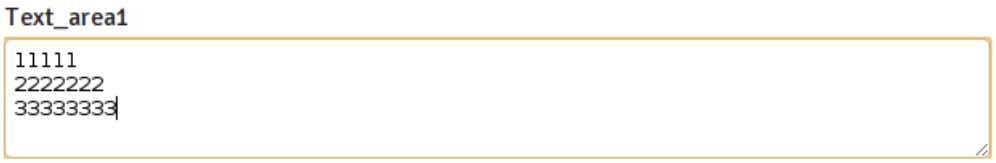
## Text Area Field Widget

Ez a vezérlő, hasonlóan más környezetekhez, a többsoros, egyszerű szövegek kezelésére alkalmas editor. A workflow és mezőváltozók konfigurálását a 3.16. ábra mutatja. A widget mögött csak *String* típusú változó állhat.




3.16. ábra. A Text Area widget konfigurálása

A 3.17. ábra futás közben mutatja a vezérlőt.




3.17. ábra. A Text Area widget futás közben

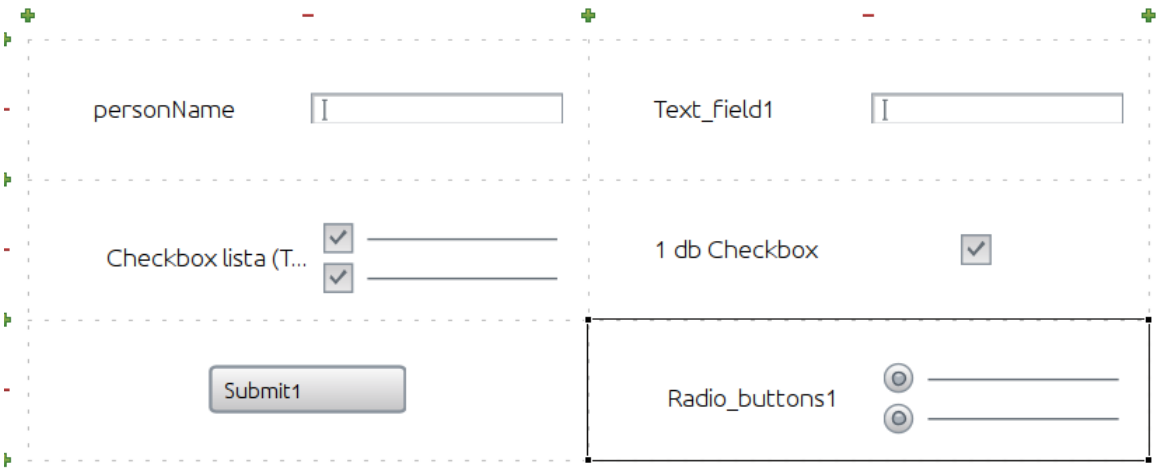
## Radio Buttons Widget

A rádiógombok listája a szokásos módon kezelődik, azaz mindig csak 1 lehet kiválasztva. A check-box listához hasonlít, de a *Selected value* ennek megfelelően csak egy *String* (és nem *String*-ek listája):

- Az *Available values* mező: egy *List<String>* típusú érték (vagy Java *Map*), azaz *String*-ek listája. Ez tartalmazza az összes elemét a rádió gomb listának, függetlenül attól, hogy az ki van-e választva.

- A *Selected value* mező: egy *String* típusú érték, ami az *Available values* mezőnél lévő értékek közül kerül ki. Azt mondja meg, hogy mely listaelem legyen kipipálva, azaz a widget megjelenésekor megnézi, hogy az *Available values* mező *String* listájának melyik elemével egyezik és azt check-eli be. Amennyiben egyikkel sem egyezik, úgy a rádió gombok egyike sem lesz kiválasztva.

A 3.18. ábra mutatja, hogy feltettük a *Test Step* formjára egy Radio Buttons widget-et.



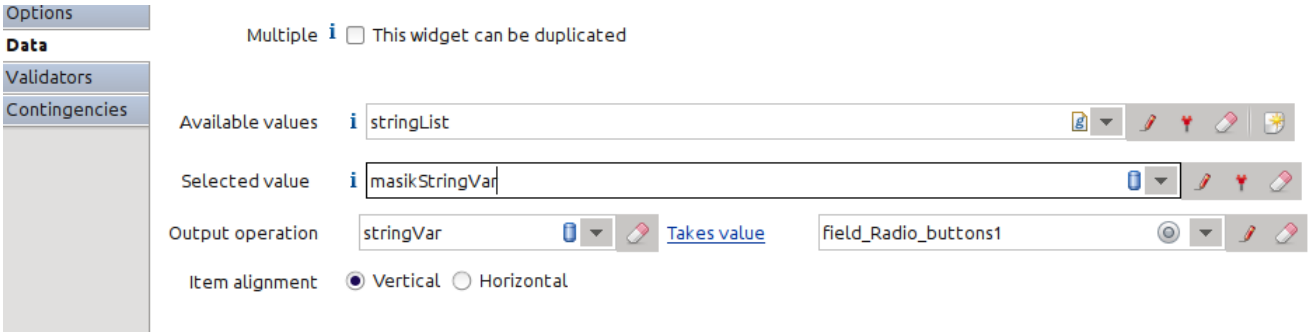

3.18. ábra. A Test Step TASK Form-jára feltettük a rádió gombok vezérlőt

Az ábrához néhány megjegyzés:

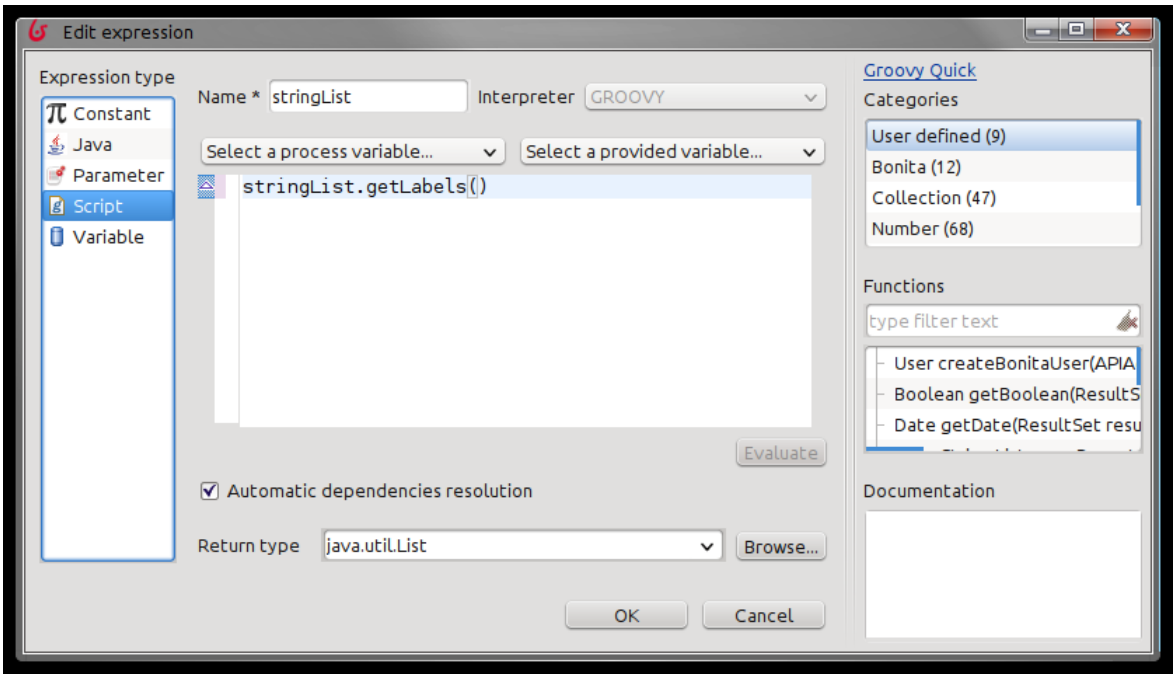
- Az *Available values* itt is ugyanaz a *stringList* változó névvel elnevezett 1 soros script (ezt látjuk, ha a ceruza ikonra kattintunk: *stringList.getLabels()*), amit a checkbox listánál használtunk (majd futáskor látjuk, hogy emiatt a látható értékek is egyeznek). A visszatérési érték is *java.util.List*. Mindezt a 3.20. ábrán meg is tekinthetjük.

- A *Selected value* egy egyszerű *String* változó: *masikStringVar*. Az értéke induláskor *aaa*, így emiatt majd ez a radio button lesz kiválasztva induláskor.

A konfigurációt a 3.19. ábra mutatja.

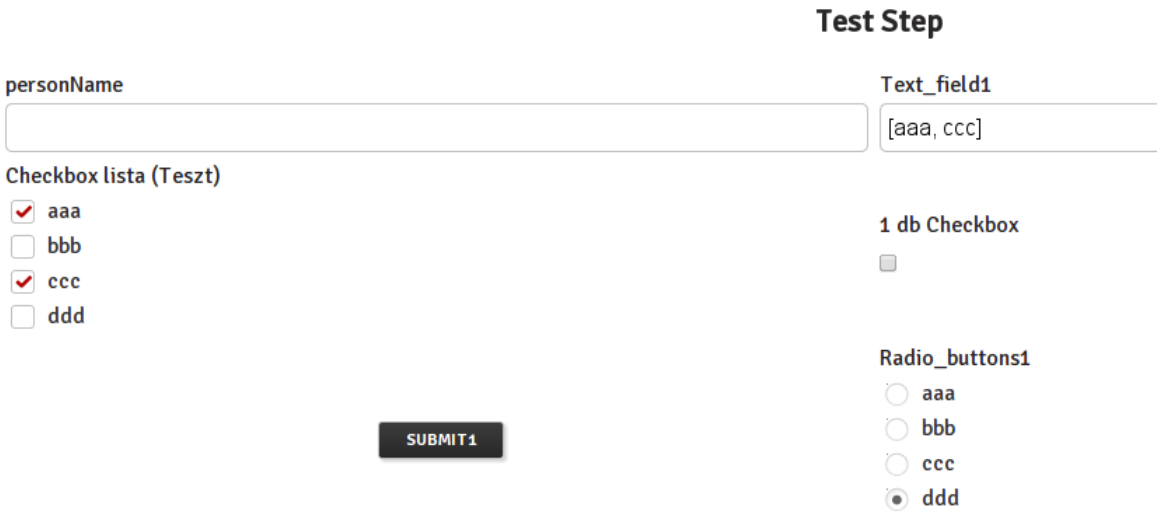



3.19. ábra. A Radio Buttons vezérlő konfigurációja

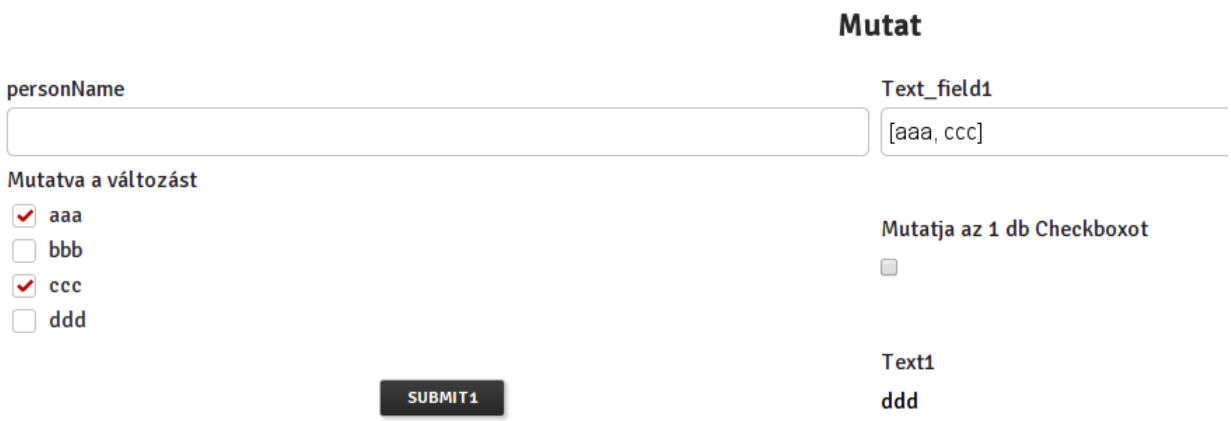



3.20. ábra. A Radio Buttons elérhető értékei

Az *Output operation* egyszerű, a *stringVar* változóba tesszük a *field_Radio_buttons1* mezőváltozót, értelemszerűen mindkettő típusa *String*, így egy action után a *stringVar* fogja tartalmazni a kiválasztott gombot reprezentáló *String* értéket. Nézzük meg mindezt futás közben is! Nem írjuk le, de a *Mutat* step formjára kitettünk egy read-only Text mezőt, hogy lássuk a rádio button-nal kiválasztott értéket. A futást a 3.21. és 3.22. ábrák mutatják.



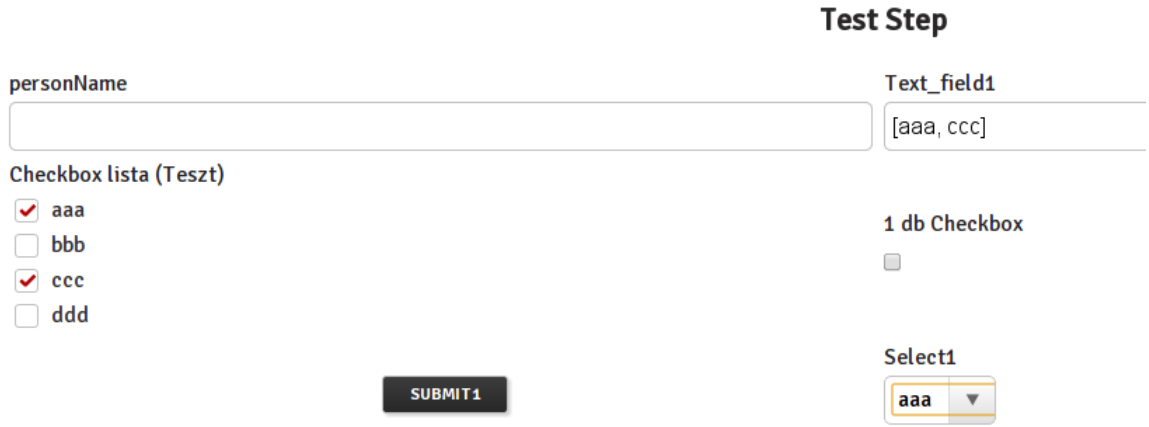

3.21. ábra. Kezdetben *aaa* volt kijelölve, *ddd*-re kattintottunk




3.22. ábra. ...és látjuk, hogy a *ddd* lett kiválasztva (jobb alsó vezérlő)

## Select (Dropdown) Field Widget

Ez a vezérlő pont úgy működik, mint egy rádió gomb, de az ismert dropdown (combobox) kinézetből lehet kiválasztani, ahogy azt a 3.23. ábra *Select1* mezője mutatja is.




3.23. ábra. A Select widget működés közben

A korábbi rádió gombok vezérlőt törölve, majd a Select komponenst téve a helyére, a konfigurációs beállítások pont azok lettek, mint amit a 3.19. ábra már megmutatott. Az egyetlen kivétel az, hogy a *Takes value* után most egy másik mezőváltozó van, esetünkben a *field_Select1* nevű és a típusa ugyanúgy ennek is *java.lang.String*.

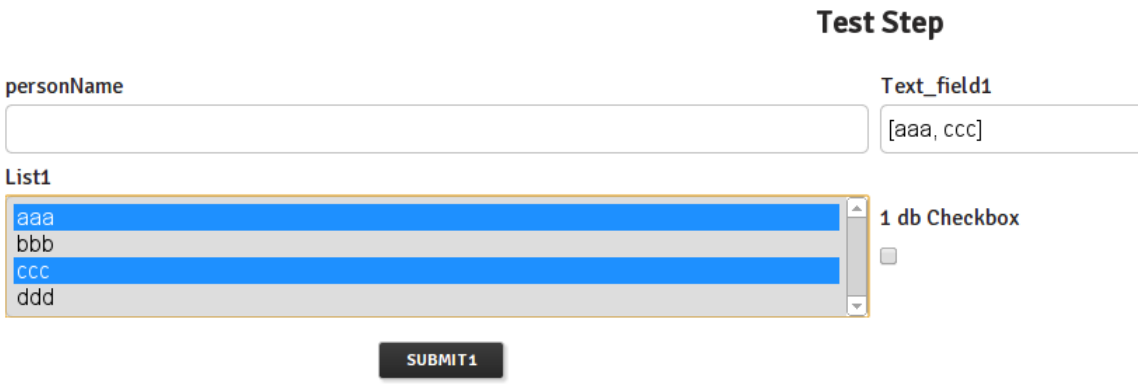
## List Field Widget

Ahogy a Select komponens a rádió gombok működését utánozza, úgy a List widget a Checkbox listáét. A következő 2 különbség van a kétféle vezérlő kinézete között:

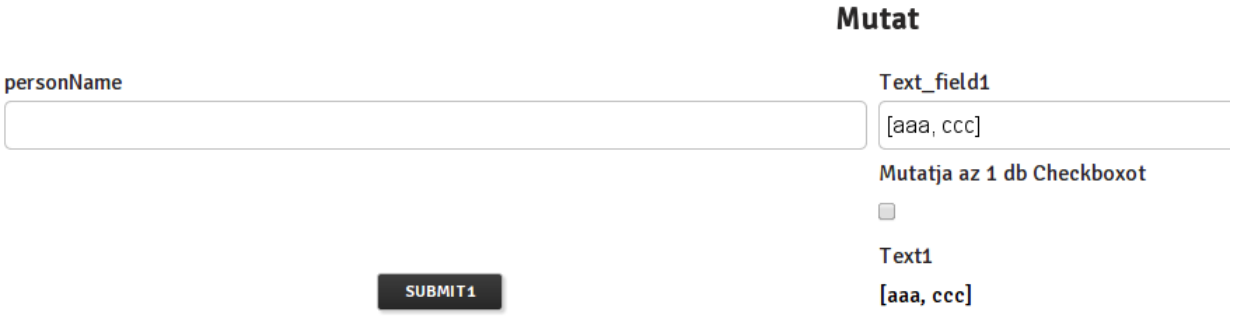
1. A checkbox lista és rádiógombok összes választható értéke egyből látszik, addig a Select vagy List komponensek esetén le kell nyitni a box-ot és esetenként scroll-ozni kell a kívánt érték megtalálásához.

2. Az első csoport sok helyet vesz el a képernyőről, így sok választható adat esetén csak a Select és List változatok jöhetnek szóba.

Vegyük le a 3.18. ábra bal oldali checkbox listáját és tegyünk fek helyette egy List vezérlőt. A konfiguráció legyen pont olyan, mint a checkbox listáé, persze a mezőváltozó neve más lesz. A 3.24. és 3.25. ábra a futási eredményt mutatja. Kiválasztottuk a List-ből az *aaa* és *ccc* értékeket, amiket a *Mutat* TASK-ban meg is jelenítünk.



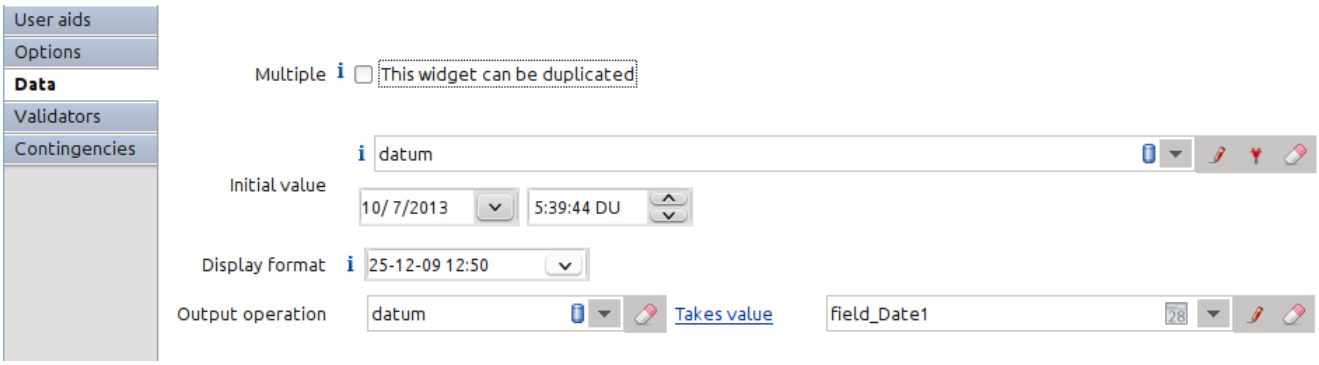

3.24. ábra. A List vezérlő működés közben, kiválasztva *aaa* és *ccc* értékeket




3.25. ábra. A List vezérlőben kiválasztott értékek megjelennek mindkét jobb oldali mezőben

## Date Field Widget

Vegyünk fel egy *datum* nevű, *java.util.Date* típusú workflow változót. A *Test Step* TASK formjára tegyünk ki egy Date vezérlőt és konfiguráljuk be a 3.26. ábra szerint.

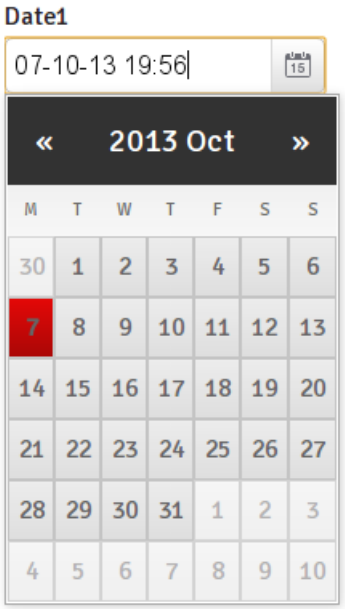



3.26. ábra. A Date widget konfigurációja

Az egyedüli érdekesség a *Display format*, itt lehet megadni a formátumot, amihez a 3.27. ábra karaktereit tudjuk használni. A többi konfigurációs mező a szokásos módon használható, mindössze a típusok helyessége (most *java.util.Date* kell legyen) a lényeges.

| | |
|---|---|
| G | Area designator |
| y | Year |
| M | Month in year |
| w | Week in year |
| W | Week in month |
| D | Day in year |
| d | Day in month |
| F | Day of week in month |
| E | Day in week |
| a | Am/pm marker |
| H | Hour in day (0-23) |
| k | Hour in day (1-24) |
| K | Hour in am/pm (0-11) |
| h | Hour in am/pm (1-12) |
| m | Minute in hour |
| s | Second in minute |
| S | Millisecond |
| z | Time zone |
| Z | Time zone |

3.27. ábra. A maszk karakterek



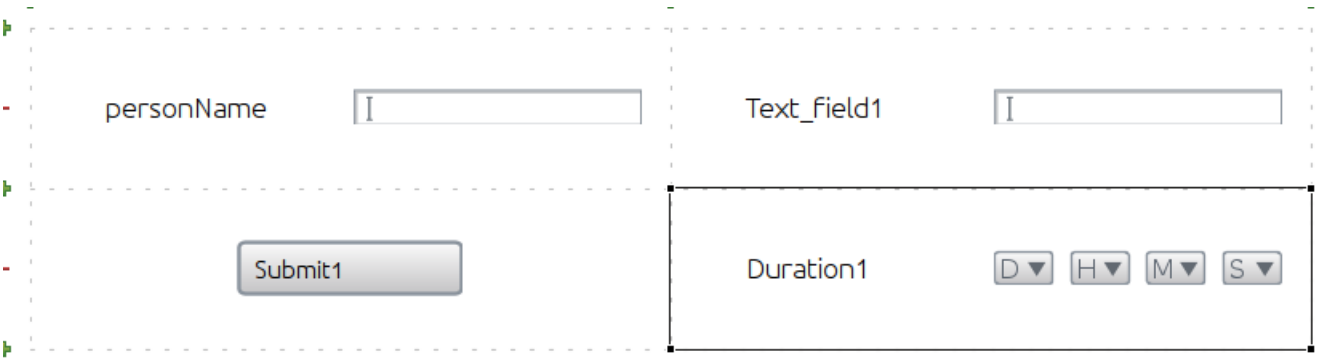

3.28. ábra. A Date vezérlő futás közben

A 3.28. ábra már a tesztelést mutatja. Az *Initial value* mezőbe természetesen bármilyen kifejezést betehetünk, így az legyen ez az 1 soros script: *new Date()*. Ekkor a *field_Date1* mezőváltozó ezzel inicializálódik és a 3a mai napi dátumot is megjelölte pirossal.

## Duration Field Widget

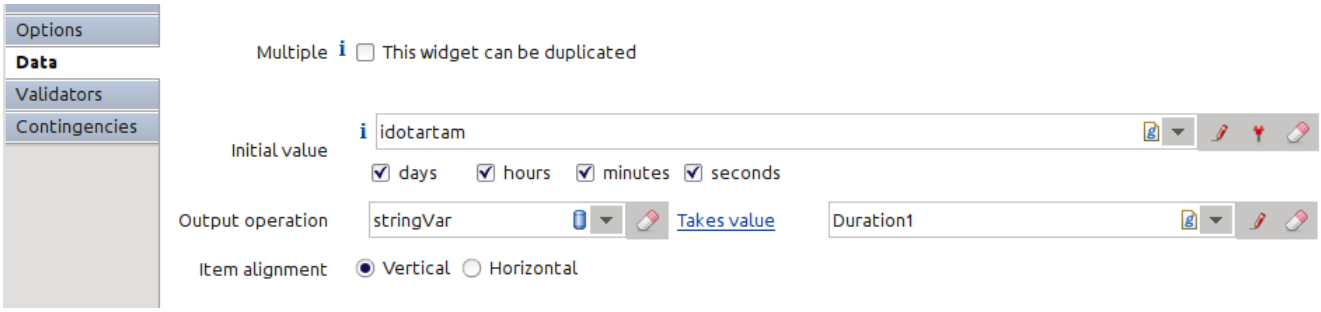
Vegyünk fel egy *idotartam* nevű workflow változót ezzel a típussal: *java.lang.Long*. Az időtartamot ezredmásodpercben tároljuk el. Egy időtartam azt jelenti, hogy valami hány nap, óra, perc,

másodpercet vesz igénybe. Tegyünk ki a *Test Step* TASK formjára egy Duration mezőt (3.29. ábra) és konfiguráljuk be 1 másodpercre (azaz 1000 ezredmásodperc).

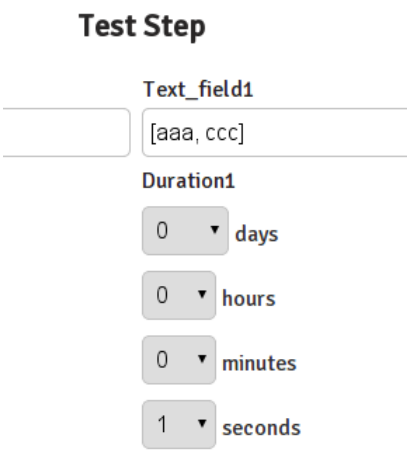



3.29. ábra. A Duration vezérlő (Days, Hours, Minutes, Seconds)

A konfigurációt a 3.30. ábra mutatja.




3.30. ábra. A Duration konfigurációja




3.31. ábra. *Test Step* Form


3.32. ábra. *Mutat* TASK formja

Az *Initial value*-t ez a script (vegyük észre a *g* betűt!) állítja be, *java.lan.Long* visszatérési értékkel:

```
idotartam=1000L
idotartam
```

Ez 1 másodpercet fog jelenteni, mint időtartam. Az eredményt az *Output operation* résznél a *stringVar*-ba tettük, ami egy *String* típus, ahogy azt korábban felvettük. A *Takes value Duration1* egy script, ezzel a kóddal:

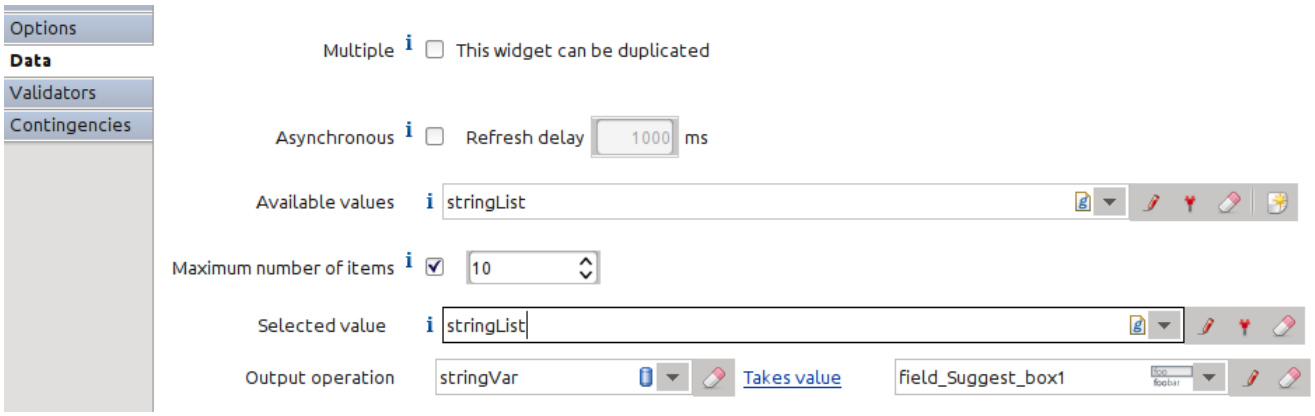
```
field_Duration1.toString()
```

Kész vagyunk az alapkonfigurálással, nézzük meg futás közben (3.31. és 3.32. ábra)!

## Suggest Box Field Widget

A Suggest Box egy Text Edit widget, ami úgy működik, hogy begépelés során mutatja azokat a lehetséges értékeket, amiket az eddig beírt prefix alapján még be lehet vinni. Amennyiben rátalálunk a mi keresett értékünkre, úgy az egy gombnyomással vagy egérkattintással egyből ki is választható. A 3.33. ábra azt mutatja, hogy a *Test Step* lépés formjára egy ilyen vezérlőt tettünk. A 3.34. ábra már ennek a komponensnek a használatát mutatja. Az *Output operation* részhez az ismert *stringVar* workflow változót írtuk, hiszen a *field_Suggest_box1* mezőváltozó is *String*.


3.33. ábra. Suggest Box widget a white boardon




3.34. ábra. A Suggest Box widget konfigurálása

Az *Available values* egy *java.util.List*, ami a felkínált értékek listája. A mutatott *stringList* egy script (a neve a szokásos módon abból a változóból képződik, amire lefut az 1 soros kifejezés):

```
stringList.getLabels()
```

A *Selected value* lehet üres, ekkor nem hoz be a vezérlő egy alapértelmezett, kiválasztott értéket. Mi azt adtuk meg, hogy a lehetséges értékek közül az elsővel legyen kitöltve, azaz a *stringList* script most ez a 2 sor:

```
List<String> s = stringList.getLabels()
s.get(0)
```

Vigyázzunk, hogy a visszatérési érték természetesen *String* típus. A fentieken kívül még beállíthatjuk a megjelenített (felkínált) értékek maximális számát is. Ez a gyakorlatban azt jelenti, hogy kisebb számra több karakter begépelése tudja majd jobban odajuttatni a program használóját, hogy a végén kiválaszthassa a neki szükséges értéket. Az asszinkron működést mindig pipáljuk be, ha egy lassú algoritmus állítja be az elérhető értékeket, azaz például egy webservice hívás a hálózaton keresztül. Nézzük mindezt meg futás közben (3.35. ábra)!

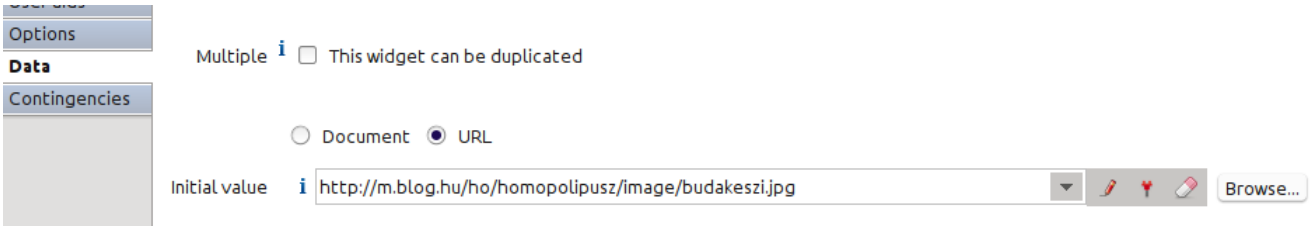

3.35. ábra. A *Test Step* Formja a Suggest Box használata közben

## Message Field Widget

Ez egy nagyon egyszerű widget, arra szolgál, hogy egy több soros, de nem szerkeszthető szöveget jelenítsen meg a form-on. Ennek megfelelően a konfigurálása is meglehetősen egyszerű, csak az *Initial value* részhez kell megadni az adatforrást, ami persze egy *String* típussal rendelkező dolog lehet.

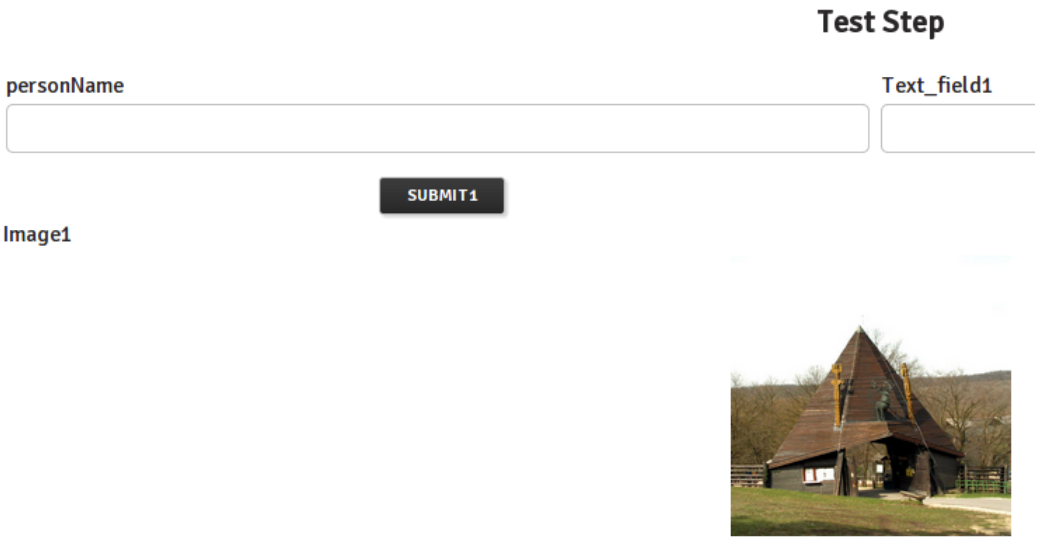
## Image Field Widget

A vezérlő feladata az, hogy egy URL (vagy dokumentum) mögött lévő képet megjelenítsen. Miután a vezérlőt egy cellába tettük a 3.36. ábra mutatja a lehetőségeinket.




3.36. ábra. Image Field widget konfiguráció

Természetesen az *Initial value* szövege itt is dinamikusan állítható elő, ez azt jelenti, hogy a képet is röptében választhatjuk ki. A 3.37. ábra futás közben mutatja a widget-et.



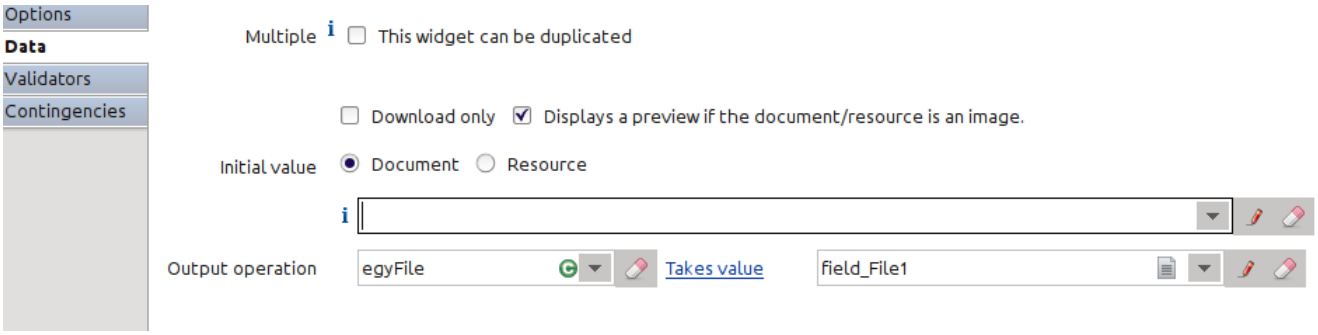

3.37. ábra. Image Field widget futás közben

Megjegyezzük, hogy a kép mérete nagyobb volt, de az *Appearance*→*Image* fülön azt 200x200 pixelre állítottuk.

## File Field Widget

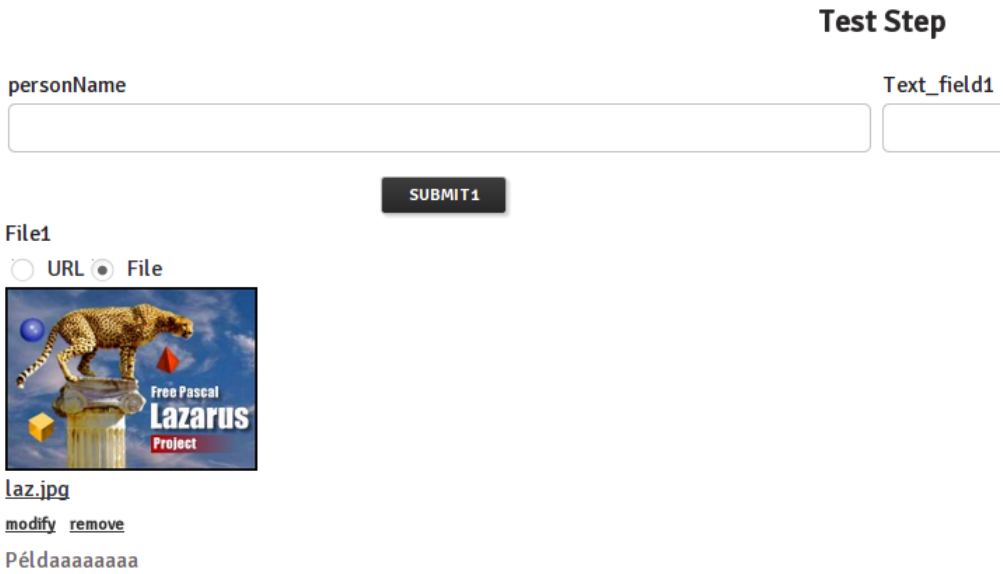
Ez a vezérlő 1 fájl feltöltésére (upload) vagy letöltésére (download) is alkalmas. A *field_File1* mezőváltozó típusa ez (az *egyFile* workflow változónak is ezt a típust adtuk):

```
org.bonitasoft.engine.bpm.document.DocumentValue
```




3.38. ábra. A File widget konfigurációja

A File vezérlő alapkonfigurációját a 3.38. ábra mutatja. A *Document* egy fájl, ami valahol van és kiválaszthatjuk feltöltésére (letöltésre). Az erőforrás (*Resource*) az alkalmazás projekt könyvtárában lévő fájl, amit be tudunk importálni, ha ezt a button-t választjuk ki. Ekkor egy Browse gomb is megjelenik, amivel az erőforrás betöltését elvégezhetjük. A 3.39. ábra már a *Test Step* TASK futási képernyőjét mutatja, miután feltöltöttünk egy képet.




3.39. ábra. Futás közben - Egy feltöltött fájl, aminek a tartalmát egyből látjuk is

Láthatjuk, hogy az *Output operation* résznél elmentjük az *egyFile* változóba a File widget mögötti mezőváltozót, ezért a következő, *Mutat* TASK formján (3.40. ábra) ezt meg is tudnánk ismét mutatni, azonban mi inkább csak MIME típusát jelenítettük meg.


3.40. ábra. A File vezérlőben lévő tartalom neve és MIME típusa

A *Text_Field1 Initial value* így lett beállítva:

```
egyFile.getFileName()+"MIME: "+egyFile.getMimeType()
```

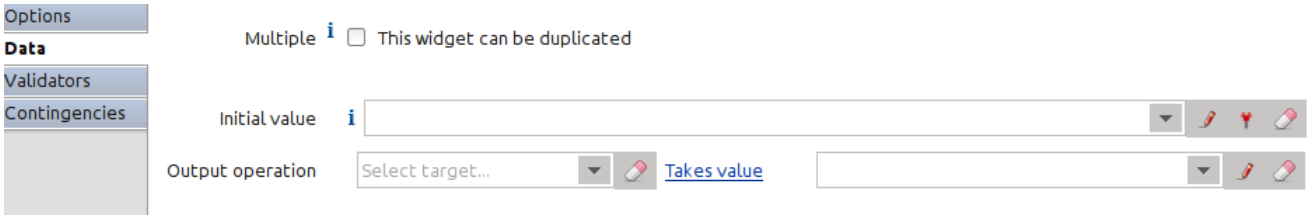
## Hidden Field Widget

Ez egy olyan Text Field mező, ami nem látszik a formon, de segítségével adatot lehet hordozni, amit a form változók is elérnek. Konfigurálása is a Text Field mezőével egyezik meg.

## Rich Text Field Widget

Egy HTML tartalom eltárolására és wyswyg szerkesztésére szolgáló komponens (3.41. ábra). A konfigurációját a 3.42. ábra mutatja, lényegében pont olyan, mint a többi text alapú widget beállítása. A 3.43. ábra a vezérlőt a *Test Step* TASK formján is megmutatja, futás közben.


3.41. ábra. A RichText field a whiteboardon

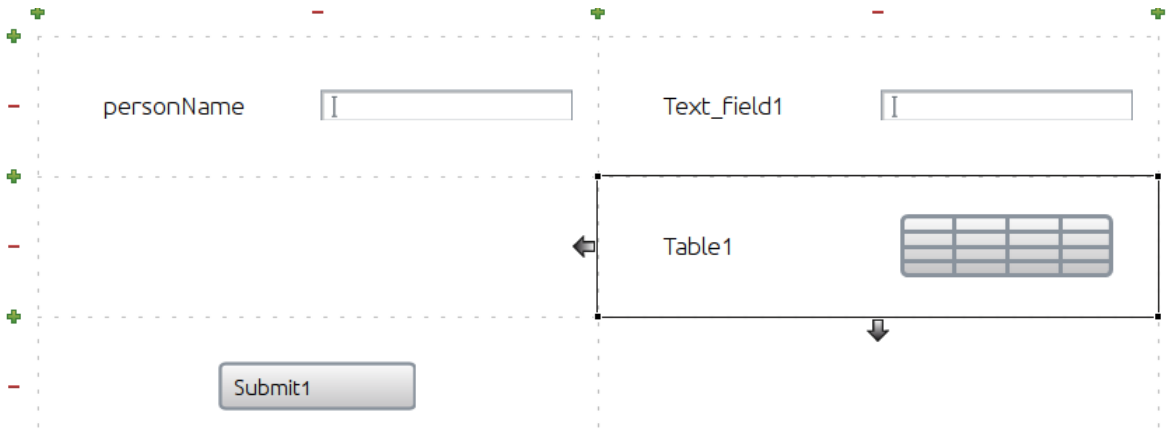



3.42. ábra. A Rich text vezérlő konfigurálása


3.43. ábra. A Rich text vezérlő futás közben

## Table Widget

Ez a widget szöveges adatok táblázatos elrendezésére szolgál (3.44. ábra).



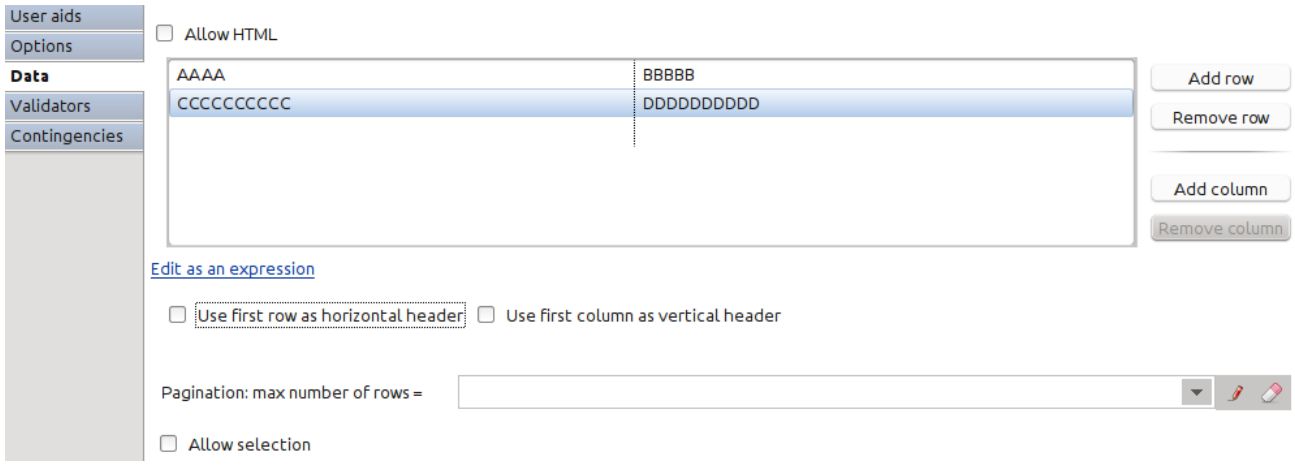

3.44. ábra. Table Field widget a tervező cellában

A vezérlőt kétféle módon tudjuk konfigurálni:

1. *Edit with table* nézetben → Ekkor a táblázat minden egyes mezőjét celláról cellára adhatjuk meg (3.45. ábra).

2. *Edit as an expression* nézetben → Ekkor egy *List<List<String>>* írja le a táblázat elemeit, azaz sorfolytonosan felsoroljuk az elemeit.

A 2 nézet a linkre kattintva váltogatható, de a táblázatunkat csak az egyik módon adhatjuk meg.

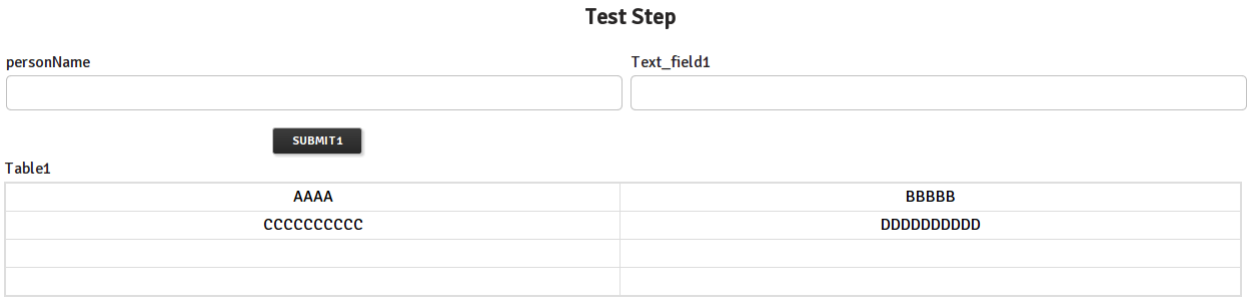



3.45. ábra. Egy táblázat konfigurálása tábla nézetben

A 3.45. ábra a vizuálisabb tervezést mutatja, láthatjuk, hogy a sorok és oszlopok számát mi adjuk meg úgy, hogy a jobb oldali nyomógombokat használjuk. Az egyes cellákhoz az értékek megadhatók, de az Edit... kiválasztásával (3.46. ábra) az ismert Groovy szerkesztő jön be, azaz minden cella egy tetszőleges kifejezés értéke lehet.

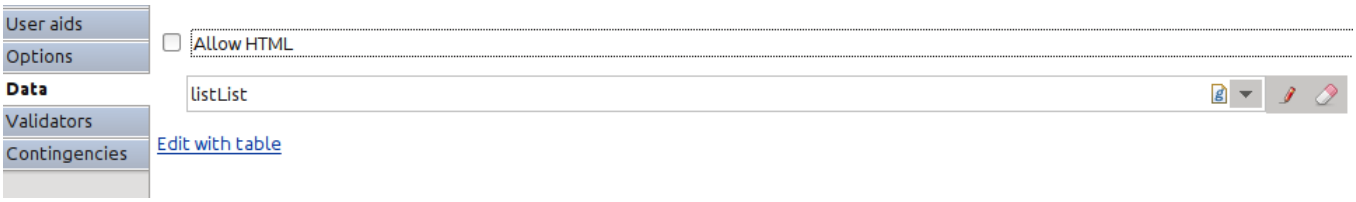

3.46. ábra. Minden cellához van egy *Edit...* lehetőség

Az 3.47. ábra a konfiguráció futási képét mutatja.



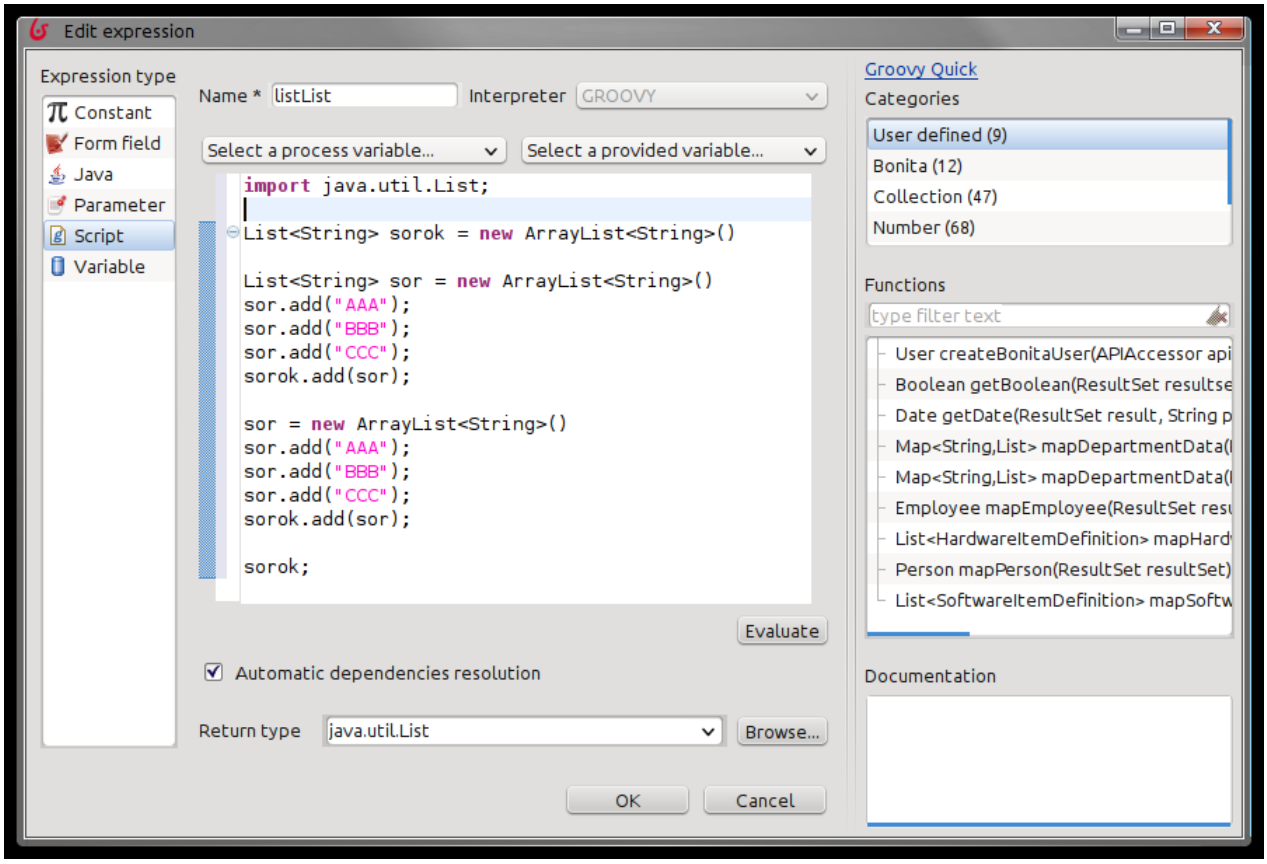

3.47. ábra. Tábla komponens futás közben (Table konfigurációval)

A másik táblázat megadási lehetőség a kifejezéssel való megadás, amit a már említett 3.48. ábra mutat.




3.48. ábra. Az expression alapú konfiguráció

A *listList* script mögötti tartalmat az 3.49. ábra tartalmazza. A listák listája egy 2 dimenziós mátrix, azaz egy táblázatot ír le.



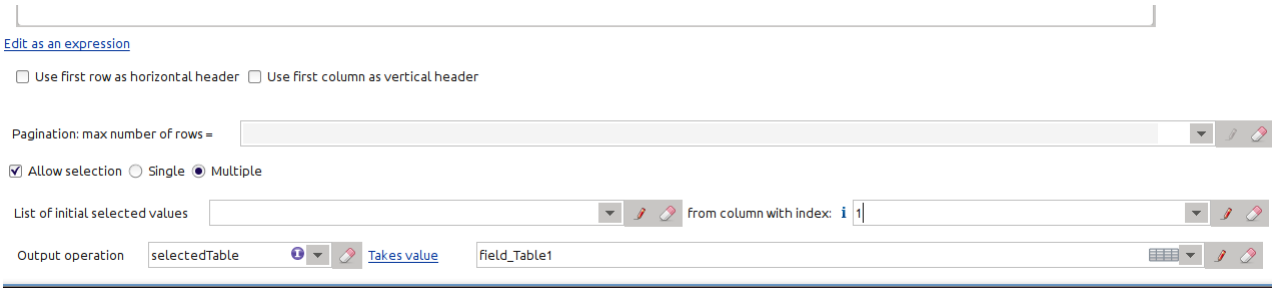

3.49. ábra. Az Expression alapú konfiguráció mögötti script

Az 3.48. és 3.49. ábrán lévő beállítások a tesztfutás során az 3.50. ábrán mutatott eredményt adják.


3.50. ábra. Tábla komponens futás közben (expression konfigurációval)

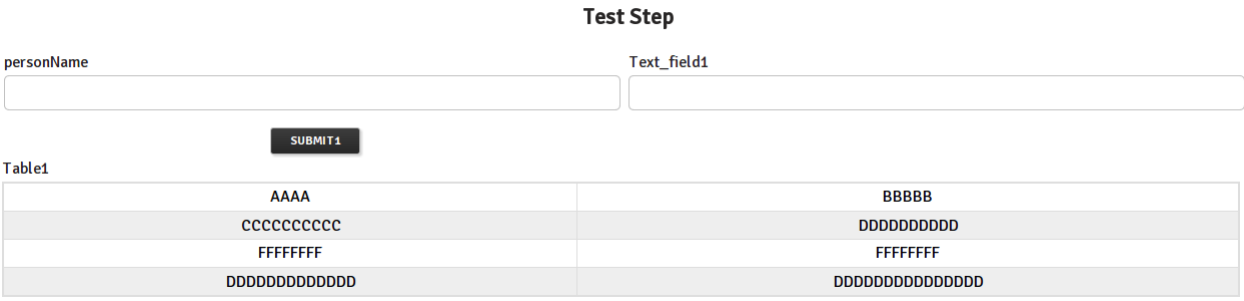
Megjegyezzük, hogy a tábla komponens csak megjelenítésre van tervezve, adatok megváltoztatására nem alkalmas. A *Pagination* mező nagyobb táblázat esetén azt mondja meg, hogy hány soronként biztosítson a vezérlő lapozást a táblázaton belül.

A komponensnek van még egy említésre méltó lehetősége, amikor a 3.51. ábrán is látható *Allow selection* checkbox-ot bekapcsoljuk. Ez mindkét táblázat konfigurációs módban a rendelkezésünkre áll.




3.51. ábra. Amikor az Allow selection lehetőséget bekapcsoljuk, akkor ezeket is megadhatjuk

Ez a mód engedélyezi, hogy a futó formban lévő táblából kiválasszunk sorokat (3.52. ábra).



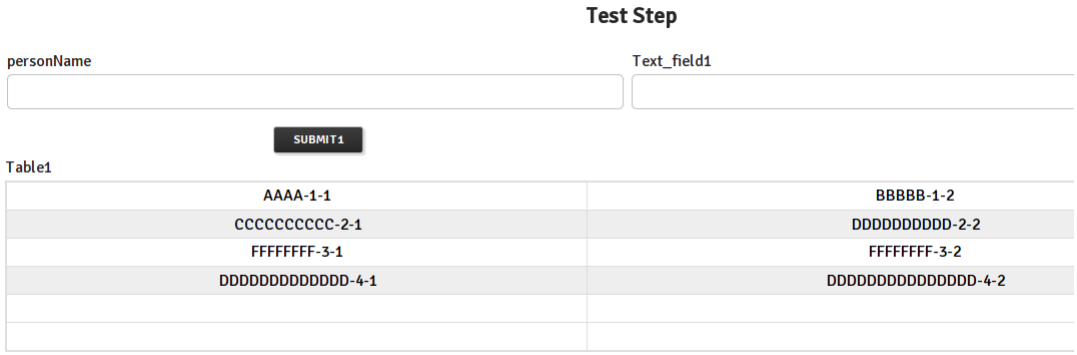

3.52. ábra. A 2. és 4. sort választottuk ki egérkattintással

A kiválasztott sorok kulcsoszlop értékei elmenthetőek. Az oszlopok indexelése 0-tól indul és a *from column with index* helyen adhatjuk, meg, hogy ez melyik legyen. Az 3.51. ábrán ez most 1, azaz a 2. oszlop. A *field_ Table1* egy *java.util.List*, ennek megfelelően a *selectedTable* workflow

változó is az. A *SUBMIT1* megnyomása utána ez utóbbi kerül a megadott oszlop (most 1.) azon sorai, amelyek ki vannak választva. A *Mutat* TASK formjára felvettünk egy *Message* komponenst megmutatni a kiválasztott elemeket. Az *Initial value*-t erre a vezérlőre ez a script állítja be:

```
String vissza = selectedTable.toString()
vissza += " / " + selectedTable.size()
vissza
```

Ennyi előzmény után nézzük meg újra a futási képernyőt, ahol a táblázat tartalmát is változtattuk egy kicsit, hogy mindez jobban látszódjon. Az 3.53. ábra a *Test Step* formja, miután kiválasztottunk 2 sort. A 3.54. ábra jobb oldala a fenti script alapján feltöltött Message komponenst mutatja a *Mutat* TASK form-ján. Láthatjuk, hogy a *selectedTabla* a kiválasztott elemek 2 elemű listáját tartalmazza.




3.53. ábra. A 2. és 4. sort választottuk ki


3.54. ábra. A kiválasztott sorok 2. oszlopa adta a listát

## Editable Gried Widget

Első lépésként egy ilyen komponenst húzzunk a whiteboard-ra (3.55. ábra).

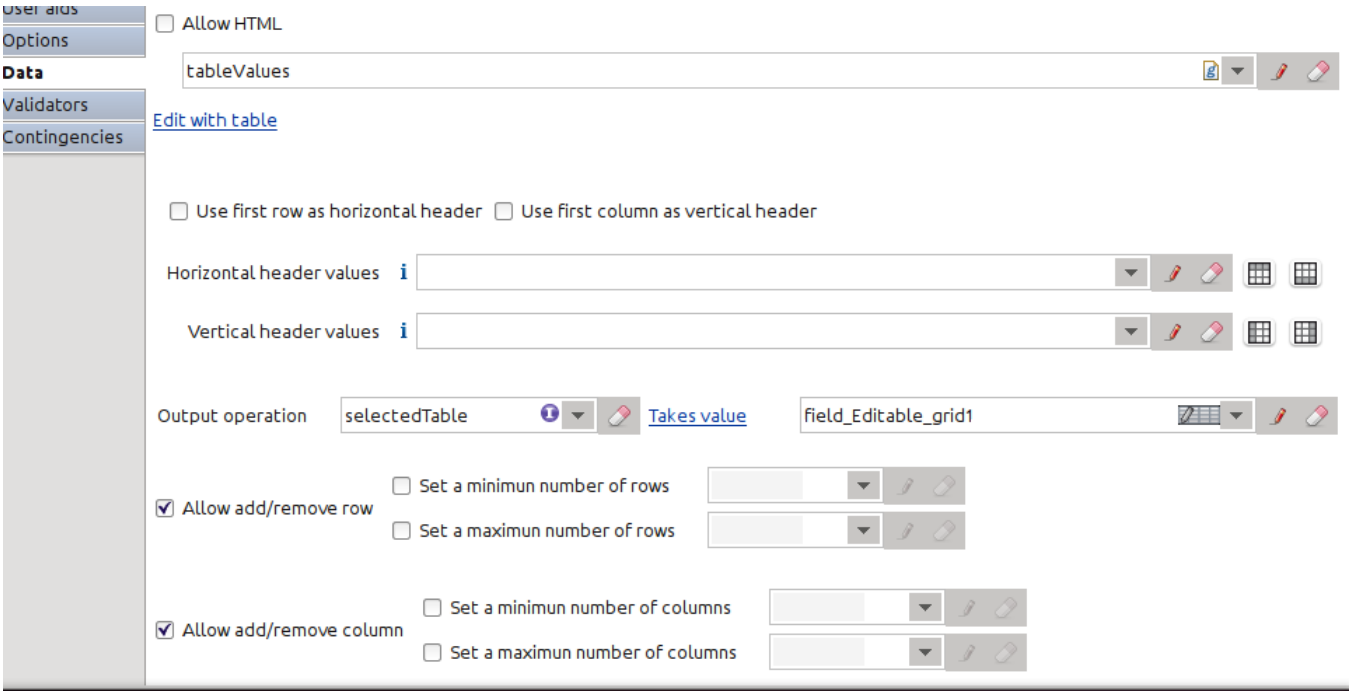

3.55. ábra. Ezzel a komponenssel a táblázatot szerkeszthetjük is

Az Editable Grid lényegében egy olyan tábla komponens (lásd az előző pontot), aminek az elemei szerkeszthetőek a Form-on. Ennek illusztrálására a már ismert *selectedTable* (típusa: *java.util.List*) workflow változót használjuk. Ez a komponens is használható ebben a 2 konfigurálási lehetőséggel:

1. *Edit with table* nézetben → Ekkor a táblázat minden egyes mezőjét celláról cellára adhatjuk meg (3.45. ábra).

2. *Edit as an expression* nézetben → Ekkor egy *List<List<String>>* írja le a táblázat elemeit, azaz sorfolytonosan felsoroljuk az elemeit.

A következőkben az expression módra adunk egy példát, a tábla mód teljesen hasonló. A konfigurálási lehetőségeket az 3.56. ábra mutatja.




3.56. ábra. Az Editable Grid konfigurálási lehetőségei (expression módban)

A *tableValues* egy script, ami pont ugyanaz, amit az 3.49. ábra is mutat. Megadhatnánk egy *List<String>* érték használatával a horizontális és vertikális fejléceket is (mi most nem tettük). Nemsokára látjuk, hogy ez a komponens új sorok és oszlopok hozzáadására és törlésére is képes, emiatt általában szükséges lehet ezen értékek megadása is:

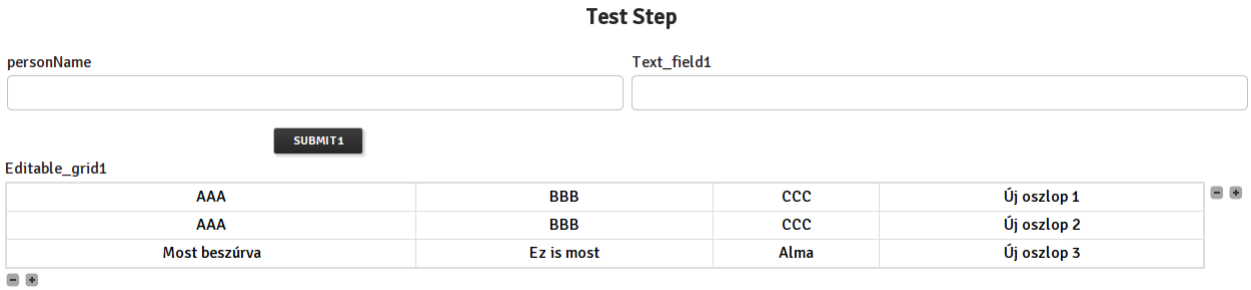
- *Allow add/remove row* → Akarok-e új sorokat hozzáfűzni. Ha igen, akkor mennyi ennek a minimum és maximum száma.

- *Allow add/remove column* → Akarok-e új oszlopokat hozzáfűzni. Ha igen, akkor mennyi ennek a minimum és maximum száma.

Az *Output operation* a szokásos, megadjuk, hogy a táblát egy form action után milyen változóba (esetünkben a *selectedTable*) akarjuk menteni. A mentés sikerességét a Mutat TASK formjára helyezett Table widget segítségével ellenőrizzük. A 3.57. ábra a futtatáskor megjelenő induló állapotot mutatja.

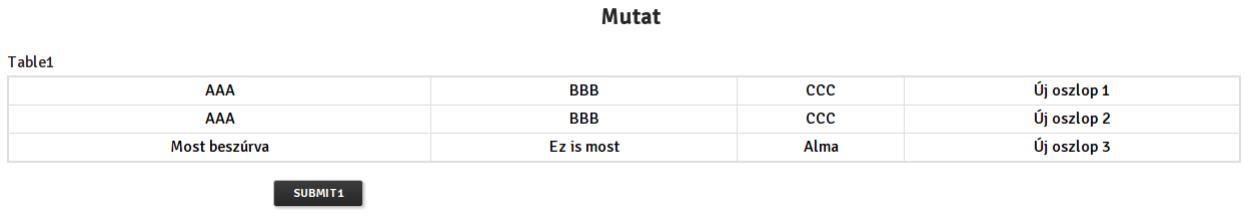

3.57. ábra. A *Test Step* TASK formja induláskor

Vegyük észre a sorok és oszlopok kiegészítésére szolgáló + és – ikonokat!




3.58. ábra. A *Test Step* TASK formja editálás után

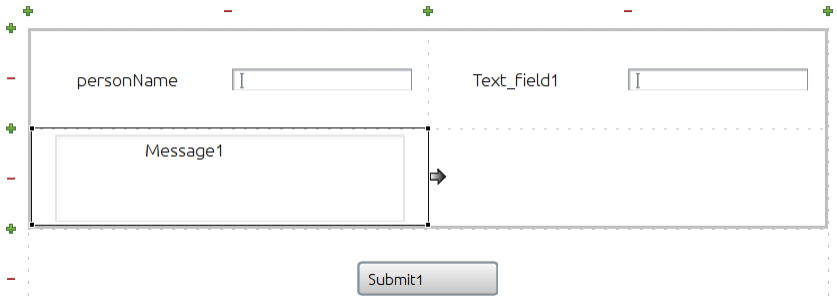
A 3.58. ábra már a szerkesztés utáni állapotot mutatja. Beszúrtunk 1 új sort és 1 új oszlopot. A cellák hasonlóan szerkeszthetőek, ahogy azt az excelben is megszoktuk. A sikerességet az is bizonyítja, hogy a *selectedTable* változó tartalma valóban a megváltozott tartalmat mutatja a 3.59. ábrán, ami egy Table komponens, mögötte ezzel a változóval.




3.59. ábra. A táblát megmutatjuk a *Mutat* TASK formján

## Group Widget

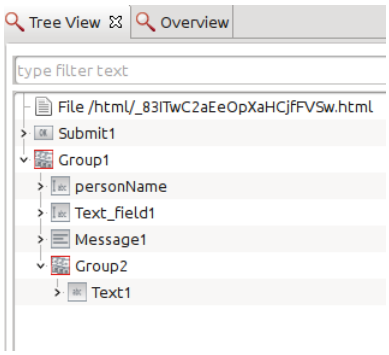
A Group widget feladata az, hogy összefogjon vezérlőket (3.60. ábra), így azokat együtt is lehet kezelni, külön stílust is kaphatnak.




3.60. ábra. A Group widget összefog több beágyazott widget-et

Mindezt futás közben a 3.61. ábra mutatja.


3.61. ábra. A Group widget összefog több beágyazott widget-et



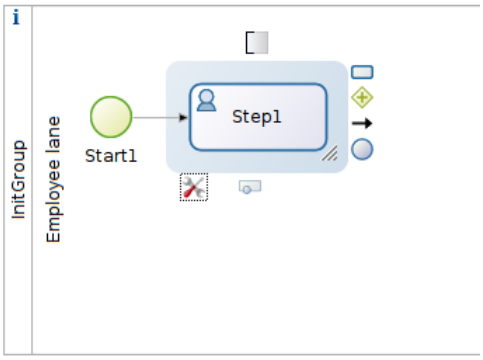

3.62. ábra.
A Group-ba egy másik Group is beágyazható, amit a Tree View segítségével érhetünk el a legkönnyebben.

Adatszerkezet oldaláról nézve, a Group-ot (esetünkben *field_Group1*) leíró mezőváltozó egy *java.util.Map* szerkezet, ahol:

- key → beágyazott widget ID (a neve, amit megadtunk. Ez egy *String*)
- value → beágyazott widget text értéke

A következőkben 2 gyakorlati példát nézünk meg, hogy a group-ok használatát jobban megértsük.

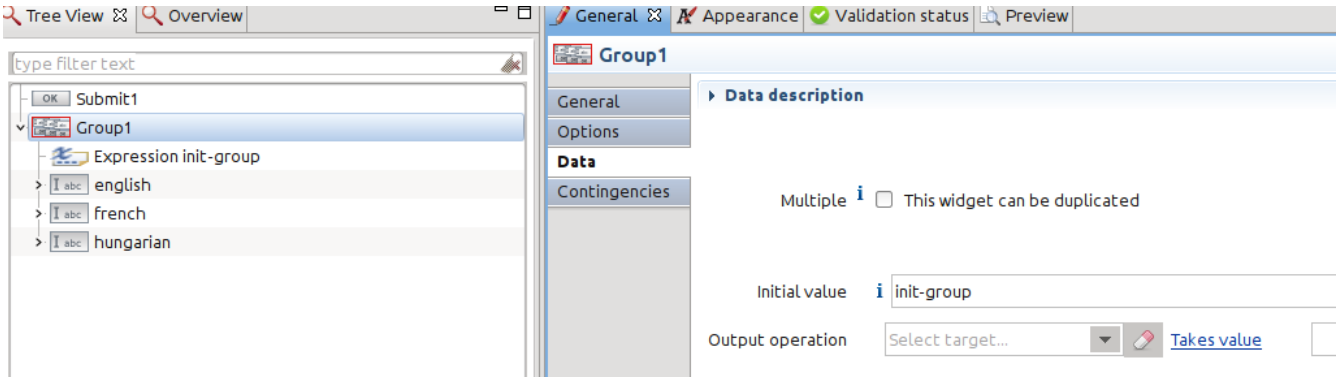
### A group elemeinek írása és olvasása




3.63. ábra. Teszt process


3.64. ábra. A Step1 formja

A 3.63. ábra 1 lépéses folyamatához készítünk egy Form-ot, amiben egy group van. A Form a 3.64. ábra szerint néz ki, a *Step1*-re 1 db group vezérlője van, 3 db Text widget-tel. Az group inicializálásán keresztül mutatjuk meg, hogy milyen módon érjük le a Java *Map* adatszerkezetét, amit fel is töltünk értékekkel.




3.65. ábra. A group példa konfigurálása

A 3.65 ábrán láthatjuk, hogy az *init-group* Groovy script csinálja a kezdeti értékadást, aminek ez a kódja:

```
return ["english":"hello", "french":"Coucou", "hungarian":"Szia"];
```

Nagyon fontos, hogy a visszatérési érték *java.lang.Object*, hiszen itt egy *Map* objektumról van szó. A futási képet a 3.66. ábra mutatja.

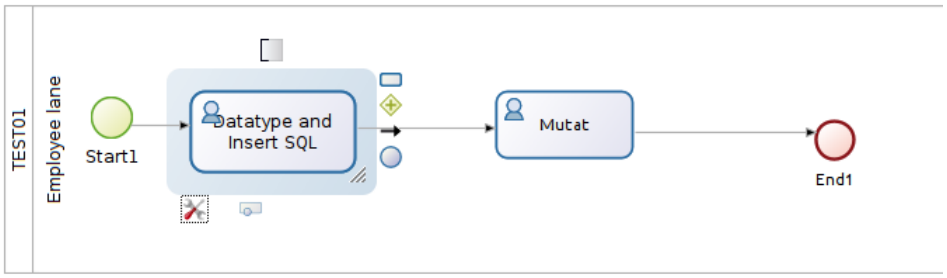

3.66. ábra. A csoportban megjelentek a mezőértékek

Megjegyzésként itt mutatunk néhány Groovy parancssor, ami a Map adatszerkezetet használja:

```
def map = [name:"Gromit", likes:"cheese",
map.get("id") == 1234
map["name"] == "Gromit"
map['id'] == 1234
def emptyMap = [:]
emptyMap.size() == 0
emptyMap.put("foo", 5)
emptyMap.size() == 1
emptyMap.get("foo") == 5
```

### A multiple group használata

A 3.67. ábra a példa BPMN ábráját mutatja.




3.67. ábra. A multiple group példa munkafolymata

Tekintsük a 3-1. Programlistán mutatott *TestGroupHolder* Java class-t, amit majd le kell fordítani és *jar*-ba tenni, hogy a Bonita projektünkhöz integrálhassuk (*dependencies*). Az osztály célja, hogy tárolja a *multiple* Group mezőket, amire a *groupItems* adattagja szolgál.

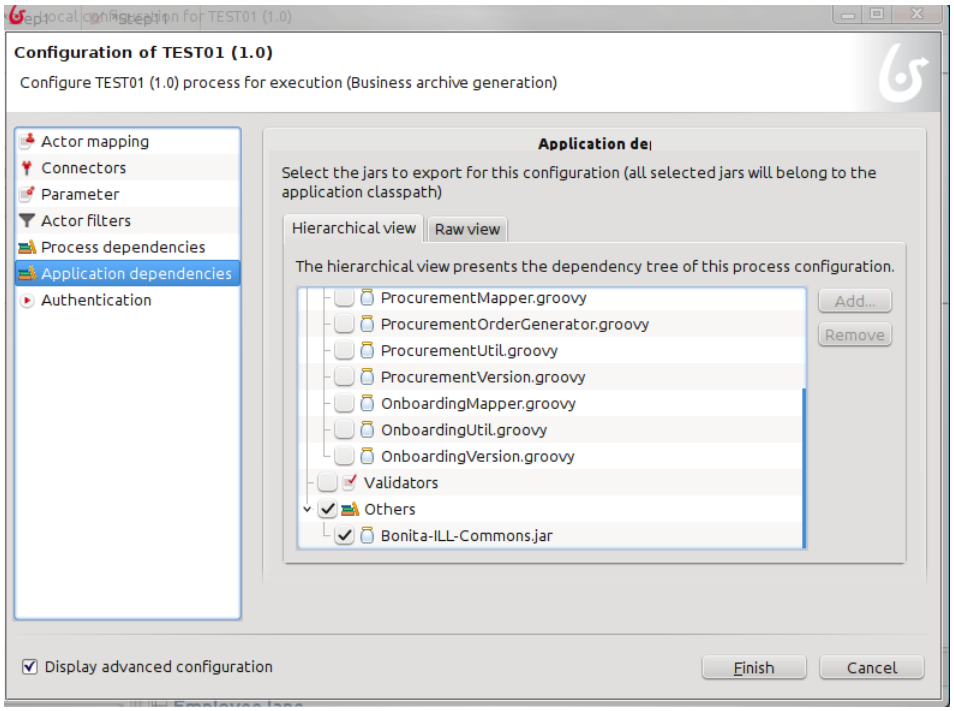
3-1. Programlista: TestGroupHolder.java

```
1  package org.anura.bonita.commons;
2
3  import java.io.Serializable;
4  import java.util.ArrayList;
5  import java.util.Date;
6  import java.util.HashMap;
7  import java.util.List;
8  import java.util.Map;
9
10 public class TestGroupHolder implements Serializable
11 {
12         public List<Map> groupItems = new ArrayList<Map>();
13
14         public TestGroupHolder()
15         {
16                 Map i = new HashMap();
17                 i.put("aa", 32);
18                 i.put("bb", "Alma");
19                 i.put("cc", new Date());
20                 groupItems.add(i);
21
22                 i = new HashMap();
23                 i.put("aa", 55);
24                 i.put("bb", "Körte");
25                 i.put("cc", new Date());
26                 groupItems.add(i);
27
28                 i = new HashMap();
29                 i.put("aa", 100);
30                 i.put("bb", "Szilva");
31                 i.put("cc", new Date());
32                 groupItems.add(i);
33         }
34
35         /**
36          *
37          * @param args
38          */
39         public static void main(String[] args)
40         {
41         }
42 }
```

A *test01* workflow változó *TestGroupHolder* típusú lett (3.68. ábra).

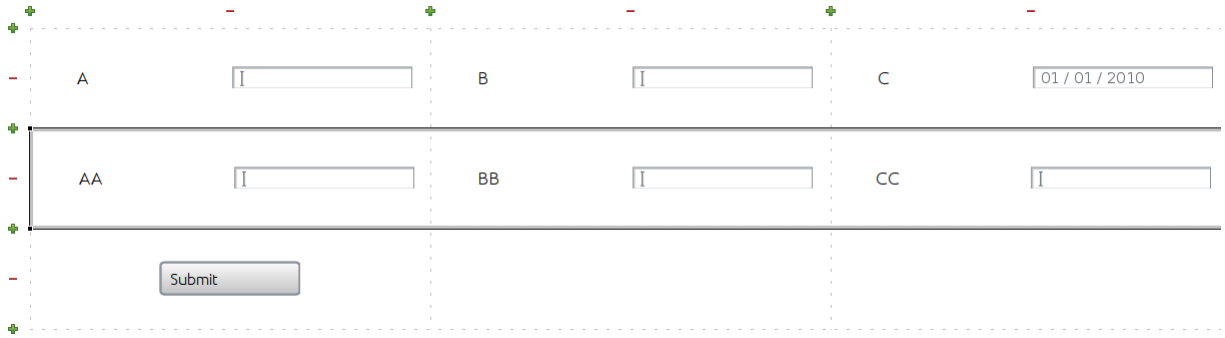

3.68. ábra. A *test01* workflow szintű változó felvétele

Ahhoz, hogy ezt az osztályt láthassuk, az őt tartalmazó *jar* fájlt importálni kell, amit a *Development* → *Manage jars* pontnál lehet megtenni. Ezután a *jar*-t a *Configure cool bar* button-nal a Process és az Application dependenciához is be kell pipálni, ugyanis a process-ben és a FORM-on is hivatkozunk rá (3.69. ábra).



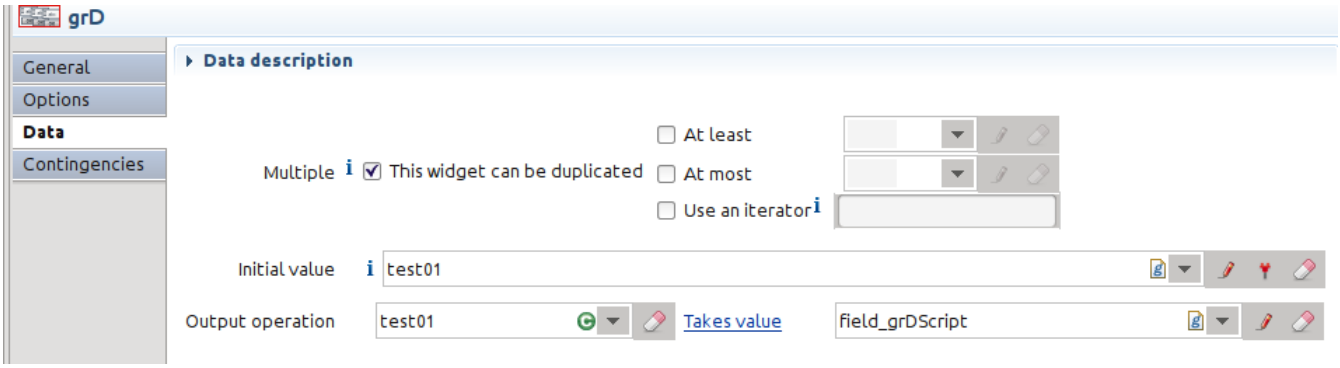

3.69. ábra. Process és Application dependenciák bepipálása

Fontos megérteni, hogy az így előállított class használata sokkal rugalmasabb, mint a *Development* → *Data types* használatával. Ott ugyanis nem tudunk metódust és konstruktort csinálni az osztályhoz. Az első step form-ját a 3.70. ábra mutatja.

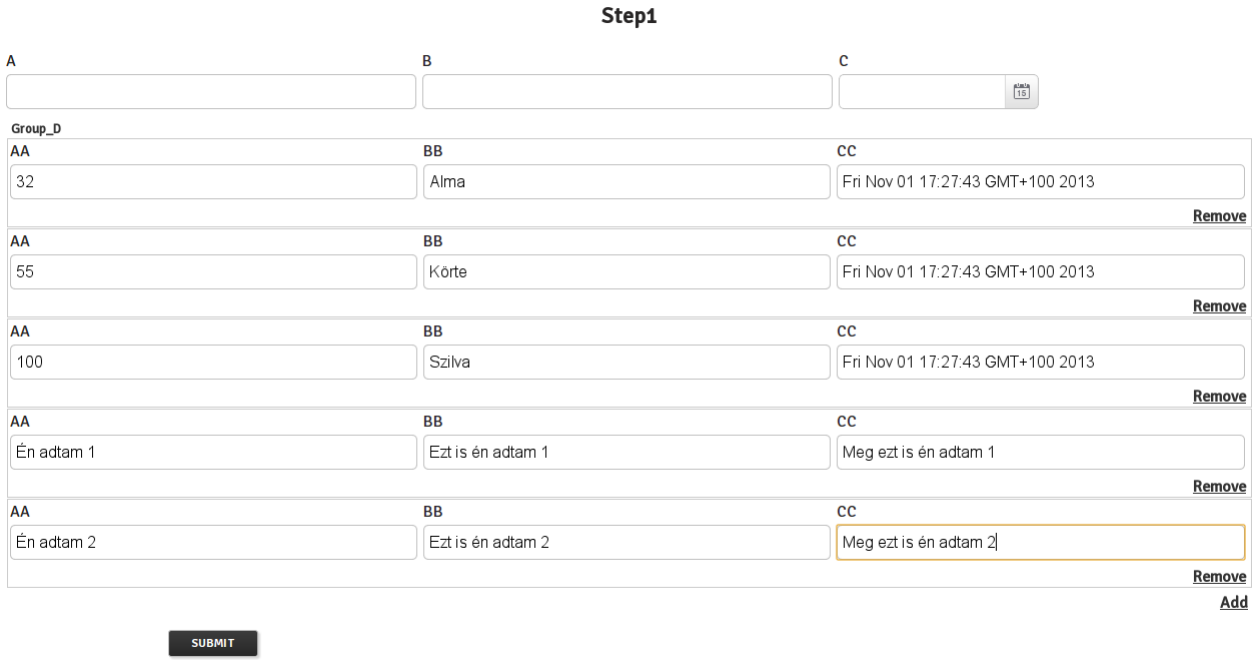



3.70. ábra. Az első step form-ja, kiemelve a group-ot

A FORM *grD* nevű group-jának a konfigurációját a 3.71. ábra mutatja, látható, hogy azt *Multiple* módra állítottuk be.




3.71. ábra. A form *grD* nevű csoportjának konfigurációja




3.72. ábra. A group-hoz még 2 sort vittünk be

Az Initial value kódja (a változó létrejöttekor már tartalmaz értéket, ahogy azt a 3-1. Programlista is mutatja):

```
org.anura.bonita.commons.TestGroupHolder h = test01;
return h.groupItems;
```

Az *Output operation* kódja, ami a változtatott értéket visszateszi *test01*-be:

```
import java.util.logging.Logger;
import org.anura.bonita.commons.TestGroupHolder;

Logger log = Logger.getLogger("bonita");
org.anura.bonita.commons.TestGroupHolder h = test01;
```

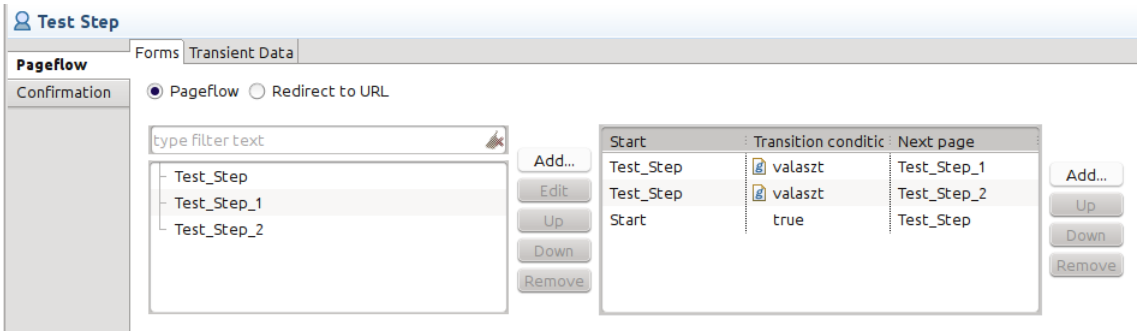
```
h.groupItems = field_grD;

return test01;
```

A 2. form az első másolata. Célja bemutatni, hogy kibővült a group. Nézzük meg mindezt futás közben (3.72. ábra)!

A kezdeti 3 sorhoz az *Add* link segítségével 2 újat adtunk, majd a Submit után a következő step formja már 5 sort hoz, azaz jól működik a program.

## Screen Flow készítés

A gyakorlatban létező Form-okat sok esetben úgy érdemes elkészíteni, mint egy Windows varázslót, azaz Form-ok szekvenciájaként. A *Previous* és *Next* gombtípus egyetlen feladata, hogy 2 Form között az átmenetet elindítsa egy Screen Flow (vagy más néven Form Flow) esetén. Ezt megérteni szintén egy példán a legegyszerűbb. A *Test Step* TASK-hoz készítünk 3 Formot, amit az *Application* → *Pageflow* fül segítségével tehetünk meg (3.73. ábra).




3.73. ábra. A 3 Form és hozzá megadva a transition szabályrendszer

Alapértelmezésben a *Next* és *Previous* gombokkal abban a sorrendben járhatjuk be a Form-okat, ahogy az a *Pageflow* listán látható. Ettől persze sokszor el kell térnünk és még az sem biztos, hogy mindegyik Form-ot be kell járnunk, így a jobb oldali beállító panelben adhatjuk meg, hogy mely Form-ról milyen logikai feltétel teljesülése esetén melyik másik Form-ra lépjünk. Mindezt általában dinamikusan kiértékelve. Az egyes formjaink tervezési képét a 3.74., 3.75. és 3.76. ábrák mutatják.

A *Test_Step* Form választó mezőjébe írt érték szerint a *Next* vagy a *Test_Step_1* Form vagy a *Test_Step_2* Form-ra visz bennünket. Ez a 3.73. ábra *Transition condition* kiértékelt logikai feltételétől függ, amelyek ezek (mindegyik egy script, ahogy látható is).

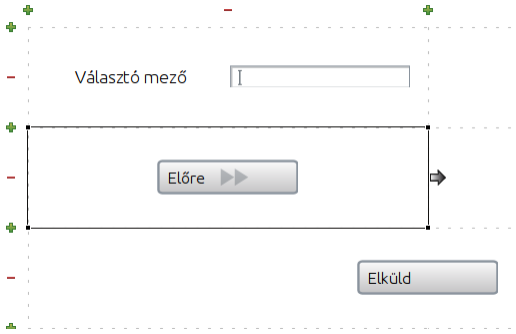
*Test_Step* → *Test_Step_1* esetén:

```
boolean b = field_Text_field1.equals("1")
b
```

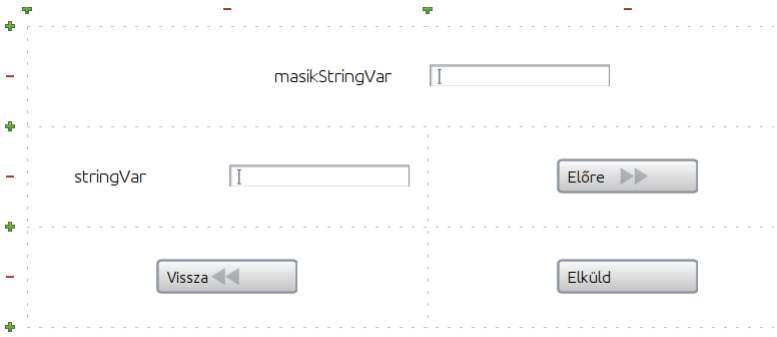
*Test_Step* → *Test_Step_2* esetén:

```
boolean b = field_Text_field1.equals("2")
b
```

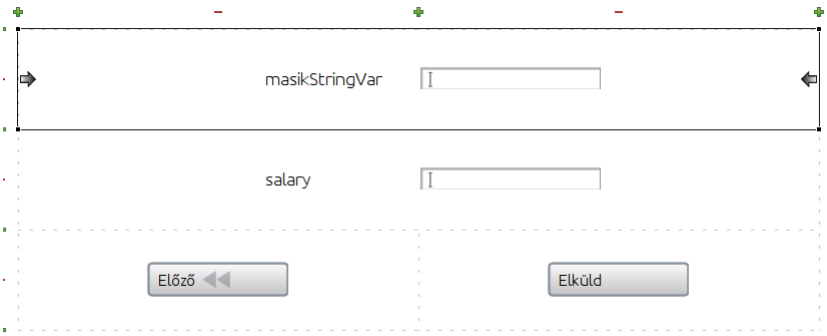
Amennyiben a választó mezőbe 1 vagy 2 értéket írjuk, úgy a *Next* gombra a megfelelő Form-ra megyünk át. A SUMBIT gomb bármikor megnyomható (perszer a Form kitöltött értékeinek valid-nak kell lenniük). A feltételvizsgálatoknál azért kell mezőváltozót használni, mert a submit gomb megnyomásáig csak ezek frissülnek, azaz a feltételekben bekövetkezett változásokat is csak ezekkel tudjuk kifejezni.




3.74. ábra. A Test_Step Form



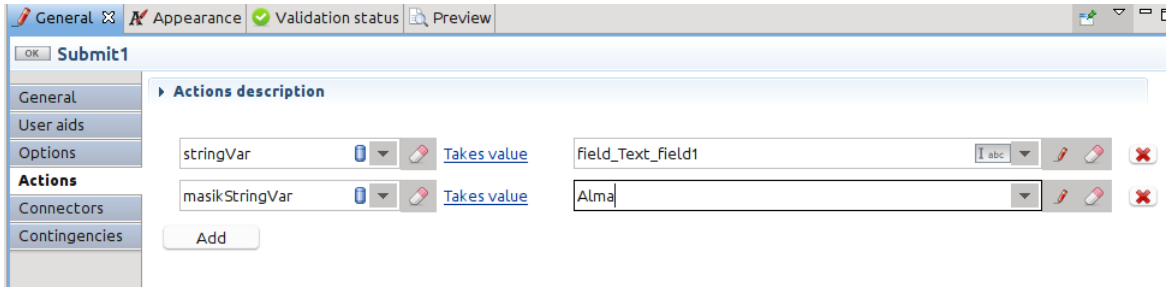

3.75. ábra. A Test_Step_1 Form




3.76. ábra. A Test_Step_2 Form

## Submit Button Widget

A Submit gomb alapvető feladata, hogy egy szerver oldali Action-t váltson ki, aminek 2 hatása lesz. Egyrészt a mezőváltozók értékei bekerülnek a workflow változókba, másrészt a következő TASK-ra (Step-re) léptet a workflow design-ban (BPMN) definiált feltételek alapján. A submit képes ezenfelül akciókat és konnektorokat is futtatni (3.77. ábra), ami azért jó, mert fontos üzleti algoritmusokat és értékbeállításokat, mentéseket tehetünk erre a pontra. Az akciók lehetővé teszik, hogy scripteket és konnektorokat futtassunk, amikkel update-elni lehet a munkafolyamat külső és belső környezetét is.

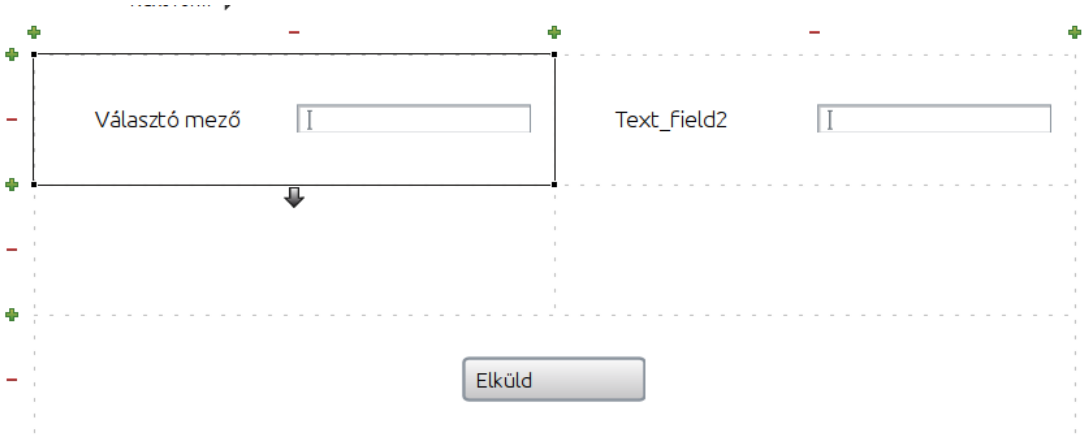



3.77. ábra. A submit gomb képes akciókat és konnektorokat futtatni

1. *Actions* → Amikor egy submit gombot megnyomunk, ahhoz tetszőleges számú akció rendelhető. Ez azt jelenti, hogy a szokásos kifejezés szerkesztőben algoritmussal (Groovy Script) egy értéket előállíthatunk, amit a workflow valamely változója kap meg értékként az action feldolgozása során. A 3.77. ábrán például az *Alma String* konstans el fog tárolódni a *masikStringVar* workflow változóba.

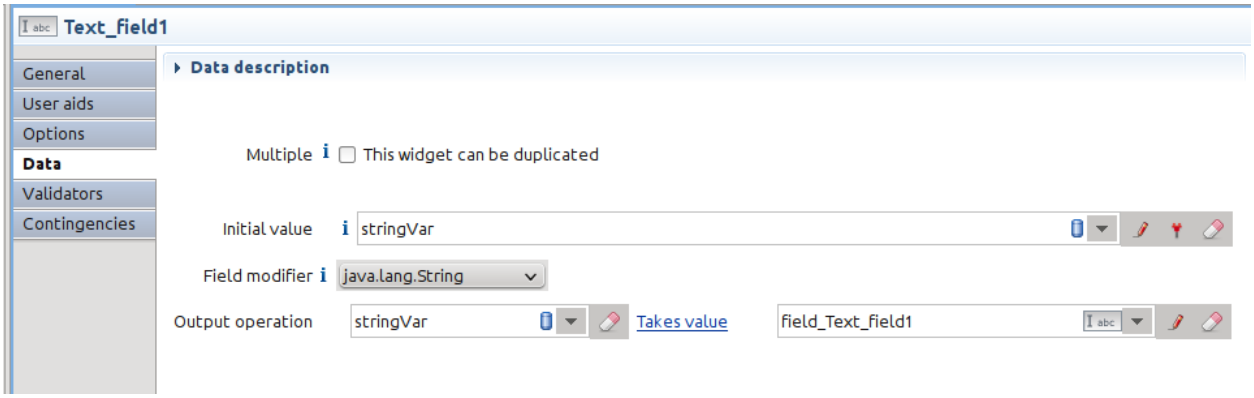
2. *Connectors* → A célja hasonló az akcióhoz, de itt egy Bonita konnektor fut le, kommunikálva a távoli rendszerekkel adat megszerzési és/vagy elküldési céllal.

Most röviden tekintsük át azt, hogy milyen módon tudnánk egy Groovy script-et futtatni úgy, hogy nem lépünk ki a TASK-ból. Ennek bemutatására a 3.78. ábra Form-ja szolgál.



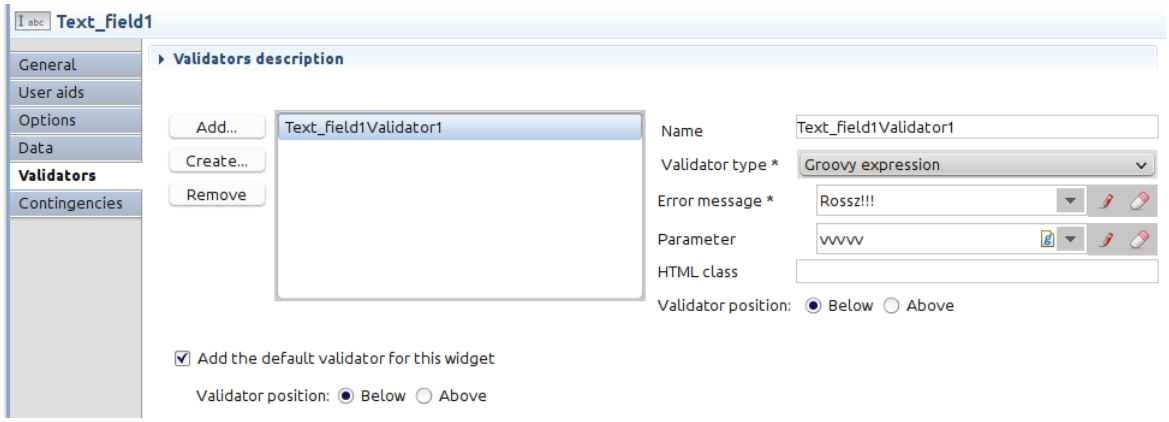

3.78. ábra. Egy script lefuttatása - nem lépünk ki a TASK-ból

A választó mező változó hozzárendelését a 3.79. ábra mutatja, azaz a *field_ Text_ field1* mezőváltozó a *stringVar*-ba kerül egy action lefutása során.




3.79. ábra. A választó mező adat hozzárendelése

Ezután a *Text_ field1* widget-hez (emögött van a *field_ Text_ field1* mezőváltozó) vegyünk fel egy validátort, aminek az egyik lépését láthatjuk a 3.80. ábrán.




3.80. ábra. A választó mező validátora, ami *S* értékre enged tovább a következő Step-re

A *Parameter* nevű mező tartalmazza ezt a Groovy Scriptet az ellenőrzéshez, ez fut le:

```
Boolean b = field_Text_field1.equals("S")
b
```

A Form működése a következő 2 lehetőséget adja a Submit (*Elküld* a címkéjének a neve) gomb megnyomására.

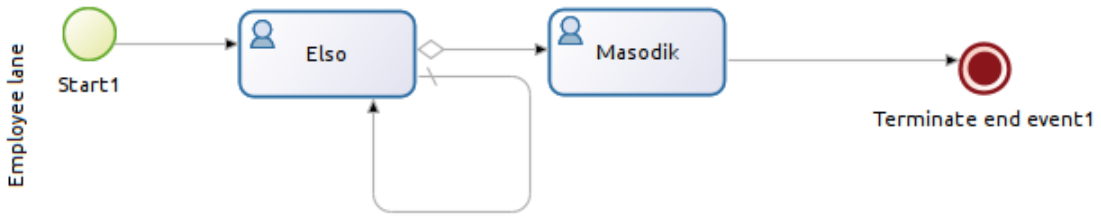
1. A text mező értéke == *S* → Ekkor a Submit kivisz a TASK-ból

2. A text mező értéke *nem S* → Ekkor a Submit nem visz ki a TASK-ból, de a script valami hasznosat tud csinálni, például ment.

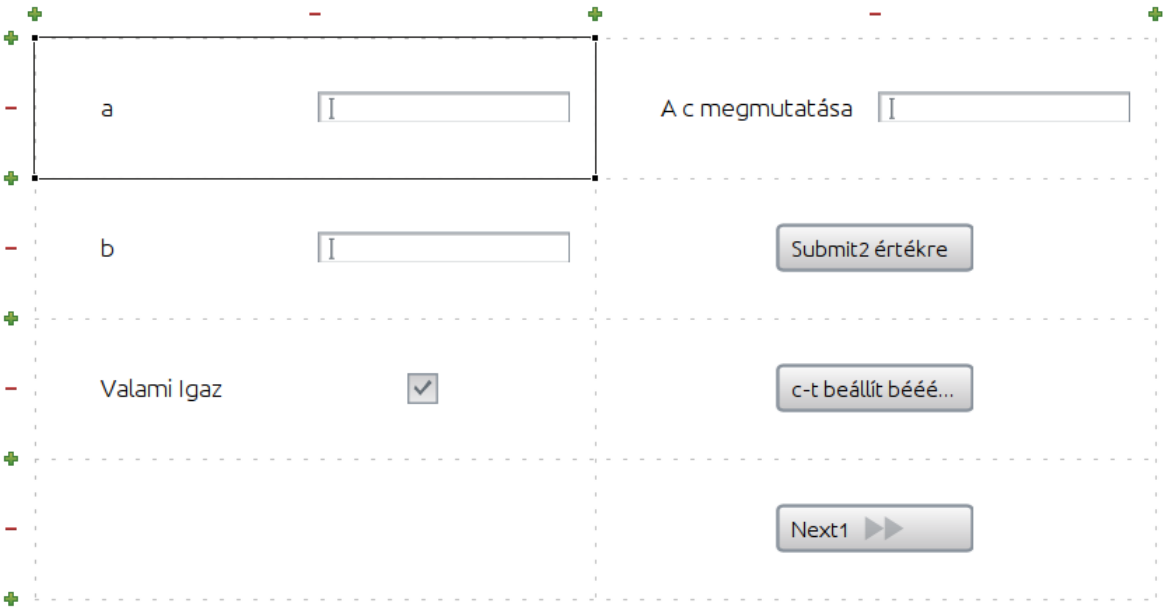
Több submit gombot is kitehetünk, így az Action és Konnektor halmaz más és más lehet mindegyikre, azok eltérő módon tudnak működni. Tipikus példa az Elfogadás vagy Elutasítás vagy További INFO kérés submit gombok kirakása.

## A Contingencies (eshetőségek) használata (AJAX lehetőségek)

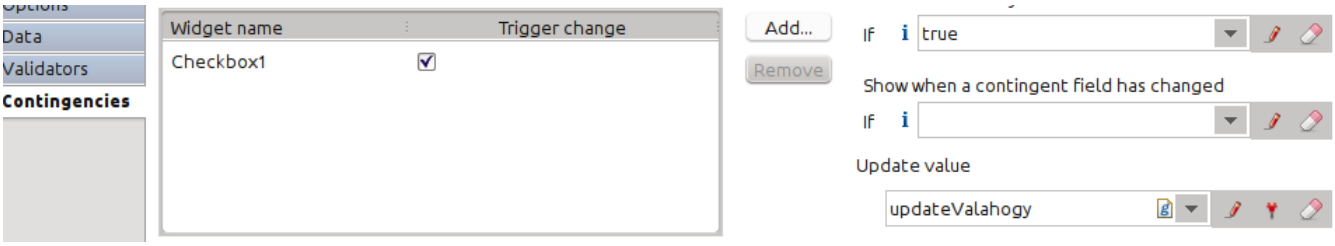
Minden Form vezérlőre beállítható a *Contingencies*, ami AJAX-os működéssel segíti a form programozóját. Amikor ráállunk egy vezérlőre, akkor a *General→Contingencies* fülről vehetjük igénybe ezt a szolgáltatást. Megadhatjuk, hogy mely mezők változása esetén kell az éppen design alatt lévő mezőt is frissíteni majd, amikor fut a form, de még nem nyomtunk meg egyetlen submit gombot sem. A példához a 3.81. ábra *Elso* TASK-jának a formját használtuk, amit a 3.82. ábra mutat.




3.81. ábra. A BPMN design



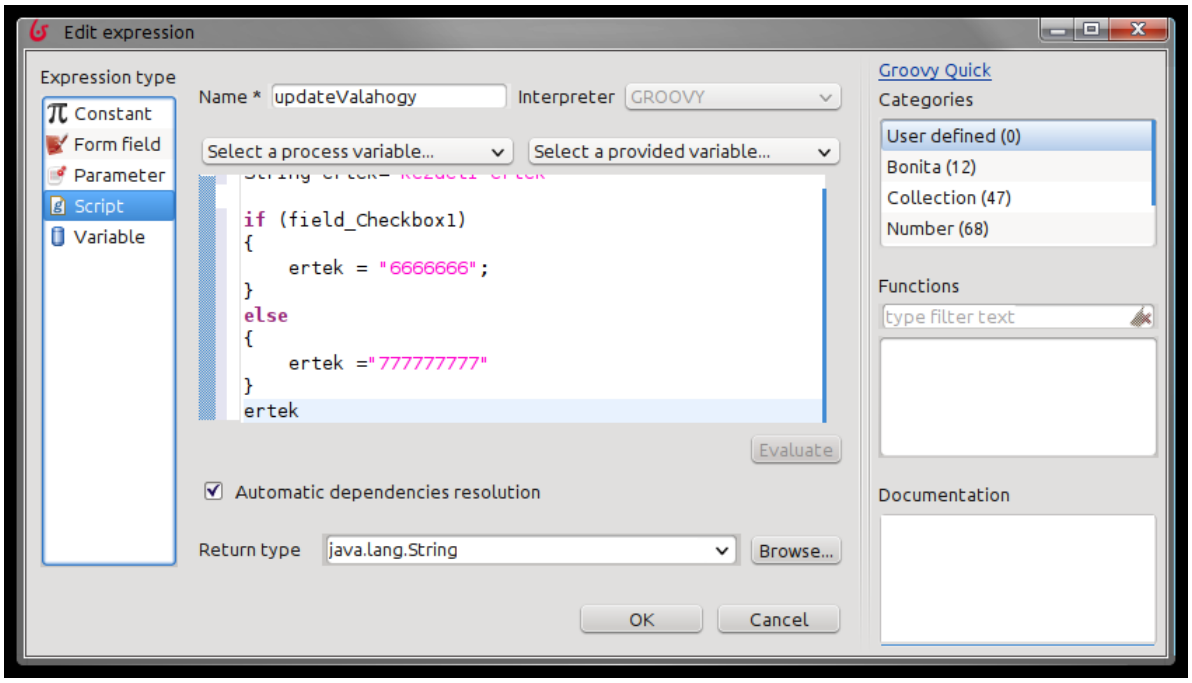

3.82. ábra. Az *Elso* TASK form-ja




3.83. ábra. Az *a* mező Contingencies beállítás

A feladat legyen ez:

- Amikor a checkbox=false, akkor a *c* mező nem látszik és az *a=7777...* legyen
- Amikor a checkbox=true, akkor a *c* mező felbukkan, az *a=6666...* legyen

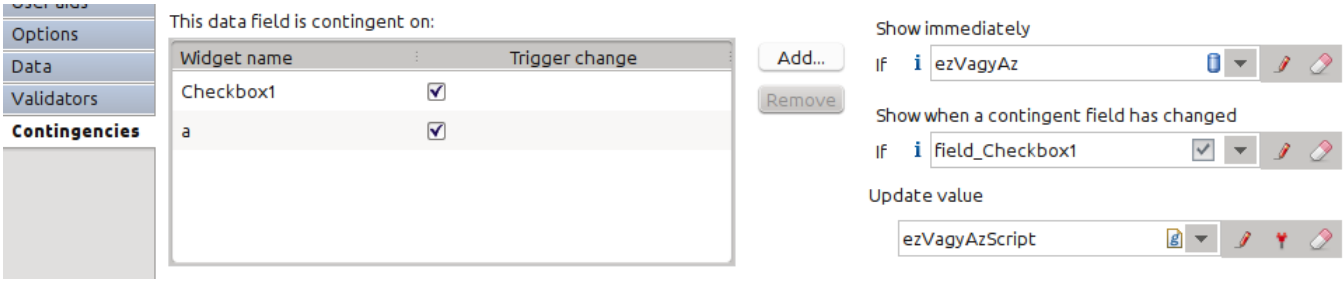



3.84. ábra. Az *updateValahogy* script

Mindez azt jelenti, hogy az *a* és *c* widgetekre (a 2 felső, bal oldali Text Edit) *Contingencies* beállításokat kell elvégeznünk, mert azok fognak változni AJAX-os módon. Nézzük előbb az *a* mezőt! Álljunk rá és válasszuk ki a *Contingencies* fület, ahogy azt a 3.83. ábra is mutatja. Tekintettel arra, hogy a változás egyetlen forrása most csak a *Checkbox1* mező megváltozása, így elég azt betenni okként a 3.83. ábra bal oldali widget listájába (*Add...* gombbal). A mező mindig látszódjon, ezért *true* a jobb felső érték, ugyanis ezzel azt is szabályozhatjuk, hogy egy vezérlő dinamikusan felbukkanjon vagy eltűnjön. Az *updateValahogy* egy script, amit a 3.84. ábra mutat. Most tessék figyelni, mert egy fontos megállapítás jön! A *Checkbox1* mezőváltozója

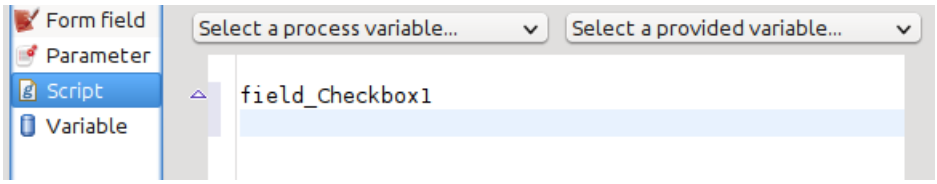
a *field_ Checkbox1*, ezt kell használnunk a scriptben és nem a mögötte lévő *ezVagyAz* változót, hiszen az csak egy szerver oldali műveletnél fog beállni, egy lefutott *action* keretében. Ez a script akkor fog meghívódni, amikor az *a* mezőhöz társított másik vezérlő, azaz esetünkben a *Checkbox1,* állapotváltozáson megy át. Emögött természetesen az AJAX-os működés rejtőzik a háttérben.

Nézzük most a c mezőt! Álljunk rá és válasszuk ki a *Contingencies* fület, ahogy azt a 3.85. ábra is mutatja.



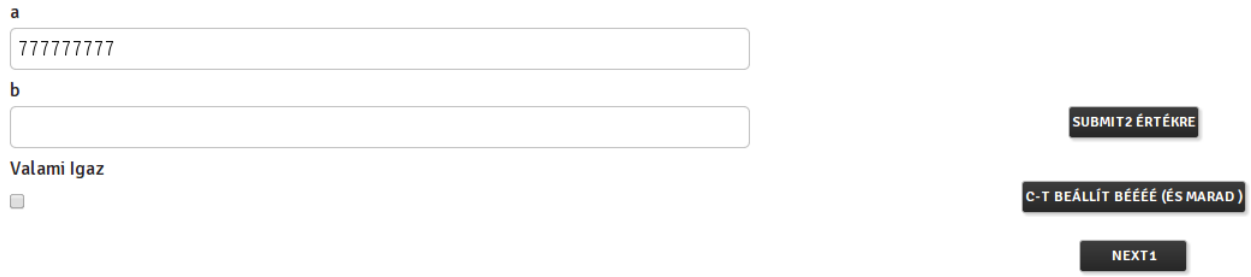

3.85. ábra. A *c* mező Contingencies beállítás

A *This data field contingent on:* (azaz mitől függ a mostani mező, azaz jelen esetben a *c*) részhez felvettük a *Checkbox1* mellé még az *a* mezőt is kísérletképpen, de ez valójában nem része a feladatnak. Csak azt szerettük volna megmutatni, hogy egy widget ily módon több másiktól is függhet. Ugyanakkor jegyezzük meg, hogy bármelyik vezérlő is okozta a változást, a c értékünkre mindig ugyanaz az *ezVagyAzScript* script fog lefutni. Nem adhatunk meg különböző vezérlőkhöz más és más lefutó akciót. Esetünkben most az *ezVagyAzScript* egyetlen sort tartalmat: *field_ Checkbox1* értékét adja vissza (3.86. ábra).




3.86. ábra. Az *ezVagyAzScript*

A 3.85. ábra jobb oldalán egy szintén logikai értékkel, azaz a *field_ Checkbox1* értékével szabályoztuk a láthatóságot. Ha ez *false*, akkor nem, ha *true*, akkor látszik a *c* mező. A 3.87. és 3.88. ábra a futás közbeni tesztet mutatja, amikor a checkbox pipa nélküli és amikor pipás.




3.87. ábra. A C változó mezője nem látszik, az a=7777...


3.88. ábra. A C változó mezője felbukkant, az a=6666...

A kívánt működést tökéletesen előállítottuk. Megjegyzés: Tekintettel arra, hogy scriptek futhatnak a *Contingencies* kezelés során, így ez jó jelölt a háttér munkák bizonyos részeinek elvégzésére is (pl. Save & Edit).

## Az iFrame Widget

Ez egy nagyon egyszerű output komponens, egyetlen paramétere az az URL (példa: `http://index.hu`), ahonnan betöltsön egy *html* tartalmat az iFrame-be.

## HTML Widget

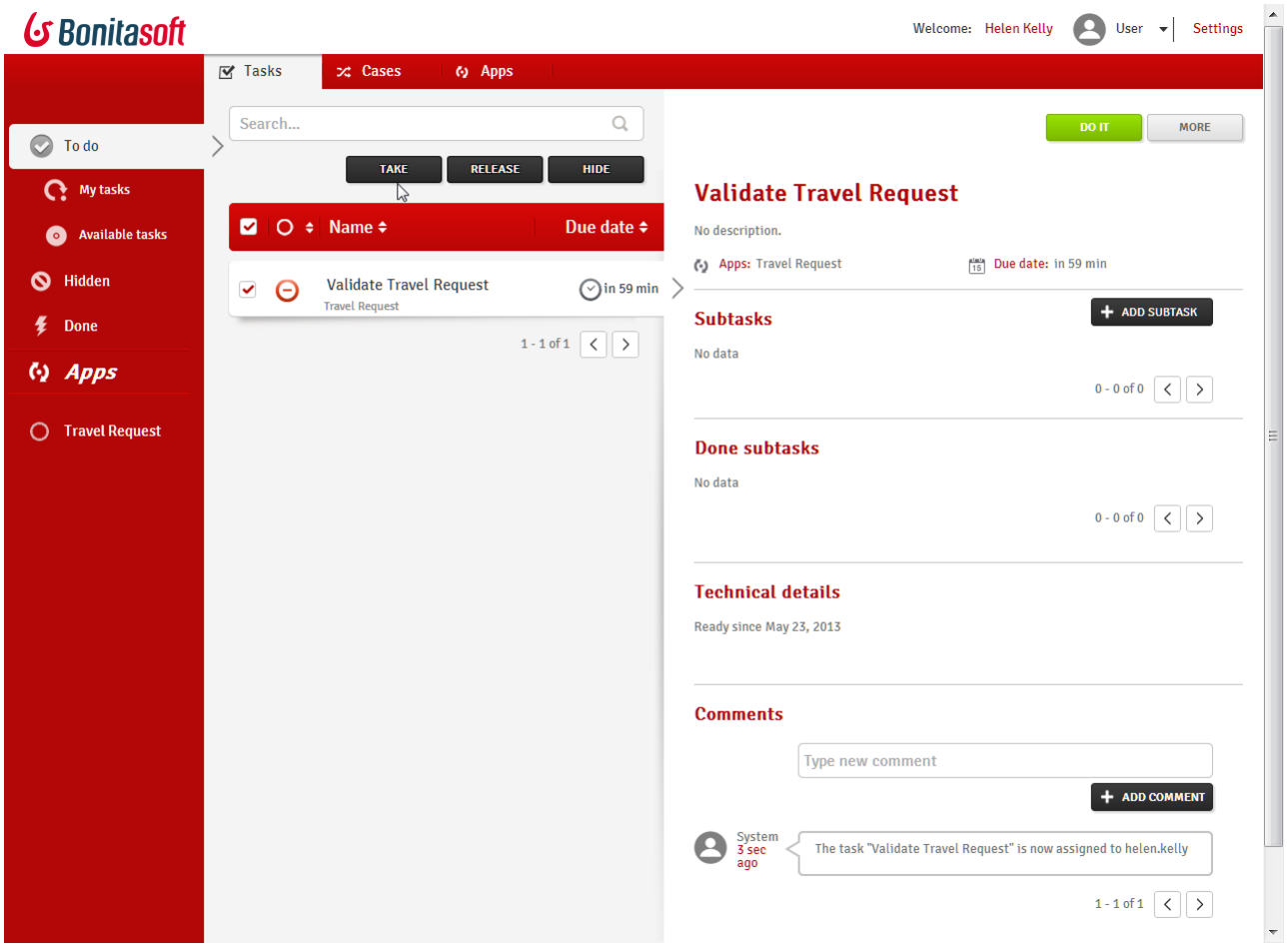
Ez egy nagyon egyszerű output komponens, egyetlen paramétere egy HTML kódrészlet (példa: *aaaaaaaaaa*), amit az *Initial value* mezőnél kell megadnunk. Természetesen ez a kódrészlet egy dinamikus script útján is megadható.

## Simple Button Widget

Amennyiben valamilyen Javascript-et akarunk futtatni, úgy egy ilyen gombra azt a HTML szabványban ismert módon rá lehet tenni.

# 4. A Bonita 6. portál áttekintése

A Bonita Portál az a hely, ahonnan a felhasználó és az adminisztrátor is el tudja érni a mindennapi munkájához szükséges funkciókat. A felhasználók ezen keresztül kezelhetik a taszkjaikat, folyamataikat, riportjaikat. Az adminisztrátor itt menedzselheti a *user*, *group* és *role* neveket, telepítheti a process alkalmazásokat.




4.1. ábra. Bonita 6 Portál - *User* nézet

## A Bonita Portál általános felépítése

A Bonita Portált alapvetően a következő 2 csoport használja:

1. A *felhasználók*: A munkafolyamatok indítása, a feladatok menedzselése és riportok lekérdezése. A 4.1. ábra ezt a nézetet mutatja, amit a jobb felső sarokban olvasható *User* felirat (dropdown vezérlő) is mutat.

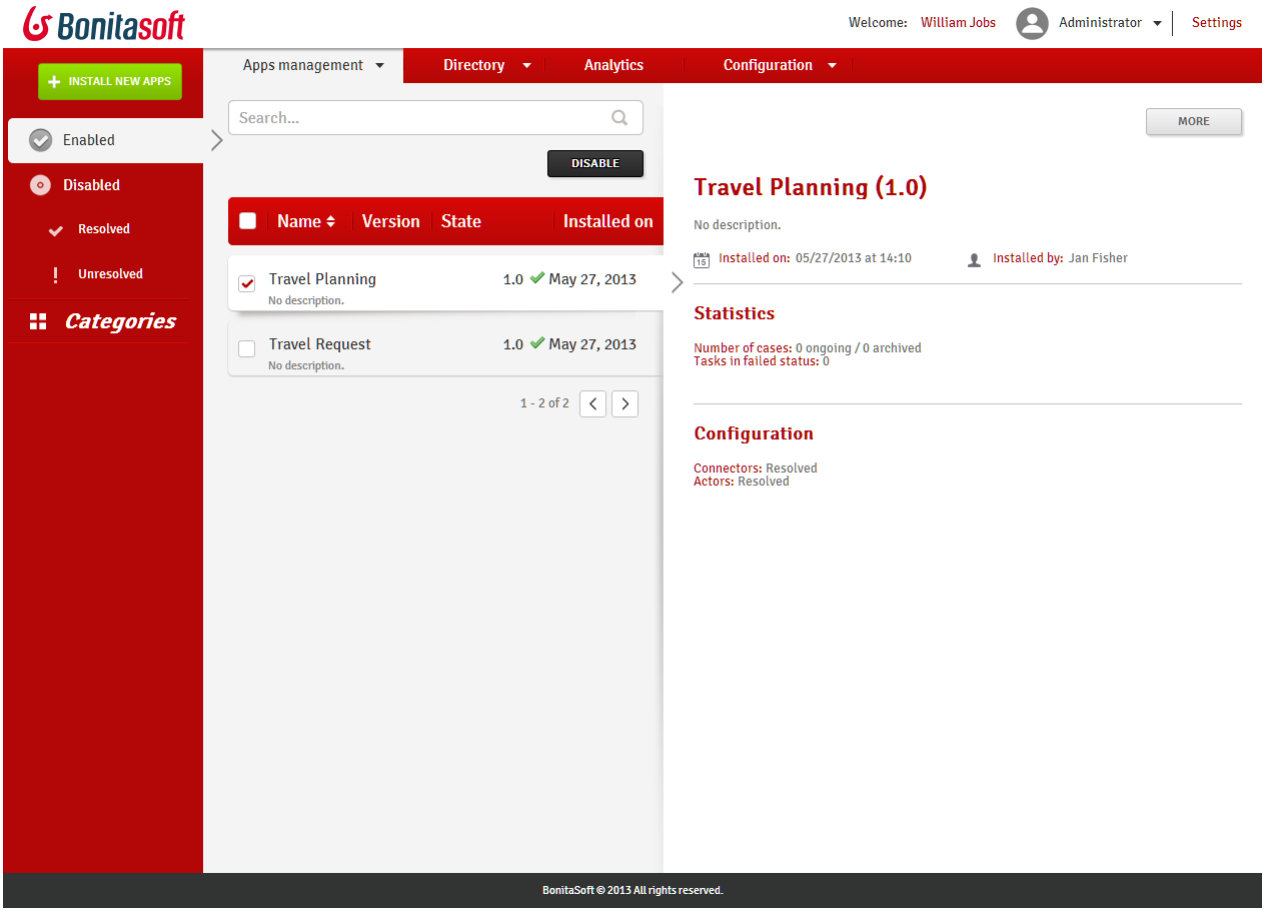
2. A *rendszergazdák*: A Bonita motor és a hozzá kapcsolódó adatbázis menedzselése. Ezt a jobb felső sarok dropdown vezérlőjének *Administrator* értékével választhatjuk ki (4.2. ábra).

A fenti 2 nézet között ide-oda kapcsolhatunk, amennyiben a bejelentkezett user rendelkezik a megfelelő jogosultsággal. A Bonita ezt nevezi *User Profile*-nak. A Bonita 6 kétféle portált bocsát rendelkezésünkre, hogy a mindennapi feladatainkat elérjük és elvégezhessük:

1. A webes verzió, ami desktopos böngészőkre lett tervezve

2. A mobil verzió, ami iOS, Android és Windows Phone alapú készülékek méretéhez van tervezve.

Amennyiben a Bonita Portál alkalmazás a `https://server:port/bonita` helyről érhető el, úgy a mobil verzió egy */mobile* URL utónév használatával innen indítható: `https://server:port/bonita/mobile`. A portált minden nyelven lehet használni, ennek feltétele mindössze egy *GNU gettext* po fájl elkészítése és telepítése. Ezt majd részletesen bemutatjuk.

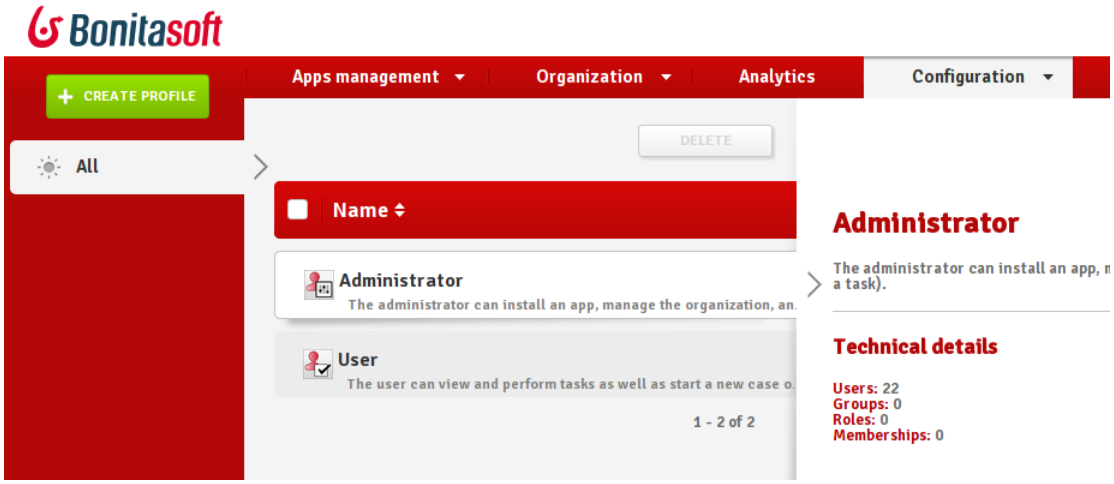



4.2. ábra. Bonita 6 Portál - *Administrator* nézet

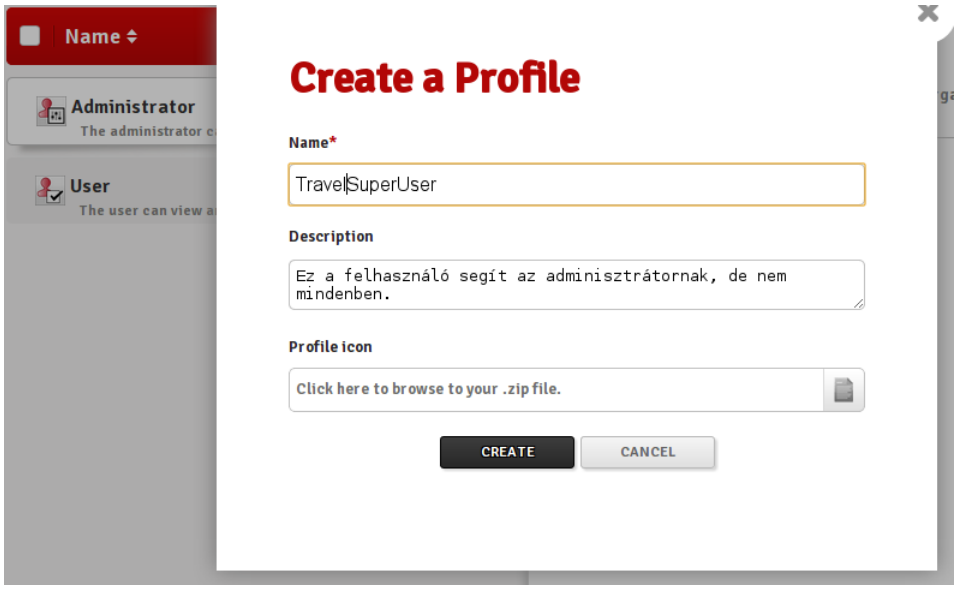
## A Bonita Portál általános felépítése

### A profilok (jogok)

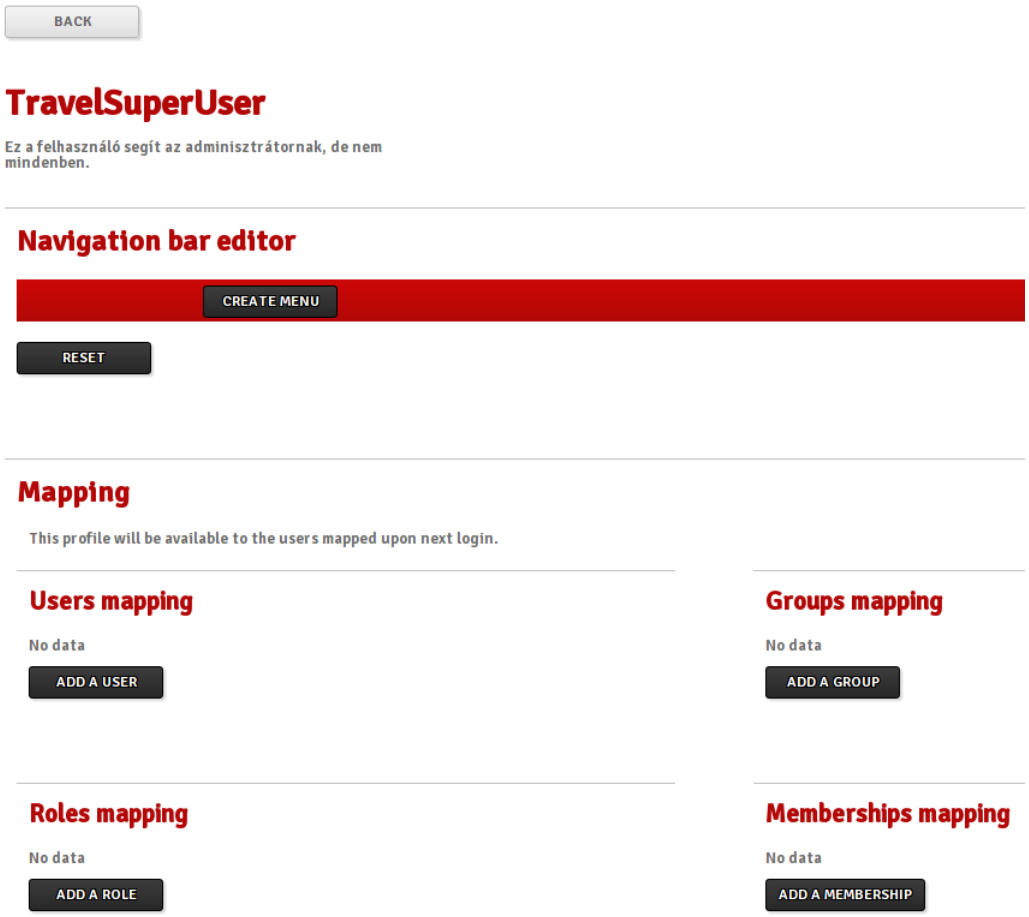
A profile a jogok egy előre megadott halmaza, amit névvel látunk el, hogy utána ezt szükség esetén hozzárendelhessük egy felhasználóhoz. Már említettük, hogy van 2 előre definiált profile: *User* és *Administrator*. Amennyiben egy user-nek a kettő között kell jogokat biztosítani, úgy egy új profile létrehozását a 4.2. ábra *Configuration→Profiles* menüpontjánál kezdeményezhetjük. Ezt régebben *User Right* (felhasználói jog) menüpontnak hívta a portál. A betöltődött lap lényegi részét a 4.3. ábra mutatja.




4.3. ábra. Profile konfigurálás - 1. kép




4.4. ábra. Profile konfigurálás - 2. kép

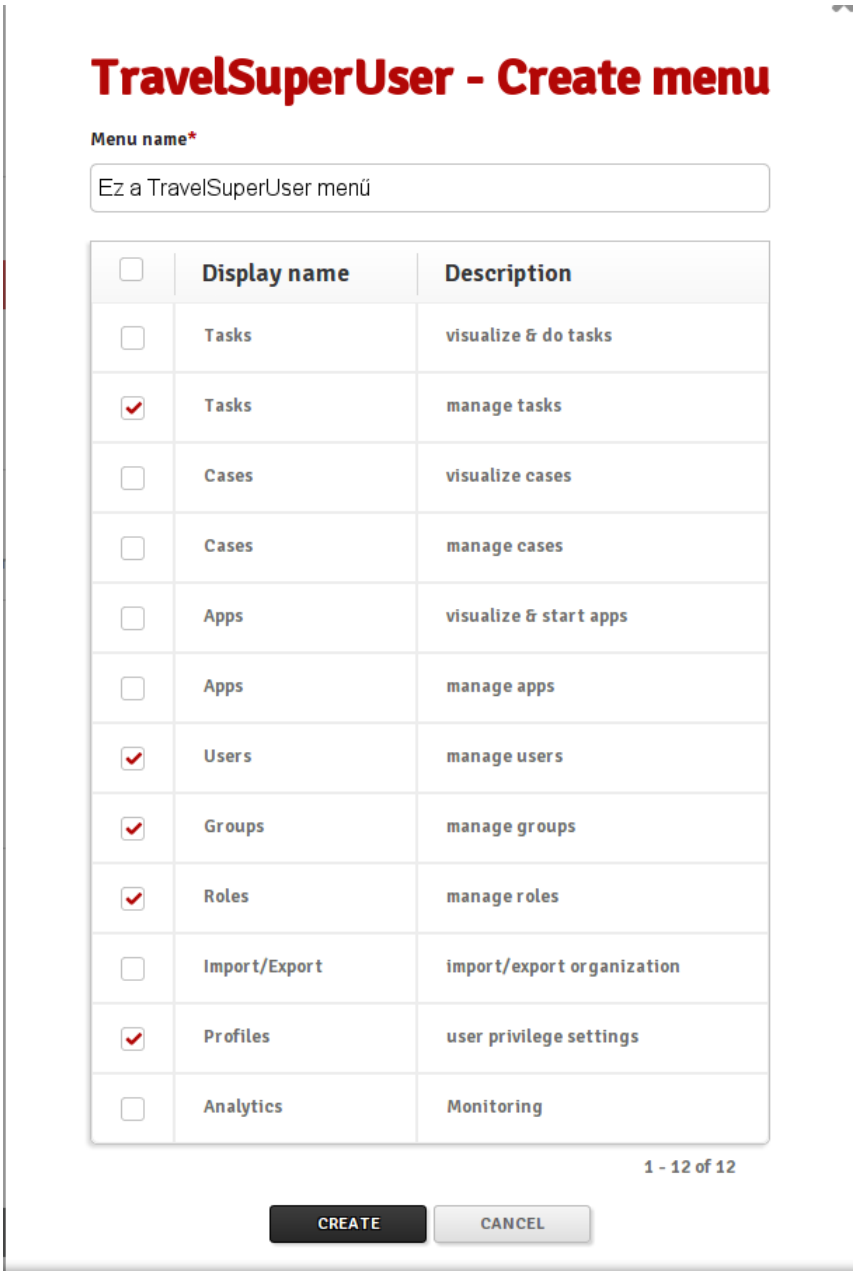



4.5. ábra. Profile konfigurálás - 3. kép

Kattintsunk a + *CREATE PROFILE* gombra, aminek a hatására a 4.4. ábrán mutatott popup ablak jön be. A példában felvettünk egy *TravelSuperUser* profile nevet és megnyomtuk a *CREATE* gombot. Megjelenik a 4.5. ábráról nézhető profil konfigurációs lap, ahol a következő műveleteket végezhetjük el:

- Egy elvégezhető feladatokat összefogó menü készítése a profile számára (*CREATE MENU*). Kiválasztva a 4.6. ábra űrlapja jön be, itt bepipálhatjuk és menü névvel láthatjuk el azokat a funkciókat, amiket ki szeretnénk ajánlani annak a felhasználónak, aki ebbe a profilba van téve.
- A *Mapping* szekció 4 részből áll (4.5. ábra):
  - *Users mapping*: Megadhatjuk, hogy mely felhasználók kerüljenek a konfigurálás alatt lévő profilba. Általában ne konkrét user-eket adjunk meg, de ez a lehetőség időnként mégis szükséges lehet.
  - *Groups mapping*: Megadhatjuk, hogy mely csoportok kerüljenek a konfigurálás alatt lévő profilba.

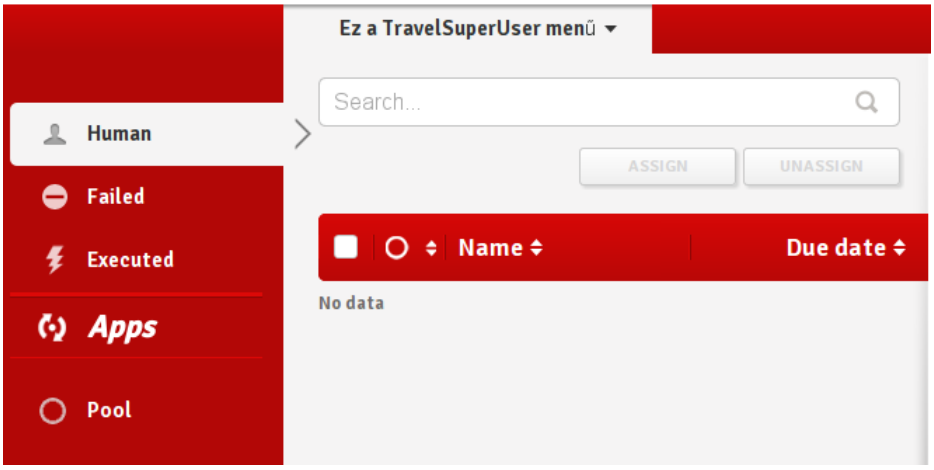
– *Roles mapping*: Megadhatjuk, hogy mely szerepkörökben lévő felhasználók kerüljenek a konfigurálás alatt lévő profilba.
– *Memberships mapping*: Megadhatjuk, hogy mely tagságban (group–role reláció) felhasználók kerüljenek a konfigurálás alatt lévő profilba.




4.6. ábra. Profile konfigurálás - 4. kép

Van egy *inyiri* felhasználónk, őt tegyük be (az *ADD A USER* gomb használatával) a példában mutatott *TravelSuperUser* profilba, majd jelentkezzünk be vele. A 4.7. ábráról láthatjuk, hogy

megkapta a definiált főmenüt, ami csak azokat a funkciókat engedi elérni, amik ebbe a profilba tartoznak.




4.7. ábra. Profile konfigurálás - 5. kép

Befejezésül szeretnénk felhívni arra a figyelmet, hogy a profilok jogosultságokat adnak egy felhasználónak, így amikor egy új user-t hozunk létre, az nem fog tudni addig bejelentkezni, amíg valamilyen profilba nem tesszük be, hiszen enélkül a rendszer nem tudna őrködni a jogosultságai felett.

### Be és kijelentkezés

A Bonita Portál használata azzal kezdődik, hogy a login ablaknál be kell lépni. Amennyiben valamilyen SSO megoldás be van konfigurálva a portált befogadó alkalmazásszerveren, úgy ezt a portál át tudja venni, ekkor nem jelenik meg a bejelentkező ablak. Az alkalmazásból kilépni a *Setting→Logout* kiválasztásával lehet.

## Nyelvi lehetőségek

A Bonita Portál lehetővé teszi a nyelvek közötti váltást a *Settings→Language* menüponton keresztül. Az alaptelepítés csak az ismertebb nyelveket tartalmazza, de egy új nyelv hozzáadása meglehetősen egyszerű. A nyelvi fájlok a *GNU gettext* csomag *po* formátumában készülnek, ezeket kell telepíteni. A *po* fájlok elkészítését az 5. fejezetben részletesen ismertetjük, míg azok telepítését a Bonita éles környezet kialakításának bemutatása során írjuk le. Egy új *po* fájl készítése célszerűen mindig az angol nyelvű változat egy másolata, amit egy *poeditor* eszközzel alakítunk át magyar nyelvűre.

## Az Organization kezelése

Az *Organization* jelentését és jelentőségét a 2. fejezetben már alaposan áttekintettük. Most azt nézzük meg, hogy a Bonita szerveren mindezt milyen módon lehet menedzselni. Ez 4 fő témakört

jelent:

- felhasználók menedzselése
- csoportok menedzselése
- szerepkörök menedzselése
- egy XML formában tárolt *Organization* importja/exportja.

Az összes funkcionalítást a portál *Organization* főmenűpontjától tudjuk kiválasztani.

### Felhasználók menedzselése

Az *Organization→Users* kiválasztásával hozhatunk létre új felhasználót és adhatjuk meg a legfontosabb leíró adatait. A *DEACTIVE* gomb segítségével a kijelölt felhasználót felfüggeszthetjük. Az *EDIT USER* gomb egy létező felhasználó adatainak karbantartását biztosítja. Nagyon fontos rész a *Memberships* adatok megadása, ugyanis itt tudjuk a felhasználók tagságát, azaz összetartozó (*group*, *role*) párjait megadni. Amikor egy felhasználót beteszünk egy csoportba, akkor a szerepkör megadása mindig kötelező, amiatt van egy általános, előre létrehozott *Member* role, ugyanis előfordulhat, hogy bizonyos esetekben csak a csoportba helyezés lényeges számunkra.

### Csoportok menedzselése

Az *Organization→Groups* kiválasztása után jönnek be azok a funkciók, ami új csoportok létrehozását, a meglévőek menedzselését szolgáltatják számunkra. Új csoportot a *CREATE A GROUP* gomb megnyomásával csinálhatunk. Egy régit pedig az *EDIT* gomb kiválasztásával tarthatunk karban.

### Szerepkörök menedzselése

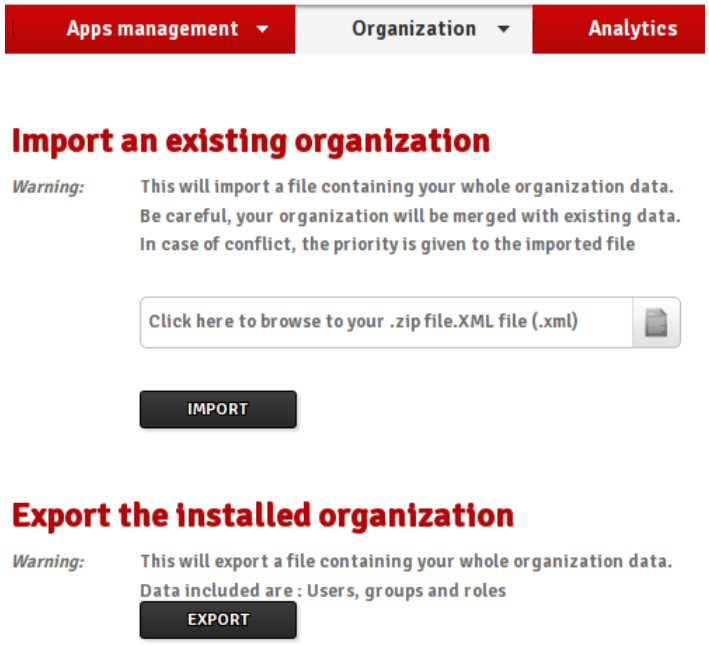
Az *Organization→Roles* kiválasztása után jönnek be azok a funkciók, ami új szerepkörök létrehozását, a meglévőek menedzselését szolgáltatják számunkra. Új szerepkört a *CREATE A ROLE* gomb megnyomásával csinálhatunk. Egy régit pedig az *EDIT* gomb kiválasztásával tarthatunk karban.

### Organization Export/Import

Az *Organization→Import/Export* kiválasztása után jelenik meg a 4.8. ábra tartalma. Itt lehet egy kapott *Organization* XML fájlt hozzáfésülni a mi identity adatbázisunkhoz vagy mi is képesek vagyunk ilyet exportálni. A Bonita Stúdióban is van lehetőség *Organization* részstruktúra létrehozására és exportálására. Itt jegyezzük meg, hogy a stúdióban is az *Organization* főmenüpont szolgál mindezek elvégzésére, a következő almenü pontokkal:

- *Organization→Manage...*: A fejlesztéshez szükséges struktúra (group, role, membership, user) létrehozása.

- *Organization→Publish...*: A stúdió beépített Bonita user adatbázisba telepíti a *Manage...* menüpontnál kialakított struktúrát.
- *Organization→Export...*: XML fájlba exportálja a kiválasztott *Organization* struktúrát.
- *Organization→Import...*: Egy XML fájlból importálja az *Organization* struktúrát.




4.8. ábra. Organization import/export

## A process alkalmazások menedzselése

### Új alkalmazás (process) telepítése

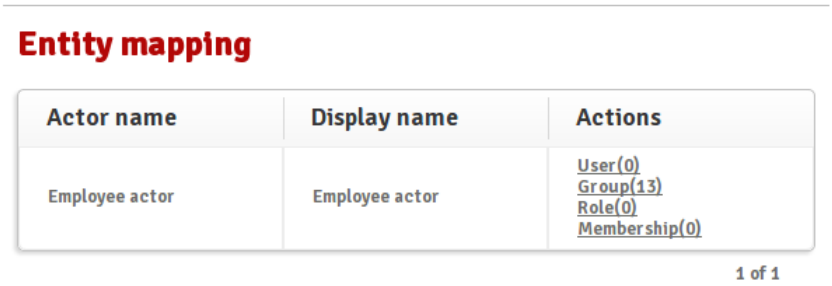
Lépjünk át *Administrator* vagy olyan profile-ba, ami lehetővé teszi a process-ek menedzselését. Az *Apps management→Apps* menüpontot válasszuk ki és nyomjuk meg az *INSTALL APPS* (zöld) gombot, amire egy fájlkiválasztó popup ablak jelenik meg. A Bonita deployment formátuma a *bar* (Business Archive) fájl, ezért olyant kiválasztva tudunk egy új alkalmazást telepíteni, ami ezután meg is jelenik az elérhető alkalmazások listájában. Időnként szeretnénk letiltani (de nem törölni) egy process application-t, ehhez a *DISABLE* gomb ad támogatás. Mindezt az *ENABLE* gombbal tudjuk visszavonni, amennyiben ismét szeretnénk az alkalmazást elérhetővé tenni.

### Az alkalmazások kategóriákba sorolása

Amikor sok process alkalmazást készítettünk már, akkor jól jön, ha ezeket kategóriákba szervezhetjük. Válasszunk ki egy process-t és nyomjuk meg a *MORE* gombot, ahol kategóriákba sorolhatjuk az alkalmazásainkat, sőt ehhez új kategóriákat is felvehetünk. Ezután *User* nézetben az *Apps* menüpontnál már ilyen kategóriákra is kattinthatunk és csak azok a processek jönnek be, amik a kiválasztottba tartoznak.

### ACTOR mapping a portálban

Mindegyik process alkalmazáshoz fejlesztési időben elkészíthetjük a 2. fejezetben bemutatott Actor Mapping műveletet, ahogy azt a 4.9. ábra is mutatja. Ez azért rugalmas, mert újratelepítés nélkül is tudjuk dinamikusan szabályozni, hogy egy ACTOR név milyen USER halmazra képződjön le.




4.9. ábra. Actor Mapping a portálon

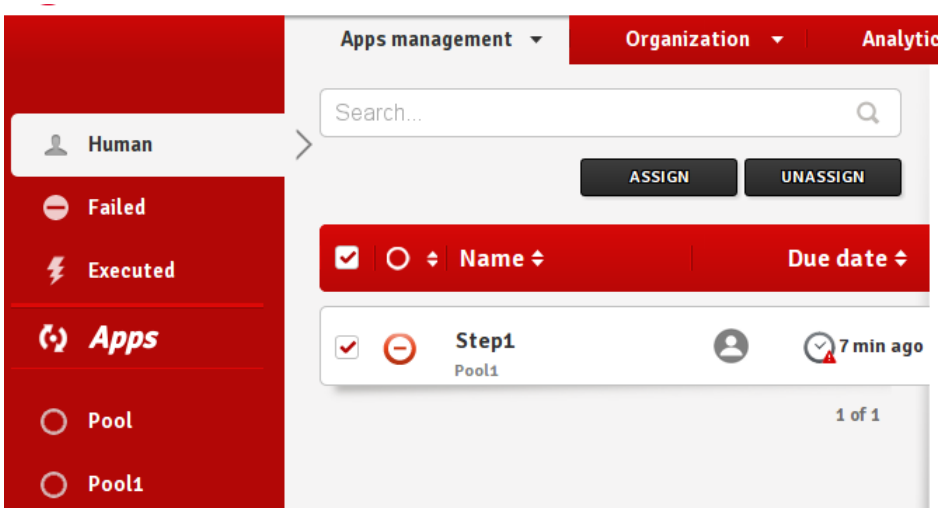
## A process példányok menedzselése

Egy process alkalmazás példányát a Bonita *Case*-nek vagy process instance-nak is nevezi. Ezt magyarul workflow példánynak nevezhetjük. Ezek menedzselését az *Apps management*→*Cases* menüpontja valósítja meg. Külön elérhetjük a még nyitott (*Opened* fül) és a már lezárult (*Archived* fül) folyamat példányokat, visszanézhetjük azok historikus adatait. A nyitott folyamat példányokat az *Apps* címke alatt külön alkalmazásonként is nézegethetjük. Miután ráálltunk egy konkrét folyamatra, kiválaszthatjuk a *MORE* gombot, ami a lefutás részleteit mutatja be, sőt tetszőleges számú megjegyzést is fűzhetünk mindehhez. Az itt lévő *OVERVIEW* gomb pedig egy összegző áttekintést ad a teljes folyamatról.

## A process TASK-ok menedzselése

Kiválasztva az *Apps management*→*Tasks* pontot egy olyan lap jön be, aminek a lényegi részét a 4.10. ábra mutatja. A *Human* fülről egy fontos adminisztrátori feladatot végezhetünk el, ugyanis minden kiválasztott TASK-hoz az *ASSIGN* és *UNASSIGN* gombokkal további feladat végrehajtókat adhatunk meg, vagy éppen vonhatunk vissza dinamikusan. A *MORE* gombra kattintva áttekinthetjük a feladat eddigi kísérő adatait, sőt mindehhez itt is fűzhetünk megjegyzéseket az

*ADD COMMENT* gomb segítségével. A *Failed* és *Executed* fülek az ide tartozó feladatokat mutatják. Az *Apps* résznél munkafolyamatonkénti bontásban is áttekinthetjük a TASK-okat.




4.10. ábra. TASK menedzsment - Administrator módban

## Analitikák admin profile-ban

### Monitorozás

Az *Analytics→Monitoring* kiválasztása után alaphelyzetben a következő információkat szolgáltatja a portál admin profile esetén:

- A hibára futott taszkok száma (Number of failed tasks)
- A lefutott workflow példányok száma (Number of apps resolved)
- A nyitott workflow példányok száma (Number of open cases)
- A nyitott taszkok száma (Number of open tasks)

### Riportok készítése

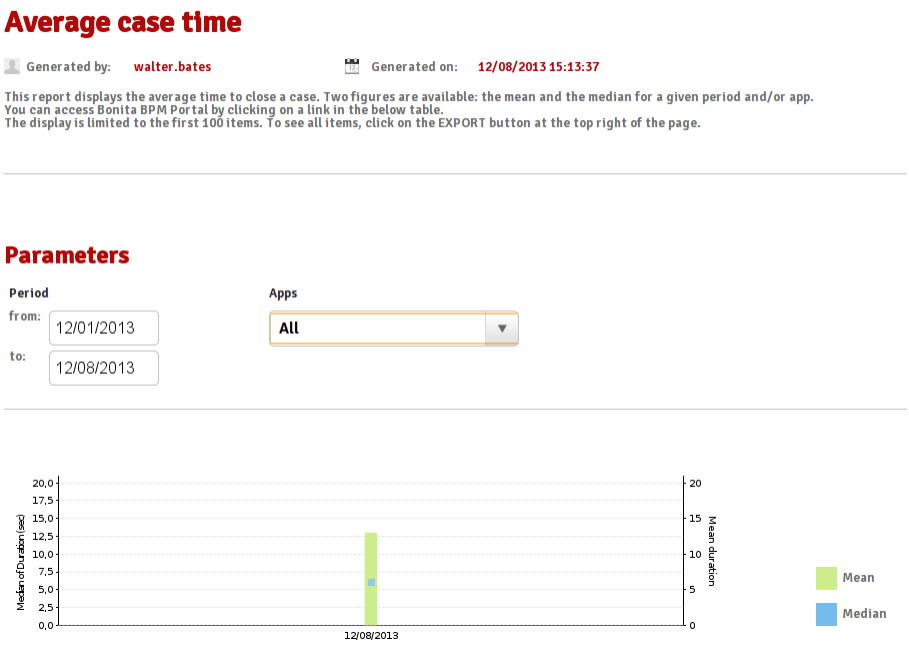
Az adminisztrátor gyárilag a következő riportokat láthatja:

- case lista
- átlagos case végrehajtási idő
- taszkok listája

A 4.11. ábra például a második lehetőséget mutatja. A lekért grafikonhoz megadhatjuk az időszak kezdetét és végét, valamint kiválaszthatjuk a vizsgálni kívánt workflow alkalmazást is (*Apps* címke). A többi riport is hasonlóan paraméterezhető. Az *EXPORT* gombbal PDF-be menthetjük a riportot.

### Egy új riport telepítése

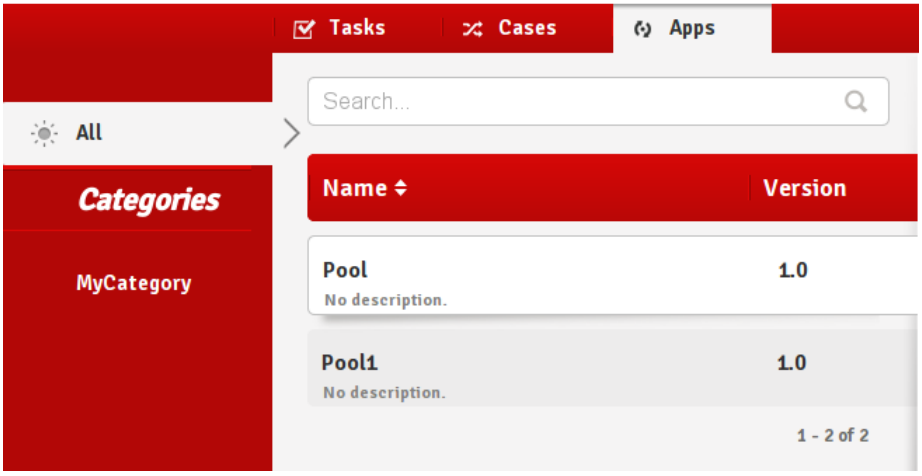
Magunk is készíthetünk riportot Jasper Reports eszközben, amit utána a gyári lehetőségek mellé telepíthetünk. Ezt az Az *Analytics→INSTALL REPORT* gombbal eszközölhetjük.




4.11. ábra. Átlagos workflow példány időtartam riport

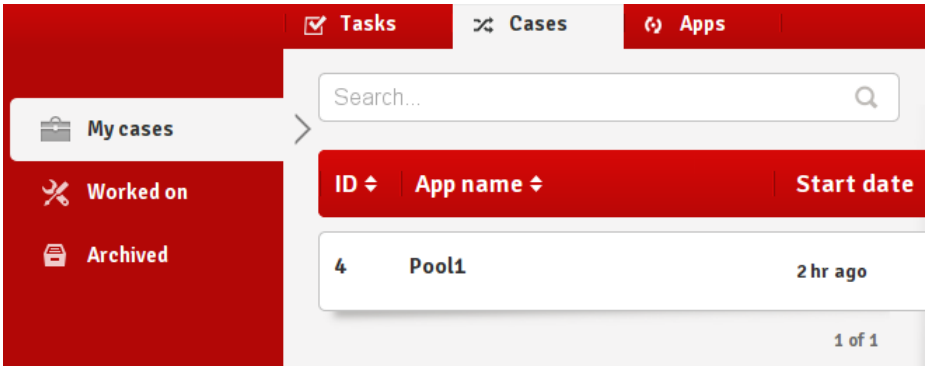
## A Bonita Portál használata felhasználói módban

### Process példányok indítása

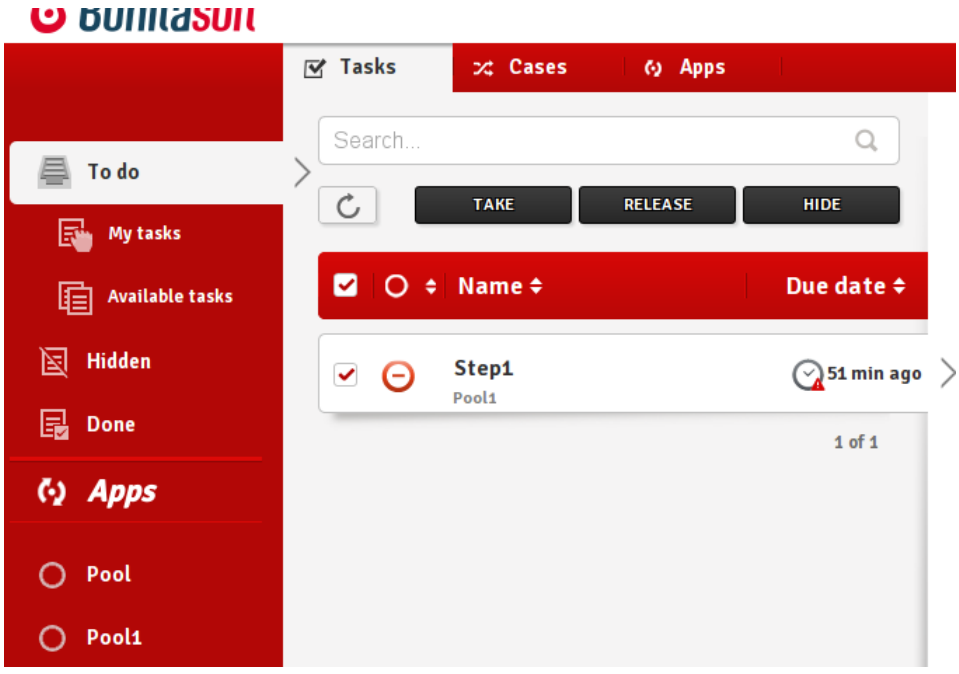



4.12. ábra. Bonita Portál - felhasználói módban - *Apps*

A 4.12. ábra a User profile induló lapjának azt a részét mutatja, ahol kezdeményezhetjük a process alkalmazások (*Apps*), példányok (*Cases*) és feladatok (*Tasks*) használatát. Amennyiben egy új munkafolyamatot szeretnénk indítani, akkor menjünk az *Apps* vízszintes fülre és válasszuk ki a megfelelő process alkalmazást, amit elindítani a *START* gombbal lehet (nem látszik az ábrán). Az ábrán most 2 kiválasztható alkalmazás van: *Pool* és *Pool1* (ezek csak tesztek, ezért nem túl beszédesek). Szeretnénk kiemelni, hogy a *Categories* résznél látható a létrehozott, jelenleg egyetlen kategóriánk, ami az elindítható process típusokat szedi össze egy-egy logikai csoportba.




4.13. ábra. Bonita Portál - felhasználói módban - *Cases*




4.14. ábra. Bonita Portál - felhasználói módban - *Tasks*

### Process példányok kezelése

A *Cases* fül (4.13. ábra) a már futó vagy lefutott (és archivált) workflow példányok kezelését teszi elérhetővé. A *My cases* azokat a process példányokat mutatja, amiket a bejelentkezett felhasználó indított el. A *Worked on* mutatja azokat, amiken az aktuális user dolgozik. Az *Archived* pedig a már lefutott workflow példányokat teszi elérhetővé. Mindezen eszközök azt segítik, hogy a végén egy konkrét workflow instance-t (aminek az egyedi azonosítóját az *ID* oszlop mutatja) ki tudjunk választani és alkalmazzuk rá a *MORE* gombot, ahol az adminisztrátori műveletekhez hasonlókat tudunk elvégezni.

### A taszkok kezelése

A 4.14. ábra a feladatok kezelésének kiinduló pontját mutatja. A *To do* az éppen elvégezhető összes feladatot mutatja, természetesen a bejelentkezett felhasználó vonatkozásában. Az ábrán látható 3 fekete gomb használata fontos, ezért nézzük meg egyenként a jelentésüket:

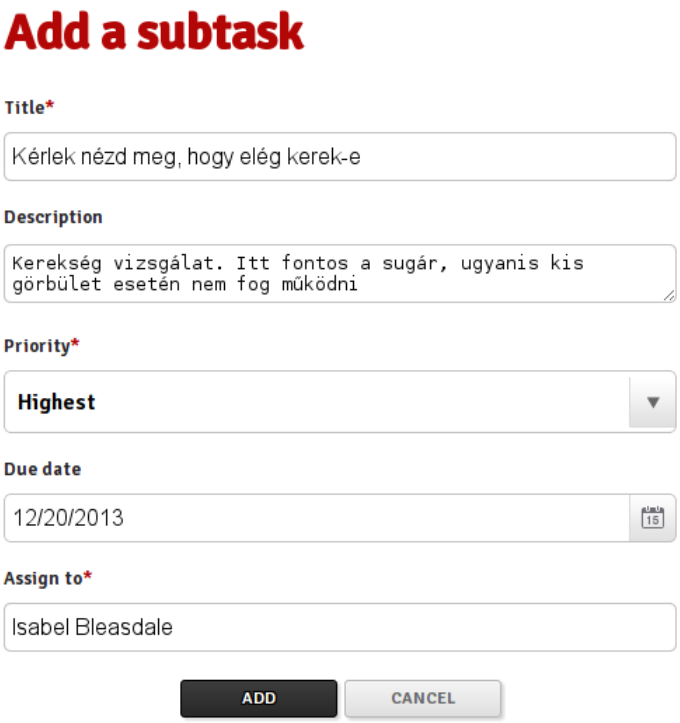
- *TAKE*: Egy kiválasztott TASK ezzel csak a miénk lesz, más szavakkal magunkhoz vesszük, hogy mi csináljuk meg. Ezzel ez bekerül a *My Tasks* mappába, ugyanis innentől kezdve csak mi látjuk.
- *RELEASE*: Amennyiben egy magunkhoz vett TASK-ot vissza szeretnénk tenni a taszk kosárba, úgy válasszuk ki és nyomjuk meg rá ezt a gombot. Ezzel újra mindenki láthatja, akik a TASK létrejöttekor jogosultak voltak azt kezelni.
- *HIDE*: A kiválasztott TASK-ot elrejti, ami ezzel megjelenik a *Hidden* mappában. Egy elrejtett feladatot a *RETRIEVE* gombbal ismét láthatóvá tehetünk a taszk kosárban.

A *Done* mappa tárolja a felhasználó összes elvégzett feladatát, azok között természetesen szintén lehet keresni. Az *Apps* címke alatt az elérhető taszkjainkat alkalmazásonként bontva is láthatjuk. Minden számunkra látható és elvégezhető taszkra kiválaszthatjuk a *DO IT* (zöld) gombot, amire bejön a taszkot kezelő form, azon elvégezhetjük, adminisztrálhatjuk a feladatunkat. A *MORE* gomb itt is ad egy áttekintési lehetőséget. Már a *DO IT* gomb megnyomása előtt is van 2 kiegészítő lehetőségünk a taszkok kezelésével kapcsolatosan:

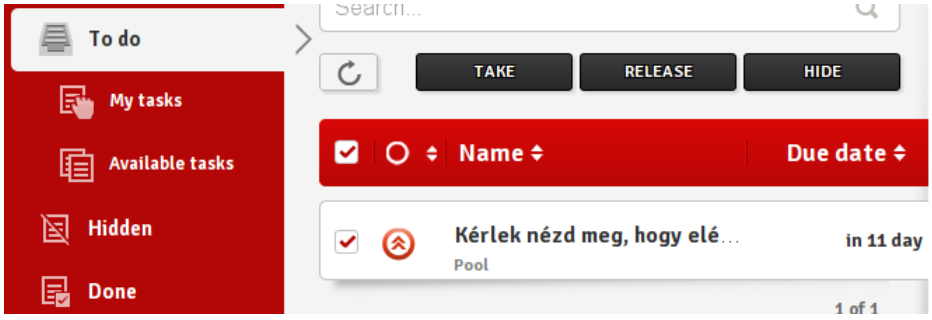
- Készíthetünk hozzá tetszőleges számú alfeladatot (*ADD SUBTASK* gomb, lásd a következő pontot)
- Fűzhetünk hozzá tetszőleges számű megjegyzést (*ADD COMMENT* gomb)

### Az alfeladat (subtask)

Amikor egy TASK példány esetén az *ADD SUBTASK* gombot nyomjuk meg, akkor másokat is bevonhatunk abba a részfolyamatba, ahogy a taszkot megoldjuk és hozzá összegyűjtjük az információkat. Ez lényegében olyan, mint egy másik felhasználótól kért megjegyzés az adott feladatra vonatkozóan. A gombnyomásra a 4.15. ábra ablaka jelenik meg, nem szükséges semmilyen magyarázat ehhez.




4.15. ábra. Alfeladat kiadása




4.16. ábra. Az alfeladat megérkezése

A 4.16. ábra már azt jeleníti meg, hogy *Isabel* belépett és a munkakosarában ezt az altaszkot ugyanúgy látja, mint mindegyik másikat. Sőt még Ő is csinálhat belőle altaszkot és adhat hozzá megjegyzést. Amennyiben megoldja, úgy egy többsoros szövegbe írhatja be a válaszát, azaz a subtask mindig ilyen formot jelenít meg.

## A BPM Portál és a Formok elérése

A Bonita Portálon kezelt TASK formok mindegyikének van egy URL-je, ezzel direkt is elérhetjük a taszkokat kezelő űrlapokat. Erre néhány szituációban szükség is van, gondoljunk csak arra, amikor

egy feladatot megoldó linket levélben küldünk ki. Tegyük fel, hogy ez az URL a portál linkje:

```
https://myserver:8080/bonita
```

Ekkor egy form direkt elérése:

```
http://myserver:8080/bonita?ui=form&locale=en#form=Task%20link%20via%20email--1.0--Task$entry&➥
    mode=app&task=82
```

Ezt a linket egy ilyen Java programmal is elő tudjuk állítani:

```
StringBuffer buffer = new StringBuffer("URL: <a href=\"");
buffer.append("http://");
buffer.append(host
buffer.append(":");
buffer.append(port);
buffer.append("/bonita?ui=form&locale=en#form=");
buffer.append(formName);
buffer.append("\$entry&mode=app&task=");
buffer.append(activityInstanceId);
buffer.append("\">here</a>");
return buffer.toString();
```

Amennyiben az autologin lehetőséget (anonymous user) szeretnénk használni, akkor a

```
&autologin=PoolName--PoolVersion
```

részt adjuk meg a # karakter (a form van ott) előtt. Az is lehet, hogy a Bonita Portálhoz köthető URL-t egyátalán nem szeretnénk használni, ekkor az URL-ben a *&mode=app* részt *&mode=form* szövegre kell cserélnünk a # karakter után.

## A mobil portál

Ahogy azt már említettük, a Bonita rendelkezik a portál mobil platformra testre szabott változatával is. Itt fontos kiemelni, hogy ekkor az *Administrator* profile nem érhető el, kizárólag csak a felhasználói feladatok elvégzése támogatott. A mobil portál egy külön nyelvi *po* fájllal rendelkezik, így ha ezt is akarjuk használni, akkor azt is telepíteni szükséges.

## Naplózás

A Bonita a Java beépített (*java.util.logging*) naplózását használja, a részleteket a Bonita dokumentáció ezen az URL-en tartalmazza: `http://documentation.bonitasoft.com/logging-overview`. Amennyiben

- Tomcat a szerver, úgy ezt érdemes átnézni: `http://tomcat.apache.org/tomcat-6.0-doc/logging.html`.
- JBOSS esetén pedig ezt: `https://community.jboss.org/wiki/JDKLoggingInJBoss51?_sscc=t`.

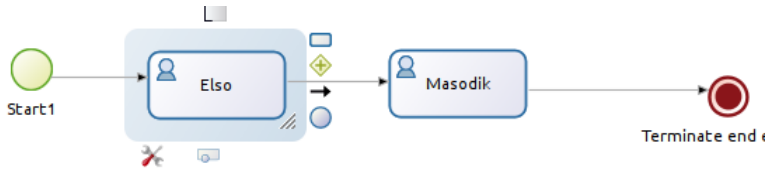
# 5. Alkalmazás fejlesztés a Bonita Stúdió 6. használatával

Ezen fejezetben bemutatjuk azokat a Bonita lehetőségeket, amik egy alkalmazás összeépítése során hasznos elemek lehetnek. Az egyes témák csak lazán csatlakoznak egymáshoz, de együtt hozzájárulnak ahhoz a tudáshoz, ami egy sikeres alkalmazás építése során szükséges.

## A TASK Operations használata

Legyen az a teszt workflow, amit az 5.1. ábra mutat, ennek a következő 2 darab pool szintű változója van:

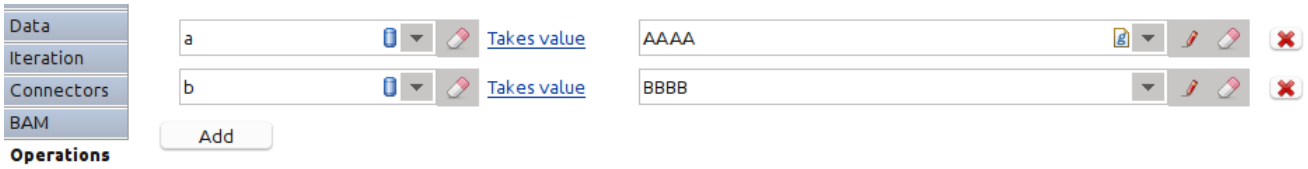
1. *a*: Text típus
2. *b*: Text típus



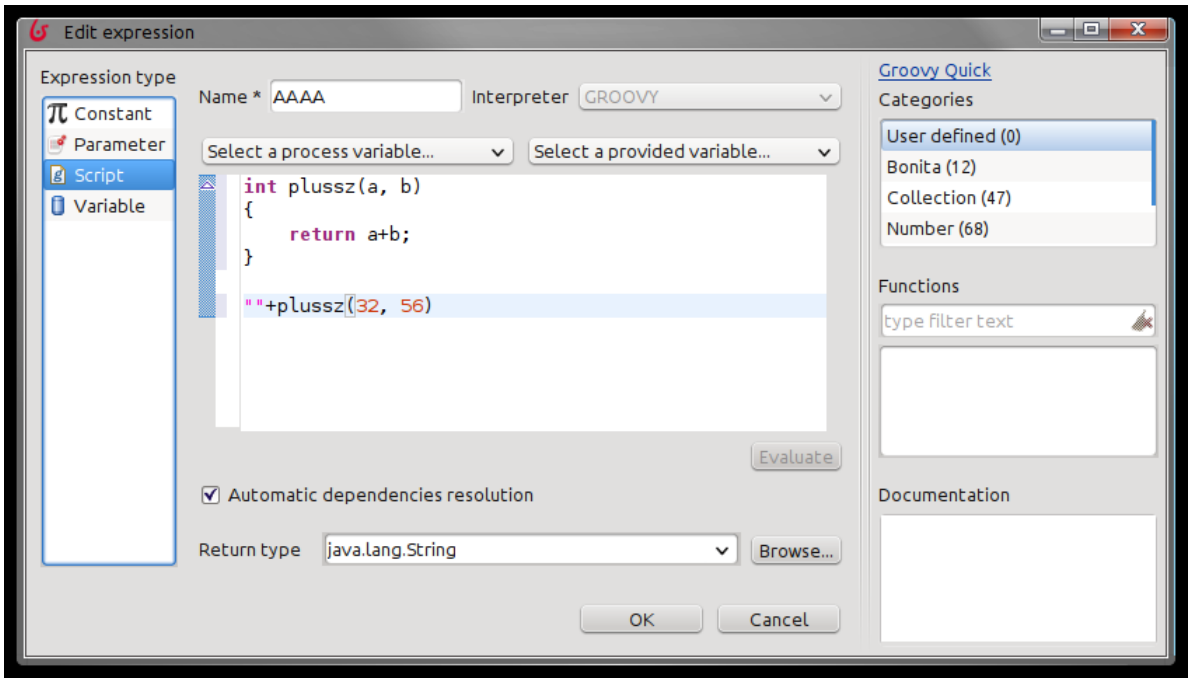

5.1. ábra. Operations használata - Teszt workflow

Tekintsük az *Elso* nevű taszkot és a többi beállítás mellett vegyünk fel 2 *Operations*-t, azaz műveletet is. Ezek olyan akciók (action), amik akkor futnak le, amikor a step kompletté válik, azaz esetünkben az *Elso* taszk elvégzése után. A 5.2. ábra már az *Elso* step konfigurációjának végeredményét mutatja, amit az *Operations* fülön végeztünk el és így kell értelmezni:

- az *a* workflow változó vegye fel azt az értéket, amit az *AAAA* nevű Groovy script (5.3. ábra) előállít.
- a *b* workflow változó vegye fel a *BBBB* konstans értéket (ez nem script most!).

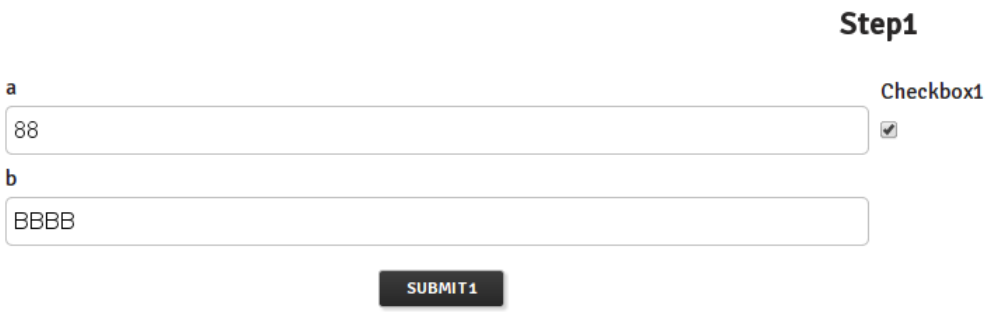



5.2. ábra. A 2 darab Operation megadva




5.3. ábra. Az *AAAA* nevű script tartalma

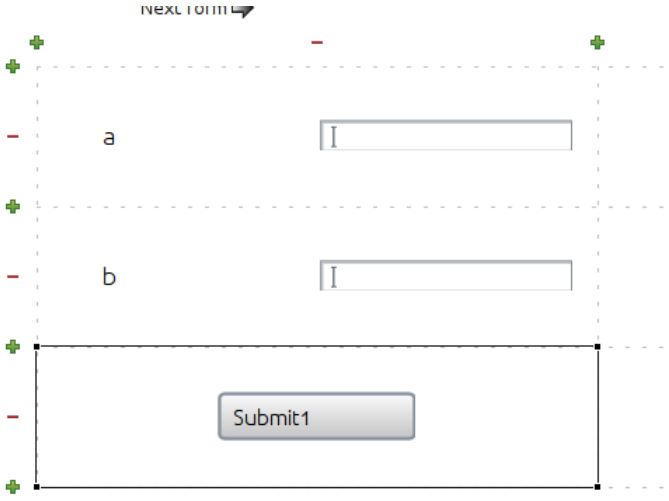
Itt most 2 példát láttunk a Script Editor használatára is, de a cél az volt, hogy megismerjük a TASK *Operations* használatát, aminek az a haszna, hogy a task befejeződésekor még a workflow belső változóin műveleteket végezhessünk, így azok állapotát beállítsuk. Tekintettel arra, hogy eközben egy script sok más műveletet is el tud végezni (adatbázis, e-mail, ...), ez egy meglehetősen általános eszköz a kezünkben. A *Masodik* nevű TASK ezen műveletek miatt már úgy indul el, hogy a változó értékek be vannak állítva, ahogy azt az 5.4. ábrán látjuk (az ábra egy tesztfutás képrészlete).




5.4. ábra. Operations - Beléptünk a *Masodik* TASK-ba

## Több submit gomb használata

Ebben a pontban röviden leírjuk azt, hogy egy Bonita Form submit típusú gombjához is lehet akciókat kötni, ezek hasonlóak az előzőekben ismertetett task operations akciókhoz. Amennyiben egy Form (egyik) submit gombját megnyomjuk, úgy majdnem mindig szeretnénk valami egyéb „maradandót" is csinálni, tipikusan futtatni egy vagy több scriptet és beállítani egy vagy több workflow változót, majd utána menjen csak a folyamat a következő task-ra (vagy taskok-ra). Nézzük meg ennek a módját a lenti példán keresztül!



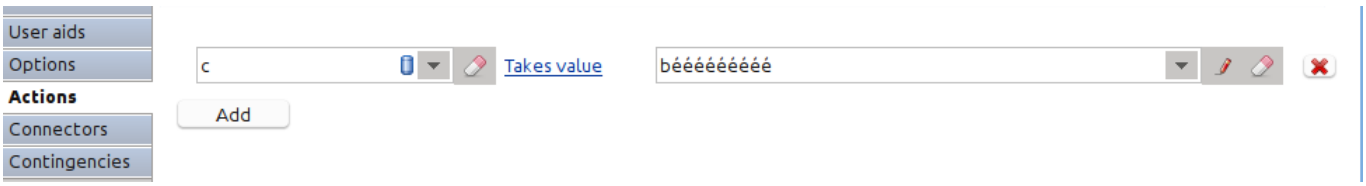

5.5. ábra. Az Elso TASK formjának egy részlete

Tekintsük ismét az 5.1. ábrát, a process-nek már 3 darab pool szintű változója van, mert felvettük a $c$-t is:

1. $a$: Text típus
2. $b$: Text típus
3. $c$: Text típus

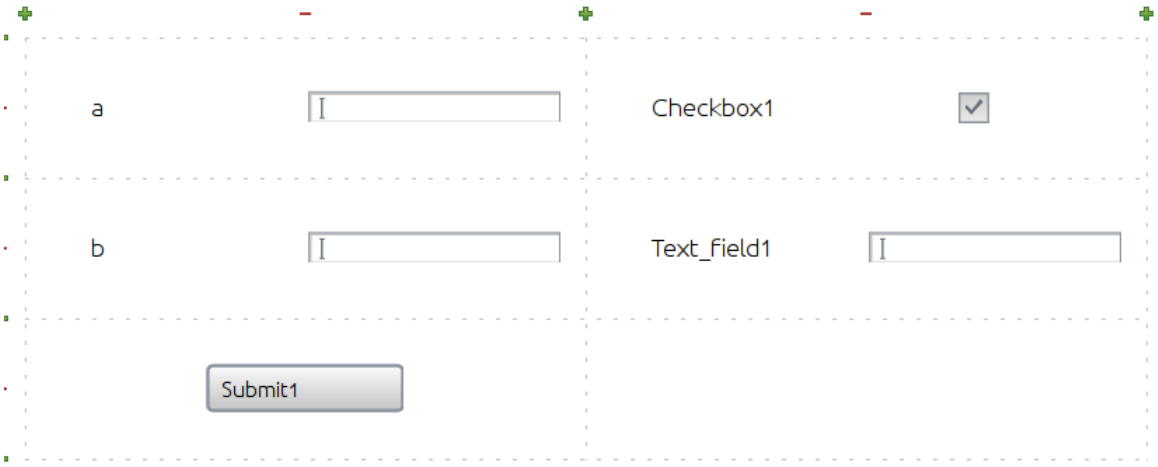
A 5.5. ábrán egy olyan Formot látunk, ami az 5.1. ábra *Elso* task-ja esetén aktiválódik.

A Form-on egy a *Submit1* gombhoz rendelünk most akciót (többet is rendelhetnénk, de példának elég egy is). Jelöljük ki a gombot és álljunk annak a *General* → *Actions* fülére (5.6. ábra).




5.6. ábra. A *Submit1* gombhoz akciót rendeltünk

Az 5.6. ábrán már 1 hozzárendelt akciót látunk, ennek kinézete és használata a TASK operations résznél ismertetésre került, azaz a példánkban a *c* változóba kerül az az érték, amit a jobb oldali Script Editor-ban elkészített megoldás tesz. Most ez csak egy *béééé...* konstans. A *Masodik* nevű TASK Form-ja a 5.7. ábra szerint van tervezve, a *Text_field1* widget a *c* változóhoz lett kötve (binding).




5.7. ábra. A *Masodik* nevű TASK form-ja

A *c* változó értékét az 5.5. ábra form-ján kiadott *Submit1* gombnyomás beállítja *béééé...* értékre, ezért azt várjuk, hogy a tesztfutás során a 5.7. ábra form-ja tényleg mutassa ezt az értéket. A próba eredményét a 5.8. ábrán látjuk, azaz minden úgy működött, ahogy vártuk.


5.8. ábra. Fut a workflow - a *Masodik* task form-ja

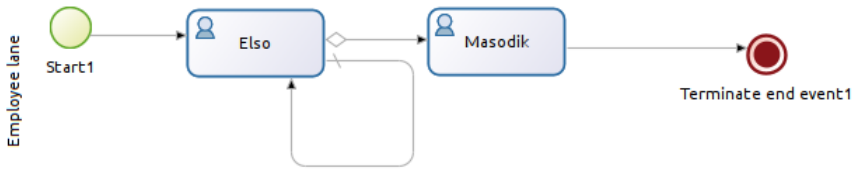
*Megjegyzések:*

1. Amennyiben több Submit gombot teszünk ki a form-ra, úgy ezzel könnyen szabályozhatjuk, hogy milyen action(ok) fussanak le.

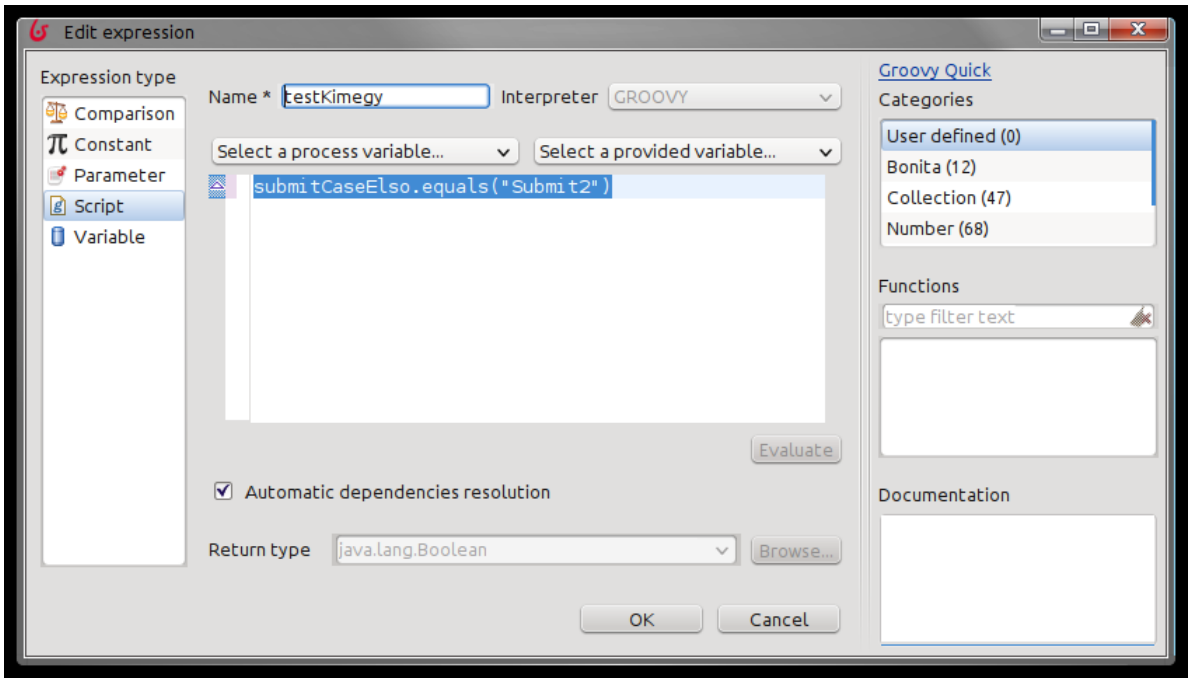
2. Az 1. megjegyzés speciális esete az, amikor azt routolásra is használhatjuk a BPMN kapukhoz. Felvehetünk egy *submitCaseElso* nevű változót (itt most arra utaltunk a névben, hogy submit és melyik TASK-ra nézve), aminek értéke lehet például az, hogy melyik Submit

gombot nyomták meg. Ezt a változót pedig valamelyik Submit gomb action-nal állíthatjuk be.

A 2. megjegyzés megoldását elkészítve az 5.9. ábra szerinti eredményt kapjuk.




5.9. ábra. Kissé megváltoztatott BPMN design




5.10. ábra. A *taskKimegy* script

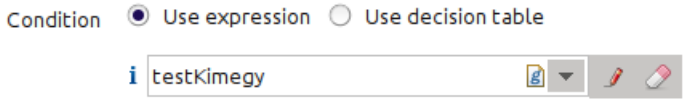
*A változtatások ezek voltak:*

1. Az *Elso* TASK-ból feltételesen kehet kimenni, amely logikai értéket egy *testKimegy* script állít elő (5.10. és 5.11. ábrák). Az 5.10. ábrán látszik, hogy felvettünk egy *submitCaseElso* Text típusú változót is.

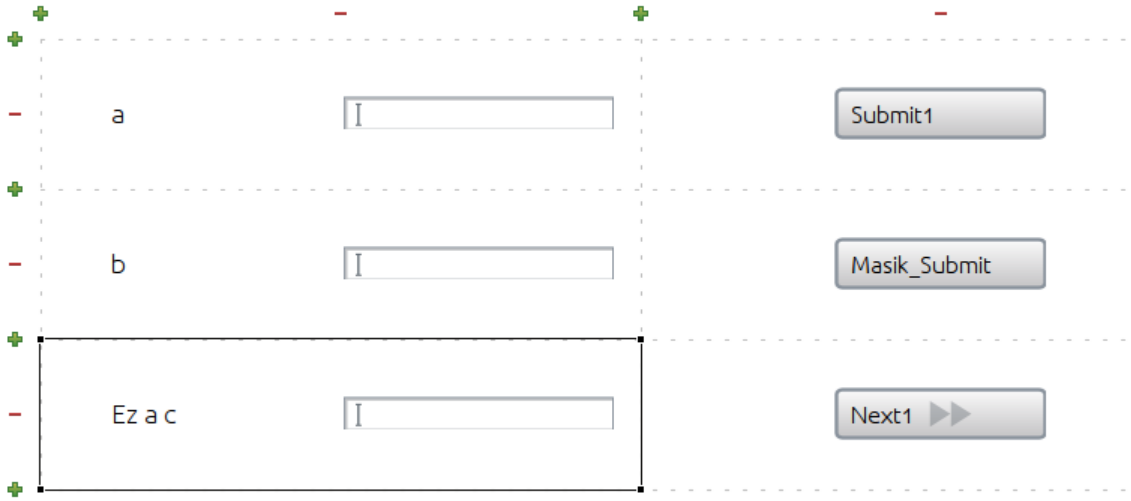
2. Az *Elso* TASK formája feltettük a *Masik_Submit* gombot (5.12. ábra).

3. A *Submit1* action-ja maradt, de a *Masik_Submit* action-ra felvettük, hogy amennyiben őt nyomják meg, úgy a *submitCaseElso* változó vegye fel a *Submit2* értéket, mert erre vizsgálunk a *testKimegy* scriptben (5.13. ábra).

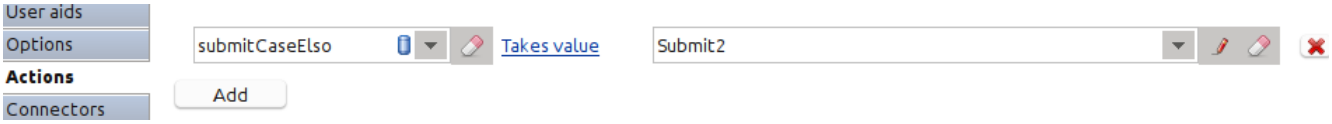
4. Különben *default* ágon visszamegy a folyamat vezérlés az *Elso* TASK-ba.




5.11. ábra. A feltételt így tettük az *Elso* task-ból kijövő nyílra




5.12. ábra. Feltettük a *Masik_Submit* gombot is




5.13. ábra. A *Masik_Submit* action-ja

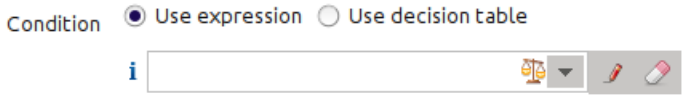
Mindez, azaz a lefutás routolása, szépen működik is, de van egy kis szépséghiba. Ezzel a módszerrel rengeteg alkalommal hozzuk létre az *Elso* task egy-egy példányát, amit a Portálon látott taskok résznél látunk is. Erre célszerű lenne egy olyan action-t találni, ami bent tartja a TASK-ban a folyamatot.

## A kifejezés szerkesztő

Az Expression Editor lehetővé teszi a programozónak definiálni egy konstans értéket, megadni egy kifejezést, összehasonlítás készítése 2 érték között (reláció), illetve egy teljes Groovy script létrehozását (amikor egy összetettebb programrészlet szükséges). Sok előre definiált kifejezést használhatunk. A kifejezés szerkesztő a Bonita Stúdióban sok helyről elérhető, hiszen gyakori igény a workflow medence (pool), sáv (lane) vagy feladat (task) szintű változóinak valamilyen célú elérése, beállítása, fölöttük valamilyen számítási algoritmus megvalósítása.

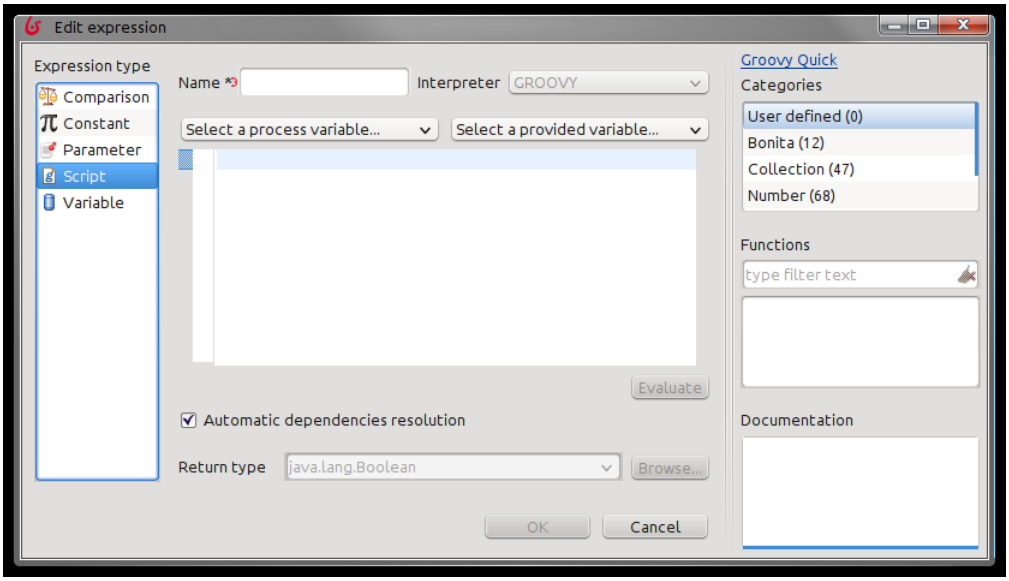
### A kifejezés szerkesztő elindítása

A munka során az lesz a leggyakoribb, hogy a kis *crayon icon*-ra (pasztellkréta ikon) kattintunk ott, ahol ez elérhető (5.14. ábra, a buborék help felirata: *Edit...*). Az ábráról mindjárt láthatjuk az egyik jellemző használatot, ahol egy logikai *true* vagy *false* eredménnyel feltételt (Condition) kell megadnunk. Sok ilyen hely van, de 2 jellemzőt példaképpen megemlítünk: egy XOR kapu elágazási feltételének megadása, egy képernyőelem látható vagy nem látható feltételének beállítása.

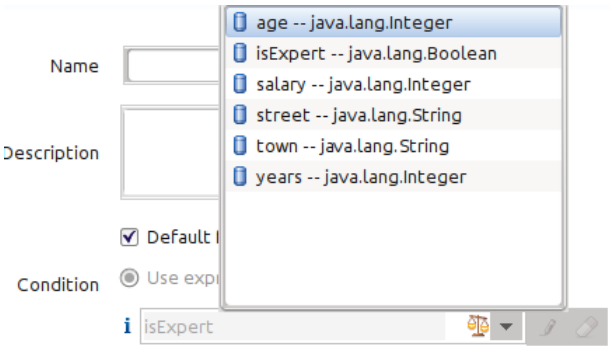



5.14. ábra. A crayon icon jobbra, a dropbox mellett van közvetlenül

A ceruza ikonra kattintva használhatjuk a kifejezés szerkesztőt (5.15. ábra).




5.15. ábra. A kifejezés szerkesztő ablak




5.16. ábra. A dropdown vezérlőn keresztül a változók elérhetőek

Az ablak bal oldalán különféle kifejezés típusokat választhatunk ki:

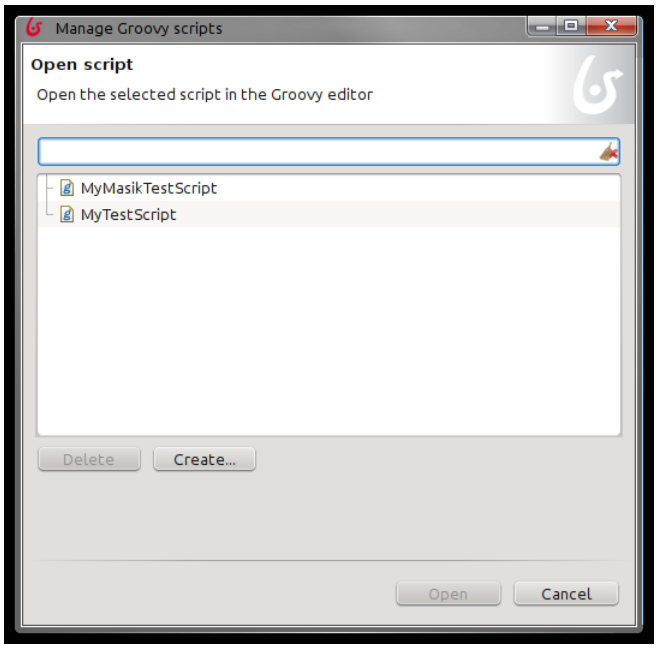
- *Comparison* (összehasonlítás): A változókra épülve olyan kifejezések készíthetőek, amikben a !, ==, !=, <, >, <= vagy >= relációs jelek (ahogy az a Java nyelvben is értelmezett) használhatóak, eredménye természetesen egy logikai érték lesz.
- *Constant* (állandó): Egy konstans érték, ami a kifejezés eredménye is lesz.
- *Parameter* (paraméter): Egy paraméter értéke.
- *Script* (egy Groovy script, 5.15. ábra): A legtöbb lehetőséget adó eset, ahol egy teljes script program is futhat, annak utolsó sora az érték, amit visszakapunk. Itt is mindig vigyázzunk a helyes visszatérési típusra.
- *Variable* (egy változó): Egy változó lesz a kifejezés értéke. Ezt a 5.16. ábrán látható módon is elérhetjük.

A 5.15. ábra jobb oldalán van a *Categories* választási lehetőség. Itt nagyon sok előre elkészíttet script van, amit a mi editorunkban is újra felhasználhatunk. Például a *Number* kategóriából választva 2 elemet, máris keletkezett egy 2 soros script, ami a példa workflow belső *salary* változóját használja:

```
org.codehaus.groovy.runtime.DefaultGroovyMethods.abs(salary)
org.codehaus.groovy.runtime.DefaultGroovyMethods.compareTo(salary,2500000)
```

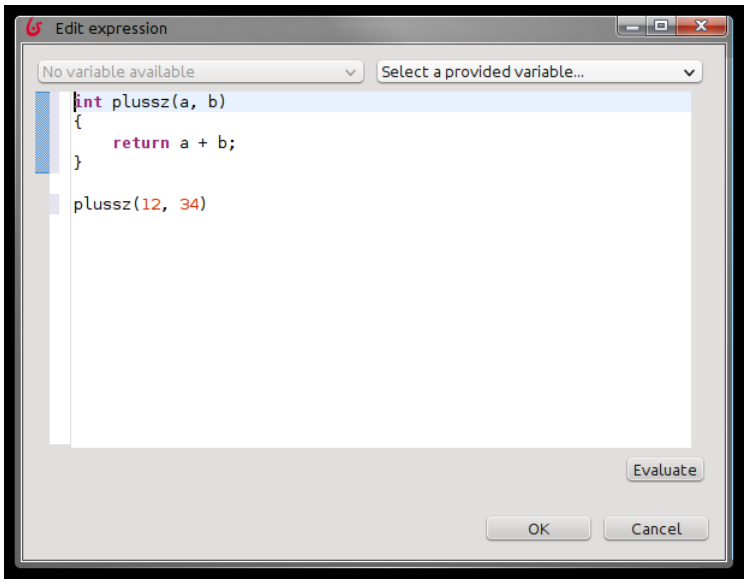
### A Groovy scriptek készítése és menedzselése

A Groovy scriptek mindig valamilyen névvel érhetőek el, azokat természetesen a Bonita Stúdióban külön is létre lehet hozni. Erre szolgál a *Development* főmenü *Manage Groovy scripts...* almenüje, amit kiválasztva az 5.17. ábra ablaka jön elő. Itt előtte már elkészítettünk 2 teszt scriptet, ezeket szerkesztésre ki is választhatnánk, de a *Create...* gombbal újat is létrehozhatunk.




5.17. ábra. A scriptek elérése és kezelése

A 5.18. ábra a Groovy editort mutatja, ide egy teljes értékű Groovy program írható. Az elérhető, rendszer által szolgáltatott változók gyors elérését segíti a jobb oldali dropdown box.




5.18. ábra. A *MyTestScript* editálása

A script az *Evaluate* gomb segítségével tesztelhető, amennyiben megnyomjuk, úgy esetünkben *46*-ot ad vissza. Fontos megérteni, hogy a script készítése során tudunk a workflow belső és mező változóira is hivatkozni. Ezen felül a következő előre megadott változókat szolgáltatja a Bonita:

- *activityInstanceId*: Az ACTIVITY (a Step vagy Subprocess) belső azonosítója
- *loggedUserId*: A bejelentkezett user, aki éppen az activity-t is végrehajtja
- *processDefinitionId*: A process definíció azonosítója
- *processInstanceId*: A process példány azonosítója
- *rootProcessInstanceId*: Amikor egy subprocess fut, akkor annak a szülő process azonosítója.

A fentiek az *apiAccessor* változón keresztül is elérhetőek. Példa:

```
apiAccessor.processAPI.getgetActivityInstance(activityInstanceId);
```

## A változók használata

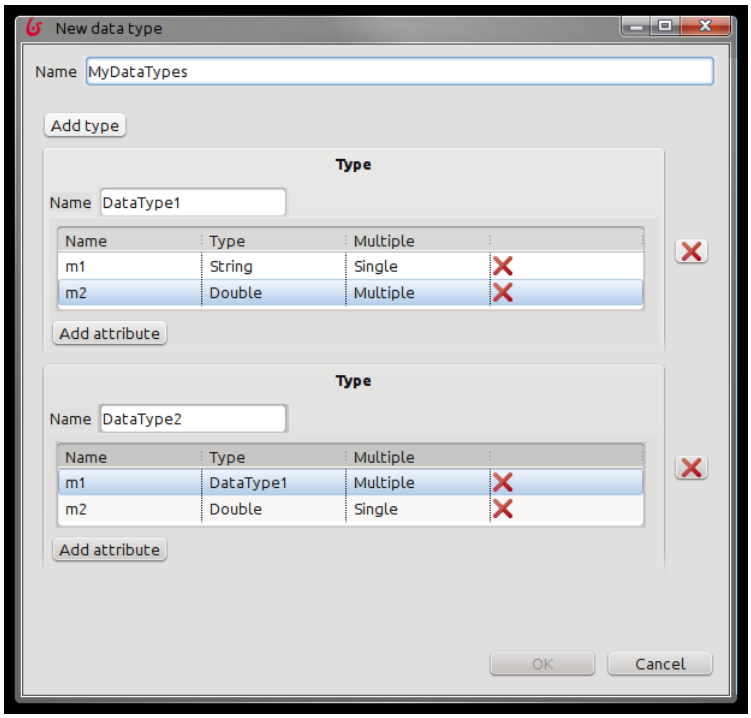
Egy változó létrehozása egy név és egy (Java) típus megadását jelenti. Opcionálisan megadható egy alapértelmezett érték vagy értéklista, így a változó létrejöttekor biztosíthatjuk annak értelmes kezdőértékét.

### Adattípusok

A Bonita előre definiált típusai a következőek: boolean, date, integer, long, double, text, Java object, XML schema (XSD) és option list. Minden változóra mondhatjuk, hogy *multiple*, amely esetben a megadott típusok kollekciója (Java *List* vagy *Map*) lesz a változó, amiket az 5.19. ábrán mutatott módon tudunk létrehozni (elérése: *General* → *Data* fül).

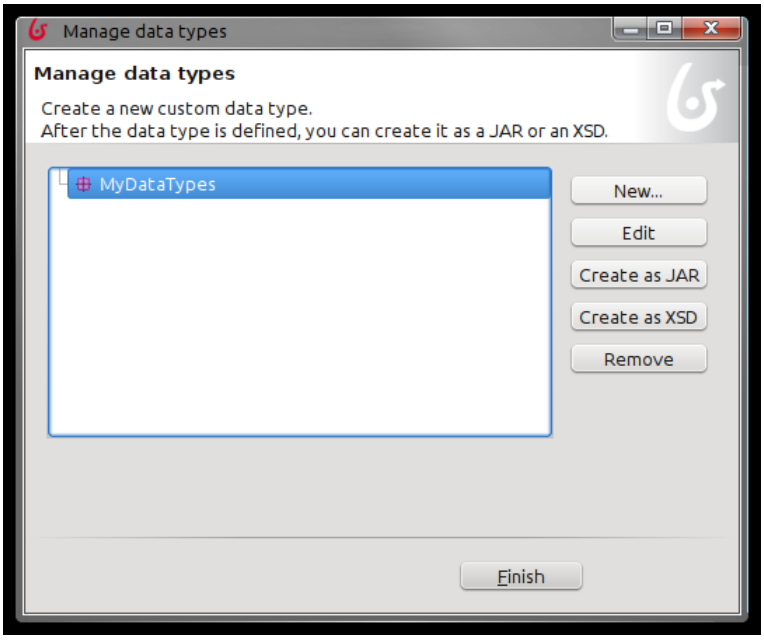

5.19. ábra. Workflow változók létrehozása

Sokszor előfordul, hogy a workflow adatmodellje csak nehézkesen alakítható ki a beépített adattípusokból, ezért van arra lehetőség, hogy a *Development* → *Data types* menü segítségével új, komplexebb típusokat is létrehozhassunk. Ezeket utána egy *jar* (java könyvtár) vagy *XSD* (XML séma) fájl fogja tárolni, akár több definíciót is egyben. Az 5.20. ábra mutatja azt az ablakot, ahol *MyDataTypes* néven (ez lesz a jar vagy XSD neve) éppen 2 új típust (class-t) hozunk létre: *DataType1* és *DataType2*. A 2. class egyik mezőjénél már fel is használtuk az első class-t. Az *Add type* gombbal bármennyi új típus létrehozható lenne, mindegyik ugyanabba a *MyDataTypes* targetbe kerülne. Az *Add type* gomb pedig egy új mező felvételét segíti. A piros *X* jelekkel törölni lehet.



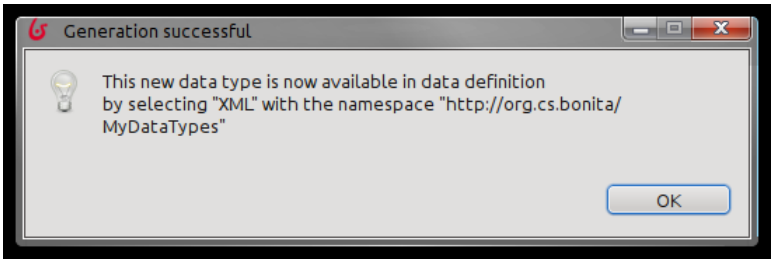

5.20. ábra. Összetett adatszerkezet létrehozása

Amennyiben kész vagyunk, nyomjuk meg az *OK* gombot, amire a 5.21. ábra ablaka bukkan fel. Itt lehetőségünk van JAR vagy XSD (vagy mindkettő) elkészítésére. Nyomjuk meg a *Create as XSD gombot*, akkor meg fogja kérdezni a *targetnamespace*-t, ami most *org.cs.bonita* lett.



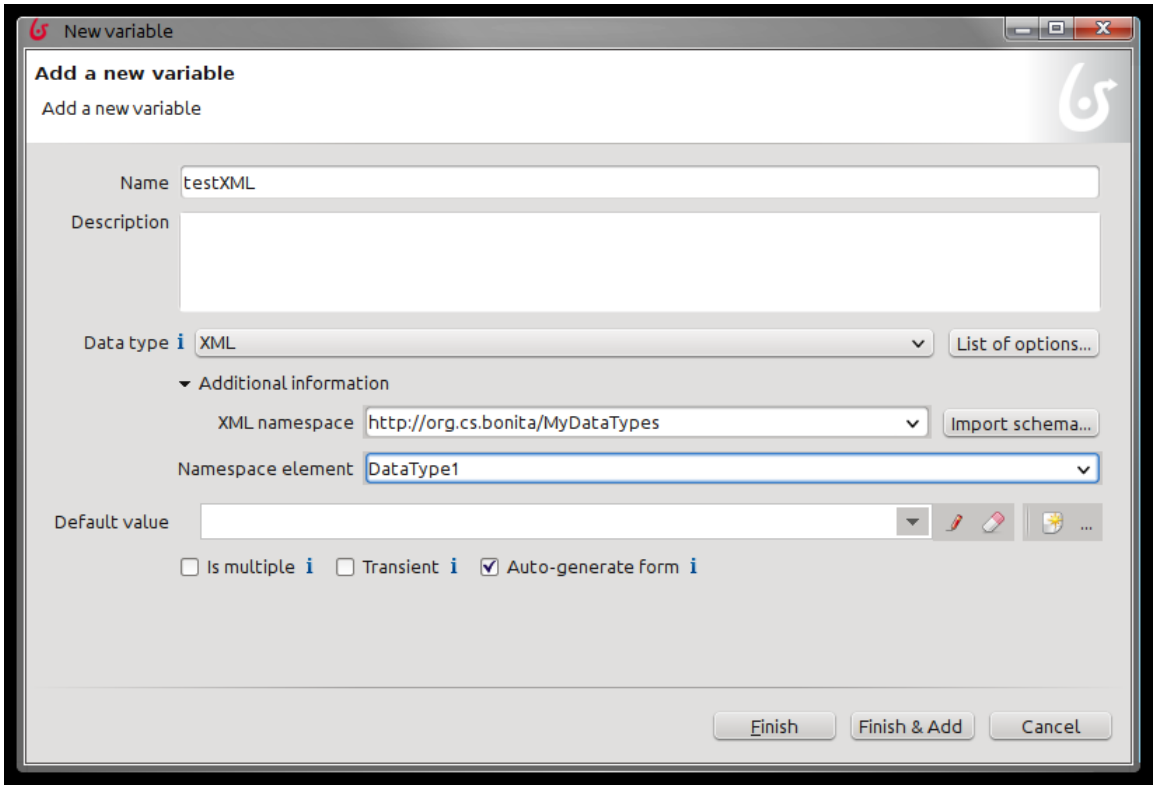

5.21. ábra. Az *MyDataTypes* alapján jar vagy XSD létrehozása

A *Finish* gombra a 5.22. ábra ablaka tájékoztat, hogy sikeresen elkészült az XSD és úgy lehet majd használni, hogy 5.19. ábra által mutatott helyen az új változó típusát XML-re állítjuk.



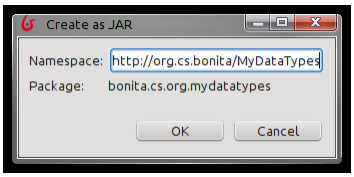

5.22. ábra. Elkészült az XSD

Próbáljuk is ki az új változó létrehozását, amit a 5.23. ábra mutat. A változó neve *testXML*, típusa *XML*. Itt meg kell adnunk, hogy mi a *namespace*, illetve melyik *XSD* típusra gondoltunk (mindegyik Java class egy XSD-t jelent).

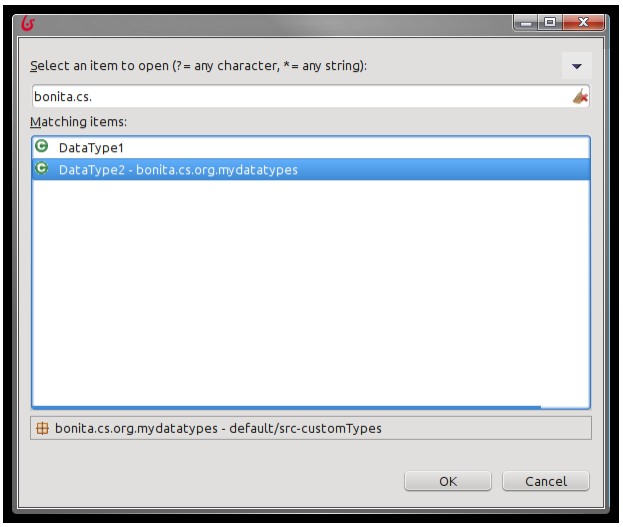



5.23. ábra. Egy új, XML típusú változó létrehozása

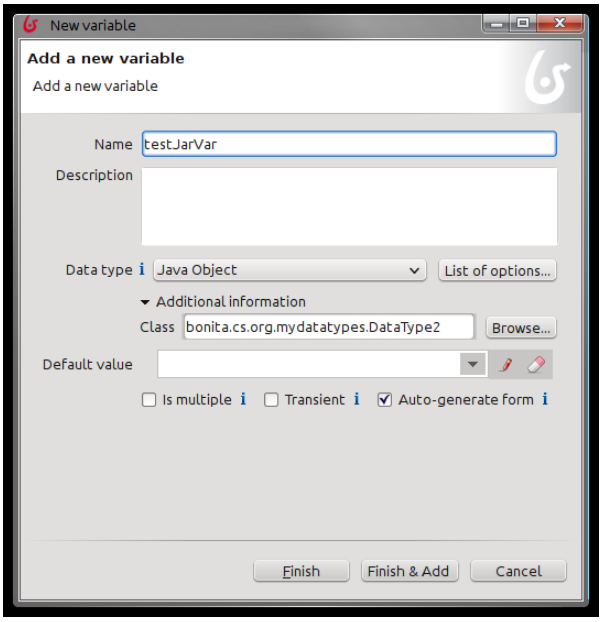
Amennyiben nem indokolt az XML, úgy sokkal jobb út egy Java class létrehozása, ekkor a generálás ezeket egy *jar* fájlba teszi, esetünkben a benne lévő 2 class a *bonita.cs.org.mydatatypes* csomagba fog kerülni (5.24. ábra).




5.24. ábra. JAR fájl létrehozása




5.25. ábra. A *testJarVar* változó most *DataType2* típusú lesz




5.26. ábra. A *testJarVar* változóra ki lett választva a típus

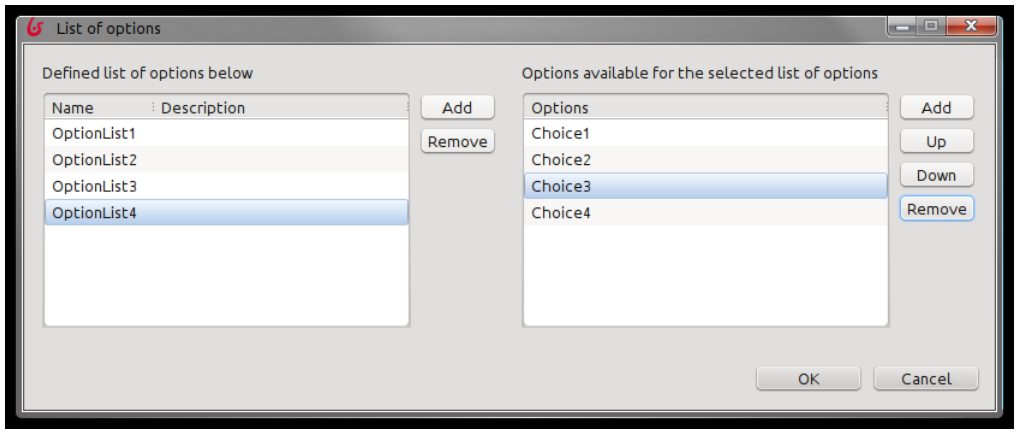
Tesztelésként vegyünk fel egy *testJarVar* változót (5.25. és 5.26. ábrák).

### Egy egyszerű adattípusú változó létrehozása

Röviden tekintsük át még egyszer a változó létrehozásának lépéseit, kis magyarázatokkal kiegészítve. Válasszuk ki a pool-t (azaz a processt) vagy a step-et, ami egyben a most létrehozandó változók hatáskörét (globális a workflow-ra, csak a step-ben látszik) is megadja. A *General* → *Data* fülre álljunk rá és elkezdhetjük a változók szerkesztését. Komoly célú alkalmazások esetén szinte kötelező a dokumentációs célokat szolgáló *Description* kitöltése. Az adattípus kiválasztását már sokszor láttuk, a *multiple* pipa szerepét is ismertettük már. Egy stephez rendelt változóra mondhatjuk, hogy *Transient*, ekkor ez nem fog az adatbázisba perzisztálódni. Ezek tipikusan segédváltozók. A saját komplex típus használatának bemutatásakor láttuk, hogy egy Java *Object* típusú változó létrehozása szükséges, amikor tetszőleges Java class-t szeretnénk típusként megadni.

### Opciók listája változó

Néha arra van szükségünk, hogy egy előre rögzített értékkészlettel rendelkező legyen a típusunk. Ekkor a 5.26. ábrán is látható *List of options...* gombot nyomjuk meg, amire a 5.27. ábra ablaka jön elő. A bal oldali lista minden tagja egy változó típusa lehet, aminek az értékeit a jobb oldal mutatja az éppen kiválasztott bal oldali elemre.



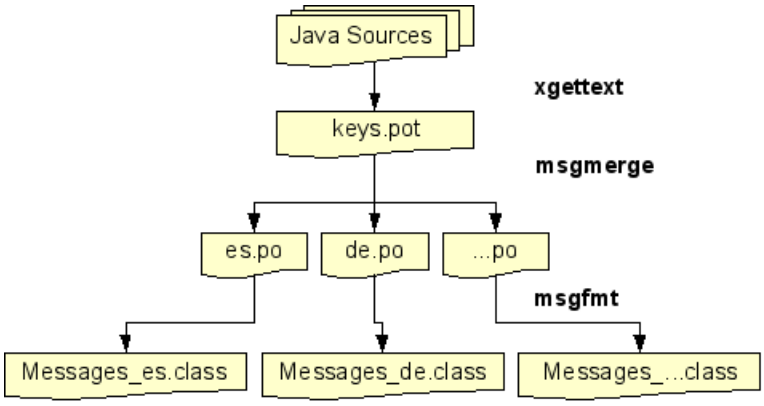

5.27. ábra. Opciók listája

Amennyiben a használt értékek halmaza gyakran változik, úgy ez a módszer ellenjavallt, esetleg egy konnektor érdemes használni ekkor az adatok megszerzéséhez.

## Több nyelv használata

A Bonita a *GNU gettext* (`https://www.gnu.org/software/gettext/`) csomagot használja a Portál I18N megvalósításhoz, ezért érdemes ezt a témát kicsit körbejárni. Sok modern Java program is ezt a régi jól bevált módszert fedezi fel újra, többek között a Google alkalmazások is. A *gettext* univerzális megoldás, a Linux (és általában a UNIX) is ezt alkalmazza az I18N megvalósításhoz. A

csomag az ismertebb nyelvi környezetekből (C/C++, Java, C#, PHP, ...) elérhető, de itt most csak a Jávából való használatot mutatjuk be, azt viszont kétféleképpen is. Megjegyezzük, hogy a C# környezetben is érdemes használni, mert ezzel valószínűleg ott is hatékonyabbá válik a nyelvfüggő szövegek kezelése.




5.28. ábra. A gettext *po* fájlok és Java *ResourceBundle* osztályok előállítása

### A po fájlok előállítása

A *po*[3] fájlok hordozzák azt az információt, hogy egy megadott nyelven megjelenő szöveget milyen célnyelvi szövegre fordítsunk. Amit fordítunk, az a kulcs (*msgid*, és érdemes az angol nyelvet választani), amire fordítjuk az a message (*msgstr*) a *po* fájlon belül. A 5.28. ábra azt a folyamatot ábrázolja, ahogy előállítjuk az egyes nyelvekhez a *po* fájlokat, sőt még azt is mutatja, hogy ezekből a hagyományos Java *MessageBundle* osztályok is létrehozhatóak. Nézzünk egy példát, hogy mindezt könnyebben megérthessük! Az 5-1. Programlista 3 Java class forráskódját mutatja: *Test1*, *Test2* és *TestMain*.

5-1. Programlista: 3 Java class szövegekkel

```
1  package org.cs.test;
2
3  public class Test1
4  {
5          public static void ir()
6          {
7                  System.out.println("This is the first String");
8                  System.out.println("This is the second String");
9                  System.out.println("This is the third String");
10         }
11 }
12
13 package org.cs.test;
14
15 public class Test2
16 {
17         public static void ir()
18         {
19                 System.out.println("This is the 1st String");
```

[3] po=portable object

```
20                  System.out.println("This is the 2nd String");
21                  System.out.println("This is the 3rd String");
22          }
23  }
24
25  package org.cs.test;
26
27  public class TestMain
28  {
29          public static void main(String[] args)
30          {
31                  System.out.println("The test will be started now...");
32
33                  Test1.ir();
34                  Test2.ir();
35          }
36
37  }
```

A program csak annyit csinál, hogy a megadott szövegeket kiírja a képernyőre. Készítsük el ennek az angol és magyar *I18N*-es változatát. Ehhez elő kell állítanunk ezt a 2 *po* fájlt, amit a *gettext* csomag *xgettext* utility programjával tehetünk meg az alábbi módon. Első lépés a kulcsok kigyűjtése egy *pot*[4] fájlba, ami ezzel a paranccsal a keys.pot fájlba keletkezik meg a 3 Java forráskódra épülve:

```
xgettext -a -ktrc:1c,2 -ktrnc:1c,2,3 -ktr -kmarktr -ktrn:1,2 -o keys.pot *.java
```

Érdemes meggondolni, hogy ez az egyik legnagyszerűbb tulajdonsága a *gettext* csomagnak, nem kell a kulcsokat manuálisan kigyűjteni. Amikor sok Java fájl van, különböző almappákban, akkor ezt a kissé módosított parancsot is bevethetjük, ahol a *find* parancs siet a segítségünkre:

```
xgettext -ktrc -ktr -kmarktr -ktrn:1,2 -o keys.pot $(find . -name "*.java")
```

A következő lépés már a *po* fájlok legenerálása. A magyar esetre ezt a parancsot használtuk:

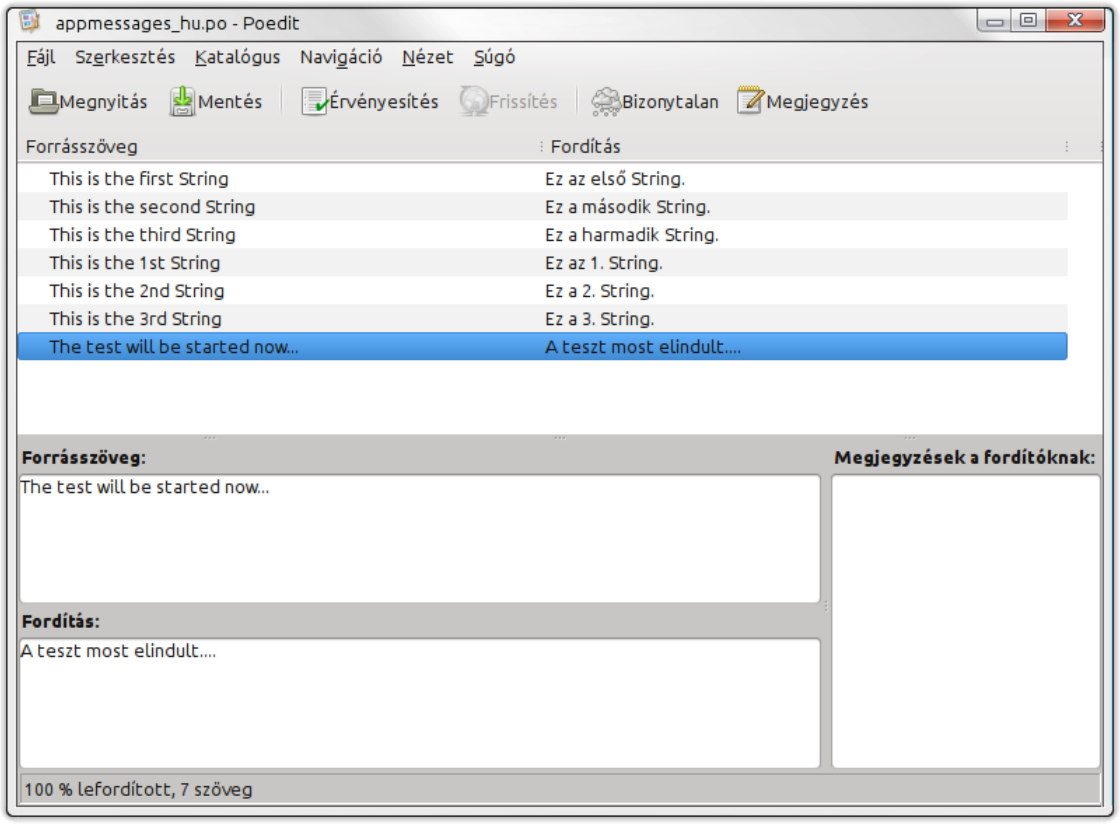
```
msginit -i keys.pot -o appmessages_hu.po -l hu_HU
```

Az angolra pedig ezt:

```
msginit -i keys.pot -o appmessages_en.po -l en_EN
```

A poedit grafikus programmal szerkeszthetjük a *po* fájlt, az 5.29. ábra például azt mutatja, hogy a magyar szövegeket tesszük hozzá a legenerált kulcsértékek mellé.

[4]pot=portable object template




5.29. ábra. poedit az *appmessages_hu.po* szerkesztésére

Az 5-2. Programlista kialakított *appmessages_hu.po* fájlt tartalmát listázza.

5-2. Programlista: A magyar po fájl tartalma

```
# Hungarian translations for PACKAGE package.
# Copyright (C) 2013 THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# Nyiri Imre <imre.nyiri@gmail.com>, 2013.
#
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2013-11-23 17:25+0100\n"
"PO-Revision-Date: 2013-11-23 17:41+0100\n"
"Last-Translator: Nyiri Imre <imre.nyiri@gmail.com>\n"
"Language-Team: Hungarian\n"
"Language: hu\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Plural-Forms: nplurals=2; plural=(n != 1);\n"
"X-Generator: Poedit 1.5.4\n"

#: /home/inyiri/workspace/VaadinSSO/TestGettextProject/src/org/cs/test/Test1.java:8
msgid "This is the first String"
msgstr "Ez az első String."
```

```
#: /home/inyiri/workspace/VaadinSSO/TestGettextProject/src/org/cs/test/Test1.java:9
msgid "This is the second String"
msgstr "Ez a második String."

#: /home/inyiri/workspace/VaadinSSO/TestGettextProject/src/org/cs/test/Test1.java:10
msgid "This is the third String"
msgstr "Ez a harmadik String."

#: /home/inyiri/workspace/VaadinSSO/TestGettextProject/src/org/cs/test/Test2.java:7
msgid "This is the 1st String"
msgstr "Ez az 1. String."

#: /home/inyiri/workspace/VaadinSSO/TestGettextProject/src/org/cs/test/Test2.java:8
msgid "This is the 2nd String"
msgstr "Ez a 2. String."

#: /home/inyiri/workspace/VaadinSSO/TestGettextProject/src/org/cs/test/Test2.java:9
msgid "This is the 3rd String"
msgstr "Ez a 3. String."

#: /home/inyiri/workspace/VaadinSSO/TestGettextProject/src/org/cs/test/TestMain.java:7
msgid "The test will be started now..."
msgstr "A teszt most elindult...."
```

Tudjuk mostanra, hogy az egyes nyelvekhez tartozó *po* fájlt milyen módon tudjuk előállítani a forráskód szövegéből. A kérdés az, hogy mit tudunk tenni, ha közben a forráskódba új szövegek kerülnek be? A *main()* metódus utolsó soraként ezt vettük fel, de ez még nincs a *po* fájlban:

```
System.out.println(tr2("The test is finished."));
```

A megoldás az, hogy ismét futtatjuk a *xgettext* parancsot, előállítva ezzel a *keys.pot* fájl egy időszerűsített változatát. Az *msmerge* gettext utility pedig a meglévő *po* fájlhoz fésüli az új sorokat, amit a magyar *po* fájlra így tettünk meg:

```
msgmerge –U appmessages_hu.po keys.pot
```

Az angolra pedig hasonlóan:

```
msgmerge –U appmessages_en.po keys.pot
```

Az eddigiek során megtanultuk a *po* fájlok létrehozásának hatékony módját, a következő 2 pontban ennek két különböző, de hasonló célú használatát mutatjuk be.

### A *ResourceBundle* class használata

A Java hagyományosan a *ResourceBundle* class segítségével kezeli a nyelvi szövegeket, ezért a most bemutatott módszer azért nagyon értékes, mert azt oldja meg, hogy a po fájlból ilyen osztályokat tudjunk előállítani. Erre szolgál a gettext *msgfmt* parancsa, amit így használtunk:

```
msgfmt ––java –d . –r myapp.i18n.Messages –l hu appmessages_hu.po
msgfmt ––java –d . –r myapp.i18n.Messages –l en appmessages_en.po
```

Az eredmény az lett, hogy a *myapp/i18n* könyvárba létrejött a magyar és angol lefordított *ResourceBundle* class: *Messages_hu.class* és *Messages_en.class*. Mindjárt betettük őket egy jar fájlba és az Eclipse build path-hoz adtuk. Ezen felül még két jar is ott van: *libintl.jar* és *gettext-commons-0.9.8.jar*, de ezek a 2. megoldáshoz kellenek csak.

```
jar cf po-messages-bundle.jar myapp/*
```

Az 5-3. Programlista mutatja ezen módszer használatát. A kód 8-18 sorai között implementált *tr1()* metódus a 14. sorban látható lépéssel hozza létre a *ResourceBundle* objektumot, ahol a *getBundle()* paramétere a *baseName*, azaz az a név, amit az *msgfmt* parancsban is megadtunk. Ezután már ismerős pályán vagyunk, hiszen Java programozóként az *rb* objektumot már bizonyára régóta megtanultuk használni. A program futtatása után láthatjuk, hogy az tényleg a kívánt módon működik, az alapértelmezett *Locale*-tól függően, hogy a magyar, hol az angol szövegeket írja ki a képernyőre.

5-3. Programlista: A ResourceBundle alapú demó

```
1  package org.cs.test;
2
3  import java.util.Locale;
4  import java.util.ResourceBundle;
5
6  public class TestMain
7  {
8          public static String tr1(String s)
9          {
10                 Locale locale = new Locale("hu");
11                 //Locale.setDefault( Locale.ENGLISH );
12                 Locale.setDefault( locale );
13
14                 ResourceBundle rb = ResourceBundle.getBundle("myapp.i18n.Messages");
15                 String retM = rb.getString( s );
16
17                 return retM;
18         }
19
20         public static void main(String[] args)
21         {
22                 System.out.println(tr1("The test will be started now..."));
23
24                 Test1.ir();
25                 Test2.ir();
26
27                 System.out.println(tr1("The test is finished."));
28         }
29 }
```

### A Google *I18n* class használata

A *gettext-commons-0.9.8.jar* (letölthető innen: `http://code.google.com/p/gettext-commons/`) könyvtár használatával egy másik, de hasonló lehetőség is van a generált *ResourceBundle* class-ok használatára, ezt példázza az 5-4. Programlista.

5-4. Programlista: A Google I18n class alapú demó

```
1  package org.cs.test;
2
3  import java.util.Locale;
4  import java.util.ResourceBundle;
5  import org.xnap.commons.i18n.I18n;
6  import org.xnap.commons.i18n.I18nFactory;
```

```
7  import myapp.i18n.Messages_hu;
8
9  public class TestMain
10 {
11         public static String tr2(String s)
12         {
13                 Locale locale = new Locale("hu");
14                 //Locale.setDefault( Locale.ENGLISH );
15                 Locale.setDefault( locale );
16
17                 I18n i18n = I18nFactory.getI18n(TestMain.class, "myapp.i18n.Messages", locale);
18                 String ret = i18n.tr(s);
19
20                 return ret;
21         }
22
23         public static void main(String[] args)
24         {
25                 System.out.println(tr2("The test will be started now..."));
26
27                 Test1.ir();
28                 Test2.ir();
29
30                 System.out.println(tr2("The test is finished."));
31         }
32 }
```

A fő különbség az, hogy most a 17. sorban egy *I18n* objektumot hozunk létre. A használata rendkívül egyszerű. Érdekes lehetőséget látunk az 5-5. Programlistán, ugyanis ez a csomag lehetővé tesz nekünk egy listenert implementálni, ami a nyelvváltáskor fut le, így a nyelvfüggő Stringek itt egy helyen átírhatóak a *localeChanged()* eseménykezelő segítségével.

5-5. Programlista: I18n és a listener használat

```
public class LocaleChangeAwareClass implements LocaleChangeListener
{
        private static I18n i18n = I18nFactory.getI18n(LocaleChangeAwareClass.class);

        String localizedString;

        public LocaleChangeAwareClass()
        {
                localizedString = i18n.tr("Hello, World");
                I18nManager.getInstance().addWeakLocaleChangeListener(this);
        }

        public void localeChanged(LocaleChangeEvent event)
        {
                // update strings
                localizedString = i18n.tr("Hello, World");
                ...
        }
}
```

Akit a *gettext* csomag mélyebben érdekel vagy más nyelvekhez is szeretné használni, annak ajánljuk a honlapon lévő dokumentáció tanulmányozását: `http://www.gnu.org/software/gettext/manual/gettext.html`

# 6. Az események használata a BPMN modellekben

A folyamatok modellezésének és implementálásának fontos része az események létrehozása, postázása és lekezelése. Ezzel menedzselhetjük a workflow szekvenciális lépései során keletkezett különleges vagy hibás helyzeteket, másfelől alkalmas a különböző folyamatok közötti asszinkron kommunikáció megszervezésére is. Az eseményekről általában az Informatikai Navigátor 5. számából tudhatunk meg többet, itt most a gyakorlati használatát mutatjuk be.

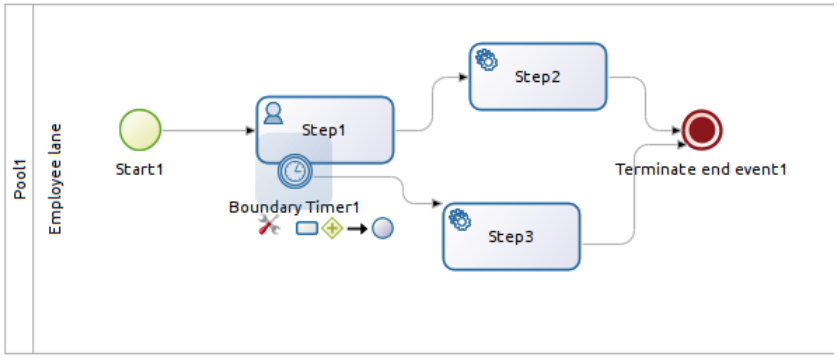
Háromfajta esemény létezik a workflow lefutása szerint vizsgálva: *kezdő* (1 vonalas, vékony karika), *záró* (1 vonalas, vastagított karika) és *köztes* (dupla körös karika). Másik felosztásban van a küldő (Sender, azaz *throw* event) és a fogadó (Receive, azaz *catch event*). A throw event-et mindig egy beszínezett, a catch event-et pedig egy körvonalas ikon reprezentál a BPMN diagramon. Befejezésül emlékezzünk arra is, hogy az eseményeket vagy egy átmenetre (transition), vagy egy taszkra kötve tehetjük be a folyamat tervünkbe. Ez utóbbit határ (*boundary*) eseménynek nevezzük és értelmezése mindig a feladat kontextusában lehetséges.

## A Timer határesemény

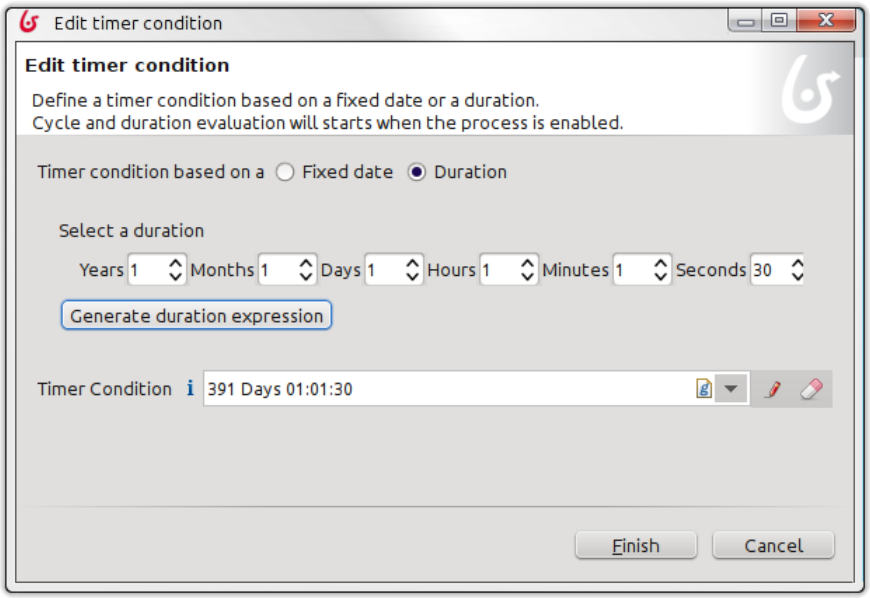
### A konfiguráció

Első példánkban tekintsük azt az esetet, amikor van egy taszkunk és onnan akkor is ki szeretnénk lépni, ha egy bizonyos idő letelt vagy egy időpont elérkezett. Példánkat a (6.1. ábra *Step1* taszkja mutatja. A kis timer dupla karikát (2 karika és körvonalas ikon, mert köztes és catch esemény, mert elkapja a „lejárt az időt") a step helyi *Add a boundary event...* ikonára kattintva választottuk ki. A timer ikonra kattintva a beállításoknál kétféleképpen is megadhatjuk a *Timer Condition*-t:

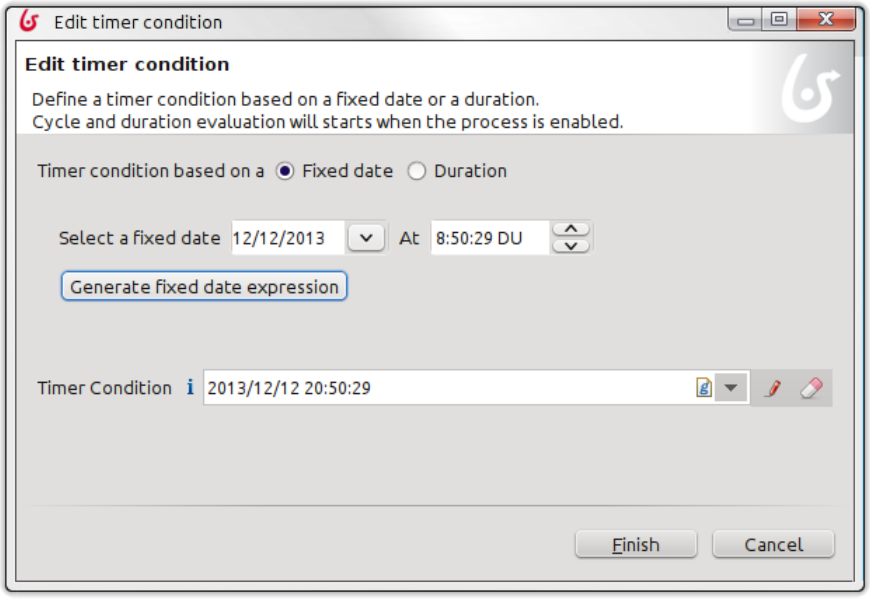
- *Fixed date* (6.2. ábra)
- *Duration* (6.3. ábra)




6.1. ábra. Egy munkafolyamat Timer határeseménnyel a *Step1* lépésnél




6.2. ábra. Timer határesemény (Timer Boundary Event) konfigurálása - *Duration*




6.3. ábra. Timer határesemény (Timer Boundary Event)konfigurálása - *Fixed date*

A *Duration* azt mondja meg, hogy mennyi ezredmásodperc múlva csattanjon el a timer. Például a 6.3. ábrán láthatjuk, hogy 1 év + 1 hónap + 1 nap + 1 óra + 1 perc + 30 másodperc múlva kell az órának jelet adnia. Ez ezredmásodpercben *33786090000L* érték, amit így lehet kiszámítani:

```
groovy -e "print 1000*(391*24*60*60+60*60+60+30)"
```

A 30 másodperc így nézne ki a *Timer Condition* sorban: *00:00:30*, ami *30000L* értékű és típusa *java.lang.Long*. A beállított érték érvényesüléséhez mindig meg kell nyomni a *Generate duration expression* gombot.

A 6.3. ábra azt az esetet mutatja, amikor fix dátumot adunk meg. Ekkor ezzel a kóddal generálódik le a fenti példára az időpont (a hónap 0-tól kezdődik):

```
Calendar calendar = GregorianCalendar.getInstance();
calendar.set(2013, 11,12, 20, 50, 29);
calendar.getTime();
```

A beállítás értéktípusa ebben az esetben *java.util.Date*. Láthatjuk, hogy mindkét esetben használhatjuk a kifejezés szerkesztőt, azaz a timer idejét dinamikusan is ki tudjuk számítani mielőtt beállítanánk.

### A működés

A workflow befejezése a szokásos módon történik, magyarázatra a Timer ágból kijövő taszk szorul, azonban a működés pont az, mint amit elvárunk. A *Step1* TASK-ból vagy úgy lépünk ki, hogy kompletté tettük, azaz valaki megcsinálta, vagy lejárt az ideje. Az első esetben a *Step2*, míg a másodikban a *Step3* felé megy tovább a vezérlés. A timer ágán való haladást a *Step1* abort vagy exception ágának is neveti a BPMN, a Bonita portálon is ilyenkor a taszk *aborted* jelzővel van ellátva.

## Signal esemény

### Áttekintés

Egy *Signal Event* képes szétszórni (broadcast), fogadni egy eseményt (nem kíséri adat) egy folyamaton belül, illetve egy másik workflow-val kommunikálva is. Egy egyszerű throw signal jelét többen is fogadhatják, azaz több catch signal is csatlakozhat rá. Egy kiadott jel nem rendelkezik semmilyen targettel, aki rácsatlakozik, az képes fogadni a jelet, amennyiben az el lett küldve. A következő 5 fajta signal létezik:

1. *Intermediate Throw Signal Event* (köztes esemény, ami dob egy jelet), ami broadcast módon dob egy jelet, amikor a process hozzáérkezik.

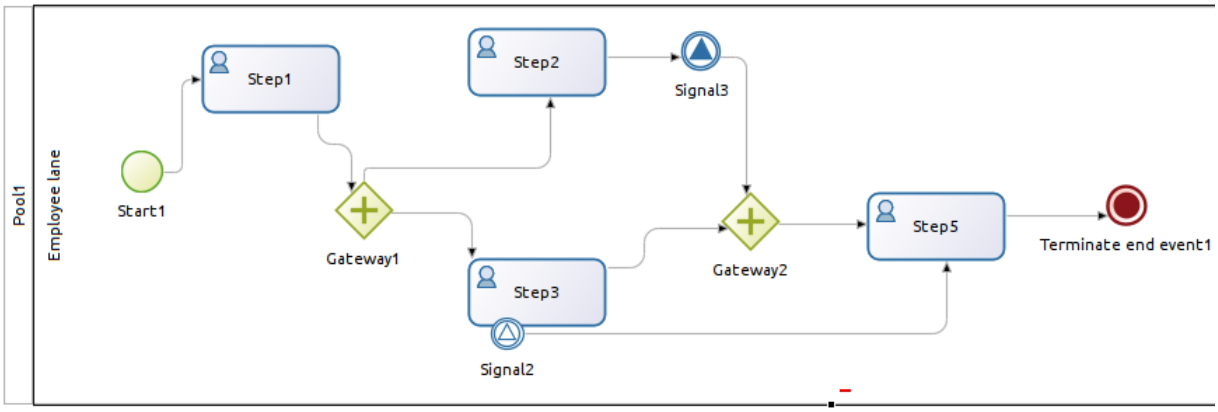
2. *Intermediate Catch Signal Event* (köztes esemény, ami fogad egy jelet), ami várakozik mindaddig, amíg meg nem érkezik az a jel, amire várakozik.

3. *Start Signal Event*: várakozik a megadott jelre, majd elindít egy process-t.

4. *End Signal Event*: befejezi azon a ponton a process-t és eldob egy jelet.

5. *Boundary Signal Event*: Elkapja a jelét és abortálja azt a step-et, amire csatlakozik.

### Egy folyamaton belüli példa




6.4. ábra. A Signal használata egy folyamaton belül

A 6.4. ábra egy példa workflow lefutást mutat, ahol mindegyik taszk humán. Az egyes feladatok konkrét tevékenysége most nem érdekes, csupán a folyamat lefutása fontos a példánk szempontjából. A *Signal3* egy intermediate throw signal, azaz köztes szignál esemény, ami broadcastolja a jelét (dupla körben befestett ikon). Egy ilyen event konfigurálása nagyon egyszerű, mindössze a következő adatokat kell megadni a *General→General* property fülnél:

- *Name*: A szignál neve.
- *Description*: A szignál dokumentációs célú leírása, ami a *Diagram→Generate documentation...* funkció lefuttatásakor szintén bekerül a Bonita Stúdió által generált dokumentációba.
- *Signal*: Ez egy tetszőleges karaktersorozat, ami a szignál jelet reprezentálja. Esetünkben ezt adtuk meg: *ALMA*. Erre tudnak feliratkozni catch signal eventek.

Ahogy azt már említettük a szignálok nem hordoznak kísérő adatokat, így ilyeneket nem is tudunk megadni. Most nézzük meg a fenti jelkiadó signal párját, ami arra fel tud iratkozni. A példában ez most a *Signal2* lett, ami egyben határesemény is, azaz a *Step3* lépéshez kötődik. Ennek konfigurációja ugyanazt a 3 mezőt kéri be, de a *Signal* adat most egy dropdown vezérlőből kiválasztható érték lesz, hiszen a fejlesztőkörnyezet tudja a már korábban megadott kiadható signal jeleket (string-eket). Itt az előzőleg definiált *ALMA* jelet választottuk ki. Ezt másképpen úgy is nevezzük, hogy a *Signal2* catch event feliratkozott az *ALMA* jelre, amit bárhonnan kaphat, de amikor beérkezik hozzá, akkor esetünkben megszakítja a *Step3* lépést és az általa megadott exception path-on megy tovább a folyamat, ami a *Step5* lépés felé tart.

A jobb megértés érdekében nézzük meg a folyamat 2 lefutási lehetőségét! A zárójelbe tett step-ek azt jelentik, hogy egyszerre ezek a taszkok léteznek a folyamat példányon belül.

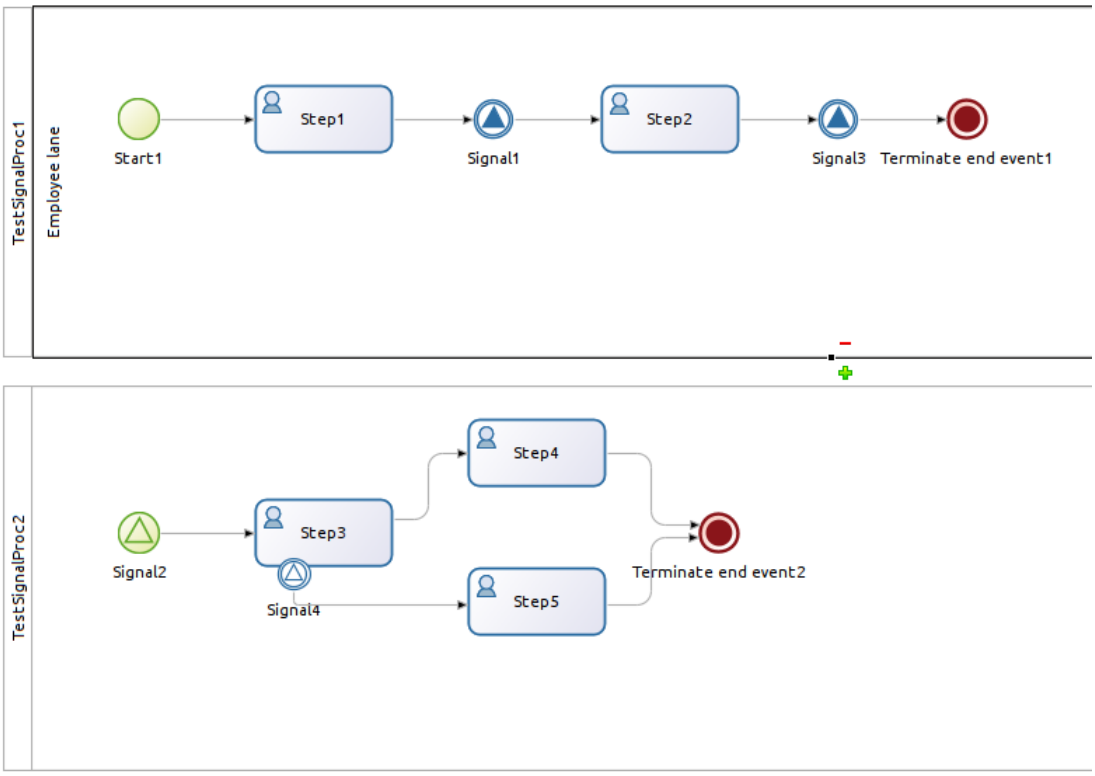
1. Lefutási lehetőség: (*Step1*) és elvégeztük → (*Step2*, *Step3*) és *Step3*-at végeztük el → (*Step2*) és elvégeztük → (*Step5*) és elvégeztük.

2. Lefutási lehetőség: (*Step1*) és elvégeztük → (*Step2*, *Step3*) és *Step*2-öt végeztük el → (*Step5*) és elvégeztük.

Mindkét esetre igaz, hogy elvégeztük a *Step1*-et és a parallel gateway miatt párhuzamosan megkeletkezett a *Step2* és *Step3*. Az 1. lehetőségnél előbb a *Step3*-at csináltuk meg, aminek nyila bemegy a 2. + gateway-be és ott várakozik a szinkronizálásra, azaz a másik ág megérkezésére. Itt megmaradt a *Step2*, ezért azt elvégezve az innen jövő token is megérkezik a kapuhoz, így beérkezvén minden jel, a vezérlés is tovább mehet és létrejön a *Step5*. A 2. lehetőség az, ahol érvényesül a szignál mechanizmus, mert az egyszerre létező *Step2* és *Step3* lépések közül először a *Step2* kerül végrehajtásra és a *Step3*-hoz nem nyúl senki. A *Step2* után azonban van egy throw signal az *ALMA* jellel, amit a *Signal2* boundary catch signal elkap (arra iratkozott fel), megszakítja a taszkot és küld egy tokent a *Step5* felé, így az létre is fog jönni. Közben a 2. + gateway sem fog már várakoztatni, mert már nem kaphat több input jelet. Így itt is eljutottunk a *Step5* lépéshez, azonban ebben a forgatókönyvben a *Step3*-at végül nem kellett elvégezni.

### Folyamatok közötti szignál (1. példa)

A 6.5. ábra 2 folyamat között mutatja a szignálok használatának 2 tipikus példáját.




6.5. ábra. A Signal használata folyamatok között - 1. példa

Az egyes szignálok célja és konfigurálása a következő:

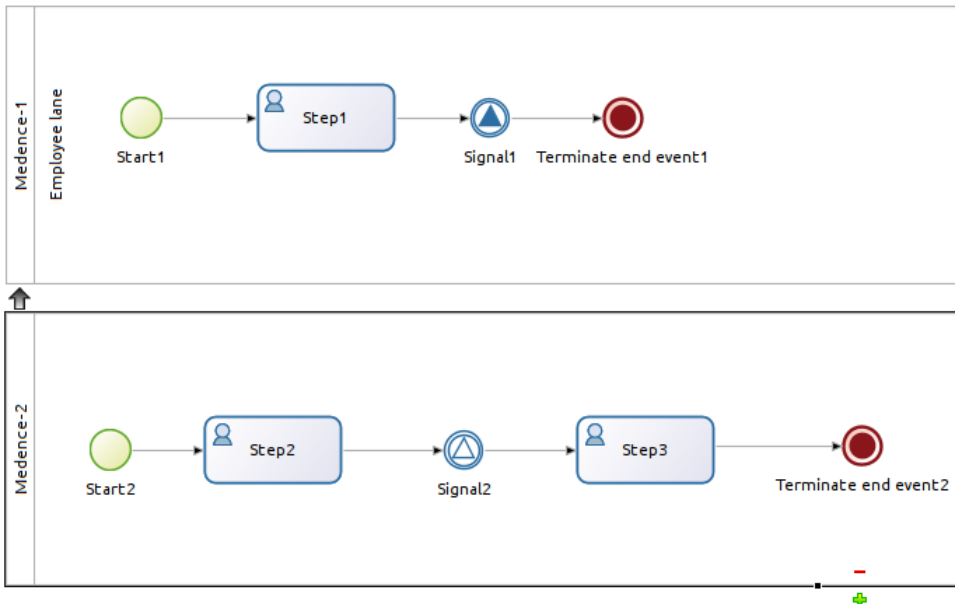
- *Signal1*: Kiad egy *Indulj* jelet, ugyanis ez a szignál érték lett beállítva.
- *Signal2*: Ez már a másik processben van, és hallgatózik az *Indulj* jelre. Ez egy start event, ezért elindítja a *TestSignalProc2* process egy példányát.
- *Signal3*: A *Step2* után kerül sorra és kiad egy *Stop* nevű jelet.
- *Signal4*: A *Step3* határeseménye, ami a *Stop* nevű jelre van feliratkozva.

A 2 folyamat együttes lefutása például történhet ezzel a lépéssorozattal:

1. Elindul a *TestSignalProc1* folyamat, mert kitöltöttük az indító formját.
2. Elvégeztük a *Step1* taszkot, ami után a *Signal1* küld egy *Indulj* broadcast jelet. Megkeletkezik a *Step2* taszk.
3. A *Signal2* fogja az *Indulj* jelet és elindul a *TestSignalProc2* folyamat, létrejön a *Step3* taszk.
4. A feladatkosárban most a *Step2* és *Step3* taszkok vannak.
5. Ha *Step2* kerül végrehajtásra, akkor a *Step3* abortál és a *Step5* taszk jön létre, az első process pedig befejeződik.
6. Ha *Step3* kerül végrehajtásra, akkor megkeletkezik a *Step4* taszk, de mellette még a *Step2* is ott van.

### Folyamatok közötti szignál (2. példa)

A 6.6. ábra egy másik szituációt mutat arra, amikor 2 folyamat között szignállal kommunikálunk.

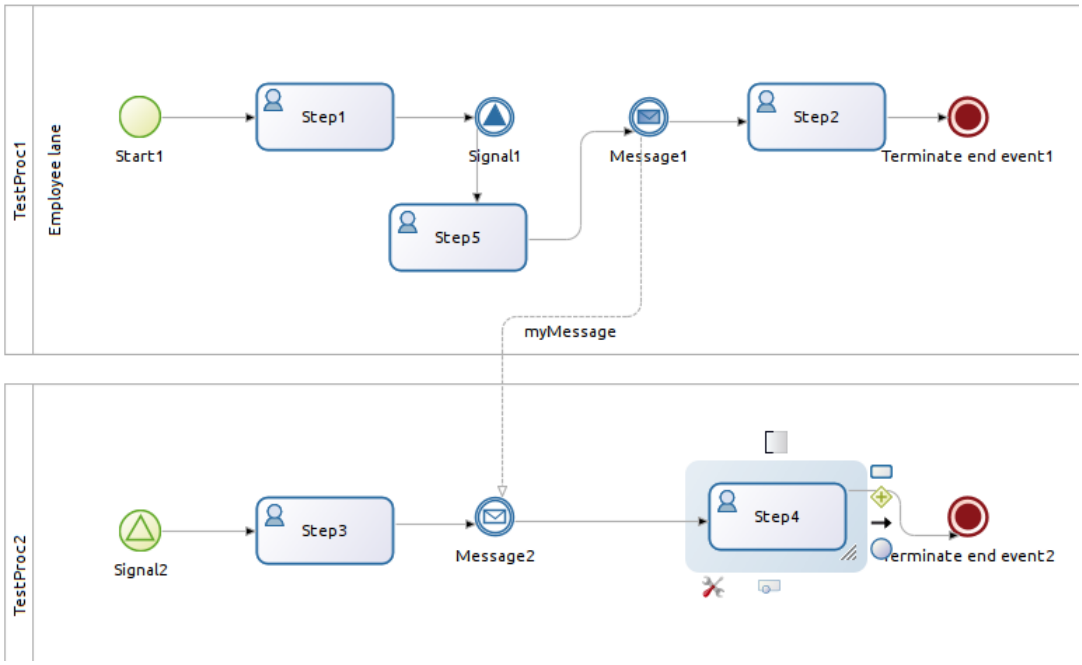



6.6. ábra. A Signal használata folyamatok között - 2. példa

Esetünkben most a *Signal2* ráiratkozott a *Signal1* által kiadott jelre. A *Medence-2* folyamatot indítsuk el például 3 példányban, ez azt fogja jelenteni, hogy a *Step2* feladat 3 példányban lesz a kosárban. Ezután indítsuk el 1 példányban a *Medence-1* folyamatot, így megkeletkezik majd a *Step1* lépés is. Ezután hajtsuk végre egymás után – természetesen tetszőleges sorrendben – a *Step2* feladatokat, így a *Medence-2* mindhárom folyamat példánya a *Signal2* catch event-nél fog várakozni a *Signal1* jelére. Hajtsuk végre a *Step1* taszkot, ami után a *Signal1* kiadja a jelet, amit mindhárom *Signal2* példány elkap és a feladat kosárban 3 darab *Step3* taszk fog várakozni.

## Message (üzenet) esemény

A 6.7. ábra egy olyan process-t mutat, amit a message event használatának bemutatására készítettünk. A *TestProc1* pool *Signal1* eseménye indítja el a *TestProc2* folyamatot, amely művelet elvégzésére a *Signal2* várakozik. Mindkét process rendelkezik a következő 2 változóval: *name* (Text) és *street* (Text). Innen fogjuk venni a message adatait.




6.7. ábra. Message használata folyamatok között

A példában mindegyik step *General→Portal* fülén beállítottuk a *Display name* jellemzőt, hasonlóan úgy, ahogy itt látjuk:

```
"Step1 ("+name+ "/" +processInstanceId + ")";
```

Ez a script a taszkoknak egy olyan nevet ad, ami utal a taszk tervezéskori nevére, de zárójelbe beteszi a *name* belső változó értékét, valamint azt, hogy a taszk példány melyik konkrét process példányban jött létre, azaz láthatjuk a folyamat egyedi azonosítóját is. Mindez egyszerűbbé fogja tenni a példa konkrét lefutásának a követését. A 2 folyamat közötti üzenet csere a *Message1* küldő

és *Message2* fogadó message event között zajlik le, de előbb nézzük át általánosságban is röviden az üzenet típusú eseményeket!

### Áttekintés

A *Message event* feladata az, hogy adatokat vigyen át egyik folyamatból a másikba. Mindez azért szükséges, mert az átmeneteket (transitions) reprezentáló nyilak nem mehetnek át egyik medencéből a másikba, így kell egy eszköz a 2 folyamat közötti kommunikációra és szinkronizációra. Az üzenetek nem küldhetőek és fogadhatóak ugyanazon folyamaton (pool) belül, de nem is arra lettek kitalálva. A message eventnek 5 fajtája van:

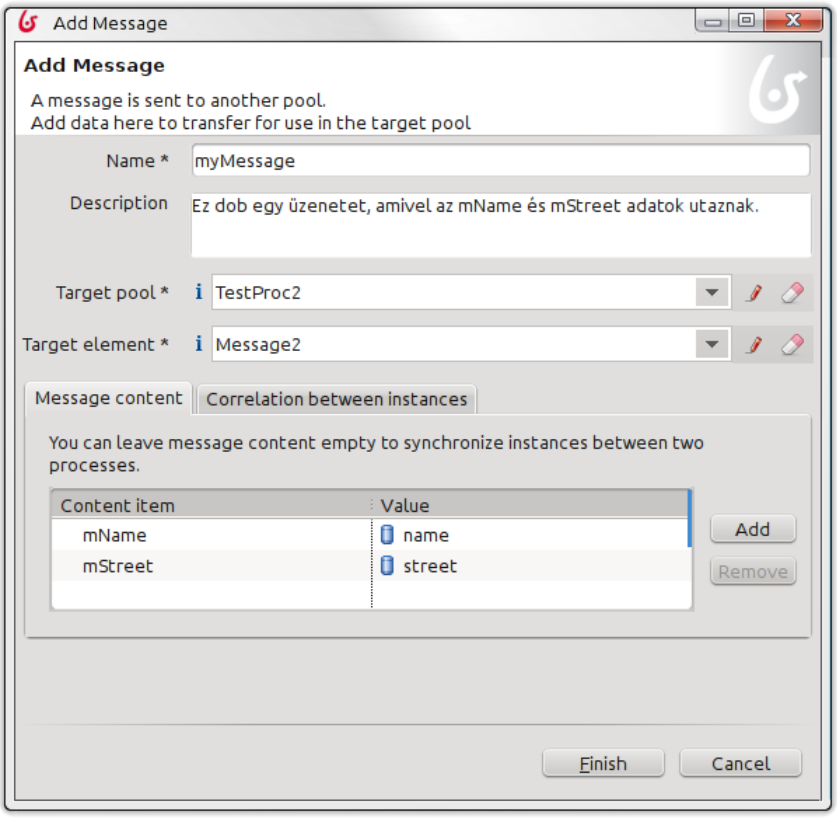
1. *Intermediate Throw Message Event*, köztes esemény és dob egy üzenetet, adatokkal kísérve (amikor a prrocess hozzáérkezik).
2. *Intermediate Catch Message Event*, köztes esemény, ami fogad egy üzenetet, az adataival együtt. Várakozik mindaddig, amíg meg nem érkezik a message.
3. *Start Message Event*: várakozik egy üzenetre, majd a kísérő adatait is felhasználva, elindít egy process-t.
4. *End Message Event*: befejezi azon a ponton a process-t és eldob egy üzenetet.
5. *Boundary Message Event*: Elkapja az üzenetet és abortálja azt a step-et, amire csatlakozik. A vezérlés ilyenkor az exception ágon megy tovább.

### Egy példa a message event használatára

Tekintsük át a 6.7. ábra példájának message event konfigurációit!

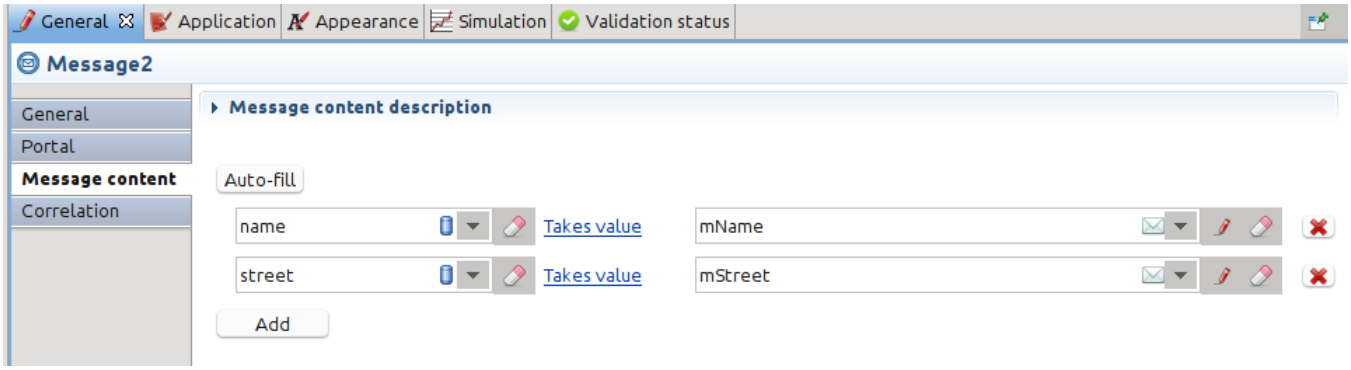
**A throw message konfigurálása** A throw message valamilyen tartalmat (adatokat) tud küldeni egy másik process számára. Emiatt amikor egy ilyet állítunk be, akkor mindig gondoljunk a párjára, azaz egy catch message-et konfiguráljunk be egy másik folyamatban. Fontos megérteni, hogy az üzenet kibocsátásának pillanatában a másik workflow catch event-jének már várakoznia kell, különben az üzenetnek nem lesz célhelye, azaz elveszik. A 6.8. ábra *Message1* throw event *General→Messages* fülről elérhető beállító ablakát mutatja. A *Name* mezőben egy nevet adunk annak az üzenetnek, amit ez az event visz magával (esetünkben: *myMessage*). A *Target pool* és *Target element* mezők teszik lehetővé, hogy megadjuk a célfolyamatot (most ez a *TestProc2* lesz) és annak valamelyik üzenetet fogadni képes elemét. Ez lehet egy catch event (esetünkben ez lesz, azaz a *Message2*) vagy egy receiver típusú taszk. A *Message content* fülön fel lehet sorolni azokat az üzenet mezőket, amiket átviszünk az üzenettel. A *Content item* oszlop az üzenetben lévő mező neve, míg a *Value* oszlop a szokásos kifejezés szerkesztő által megadható valamilyen érték (akár Groovy scripttel előállítva) lehet. Esetünkben 2 tagja van az üzenetnek és ezek értéke közvetlenül a workflow már említett 2 belső változójához lett mappelve. A *Correlation between instances* füllel most ne foglalkozzunk még. A 6.7. ábra *myMessage* nyila automatikusan berajzolódik majd, amikor a catch message-et is hozzákonfiguráltuk ehhez a throw üzenethez, azonban már itt

szeretnénk kiemelni, hogy ez a nyíl üres háromszögű fejjel rendelkezik és szaggatott a szára, ami mutatja, hogy itt esemény közlekedik és nem transition nyílról van szó.



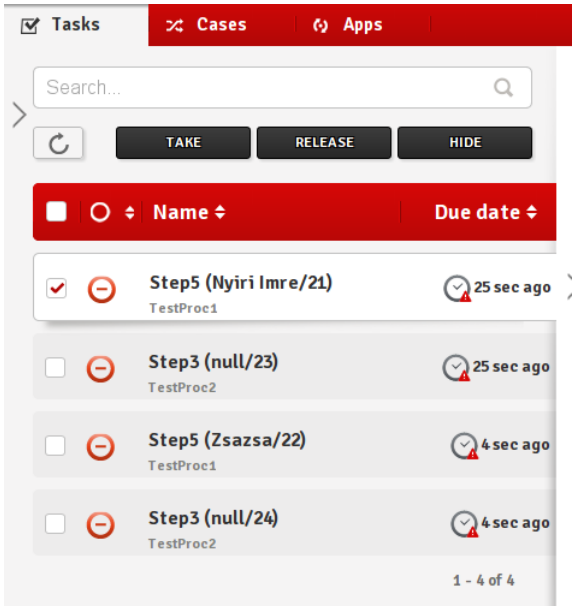

6.8. ábra. A throw message konfigurálása

**A catch message konfigurálása** A 6.9. ábra a *TestProc2* process *Message2* karikájának konfigurálását mutatja. Először a *General→General* fül *Catch message* mezőjénél megadtuk, hogy most a *myMessage* üzenetet akarjuk elkapni.

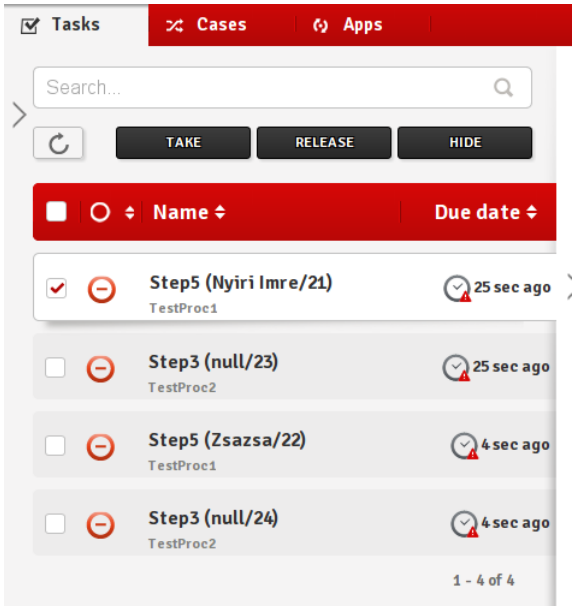



6.9. ábra. A catch message konfigurálása

Az ábrán a *Message content* fül formját láthatjuk éppen, ahogy azt adjuk meg egyenként, hogy a kapott *myMessage* üzenet egyes mezőit a *TestProc2* process mely változói veszik át a kommunikáció megvalósulása során. A *Correlation* fület most is hagyjuk későbbre, a példánál most még nem mutatjuk be a szerepét.




6.10. ábra. Message event tesztelés - 1

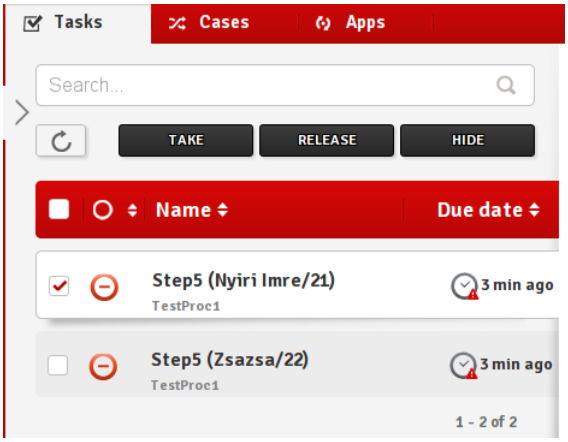



6.11. ábra. Message event tesztelés - 2

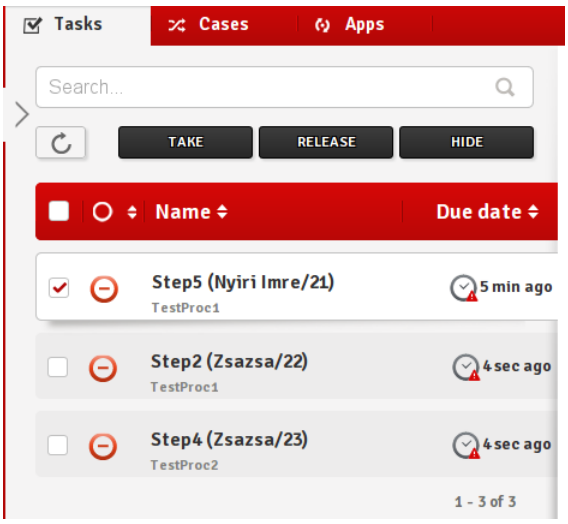
**Tesztelés** A throw és catch message-ek konfigurálása után itt az idő, hogy kipróbáljuk a példát. A *TestProc1* folyamatot indítsuk el 2 példányban, majd mindegyik instance *Step1* lépését hajtsuk is végre. Ekkor a *Signal1* kiadja mindkét példány esetén a jelet és ezzel a *TestProc2* folyamatból is elindul 2 példány. Ezzel 4 darab taszk várakozik végrehajtásra, ahogy a 6.10. ábra is mutatja.

A következő főbb mérföldkövet már a 6.11. ábra tartalmazza. A *TestProc2* mindkét példányára megcsináltuk a *Step3* lépést, így ezzel mindkét catch event várakozásba lépett, hogy fogadjon egy-egy *myMessage* üzenetet. Enélkül a *Message1* üzeneteit nem fogadná senki, ezért végeztük el előtte a *Step3*-mat.

A következő stáció (6.12. ábra) azt mutatja, hogy a *TestProc1* folyamat *Step5* lépéseit végezzük el, így mindkét process példány (látható, hogy ezek a 21 és 22 példány azonosítóval rendelkeznek) eljut a *Message1* ponthoz, ahol kiadódik a *myMessage* üzenet.



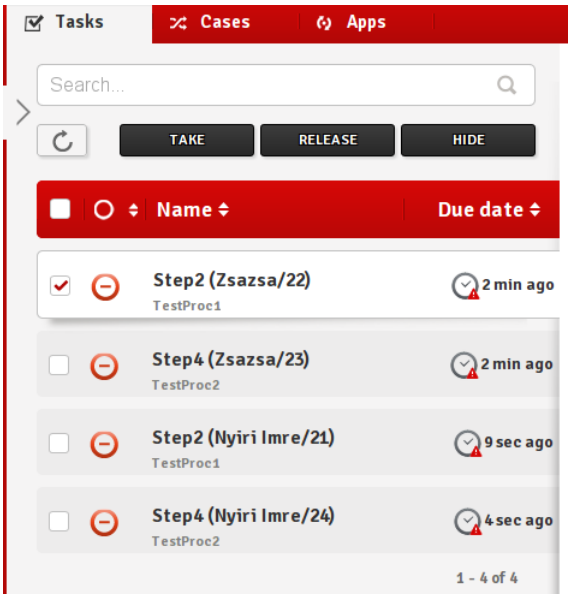

6.12. ábra. Message event tesztelés - 3




6.13. ábra. Message event tesztelés - 4

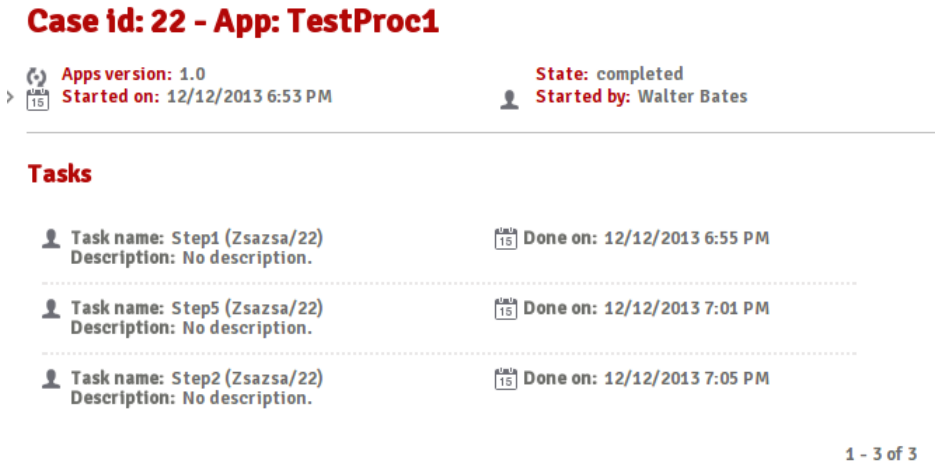
A 6.13. ábra a 22. folyamat *Step5* taszkjának elvégzése utáni állapotot rögzíti, azaz a 23. folyamat azonosítójú *TestProc2* folyamat *Step4* lépéséhez kerültünk.

A 6.14. ábra a másik *Step5* (ami a 21. számú folyamaté) taszk elvégzése utáni állapotot mutatja. Vegyük észre a taszk neveiből, hogy a kommunikációk sikeresek voltak, ami abból is látszik, hogy a *TestProc2* folyamat taszkjainak nevein látszódik az üzenet helyes átvétele, hiszen ott a *name* változó értékét küldtük el a *TestProc1* folyamat példányaiból, ami része a taszk címkéjének is.




6.14. ábra. Message event tesztelés - 5

A 6.15. ábrát csak annak a demonstrálására tettük be, hogy lássuk a taszk neveinek helyes megjelenését is a history-van is, azaz ott, ahol a lezárult case-ek (folyamatok) tekinthetőek át.

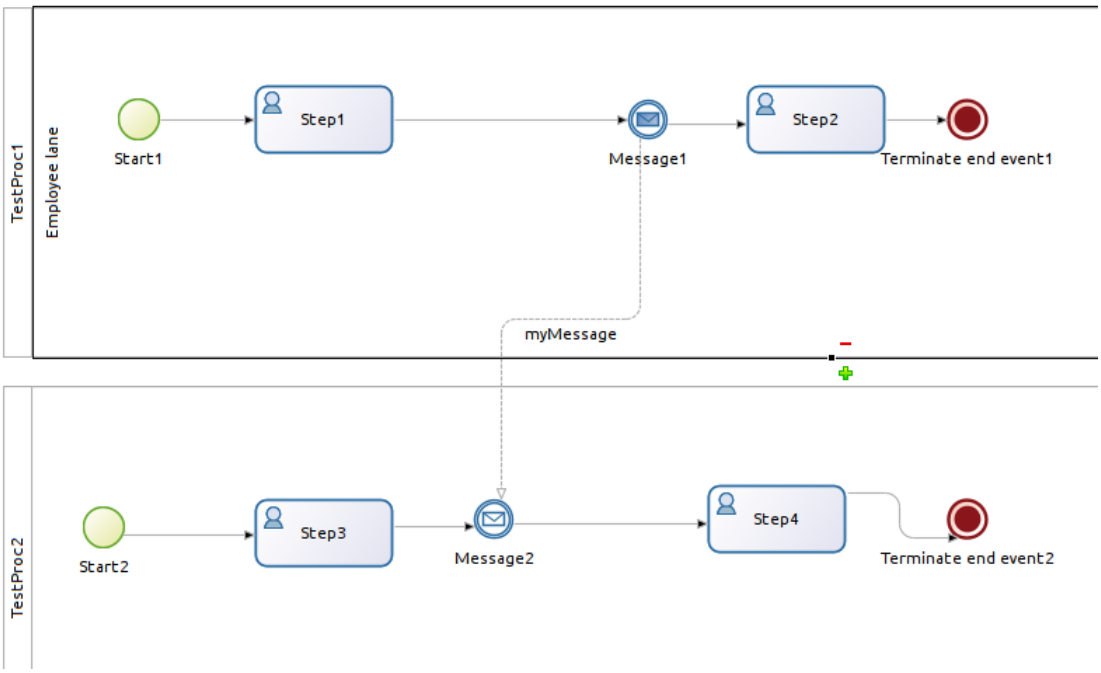



6.15. ábra. A 22. folyamat history-jának egy részlete

**A példa összefoglalása** A tesztelés után 2 lényeges dolgon bizonyára mindenki elgondolkodott. Az egyik az, hogy a példa lefutásában több catch event várakozott az elküldött *myMessage* üzenetre, vajon mi a szabály arra nézve, hogy melyik process példánya lesz kiválasztva? A másik kérdés az, hogy milyen módon tudunk egy adott catch event-et direktben megcímezni, hiszen az esetek legnagyobb részénél ez a feladat, azaz az azonos üzleti azonosítókkal rendelkező folyamatok szeretnének egymással kommunikálni. Ez utóbbira a következő pontban adunk választ, ahol áttekintjük a correlation (azonosítás, kölcsönösség) használatát. Az első kérdésre pedig az a válasz, hogy correlation hiányában a catch eventekhez való érkezés sorrendje adja meg azt a sorrendet, ahogy a throw eventek, azaz a a mi *myMessage*-ünk fogadásra kerülnek. Sokszor ez a működés is értelmes, amennyiben egy kiszolgálási sorrendet szeretnénk tartani és nem fontos az, hogy kinek küldjük az üzenetet.

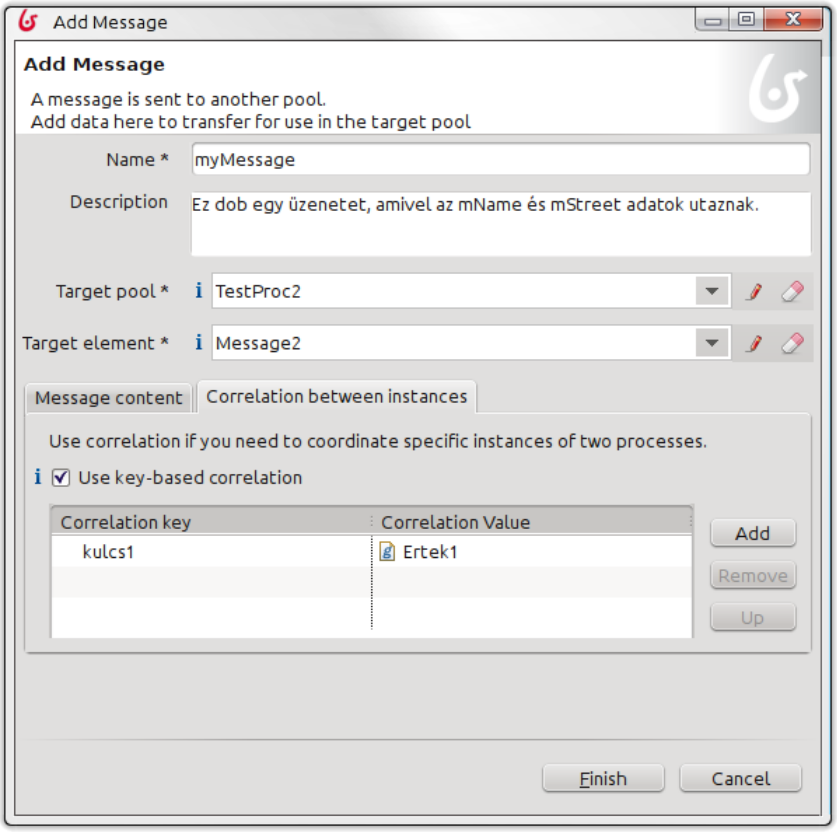
## Message event - A *Correlation* használata

Alapvető igény, hogy 2 process példány úgy tudjon egymással üzenetet váltani, hogy egy kulccsal az üzenetet küldő process kiválaszthassa azt a cél folyamat példányt, ahova az üzenetet szánja. Például lehetséges, hogy egy ügyben 2 különböző folyamat párhuzamosan is elindul és mindkettőben a dolgozó törzsszáma az üzleti azonosító. Az 1. folyamat egy ponton üzenetet tud küldeni, míg a 2. folyamat vár egy üzenetet. Ekkor a törzsszám használatával az 1. folyamat kiválaszthatja a konkrét 2. folyamatot. Ezt a célzott üzenet küldési/fogadási módszert nevezzük *correlation* mechanizmusnak. A 6.16. ábra az előzőleg látott példa átalakításával keletkezett, hogy mindezt a gyakorlatban is bemutathassuk.




6.16. ábra. Message event és a Correlation - 1

A *name* (Text) és *street* (Text) workflow változók is maradtak mindkét processben, továbbra is az ismertetett *myMessage* üzenetet küldi és fogadja a példa, ami egy kicsit egyszerűbb lett, mert a *TestProc2* példányt már kézzel, azaz egy indító form kitöltésével hozzuk létre.



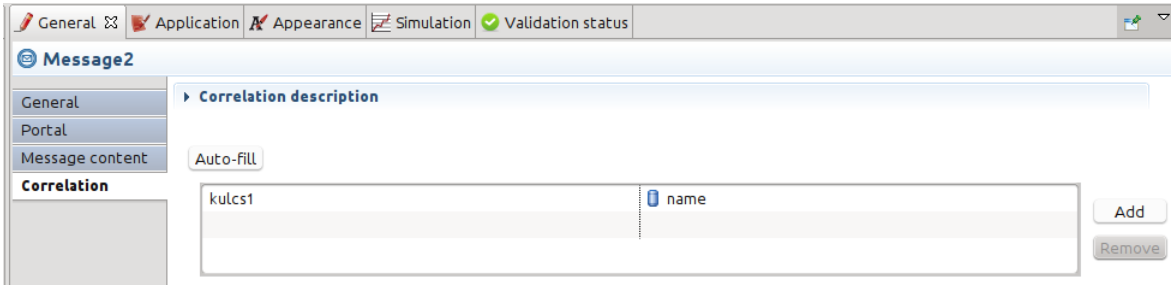

6.17. ábra. Message event és a Correlation - 2

Menjünk először a *TestProc1* diagram *Message1* üzenetküldő elemére és az eddigieken kívül konfiguráljuk be a *Correlation between instances* fülön lévő paramétereket is (6.17. ábra). Itt maximum 5 darab, a megadás sorrendjében figyelembevett kulcsot adhatunk meg (*Correlation key*), amelyek konkrét értéke az, amit a workflow motor felhasznál majd a célfolyamat példány kiválasztásához. Esetünkben csak 1 darab kulcsot használtunk, aminek *kulcs1* nevet adtunk és az üzenet küldésekor kialakult értékét ezzel a Groovy scripttel számoltuk ki:

```
name; // java.lang.String érték, amit visszaad
```

Ez azt jelenti, hogy a *TestProc1* példány éppen aktuális *name* mezőjének az értéke lesz a kulcs, azaz olyan folyamathoz fog menni az üzenet, ami egy *myMessage-et* vár éppen és ezzel a kulccsal várja azt. A 6.18. ábra már azt tartalmazza, hogy a *TestProc2* diagram *Message2* fogadó elemét minderre milyen módon lehet bekonfigurálni. A catch message event esetén megjelenik a *Correlation* fül, amit az ábra éppen mutat. Itt már ki is választhatjuk a *kulcs1* szempontot, a stúdió már ismeri, hiszen előbb már ezt a kulcsot megadtuk. Amennyiben több kulcs lenne, úgy célszerű az *Auto-fill* gombot megnyomni, mert akkor az kulcsok oszlopa automatikusan kitöltődik

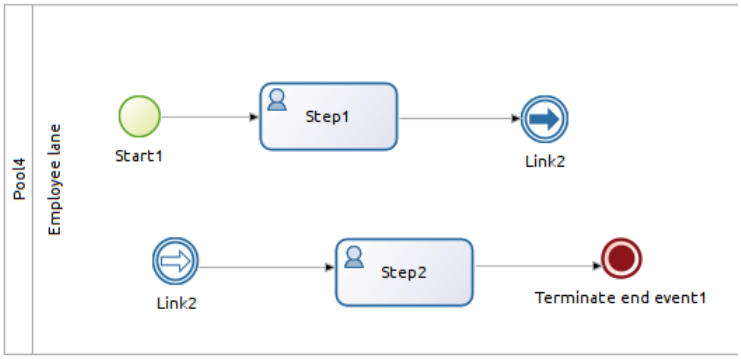
és csak az értékeket kell a 2. oszlopban megadnunk. Esetünkben az egyetlen *kulcs1* szemponthoz a *TestProc2* diagram *name* mezőjét adtuk meg, azaz azt mondjuk ezzel, hogy egy olyan üzenetre várunk, ami az itt megadott kulccsal jön. Mindez a workflow motornak fontos, mert ezzel tudja kiválasztani az üzenet címzettjét. Az engine egyébként a workflow példányok egyéb adataival ezeket a kulcsokat is tárolja a perzisztens rétegben (adatbázisban), ezeket automatikusan indexeli is, hogy hatékony legyen a működés.




6.18. ábra. Message event és a Correlation - 3

Röviden tekintsük is át a fenti 2 process egy lehetséges használati esetét. Indítsuk el mondjuk a *TestProc1* példányt és maradjunk a *Step1* lépésénél. Induláskor a *name=Nyiri Zsazsa* értéket adtuk meg. Ezután a *TestProc2* példányból indítsunk 3 darabot és a 2. példány ugyancsak *name* mezője is legyen *Nyiri Zsazsa* értékű. Itt mindhárom példány *Step3* lépését is végezzük el, így azok mindegyike a saját *Message2* pontnál fog várakozni, hogy jöjjön egy *myMessage* üzenet, aminek *Nyiri Zsazsa* a kulcsa. Ezután az 1. process *Step1* lépését is csináljuk meg, amire a *myMessage* eldobódik *Nyiri Zsazsa* correlation key értékkel. A futás során a 2. process példányhoz kerül az üzenet, tehát a kívánt működés az, amire terveztük.

## Link esemény



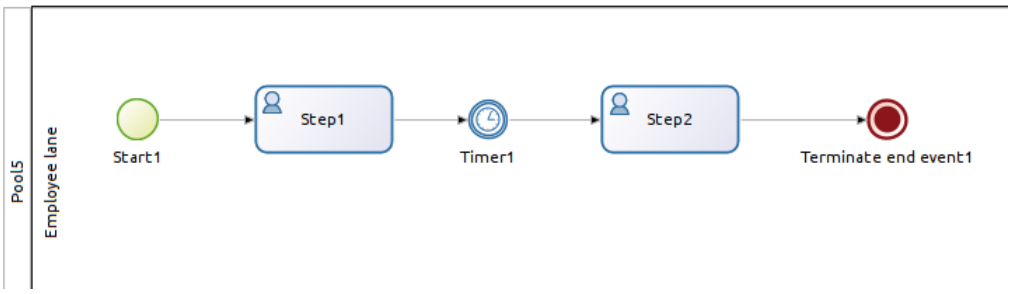

6.19. ábra. Link event

A 6.19. ábra a link esemény használatát tanítja meg. Ez az event egy diagram készítést egyszerűsítő technika, ebben az értelemben nem hasonlít az eddig bemutatott eseményekhez. Egyszerűen csak

arról van szó, hogy egy BPMN ábra lehet nagy is, így azt el kell tudnunk törni kisebb darabokra. A link event ezeknek a töréspontoknak a 2 vége. Az ábra logikailag tehát azt jelenti, hogy a *Step1* és *Step2* egymás után fog végrehajtódni, mintha nem is lenne köztük semmilyen egyéb jel csak egy transition nyíl. A *Link2* throw event esetén a *General→General* fül *Go to* mezőjénél kiválaszthatunk egy catch link eventet. A *Link2* catch event *From links* mezőjébe több throw link eventet is betehetünk, hiszen több helyről is folytatódhatna innen a munkafolyamat. Most természetesen csak a *Link2* érték lesz itt.

## Közbülső timer event

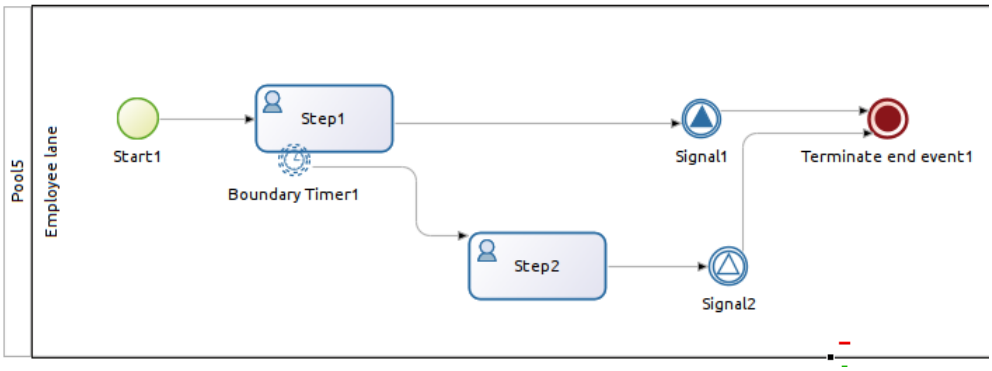
A 6.20. ábra egyszerű példáját csak annak fontossága miatt szeretnénk megmutatni. A *Step1* lépés után a már ismert timer felprogramozás idejére várakozik a munkafolyamat, majd a *Step2* következik. A használat egyszerű, de sok esetben szükségesek ezek a várakozások a munkafolyamatainkban.




6.20. ábra. Intermediate Timer event használata

## Nem megszakítható boundary timer event

Már láttuk a boundary timer eventet, de annak van egy fontos testvére is, azaz egy olyan változat, ami a timer trigger hatására szintén elcsattan, de nem szakítja meg azt a taszkot, amihez csatlakozik. Ilyet mutat a 6.21. ábra.

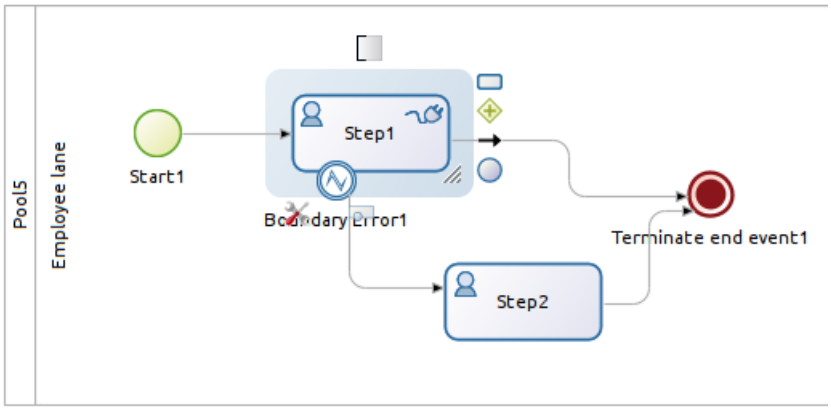



6.21. ábra. Nem megszakítható boundary timer event

Amennyiben a *Step1* határideje lejár, úgy a szaggatott karikával jelzett timer nem szakítja azt meg, de a *Step2* feladat is megkeletkezik. A *Signal2* feladata a példában csak annyi, hogy a *Step2* elvégzése után várakoztatja a folyamat, hogy az ne fejeződjön be. A *Step1* elvégzése után kiadódik a *Signal1* és a munkafolyamat is véget ér.

## Error event

A workflow futása során számos ponton keletkezhet hiba, ezért sok helyen van arra lehetőség, hogy megadjuk azt, hogy ilyenkor egy névvel ellátva az eldobódjon. Itt nincs közbülső throw error, hiszen hiba bárhol keletkezhet. Elkapni egy start vagy boundary eventből tudjuk ezeket a kiváltódott és névvel megadott hibákat. A 6.22. ábra éppen egy határeseményként kapja el azokat a hibákat, amik a *Step1* lépésben keletkezhetnek és megkértük arra, hogy azt kapja el. A példában látható, hogy van egy konnektor (egy másik fejezetben részletesen is bemutatjuk), ami hibát is tud dobni. Ezt a hibát a példában *ALMA* névre kereszteltük a konnektor példány konfigurálása során. Megadtuk az *If connector fails...* mezőben, hogy ekkor dobjon egy hibát (*throw error event*) ilyen névvel. A *Boundary Error 1* catch event *Associated error* mezőjénél pedig kiválasztottuk a korábban definiált *ALMA* error eseményt. Amennyiben a hiba fellép, úgy a *Step1* megszakítódik és a *Step2* keletkezik meg.




6.22. ábra. Error event kezelése

## A Message eventek és a Taszkok

A fenti példák jól összefoglalják és be is mutatják azokat a tipikus helyzeteket, amikor eseményeket érdemes használni. Befejezésként szeretnénk kiemelni, hogy a Message eventek esetén sok esetben használhatjuk a hasonló működésű és konfigurálású taszkokat:

- Send taszk: konfigurálása a throw message event alapján történik.
- Receive taszk: konfigurálása a catch message event alapján történik.

# 7. A Bonita 6. API használata

Bármely szerver szoftvert akkor tudunk igazán megismerni, amikor elkezdjük használni az API felületét és programokat készítünk rá. Ez volt a szerző hitvallása mindaddig, amíg nem kezdett el mélyebben foglalkozni az Open Source világ barátságos, másképpen gazdaságos és tudományos szellemiségével. Ma már megadatik, hogy sok nagyszerű programot ne csak fekete dobozként próbálgassunk az API-ján keresztül, hanem fehér dobozként bele is nézhessünk. Ezen 2 tanulási módszer remekül kiegészíti egymást, tekintsünk hát bele a Bonita Engine API-ba!

A könyv ezen részében a Bonita API programozását mutatjuk be, aminek 2 oka is van. Megismerni a workflow motor belső részleteit, ezáltal jobban megérteni a működési módját. Másfelől olyan tudás birtokába kerülni, amellyel saját programrészekkel is elérhetjük a szervert. Ez tipikusan 3 használati módot tesz majd lehetővé:

1. A Bonita Stúdió Form Builder-ből az API használata.
2. Egy TASK formja lehet egy teljesen testre szabott külső URL mögötti weblapon (ekkor a stúdióban a redirect URL opciót kell választani a form kialakításánál), ami akármilyen webes technológiával elkészíthető (JavaServer Faces, JavaServer Pages, ASPX, PHP).
3. Egy teljesen egyedi külső alkalmazást készítünk, ami csak a workflow engine-t használja.

A következő példákban a 3. és egyben legáltalánosabb módon használva mutatjuk be az API-t, azaz *Eclipse* környezetben parancssoros Java programokat készítünk és külső alkalmazásként futtatjuk őket.

## A Bonita Home fogalma

A Bonita Home egy olyan mappa a fájlrendszerben, ami egy konfigurációs környezetet biztosít a Bonita motort elérni akaró kliens programok számára, így ezt kell elsőnek áttekintenünk.

### A Bonita Home létrehozása és használata

Amikor egy külső kliens programot fejlesztünk a Bonita engine eléréséhez, akkor az első lépés *bonita.home* létrehozása, ami a külső property értékek beállítását jelenti. A *bonita.home* Java system property azt a könyvtárat mutatja, ahonnan a kliens alkalmazás a működési környezetét inicializálja. Ennek a könyvtárnak speciális szerkezete van, mert ezeket az alkönyvtárakat kötelezően tartalmazza:

- *${bonita.home}/client*
- *${bonita.home}/client/conf*

A *conf* alkönyvtár tartalmaz egy *bonita-client.properties* fájlt, ami definiálja azt a módszert, ahogy a Bonita kliens program kapcsolódik (connect) a Bonita engine-hez. Az 7-1. Programlista egy példát mutat a tartalmára.

7-1. Programlista: bonita-client.properties példa

```
# bonita-client.properties példa
application.name=myClientExample
org.bonitasoft.engine.api-type=HTTP
server.url=http://localhost:8080
org.bonitasoft.engine.api-type.parameters=server.url,application.name
```

A példában a szerver elérés a *HTTP* protokollt használja. Az egyik legismertebb Bonita kliens maga a Bonita portál, így érdemes belenézni az ő *bonita-client.properties* fájl tartalmába (7-2. Programlista).

7-2. Programlista: A Bonita portál bonita-client.properties tartalma

```
###template file
# LOCAL
org.bonitasoft.engine.api-type = LOCAL

# HTTP
#org.bonitasoft.engine.api-type = HTTP
#server.url = http://localhost:8080
#application.name = bonita

# Remote: EJB3 / JBoss 5
#org.bonitasoft.engine.api-type = EJB3
#java.naming.factory.initial = org.jnp.interfaces.NamingContextFactory
#java.naming.provider.url = jnp://localhost:1099
```

A fájl mutatja, hogy 3 módon is elérhetjük az engine-t, de egyszerre csak egy lehet aktív a kliens számára:

- Local módon → Az API elérése csak akkor lehet LOCAL módban, ha a kliens ugyanazon a gépen fut.
- HTTP protokoll használatával → Egy távoli gépről a HTTP protokoll feletti API használatát biztosítja. Ez egy remek módszer, ha a kliens és a szerver kommunikációját az internetes HTTP rétegre szeretnénk bízni. Ilyenkor a HTTP szerver elérhetőségét (host:port) és az azon futó webszolgáltatás alkalmazás (alkalmazás context név) nevét kell megadnunk.
- EJB3 kliensként → Az EJB[5] feletti API elérés. A példa JBOSS esetére mutatja be az EJB kliens létrehozásához szükséges paramétereket: A JNDI[6] provider szerver URL-je, illetve az a factory, amivel le tudjuk gyártani az EJB kliens proxy objektumot.

A gyakorlatban egy futó Bonita motor már tartalmazza a *bonita.home* könyvtárat és azt is használhatjuk, aminek az a feltétele, hogy a kliens ugyanazon a gépen fusson, mint a szerver. A Bonita Stúdióban lévő *bonita.home* helye, amennyiben *$BONITA* helyre telepítettük a Bonita szerver csomagot: *$BONITA/workspace/bonita*. A most bemutatott 3 API elérési lehetőség mindegyike natív módon, az API osztályok közvetlen használatával biztosítja a felületet a programozók részére, mindössze a kliens/szerver kommunikációs réteg más, de ez rejtve marad a háttérben. Szeretnénk kiemelni, hogy a *restful* típusú webservice réteget is biztosítja a Bonita engine, ezzel megteremtve az API platformfüggetlen elérését is.

[5] Java Enterprise JavaBean
[6] Java Naming and Directory Interface

### Egy saját Bonita Home kialakítása

Amikor másik home-ot készítünk egy kliensnek (akár egy távoli, akár a lokális gépen), akkor az alábbi 4 lépést kell végrehajtani:

1. Létrehozni egy könyvtárat, ami a *bonita.home* lesz.
2. Egy már kialakított .../*bonita/client* tartalmát átmásolni ebbe a könyvtárba.
3. A deploy csomagban szintén meglévő .../*bonita/server* tartalmát ugyancsak ide kell másolni.
4. A *client* könyvtárban módosítani a *bonita-client.properties* fájlt HTTP vagy EJB3 elérésre (esetünkben az 7-3. Programlista szerint alakítottuk ki). Természetesen itt át kell írni a konkrét elérési paramétereket arra az értékre, ami a távoli Bonita engine eléréséhez szükséges.

7-3. Programlista: bonita-client.properties a saját Bonita Home-ban

```
1 application.name=bonita
2 org.bonitasoft.engine.api-type=HTTP
3 server.url=http\://localhost|:8082
4 org.bonitasoft.engine.api-type.parameters=server.url,application.name
```

Amennyiben például a *bonita.home* helye a */home/tanulas/bonita/bonita-home* könyvtár, úgy a programokban érdemes lesz ezt a konstanst felvenni:

```
public static final String BONITA_HOME = "/home/tanulas/bonita/bonita-home";
```

Ezután a következő beállítással a kliens programban már az új home fog működni a Bonita engine eléréséhez:

```
System.setProperty("bonita.home", BONITA_HOME);
```

## A rendszerben regisztrált felhasználók listája

Itt az idő, hogy elkészítsük az első parancssoros Java programunkat, aminek a feladata az, hogy kapcsolódjon rá a futó bonita motorra és kérje le a user-eket, de maximum csak 20 főt. A példa megoldása (7-4. Programlista) során a Stúdióba integrált Tomcat-et és az ott lévő *bonita.home*-ot használtuk (17. sor). A *main()* metódus a 25. sorban beállítja a *bonita.home* értékét, használva a 41-44 sorok között létrehozott *initHome()* metódust. A 27. sor lekér egy *LoginAPI* objektumot, amivel a 29. sorban be is jelentkezünk (user/password=install/install), megkapva a *session* objektumot, ami reprezentálja a kapcsolatot. A 30. sorban ezek után már meg tudunk szerezni egy *IdentityAPI* objektumot, ami a felhasználókkal kapcsolatos teendőket képes összefogni. A 31. sor 2 dolgot is elvégez. Összeállít egy kereső feltételt, majd ezzel lekér egy eredményt az API-tól. A *SearchOptionsBuilder* class építi össze azt a feltételt (filter condition), ami alapján a keresés majd történik. Érdekességként vegyük észre, hogy az API itt a builder design pattern-t használja, azaz a *done()* a keresési feltételt adja vissza, ami most nagyon egyszerű: minimum 0, maximum 20 elem lehet az eredménylistában. Az *identityAPI.searchUsers()* hívás egy *SearchResult<User>* objektumot szolgáltat, azaz a *User* példányok listáját. A *User* class természetesen a felhasználót

reprezentáló osztály. A 33-36 sorok ciklusban kiírják a felhasználó login nevét és belső egyedi azonosítóját. A 37. sor megszünteti a szerverrel a kapcsolatot.

7-4. Programlista: GetUserTest.java - Az első 20 user

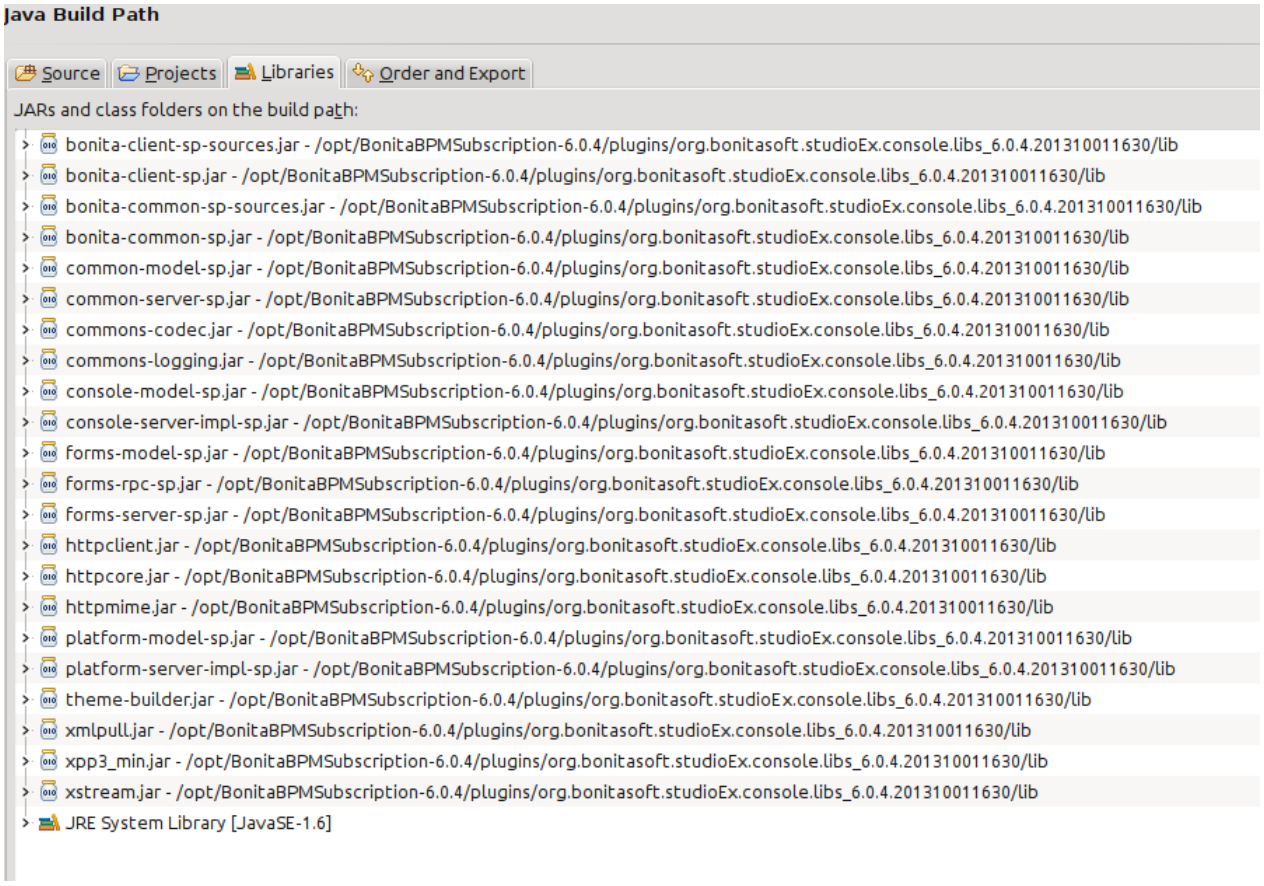
```
1  package org.cs.bonita.apitest;
2
3  import java.io.File;
4  import java.io.FileWriter;
5  import java.io.IOException;
6  import java.util.Properties;
7  import org.bonitasoft.engine.api.IdentityAPI;
8  import org.bonitasoft.engine.api.LoginAPI;
9  import org.bonitasoft.engine.api.TenantAPIAccessor;
10 import org.bonitasoft.engine.identity.User;
11 import org.bonitasoft.engine.search.SearchOptionsBuilder;
12 import org.bonitasoft.engine.search.SearchResult;
13 import org.bonitasoft.engine.session.APISession;
14
15 public class GetUserTest
16 {
17    public static final String BONITA_HOME = "/opt/BonitaBPMSubscription-6.0.4/workspace/bonita";
18
19    /**
20     * @param args
21     * @throws Exception
22     */
23    public static void main(String[] args) throws Exception
24    {
25       initHome(BONITA_HOME);
26
27       final LoginAPI loginAPI = TenantAPIAccessor.getLoginAPI();
28       System.out.println("login with install//install");
29       final APISession session = loginAPI.login("install", "install");
30       final IdentityAPI identityAPI = TenantAPIAccessor.getIdentityAPI(session);
31       final SearchResult<User> searchUsers = identityAPI.searchUsers(new SearchOptionsBuilder(0,➥
             20).done());
32       System.out.println("20 first users:");
33       for (final User user : searchUsers.getResult())
34       {
35          System.out.println(" * " + user.getUserName() + " -- " + user.getId());
36       }
37       loginAPI.logout(session);
38       System.out.println("logged out");
39    }
40
41    private static void initHome(String homeFolderName) throws IOException
42    {
43       System.setProperty("bonita.home", BONITA_HOME);
44    }
45 }
```

A következő lépés a program futtatása. A fejlesztés során az Eclipse környezetben beállított jar-okat a 7.1. ábra mutatja. Ezek már a fejlesztéshez is kellenek természetesen, így innen könnyen tudjuk futtatni a programot is.

Amikor Eclipse környezeten kívül is szeretnénk használni a *GetUserTest* osztályt, akkor az 7-5. Programlista-hoz hasonló script-et érdemes rá készíteni. A *for* ciklus beállítja a CLASSPATH változót úgy, hogy a *lib* könyvtárban lévő jar-ok mindegyikét tartalmazza, majd lefuttatja az osztályt.

7-5. Programlista: Az LDAPSynchronizer futtatását végző shell script

```
#!/bin/sh
for jar in $(ls lib/*.jar); do
        CLASSPATH=$CLASSPATH:${jar}
done
java -classpath $CLASSPATH com.bonitasoft.ldapsynchronizer.LDAPSynchronizer $1
```




7.1. ábra. A kliens program által használt Java jar fájlok

Az 7-6. Programlista a futtatás eredményének egy részletét mutatja, kihagyva a közepét.

7-6. Programlista: A futás eredményének egy részlete

```
login with install//install
20 first users:
 * anthony.nichols -- 18
...
 * daniela.angelo -- 17
 * virginie.jomphe -- 7
 * walter.bates -- 4
logged out
```

## A naplózás használata

Egy programnak mindig naplóznia kell a működését, ezért annak beállítását mutatjuk be a következőkben. A fenti példa *System.out.println* hívásait naplózó hívásokra érdemes átalakítani.

### A naplózás használata a saját kliens programjainkban

A Bonita bármelyik ismertebb Java logging megoldással együtt tud működni, most nézzük meg a *log4j*-n keresztül az alaplehetőséget, amit csak a külső programok készítéséhez érdemes így használni. Egy szerver környezetben a *log4j* mindig a helyi server policy szerint használatos. A *log4j* konfigurációs fájlt tegyük például ide: */home/tanulas/bonita/bonita-home/log4j.properties*, aminek a tartalmát a 7-7. Programlista mutatja. Akit a beállítások részletei is érdekelnek, tanulmányozza a LOG4J naplózó rendszer dokumentációját.

7-7. Programlista: log4j.properties

```
1  # Define the root logger with appender file
2  log = /home/tanulas/bonita/bonita-home
3  #log4j.rootLogger = DEBUG, FILE
4  log4j.rootLogger = INFO, FILE
5
6  # Define the file appender
7  log4j.appender.FILE=org.apache.log4j.FileAppender
8  log4j.appender.FILE.File=${log}/log.out
9
10 # Define the layout for file appender
11 log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
12 log4j.appender.FILE.layout.conversionPattern=%m%n
```

A 7-8. Programlista a *GetUserTest* class azon változatát mutatja, ahol már naplózunk is. A 9. sor inicializálja a LOG4J rendszert a külső konfigurációs fájl szerint, majd a 11. sorban lekérünk egy loggert, amivel készíthetjük a naplóbejegyzéseket.

7-8. Programlista: GetUserTest class naplózással

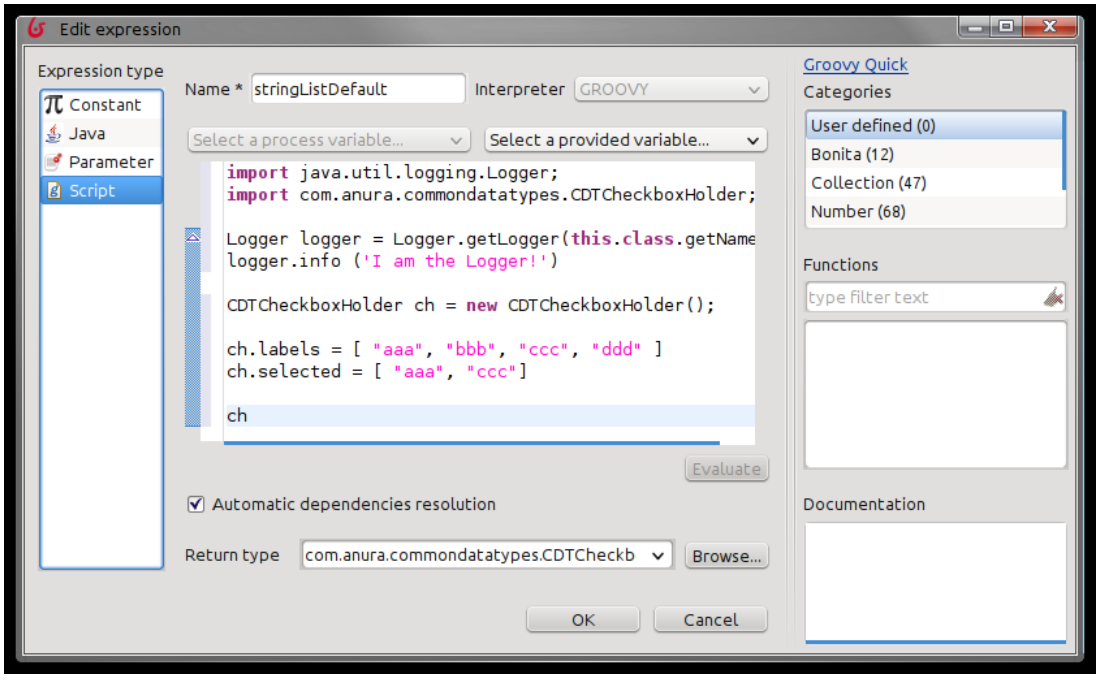
```
1  ...
2  public class GetUserTest
3  {
4     public static final String BONITA_HOME = "/home/tanulas/bonita/bonita-home";
5     public static final String LOG_CONF = "/home/tanulas/bonita/bonita-home/log4j.properties";
6  ...
7  public static void main(String[] args) throws Exception
8  {
9     PropertyConfigurator.configure( LOG_CONF );
10    initHome(BONITA_HOME);
11    Logger log = Logger.getLogger("org.cs.bonita.apitest");
12    final LoginAPI loginAPI = TenantAPIAccessor.getLoginAPI();
13
14    //System.out.println("login with install//install");
15    log.info( "login with install//install" );
16 ...
17    loginAPI.logout(session);
18    System.out.println("logged out");
19 }
```

A program futtatása után a napló egy részletét a 7-9. Programlista mutatja, amiből láthatjuk, hogy a 7-3. Programlistán látható *bonita.home*-ot használtuk. A naplófájlok a konfiguráció *log*

sorában megadott könyvtárba fognak kerülni. A naplózás nagyon részletes, mert éppen engedélyeztük a DEBUG üzemmódot. Ez esetünkben egy 25000 byte-os tartalom lett az eredeti néhány sor helyett. A lentiek tanulsága az is, hogy a Bonita API átveszi azt a naplózó rendszert, amit éppen bekonfiguráltunk.

7-9. Programlista: A napló tartalmának egy részlete

```
1  login with install//install
2  login with install//install
3  Connection request: [route: {}->http://localhost:8082][total kept alive: 0; route allocated: 0 ➥
       of 2; total allocated: 0 of 20]
4  Connection leased: [id: 0][route: {}->http://localhost:8082][total kept alive: 0; route ➥
       allocated: 1 of 2; total allocated: 1 of 20]
5  Connecting to localhost:8082
6  CookieSpec selected: best-match
7  Auth cache not set in the context
8  Target auth state: UNCHALLENGED
9  Proxy auth state: UNCHALLENGED
10 Attempt 1 to execute request
11 Sending request: POST /bonita/serverAPI/org.bonitasoft.engine.api.LoginAPI/login HTTP/1.1
12 >> "POST /bonita/serverAPI/org.bonitasoft.engine.api.LoginAPI/login HTTP/1.1[\r][\n]"
13 >> "Content-Length: 465[\r][\n]"
14 >> "Content-Type: application/x-www-form-urlencoded; charset=UTF-8[\r][\n]"
15 >> "Host: localhost:8082[\r][\n]"
16 >> "Connection: Keep-Alive[\r][\n]"
17 >> "User-Agent: Apache-HttpClient/4.2.5 (java 1.5)[\r][\n]"
18 >> "[\r][\n]"
19 >> POST /bonita/serverAPI/org.bonitasoft.engine.api.LoginAPI/login HTTP/1.1
20 >> Content-Length: 465
21 >> Content-Type: application/x-www-form-urlencoded; charset=UTF-8
22 >> Host: localhost:8082
```




7.2. ábra. Naplózás a Stúdióból

### A naplózás használata a Bonita Stúdióban

A 7.2. ábra mutatja azt a módszert, ahogy egy Logger megszerezhető (3. sor) egy script szerkesztőn belül. Tomcat alatt a *$BONITA/workspace/tomcat/logs* könyvtárban több előre konfigurált napló is van, ez a kettő a legnagyobb hasznú a fejlesztő számára:

- *bonita.2013-10-05.log* (naponta fordul) → a hibák innen nézhetőek meg
- *catalina.2013-10-05.log* (naponta fordul) → a mi napló üzeneteink is itt vannak.

## User létrehozása és adatainak változtatása

A mindennapi munkában gyakran adódik, hogy a felhasználókat egy külső programmal szeretnénk menedzselni. Ilyen program például az LDAP synchronizer is, amit akár magunk is megírhatnánk.

### Új felhasználó létrehozása

A *testCreateUser()* metódus (7-10. Programlista) bemutatja az új user létrehozását. Az egyedüli új ismeret a 14. sorban látható, ahol a *createUser()* hívással létrehozunk egy *inyiri* felhasználót *titok* jelszóval.

7-10. Programlista: testCreateUser() - Egy új user létrehozása

```
1  private static User testCreateUser() throws Exception
2  {
3      PropertyConfigurator.configure(LOG_CONF);
4      initHome(BONITA_HOME);
5      Logger log = Logger.getLogger("org.cs.bonita.apitest");
6
7      // Megszerzi a login-t végrehajtó objektumot
8      final LoginAPI loginAPI = TenantAPIAccessor.getLoginAPI();
9      log.info("login with install//install");
10     final APISession session = loginAPI.login("install", "install");
11
12     final IdentityAPI identityAPI = TenantAPIAccessor.getIdentityAPI(session);
13     // A paraméterek: user és password
14     User user = identityAPI.createUser("inyiri", "titok");
15
16     loginAPI.logout(session);
17     return user;
18 }
```

A metódus lefuttatása után a Bonita Portálon (és ezzel a user adatbázisban) megjelenik az *inyiri* user, akinek *titok* lett a jelszava. Amennyiben jobban megnézzük ezt a user-t, láthatjuk, hogy semmilyen profil adata nincs kitöltve, ezért a következőkben nézzük meg, hogy ezt mi módon tudjuk bepótolni, amit a *testUpdateUser()* metódus mutat meg nekünk.

### Egy felhasználó adatainak módosítása

Az *identityAPI* változó lekérésig ugyanaz a működés, mint eddig, új elemek csak a 14-19 sorok között jelennek meg. A 14. sorban az *inyiri* user *User* objektumot állítja elő a *getUserByUserName()* metódus. A 15. sor egy *UserUpdater* objektumot állít elő, amin keresztül a felhasználó

adatait tudjuk beállítani a következő sorokban, hogy a végén (19. sor) az *updateUser()* hívással azt meg is tegyük. Ennek 2 paramétere van:

- A felhasználó belső azonosítója, amit a User objektumtól tudunk elkérni a *user.getId()* segítségével.
- Az elkészített *UserUpdater* objektum (*userUpdater*), ami tartalmazza a megváltoztatni kívánt user profile adatokat.

7-11. Programlista: testUpdateUser() - Egy felhasználó adatainak módosítása

```
1  private static User testUpdateUser() throws Exception
2  {
3      PropertyConfigurator.configure(LOG_CONF);
4      initHome(BONITA_HOME);
5      Logger log = Logger.getLogger("org.cs.bonita.apitest");
6
7      // Megszerzi a login-t végrehajtó objektumot
8      final LoginAPI loginAPI = TenantAPIAccessor.getLoginAPI();
9      log.info("login with install//install");
10     final APISession session = loginAPI.login("install", "install");
11
12     final IdentityAPI identityAPI = TenantAPIAccessor.getIdentityAPI(session);
13
14     User user = identityAPI.getUserByUserName( "inyiri" );
15     UserUpdater userUpdater = new UserUpdater();
16     userUpdater.setFirstName("Imre");
17     userUpdater.setLastName("Nyiri");
18     userUpdater.setJobTitle("Integration Expert");
19     identityAPI.updateUser(user.getId(), userUpdater);
20
21     loginAPI.logout(session);
22
23     return user;
24 }
```

A Bonita portálon valóban megjelennek ezek az adatok az *inyiri* felhasználó mellett, tehát a művelet sikeres lett. Megjegyezzük, hogy létezik *UserCreator* osztály, amihez a *createUser()* metódusnak van egy olyan változata, ahol ezeket az adatokat már a felhasználó létrehozásakor is megadhattuk volna. A portálon keresztül látjuk az *inyiri* user-t, most próbáljunk vele belépni! Azt fogjuk kapni, hogy bár ilyen user létezik, de nincs profilja még, azaz nem soroltuk be, hogy ez egy user vagy adminisztrátor, így addig a rendszer nem engedi belépni sem. Ezt a következő pontban bemutatott *testSetUserProfile()* metódussal tudjuk majd pótolni.

### A felhasználó profiljának beállítása

Bármilyen új user az API-n való automatikus felvételének szükséges lépése a *testSetUserProfile()* metódus (7-12. Programlista) által bemutatott profile-ba helyezés, különben a felhasználó nem lesz alkalmas a rendszerbe való belépésre. Mindehhez a *ProfileAPI* és a 19-30 sorok között lévő osztályokat kapjuk segítségül. A metódust lefuttatva, majd az eredményt a portálon megtekintve az *inyiri* user bekerült a *User* profilba és be is lehet már vele jelentkezni. A 20. sorban lekérünk egy *ProfileAPI* objektumot, a szokásos módon a *session*-re hivatkozva. A 21-23 sorok egy keresést valósítanak meg, azaz keressük a *user* nevű profilt, aminek objektum reprezentációját a 27.

sorban el is tároljuk a *profile* változóba. A *ProfileMemberCreator* objektum (28. sor) már ennek a profilnak a belső azonosítójával jön létre, így ez a kreátor már magában hordozza a „user”-séget. A *setUserId()* metódussal megadjuk az éppen kezelendő user belső azonosítóját, majd a 30. sorban ténylegesen elvégezzük a profilba helyezés műveletét.

7-12. Programlista: testSetUserProfile() - A felhasználó profiljának beállítása

```
1  private static void testSetUserProfile() throws Exception
2  {
3      PropertyConfigurator.configure(LOG_CONF);
4      initHome(BONITA_HOME);
5      Logger log = Logger.getLogger("org.cs.bonita.apitest");
6
7      // Megszerzi a login-t végrehajtó objektumot
8      final LoginAPI loginAPI = TenantAPIAccessor.getLoginAPI();
9      log.info("login_with_install//install");
10
11     // Bejelentkezés a Bonita motorba
12     final APISession session = loginAPI.login("install", "install");
13
14     final IdentityAPI identityAPI = TenantAPIAccessor.getIdentityAPI(session);
15
16     // Lekérjük az inyiri User objektumot
17     User user = identityAPI.getUserByUserName( "inyiri" );
18
19     // A profil beállítása user-ként (lehetne például Administrator is)
20     ProfileAPI profileAPI = TenantAPIAccessor.getProfileAPI(session);
21     SearchOptionsBuilder searchOptionsBuilder = new SearchOptionsBuilder( 0, 10 );
22     searchOptionsBuilder.filter(ProfileSearchDescriptor.NAME, "user");
23     SearchResult<Profile> searchResultProfile = profileAPI.searchProfiles(searchOptionsBuilder.➥
           done());
24
25     // 1 sor eredmény biztos lesz
26     if (searchResultProfile.getResult().size()!=1) { return; }
27     Profile profile = searchResultProfile.getResult().get(0);
28     ProfileMemberCreator profileMemberCreator = new ProfileMemberCreator( profile.getId() );
29     profileMemberCreator.setUserId( user.getId());
30     profileAPI.createProfileMember( profileMemberCreator );
31
32     loginAPI.logout(session);
33     return;
34 }
```

## Group (csoport) létrehozása

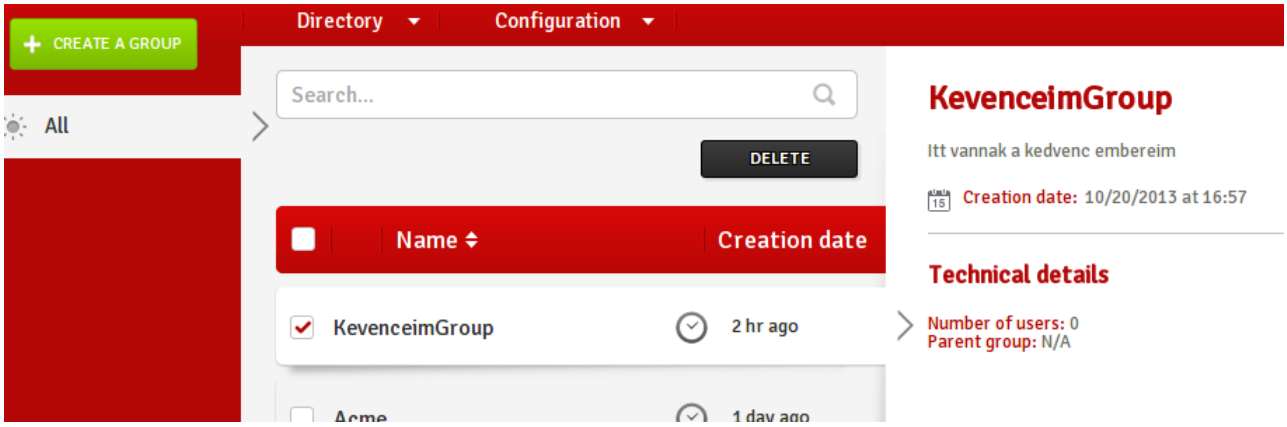
Az *testAddGroup()* metódus létrehoz egy *KevenceimGroup* nevű csoportot.

7-13. Programlista: testAddGroup() - Csoport létrehozása

```
1  private static void testAddGroup() throws Exception
2  {
3      PropertyConfigurator.configure(LOG_CONF);
4      initHome(BONITA_HOME);
5      Logger log = Logger.getLogger("org.cs.bonita.apitest");
6
7      // Megszerzi a login-t végrehajtó objektumot
8      final LoginAPI loginAPI = TenantAPIAccessor.getLoginAPI();
9      log.info("login_with_install//install");
10
```

```
11      // Bejelentkezés a Bonita motorba
12      final APISession session = loginAPI.login("install", "install");
13
14      final IdentityAPI identityAPI = TenantAPIAccessor.getIdentityAPI(session);
15
16      final GroupCreator groupCreator = new GroupCreator("KevenceimGroup").setDescription("Itt ➥
            vannak a kedvenc embereim");
17      final Group group = identityAPI.createGroup( groupCreator );
18
19      loginAPI.logout(session);
20      return;
21  }
```

A kódban semmi nehéz nincs, a *GroupCreator* objektum (16. sor) létrehozza a csoport nevére és leírására való információt, amit a következő sor *identityAPI.createGroup()* hívása használ fel. Az eredményt a portálon tekinthetjük meg (7.3. ábra).




7.3. ábra. A programmal létrehozott GROUP

## Role (szerepkör) létrehozása

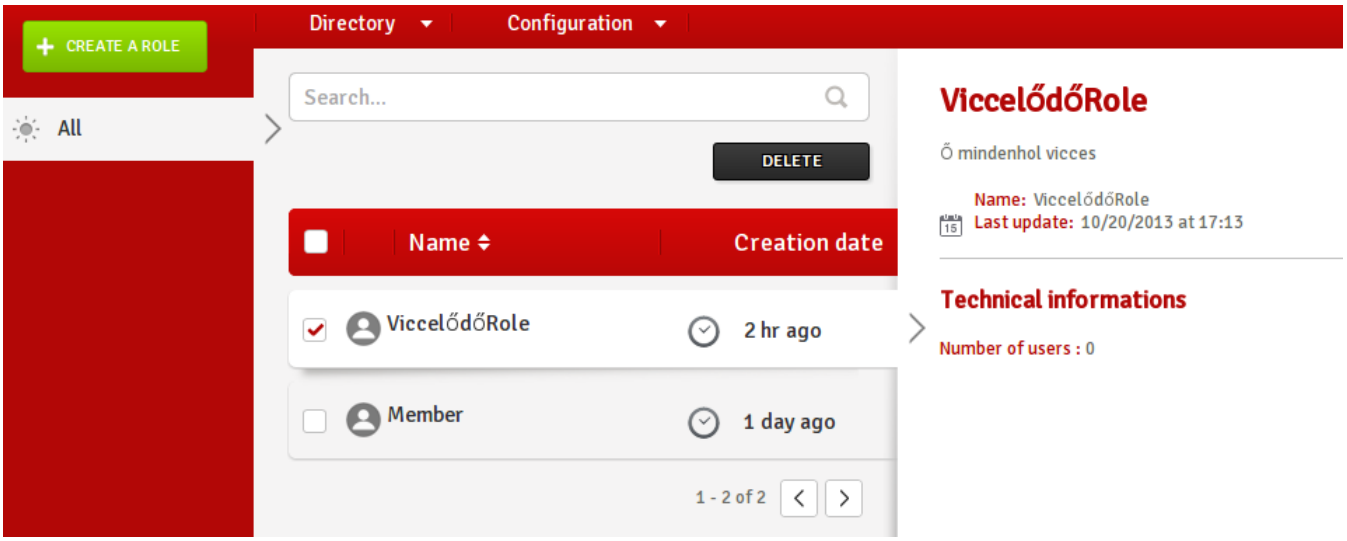
A szerepkör az authorization és Actor mapping másik fontos komponense, ezért a *testAddRole()* metódussal bemutatjuk a létrehozását, ami szinte teljesen analóg az előbbiekben bemutatott group létrehozással (7-14. Programlista). A lényeg a 16-17. sorokban található.

7-14. Programlista: testAddRole() - Szerepkör létrehozása

```
1   private static void testAddRole() throws Exception
2   {
3       PropertyConfigurator.configure(LOG_CONF);
4       initHome(BONITA_HOME);
5       Logger log = Logger.getLogger("org.cs.bonita.apitest");
6
7       // Megszerzi a login-t végrehajtó objektumot
8       final LoginAPI loginAPI = TenantAPIAccessor.getLoginAPI();
9       log.info("login with install//install");
10
11      // Bejelentkezés a Bonita motorba
12      final APISession session = loginAPI.login("install", "install");
13
14      final IdentityAPI identityAPI = TenantAPIAccessor.getIdentityAPI(session);
```

```
15
16      final RoleCreator roleCreator = new RoleCreator("ViccelődőRole").setDescription("Ő mindenhol ➥
            vicces");
17      final Role role = identityAPI.createRole(roleCreator);
18
19      loginAPI.logout(session);
20      return;
21  }
```

A futási eredményt ismét megtekinthetjük a Bonita Portálon (7.4. ábra).




7.4. ábra. A programmal létrehozott ROLE

## Egy user hozzáadása egy csoporthoz

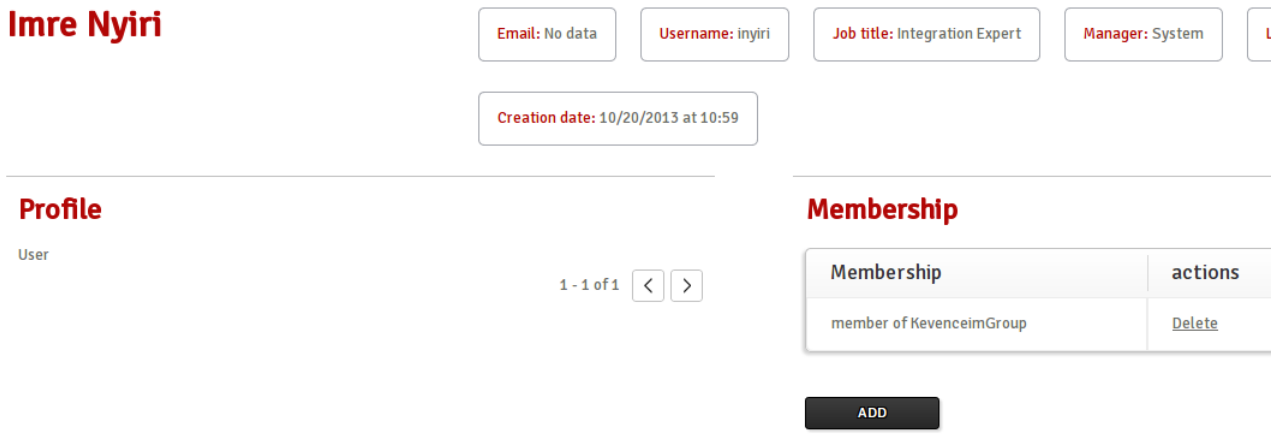
Elöljáróban megjegyezzük, hogy egy user csak úgy adható hozzá egy csoporthoz, ha megadjuk a szerepkört is, amit itt betölt. Ez nem jelent komoly korlátozást, mert a beépített *member* role mindig használható, amennyiben csak azt akarjuk, hogy a user egy group-hoz legyen rendelve. A tagság szerepkör ugyanis mindig fennáll. A példában (7-15. Programlista) az *inyiri* user-t betesszük a *KevenceimGroup* csoportba, aminek az eredményét a portálon (7.5. ábra) láthatjuk.

7-15. Programlista: testAddUserToGroup() - Egy user elhelyezése egy group-ba

```
1   private static void testAddUserToGroup() throws Exception
2   {
3       PropertyConfigurator.configure(LOG_CONF);
4       initHome(BONITA_HOME);
5       Logger log = Logger.getLogger("org.cs.bonita.apitest");
6
7       // Megszerzi a login-t végrehajtó objektumot
8       final LoginAPI loginAPI = TenantAPIAccessor.getLoginAPI();
9       log.info("login with install//install");
10
11      // Bejelentkezés a Bonita motorba
12      final APISession session = loginAPI.login("install", "install");
13      final IdentityAPI identityAPI = TenantAPIAccessor.getIdentityAPI(session);
14
```

```
15      // Lekérjük az inyiri User objektumot
16      User user = identityAPI.getUserByUserName( "inyiri" );
17      Group group = identityAPI.getGroupByPath("KevenceimGroup");
18      Role role = identityAPI.getRoleByName("Member");
19
20      UserMembership membership = identityAPI.addUserMembership(user.getId(), group.getId(), role.➥
            getId());
21
22      loginAPI.logout(session);
23      return;
24  }
```

A 16-18 sorokban a *User*, *Group* és *Role* objektumokat kérjük le a nevük alapján, majd a 20. sorban betesszük a user-t a csoportba.




7.5. ábra. Az *inyiri* bekerült a *KevenceimGroup* csoportba

## Egy group felhasználóinak listája

Gyakran szükség van azt tudni, hogy egy adott csoportnak mely felhasználók a tagjai. A *testGetUsersOfGroup()* metódus (7-16. Programlista) ezt mutatja be. A futási eredmény a csoport egyetlen felhasználójának a vezetékneve lesz (azaz Nyiri).

7-16. Programlista: testGetUsersOfGroup() - A csoport felhasználónak listája

```
1   private static void testGetUsersOfGroup() throws Exception
2   {
3       PropertyConfigurator.configure(LOG_CONF);
4       initHome(BONITA_HOME);
5       Logger log = Logger.getLogger("org.cs.bonita.apitest");
6
7       // Megszerzi a login-t végrehajtó objektumot
8       final LoginAPI loginAPI = TenantAPIAccessor.getLoginAPI();
9       log.info("login with install//install");
10
11      // Bejelentkezés a Bonita motorba
12      final APISession session = loginAPI.login("install", "install");
13      final IdentityAPI identityAPI = TenantAPIAccessor.getIdentityAPI(session);
14
15      // Lekérjük a KevenceimGroup tagjait
```

```
16      Group group = identityAPI.getGroupByPath("KevenceimGroup");
17      final SearchOptionsBuilder builder = new SearchOptionsBuilder(0, 100);
18      builder.filter( UserSearchDescriptor.GROUP_ID, group.getId() );
19      final SearchResult<User> userResults = identityAPI.searchUsers(builder.done());
20
21      // long count = userResults.getCount();
22      List<User> users = userResults.getResult();
23      for ( User user : users )
24      {
25         System.out.println(   user.getLastName()   );
26      }
27
28      loginAPI.logout(session);
29      return;
30  }
```

A 17-18 sorokban ismét megjelenik a *SearchOptionsBuilder* class, hiszen most is keresünk. Keressük azon user-ek halmazát, amik egy megadott csoportba tartoznak, így a 18. sorban a *builder* változóra még egy *filter()* hívást is tettünk, azaz szűkítettük a kört a mi csoportunkra, azaz a *KevenceimGroup*-ra. A 19. sor hívja le a *User* objektumok eredmény listáját.

## Egy megadott group adott role-ban lévő user-einek listája

Az előző feladatot egészítsük egy kicsit ki, mert csak azt a user halmazt szeretnénk megkapni, akik a csoporton belül egy megadott role-ban vannak. A megoldás (7-17. Programlista) az lesz, hogy a filter-t kiegészítjük egy újabb feltétellel (21-22 sorok), ahogy a *testGetUsersOfGroup()* metódus is csinálja.

7-17. Programlista: testGetUsersOfGroup() - Csoportba és szerepkörbe tartozás

```
1   private static void testGetUsersOfGroup() throws Exception
2   {
3       PropertyConfigurator.configure(LOG_CONF);
4       initHome(BONITA_HOME);
5       Logger log = Logger.getLogger("org.cs.bonita.apitest");
6
7       // Megszerzi a login-t végrehajtó objektumot
8       final LoginAPI loginAPI = TenantAPIAccessor.getLoginAPI();
9       log.info("login with install//install");
10
11      // Bejelentkezés a Bonita motorba
12      final APISession session = loginAPI.login("install", "install");
13      final IdentityAPI identityAPI = TenantAPIAccessor.getIdentityAPI(session);
14
15      // Lekérjük a KevenceimGroup tagjait
16      Group group = identityAPI.getGroupByPath("KevenceimGroup");
17      final SearchOptionsBuilder builder = new SearchOptionsBuilder(0, 100);
18      builder.filter( UserSearchDescriptor.GROUP_ID, group.getId() );
19
20      // Ez a további filter feltétel
21      Role role = identityAPI.getRoleByName("Member");
22      builder.filter(UserSearchDescriptor.ROLE_ID, role.getId());
23
24      final SearchResult<User> userResults = identityAPI.searchUsers(builder.done());
25
26      // long count = userResults.getCount();
27      List<User> users = userResults.getResult();
28      for ( User user : users )
29      {
```

```
30          System.out.println(   user.getLastName()   );
31      }
32
33      loginAPI.logout(session);
34      return;
35  }
```

## Egy új process példány létrehozása és indítása

Ebben a részben egy izgalmas témát tekintünk át, nevezetesen azt, hogyan tudunk egy process-t létrehozni és elindítani.

### Első példa - minden process változó String

Az első példában egy kicsit egyszerűsítünk és az elindított process összes belső változója *String* típusú lesz. Nézzük meg elöljáróban az általános célú *buildAssignOperation()* metódust (7-18. Programlista).

7-18. Programlista: buildAssignOperation()

```
1   /**
2    *
3    * @param dataInstanceName – A változó neve
4    * @param newConstantValue – Egy érték a változó számára
5    * @param className – A változó típusának (azaz osztályának a neve)
6    * @param expressionType – Az a lehetőség, ami a Stúdióban is van, azaz konstans, Groovy script,➥
         stb.
7    * @return
8    * @throws InvalidExpressionException
9    */
10  private static Operation buildAssignOperation( final String dataInstanceName,
11                                                 final String newConstantValue,
12                                                 final String className,
13                                                 final ExpressionType expressionType)
14                                                         throws InvalidExpressionException
15  {
16      final LeftOperand leftOperand = new LeftOperandBuilder()
17              .createNewInstance().setName(dataInstanceName).done();
18
19      final Expression expression = new ExpressionBuilder()
20              .createNewInstance(dataInstanceName)
21              .setContent(newConstantValue)
22              .setExpressionType(expressionType.name())
23              .setReturnType(className).done();
24
25      final Operation operation;
26      operation = new OperationBuilder().createNewInstance().setOperator("=")
27              .setLeftOperand(leftOperand).setType(OperatorType.ASSIGNMENT)
28              .setRightOperand(expression).done();
29
30      return operation;
31  }
```

Az *org.bonitasoft.engine.operation.Operation* interface egy műveletet reprezentál a Bonitában, ami 3 részből épül fel:

1. Egy balérték, ahogy azt a C++ terminológia is érti, azaz egy olyan változó, amihez egy értéket tudunk rendelni (az értékadó utasítás bal oldalán áll).

2. Egy operátor típus.

3. Egy kifejezés (jobb érték a C++ terminológiában), amit ki kell számítani. Ez megegyezik azokkal a választási lehetőségekkel, amit a Bonita Stúdió is felkínál, amikor a kifejezés szerkesztőben dolgozunk (konstans, változó, Groovy script).

A *buildAssignOperation()* metódus megkap minden paramétert, ami egy *Operation* objektum létrehozásához szükséges. A *leftOperand* egy változó dinamikus reprezentációja, azaz egy változót jelent, aminek a példában a *dataInstanceName* lesz a neve. Az létrehozott *expression* objektum egy teljes értékadó kifejezés, ahol megadjuk a következőket:

1. A bal oldali változó nevét, ami ismét a *dataInstanceName*.

2. A jobb oldali értéket, ami most *newConstantValue*.

3. Deklaráljuk, hogy ez a kifejezés milyen típusú: *expressionType.name()*, azaz konstans, Groovy, változó, ...

4. Végül természetesen minden kifejezésnek van egy visszatérési típusa: *className*.

A *buildAssignOperation()* metódus nem bonyolult, de nagyon hasznos, mert segítségével újra és újra sokkal könnyebben inicializálhatjuk a létrehozott új process változóit. Nézzük meg, hogy magát a process-t hogyan tudjuk létrehozni, inicializálni és elindítani (7-19. Programlista). Láthatjuk, hogy használjuk a fentiekben bemutatott *buildAssignOperation()* metódust.

7-19. Programlista: createInstance() - A process példány létrehozása

```
1  /**
2   * Egy process létrehozása igényli ezeket a paramétereket:
3   *
4   * @param processDefinitionName - A process BPMN neve
5   * @param processVersion - A process mindig egy verzióval rendelkezik
6   * @param variables - Változók Map-ja (változó név, érték) párok
7   * @param apiSession - A login után kapott session object
8   */
9  private static void createInstance( String processDefinitionName,
10                                     String processVersion,
11                                     Map<String, Object> variables,
12                                     APISession apiSession)
13 {
14    ProcessAPI processAPI;
15    try
16    {
17       processAPI = TenantAPIAccessor.getProcessAPI(apiSession);
18
19       // Létrehoz egy új process-t (még nem indul el)
20       long processId = processAPI.getProcessDefinitionId(processDefinitionName, processVersion);
21
22       // Műveletek listája
23       List<Operation> listOperations = new ArrayList<Operation>();
24       for (String variableName : variables.keySet())
25       {
26          if (variables.get(variableName) == null) continue;
27          Operation operation = buildAssignOperation( variableName,
28                      variables.get(variableName).toString(),
29                               String.class.getName(),
```

```
                        ExpressionType.TYPE_CONSTANT);
            listOperations.add(operation);
        }

        // A process most fog indulni
        processAPI.startProcess(processId, listOperations, null);
    } catch (Exception e)
    {
        e.printStackTrace();
    }
}
```

A *createInstance()* első lépésben egy a Bonita Stúdióban megtervezett nevű és verziójú process példányt hoz létre, amire a későbbiekben a *processId* long típusú azonosítóval tudunk hivatkozni. A process változóinak inicializálása a következő lépés, amihez az input egy *variables* nevű *Map* objektum. A példánkban említettük, hogy most minden változó *String* típusú lesz, ez az oka annak, hogy a *buildAssignOperation()* 3. paramétere a Java *String* típus teljes neve lett. A 4. paraméter pedig azt mondja meg, hogy itt jobb oldalon egy konstans lesz. A *Map* alapján gyártott *Operation* objektumok a végén a *listOperations* listában gyűlnek össze. A metódus utolsó művelete a *processAPI.startProcess()* hívás, amivel a process instance megjelenik az engine számára.

### Második példa - A process változók tetszőleges típusúak

A következő példa (7-20. Programlista) már nem tételezi fel, hogy minden változó *String* típusú, ezért ez az általános módszer egy workflow instance létrehozására.

7-20. Programlista: createAndStartCase()

```
public void createAndStartCase( String processDefinitionName,
                                 String processVersion,
                                 Map<String, Object> variables,
                                 ProcessAPI processAPI)
{
    try
    {
        long processId = processAPI.getProcessDefinitionId(processDefinitionName, processVersion);

        // ------ create list of operations ------
        List<Operation> listOperations = new ArrayList<Operation>();
        Map<String, Serializable> listVariablesSerializable = new HashMap<String, Serializable>();

        for (String variableName : variables.keySet())
        {
            if (variables.get(variableName) == null
                    || (!(variables.get(variableName) instanceof Serializable)))
                continue;
            Object value = variables.get(variableName);
            Serializable valueSerializable = (Serializable) value;

            variableName = variableName.toLowerCase();
            Expression expr = new ExpressionBuilder().createExpression(
                                        variableName,
                                        variableName,
                                        value.getClass().getName(),
                                        ExpressionType.TYPE_INPUT);
            Operation op = new OperationBuilder().createSetDataOperation(variableName, expr);
            listOperations.add(op);
            listVariablesSerializable.put(variableName, valueSerializable);
```

```
31          }
32
33          // ----- start process instance -----
34          processAPI.startProcess(processId, listOperations, listVariablesSerializable);
35
36          // System.out.println("*** End Create Case (process) ****");
37
38      } catch (Exception e)
39      {
40          e.printStackTrace();
41      }
42  }
```

A fenti *createAndStartCase()* metódus már képes olyan process létrehozására, ahol annak változói tetszőleges, de a *Serializable* interface-t megvalósító Java típus lehet. Ennek tárolása egy külön *Map* adatszerkezetben történik, aminek a kulcsa természetesen a változó neve (azaz *String* a kulcs). Ezen metódus egy fontos része lehet egy facade (homlokzat) mintájú megközelítésnek, ahol eltakarjuk a Bonita belső jellegzetességeit.

## A Bonita Engine-be telepített process típusok listája

Nézzük meg, hogy milyen módon tudjuk visszakeresni a rendszerbe telepített process típusok listáját (7-21. Programlista)!

7-21. Programlista: testProcessList() - A process típusok listája

```
1   private static void testProcessList() throws Exception
2   {
3       PropertyConfigurator.configure(LOG_CONF);
4       initHome(BONITA_HOME);
5       Logger log = Logger.getLogger("org.cs.bonita.apitest");
6
7       // Megszerzi a login-t végrehajtó objektumot
8       final LoginAPI loginAPI = TenantAPIAccessor.getLoginAPI();
9       log.info("login with install//install");
10
11      // Bejelentkezés a Bonita motorba
12      final APISession session = loginAPI.login("walter.bates", "bpm");
13      // final APISession session = loginAPI.login("install", "install");
14
15      // A process-ek lekérése
16      final ProcessAPI processAPI = TenantAPIAccessor.getProcessAPI(session);
17
18      final SearchOptionsBuilder builder = new SearchOptionsBuilder(0, 100);
19
20      final SearchResult<ProcessDeploymentInfo> deploymentInfoResults = processAPI
21              .searchProcessDeploymentInfos(builder.done());
22
23      // Listázás
24      System.out.println("Telepített process fejlesztések. user="
25              + session.getUserName());
26
27      List<ProcessDeploymentInfo> processes = deploymentInfoResults.getResult();
28
29      for (final ProcessDeploymentInfo process : processes)
30      {
31          log.info(" * " + process.getDisplayName() + " -- "
32                  + process.getId());
33      }
```

```
34
35      // Kijelentkezés a Bonita motorból
36      loginAPI.logout(session);
37      log.info("logged out");
38  }
```

A *testProcessList()* működése szintén a *SearchOptionsBuilder* objektumra épül, ami a keresési opciókat szedi össze a builder (építő) tervezési minta szerint. A *deploymentInfoResults* tartalmazza a telepített process alkalmazások listáját, amit a builder opciói szerint kérdeztünk le. Magát a process listát pedig a *processes* lista tartalmazza. A program futási eredménye a jelenleg telepített egyetlen process alkalmazást tartalmazza:

```
login with install//install
 * TestPool -- 2
logged out
```

## Adott felhasználóhoz rendelt process példányok listájának megszerzése

Amikor saját alkalmazással kezeljük a Bonita motort, gyakori feladat az, hogy egy user milyen process instance-okat lát. A *testMyProcessList()* pont ezt a feladatot oldja meg (7-22. Programlista).

7-22. Programlista: testMyProcessList()

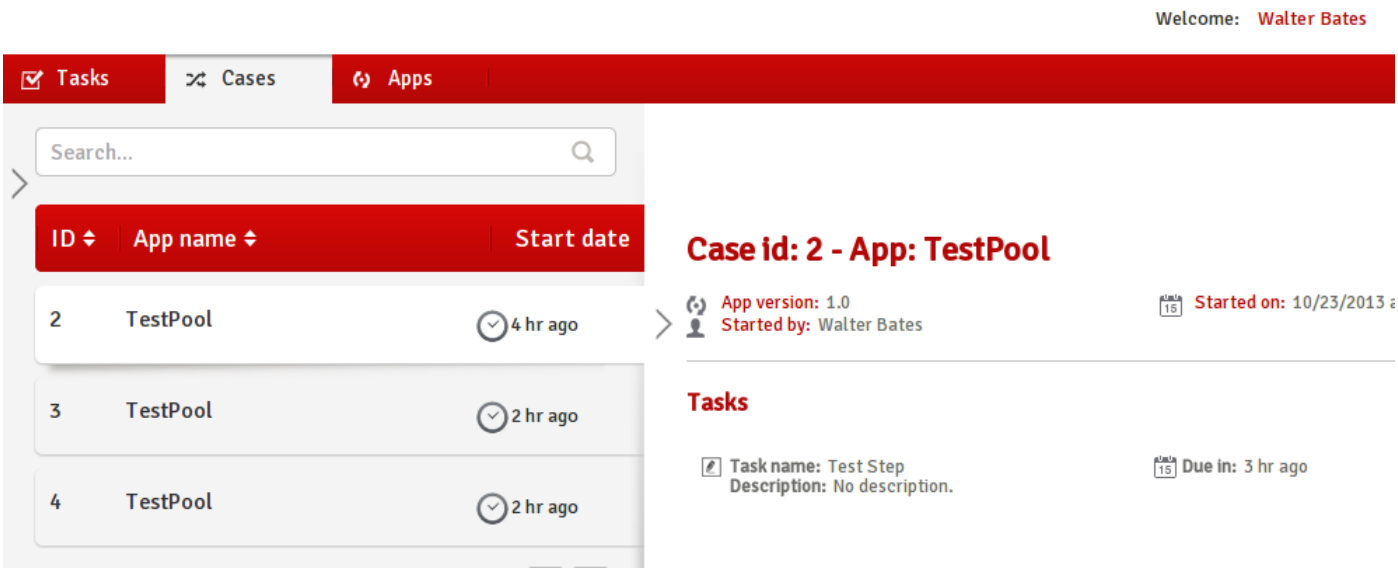
```
1   private static void testMyProcessList() throws Exception
2   {
3       PropertyConfigurator.configure(LOG_CONF);
4       initHome(BONITA_HOME);
5       Logger log = Logger.getLogger("org.cs.bonita.apitest");
6
7       // Megszerzi a login-t végrehajtó objektumot
8       final LoginAPI loginAPI = TenantAPIAccessor.getLoginAPI();
9       log.info("login with install//install");
10
11      // Bejelentkezés a Bonita motorba
12      final APISession session = loginAPI.login("walter.bates", "bpm");
13      // final APISession session = loginAPI.login("install", "install");
14
15      // A process-ek lekérése
16      final ProcessAPI processAPI = TenantAPIAccessor.getProcessAPI(session);
17
18      final SearchOptionsBuilder builder = new SearchOptionsBuilder(0, 100);
19
20      builder.filter(ProcessInstanceSearchDescriptor.STARTED_BY, session.getUserId());
21      final SearchResult<ProcessInstance> processInstanceResults = processAPI.➥
            searchOpenProcessInstances(builder.done());
22
23
24      List<ProcessInstance> processes = processInstanceResults.getResult();
25      // Listázás
26      System.out.println("Telepített process fejlesztések. user="
27              + session.getUserName() + " Darab="+processes.size());
28
29
30
31      for (final ProcessInstance process : processes)
32      {
33          log.info("* " + process.getName() + " Process Instance ID=" + process.getId());
```

```
34      }
35
36      // Kijelentkezés a Bonita motorból
37      loginAPI.logout(session);
38      log.info("logged out");
39  }
```

Nézzük meg előbb a futtatás outputját, utána pár szóban elmagyarázzuk a működést:

```
login with install//install
 * TestPool Process Instance ID=4
 * TestPool Process Instance ID=2
 * TestPool Process Instance ID=3
logged out
```

Ez azt jelenti a *walter.bates* nevű user-nek jelenleg 3 processe (azaz case-e) van, amik 4, 2 és 3 process instance azonosítóval érhetőek el. Mint eddig is sokszor, a visszakeresésnél a *SearchOptionsBuilder* osztályt használtuk, de a maximum 100 elemű eredmény feltételét még kiegészítettük azzal, hogy ki indította el a process-t. A keresés a *processAPI* változó és a *builder* által megvalósított szűrés együttműködésére épül, ami az eddigi példák ismeretében talán már nem meglepő.




7.6. ábra. A portálon is ezt a 3 case-t láthatjuk a *walter.bates* user-hez

A jó működést a 7.6. ábra is mutatja, azaz a Bonita Portálon is pont ezek a process instance-ok jelennek meg.

## A process változóinak írása és olvasása

Egy nagyon fontos ponthoz érkeztünk, bemutatjuk, hogy egy tetszőleges Java *Object* típusú adatot milyen módon tudunk elérni, illetve megváltoztatni. Ehhez a *com.anura.commondatatypes.CDTCheckboxHolde* típusú *stringList* workflow változót fogjuk használni. A tesztprogramot a 7-23. Programlista tartalmazza.

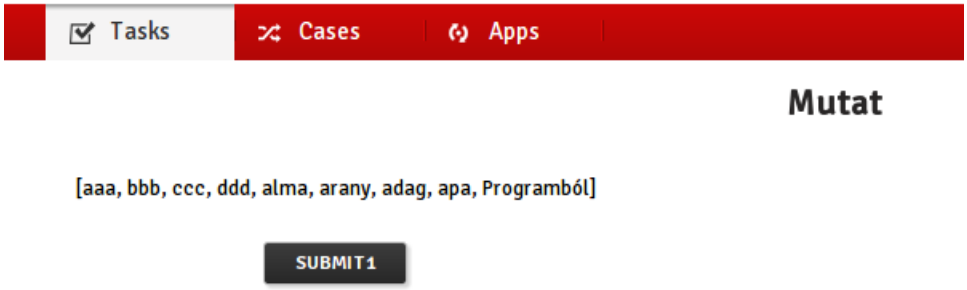
7-23. Programlista: testProcessVars() - A workflow változók írása, olvasása

```
1  private static void testProcessVars() throws Exception
2  {
3      PropertyConfigurator.configure(LOG_CONF);
4      initHome(BONITA_HOME);
5      Logger log = Logger.getLogger("org.cs.bonita.apitest");
6
7      // Megszerzi a login-t végrehajtó objektumot
8      final LoginAPI loginAPI = TenantAPIAccessor.getLoginAPI();
9      log.info("login with install//install");
10
11     // Bejelentkezés a Bonita motorba
12     final APISession session = loginAPI.login("walter.bates", "bpm");
13     // final APISession session = loginAPI.login("install", "install");
14
15     // A process-ek lekérése
16     final ProcessAPI processAPI = TenantAPIAccessor.getProcessAPI(session);
17
18     //ProcessInstance pinst = processAPI.getProcessInstance(5);
19
20     DataInstance dinst = processAPI.getProcessDataInstance("stringList", 5);
21
22     com.anura.commondatatypes.CDTCheckboxHolder adat = (com.anura.commondatatypes.➥
           CDTCheckboxHolder)dinst.getValue();
23
24     log.info( adat.getLabels() );
25
26     adat.getLabels().add("Programból");
27
28     log.info( adat.getLabels() );
29
30     processAPI.updateProcessDataInstance("stringList", 5, adat);
31
32     // Kijelentkezés a Bonita motorból
33     loginAPI.logout(session);
34     log.info("logged out");
35 }
```

A 20. sor lekéri a 5-ös azonosítójú process példány *stringList* változó nevű adatát, ami a fent említett osztálybeli objektum (2 darab *String* lista tárolására alkalmas). A *dinst* változó egy *DataInstance*, amitől már elkérhető a serializált *CDTCheckboxHolder* típusú objektum is. Előbb kiírjuk a jelenlegi értékét az egyik listának (a *label*-eknek), majd egy értékadással ezt megváltoztatjuk, utána ismét kiírjuk. Ez lesz a futási eredmény:

```
login with install//install
[aaa, bbb, ccc, ddd, alma, arany, adag, apa]
[aaa, bbb, ccc, ddd, alma, arany, adag, apa, Programból]
logged out
```

Minden tökéletesen úgy működött, ahogy szerettük volna, de csináljunk egy próbát! Indítsuk el az üzleti folyamatot, ami 2 lépéses. Az első lépés TASK-ja mutatja a lista első, induló változatát, azaz az *apa* a vége. Mielőtt átmegyünk a *Mutat* TASK-ra, futassuk le ezt a külső programot, aminek a hatására a lista ki fog egészülni egy új, *Programból* nevű elemmel. Ezután menjünk a 2. (*Mutat* nevű) TASK-ra, ami csak megjeleníti a lista tartalmát. Az eredményt a 7.7. ábra mutatja, sikeresen elvégeztük a workflow változó frissítését.




7.7. ábra. A külső program sikeresen megváltoztatta a *stringList* process változót

## Egy felhasználó TASK-jainak lekérése

Amikor egy TASK a mi látókörünkbe kerül, akkor azt úgy látjuk, hogy a megvalósítása függőben van, azaz PENDING állapotba került. A Bonita Portál *To Do* részénél látjuk az összes TASK-unkat. Amennyiben ezt *TAKE*-kel magunkhoz vesszük, akkor az *Assigned* lesz, ezt a *RELEASE* segítségével tudjuk visszatenni a TASK kosárba, ahonnan azt mások is ki tudják venni. A *HIDE* elrejti a TASK-ot, amit a *RETRIEVE* segítségével lehet ismét láthatóvá tenni a kosárban.

Az összes PENDING TASK listázását teszi lehetővé a *testGetPendingTasks()* metódus (7-24. Programlista).

7-24. Programlista: testGetPendingTasks() - A felhasználó feladatainak lekérése

```
1  private static void testGetPendingTasks() throws Exception
2  {
3      PropertyConfigurator.configure(LOG_CONF);
4      initHome(BONITA_HOME);
5      Logger log = Logger.getLogger("org.cs.bonita.apitest");
6
7      // Megszerzi a login-t végrehajtó objektumot
8      final LoginAPI loginAPI = TenantAPIAccessor.getLoginAPI();
9      log.info("login with install//install");
10
11     // Bejelentkezés a Bonita motorba
12     final APISession session = loginAPI.login("walter.bates", "bpm");
13     // final APISession session = loginAPI.login("install", "install");
14
15     // A process-ek lekérése
16     final ProcessAPI processAPI = TenantAPIAccessor.getProcessAPI(session);
17
18     // Pending TASKs
19     final List<HumanTaskInstance> pendingTasks = processAPI
20             .getPendingHumanTaskInstances(session.getUserId(), 0, 20, ActivityInstanceCriterion.➥
                   PRIORITY_ASC);
21
22     System.out.println("Pending tasks for user " + session.getUserName()
23             + ": " + pendingTasks);
24
25     // Kijelentkezés a Bonita motorból
26     loginAPI.logout(session);
27     log.info("logged out");
28 }
```

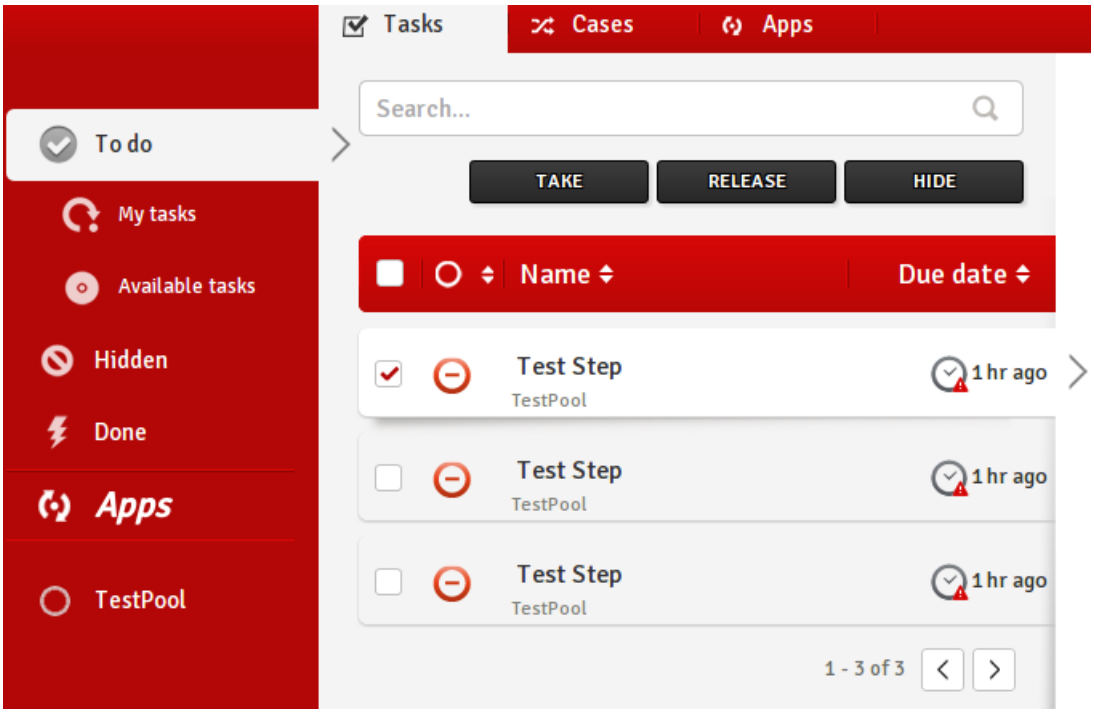
A futási eredmény:

```
Pending tasks for user walter.bates: [FlowNodeInstanceImpl [parentContainerId=5, state=ready, ➥
    stateCategory=NORMAL, rootContainerId=5, processDefinitionId=5039137620009662369, ➥
    parentProcessInstanceId=5, displayDescription=null, displayName=Mutat, description=null, ➥
    executedBy=0, flownodeDefinitionId=-5430170137475346420]]
```

Amikor a *HumanTaskInstance* objektum a kezünkben van, akkor pedig már azt csináljuk vele, amit szeretnénk. A magunkhoz vett (*TAKE*) taskok listáját így kapjuk meg:

```
final List<HumanTaskInstance> allTasks = processAPI.
      getAssignedHumanTaskInstances(session.getUserId(), 0, 20,                    ➥
                                                                  ActivityInstanceCriterion.➥
          PRIORITY_ASC);
```

Befejezésül nézzük meg, hogy milyen utasítással tudunk egy PENDING TASK-ot ASSIGNED-be tenni. A 7.8. ábra mutatja, hogy van 3 pending TASK-unk.



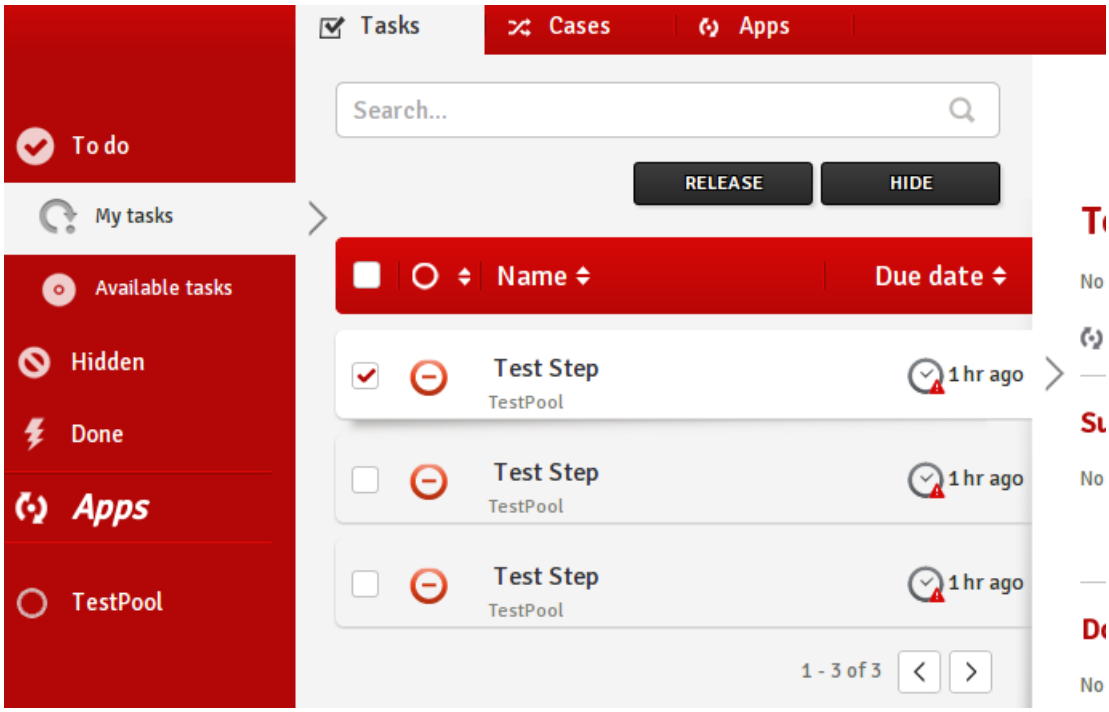

7.8. ábra. PENDING taszkok a portálon megtekintve

Ezzel a kis programrésszel mindegyiket egy felhasználóhoz tudjuk rendelni, azaz a *My Tasks* mappában kerülnek:

```
final List<HumanTaskInstance> pendingTasks = processAPI
   .getPendingHumanTaskInstances(session.getUserId(), 0, 20,
                          ActivityInstanceCriterion.PRIORITY_ASC);

for (HumanTaskInstance pendingTask : pendingTasks)
{
   // assign the task to the user
   processAPI.assignUserTask(pendingTask.getId(), session.getUserId());
   // execute the task
```

```
    //processAPI.executeFlowNode(pendingTask.getId());
}
```

Ez a portálon úgy jelenik meg, hogy ez a 3 TASK most a *My Tasks* mappába került (7.9. ábra).




7.9. ábra. A 3 darab TASK átkerült a My Task mappába

# 8. Bonita 6. konnektorok áttekintése

A konnektorok biztosítják, hogy a Bonita munkafolyamatok integrálódhassanak a környezetükhöz. Ez egyaránt vonatkozik az input (amikor a munkafolyamat igényel valamilyen adatot) és output (a munkafolyamat küld egy másik rendszer számára információt) jellegű kommunikációra. Tekintettel arra, hogy sokféle információs rendszer létezik, rengeteg Bonita konnektor érhető el, de magunk is készíthetünk újakat.

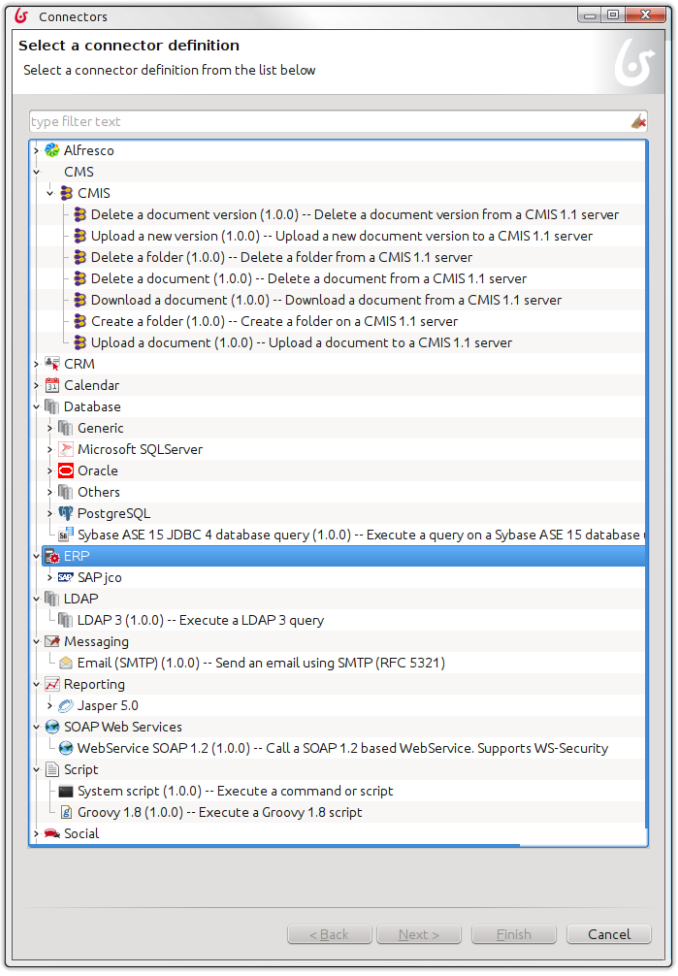
## A konnektorok áttekintése

A konnektorok biztosítják, hogy a munkafolyamat bizonyos pontjairól a külvilág rendszereivel kommunikálhassunk. Ez jelenthet adatküldést, fogadást vagy mindkét irányba való adatcserét is. Konnektor adhatunk a *pool* (munkafolyamat), *lane* (egy úszósáv) és *task* (egy feladat) szintjén is az elkészítendő workflow-hoz. A konnektort input adatait a workflow belső változói, illetve egy tetszőleges script által előállított futási eredmény állíthatja elő. Ezután a konnektor lefut és tetszőleges számú eredmény értékkel (output) térhet vissza, amit a workflow átvesz és eltárol. Egy kész konnektor használata mindig ezekből a főbb lépésekből áll:

1. A Bonita Stúdióban álljunk rá arra a pool, lane vagy step elemre, amihez egy konnektort szeretnénk rendelni.

2. A *General→Connectors* fület válasszuk ki és nyomjuk meg az *Add...* gombot, amire megjelenik a 8.1. ábrán látható ablak. Válasszuk ki azt a konnektort, amit használni szeretnénk!

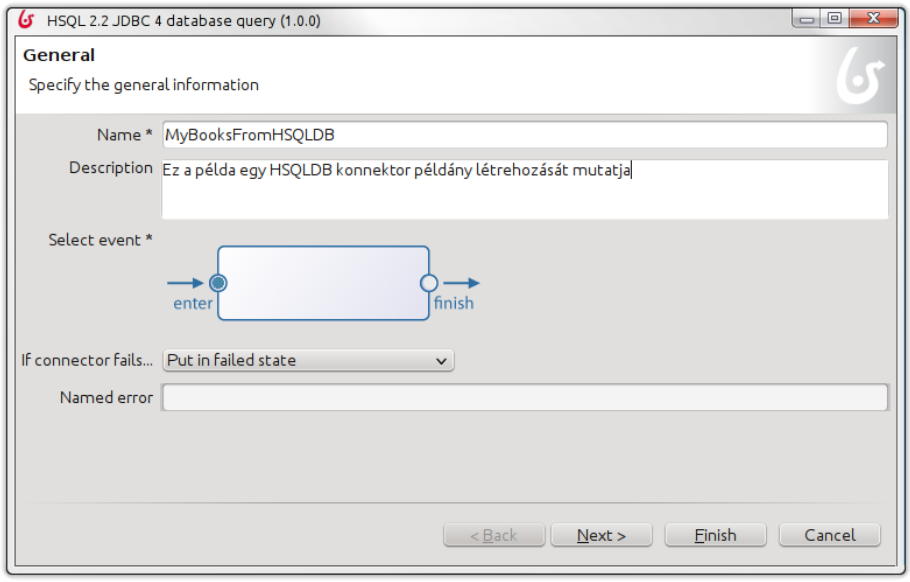
3. Kövessük a varázsló útmutatásait és adjuk meg a konnektor által megkövetelt input és output adatokat, konfigurációs részleteket és a mapping szabályokat (a workflow változói és a konnektor input/output adatai közötti leképzés).



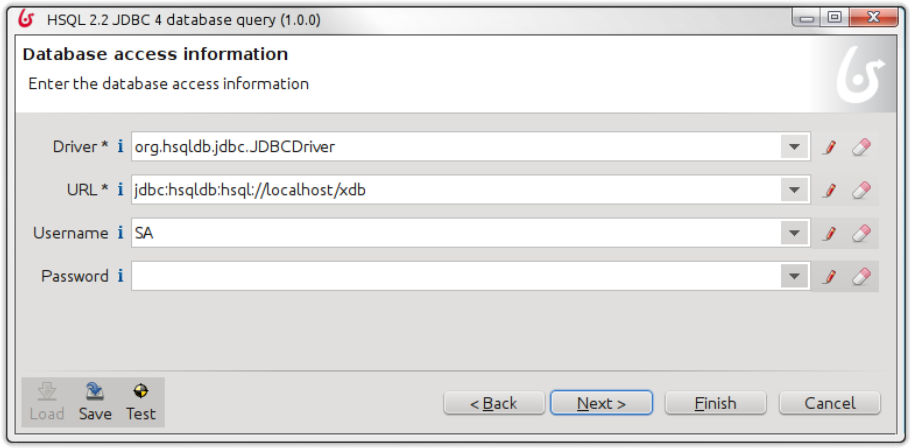

8.1. ábra. A fontosabb konnektorok

## Az adatbázis konnektor

Bármely adatbázis-kezelő felé kapcsolódhatunk a 8.1. ábrán látható *Generic* lehetőség kiválasztásával, a többi megnevezett RDBMS lehetőség csak annyiban különbözik ettől, hogy arra az adatbázisra testre szabott a varázslója, így gyorsabb egy kicsit egy új kapcsolat bekonfigurálása. A továbbiakban bemutatunk egy konkrét RDBMS konnektor példány létrehozását, amihez az Informatikai Navigátor 7. számában ismertetett *HSQLDB* adatbázisunkat fogjuk használni. Válasszunk ki például egy step-et és jussunk el az ismertetett módon a 8.1. ábra popup ablakáig. A *Database→Others* kibontása után fogjuk látni a HSQL 2.2 JDBC kapcsolat kiválasztásának lehetőségét, ami után a varázsló 8.2. ábrán látható első ablaka jön be.

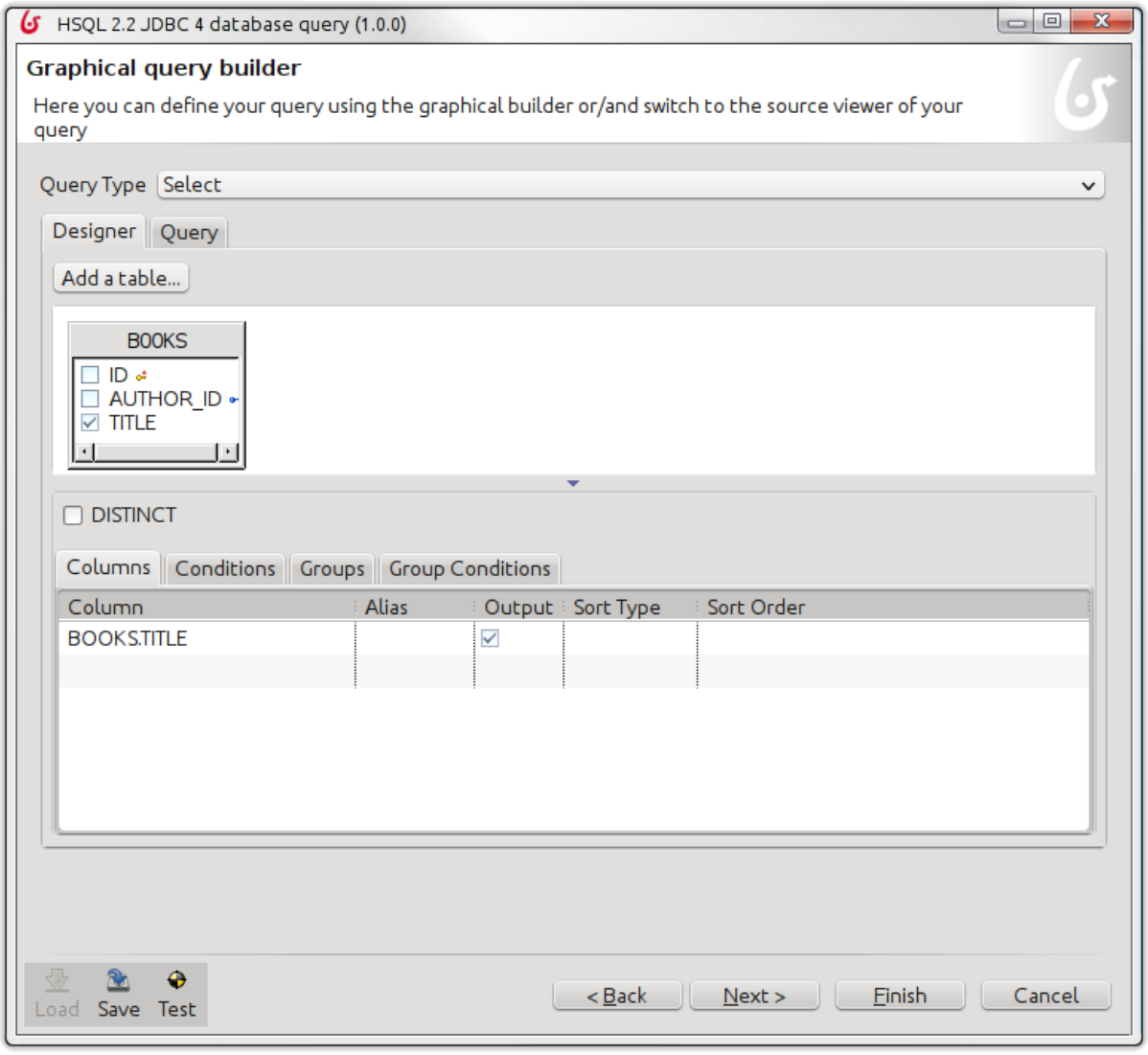



8.2. ábra. RDBMS (HSQLDB) Konnektor példány létrehozása - 1




8.3. ábra. RDBMS (HSQLDB) Konnektor példány létrehozása - 2

Megadhatjuk, hogy a konnektor a BPMN elembe való belépéskor (*enter*) vagy kilépéskor (*finish*) csattanjon-e el. A *Name* mező a most létrehozás alatt lévő konnektor példány azonosítója. Vegyük észre, hogy a konnektor hibáját is tudjuk kezelni, amennyiben megadunk egy *Named error* stringet, úgy azt az események használatánál bemutatott Error Event képes elkapni és egy kivételes ágon haladva lekezelni (lásd a 6. fejezet végét). A *Next* után láthatjuk az *Add or select database driver*: *hsqldb.jar* kiválasztási lehetőséget, ami rendben van, majd a következő ablakban a JDBC connection paraméterek megadása következhet (8.3. ábra).



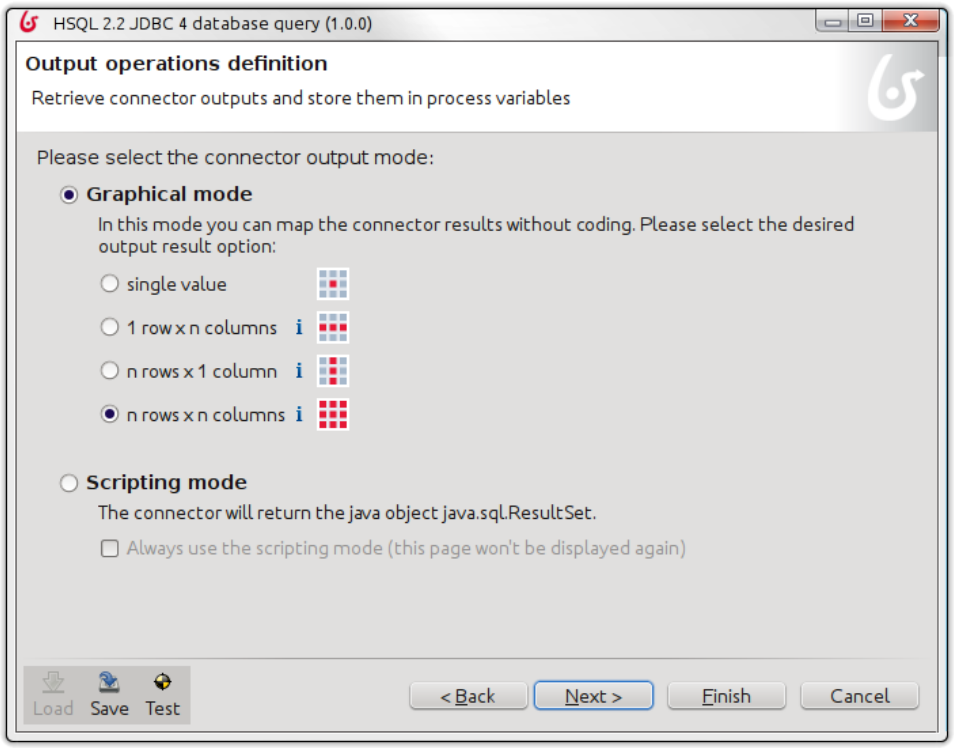

8.4. ábra. RDBMS (HSQLDB) Konnektor példány létrehozása - 3

A *Next* gombra következik a 8.4. ábrán látható ablak. A *Query Type* select, insert, update, delete lehet. A felület lehetőséget ad arra, hogy *Designer* vagy *Query* módban szerkesszük az SQL parancsot, sőt ide-oda is kapcsolgathatunk a kettő között. A példa adatbázisban van egy *books*

nevű tábla, amit az *Add a table...* gomb megnyomása után választottunk ki. A tábla mezői közül csak a *title* oszlopot kérjük most, aminek a *Query* fül nézete ilyen lett:

```
SELECT TITLE FROM PUBLIC.BOOKS
```

A *Designer* fülről még a *Conditions*, *Groups*, *Group Conditions* lehetőségek vizuális szerkesztése is lehetséges. Az SQL parancs összeállítása után a *Next* gombra kattintva jön be a 8.5. ábra ablaka.

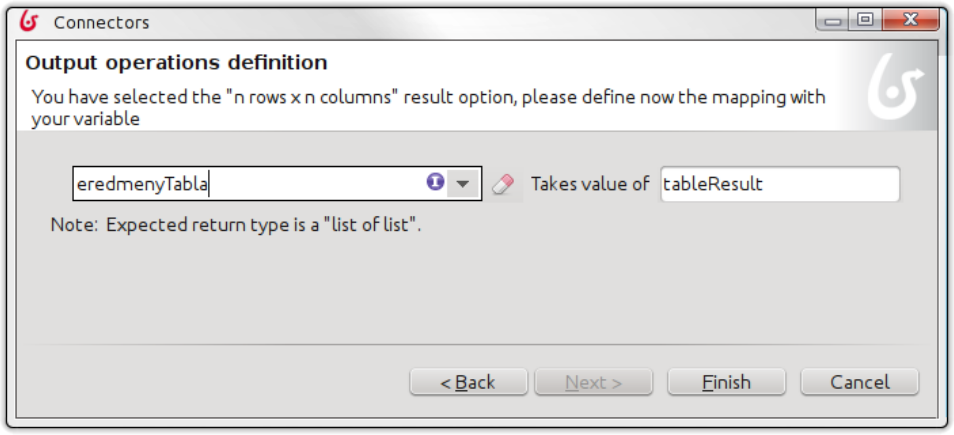



8.5. ábra. RDBMS (HSQLDB) Konnektor példány létrehozása - 4

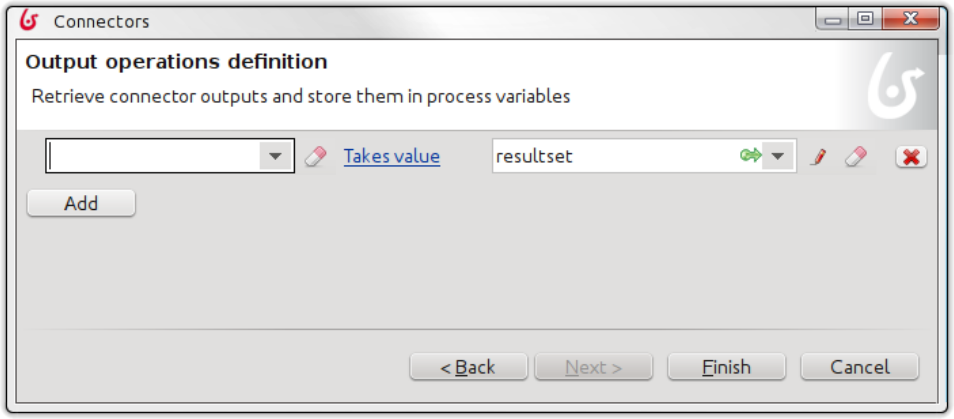
Itt található 2 különböző beállítási lehetőség:

1. *Graphical mode*: Ekkor a konnektor lefutása utáni output paramétereket a 8.6. ábra szerinti *List<List<Object>>* típusú változó fogja fogadni, ami listák listája, azaz egy táblázat lehetséges reprezentációja.

2. *Scripting mode*: Amennyiben ezt választjuk, úgy a konnektor lefutása utáni output paramétert a 8.7. ábra szerinti ablakkal adhatjuk meg. Ekkor egy Java *java.sql.ResultSet* lesz a visszaadott érték, azaz a workflow fogadó változójának is olyannak kell lennie. Ebben az esetben ezt az SQL kurzort programmal kell feldolgozni.




8.6. ábra. RDBMS (HSQLDB) Konnektor példány létrehozása - 5




8.7. ábra. RDBMS (HSQLDB) Konnektor példány létrehozása - 6

A *Graphical mode* rejti azt a lehetőséget, hogy csak minimális programozással vegyük át az SQL parancs futási eredményét. Ennek 4 féle módja lehetséges, ahogy azt a 8.5. ábra is prezentálja:

1. *single value*: Ekkor a 8.6. ábra ablaka olyan szerkezetű lesz, hogy csak a kiválasztott egyetlen oszlop értéket lehet a workflow megfelelő típusú változójához rendelni. Ekkor esetünkben a *Take value of* cella értéke fix *TITLE* lesz és azt választhatjuk csak ki, hogy mely változóba mentjük azt. Amennyiben az SQL parancsban több oszlop van, úgy ez az opció nem is választható ki.

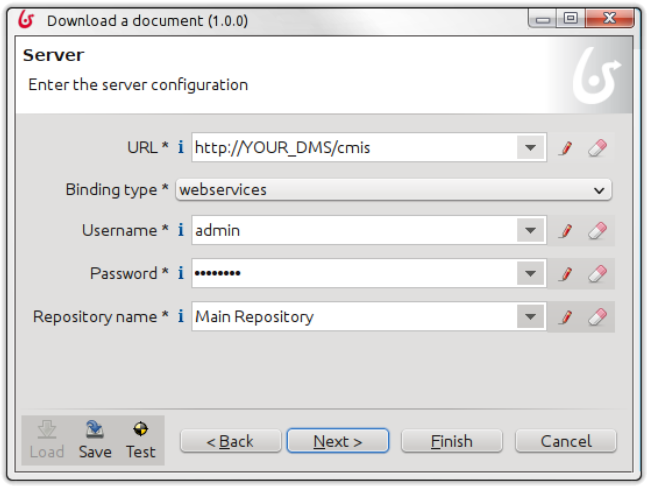
2. *1 row x n columns*: Ekkor szintén gondoskodnunk kell arról, hogy az eredménytábla 1 soros legyen, de itt az SQL parancs minden oszlopát hozzárendelhetjük valamelyik belső workflow változóhoz. Ez azt jelenti, hogy a 8.6. ábra ablaka olyan szerkezetű lesz, ami ezt lehetővé teszi.

3. *n rows x 1 column*: Ebben az esetben több soros lehet az eredménytábla, de csak 1 oszlopos, aminek az értékét egy *List* típusú változóba menthetjük.

4. *n rows x n columns*: Ezt az esetet mutatja a 8.6. ábra.

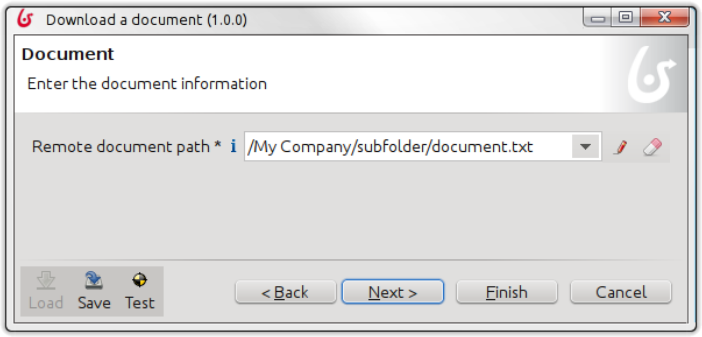
A fenti beállításokkal kész a konnektor példány. Vegyük észre, hogy a grafikus mód, programozás nélkül is nagyon le tudja egyszerűsíteni a workflow adatainak inicializálását vagy mentését.

## A CMIS konnektor




8.8. ábra. A CMIS konnektor konfigurálása - 1

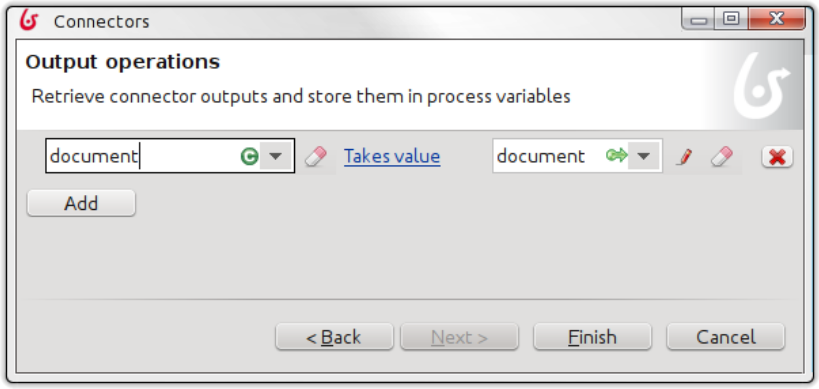
A *CMIS*[7] egy szabvány a különböző content management rendszerek általános elérésére. A *CMIS* definiál egy adatmodellt, ami egy interface adatszerkezet a különféle DMS rendszerek felé. Itt megtalálhatóak azok a fogalmak, mint például a típusos fájl, a folder és annak jellemzői. A 8.1. ábra mutatja a *CMIS* konnektor kategória alatt elérhető konnektorokat, amik a DMS rendszerek fölött értelmezhető műveleteket valósítják meg: upload, download, .... Amelyik rendszer a check-in és check-out (verziókezelés) műveleteket is támogatja, az ily módon érhető el a *CMIS* fölött. A *CMIS* és emiatt a Bonita konnektor 2 féle elérési protokoll kötést támogat mindehhez: SOAP és REST.




8.9. ábra. A CMIS konnektor konfigurálása - 2

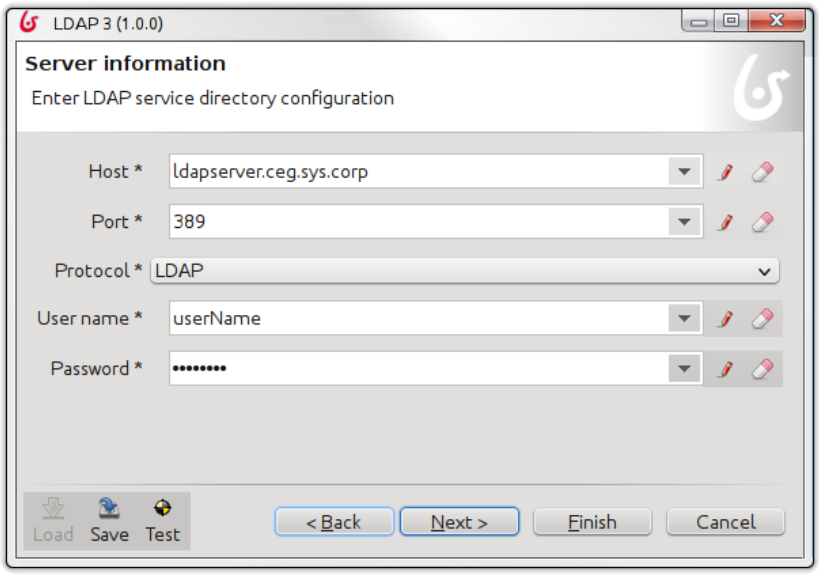
[7] CMIS=Content Management Interoperability Services

A 8.8. ábra egy éppen elkezdett Webservices alapú *CMIS* konnektor konfigurálás első lépését mutatja, a megadandó paraméterek egyértelműek. Fontos megemlíteni, hogy az összes ismert DMS támogatja a *CMIS*-t, például az *MS Sharepoint* is. A példa egyébként a document download konnektort használja. Nyomjuk meg a *Next* gombot, ami után a 8.9. ábra jön elő, aminek a tartalma szintén egyértelmű. A varázsló utolsó lépését a 8.10. ábra mutatja be. Itt már csak az output beállítása van hátra, ami a *document* nevű változóba (ez egy *org.bonitasoft.engine.bpm.document.DocumentValue* típusú objektum) tölti a dokumentumot és a konnektor elvégezte a feladatát.



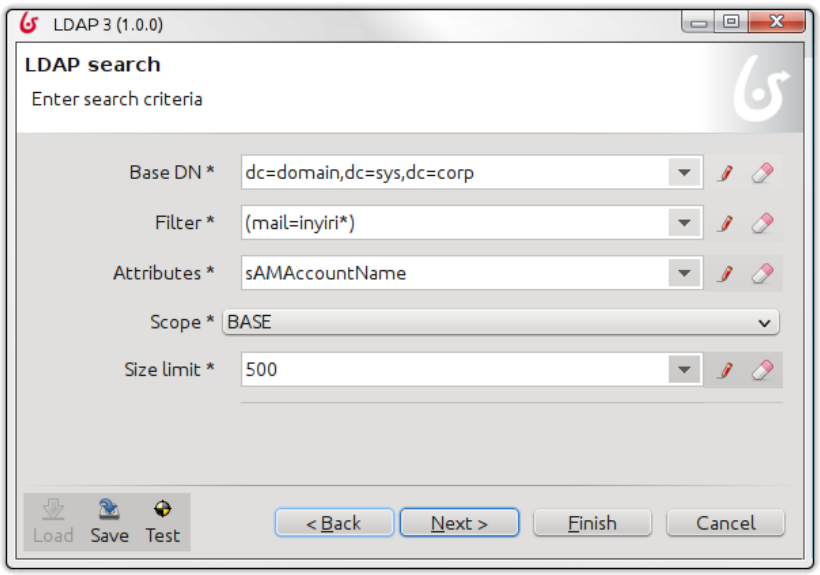

8.10. ábra. A CMIS konnektor konfigurálása - 3

## Az LDAP konnektor

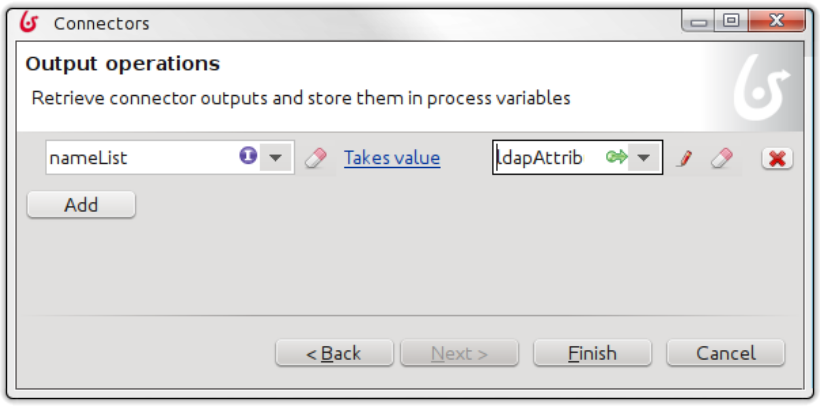



8.11. ábra. Az LDAP konnektor konfigurálása - 1

A Bonita LDAP konnektor a címtárak, például az Active Directory elérését teszi lehetővé. A 8.11. ábra ablaka a kapcsolódási paramétereket fogja össze. A *Next* hatására bejövő 8.12. ábra azt specifikálja, hogy mely LDAP mezőket szeretnénk lekérni.




8.12. ábra. Az LDAP konnektor konfigurálása - 2




8.13. ábra. Az LDAP konnektor konfigurálása - 3

A 8.13. ábra a varázsló utolsó ablaka, itt is a szokásos output mapping megadás történik. A *nameList* változó egy Java *List*, aminek az az oka, hogy az előző képernyőn több lekérendő mezőt is megadhatunk. A 8.14. ábra egy tömör összefoglaló azon tudnivalókról, amiket az LDAP keresés megadásakor kell felhasználnunk.

| Input | Description | Type |
|---|---|---|
| Base DN | the Distinguished Name at which to start search | string |
| Filter | specify a subset, e.g. (& (objectClass=person)(givenname=John)) | string in accordance with LDAP syntax |
| Attributes | define attributes to return in result entries using LDAP syntax | string of strings separated by "," |
| Scope | ◦ subtree: entire subtree starting as the base DN<br>◦ one level: entries immediately below the base DN<br>◦ base: search just the named entry | select |
| Size limit | maximum number of entries to return | number |
| Time limit (in seconds) | maximum time to allow search to run | number |
| Referral handling | ignore or follow referrals | select |
| Alias dereferencing | ◦ always: always dereference aliases<br>◦ searching: dereference aliases only after name resolution<br>◦ never: never dereference aliases<br>◦ finding: dereference aliases only during name resolution | select |

8.14. ábra. Az LDAP kereső kritériumok megadási lehetőségeinek összefoglalása

## A SAP JCo 2 konnektor

### A SAP JCo 2 áttekintése és telepítése

A SAP JCo egy olyan Java könyvtár, ami lehetővé teszi az SAP RFC függvények (és BAPI-k) kétféle használatát:

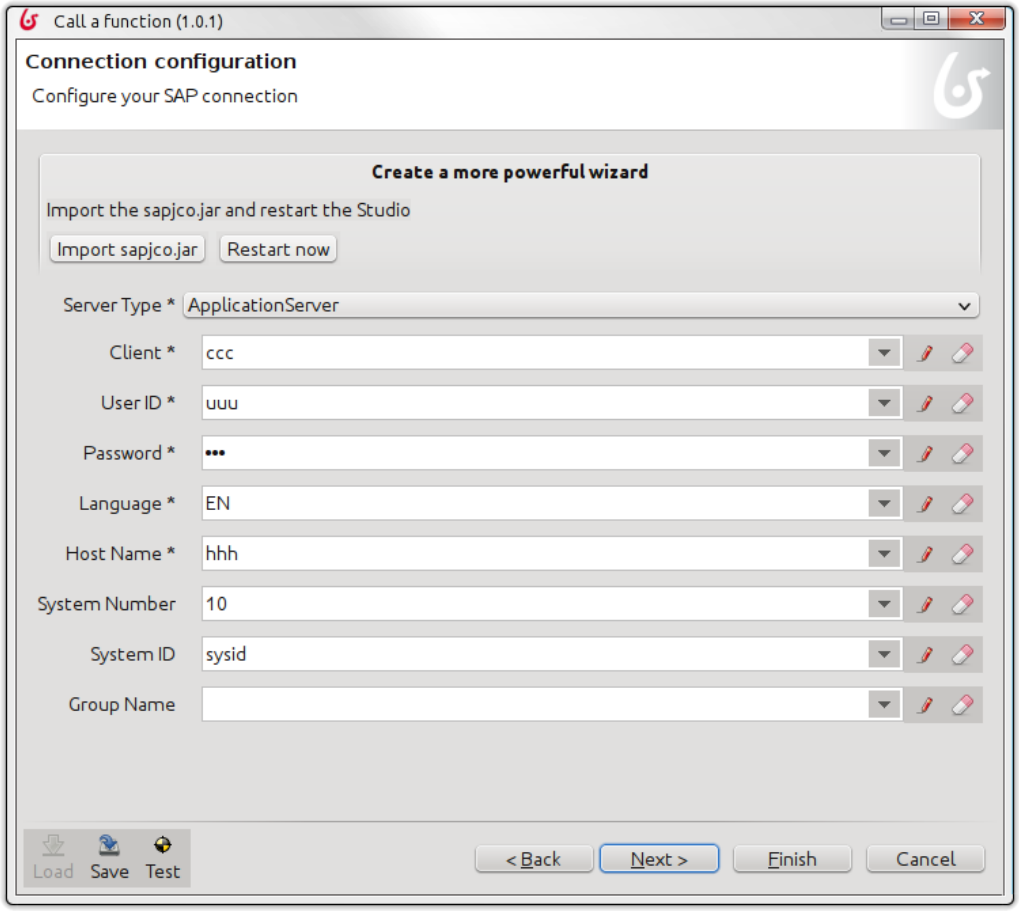
1. Amikor a Java hív egy SAP függvényt
2. Amikor az SAP hív egy Java metódust

Ez a könyvtár támogatott az SAP R/3 3.1H és magasabb verziók számára. Innen tölthető le a könyvtári csomag: `http://service.sap.com/connectors`. A csomag neve például egy 32 bites Linux esetén: *sapjco-linuxintel-2.1.9.tgz*. Háromféle hívási módszert támogat: szinkron, tranzakcionális és queued RFC. A telepítés lépései a következőek:

1. Egy erre a célra szolgáló könyvtárba csomagoljuk ki a letöltött csomagot és lépjünk is be a könyvtárba. Legyen ez a könyvtár a *{sapjco-install-path}*.
2. A csomag *so* (shared object) fájlt is tartalmaz, ezért erre a könyvtárra is állítsuk be az *LD_LIBRARY_PATH* környezeti változót.
3. A *{sapjco-install-path}/sapjco.jar* fájlt adjuk hozzá a *CLASSPATH* environment változóhoz.

A 8.15. ábrán látható az *Import sapjco.jar* gomb, amivel az említett fájlt be lehet importálni, aminek a célhelye a *bonita_home/endorsed* könyvtár lesz (ezt akár kézzel is elvégezhetjük így). Szeretnénk kiemelni, hogy a JCo 2 elterjedt, de az SAP 2013.03.31 óta már nem fejleszti, így lassan mindenkinek a JCo 3 verzióra kell majd átállnia.

### A SAP JCo 2 és JCo 3 konnektorok használata




8.15. ábra. SAP JCo 2 - Bapi hívás - 1

Egy SAP JCo2 konnektor példány konfigurálása a következő lépésekből áll:

1. Válasszuk ki azt a BPMN diagram elemet, amire SAP JCo konnektort szeretnénk konfigurálni és az ismert módon jussunk el a 8.1. ábráról már megtanult popup ablakhoz. Menjünk a *SAP JCo→Call a function JCo2* faághoz és az ismert módon jussunk el a 8.15. ábra ablakához.

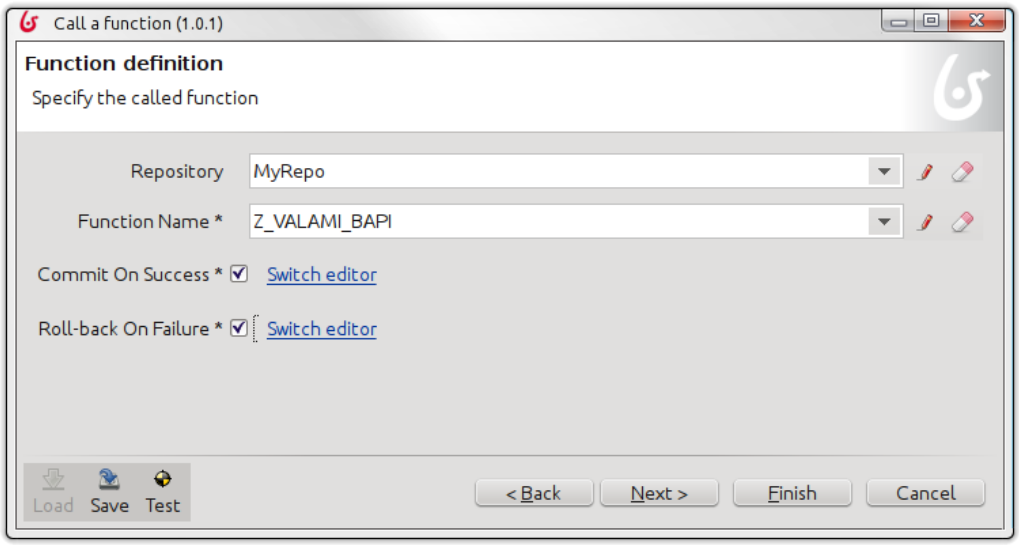
2. Töltsük ki az ablak formját, ami az SAP rendszerhez való kapcsolódás adatait hordozza.

3. A következő ablak (8.16. ábra) szolgál a meghívandó SAP RFC függvény nevének megadására.

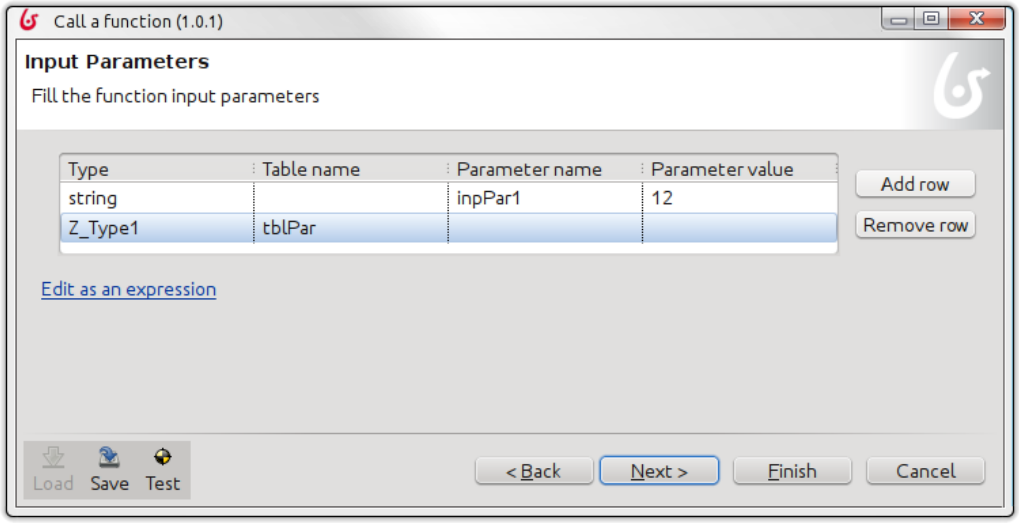
4. A 8.17. ábra az input függvény paraméterek megadását mutatja. Ez nem kötelező, mert nem mindegyik SAP függvény rendelkezik inputtal.

5. Egy *path* megadása következik, amit a SAP JCo igényel, mert a hívásról legyárt egy eredmény html fájlt.

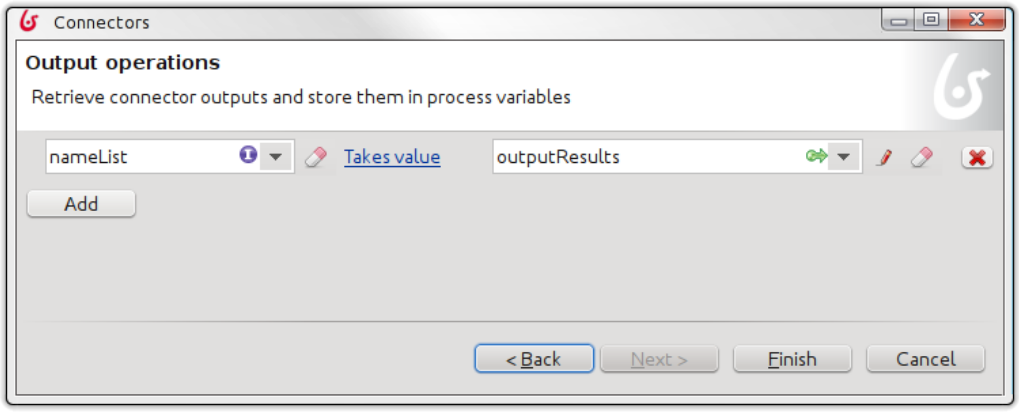
6. A 8.18. ábra a függvény output paramétereinek mappingjét mutatja. Ez egy *java.util.List*, mert értéklistát tartalmaz. Emiatt az elemeire így hivatkozhatunk: *outputResults.get(i)*.




8.16. ábra. SAP JCo 2 - Bapi hívás - 2




8.17. ábra. SAP JCo 2 - Bapi hívás - 3



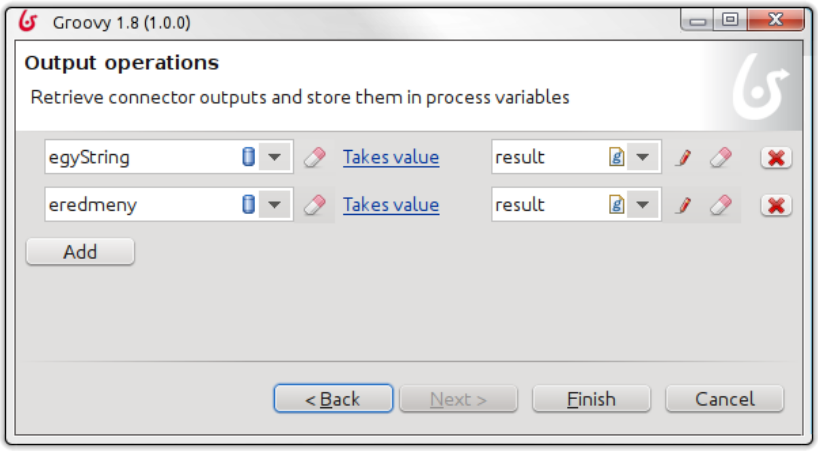

8.18. ábra. SAP JCo 2 - Bapi hívás - 4

A SAP JCo 3 konfigurálása és használata szinte teljesen megegyezik a 2. verzióval, így ezt nem részletezzük külön.

## A Script konnektor

Minden előzetes magyarázat nélkül tekintsünk egy példát! Legyen egy teszt workflow, aminek ez a 2 változója is van: *egyString* (Text) és *eredmeny* (Text). Kössünk egy humán taszkhoz egy Groovy scriptet, aminek *java.lang.Object* a statikus típusú visszatérési értéke, azonban a dinamikus típusa *java.util.List* lesz. A script konnektor *Script* mezőjénél megadva a Groovy script ez lesz:

```
List list = new ArrayList()
list.add("Első érték");
list.add("Második érték");
return list;
```



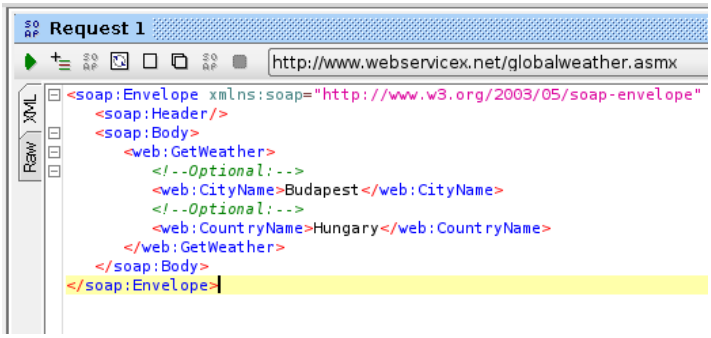

8.19. ábra. A Script típusú konnektor output mappingje

Mindkét jobb oldali *result* érték mögött egy-egy script van, ami így néz ki mindkét esetben, eltekintve attól, hogy a *get()* indexe az *eredmeny* változóhoz *1*, azaz *get(1):*

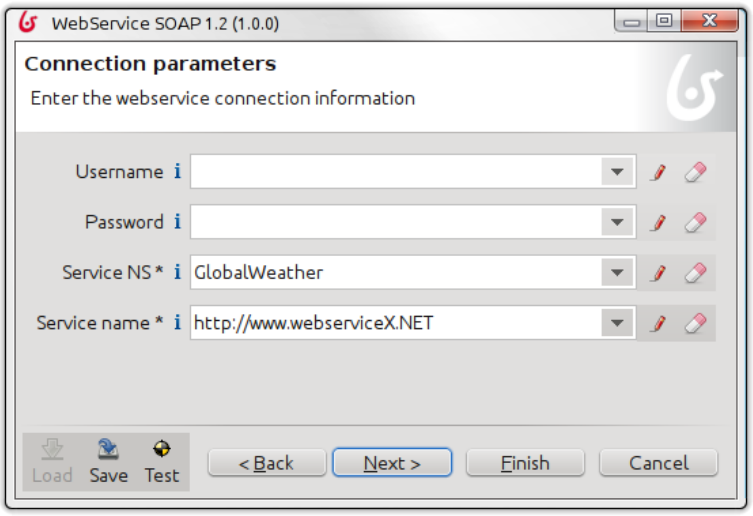
```
List list = result;
return (String)list.get(0)
```

## A Webservice konnektor

A Bonita 6. része egy Webservice konnektor, ami a SOAP 1.2 szabvány szerinti kliens szerint működik. Az Interneten található a következő websevice: `http://www.webservicex.net/globalweather.asmx?WSDL`. A következőekben megmutatjuk, hogy ezt az időjárás lekérdezést szolgáltatást miképpen tudjuk elérni a Bonita webservice konnektor segítségével. Első lépésként kipróbáltuk az ismert SOAP UI eszközzel (honlap: `http://www.soapui.org/`) és tökéletesen működik. A 8.20. ábra a generált SOAP 1.2 kérés XML-t mutatja. Figyeljünk arra, hogy a SOAP UI más kéréseket is felkínál, de a Bonita miatt mi a korszerű SOAP 1.2 verziót használjuk.



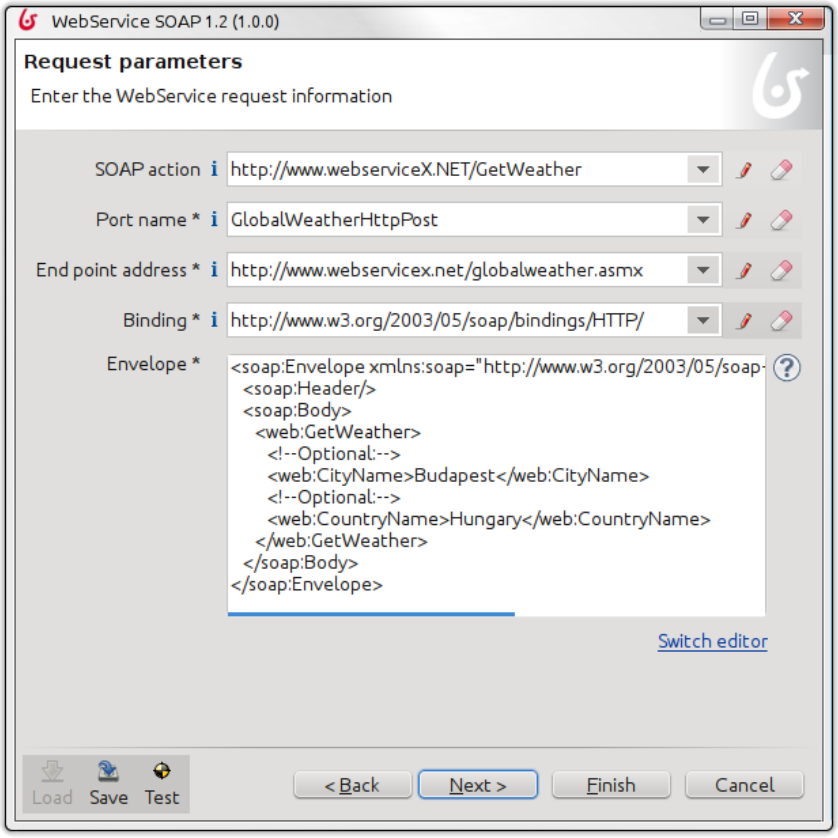

8.20. ábra. SOAP UI - A generált kérés XML




8.21. ábra. Webservice konnektor - 1

A konnektor varázsló elindítása után meg kell adnunk a webservice WSDL URL-t, majd az *Introspect* gomb megnyomása után a 8.21. ábra formja jelenik meg, előre kitöltve. Itt nincs

user/password, ezért ezeket nyugodtan hagyjuk üresen és nyomjuk meg a *Next* gombot, hogy jöjjön a varázsló következő ablaka (8.22. ábra). Az *Envelope* mezőt a SOAP UI által generált tartalommal töltöttük ki, változtatás nélkül. Amennyiben bármi más adat is hiányozna, a SOAP UI-ból azokat is ki tudjuk tölteni.




8.22. ábra. Webservice konnektor - 2

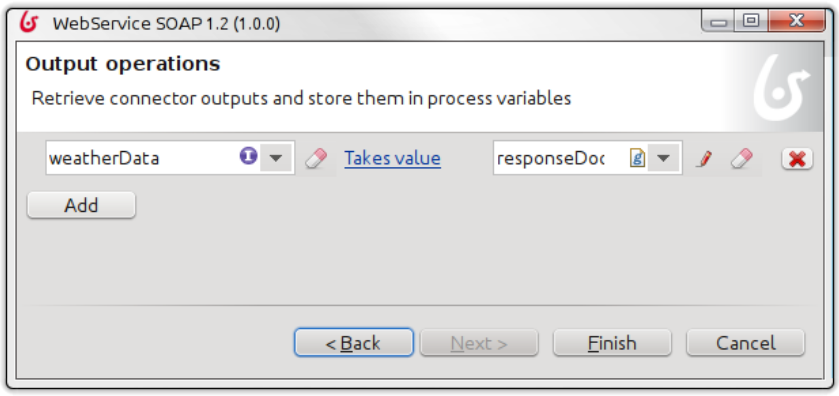
A következő ablakban HTTP header-eket adhatnánk meg, amennyiben az szükséges. Jelen esetben ilyen nem kell, ezért egyszerűen nyomjunk *Next* gombot, ahol a varázsló megkérdezi tőlünk, hogy milyen válaszokat adjon. Ezek a checkboxok vannak:

- *Returns envelope*: ezt nem kértük
- *Returns body*: ezt a választ kérjük
- *Print debug info*: ezt nem kértük

A következő ablakot a 8.23. ábra mutatja. Előzetesen megjegyezzük, mert eddig nem említettük, hogy ehhez a példához 3 workflow változót vettünk fel:

- *weatherData*: *java.util.Map*. Ez fogadja a webservice választ (8.23. ábra)
- *orszag*: *String*. Feladata, hogy paraméterezhető legyen ezzel a webservice.

- *varos*: *String*. Feladata, hogy paraméterezhető legyen ezzel a webservice.



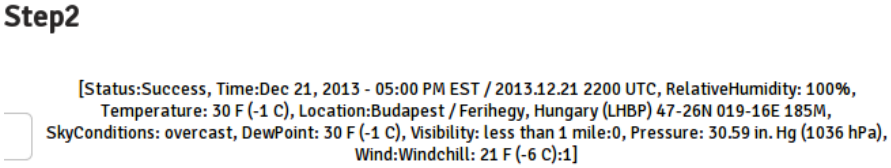

8.23. ábra. Webservice konnektor - 3

A 8.23. ábrán látható *responseDocumentEnvelope* egy script, aminek a visszatérési értéke (*java.util.Map*) kerül értkadásra a *weatherData* számára. Maga a script érthető, ezért magyarázatok nélkül itt van a teljes kód, amit a kifejezés szerkesztőbe írtunk be:

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;

// Clean response xml document
responseDocumentBody.normalizeDocument();
// Get result node
NodeList resultList = responseDocumentBody.getElementsByTagName("GetWeatherResult");
Element resultElement = (Element) resultList.item(0);
String weatherDataAsXML = resultElement.getTextContent();

// Check for empty result
if ("Data Not Found".equalsIgnoreCase(weatherDataAsXML))
        return null;

// Parse embedded XML of result
DocumentBuilder documentBuilder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
InputSource inputSource = new InputSource();
inputSource.setCharacterStream(new StringReader(weatherDataAsXML));
Document weatherDataDocument = documentBuilder.parse(inputSource);
Node weatherNode = weatherDataDocument.getDocumentElement();

// Save weather data
Map<String,String> data = new HashMap<String,String>();
NodeList childNodes = weatherNode.getChildNodes();
for (int i=0; i<childNodes.getLength(); i++)
{
                Node node = childNodes.item(i);
                if (node.getNodeType() == Node.ELEMENT_NODE)
                {
                                String key = node.getNodeName();
```

```
                                String value = node.getTextContent();
                                data.put(key, value);
                }
}
return data;
```

A futási eredményt mutatja a 8.24. ábra úgy, hogy a *weatherData.toString()* metódus értékét egy *Message* widget-be tettük be.




8.24. ábra. Webservice konnektor - 4

Persze ezzel a kóddal az egyes adatelemeket szépen elérhetnénk már, de most nem egy csinos form elkészítése volt a célunk:

```
import java.util.Map.Entry;

List<List<String>> table = new ArrayList<List<String>>();
Set<Entry<String,String>> weatherDataEntries = weatherData.entrySet();
for (Entry<String,String> entry : weatherDataEntries)
{
        List<String> row = new ArrayList<String>();
        row.add(entry.getKey());
        row.add(entry.getValue());
        table.add(row);
}
return table;
```

Végül szeretnénk paraméterezetté tenni a webservice konnektor használatát, ezért a SOAP 1.2 kérés XML-t így alakítottuk át:

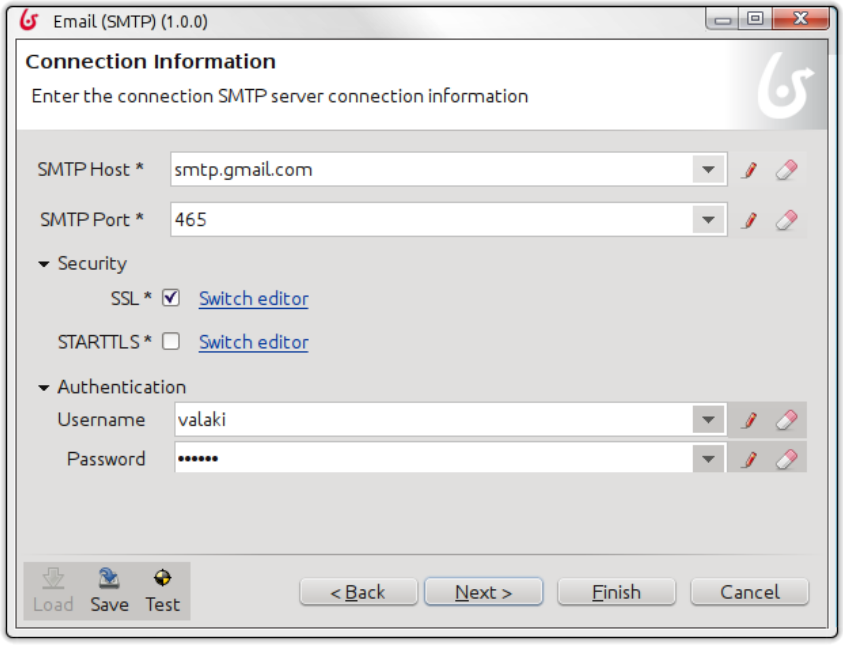
```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope" xmlns:web="http://www.➥
    webserviceX.NET">
   <soap:Header/>
   <soap:Body>
      <web:GetWeather>
         <!--Optional:-->
         <web:CityName>${varos}</web:CityName>
         <!--Optional:-->
         <web:CountryName>${orszag}</web:CountryName>
      </web:GetWeather>
   </soap:Body>
</soap:Envelope>
```

A változtatás lényege az, hogy kihasználva a Groovy script makróhelyettesítés lehetőségét a *Budapest→${varos}* és *Hungary→${orszag}* változtatásokat tettük meg.

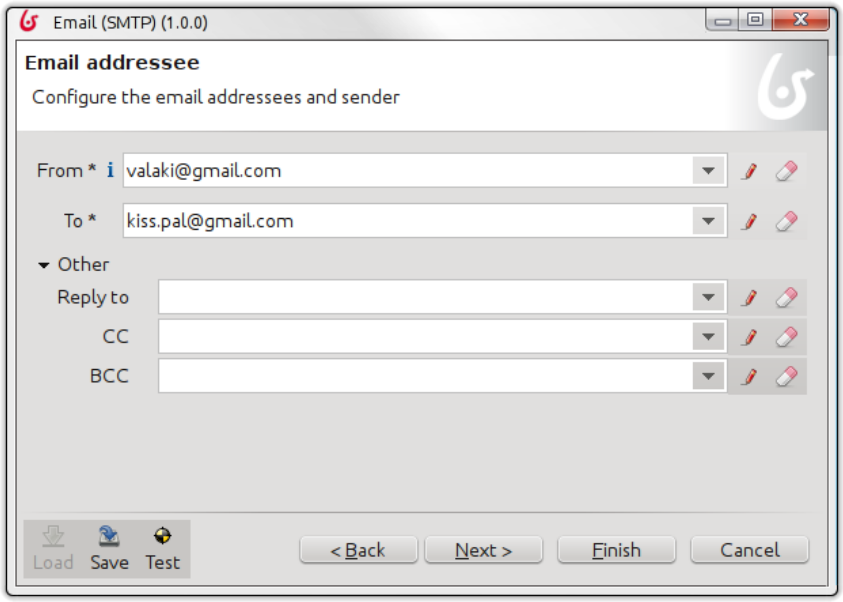
## Az E-Mail konnektor

Az E-Mail konnektor az elvárt, szokásos beállításokat tartalmazza. A konnektor varázsló először bekéri a levelező (SMTP) szerver kapcsolódási adatait, ahogy a 8.25. ábra is mutatja. A példában

a *gmail* szerverét használjuk.



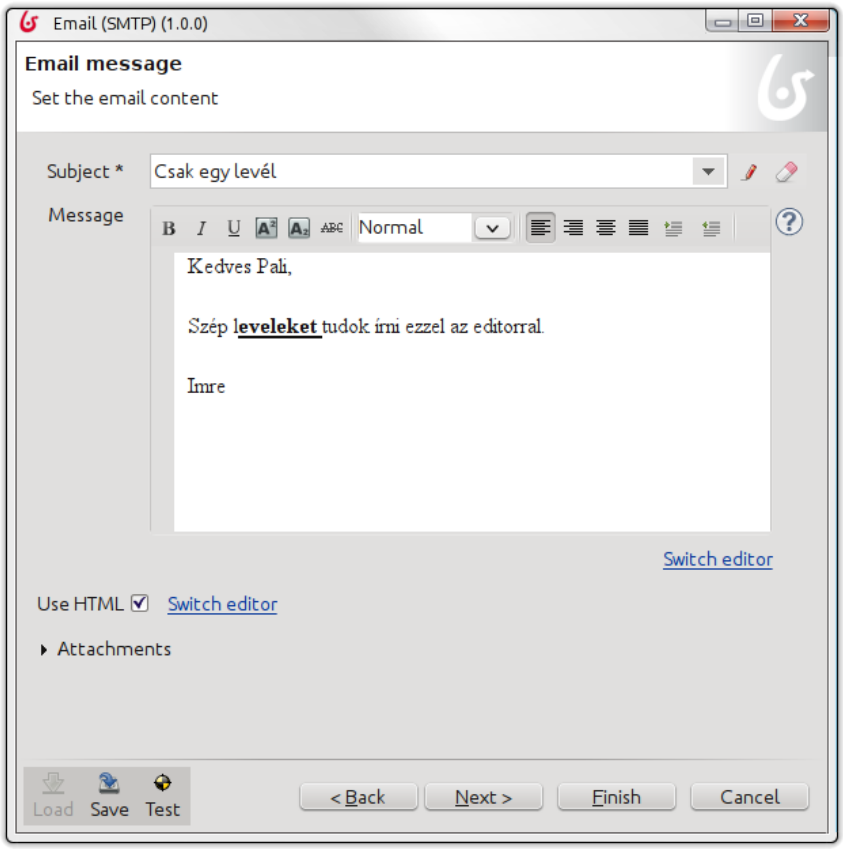

8.25. ábra. Az E-Mail konnektor használata - 1




8.26. ábra. Az E-Mail konnektor használata - 2

A következő képernyő (8.26. ábra) szolgál a feladó és a címzettek megadására. Az olyan SMTP szerverek esetén, ahol a hitelesítés kötelező, lényeges a *From* és az előzőleg megadott username

értékek összhangja. A levél üzenet része alakítható ki a varázsló utolsó előtti képernyőjén (8.27. ábra). Itt most konkrét értékeket (szövegeket) adtunk meg a *Subject* és *Message* résznél is, de természetesen itt is használható a Groovy *${változó}* makróhelyettesítés, azaz levél template-ek készíthetőek. Látható az is, hogy tetszőleges számú levélmellékletet tehetünk az üzenethez.




8.27. ábra. Az E-Mail konnektor használata - 3

A varázsló utolsó képernyőjén a levélszöveg Character Set beállítását specifikálhatjuk, javasoljuk, hogy mindig maradjunk az UTF-8 mellett. Ugyanezen a formon még levél message fejléceket is adhatunk meg a *Headers* mezőnél. Ezt sem használtuk most a példában.

## A Jasper 5 konnektor - Riport készítés

Az Informatikai Navigátor 8. száma teljes egészében a Jasper Reports 5 riportkészítő eszközről szól, így akit minden részlet érdekel, ott megtalálhatja. A Jasper 5 konnektor varázsló első lépése az, hogy meg kell adnunk az adatbázishoz való kapcsolódás adatait: driver, connection URL, a user neve és jelszava. A következő lépés a riport *JRXML*, az esetleges riport paraméterek és legyártandó céldokumentum formátumának megadása (PDF, HTML, XML). Az utolsó képernyőn a legyártott és már ismert *DocumentValue* objektum hozzárendelése a workflow ugyanilyen típusú belső változójához.

# 9. Saját Bonita konnektor készítése

Az előző fejezetben bemutattunk néhány általunk fontosnak gondolt gyári Bonita konnektort. Van persze ezenfelül még számos 3rd party konnektor is, ugyanis a Bonita fejlesztői gondoskodtak arról, hogy jól definiált módon saját konnektorokat is készíthessünk. Ez utóbbi lehetőséget mutatja be ez a fejezet.

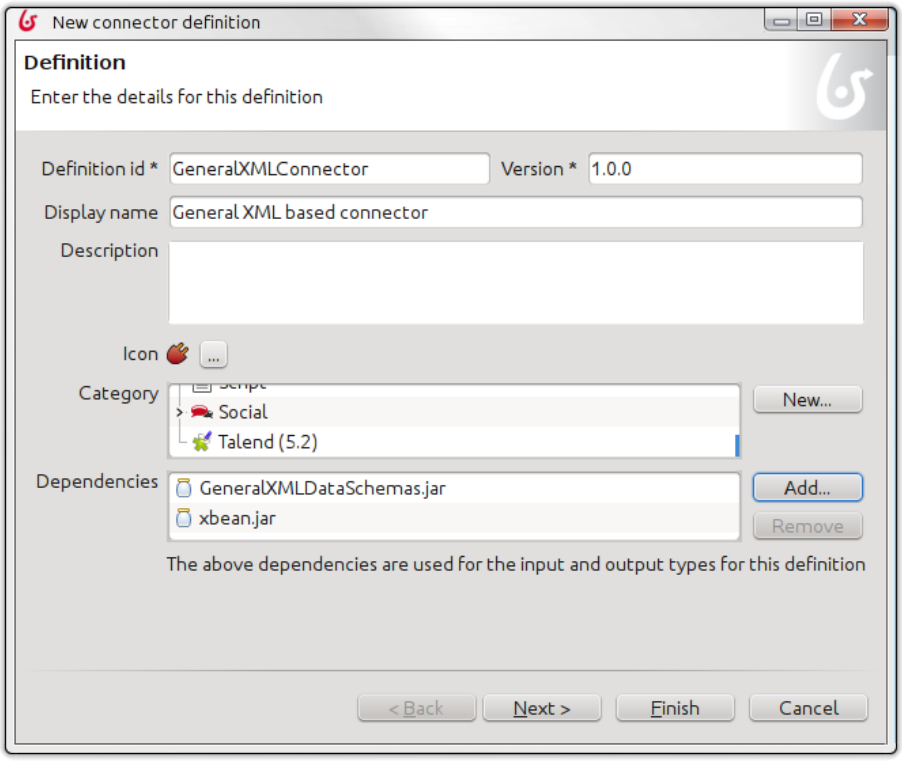
A saját konnektor készítését egy olyan példán keresztül mutatjuk be, ami meglehetősen általános, mert az inputot és az outputot is egy XML adatszerkezet adja. Ez azért fontos példa, mert XML-lel tetszőleges összetettségű adatszerkezetet tudunk reprezentálni, azaz a konnektor paraméterezése is meglehetősen rugalmas lehet. Az input és output XML az egyszerűség kedvéért legyen most ugyanaz és ez lesz az XML sémája:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" targetNamespace="org.cs.jaxb.test" xmlns:tns="org.cs.jaxb.test" ➥
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="classSecond" type="tns:classSecond"/>

  <xs:complexType name="classSecond">
    <xs:sequence>
      <xs:element name="description" type="xs:string" minOccurs="0"/>
      <xs:element name="title" type="xs:string" minOccurs="0"/>
      <xs:element name="pages" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

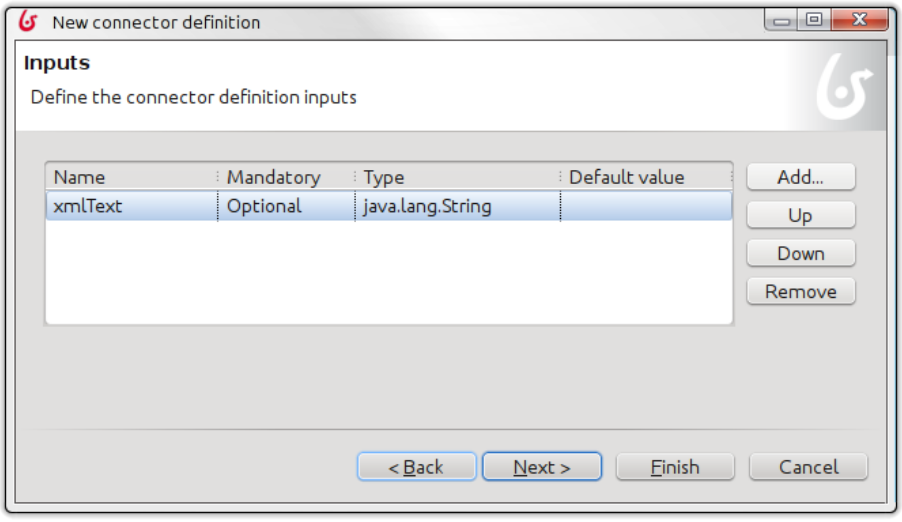
Az XML előállításhoz és feldolgozáshoz az Apache XMLBeans technológiát fogjuk használni, mert ezzel úgy tudjuk az XML-t kezelni, mintha Java objektumok lennének (honlap: `http://xmlbeans.apache.org/`). Ebben a csomagban van egy séma fordító (*scomp*), ami egy tetszőleges XSD-ből egy Java könyvtárat készít- Esetünkben erre ezt a parancsot használtuk:

```
xmlbeans-2.5.0/bin/scomp -out GeneralXMLDataSchemas.jar test.xsd
```

A Bonita 6. verzióban a konnektorok készítésénél lehetőségünk van annak az interface-ét (hívási felület) létrehozni, amit a *Development→Connectors→New definition...* (vagy *Edit definition...*) menüpontból kezdeményezhetünk és a 9.1. ábra ablakát kell első lépésben kitöltenünk. A konnektorral ezután majd a *General XML based connector* néven fogunk találkozni akkor, amikor valahol példányosítani akarjuk. Vegyük észre, hogy megadtuk az XMLBeans használatával összefüggő 2 *jar* fájlt is, mint dependencia. A *Next* megnyomása után a 9.2. ábra jön be. Itt kell felsorolnunk a konnektor összes input paraméterét. Ezeket az *Add...* gombbal vehetjük fel. Most érdekes a helyzet, mert az XSD 3 mezőt is definiál. de mi egy egész XML stringet adunk át, emiatt csak 1 darab paraméterünk lesz, amit *xmlText* nevűre kereszteltünk. Ismét *Next* és máris a 9.3. ábra ablaka jelenik meg a konnektor interface létrehozó varázsló következő lépéseként. Némi gondolkodás után könnyen rá lehet jönni, hogy itt most az előző lépésben megadott input paraméterekhez készítünk egy olyan varázsló oldalt és az azokon működő widget-eket, amit majd a konnektor példány létrehozásakor fogunk használni. Ez igen jó dolog, mert ezek szerint nem csak a konnektor Java interface, hanem annak példányosítást készítő felülete is elkészül ezzel a lépéssel.



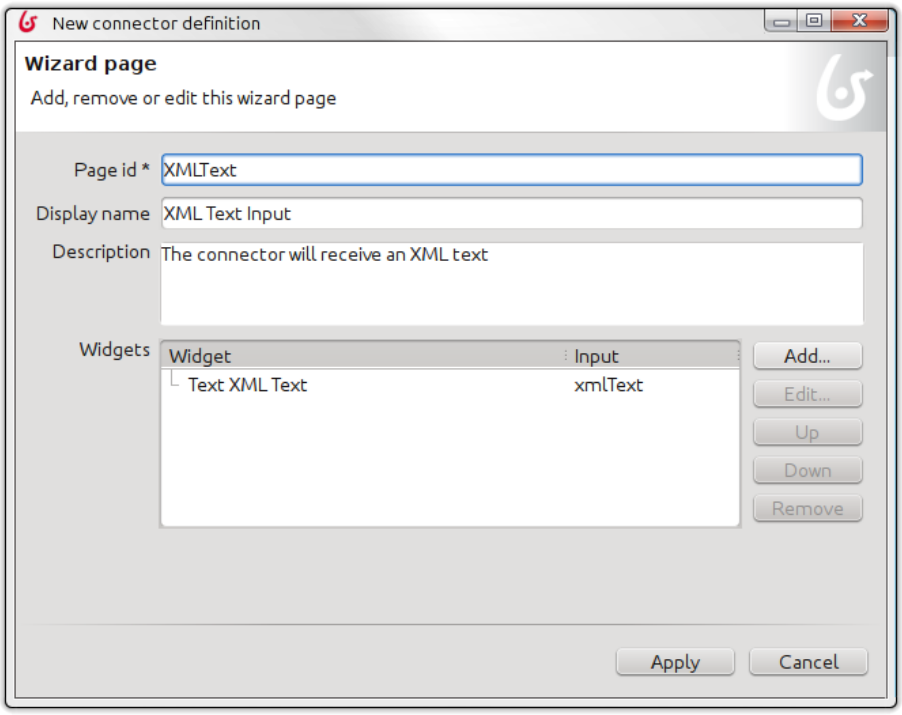

9.1. ábra. A GeneralXMLConnector interface specifikációja - 1



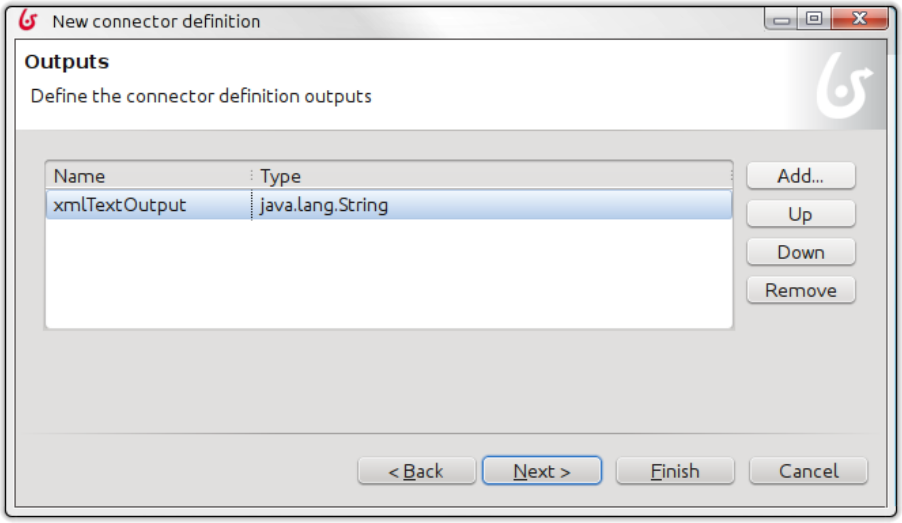

9.2. ábra. A GeneralXMLConnector interface specifikációja - 2

A 9.3. ábra azt mutatja éppen, ahogy az 1 darab *xmlText* nevű input paraméterünknek csináltunk 1 darab Widget-et, ahol egy egész XML string lesz megadható majd. *Next*, majd bejön az interface varázsló utolsó ablaka (9.4. ábra), ahol a konnektor output paramétereit kell felsorolnunk.

Tekintettel arra, hogy az output is XML lesz, ezért ebből is csak 1 darabot használunk a példában és az *xmlTextOutput* nevet adtuk neki.



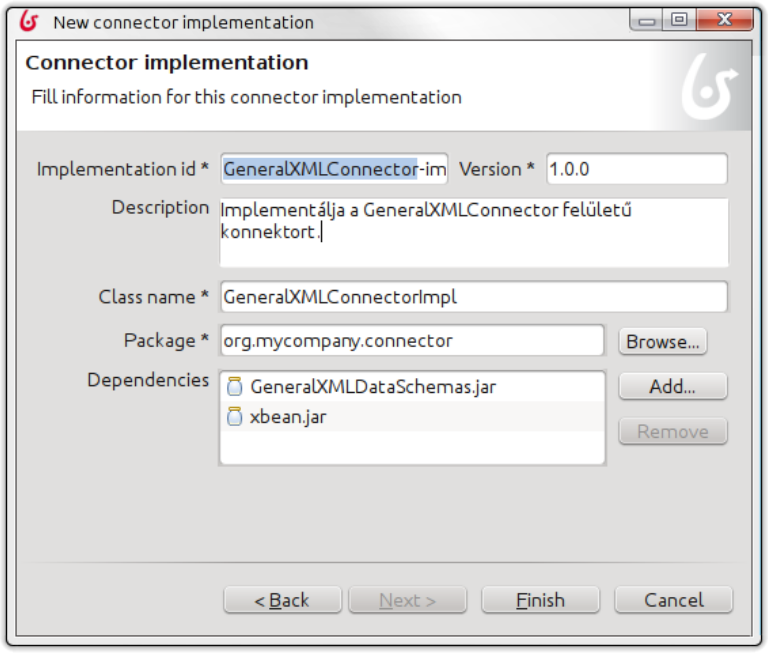

9.3. ábra. A GeneralXMLConnector interface specifikációja - 3




9.4. ábra. A GeneralXMLConnector interface specifikációja - 4

Elkészültünk a konnektor interface specifikációjával, most következzen annak az implementációja. Ezt a *Development→Connectors→New implementation...* (vagy *Edit implementation...*)

menüpontnál kezdeményezhetjük, ahol a Bonita Stúdió megkérdezi, hogy melyik konnektor specifikációt szeretnénk implementálni. Természetesen válasszuk ki a most specifikáltat, majd töltsük ki a 9.5. ábra ablakát úgy, ahogy azt látjuk.




9.5. ábra. A GeneralXMLConnector interface implementációja

A *Finish* gomb hatására a Stúdió megnyit egy kódszerkesztő ablakot ahol az implementációt végül a 9-1. Programlista alapján készítettük el.

9-1. Programlista: A GeneralXMLConnector implementációja

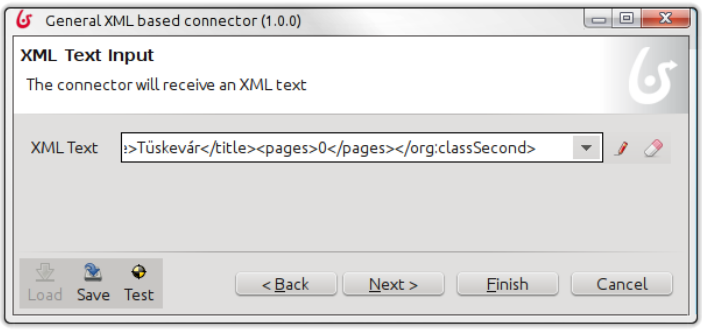
```
1  package org.mycompany.connector;
2
3  import org.apache.xmlbeans.XmlException;
4  import org.apache.xmlbeans.XmlObject;
5  import org.bonitasoft.engine.connector.ConnectorException;
6
7  import test.jaxb.cs.org.ClassSecondDocument;
8
9  /**
10  *The connector execution will follow the steps
11  * 1 - setInputParameters() --> the connector receives input parameters values
12  * 2 - validateInputParameters() --> the connector can validate input parameters values
13  * 3 - connect() --> the connector can establish a connection to a remote server (if necessary)
14  * 4 - executeBusinessLogic() --> execute the connector
15  * 5 - getOutputParameters() --> output are retrieved from connector
16  * 6 - disconnect() --> the connector can close connection to remote server (if any)
17  */
18 public class GeneralXMLConnectorImpl extends AbstractGeneralXMLConnectorImpl {
19
20         @Override
21         protected void executeBusinessLogic() throws ConnectorException{
22
23                 ClassSecondDocument xmlObj = null;
```

```
24
25                  try {
26                          xmlObj = ClassSecondDocument.Factory.parse( getXmlText() );
27
28                          String desc = xmlObj.getClassSecond().getDescription();
29                          String title = xmlObj.getClassSecond().getTitle();
30                          int pages = 150;
31                          xmlObj.getClassSecond().setPages( pages );
32
33                          setXmlTextOutput( xmlObj.toString()  );
34
35                  } catch (XmlException e) {
36                          setXmlTextOutput( getXmlText()  );
37                  }
38          }
39
40          @Override
41          public void connect() throws ConnectorException{
42                  //[Optional] Open a connection to remote server
43
44          }
45
46          @Override
47          public void disconnect() throws ConnectorException{
48                  //[Optional] Close connection to remote server
49
50          }
51  }
```

Érdemes pár szóban összefoglalni, hogy mit is csinál a fenti kód. A generált programvázban csak a 21-38 sorok közötti részt változtattuk, a többi úgy maradt, ahogy generálódott. Ez a *executeBusinessLogic()* metódus, ami a konnektor tényleges feladatának a végrehajtása. Átírhatnánk a többi metódust is, ha valamilyen követelményünk lenne a konnektor futtatásának előkészítésével vagy befejezésével szemben, de itt most olyan nincs. A 23-31 sorok között az XML objektumot építjük fel abból az input XML stringből, amit a konnektor átvett és a *getXmlText()* metódussal tesz számunkra elérhetővé. Az üzleti logika csak a példa kedvéért nagyon egyszerű, mindössze beállítjuk a *pages* értéket 150-re (30-31 sorok). Az implementáció utolsó fontos része, hogy be kell állítanunk a kiszámított output értékeket, amiből tudjuk, hogy most csak 1 darab van. Ezt végzi a 33. sorban látható *setXmlTextOutput()* metódus, ami az XML objektumunknak a string reprezentációját adja vissza, hiszen a konnektor specifikációnál ezt adtuk meg. Amennyiben valamilyen kivétel keletkezik, úgy a mostani konnektor implementáció egyszerűen csak visszaadja a kapott XML-t. Elkészült a konnektorunk, teszteljük le ezzel az input stringgel:

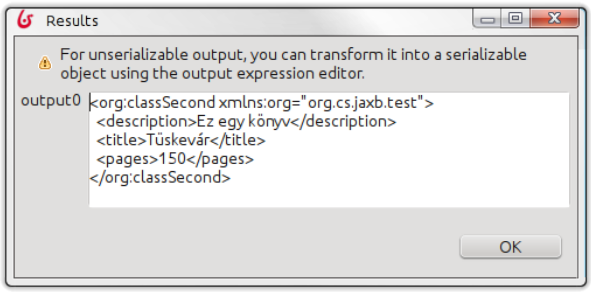
```
<org:classSecond xmlns:org="org.cs.jaxb.test"><description>Ez egy könyv</description><title>➥
    Tüskevár</title><pages>0</pages></org:classSecond>
```

Menjünk egy stephez és adjuk hozzá a mi konnektorunk egy példányát. Eközben a 9.6. ábra ablaka is bejön és láthatjuk, hogy a korábbiakban elkészített varázslónk is szépen működik. Az ábrán mutatott módon szúrjuk be a fenti teszt XML-t és nyomjuk meg a *Test* gombot!




9.6. ábra. Egy GeneralXMLConnector példány konfiguráció - Input mapping

A teszt eredményét a 9.7. ábra mutatja és az is látható, hogy jól működik a konnektorunk.




9.7. ábra. Egy GeneralXMLConnector példány konfiguráció - Test gomb megnyomva...

Egy valós workflow esetén ez a script állítaná elő az XML stringet a workflow változóiból:

```
import org.apache.xmlbeans.XmlException;
import org.apache.xmlbeans.XmlObject;
import test.jaxb.cs.org.ClassSecondDocument;

ClassSecondDocument xmlObj = ClassSecondDocument.Factory.newInstance();
xmlObj.addNewClassSecond();
xmlObj.getClassSecond().setDescription( eredmeny );
xmlObj.getClassSecond().setTitle( egyString );
xmlObj.getClassSecond().setPages(0);

xmlObj.toString();
```

És ez hívná le a válasz eredményt például a *description* mezőre, mint Output operation Groovy script:

```
import org.apache.xmlbeans.XmlException;
import org.apache.xmlbeans.XmlObject;
import org.bonitasoft.engine.connector.ConnectorException;

import test.jaxb.cs.org.ClassSecondDocument;

ClassSecondDocument xmlObj = null;
xmlObj = ClassSecondDocument.Factory.parse( xmlTextOutput );

return xmlObj.getClassSecond().getDescription();
```

# 10. Bonita 6. Actor Filter készítés

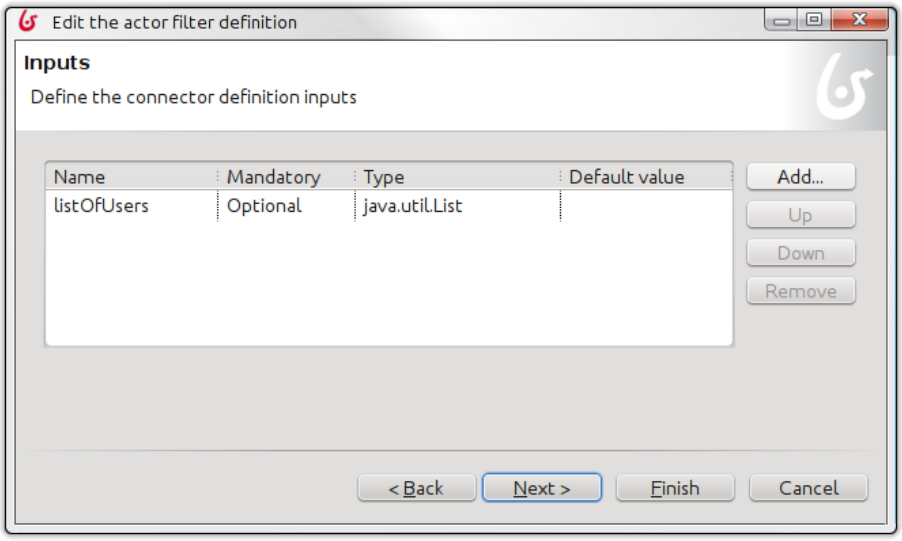
Már megtanultuk, hogy egy BPMN design humán taszkjaihoz Actor-okat rendelünk, amiket futási időben felhasználók halmazára kell tudnunk leképezni. Ennek volt az eszköze az *Actor Mapping* vagy a Portálon használható *Entity Mapping*. Néha olyan megoldásra van szükségünk, amiket ezekkel az eszközökkel nem tudunk elvégezni, ilyenkor merül fel, hogy mi is készítsünk egy új *Actor Filter*t.

A 2. fejezetben már bemutattuk, hogyan készíthetünk egy saját készítésű *Actor Filter*t. Itt most egy új példát mutatunk rá és le is teszteljük a helyes működését.

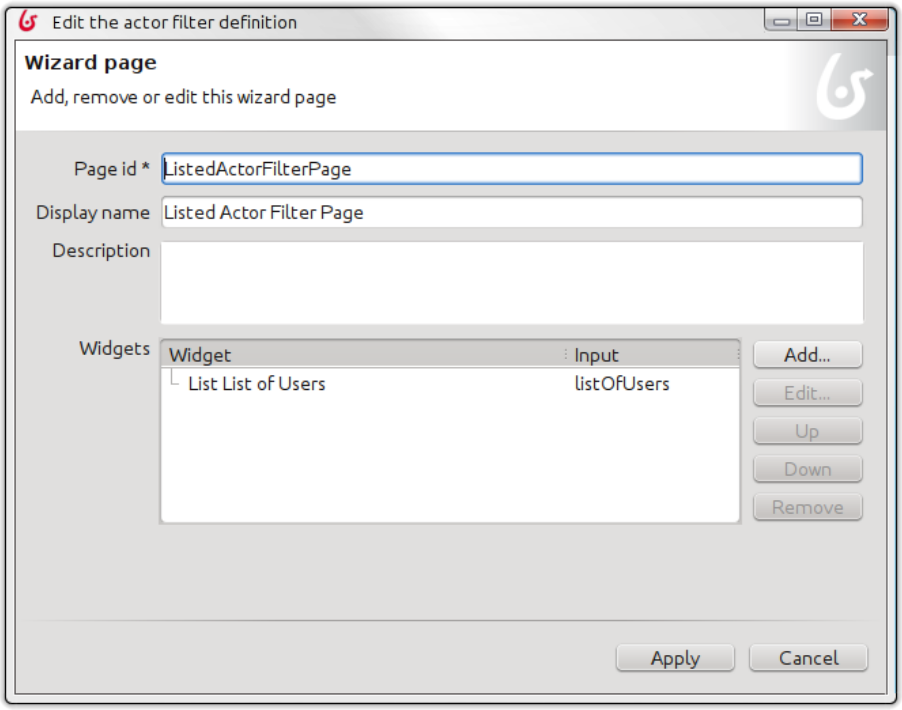
## A feladat megfogalmazása

A Bonita Stúdió tesztkörnyezete szolgáltat egy user adatbázist (a neve: *ACME*), ezt fogjuk használni. A feladat az, hogy a munkafolyamat indítója és minden taszk végrehajtó dinamikusan megadhassa, hogy a következő step kikhez kerülhet. Emiatt minden formra kerüljön fel egy felhasználónév lista megadási lehetőség, legyen innen kivéve a következő feladat lehetséges végrehajtóinak a halmaza.

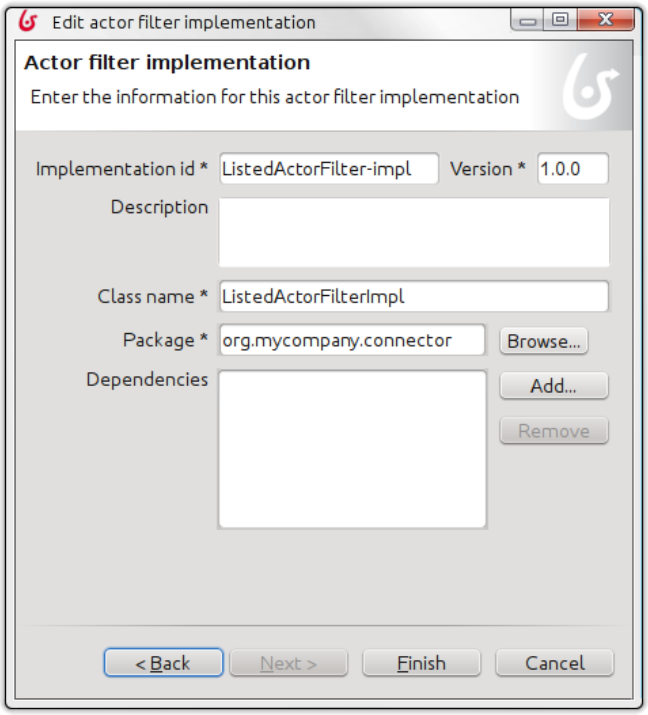
## A megvalósítás




10.1. ábra. A *ListedActorFilter* inputja egy Java *List*




10.2. ábra. A filter példány készítő varázsló legyen ilyen




10.3. ábra. A *ListedActorFilter* implementálásának elkezdése

A megoldáshoz egy Actor Filtert készítünk, aminek kegyen ez a neve: *ListedActorFilter*. Hasonlóan a konnektorokhoz, az aktor filterek is rendelkeznek interface specifikációval és annak megvalósításával. A működésük és készítésük logikája szinte teljesen megegyezik a konnektorokéval. A *Development→Actor filters→New definition...* (vagy *Edit definition...*) menüpontból kezdeményezhetünk egy új filter specifikáció elkészítését, ami után egy varázsló indul el. A 10.1. ábra azt mutatja, amikor az Acror Filterre megmondjuk, hogy milyen input paraméterek szerint kell majd dolgoznia. Esetünkben a képernyőn megadott userek listája lesz ez, ezért a formon a l*istOfUsers* input paramétert is *List* típusúnak adtuk meg. A 10.2. ábra a következő képernyő, szerkezete teljesen megegyezik a konnektoroknál is látott wizard készítő képernyőével. Itt is ezt csináljuk éppen, azaz megadjuk, hogy a l*istOfUsers* input paramétert, hogyan fogjuk majd bekérni akkor, amikor az Actor Filter éppen példányosítjuk valahol. Az interface felület kialakítása után a 10.3. ábra már az Actor Filter implementációs vázat elkészítő varázslót mutatja. Ide a *Development→Actor filters→New implementation...* (vagy *Edit implementation...*) menüpontból jutottunk el. A megvalósítás kódjának a váza legenerálásra kerül, azt átalakítva jutottunk el a filterünk implementációjához, amit a 10-1. Programlista mutat.

10-1. Programlista: A ListedActorFilter implementációja

```
1  package org.mycompany.connector;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  import org.bonitasoft.engine.api.UserAPI;
6  import org.bonitasoft.engine.connector.ConnectorValidationException;
7  import org.bonitasoft.engine.filter.UserFilterException;
8  import org.bonitasoft.engine.identity.UserNotFoundException;
9  import com.bonitasoft.engine.api.IdentityAPI;
10
11 /**
12  *The actor filter execution will follow the steps
13  * 1 - setInputParameters() --> the actor filter receives input parameters values
14  * 2 - validateInputParameters() --> the actor filter can validate input parameters values
15  * 3 - filter(final String actorName) --> execute the user filter
16  * 4 - shouldAutoAssignTaskIfSingleResult() --> auto-assign the task if filter returns a single result
17  */
18 public class ListedActorFilterImpl extends AbstractListedActorFilterImpl {
19
20         @Override
21         public void validateInputParameters() throws ConnectorValidationException {
22                 //TODO validate input parameters here
23         }
24
25         @Override
26         public List<Long> filter(final String actorName) throws UserFilterException {
27
28                 org.bonitasoft.engine.api.IdentityAPI iapi = getAPIAccessor().getIdentityAPI();
29                 List<Long> listOfUsers = new ArrayList<Long>();
30                 List users = getListOfUsers();
31
32                 for ( Object o : users )
33                 {
34                         String userName = (String)o;
35                         org.bonitasoft.engine.identity.User user;
36                         try {
37                                 user = iapi.getUserByUserName(userName);
38                                 listOfUsers.add( user.getId() );
39                         } catch (UserNotFoundException e) {
40                                 ;
41                         }
42                 }
43                 return listOfUsers;
44         }
45
46         @Override
47         public boolean shouldAutoAssignTaskIfSingleResult() {
48                 // If this method returns true, the step will be assigned to
49                 //the user if there is only one result returned by the filter method
50                 return super.shouldAutoAssignTaskIfSingleResult();
51         }
52 }
```

A legenerált váznak csak a *filter()* metódusát (26-44 sorok) írtuk át, a forráskód többi részéhez nem nyúltunk. A 28. sorban lekértünk egy hivatkozást az identity API-ra, amit az *iapi* változóban tároltunk el. A 29. sorban létrehoztuk azt az üres listát, ami majd a legenerálandó user halmazt (a user-ek *Long* típusú azonosítóit) fogja tartalmazni és a *filter()* metódus is ezt fogja visszatérési értékként szolgáltatni. A 30. sorban átvesszük az a listát, ami a filter input adata. Ezt a *users* kollekcióban tároljuk. A 32-42 sorok közötti *for* ciklus végigmegy a *users* lista elemein, amik user nevek lesznek. A *getUserByUserName()* metódus képes visszaadni egy *User* objektumot, amennyiben annak felhasználói neve a paraméter. A 38. sorban már egyszerű dolgunk van, mindössze le kell kérni ettől az objektumtól a user ID-t és hozzácsatolni a *listOfUsers* listához. A ciklusból kilépve már nem maradt más dolgunk csak visszaadni a legyártott *List<Long>* listát.

## Tesztelés

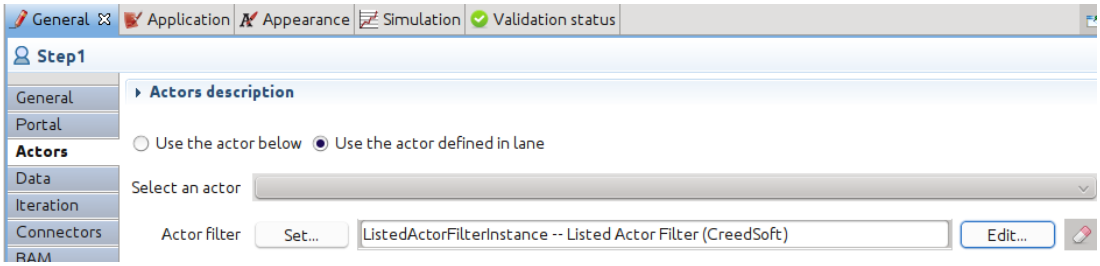
Befejezésül próbáljuk ki az új Actor Filter-ünket! A munkafolyamat most 2 humán step-ből fog állni. Ezek lesznek a process belső változói:

- *taskExecutors* (*java.util.List*): Ez tartalmazza majd a képernyőn megadott user-ek listáját.
- *info* (Text): Egy többsoros szöveg értékét tárolja el, célja csak az, hogy a taszkok rendelkezzenek valami apró adattal is.
- *decision* (Boolean): Egy döntés *true/false* értékét tárolja el.

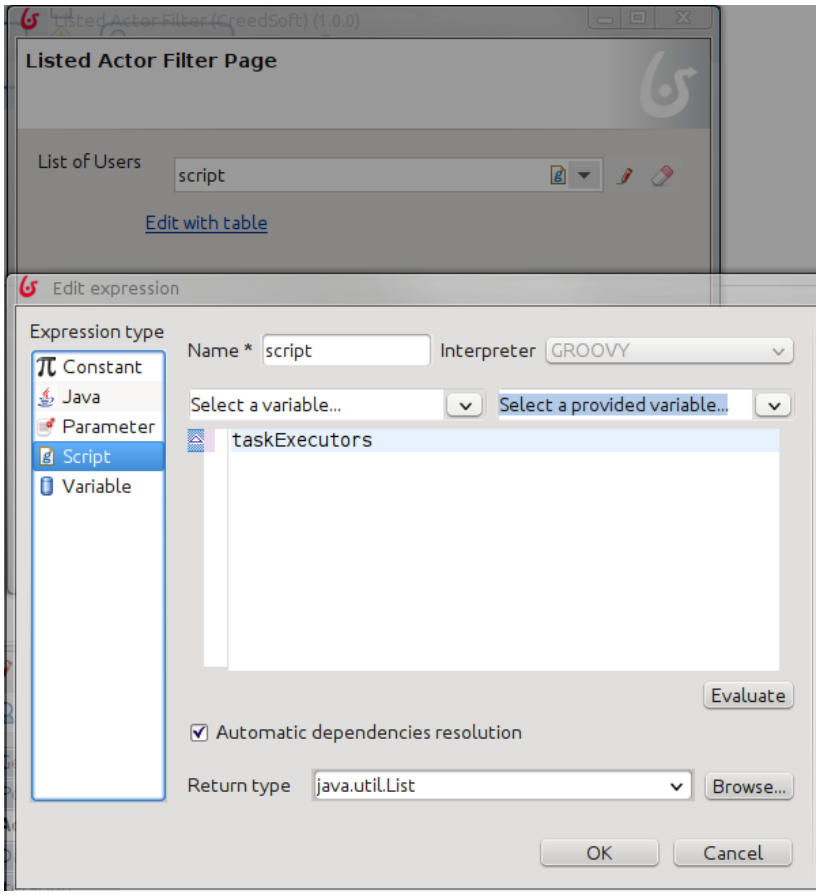
A *taskExecutors* értékét 3 helyen módosítjuk:

1. A workflow instance indító formon. Amit itt megadunk, azokhoz fog menni a feladat (ezt láthatjuk a 10.6. ábra képernyőjén).
2. Az 1. step formján
3. A 2. step formján

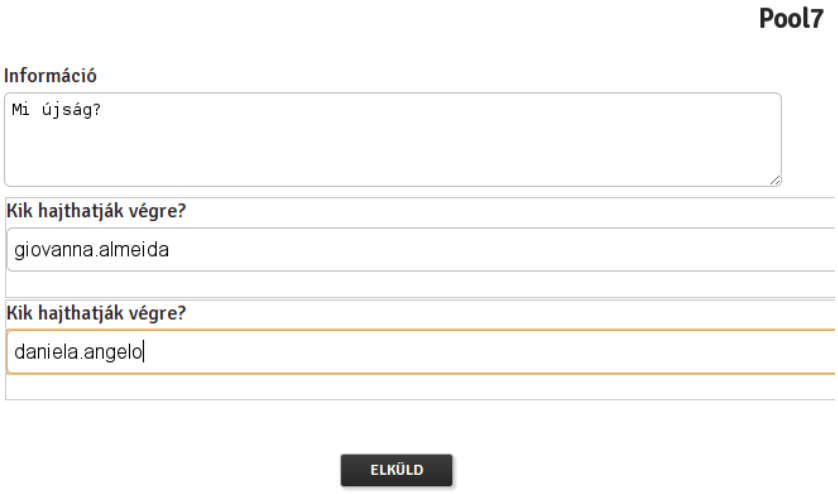
A 10.4. ábra szerinti módon konfiguráljuk rá mindkét step-re a filterünk, ahol a 10.5. ábra szerinti képernyőn adtuk meg az inputot a filternek. A teszteléshez ezeket a user neveket használtuk: giovanna.almeida, daniela.angelo, anthony.nichols, thomas.wallis, michael.morrison.




10.4. ábra. A *ListedActorFilter* használata a *Step1* taszknál




10.5. ábra. A *ListedActorFilter* példány konfigurálása




10.6. ábra. A tesztelés elkezdése

# 11. A Bonita 6. Web Restful API

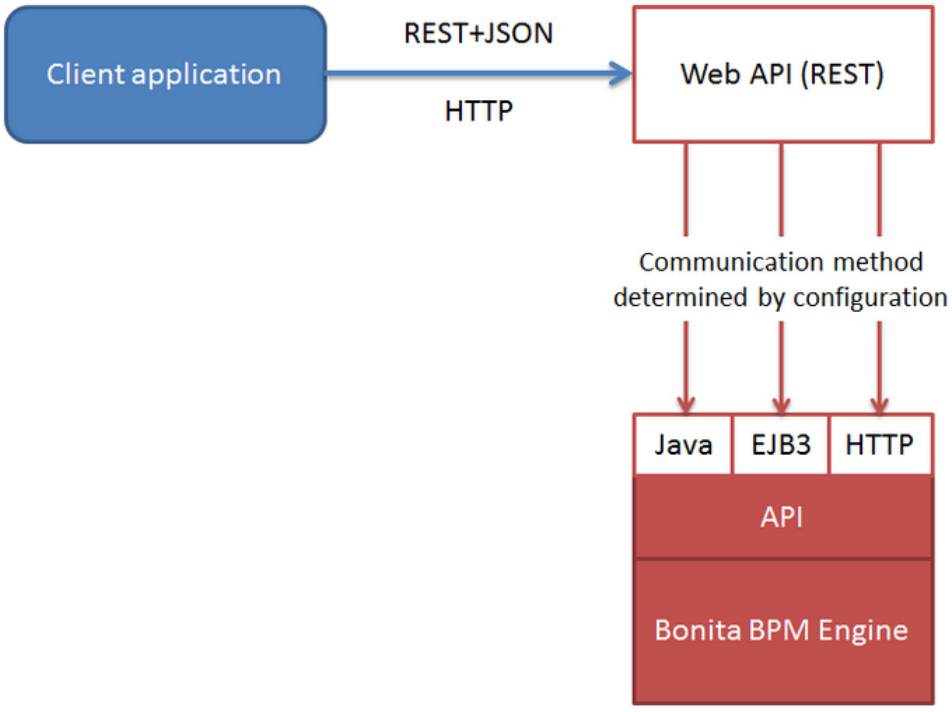
Ebben a fejezetben a Bonita engine a platform független *RESTful* protokollon keresztül történő elérését tekintjük át. Ez a lehetőség azért jelentős, mert ez lehetővé teszi a szolgáltatások elérését nem Java (.NET, PHP) környezetből is. A *REST* (Representational State Transfer) egy szoftver-architektúra típus elosztott hipermédia rendszerek számára, amilyen például a világháló. Azokat a rendszereket, amelyek eleget tesznek a *REST* megszorításainak, *RESTful*-nak nevezik.

## A Restful API eléréséhez használható eszközök

Aki még nem ismeri a Restful technológiát, az is könnyen használhatja ezt az API-t, de javasoljuk, hogy legalább a wiki leírását tekintse át: `http://hu.wikipedia.org/wiki/REST`. A *JAX-RS* Java API könnyű módszert biztosít új szervizek készítéshez, ezért ezt is érdemes átfutni: `http://hu.wikipedia.org/wiki/JAX-RS`. Jelenleg 2 elterjedtebb lehetőséget szoktak használni a Java Restful kliens készítéséhez:

1. A *Jersey* nevű *JAX-RS* referencia implementáció (webhely: `https://jersey.java.net/`)
2. *Apache HTTP Components* (webhely: `http://hc.apache.org/httpcomponents-client-ga/`)

Ebben a részben mi a *HTTP Components* Java könyvtárat fogjuk használni, ami egy teljes és megbízható HTTP protokoll implementáció.




11.1. ábra. A Restful API kialakítása

A 11.1. ábra a Bonita Restful elérésének kialakítását mutatja. Láthatjuk, hogy a 3 natív API elérési lehetőség tetejére egy *Web API* szoftver réteg került, ami egyik irányba a Restful kliensek felé mutatja az arcát, a Bonita irányába pedig használja az ismert API elérési módszerek valamelyikét (Java local, EJB 3, HTTP Java klienssel). Az ábráról az is látszik, hogy a Restful felett a *JSON*[8] formátum használt az adatok cseréjére, ehhez a *json-lib* Java könyvtárat használhatjuk (webhely: `http://json-lib.sourceforge.net/`). A továbbiakban bemutatott konkrét példák a Bonita stúdióból elindított engine-t használják, így azt bárki gyorsan ki tudja próbálni.

## A Restful API használata - Egy felhasználó adatainak lekérdezése

### A login és logout használata

A Restful API használatának első és utolsó lépése mindig a login és logout, ezért először ezt mutatjuk meg. A hitelesítést a Bonita portál végzi, ami sikeres esetben egy cookie-t ad vissza erről. A következő kéréseket már ez a süti fogja hitelesíteni, így annak mindig a http kérés fejlécében kell lennie. A login művelet egy http post a Bonita portál felé úgy, ahogy a következő táblázat ezt definiálja.

*Bejelentkezés (login operation) a Bonita szerverbe*

| | |
|---|---|
| Request URL | http://host:port/bonita/loginservice |
| Request Method | POST |
| Form Data | *username*: a felhasználó login neve<br>*password*: a felhasználó jelszava<br>*redirectURL*: Kell átirányítás a Bonita portálra? Legyen *false*. |

Az URL eleje természetesen az éppen aktuálisan használni kívánt Bonita szerver, ami egy távoli gépen is lehet. A kérésnek 3 paraméteres van. A *redirectURL* megadása azért kell, mert most nem a portál web alkalmazást akarjuk használni, így az oda való átirányítás szükségtelen. Itt jegyezzük meg, hogy a Portál alkalmazás is ezt a hitelesítést használja. Amikor végeztünk a feladatokkal, akkor a következő táblázat szerint megadott http GET hívással kell kijelentkeznünk.

*Kijelentkezés (logout operation) a Bonita szerverből*

| | |
|---|---|
| Request URL | http://host:port/bonita/logoutservice |
| Request Method | GET |
| Form Data | *redirectURL*: Kell átirányítás a Bonita portálra? Legyen *false*. |

A 11-1. Programlista ezen 2 művelet használatát mutatja be. A *loginBaseURL* (26. sor) a bejelentkezéshez szükséges portál, míg a *restBaseURL* (27.sor) a szolgáltatások URL-jének az eleje. A példáinkban fokozatosan kialakítandó *BonitaRestFacade* class konstruktora ezt a két értéket

[8] *http://hu.wikipedia.org/wiki/JSON*

kapja meg és elmenti őket. A *login()* művelet a 36-61 sorok között lett implementálva. Itt egy HTTP POST művelet szükséges, ezért az Apache HTTP Client *HttpPost* osztályát használjuk (38. sor). A 39-43 sorok a POST elküldendő paramétereit készítik elő. A 46. sorban történik a tényleges POST művelet, aminek az eredményét vissza is olvassuk a 48-54 sorok között. Mindeközben a naplót is érdemes itt nézni: *.../workspace/tomcat/logs*. A 63-76 sorok között a *logout()* kerül megvalósításra. Ez egy HTTP GET művelet, ami csak a *redirect* paramétert használja. Tekintettel arra, hogy GET esetén a paraméterek az URL-be vannak kódolva, ezért érdemes használni a builder mintával működő *URIBuilder* osztályt.

11-1. Programlista: A BonitaRestFacade class fejlesztése

```
1  package org.cs.bonita.rest;
2
3  import java.io.BufferedReader;
4  import java.io.IOException;
5  import java.io.InputStreamReader;
6  import java.net.URI;
7  import java.util.ArrayList;
8  import java.util.List;
9  import net.sf.json.JSONObject;
10 import org.apache.http.HttpResponse;
11 import org.apache.http.NameValuePair;
12 import org.apache.http.client.ClientProtocolException;
13 import org.apache.http.client.HttpClient;
14 import org.apache.http.client.entity.UrlEncodedFormEntity;
15 import org.apache.http.client.methods.HttpGet;
16 import org.apache.http.client.methods.HttpPost;
17 import org.apache.http.client.utils.URIBuilder;
18 import org.apache.http.impl.client.DefaultHttpClient;
19 import org.apache.http.message.BasicNameValuePair;
20
21 public class BonitaRestFacade
22 {
23 //      String urlLogin  = "http://localhost:8080/bonita/loginservice";
24 //      String urlLogout = "http://localhost:8080/bonita/logoutservice";
25
26         String loginBaseURL = null;
27         String restBaseURL  = null;
28         HttpClient client = new DefaultHttpClient();
29
30         public BonitaRestFacade(String loginBaseUrl, String restBaseURL)
31         {
32                 this.loginBaseURL = loginBaseUrl;
33                 this.restBaseURL  = restBaseURL;
34         }
35
36         public void login(String user, String password) throws Throwable
37         {
38                 HttpPost post = new HttpPost(loginBaseURL + "/loginservice");
39                 List<NameValuePair> nameValuePairs = new ArrayList<NameValuePair>();
40                 nameValuePairs.add(new BasicNameValuePair("username", user));
41                 nameValuePairs.add(new BasicNameValuePair("password", password));
42                 nameValuePairs.add(new BasicNameValuePair("redirect", "false"));
43                 post.setEntity(new UrlEncodedFormEntity(nameValuePairs));
44                 try
45                 {
46                         HttpResponse response = client.execute(post);
47
48                         BufferedReader rd = new BufferedReader(new InputStreamReader(response.➥
                               getEntity().getContent()));
49                         String line = "";
```

```
50
51                          while ((line = rd.readLine()) != null)
52                          {
53                                  System.out.println(line);
54                          }
55
56                  } catch (Throwable e)
57                  {
58                          e.printStackTrace();
59                          throw e;
60                  }
61          }
62
63          public void logout() throws Throwable
64          {
65                  URI uri = (new URIBuilder(loginBaseURL + "/logoutservice").addParameter("➥
                        redirect", "false")).build();
66                  HttpGet request = new HttpGet( uri );
67
68                  try
69                  {
70                          HttpResponse response = client.execute(request);
71                  } catch (Throwable e)
72                  {
73                          e.printStackTrace();
74                          throw e;
75                  }
76          }
77  ...
78  } // end BonitaRestFacade
```

### Egy felhasználó adatainak lekérdezése

Működik a login és logout, így legyen a következő feladatunk az, hogy lekérdezzük valamelyik user adatait. Ehhez az *identity API-t* kell használnunk, amit a *restBaseURL + /identity/* helyről tudunk használni és lekérdezés lévén a HTTP GET operátor használható. A 11-2. Programlista mutatja a megoldást, ahol most */user* műveletet kérjük az *id* értékre, ami a user ID. A hívás JSON formában adja vissza a user rekordot, amit a *getUser()* változtatás nélkül továbbad.

11-2. Programlista: A BonitaRestFacade class fejlesztése - getUser(long id)

```
1   package org.cs.bonita.rest;
2   ...
3   public class BonitaRestFacade
4   {
5           String loginBaseURL = null;
6           String restBaseURL  = null;
7           HttpClient client = new DefaultHttpClient();
8   ...
9           public String getUser(long id) throws Exception
10          {
11          HttpGet request = new HttpGet(restBaseURL + "/identity/user/" + id);
12                  HttpResponse response = client.execute(request);
13                  StringBuffer sb = new StringBuffer();
14
15                  BufferedReader rdx = new BufferedReader(new InputStreamReader(response.getEntity➥
                        ().getContent()));
16                  String line = "";
17
18                  while ((line = rdx.readLine()) != null)
```

```
19                  {
20                          sb.append( line );
21                  }
22
23                  return sb.toString();
24          }
25  ...
26          public static void main(String[] args) throws Throwable
27          {
28                  System.out.println("Start...");
29                  BonitaRestFacade prg = new BonitaRestFacade("http://localhost:8080/bonita", "➥
                        http://localhost:8080/bonita/API");
30
31                  prg.login("daniela.angelo", "bpm");
32                  JSONObject jsonObj = JSONObject.fromObject( prg.getUser(1 ));
33                  System.out.println( prg.getUser(1) );
34                  System.out.println( jsonObj.get("userName") );
35                  prg.logout();
36                  System.out.println("Stop...");
37          }
38  } // end BonitaRestFacade
```

A működést a 26-37 sorok között található *main()* metódussal le is teszteltük. A 11-3. Programlista mutatja a visszaadott USER objektum JSON string reprezentációját, amit a 33. sor ír ki a képernyőre. A 32. sor azt is megmutatja, hogy a json-lib könyvtár segítségével milyen módon tudjuk ezt a stringet parsolni és a mezőit elérni. Kipróbáltunk egy fontos dolgot is, de ezt a kód most nem mutatja. A *getUser()* metódust meghívtuk a *logout()* után is és azt kaptuk, amit vártunk. Egy kijelentkezett felhasználóra már kivételt dob a program.

11-3. Programlista: A visszaadott USER objektum JSON reprezentációja

```
1  {"last_connection":"","created_by_user_id":"-1","creation_date":"2013-12-25 09:27:45.344","id":"1","icon":"/➥
       default/icon_user.png","enabled":"true","title":"Mr","manager_id":"0","job_title":"Chief Executive Officer"➥
       ,"userName":"william.jobs","lastname":"Jobs","firstname":"William","password":"","last_update_date":"➥
       2013-12-25 09:27:45.344"}
```

## A Bonita Restful API áttekintése

A Bonita Restful felület lehetővé teszi, hogy elérjük ezeket az API-kat:

- *identity*: Az organization (user, group, role, membership) karbantartásához szükséges műveletek. Ide tartozott az előző példa *getUser()* implementációjának háttere is.
- *system*: A nyelvi beállítások kezelése (például a session magyar nyelvre állítása).
- *portal*: A különféle portál műveletek elérését biztosítja (profile kezelés, a felhasználó által végrehajtható akciók elérése).
- *bpm*: Minden művelet, ami a munkafolyamat kezeléséhez szükséges (például egy case-hez kötött comment lekérdezése).

Egy Restful API URL általános szerkezete így néz ki:

```
http://host:port/bonita/API/{API_name}/{resource_name}/
```

A fenti 4 API létezik, amikre a következőekben megadott táblázatokban lévő resource-okat lehet használni. A korábbi *getUser()* példánkban ez így alakult:

- *http://localhost:8080/bonita/API*
- *{API_name}* → *identity*
- *{resource_name}* → *user*

A kérés és válasz adatcsere formátuma a JSON. Egy erőforrás olvasása például ilyen elemekből tevődik össze, azaz így specifikálódik a felhasználó programozó felé:

- Request URL: *http://.../API/{API_name}/{resource_name}/{id}*. Az *id* a kért objektum belső *Long* azonosítója.
- HTTP Request Method: *GET*
- Response: Egy JSON string
- Példa: *http://.../API/identity/user/5*

**Az *identity* API áttekintése**

| Resource | Leírás |
|---|---|
| *user* | A felhasználó adatai |
| *role* | A szerepkörök, amiket a csoporton belül használhatunk |
| *group* | A csoportok, amik hierarchikusak |
| *membership* | A csoport és szerepkör relációk |
| *professionalcontactdata* | A felhasználó szakmai kontakt információi |
| *personalcontactdata* | A felhasználó személyes kontakt információi |

**A *system* API áttekintése**

| Resource | Leírás |
|---|---|
| *i18nlocale* | A kliens *locale* lehetőségek használata |
| *i18ntranslation* | A nyelvfüggő fordítások elérése |
| *session* | A session elérése |

**A *portal* API áttekintése**

| Resource | Leírás |
|---|---|
| *profile* | Az elérési jogokat szabályozó profile beállítása |
| *profileEntry* | Egy jogosultsági elem beállítása egy adott profilhoz, ami biztosítja azt, hogy erre az akcióra joga legyen az ilyen profilba tartozó user-nek |
| *profileMember* | Ez egy user, aki ebbe a profilba tartozik |

### A *bpm* API áttekintése

| Resource | Leírás |
|---|---|
| *activity* | Ez az erőforrás egy ismert Bonita activity (humán vagy service task, call activity vagy subprocess) elérését biztosítja |
| *archivedActivity* | Egy olyan activity, ami egy archivált process instance része |
| *humanTask* | Egy olyan TASK a processen belül, amit egy ember hajt végre |
| *userTask* | Egy olyan humán TASK, amit egy user fog végrehajtani |
| *archivedHumanTask* | Egy olyan humán TASK, ami egy már archivált processben van |
| *archivedUserTask* | Egy olyan user TASK, ami egy már archivált processben van |
| *process* | A process definíciója (nem process instance, vagy más néven case) |
| *category* | Egy név, ami összefog több process definíciót. A portálon is külön csoportosítva jelennek meg ezek a process template-ek. |
| *processCategory* | Egy process, ami egy kategóriához tartozik |
| *processConnector* | Egy konnektor, amit egy process használ |
| *case* | A process instance (példány) egyik megnevezése |
| *archivedCase* | Egy befejezett process példány |
| *comment* | Egy aktív process példányhoz tartozó megjegyzés |
| *archivedComment* | Egy archivált process példányhoz tartozó megjegyzés |
| *actor* | Egy név, ami reprezentálja azt, aki végre fogja hajtani a feladatot |
| *actorMember* | Egy user, aki hozzá van rendelve az actor-hoz |
| *hiddenUserTask* | Egy rejtett TASK (a portálról is el lehet végezni az elrejtést) |
| *task* | Egy TASK a processen belül |
| *archivedTask* | Egy TASK, ami egy archivált process példányhoz tartozik |
| *archivedFlowNode* | Egy gateway vagy event node, ami egy process példányhoz tartozik |
| *processResolutionProblem* | Egy elem a processben, amiben egy hivatkozást fel kell oldani |
| *caseDocument* | Egy process példányhoz csatolt dokumentum |
| *connectorInstance* | Egy példányosított konnektor, ami egy process példányhoz tartozik |
| *archivedConnectorInstance* | Az előző *connectorInstance*, ami egy archivált process példányhoz tartozik |
| *processConnectorDependency* | A konnektor függőségei külső jar fájloktól |
| *caseVariable* | Egy process példány változó |
| *processParameter* | Egy process paraméter |
| *manualTask* | Egy külső TASK, aminek csak a státusza kezelt a workflowban |
| *archivedManualTask* | A *manualTask*, ami egy archivált process példányhoz tartozik |
| *connectorFailure* | Egy error üzenetet ad, amikor a konnektor hibára fut |

## A Restful API áttekintése

### Írás és olvasás

Az erőforrás olvasást már röviden megnéztük a fentiekben, most nézzük meg, hogy milyen lehetőségek vannak még! Egy erőforrás létrehozása a Bonita Restful API-n keresztül ezeket az adatokat

jelenti:

- Request URL: *http://.../API/{API_name}/{resource_name}/*
- HTTP Request Method: *POST*
- Request Payload: JSON string
- Response: ugyanaz a JSON string, esetleg kiegészítve a létrehozás utáni mezőkkel

Egy összetett azonosítójú erőforrás olvasás használata:

- Request URL: *http://.../API/{API_name}/{resource_name}/{id_part1}/{id_part2}*
- HTTP Request Method: *GET*
- Response: JSON string
- Példa: *http://.../API/identity/membership/5/12/24*

Egy erőforrás update elemei:

- Request URL: *http://.../API/{API_name}/{resource_name}/{id}*
- HTTP Request Method: *PUT*
- Request Payload: Egy Map a JSON-on belül, ami tartalmazza az új értékeket azokra az attribútumokra, amikor frissíteni szeretnénk.
- Response: A megváltoztatott JSON string
- Példa: *http://.../API/identity/user/5*

Egy összetett azonosítójú erőforrás update használata:

- Request URL: *http://.../API/{API_name}/{resource_name}/{id_part1}/{id_part2}*
- HTTP Request Method: *PUT*
- Request Payload: Egy Map a JSON-on belül, ami tartalmazza az új értékeket azokra az attribútumokra, amikor frissíteni szeretnénk.
- Response: A megváltoztatott JSON string
- Példa: *http://.../API/identity/membership/5/12/24*

A multiple update használata:

- Request URL: *http://.../API/{API_name}/{resource_name}/*
- HTTP Request Method: *PUT*

- Request Payload: Elemek listája, amik tartalmazzák az új értékeket is
- Response: Ugyanaz a lista, de már a megváltoztatott értékű elemekkel
- Példa: *http://.../API/identity/membership*

Egy erőforrás törlése:

- Request URL: *http://.../API/{API_name}/{resource_name}/{id}*
- HTTP Request Method: *DELETE*
- Response: üres vagy egy kivétel dobása
- Példa: *http://.../API/identity/user/5*

Egy erőforrás törlése összetett azonosítóval:

- Request URL: *http://.../API/{API_name}/{resource_name}/{id_part1}/{id_part2}*
- HTTP Request Method: *DELETE*
- Response: üres
- Példa: *http://.../API/identity/membership/5/12/24*

A multiple delete használata:

- Request URL: *http://.../API/{API_name}/{resource_name}/*
- HTTP Request Method: *DELETE*
- Request Payload: A törlendő elemek listája
- Response: üres
- Példa: *http://.../API/identity/membership/*

### Egy erőforrás érték keresése

A Bonita Restful API az objektumok keresését is támogatja, egy-egy keresés esetén a filter feltételeket URL kódolt paraméter értékekkel kell megadni. A visszaadott eredmény egy lapokra bontott lista, ahol az 1 lapon lévő elemek maximális számát is megadhatjuk. A használható filterek a lekért elemek attribútumai, de vannak további speciális szűrők is. A filterezett URL használatának általános formája így adható meg:

- Request URL: *http://.../API/{API_name}/{resource_name}?p={page}&c={count}&o={order}&s={*
- HTTP Request Method: *GET*

- Response: elemek tömbje JSON string formátumban
- Példa: *http://.../API/identity/user?p=0&c=10&o=firstname&s=test&f[]=manager_id:3*

Az URL-ben megadott speciális paraméterek jelentése a következő:

- *p*: A visszakért (megjelenített) lap, annak a tartalma kell most nekünk.
- *c*: A maximum darabszáma a visszaadott elemeknek.
- *o*: Ezzel a lekérdezett eredmény sorrendjét (order) adhatjuk meg (az SQL-hez hasonlóan lehet ASC vagy DESC).
- *f*: Filterek listája. Az item attrubútumokon kívül még adhatunk további jellemzőket is: priority, state, processId, user_id, assignee_id.
- *d*: Attribútumok listája, amiket be szeretnénk vonni a válasz rekordba.

Itt egy teljes példa egy lekérdezésre:

```
http://localhost:8080/bonita/API/bpm/humanTask?p=0&c=10&o=priority%20DESC&f=state%3dready&f=➥
    user_id%3d104&d=processId
```

Ez az alábbiakat valósítja meg:

- TASK-okat kér le,
- a prioritás szerint, csökkenő sorrendben, rendezetten adja vissza (priority DESC)
- lekéri hozzá a process objektumot (d=processID)
- alkalmaz 2 darab filtert (state=ready és user_id=104)

## A JAX-RS alapú Restful kliens használata

A 11-4. Programlista egy JAX-RS API alapján készített utility, aminek minden egyes metódusát érdemes átnézni. Itt most nem célunk, hogy a JAX-RS használatát ismertessük, de talán anélkül is érthetőek ezek a rövid metódusok. Működésük a fentiekben ismertetett elemekre épülnek.

11-4. Programlista: A BonitaAPIClient

```
1
2  package org.bonitasoft.test.toolkit.api;
3
4  import java.util.List;
5
6  import javax.ws.rs.Consumes;
7  import javax.ws.rs.DELETE;
8  import javax.ws.rs.FormParam;
9  import javax.ws.rs.GET;
10 import javax.ws.rs.POST;
11 import javax.ws.rs.PUT;
12 import javax.ws.rs.Path;
13 import javax.ws.rs.PathParam;
```

```
14 import javax.ws.rs.Produces;
15 import javax.ws.rs.QueryParam;
16
17 import org.jboss.resteasy.client.ClientResponse;
18
19 /**
20  * JSON API interface.
21  *
22  * @author truc
23  */
24 public interface BonitaAPIClient {
25
26     /** User. */
27     String USER_API_PATH = "API/identity/user";
28
29     /** Group. */
30     String GROUP_API_PATH = "API/identity/group";
31
32     /** Role. */
33     String ROLE_API_PATH = "API/identity/role";
34
35     /** Process. */
36     String PROCESS_API_PATH = "API/bpm/process";
37
38     /** Actor. */
39     String ACTOR_API_PATH = "API/bpm/actor";
40
41     /** Actor mapping. */
42     String ACTORMEMBER_API_PATH = "API/bpm/actorMember";
43
44     /** Profile. */
45     String PROFILE_API_PATH = "API/portal/profile";
46
47     /** Translation. */
48     String TRANSLATION_API_PATH = "API/system/i18ntranslation";
49
50     /**
51      * Login.
52      *
53      * @param pUserName
54      * @param pPassword
55      * @return
56      */
57     @POST
58     @Path("loginservice")
59     ClientResponse<String> login(@FormParam("username") String pUserName,
60             @FormParam("password") String pPassword);
61
62     /**
63      * Logout.
64      *
65      * @return
66      */
67     @GET
68     @Path("logoutservice")
69     ClientResponse<String> logout();
70
71     /**
72      * Search.
73      *
74      * @return
75      */
76     @GET
77     @Path("{path}")
78     @Produces("application/json")
```

```
79      ClientResponse<String> search(@PathParam("path") String path, @QueryParam("p") int pParam, ➥
            @QueryParam("c") int cParam, @QueryParam("o") String oParam,
80              @QueryParam("f") String fParam,
81              @QueryParam("d") String dParam, @QueryParam("n") String nParam);
82
83      /**
84       * Get users.
85       *
86       * @param pStart
87       * @param pCount
88       * @return
89       */
90      @GET
91      @Path(USER_API_PATH)
92      @Produces("application/json")
93      ClientResponse<String> getUsers(@QueryParam("p") int pStart, @QueryParam("c") int pCount);
94
95      /**
96       * Delete users.
97       *
98       * @param pBody
99       * @return
100      */
101     @DELETE
102     @Path(USER_API_PATH)
103     @Consumes("application/json")
104     ClientResponse<String> deleteUsers(String pBody);
105
106     /**
107      * Create a user.
108      *
109      * @param pBody
110      * @return
111      */
112     @POST
113     @Path(USER_API_PATH)
114     @Consumes("application/json")
115     ClientResponse<String> createUser(String pBody);
116
117     /**
118      * Set user manager.
119      *
120      * @param pUserId
121      * @param pBody
122      * @return
123      */
124     @PUT
125     @Path(USER_API_PATH + "/{id}")
126     @Consumes("application/json")
127     ClientResponse<String> setUserManager(@PathParam("id") String pUserId, String pBody);
128
129     /**
130      * Delete groups.
131      *
132      * @param pBody
133      * @return
134      */
135     @DELETE
136     @Path(GROUP_API_PATH)
137     @Consumes("application/json")
138     ClientResponse<String> deleteGroups(String pBody);
139
140     /**
141      * Create a group.
142      *
```

```
143         * @param pBody
144         * @return
145         */
146        @POST
147        @Path(GROUP_API_PATH)
148        @Consumes("application/json")
149        ClientResponse<String> createGroup(String pBody);
150
151        /**
152         * Delete groups.
153         *
154         * @param pBody
155         * @return
156         */
157        @DELETE
158        @Path(ROLE_API_PATH)
159        @Consumes("application/json")
160        ClientResponse<String> deleteRoles(String pBody);
161
162        /**
163         * Create a group.
164         *
165         * @param pBody
166         * @return
167         */
168        @POST
169        @Path(ROLE_API_PATH)
170        @Consumes("application/json")
171        ClientResponse<String> createRole(String pBody);
172
173        /**
174         * Add to profile.
175         *
176         * @param pBody
177         * @return
178         */
179        @POST
180        @Path("API/userXP/profileMember")
181        @Consumes("application/json")
182        ClientResponse<String> addToProfile(String pBody);
183
184        /**
185         * Get profiles.
186         *
187         * @param pStart
188         * @return
189         */
190        @GET
191        @Path(PROFILE_API_PATH)
192        @Produces("application/json")
193        ClientResponse<String> getProfiles(@QueryParam("p") int pStart, @QueryParam("c") int cParam,➥
               @QueryParam("o") String pOrder);
194
195        /**
196         * Delete profiles.
197         *
198         * @param pBody
199         * @return
200         */
201        @DELETE
202        @Path(PROFILE_API_PATH)
203        @Consumes("application/json")
204        ClientResponse<String> deleteProfiles(String pBody);
205
206        /**
```

```
207         * Create profile.
208         *
209         * @param pBody
210         * @return
211         */
212        @POST
213        @Path(PROFILE_API_PATH)
214        @Consumes("application/json")
215        ClientResponse<String> createProfile(String pBody);
216
217        /**
218         * Install a process.
219         *
220         * @param pBody
221         * @return
222         */
223        @POST
224        @Path(PROCESS_API_PATH)
225        @Consumes("application/json")
226        ClientResponse<String> installProcess(String pBody);
227
228        /**
229         * Import organization.
230         *
231         * @param pBody
232         * @return
233         */
234        @POST
235        @Path("services/organization/import")
236        @Consumes("application/json")
237        ClientResponse<String> importOrganization(String pBody);
238
239        /**
240         * Get processes.
241         *
242         * @param pStart
243         * @return
244         */
245        @GET
246        @Path(PROCESS_API_PATH)
247        @Produces("application/json")
248        ClientResponse<String> getProcesses(@QueryParam("o") String pOrder, @QueryParam("f") String ➥
               pFilterExpression);
249
250        /**
251         * Set process state.
252         *
253         * @param pProcessId
254         * @param pBody
255         * @return
256         */
257        @PUT
258        @Path(PROCESS_API_PATH + "/{id}")
259        @Consumes("application/json")
260        ClientResponse<String> setProcessState(@PathParam("id") String pProcessId, String pBody);
261
262        /**
263         * Set process display name.
264         *
265         * @param pProcessId
266         * @param pBody
267         * @return
268         */
269        @PUT
270        @Path(PROCESS_API_PATH + "/{id}")
```

```
271     @Consumes("application/json")
272     ClientResponse<String> setProcessDisplayName(@PathParam("id") String pProcessId, String pBody);
273
274     /**
275      * Delete processes.
276      *
277      * @param pBody
278      * @return
279      */
280     @DELETE
281     @Path(PROCESS_API_PATH)
282     @Consumes("application/json")
283     ClientResponse<String> deleteProcesses(String pBody);
284
285     /**
286      * Get members mapped to an actor.
287      *
288      * @param pStart
289      * @return
290      */
291     @GET
292     @Path(ACTORMEMBER_API_PATH)
293     @Produces("application/json")
294     ClientResponse<String> getActorMembers(@QueryParam("p") int pStart, @QueryParam("c") int pCount, @QueryParam("o") String pOrder,
295             @QueryParam("f") List<String> pFilterExpressions);
296
297     /**
298      * Create a relationship between member and actor.
299      *
300      * @param pBody
301      * @return
302      */
303     @POST
304     @Path(ACTORMEMBER_API_PATH)
305     @Consumes("application/json")
306     ClientResponse<String> mapToActor(String pBody);
307
308     /**
309      * Delete members.
310      *
311      * @param pBody
312      * @return
313      */
314     @DELETE
315     @Path(ACTORMEMBER_API_PATH)
316     @Consumes("application/json")
317     ClientResponse<String> deleteActorMembers(String pBody);
318
319     /**
320      * get actors for a process.
321      *
322      * @param pBody
323      * @return
324      */
325     @GET
326     @Path(ACTOR_API_PATH)
327     @Consumes("application/json")
328     ClientResponse<String> getActors(@QueryParam("p") int pStart, @QueryParam("c") int pCount, @QueryParam("o") String pOrder,
329             @QueryParam("f") String pFilterExpression);
330
331 }
```

# 12. A Bonita 6. produktív környezet telepítése

Most a Bonita 6.x szerver produktív telepítését mutatjuk be. A leírás alapja egy nagyvállalatnál ténylegesen használt éles környezet. A telepítés természetesen a Bonita leírás alapján készült, azonban tartalmaz néhány figyelemre méltó érdekességet is, ezért ezen rész áttanulmányozása segíthet azokon, akik az éles környezetüket szeretnék üzembiztosra és a vállalati policy-t is figyelembe véve kialakítani.

## A telepítés fő lépései

Mielőtt nekikezdünk a Bonita produktív szerver környezet telepítési leírásának, előtte érdemes egy gyors áttekintést tenni arról, hogy ez milyen lépésekből fog állni. Magát a telepítést egy Suse Linux Enterprise Server környezetben fogjuk megtenni, ahol alkalmazás szerverként a JBOSS-t választottuk. A Bonita 6.x futtató környezeteként a Java 6 JDK verziót választottuk.

1. A JBOSS+Bonita integrált csomag kibontása egy adott könyvtárba
2. Fizetős Bonita esetén a licence fájl legenerálása és feltelepítése
3. Oracle adatbázis konfiguráció (a Bonita perzisztencia beállítása Oracle alapra)
4. A JBOSS első indítása, a Bonita portál első használata
5. Az *LDAP synchonizer* konfigurálása
6. Opcionális lépés: A Bonita portál testreszabása
7. További, az adott működési környezetre jellemző beállítások

## A Bonita 6.x letöltése és kicsomagolása

A Bonita egyes verzióit innen lehet letölteni:

- Community Edition: `http://community.bonitasoft.com/`
- Subscription Edition: `http://www.bonitasoft.com/resources/customer-portal`

A subscription edition többféle licence konstrukcióban vásárolható meg és tartalmaz több olyan funkciót, amit a community edition-ben külön fejlesztéssel kéne megvalósítani. A 2 lehetőség között az is különbség, hogy a fizetős verzióra support van és elérhető a Bonita Customer Portál. Két fájlt kell letöltenünk (a telepítéshez a Bonita 6.1.2 verzióit választottuk):

- *BonitaBPMSubscription-6.1.2-JBoss-5.1.0.GA.zip* → A JBOSS+Bonita integrált szerver csomag

- *BonitaBPMSubscription-6.1.2-deploy.zip* → Ez minden olyan komponenst tartalmaz, amivel egy tetszőleges alkalmazás szerveren is kialakítható a Bonita futtató környezet.

A JBOSS mellett Tomcat-re is lehetséges egy előre összeintegrált csomag letöltése. A bemutatott környezetben a Linux */opt/bonita* könyvtára alá csomagoltuk ki az első *zip* fájlt, amely könyvtárat a továbbiakban *JBOSS_HOME* néven fogunk nevezni. Ezzel már van is egy *HSQLDB* adatbázist használó Bonita szerverünk, amihez első lépésként szerezzük meg a licence fájlt. Amennyiben community edition-t használunk, úgy erre a lépésre természetesen nem lesz szükség.

## A licence fájl megszerzése és telepítése

A *JBOSS_HOME/request_key_utils* könyvtárba lépjünk be és az ott található scriptet futtassuk le:

```
./generateRequestForAnyEnvironment.sh

(MWwwJcuOKDnSQdphFsUJ6LBSm8CUWPzxR35Q7GxwIeH9+Oy6KHkyJh27+Nu5aSok)
```

A shell script lefutása után a képernyőre kiíródik egy 2-3 soros licence kérő karaktersorozat, ami „*(...)*” jelek között van. Ezt másoljuk a vágólapra, majd menjünk a Bonita Customer portálon ide: `http://www.bonitasoft.com/resources/customer-portal/license/request`. Töltsük ki a következő mezőket:

- *Start Date* (example : 2011-12-30): Ettől a naptól lesz a licence fájl érvényes
- *License type*: Development, Production, Qualification
- *Assignee Name*: Egy ember neve, aki kezeli a licence ügyeket
- *Assignee Email*: Az előző ember e-mail címe, ide fog postázódni a generált licence fájl
- *Choose a version*: Valamelyik Bonita verziót kell kiválasztanunk
- *Request Key*: A fenti script által generált karaktersorozat kerül ide (fontos: a nyitó és csukó zárójelek is!)

Nyomjuk meg a licence fájl kérő *Request* gombot, majd várjuk meg a licence fájl érkezését abba mailbox-ba, amit megadtunk. Ez egy *lic* kiterjesztésű fájl lesz, amit be kell másolnunk ide:

```
JBOSS_HOME/bonita/server/licenses
```

Emlékezzünk arra, hogy a *JBOSS_HOME/bonita* részét a könyvtárnak Bonita Home néven is nevezzük, így azt is mondhatjuk, hogy a licence fájlt a Bonita Home *server/licenses* könyvtárába kell elhelyezni.

**Megjegyzés** A licence fájl kérés generálásánál megváltoztathatjuk a CPU core-ok számát, például 8 helyett mondhatjuk, hogy 2. Ekkor a licence csak 2 processor core-ra lesz érvényes, amit úgy tudunk biztosítani Linux alatt, hogy használjuk a *taskset* parancsot, amikor indítjuk a JBOSS-t:

```
taskset -c [list_of_cpu_cores_ids] [your_command]
```

Ez egy opcionális lehetőség, amikor a licence-szel akarunk gazdálkodni vagy nincs megfelelő mennyiség a birtokunkban. Példa arra, amikor 2 darab core-t engedélyezünk a *run.sh* (JBOSS) számára:

```
taskset -c 0,1 /opt/bonita/server/bin/run.sh
```

## Az Oracle környezet konfigurálása

A most következő lépés sorozatban az előre bekonfigurált HSQLDB adatbáziskezelő helyett az Oracle-t fogjuk beállítani, ami enterprise megoldások esetén biztosabb és jobban menedzselhető.

### A használt Oracle JDBC driver telepítése

Esetünkben az *ojdbc6.jar* JDBC drivert használtuk, aminek a telepítése mindössze annyi, hogy bemásoljuk ebbe a könyvtárba:

```
JBOSS_HOME/server/default/lib
```

### Két új Oracle datasource készítése és telepítése

Bármilyen néven is elkészíthetnénk, de maradva a Bonita elnevezésnél hozzuk létre a *bonita-ds.xml* fájlt, ilyen tartalommal (a host és port helyett mindenki írja be a saját adatbázis szerverének az elérhetőségét):

```
<datasources>

<!-- Oracle -->
        <xa-datasource>
                <jndi-name>bonitaDS</jndi-name>
                <use-java-context>false</use-java-context>
                <xa-datasource-class>oracle.jdbc.xa.client.OracleXADataSource</xa-datasource-class>
                <xa-datasource-property name="URL">jdbc:oracle:thin:@host:1521:testutf8</xa-datasource-property>
                <xa-datasource-property name="User">bonitatest</xa-datasource-property>
                <xa-datasource-property name="Password">TokMindegy_123</xa-datasource-property>
                <track-connection-by-tx/>
        </xa-datasource>

        <no-tx-datasource>
                <jndi-name>bonitaSequenceManagerDS</jndi-name>
                <connection-url>jdbc:oracle:thin:@host:1521:testutf8</connection-url>
                <driver-class>oracle.jdbc.OracleDriver</driver-class>
                <use-java-context>false</use-java-context>
                <user-name>bonitatest</user-name>
                <password>TokMindegy_123</password>
                <check-valid-connection-sql>SELECT 1 FROM dual</check-valid-connection-sql>
                <background-validation-millis>30000</background-validation-millis>
                <idle-timeout-minutes>0</idle-timeout-minutes>
        </no-tx-datasource>

        <xa-datasource>
                <jndi-name>bonitaXADataDS</jndi-name>
                <use-java-context>false</use-java-context>
                <xa-datasource-class>oracle.jdbc.xa.client.OracleXADataSource</xa-datasource-class>
```

```
                <xa-datasource-property name="URL">jdbc:oracle:thin:@host:1521:testutf8</xa-➥
                    datasource-property>
                <xa-datasource-property name="User">bonita_data_test</xa-datasource-property>
                <xa-datasource-property name="Password">TokMindegy_123</xa-datasource-property>
                <track-connection-by-tx/>
        </xa-datasource>

        <no-tx-datasource>
                <jndi-name>bonitaDataDS</jndi-name>
                <connection-url>jdbc:oracle:thin:@host:1521:testutf8</connection-url>
                <driver-class>oracle.jdbc.OracleDriver</driver-class>
                <use-java-context>false</use-java-context>
                <user-name>bonita_data_test</user-name>
                <password>TokMindegy_123</password>
                <check-valid-connection-sql>SELECT 1 FROM dual</check-valid-connection-sql>
                <background-validation-millis>30000</background-validation-millis>
                <idle-timeout-minutes>0</idle-timeout-minutes>
        </no-tx-datasource>
</datasources>
```

A fenti XML fájl egy JBOSS datasource definiáló fájl, amit úgy tudunk telepíteni, hogy bemásoljuk ide:

```
JBOSS_HOME/server/default/deploy/bonita-ds.xml
```

Fontos, hogy mindkét sémára ezek a jogok meg legyenek adva (itt most a példa user neve bonita):

```
DROP user bonita cascade;
CREATE USER bonita IDENTIFIED BY bonita;
GRANT connect, resource TO bonita IDENTIFIED BY bonita;
GRANT select ON sys.dba_pending_transactions TO bonita;
GRANT select ON sys.pending_trans$ TO bonita;
GRANT select ON sys.dba_2pc_pending TO bonita;
GRANT execute ON sys.dbms_system TO bonita;
```

### További opcionális Oracle datasource(-ok) telepítése

Az esetlegesen használt további Oracle datasource-ok telepítése egy opcionális, de a gyakorlatban mindig felmerülő lépés. Itt arról van szó, hogy a workflow megoldások is igényelnek Oracle táblákat és egyéb adatbázis lehetőségeket, így ezek számára is szükséges egy vagy több új datasource létrehozása. Ezeket javasoljuk egy *bonita-business-ds.xml* nevű fájlba tenni és azt telepíteni.

### A *bonita-platform.properties* fájl konfigurálása Oracle adatbázisra

Keressük meg ezt a fájlt:

```
JBOSS_HOME/bonita/server/platform/conf/bonita-platform.properties
```

A változtatás abban áll, hogy beállítjuk a *db.vendor* változó értékét:

```
#Try to inherit the property from the System properties
db.vendor=${sysprop.bonita.db.vendor:oracle}
```

Itt a default érték ez volt, ezt tegyük megjegyzésbe így:

```
#Try to inherit the property from the System properties
#db.vendor=${sysprop.bonita.db.vendor:h2}
```

### A *properties-service.xml* fájl konfigurálása Oracle adatbázisra

A kérdéses XML fájl itt található:

```
JBOSS_HOME/server/default/deploy/properties-service.xml
```

Itt a *sysprop.bonita.db.vendor* változó értékét szintén állítsuk Oracle-re, itt látható a fájl releváns része:

```
...
<attribute name="Properties">
      sysprop.bonita.db.vendor=oracle
      file.encoding=UTF-8
</attribute>
...
```

## A Bonita szerver és portál első használata

Ennyi konfiguráció után már van egy licence-elt Bonita szerverünk, ami JBOSS felett fut és Oracle adatbázisban tárolja a működéséhez szükséges adatokat. Elérkeztünk oda, hogy első alkalommal elindíthatjuk a Bonita szervert, tegyük is meg:

```
/opt/bonita/server/bin/run.sh -b 0.0.0.0 &
```

A *-b* (bind) kapcsoló azt mondja meg a JBOSS-nak, hogy minden hálózati kártyára figyeljen, ezzel más gépekről is elérhetővé válik a szerver. Az első induláskor a *bonitaDS* datasource mögötti sémában automatikusan létrejön minden tábla, index és egyéb objektum, amire a Bonitának szüksége van. Miután elindult a JBOSS, írjuk be a böngészőbe a következő sort:

```
http://<SERVERADDRESS>:8080/bonita
```

Amennyiben eddig mindent jól csináltunk, úgy be kell jönnie a portál login ablakának, ahova egyelőre csak egy *user/password=install/install* technikai account-tal tudunk belépni, így ezzel lépjünk be. Amennyiben ezt a technikai usert szeretnénk megváltoztatni, úgy *conf\bonita_server.properties* fájlba ezt is megtehetjük. Az utolsó lépés, hogy hozzunk létre egy adminisztrátort, aki a portál első ilyen felhasználója lesz. Az új felhasználó létrehozását a Bonita Portálról írt fejezet részletesen leírja, ezért itt most csak azt javasoljuk, hogy ez az első user legyen *admin* nevű és tegyük be az *Administrators* profile-ba.

## A Bonita portál testreszabása

A most következő részben néhány nem kötelező, de hasznos Bonita Portál konfigurációs lehetőséget mutatunk be. Előtte állítsuk le a JBOSS szervert, ha futna.

### Nyelvi fájlok telepítése

A korábbi cikkekben említettük, hogy a Bonita Portál *GNU gettext po* fájlokat használ az egyes nyelvekhez. Amennyiben nincs megfelelő nyelv (például a magyar), úgy először azt *poeditor*-ral el kell készítenünk, majd ide másolni őket:

```
JBOSS_HOME/bonita/client/platform/work/i18n
```

Ezzel a lépéssel a Bonita Portál automatikusan felismeri az új nyelvet és arra át is tudjuk kapcsolni. Az általunk használt új po fájlok ezek voltak: *portal-sp_hu*, *portal_hu.po*, *mobile_hu.po*.

### A portál default CSS layout módosítása

A *JBOSS_HOME/server/default/deploy/bonita-all-in-one-6.1.2.ear* fájl tartalmazza a Bonita workflow engine-t és a Bonita Portál web alkalmazást (*bonita.war*). A *{bonita.war}/portal/css* tartalmazza ezeket a CSS fájlokat:

- *bonita.css* (például több hely legyen a portál középső részén a jobb oldal terhére)
- *bonita_forms.css* (a formok megjelenítő keretére hat)

Ezekkel lehet testre szabni a portál kinézetét.

### A portálon látható logo cseréje

A *{bonita.war}/portal/css/skins/default/images* könyvtárban ezeket a képfájlokat lehet felülírni:

- *logo.png*
- *login-logo.png*

Ekkor az adott szervezetre illeszkedő logo jelenik meg a Bonita alapértelmezett képek helyett.

### A portálon látható címsor cseréje

Az alábbi mindkét könyvtárban megjelenik a *BonitaConsole.html* fájl:

- *JBOSS_HOME/bonita/client/platform/tenant-template/work/looknfeel/default*
- *JBOSS_HOME/bonita/client/tenants/1/work/looknfeel/default*

A *Bonita BPM Portal* címsor szöveget itt lehet beállítani a kívánt értékre, például *Céges munkafolyamatok*.

### A *BonitaForm.html* esetleges módosítása

A *{bonita.war}/portal* alatt található, a fix szövegek vagy egyéb részek itt is cserélhetőek.

### A *login.jsp* esetleges módosítása

A *{bonita.war}* gyökérben található, a fix szövegek vagy egyéb részek itt is cserélhetőek.

## A *bonita-all-in-one-6.1.2.ear* további átalakításai

Ez a lépés is opcionális, de a Bonitasoft szakemberei javasolják.

### A jar fájlok egy helyre másolása az EAR fájlon belül

Ez a lépés azt valósítja meg, hogy minden *jar* fájlt másoljunk az *EAR lib* alá:

```
unzip -d bonita bonita.war
mv bonita/WEB-INF/lib/* lib/
```

### Az slf4j jar fájlok törlése az EAR lib folder alól

```
rm lib/antlr-2.7.6.jar
rm lib/slf4j-api-1.6.1.jar
rm lib/slf4j-log4j12-1.6.1.jar
```

Ezután csomagoljuk össze újra az *ear* fájlt.

## A nem alkalmazás specifikus további *jar* fájlok telepítése

Amennyiben közhasznú Java könyvtáraink is vannak, amiket mások és/vagy mi fejlesztettünk, úgy azokat ide kell másolni:

```
JBOSS_HOME/server/default/lib
```

Erre automatikusan beállítódik a szerver szintű *CLASSPATH*. Ugyaninnen a HSQLDB (H2) specifikus jar fájlokat érdemes visszamozgatni:

```
rm bonita-jboss-h2-mbean-1.0.0.jar bonita-tomcat-h2-listener-1.0.1.jar h2-1.3.170.jar
```

## A naplózás beállítása

A JBOSS naplózás a LOG4J rendszert használja, amit itt lehet konfigurálni:

```
JBOSS_HOME/server/default/conf/jboss-log4j.xml
```

Például egy új *category* felvétele:

```
<category name="org.cs.myworkflow">
   <priority value="TRACE"/>
</category>
```

A LOG4J konfigurációs lehetőségeket (például a lehetséges *priority* értékek) innen lehet megtudni: `http://logging.apache.org/`. Minden process alkalmazás számára egy külön *category* felvétele javasolt. Az összes üzenetet a szerver naplóban meg lehet találni:

```
JBOSS_HOME/server/default/log/server.log
```

## A JVM memória konfigurációja

A *JBOSS_HOME/bin/run.conf* fájlban állítsuk be a JVM által használt memóriát:

- *Xms2048m* → A kezdeti memória mérete, amit a JVM használ
- *Xmx2048m* → A JVM maximum ekkora memóriát használhat
- *XX:MaxPermSize=512m* → A heap maximális mérete.

## Az *LDAP synchonizer* konfigurálása és használata

### Az *LDAP synchonizer* feladata

Az *LDAP synchonizer* feladata az, hogy a vállalati címtárban (például MS AD) tárolt felhasználókat és csoportokat folyamatosan frissítse a Bonita user adatbázisba is. Tekintettel arra, hogy ez az eszköz több AD domain-ra is ráállítható, így ennek egy centralizált user adatbázis fenntartása lesz az eredménye. Ezt alapvetően kétféleképpen tudjuk használni:

1. Az authentication (hitelesítés) és authorization (jogosultságkezelés) is a Bonita saját, ily módon felépített és karbantartott user adatbázisára épül. Ez esetben az *LDAP synchonizer* a jelszó helyére is a user nevet generálja, de az természetesen átírható.

2. Az authentication az AD-val szemben, míg az authorization a Bonita saját adatbázisával történik.

Ahol van AD (vagy valamilyen más LDAP címtár), ott a 2. forgatókönyvet részesítik előnyben. Azt is kétféleképpen lehet használni:

1. A login ablak mindig kikerül és a megadott user/password az LDAP címtárral szemben lesz érvényesítve

2. Amennyiben van valamilyen SSO megoldás, akkor azt a Bonita át tudja venni. AD esetén a SPNEGO használható, amiről részletesen az Informatikai Navigátor 6. számában írtunk.

A működési modell egyszerű. Valamilyen módon történik egy hitelesítés, ami után a username lesz az, amire a Bonita authorization-t használjuk.

### Az *LDAP synchonizer* telepítése és konfigurálása

Az *LDAP synchonizer* a *BonitaBPMSubscription6.1.2deploy.zip* fájl része, aminek a tartalmát csomagoljuk ki egy erre a célra szolgáló mappába. Itt fogunk találni egy *BonitaBPMSubscription-6.1.2LDAPSynchronizer* almappát, ez tartalmazza ezt az eszközt. A könnyebb hivatkozás kedvéért ezt a mappát *LDS_HOME*-nak fogjuk nevezni a továbbiakban. A konfigurálás abból áll, hogy az *LDS_HOME/conf/default* könyvtárban található 5 darab properties fájlt kell beállítani a helyi viszonyoknak megfelelően. Az elsőt a 12-1. Programlista mutatja, az a feladata, hogy specifikálja a Bonita Home helyét és a használható user/password-öt. Ahogy a konfigurációból is kitűnik, érdemes azt a Bonita Home-ot használni, amit belecsomagoltak a *zip* fájlba. Szeretnénk kiemelni, hogy a *HTTP API*-t kell használni, azaz a *bonita-client.properties* fájlba ez legyen beállítva (a részleteket lásd a 7. fejezetben, ahol bemutattuk a Bonita Home kialakítását is).

12-1. Programlista: bonita.properties - Bonita Home és a login információ megadása

```
######
# Bonita connection settings
######
bonita_home = /opt/bonita/unzip_files/BonitaBPMSubscription-6.1.2-deploy/bonita_home-6.1.2
######
# Bonita account used for sync (needs admin priviledges)
######
login             = install
password          = install
technicalUser     = platformAdmin
technicalPassword = platform
```

A második konfigurációs fájlt a 12-2. Programlista tartalmazza. Az AD elérésére vonatkozó adatokat fogja össze, aki ismeri az LDAP-ot, annak itt nincs semmi újdonság.

12-2. Programlista: ldap.properties - Az AD (LDAP) elérés konfigurációja

```
######
# LDAP connection settings
######
host_url                =       ldap://windowsdc01.ceg.sys.corp:389/
auth_type               =       simple

######
# LDAP account used for browsing
######
principal_dn            =       _srvBonita
principal_password      =       titok

######
# User type ('person' for LDAP, 'user' for AD)
######
directory_user_type = user

######
# Paged search
# Not supported by all LDAP servers
######
use_paged_search = true
page_size = 1000
```

A harmadik konfigurációs fájlt a 12-3. Programlista mutatja. A használata azért fontos, mert itt adjuk meg a futási naplózás beállításait, amely napló meglétének fontosságát talán nem kell hangsúlyoznunk.

12-3. Programlista: logger.properties - A naplózás konfigurációja

```
######
# LOGGER CONFIGURATION
# Provides the settings for the logger
# Default settings should be fine for most uses
# Relevant values for the logger level are:
#   INFO - for production use
#   FINE - for debug use
######

log_dir_path          = logs/
log_file_date_prefix  = yyyy-MM-dd
log_level             = INFO
```

A negyedik konfigurációs fájl (12-4. Programlista) egy adat mapping az LDAP (AD) címtár és a Bonita user adatbázis mezői között. Amik itt meg vannak adva, azok az adatelemek fognak szinkronizálódni a címtárból a Bonitába. Az értékadás bal oldalán a Bonita, míg a jobb oldalán az LDAP mezőnév található.

12-4. Programlista: mapper.properties - Az AD és a Bonita user adatbázis mezők összerendelése

```
######
# MAPPER CONFIGURATION
# Provides the field mapping between Bonita to LDAP such as:
# bonita_poperty = ldap_property
#
# user_name is the only mandatory property as it is the key defined for matching users, all ➥
    other properties are optionals.
# Unused properties should be commented out.
######
```

```
9
10 ######
11 # GENERAL INFORMATIONS
12 ######
13 user_name                =       sAMAccountName
14 first_name                =       givenName
15 last_name                 =       sn
16 #title                    =
17 job_title                 =       displayName
18 #manager                  =       manager
19 #delegee                  =
20
21 ######
22 # PROFESSIONAL INFORMATIONS
23 ######
24 pro_email                =       mail
25 pro_phone                =       telephoneNumber
26 pro_mobile               =       mobile
27 #pro_fax                  =       facsimileTelephoneNumber
28 #pro_website    =
29 #pro_room                 =
30 #pro_building   =
31 #pro_address              =       street
32 #pro_city                 =       city
33 #pro_zip_code   =       postalCode
34 #pro_state                =       state
35 #pro_country              =       country
36
37 ######
38 # PERSONNAL INFORMATIONS
39 ######
40 #perso_email    =
41 #perso_phone              =       homePhone
42 #perso_mobile   =
43 #perso_fax                =
44 #perso_website  =
45 #perso_room               =
46 #perso_building =
47 #perso_address  =
48 #perso_city               =
49 #perso_zip_code =
50 #perso_state    =
51 #perso_country  =
```

Végül az ötödik (12-5. Programlista) konfigurációs egység azt mondja meg, hogy a címtárból mely tételek (rekordok) szinkronizálódjanak át. Ez alapvetően egy-egy szűrési feltétel megadását jelenti, ahogy láthatjuk is. Itt a user-ek és a group-ok szűrését is megadhatjuk, ugyanis a felhasználók mellett az LDAP csoportok átvitelét is támogatja az eszköz.

12-5. Programlista: sync.properties - A szinkronizálandó rekordok filterének beállítása

```
1 ######
2 # SYNCHRONIZATION CONFIGURATION
3 # Provides the settings for the synchronization between Bonita and LDAP
4 # See also mapper.conf
5 ######
6
7 ######
8 # ERROR BEHAVIOR SETTINGS
9 # Defines the synchronization error behavior settings
10 ######
11
12 # Specifies whether an error should be blocking upon getting related users (manager)
```

```
13 error_level_upon_failing_to_get_related_user = warn
14
15 # AD domain name
16 ad_domain_name = MYDOM
17
18
19 ######
20 # LDAP SYNC SEARCH SETTINGS
21 # Defines the LDAP watched directory
22 ######
23
24 # Declare a list of LDAP watched directories
25 ldap_watched_directories = dir1
26
27 # Specify dir1 settings
28 dir1.ldap_search_dn      =       OU=MYDOMUsers,DC=ceg,DC=sys,DC=corp
29 dir1.ldap_search_filter =        cn=*
30
31 # Specify dir2 settings
32 #dir2.ldap_search_dn     =       ou=OtherPeople,dc=bonita,dc=com
33 #dir2.ldap_search_filter         =       cn=*
34
35
36 ######
37 # BONITA USER SYNC SETTINGS
38 ######
39
40 # Specifies the username case of the Bonita imported users
41 bonita_username_case =  lowercase
42
43 # Specify Bonita users who should not be synchronized (user names separated by commas)
44 #bonita_nosync_users     =       admin,john,james,jack
45
46 # Specifies whether the tool should remove Bonita users which are not present in LDAP
47 bonita_remove_users =   false
48
49 # Specify the role that will be affected to Bonita users
50 bonita_user_role = user
51
52
53 ######
54 # LDAP GROUP SYNC SETTINGS
55 # Defines the LDAP groups that are synchronized
56 ######
57
58 # Specifies whether recursive groups (sub groups) should also be synchronized
59 allow_recursive_groups =        false
60
61 # List of groups to synchronize
62 ldap_groups = group1,group2,group3
63
64 # Specify group1 settings
65 group1.ldap_group_dn                    =       CN=Bonita_staff,OU=MYDOMGroups,DC=ceg,DC=sys,DC=➥
       corp
66 #group1.forced_bonita_group_name        =       forced test
67
68 # Specify group2 settings
69 group2.ldap_group_dn                    =       CN=App-Buyer,OU=Bonita,OU=MYDOMGroups,DC=ceg,DC=➥
       sys,DC=corp
70
71 group3.ldap_group_dn                    =       CN=PortalAdministrator,OU=Bonita,OU=MYDOMGroups,➥
       DC=ceg,DC=sys,DC=corp
```

Amennyiben több domain is van, úgy annyi *LDAP synchonizer* alkönyvtárat alakítsunk ki, mindegyikhez külön bekonfigurálva ezt az 5 darab fájlt.

### Az *LDAP synchonizer* futtatása

A szinkronizáló eszközünk tartalmazza a 12-6. Programlista scriptjét, így azt kell lefuttatni és megtörténik az LDAP→Bonita user adatbázis szinkronizáció.

12-6. Programlista: BonitaBPMSubscription6.1.2LDAPSynchronizer.sh)

```
#!/bin/sh
SCRIPTPATH=$( cd $(dirname $0) ; pwd )
cd $SCRIPTPATH
for jar in $(ls $SCRIPTPATH/lib/*.jar); do
CLASSPATH=$CLASSPATH:${jar}
done
java classpath $CLASSPATH com.bonitasoft.ldapsynchronizer.LDAPSynchronizer $1
```

Természetesen éles környezetben nem kézzel akarjuk futtatni ezt az eszközt, ezért a script meghívását a *crontab*-ra bízhatjuk. Abban az esetben, ha több domain-t is bekonfiguráltunk, úgy annyi bejegyzés tegyünk, hiszen meghívandó script is annyi lesz. Itt érdemes gondoskodni arról, hogy egyszerre csak egy *LDAP synchonizer* script fusson.

## A hitelesítés beállítása

Opcionálisan eltérhetünk a Bonita alapértelmezett login manager-étől, amit ekkor el kell készítenünk és be kell konfigurálnunk. Az elkészítésére a következő fejezetben mutatunk példát, a konfigurálását a *BONITA_HOME/client/platform/conf* könyvtár *loginManager-config.properties* fájlban tehetjük meg, átírva ezt az alapértelmezett implementációs osztály nevét a miénkre:

```
login.LoginManager = org.bonitasoft.console.common.server.login.impl.standard.➥
    StandardLoginManagerImplExt
```

## A JBOSS elindítása és leállítása

Éles üzemben a JBOSS így indítandó:

```
/opt/bonita/BonitaBPMSubscription6.1.2 JBoss5.1.0.GA/bin/run.sh b 0.0.0.0 &
```

Leállítani pedig így célszerű:

```
/opt/bonita/BonitaBPMSubscription6.1.2 JBoss5.1.0.GA/bin/shutdown.sh s jnp://0.0.0.0:1099
```

Fejlesztői környezetben szükség lehet a processor core-ok korlátozására, így például a 2 core-os indítást így tehetjük meg a *taskset* paranccsal:

```
taskset c 0,1 /opt/bonita/BonitaBPMSubscription6.1.2 JBoss5.1.0.GA/bin/run.sh b 0.0.0.0 &
```

## A JBOSS HTTPS bekonfigurálása

### A privát kulcs generálása és a tanúsítvány megszerzése

Tegyük fel, hogy a szerver bejegyzett DNS neve *bonita.ceg.hu*. Fontos a név megléte, hiszen a *HTTPS* kapcsolat éppen a szerver nevének hitelességét vizsgálja. Másfelől az is lényeges, hogy az *nslookup* parancs a gép IP címére ugyanezt a DNS nevet adja vissza, amit így tudunk leellenőrizni:

```
nslookup <IP Cím>
```

A kulcsok és tanúsítványok előállításához az *openssl* parancsot fogjuk használni. Az első lépés a privát kulcs legenerálása lesz:

```
openssl genrsa -des3 -rand file1:file2:file3:file4:file5 -out bonita.ceg.hu-server.key 1024
```

Itt megadtunk egy jelszót, ami a kulcshasználatot védi, esetünkbe ez *Titok123* lett. A következő lépés a tanúsítvány kérelem *csr* fájl előállítása:

```
1  openssl req -new -key bonita.ceg.hu-server.key -out bonita.ceg.hu-server.csr
2
3  Enter pass phrase for eform-server.key:
4  You are about to be asked to enter information that will be incorporated into your certificate ➥
       request.
5  What you are about to enter is what is called a Distinguished Name or a DN.
6  There are quite a few fields but you can leave some blank For some fields there will be a ➥
       default value, If you enter '.', the field will be left blank.
7
8  Country Name (2 letter code) [AU]:HU
9  State or Province Name (full name) [Some-State]:Hungary
10 Locality Name (eg, city) []:Budapest
11 Organization Name (eg, company) [Internet Widgits Pty Ltd]:A cég neve
12 Unit Name (eg, section) []:A cég IT szervezeti egységének a neve
13 Common Name (e.g. server FQDN or YOUR name) []:bonita.ceg.hu
14 Email Address []:bonita@ceg.hu
15 Please enter the following 'extra' attributes to be sent with your certificate request A ➥
       challenge password []:bonita
16 An optional company name []:bonita
```

Az 1. sorban a *csr* kérelem fájl előállításához szükséges parancsot látjuk, ami a 3. sorban látható módon egyből bekéri a privát kulcs jelszavát, megadtuk a *Titok123* jelszót. Itt jegyezzük meg, hogy egy privát kulcshoz tetszőleges számú nyilvános kulcsot tudunk generálni, egy ilyen meg is történik, ugyanis a certificate-nek tartalmaznia kell egy public key-t. A következőekben a 8-16 sorok között válaszoljunk meg azokra a kérdésekre, amelyekre adott válaszokat fogja majd a kiadott certificate tanúsítani. Például a 13. sorban található *bonita.ceg.hu* válasz most azt jelenti, hogy ez lesz a szerver gép neve. A keletkezett *bonita.ceg.hu-server.csr* tanúsítvány kérelem fájlt juttassuk el a tanúsítványt kibocsátani képes hatósághoz. Ez lehet a cégen belül is, amennyiben ott létezik egy PKI környezet. Ők elkészítik ebből a tanúsítványt és visszaküldenek egy *cer* (certificate) fájlt, például ilyen névvel: *bonita.ceg.hu-server.cer*. Ezzel készen vagyunk arra, hogy a JBOSS HTTPS kapcsolat konfigurálását elkezdjük.

### A Java keystore elkészítése

Amennyiben a kapott tanúsítvány *DER* formátumú (az esetünkben azt kaptunk), úgy abból készítsünk egy *PEM* formátumú változatot is, ugyanis a következő lépésben majd *PKCS#12* keystore fájlt szeretnénk készíteni:

```
openssl x509 -inform der -in bonita.ceg.hu-server.cer -out bonita.ceg.hu-server.pem
```

És a *PKCS#12* keystore fájl elkészítése ezzel a paranccsal lehetséges:

```
openssl pkcs12 -export -in bonita.ceg.hu-server.pem -inkey bonita.ceg.hu-server.key -out bonita-➥
    keystore.p12
```

Látható, hogy a *bonita-keystore.p12* fájlba a privát kulcsot és a tanúsítványt is beimportáltuk. Ezt a fájlt is jelszó védi, itt adjuk meg ugyanazt, mint ami a privát kulcsé volt (*Titok123*). A következő lépésben a *p12* fájlt egy Java keystore-ba fogjuk importálni:

```
keytool -importkeystore -destkeystore bonita-server.jks -srckeystore bonita-keystore.p12 -➥
    srcstoretype pkcs12 -alias 1
```

A fenti parancs létrehoz egy *bonita-server.jks* nevű Java keystore-t, aminek a jelszava itt is a privát kulcs jelszava legyen. Az most beimportált tartalomra érdemes egy beszédesebb alias nevet (*jbosscert* lesz) adni ezzel a paranccsal:

```
keytool -changealias -alias 1 -destalias jbosscert -keypass Titok123 -keystore bonita-server.jks
```

A létrehozott *jks* fájlt másoljuk egy biztonságos könyvtárba.

### A JBOSS HTTPS kapcsolat bekonfigurálása

A HTTPS protokoll bekonfigurálását ebben a fájlban tudjuk elvégezni:

```
JBOSS_HOME/server/default/deploy/jbossweb.sar/server.xml
```

Keressük meg ezt a megjegyzésbe rakott rész és aktiváljuk ezzel a tartalommal:

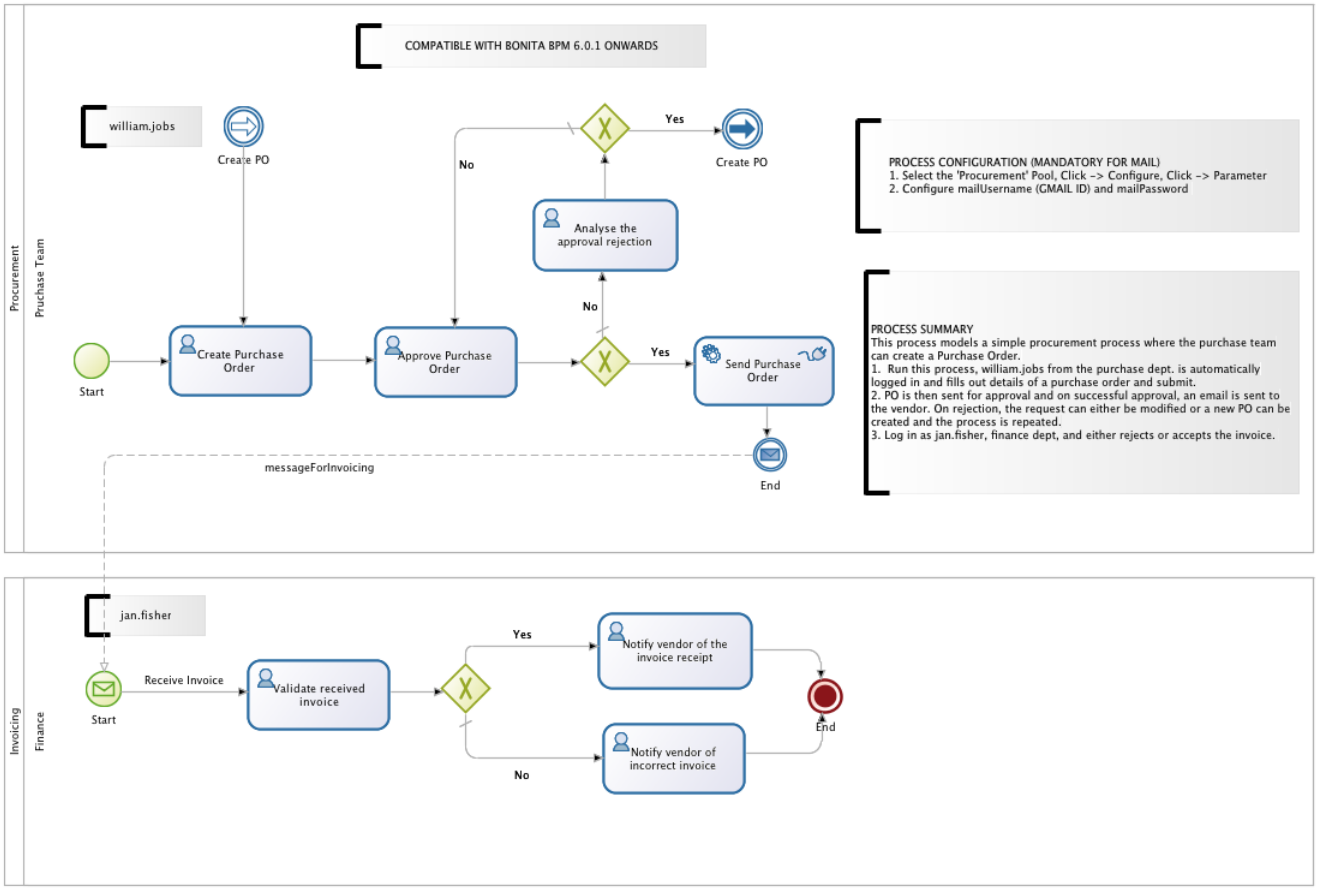
```
<Connector protocol="HTTP/1.1" SSLEnabled="true"
port="8443" address="${jboss.bind.address}"
scheme="https" secure="true" clientAuth="false"
keystoreFile="${jboss.server.home.dir}/conf/bonita-server.jks"
keystorePass="kspassword" sslProtocol = "TLS" />
```

A HTTP kapcsolatot meghagyhatjuk, illetve a HTTPS portját akár 80-ra is vehetjük. Ezzel elkészültünk a JBOSS HTTPS konfigurálással.

# 13. A Bonita Business Activity Monitoring

A BAM (Business Activity Monitoring) egy olyan eszköz, amivel mérni tudjuk a folyamataink működését, erről riportokat kérhetünk le. Mindez lehetővé teszi, hogy a munkafolyamataink korábban definiált mérési szempontjait (KPI-ok) kiértékeljük. Ez a lehetőség természetesen azért van, hogy az üzleti folyamataink eredményességét időnként tudjuk tökéletesíteni, a bennük lévő hiányosságokon javíthassunk.

Az alábbiakban közölt BAM ismertető a Bonita dokumentációban található példa alapján készült. A Bonita BAM megoldás lehetővé teszi, hogy egy saját riportot telepítsünk, amely képes megjeleníteni a méréseket.




13.1. ábra. A BAM használatát ezen a workflow terven nézzük meg

Ezen fejezetből megtanulható, hogy miképpen tudunk új KPI-okat létrehozni és használni azokat egy olyan Jasper Reports riporttal, amit magunk készítettünk el. Itt további különlegesség az is, hogy ezeket a riportokat úgy fogjuk a Bonitába integrálni, mintha azok gyári megoldások lennének. A példához használt munkafolyamatot a 13.1. ábra mutatja.

## Egy új Jasper Reports riport integrálása a Bonita Portálra

A feladatot a következő 3 lépés oldja meg:

1. Előkészíteni az adatbázist arra, hogy abban olyan adatok is tárolódjanak, ahogy azt a riportkészítő eszköz fel tudja dolgozni.

2. Létrehozni *iReport* vagy más eszközzel a Jasper Reports *jrxml* fájlt.

3. Az elkészült riport telepítése abból áll, hogy a portál segítségével, adminisztrátor módban feltöltjük azt. Ezt az *Analytics* → *Reports* fülön az *INSTALL REPORT* zöld gomb segítségével tudjuk kezdeményezni.

## Az adatok előkészítése

A riportoknak valamilyen adatforrásra van szükségük, azaz egy táblára (általánosabban SQL selectre több tábla fölött is). A Bonita engine saját célra eltárolja a munkafolyamatok fontos adatait, ezért elvben erre is épülhetnek a riportok, amit a Bonita egy külön beépített *Domain Specific Language to query* eszközzel valóban támogat is. Természetesen direkt módon is előállíthatjuk a riport alapját képező táblákat. Összefoglalva 2 módszer van az adatforrás kialakítására:

1. A *Domain Specific Language to query* eszköz használata, ami a Bonita saját tábláit használja. Ez nagy előny, hiszen az adatok egyből elérhetőek, de a teljesítményben egy kis hátránya van.

2. Egy külön adatbázis séma/tábla használata (13.2. ábra), ami csak a riport miatt létezik.

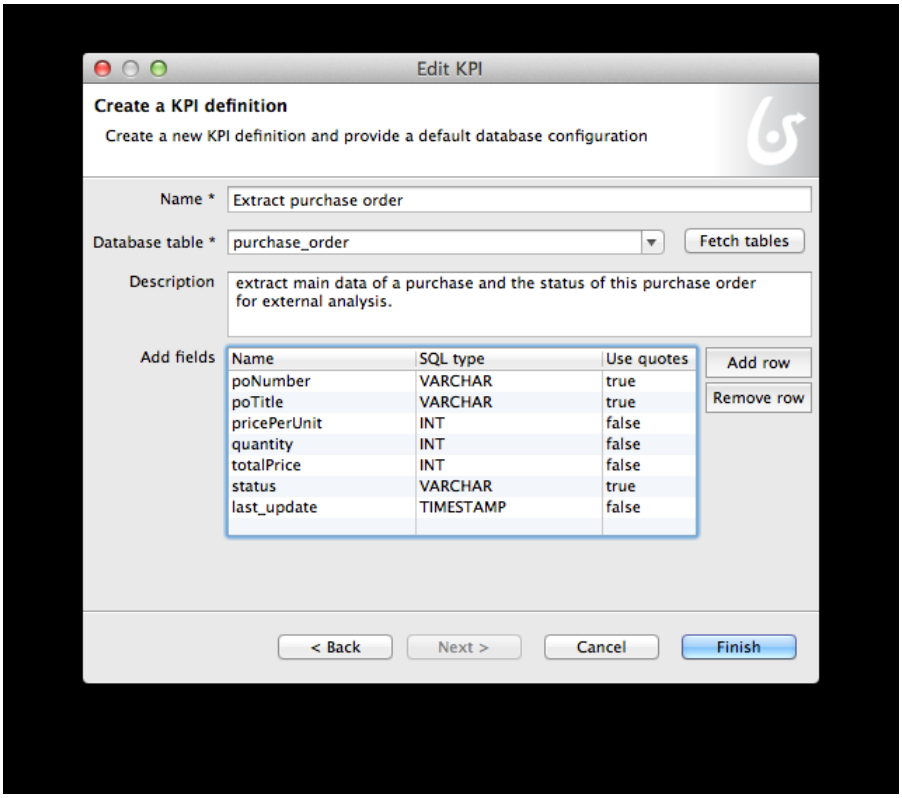
```
CREATE TABLE `BonitaReport`.`purchase_order` (
    `poNumber` VARCHAR( 20 ) NOT NULL ,
    `poTitle` VARCHAR( 20 ) NOT NULL ,
    `pricePerUnit` INT NOT NULL ,
    `quantity` INT NOT NULL ,
    `totalPrice` INT NOT NULL ,
    `status` VARCHAR( 10 ) NOT NULL ,
    `last_update` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

13.2. ábra. Egy külön adatbázis séma/tábla használata

Itt fontos megérteni, hogy mit jelent a *KPI* a Bonitában. A *KPI* egy technikai fogalom, ami arra utal, hogy a process adataiból valamilyen részeket, valamely helyről kiszedünk és eltároljuk azt egy erre a célra kitalált adatbázis táblában.

## Egy új KPI létrehozása

Válasszuk ki a *KPI* → *Create KPI...* (vagy *Edit KPI...*) menüpontot, adjuk meg a használni kívánt adatbázis elérési jellemzőit (típus, host, port, név, user, password), majd nyomjuk meg a *Next* gombot, amire a 13.3. ábra ablaka jön be. Az ábrán már a kitöltött állapotot láthatjuk. A *Fetch tables* és az *Add row* gombok segítenek a konfigurálásban. Előtte még fontos lépés volt, hogy a KPI-nak nevet is adtunk, esetünkben ez lett: *Extract purchase order*.

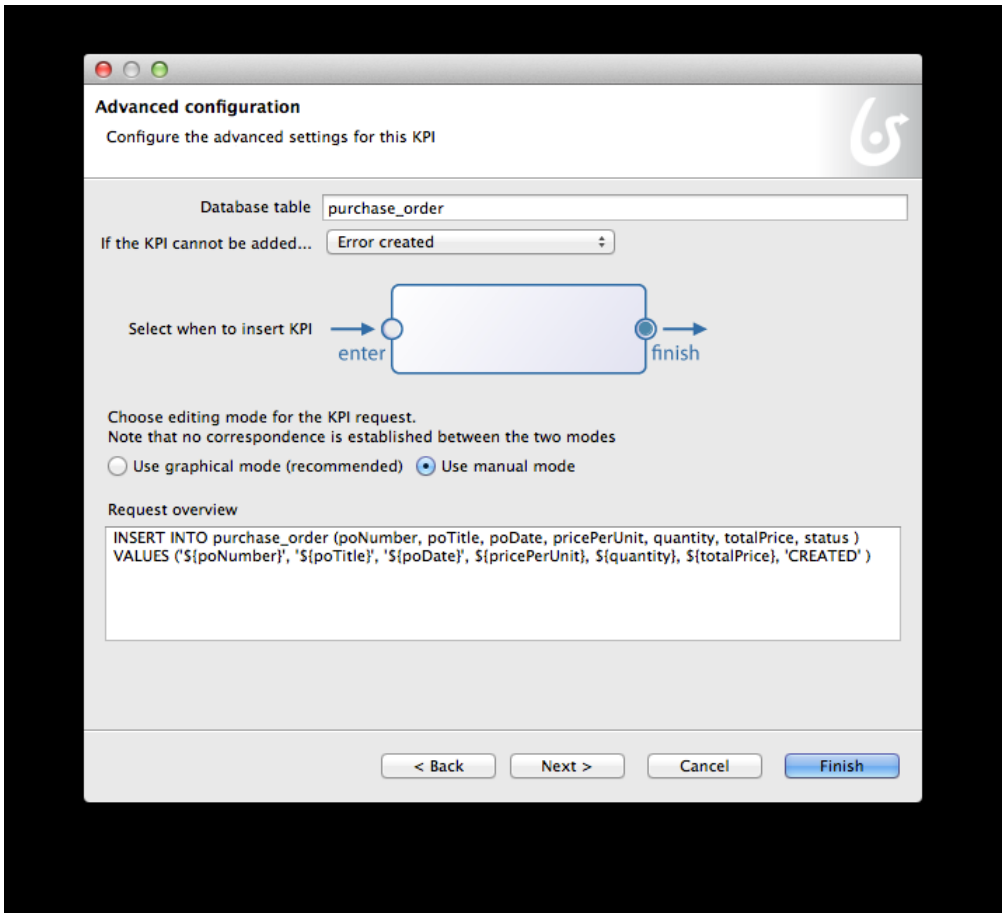



13.3. ábra. Egy új KPI definíció létrehozása

## A létrehozott KPI használata

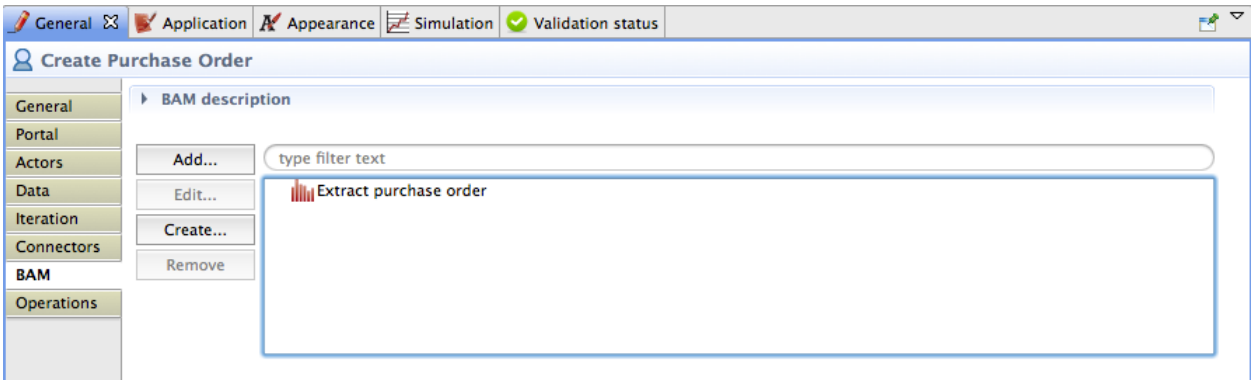
A továbbiakban azt mutatjuk meg, hogy a most létrehozott KPI-t miképpen rendeljük hozzá a *Create Purchase Order* step-hez (13.1. ábra). Válasszuk ki a step-pet, majd a *General* → *BAM* fület és nyomjuk meg az *Add* gombot. Ezután választanunk kell egy már definiált KPI-t, mi most az előbb létrehozott *Extract purchase order* nevűt válasszuk ki és nyomjunk *Next* gombot. Válasszuk ki a *finish* eseményt és adjuk meg azt az INSERT SQL parancsot (13.4. ábra):

```
INSERT INTO purchase_order (poNumber, poTitle, pricePerUnit, quantity, totalPrice, status ) ➥
    VALUES ('${poNumber}', '${poTitle}', ${pricePerUnit}, ${quantity}, ${totalPrice}, 'CREATED' ➥
    )
```




13.4. ábra. A KPI milyen eseményre aggregálódjon

A *Finish* gombra kattintva látjuk a BAM fülön, hogy erre a step-re immár rendelkezünk egy BAM konfigurációval (13.5. ábra).




13.5. ábra. Egy step-hez bekonfigurált PKI

## További 2 új KPI létrehozása

### *Analyse the approval system* step

Az enter event-re ezzel az SQL-lel hozzuk létre:

```
INSERT INTO purchase_order (poNumber, poTitle, pricePerUnit, quantity, totalPrice, status ) ➥
    VALUES ('${poNumber}', '${poTitle}', ${pricePerUnit}, ${quantity}, ${totalPrice}, 'REJECTED'➥
     )
```

### *Send purchase order* step

Az enter event-re ezzel az SQL-lel hozzuk létre:

```
INSERT INTO purchase_order (poNumber, poTitle, pricePerUnit, quantity, totalPrice, status ) ➥
    VALUES ('${poNumber}', '${poTitle}', ${pricePerUnit}, ${quantity}, ${totalPrice}, 'APPROVED'➥
     )
```

## JDBC driver hozzáadás

A *Configure* → *Process Dependencies* menüpontnál adjuk hozzá azt a JDBC jar-t, amilyen adatbázist használtunk a KPI táblák számára. Futtassunk le néhány process példányt és látni fogjuk, ahogy a rekordok gyűlnek a *purchase_ order* táblában (13.6. ábra)

| | poNumber | poTitle | pricePerUnit | quantity | totalPrice | status | last_update |
|---|---|---|---|---|---|---|---|
| | PO6785 | order 1 | 50 | 50 | 2500 | CREATED | 2013-11-07 17:13:31 |
| | PO11456 | order 2 | 5 | 10 | 50 | CREATED | 2013-11-07 17:14:09 |
| | PO6785 | order 3 | 10 | 300 | 3000 | CREATED | 2013-11-07 17:14:38 |
| | PO6785 | order 4 | 5 | 47 | 235 | CREATED | 2013-11-07 17:15:06 |
| | PO11456 | order 5 | 30 | 46 | 1380 | CREATED | 2013-11-07 17:15:52 |
| | PO11456 | order 6 | 30 | 53 | 1590 | CREATED | 2013-11-07 17:16:22 |

13.6. ábra. Néhány process instance után a *purchase_ order* tábla

## A Jasper Reports konfiguráció

Akit a Jasper Reports mélyebben érdekel, annak ajánljuk az Informatikai Navigátor 8. számát. A Jasper Reports motor része a Bonita motornak, sőt ezek a riportok már előre, gyárilag is elérhetőek a portálon keresztül:

- Case average time
- Case list
- Case history
- Task list

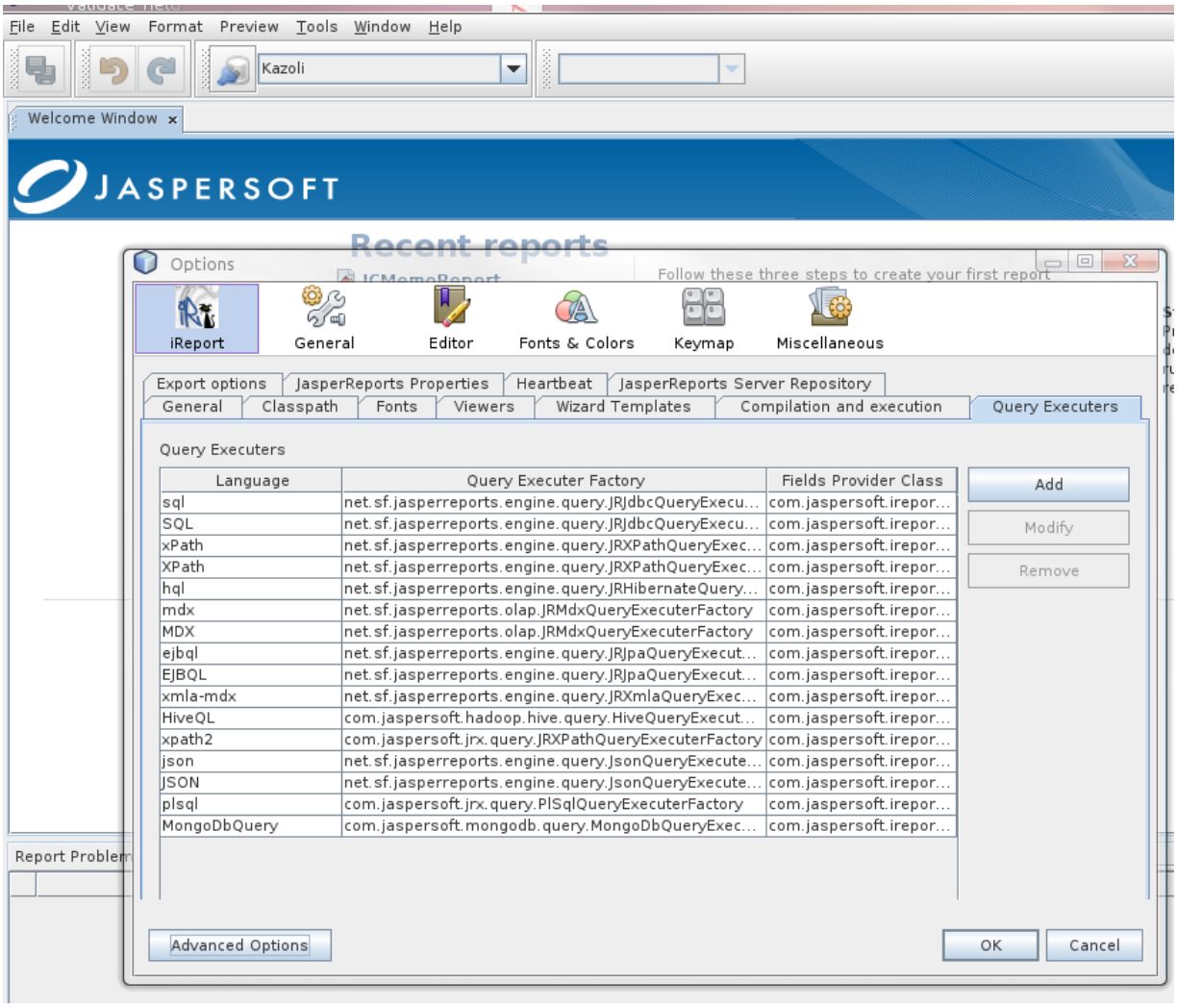
A saját *jasper* fájlunkat (ez a lefordított riport fájl) és a hozzá tartozó erőforrásokat egy *zip* fájlba tehetjük és a fentebb leírt módon a portál segítségével feltölthetjük, ezzel ez a riport is rendelkezésre fog állni. Tekintettel arra, hogy esetünkben *purchase_order* tábla szolgáltatja az adatforrást a riport részére, ezért ez egy szokásos használati módnak tekinthető. A következőkben nézzük meg a *Domain Specific Language to query* eszközt.

## BonitaSQL QueryExecuter

Az iReport és a Bonita adatbázis összeköthető egymással, ugyanis a Bonita biztosítja ezt az osztályt:

```
org.bonitasoft.JRBonitaJdbcQueryExecuterFactory
```

Ennek a konfigurációs helyét mutatja a 13.7. ábra *Query Executers* füle.




13.7. ábra. A BonitaSQL QueryExecuter beállítása az iReport-ban

Amikor egy új *Query Executer*-t konfigurálunk be Bonitához, ezeket az adatokat adjuk meg:

- Language: *BONITASQL*
- Factory class: *org.bonitasoft.JRBonitaJdbcQueryExecuterFactory*
- Fields Provider class (optional): *com.jaspersoft.ireport.designer.data.fieldsproviders.SQLFieldsProvider*

Az iReport CLASSPATH-hoz adjuk hozzá ezt a jar fájlt: *console-reporting-XXX.jar* (esetünkben: *console-reporting-6.1.2.jar*).

## A riport készítéséhez szükséges tábla elérése

Amikor a riportot telepítjük a Bonita engine-re, akkor szükséges, hogy ez tartalmazza az adatbázis elérési információt is, amit a *connection.properties* fájlba kell megadnunk és hozzácsomagolni a report archive-hoz:

- *dbURL*: A JDBC URL, ahol a BAM tábla van
- *dbDriverClassName*: A JDBC driver class neve
- *dbUser*: username
- *dbPassword*: password

Mindig készítsünk iReport datasource-t is, mert ez teszi lehetővé, hogy használhassuk a *query designer* és *auto read fields* lehetőségeket.

## Paraméter form hozzárendelése a riporthoz

Amikor kipróbáljuk a már beépített (jelenleg 4 darab) riportot, akkor láthatjuk, hogy azok egy FORM segítségével paramétereket is kérnek. A mi riportunk számára is rendelkezésre áll ez a lehetőség, azaz szűrő képernyőt készíthetünk a riport elé, hiszen általában mi szeretnénk szabályozni azt, hogy mi kerüljön be a jelentésbe. Ehhez a következő lépéseket kell tennünk.

**A riport kiegészítése egy Html komponenssel** Egy Jasper Reports *Html* komponenst helyezzünk el az éppen fejlesztés alatt lévő riportunkra. Mindezt természetesen az iReport-ban kell elvégeznünk. Ezen komponens property lapján keressük meg a *HTML Content Exp.* jellemzőt és állítsuk be erre az értékre: *$P{BONITA_HTML_FORM}*. Ez azt jelenti, hogy a riport ezzel a paraméterrel lesz meghívva, ami egyben a most feltett *Html* komponens tartalma is lesz.

**A filter form widget-ek kialakítása** A *jrxml* fájlba *property*-k is elhelyezhetőek, ezekkel fogjuk leírni ezeket a widget-eket. Minden widget-et 3 property ír le, ahol X=1, 2, ..., azaz a widget szám alapú egyedi azonosítója.

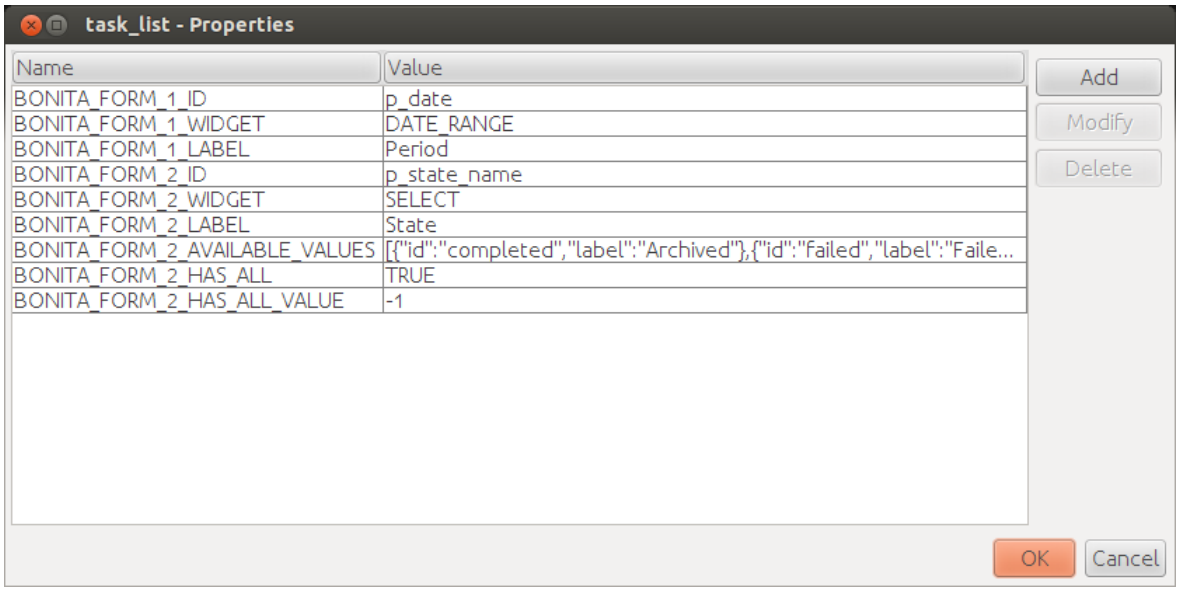
- *BONITA_FORM_X_ID*: Az értéke az a riport paraméter név, aminek az értékét ez a widget adja majd.
- *BONITA_FORM_X_WIDGET*: A widget típusa (*DATE:* Date picker, *DATE_RANGE*: date range picker, *SELECT*: Single selection widget)
- *BONITA_FORM_X_LABEL*: A widget címkéje.

*DATE* és *DATE_RANGE* esetén még további 2 specifikus property is használható:

- *BONITA_FORM_X_INITIAL VALUE*: Kezdeti értékadás. Példa egy dátumra: „*2013-12-29*”. Egy intervallum: “*2013-03-27 - 2013-03-28*”
- *BONITA_FORM_X_QUERY*: Egy lekérdezés, ami megszerzi a kezdeti értéket.

*SELECT* esetén pedig ez a további 4 specifikus property használható még:

- *BONITA_FORM_X_AVAILABLE_VALUES*: Egy JSON érték adható: *[{"id:"ID_PRODUCT","label":"product"}]*
- *BONITA_FORM_X_QUERY*: Egy lekérdezés, ami megszerzi a kezdeti értékeket.
- *BONITA_FORM_X_HAS_ALL*: Azt jelenti, hogy mindent kiválasztok.
- *BONITA_FORM_X_HAS_ALL_VALUE*: Az értékét definiálja az "All" választásnak.




13.8. ábra. 2 widget létrehozása után a property-k alakulása az iReport-ban

## Az elkészült riport csomagolása és telepítése

A riportot fordítsuk le, ennek eredménye lesz egy *jasper* fájl. Ezt egy *zip* fájlba kell csomagolni, ahol még ezek a komponensek is vannak:

- Java *jar* könyvtárak
- *.properties* fájlok, képek
- egy alriportot egy alkönyvtárba kell tenni a *zip* fájlon belül

Az így kialakult *zip* fájlt kell feltölteni a portál *Analytics* → *Install a Report* menüjénél.

## Tesztelés

Az első riport csomagolása (futási eredmény részlet: 13.9. ábra):

- *Purchase order follow up.jasper* → a lefordított riport
- *bonita_ report_ style_ HTML.jrtx* → stílus template

| Step | Orders |
|---|---|
| Validate received invoice | 1 |
| Approve Purchase Order | 4 |
| Analyse the approval rejection | 1 |

13.9. ábra. Amikor a Bonita BPM típusú datasource-ot használtuk

Az második riport csomagolása (futási eredmény részlet: 13.10. ábra):

- *Purchase order follow up.jasper* → a lefordított riport
- *connection.properties* → a connection paraméterek
- *mysql-connector-java-5.1.23-bin.jar* → a használt adatbázis JDBC drivere.

Purchase order status

Approved

| purchase_order_stat | purchase_order_su |
|---|---|
| APPROVED | 50 |

13.10. ábra. Amikor külső, KPI alapú adatforrást használtuk

# 14. Bonita 6. - Tippek, Trükkök és néhány apróság

Vannak olyan apró témák, amik nem illenek be egy-egy fejezetbe, de nagyon értékes kisebb nagyobb gondolatokat, megoldásokat, ötleteket adnak. Az Informatikai Navigátor minden száma tartalmazza ezt a rovatot, azonban most a kiadvány témájához illeszkedve csak a Bonitához kötődő témákból válogattunk. Szeretnénk megkérni mindenkit, hogy küldje el ötleteit, írásait, ami ebbe a rovatba bekerülhet. A cím ahova az ötleteket és kidolgozott írásokat várjuk: *creedsoft.org@gmail.com.*

## Melyik Simple Button gombot nyomtuk meg?

A Bonita rendelkezik a már ismertetett *Simple Button* vezérlővel, ami nem küld action-t, de a *contingencies* mechanizmusban hasznos módon részt tud venni. Egy példa workflow rendelkezzen 1 darab változóval, aminek a neve *gomb* és text típusú. Legyen 2 darab step, a 2. csak azért, hogy a működést, azaz a *gomb* változó értékének a beállítását láthassuk. Az első step formja ezeket az elemeket tartalmazza:

- widget name=*gomb1* nevű Text Field
- 3 darab Simple Button: widget nevek=*Simple_ button1*, *Simple_ button2*, *Simple_ button3*
- Egy Submit gomb (widget name=*Submit1*)

A kérdés az, hogy milyen módon tudhatjuk meg, hogy melyik Simple Button gombot nyomtuk meg? Esetleg milyen módon tudunk rá eseménykezelőt tenni? A Form Builder jelenleg nem támogatja azt, hogy lekérdezzük melyik volt a megnyomott Simple Button. Erre létezik egy trükk, azaz mindegyik gombhoz felveszünk egy háttér hidden típusú vezérlőt, esetünkben az egyes gombokhoz ezeket vettük fel: *Hidden1*, *Hidden2*, *Hidden3*. Ezután ezeket a *contingencies* beállításokat tettük meg:

- *Hidden1* változik a *Simple_ button1* gomb megnyomására.
- *Hidden2* változik a *Simple_ button2* gomb megnyomására.
- *Hidden3* változik a *Simple_ button3* gomb megnyomására.

Ebben a megoldásban az a szép, hogy mindegyik gomb a maga *Hiddenx* mezőjébe teszi az értéket, ami a saját *Update value* beállítása szerint tud működni. Ez azt jelenti, hogy ide nem csak az előállított értéket tudja kiszámítani, de itt – ahogy tudjuk – egy Groovy script is futhat érték létrehozóként. Ez pedig olyan, mintha egyben egy eseménykezelő volna ez. Megjegyezzük, hogy a hidden mezőknek semmilyen más szerepük nincs, ezért részére sem az *Initial value*, sem az *Output operation* nem kerül konfigurálásra. Az érték úgyis a gombnyomáskor fog beállítódni. Egy tipikus használat az, hogy amikor megnyomjuk a *Submit1* gombot, akkor ott – például az *Actions* lehetőségnél – elérjük mindhárom hidden vezérlő field értéket: *field_ Hidden1*, *field_ Hidden2*, *field_ Hidden3*. Így lehetővé válik a következő formra lépés előtt egy olyan feldolgozás, ami attól

függ, hogy melyik gombot nyomtuk meg. Sőt maga a feldolgozandó hidden értékek is ettől függtek. Itt például beállíthatjuk a workflow *gomb* nevű változóját, aminek az értékét a következő formon láthatjuk.

## I18N használat

A Bonita minden szövegét egy Groovy scripten keresztül is ki lehet írni, amikor nem szöveg konstanst adunk meg. Ilyenkor néha jól jön, ha közvetlenül is meg tudjuk állapítani, hogy éppen melyik *Locale*-ban vagyunk. A Bonita 6.1.1 óta létező új elem, hogy a script editor ablakban a Bonita által szolgáltatott változók (a neve: *Select a provided variable...*) között megjelent a *locale*, ami természetesen *java.util.Locale* típusú. Ezzel az információval már tudni fogjuk, hogy éppen milyen nyelven kell dolgoznia a rendszerünknek.

## Az alkalmazás feliratainak testreszabása

Az alkalmazásokban (és a portálban) megjelenő néhány szöveget a Bonita automatikusan generálja, amit általában érdemes testre szabni legalább 2 ok miatt:

1. A szövegek kifejezőek legyenek, azaz arra utaljanak amivel kapcsolatosan megjelentek.
2. Az I18N megvalósítható legyen

A következőkben 3 tipikus esetre mutatjuk be, hogy mindezt hogyan kell konfigurálni, azonban szeretnénk kiemelni, hogy ahol megjelenik a details panel *General→Portal* füle, ott mindig lehet ilyen beállítást is tenni.

**A Step (TASK) felirata** A step-pek portálon megjelenő nevei alaphelyzetben mindig az a szöveg lesz, amilyen névre elkereszteltük a *General→General* fül *Name* mezőjénél. A *General→Portal* fül *Display name* mezőjében megadható, hogy itt milyen szöveg legyen ehelyett. A Groovy kifejezés szerkesztő is használható, így ez a szöveg *I18N* tudású, dinamikusan generált tartalom is lehet (a korábbiakban erre láttunk is példát az eseményekről szóló fejezetben). Ezen a beállító lapon van még 2 érdekes, esetenként jól használható szövegmegadási lehetőség is:

- *Display description*: Amikor a TASK megkeletkezett, akkor annak a neve alatt olvasható kis másodlagos szöveg. Mindaddig ez látszik, amíg step nem fejeződik be.
- *Description after completion*: Amikor a step befejeződik, ez a szöveg fog megjelenni alatta.

**A generált FORM felirata** Amikor bejön a FORM, akkor a fejsorban egy címe is van, amitől szintén eltérhetünk, ha elmegyünk a FORM *General→General* fülére és megadunk egy értéket a *Show page title* mezőnél (ami itt is Groovy script lehet). Ide alapértelmezésként a FORM neve kerül, amennyiben ezt a lépést nem végezzük el. Azért is érdemes ezt a mezőt használni, mert a dinamikusan generált tartalom az adott konkrét esetre is szolgáltathat hasznos, járulékos információt.

**A TASK befejezésekor kiírt szöveg megadása** Egy TASK formjának elküldésekor ez az alapértelmezett elköszönő szöveg: *The information has been submitted.* A details panel *Application→Confirmation* fülénél ettől is eltérhetünk, ha kitöltjük a *Confirmation message* mezőt. Itt még ennek a layout-ját is megadhatjuk, amennyiben a beépítettel nem vagyunk elégedettek az adott alkalmazás vonatkozásában (*Confirmation layout* mező).

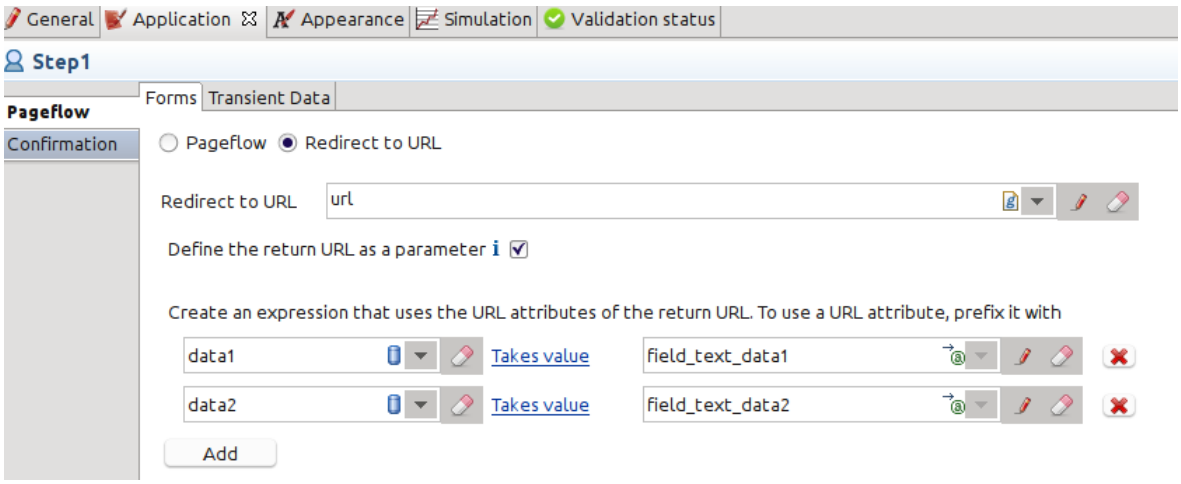
## Egy külső form használata (*Redirect to URL*)

### A feladat

Egy pool indító vagy egy step végrehajtó formját elkészíthetjük egy külső alkalmazásként is. Erre akkor lehet szükség, ha a beépített általános form építő eszközökkel nehézkes vagy éppen lehetetlen a feladat megoldása. Válasszuk ki azt a BPMN elemet (pool, step), amihez a formot szeretnénk rendelni, majd menjünk el a *Application → Pageflow* helyre. Itt most ne a *Pageflow*, hanem a *Redirect to URL* rádiógombot válasszuk ki, amihez meg is jelennek a konfigurációs lehetőségek (14.1. ábra). A most bemutatandó példa BPMN terve 2 step-pől áll: *Step1* és *Step2*. A pool 2 változóval rendelkezik: *data1* (Text) és *data2* (Text). Amint az ábráról is látszik, a *Step1*-hez fogunk készíteni egy külső formot, amit egy JSP lap fog megvalósítani. A példában ez a külső web alkalmazás ugyanarra a web szerverre van telepítve, mint a Bonita portál, így ezek egymás számára a *localhost*-on futnak. Természetesen ez nem követelmény, a külső form egy távoli szerveren is lehet.

### A külső form használatának kialakítása

A 14.1. ábra mutatja a kitöltött konfigurációs panelt.




14.1. ábra. Egy külső form konfigurálása, mint a TASK formja

Az első, amit meg kell adnunk az az URL, ahova át szeretnénk irányítani a működést. Ezt a *Redirect to URL* mezőben adhatjuk meg, esetünkben ezt az URL-t egy Groovy script állítja elő (14-1. Programlista). Látható, hogy a mi JSP lapunkra megy a vezérlés, illetve a külső formnak

átadtuk a workflow változóinak értékeit is. Itt jegyezzük meg, hogy összetettebb esetben jobb gyakorlat lehet a process és task instance ID-k átadása, majd minden további adatot a Bonita API-val meg tudunk szerezni a form működéséhez. A *Define the return URL as a parameter* checkbox bepipálása esetén kiegészül ez az URL egy *submitURL* nevű paraméterrel is, aminek a teljes értékét láthatjuk a 14.2. ábra *A visszatérés helye ez lesz:* szövege alatt. Amennyiben szeretnénk, akkor a form ACTION értékének ezt lehet majd adni, mi a példában ki is használjuk ezt a lehetőséget. Végül arra is van mód, hogy az ACTION válasz URL-be válasz paramétereket is tegyünk (a JSP lapra nézve látható, hogy ez *text_data1* és *text_data2* lesz). Ez a 2 érték *URL Attribute*, ezt jelöli *a* kis bekarikázott a betű, amire a Bonita a *field_text_data1* és *field_text_data2* mezőneveket generálja. Ezzel a módszerrel elegánsan tudjuk a workflow változóit vissza update-elni.

14-1. Programlista: Az url nevű script kódja (Redirect to URL mező)

```
String url = "http://localhost:8082/TestBonitaRedirectToURL/step1Form.jsp?";
url += "data1=" + data1;
url += "&data2=" + data2;
return url;
```

A *step1Form.jsp* lap kódját a 14-2. Programlista mutatja. Vegyük észre a következőket:

1. Az ACTION mindig *GET*, így az URL kódolt válaszparaméterek is előállnak (a Bonita 6.1.x itt nem fogadja el a POST-ot)

2. A kapott *submitURL* string értékeit kivettük és úgy gondoskodtunk azok visszaküldéséről, hogy rejtett mezőbe tettük.

14-2. Programlista: step1Form.jsp - megvalósítja a külső űrlapot

```
1
2  <%@page import="java.net.URLDecoder"%>
3  <%@page import="java.util.Map"%>
4  <%@page import="java.util.LinkedHashMap"%>
5  <%@page import="java.util.StringTokenizer"%>
6  <%@page import="java.net.URL"%>
7  <%@page contentType="text/html" pageEncoding="UTF-8"%>
8  <!DOCTYPE html>
9  <html>
10     <head>
11         <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
12         <title>Teszt Bonita Redirect to URL</title>
13     </head>
14     <body>
15
16         <%
17             String backURL = request.getParameter("submitURL");
18             String data1 = request.getParameter("data1");
19             String data2 = request.getParameter("data2");
20
21             Map<String, String> query_pairs = new LinkedHashMap<String, String>();
22             URL url = new URL(backURL);
23             String query = url.getQuery();
24             String[] pairs = query.split("&");
25
26             for (String pair : pairs) {
27                 int idx = pair.indexOf("=");
28                 query_pairs.put(URLDecoder.decode(pair.substring(0, idx), "UTF-8"), URLDecoder.decode(pair.➥
                       substring(idx + 1), "UTF-8"));
29             }
30
31         %>
32
33         A visszatérés helye lesz:<hr>
34         <%= backURL%><br>
35
36         <hr>
```

```
37
38          <FORM METHOD="GET" ACTION="<%=_backURL%>">
39
40              <input type="text" name="text_data1" value="<%=data1%>">
41              <input type="text" name="text_data2" value="<%=data2%>">
42
43              <input type="hidden" name="executeActions" value="<%=query_pairs.get("executeActions")%>">
44              <input type="hidden" name="form" value="<%=query_pairs.get("form")%>">
45              <input type="hidden" name="task" value="<%=query_pairs.get("task")%>">
46              <input type="hidden" name="mode" value="<%=query_pairs.get("mode")%>">
47              <input type="hidden" name="assignTask" value="<%=query_pairs.get("assignTask")%>">
48
49              <P><INPUT TYPE=SUBMIT value="Elküld">
50          </FORM>
51
52      </body>
53  </html>
```

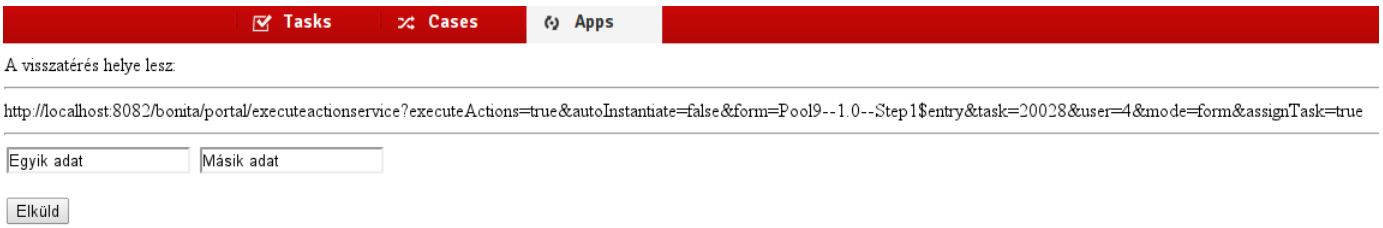
### Tesztelés

Nézzük meg a működést is! A process példány indító formon ezeket az értékeket állítottuk be:

- *data1* → *Egyik adat*
- *data2* → *Másik adat*

Ahogy a 14.2. ábráról látszik, a *Step1* elvégzéséhez tényleg elkerültünk a saját, külső JSP lapunkra. Ott megváltoztattuk a 2 adat értékét és megnyomtuk az Elküld gombot, ami után a következő 3 dolgot figyelhetjük meg:

1. Visszakerültünk a portálra, ahogy az szokásos
2. A *Step1* kompletté vált és megkeletkezett a *Step2* feladat
3. Megnézve a *Step2* feladaton keresztül a workflow adatait, azok is rendben módosultak.



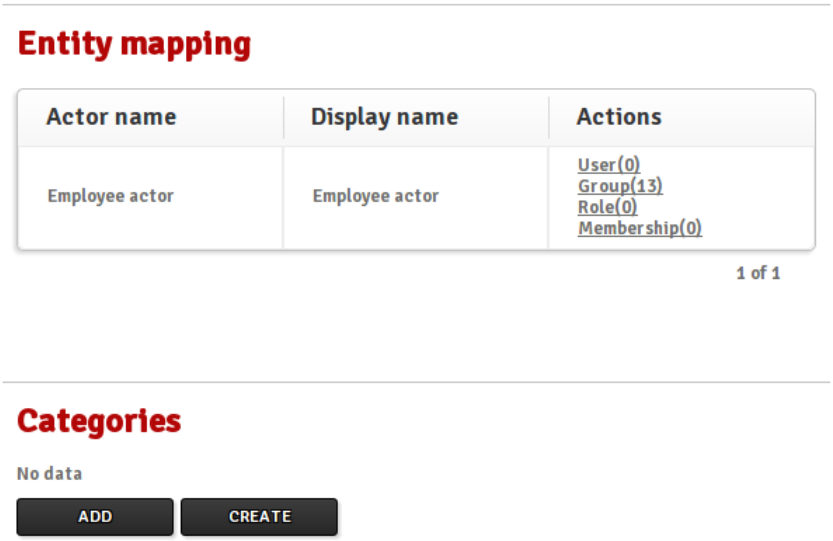

14.2. ábra. Egy külső form konfigurálása - Tesztfutás közben

### Megjegyzés

Lényegesnek tartjuk, hogy külön kiemeljük azt, hogy egy FORM másképpen is működhet. Az is lehet, hogy az ACTION URL nem a portálra mutat vissza, hanem egy másik alkalmas helyre (például saját magára). Ez nem okoz gondot. Az sem követelmény, hogy a workflow adatait a válasz URL-be kódolva visszaadott paraméterekkel frissítsük. Egy modern webes keretrendszer (például *Vaadin*, *JavaServer Faces*, *zkoss*) esetén azt is megtehetjük, hogy a Bonita API segítségével tesszük meg mindezt, sőt bizonyos számú adat esetén valószínűleg mindig ez lesz a jó megoldás.

## Az ACTOR-ok adminisztrálása a portálon

Amikor az ACTOR neveket konkrét USER halmazra oldjuk fel, akkor ezt *Actor Mapping*nek nevezzük. Azt már leírtuk részletesen, hogy ezt fejlesztési időben miképpen tudjuk elvégezni, sőt már utaltunk a most röviden bemutatandó *Entity mapping* lehetőségre is. Amennyiben *Administrator* profillal lépünk be a portálra, aztán kiválasztva egy alkalmazást (*Apps management*→*Apps* és már választhatjuk is), majd megnyomva a *MORE* gombot jön elő az az oldal, aminek egy részletét mutatja a 14.3. ábra. Itt azt nézhetjük meg, hogy jelen pillanatban az ACTOR kikre van feloldva, azaz láthatjuk azokat a konkrét *User*, *Group*, *Role* és *Membership* listákat ahonnan jöhet a generált USER halmaz. Mindez a process telepítésekor állítódik be, de a Bonita lehetővé teszi, hogy ezt az *Actions* oszlopban dinamikusan is változtassuk, ami nagy rugalmasságot és könnyebbséget ad a fejlesztőnek és üzemeltetőnek egyaránt. Itt jegyezzük meg, hogy amennyiben még semmi nincs az ACTOR-hoz rendelve, úgy a process alkalmazás disabled státuszba kerül. Miután beállítottuk a megfelelő feloldásokat, kézzel kell a process-t engedélyezni az *Enable* gomb megnyomásával.




14.3. ábra. Entity mapping az ACTOR-USER feloldáshoz

## A végrehajtási sorrendek és állapotok részletesebb áttekintése

### A process indító form létrehozása

1. Az engine létrehozza és inicializálja a transient adatokat. A kezdőérték az lesz, amit a default értékek résznél egy kifejezéssel definiáltunk.
2. A következő (SP only) lépésnél a motor a *case start form* megjelenítése előtt meghívja az ott megadott konnektorokat, amik szintén a kezdőértékeket állítják be. Ez felül tudja írni az előzőleg beállított értékeket is.
3. Minden egyes widget-re egy kifejezés értékelődik ki, aminek az a feladata, hogy meghatározza a megjelenítendő vezérlőket. Ez csak egyszer fog kiértékelődni.

4. A motor kiértékeli a *contingency condition* alapján, hogy mely widget-eket kell azonnal megjeleníteni az indító form esetében (SP only).

5. Az indító form minden egyes widget-jének az értéke beállításra kerül. Itt használhatunk paramétereket, konstansokat is.

**A process példány létrehozása az indító form segítségével**

1. Az indító form megjelenik a felhasználó számára

2. A felhasználó beviszi az adatokat

3. Amennyiben a form tartalmaz kontingencia beállításokat, úgy azt a motor kiértékeli és végrehajtja a megfelelő widget-ek értékének a frissítését (SP only)

4. Az engine kiértékeli, hogy mely widget-eket kell megjeleníteni és melyek maradnak rejtve. Mindez szintén a contingency feldolgozásra épül.

5. Lényeges megjegyezni, hogy amennyiben több contingent elem függ egy értéktől, úgy a végrehajtási sorrend nem garantált.

6. A Form elküldése akkor történik, amikor megnyomjuk a Submit button-t.

7. A Submit button hatására a rendszer végrehajtja a validator-okat az összes widget-re.

8. Amennyiben mindegyik widget szintű validátor eredménye pozitív, úgy ez elindítja a lap (form) szintű validátorokat.

9. A portál meghívja az új process példány létrehozás funkciót.

10. Az Engine leellenőrzi, hogy a process engedélyezett-e (enabled vagy disabled lehet)

11. A motor lekéri a process definition-t, majd létrehozza a process instance-t a megfelelő inicializációval.

12. Minden egyes process változóra a motor beállítja a kezdőértékeket, ha van olyan.

13. Az Engine létrehozza és inicializálja search indexeket.

14. A következő lépés a Documents-ek beállítása, ha van.

15. Végrehajtódnak a definiált konnektorok (szinkron hívás). A hívási sorrend azzal egyezik meg, ahogy a Stúdióban megadtuk őket.

16. Az akciók is végrehajtódnak, a sorrend itt is a megadás sorrendje.

17. Submit button, amiről az Engine informálja a portált.

18. A következő lépés a process példány létrehozásának a befejezése. Az Engine példányosítja és végrehajtja az *on enter* konnektorokat (evaluate input expression, execute, evaluate output operation). Ez egy asszinkron művelet, ahol a végrehajtás ideje 5 percben van limitálva alapértelmezetten.

19. Subprocess esetén a motor specifikus inicializálást is végez (data mapping).

20. Kiértékelésre és megjelenítésre kerül a case start form confirmation message.

21. Elkezdődik a process flow.

**Egy Step végrehajtásának lépései**

1. A Bonita motor inicializálja a step változókat, ha van, akkor a default értékkel.

2. A motor végrehajtja az Actor Filter-t.

3. Végrehajtódnak az *on enter* konnektorok.

4. Kiértékelődik a Step dinamikus neve és leírása.

5. Mindezek amik eddig voltak, csak 1 alkalommal hajtódnak végre.

6. A portál kiteszi a TASK formját, a felhasználó beviszi az adatait.

7. Contingency kiértékelés. (SP only)

8. Amikor a Next vagy Submit button megnyomásra kerül, ezek történnek:

   (a) A widget-ek validátorai lefutnak (SP only).

   (b) Amennyiben mindegyik validátor pozitív visszajelzést ad, lefut a lap szintű validátor. A Previous button esetén nem fut validátor. (SP only)

   (c) A Next button hatására a motor kiértékeli, hogy melyik lapra menjen tovább.

9. Végrehajtódnak az Operations műveletek, amik az adott step-hez vannak rendelve.

10. Lefutnak a kilépő konnektorok.

11. Az Engine kiértékeli a Description és after step *Step confirmation* üzeneteket és a portál megjeleníti.

12. A step kompletté válik.

### Flow node állapotok és átmenetek

Az állapot lehet *stable* vagy *terminal*:

- *stable*: A flow node még nem fejeződött be és várakozik valamilyen inputra, ami tipikusan egy felhasználói interrakció.
- *terminal*: Ez a legutolsó állapota egy flow node-nak.

## Form és mező validáció

Az előző pont pontban már említettük, hogy először mindig a mezőszintű validáció fut le, ha az sikeres, akkor a lap szintű. Érdemes sok ellenőrző feltételt megfogalmazni, mert jó inputra a programunk helyesen és sokkal stabilabban fog működni. Egy új validátor a FORM details panel *General→Validators* fülön tudunk hozzáadni az egész form-hoz vagy az éppen kijelölt widget-hez. Minden widget egyébként rendelkezik egy rá jellemző default validátorral is, amit az *Add to default validator for this widget* checkbox bepipálásával kérhetünk. A validátorok használatát a 3. fejezet *Submit Button* részében már bemutattuk, így itt most csak néhány további információt szeretnénk átadni. Az előre elkészített validátorok a következőek:

- *Character* validator: Egy szimpla karaktert ellenőriz le.
- *Date* validator: Egy dátum ellenőrzése
- *Decimal* vagy *Double* decimal validator: Számokra vonatkozóan fogalmazhatunk meg érték-ellenőrzést.
- *Groovy* validator: A validátor paramétere ilyenkor maga a Groovy script, ami lefut és *true* vagy *false* értéket ad vissza.
- *Short integer*, *Integer*, *Long integer* validator: integer érték ellenőrzése.
- *Length*: Leellenőrzi, hogy a paraméterül kapott számérték egyenlő-e a string hosszával.
- *Mail validator:* Ellenőrzi, hogy a paraméterül kapott string egy érvényes e-mail cím.
- *Phone number*: Ellenőrzi, hogy a paraméterül kapott string egy érvényes telefonszám.
- *Regex validator*: Ellenőrzi, hogy a paraméterül kapott string (ami egy reguláris kifejezés) illeszkedik-e a stringre.

A *Development→Validators* menüpontnál mi is készíthetünk egy saját validátort, aminek a generált, kiinduló kódját a 14-3. Programlista és 14-4. Programlisták mutatják.

14-3. Programlista: MyCheck class - Field szintű saját validátor váz

```
package org.cs.validator;

import java.util.Locale;
```

```
import org.bonitasoft.forms.client.model.FormFieldValue;
import org.bonitasoft.forms.server.validator.IFormFieldValidator;

public class MyCheck implements IFormFieldValidator {

        @Override
        public String getDisplayName() {
                // TODO Auto-generated method stub
                return null;
        }

        @Override
        public boolean validate(FormFieldValue fieldInput, Locale locale) {
                // TODO Auto-generated method stub
                return false;
        }
}
```

14-4. Programlista: MyValidator2 class - Page szintű saját validátor váz

```
package org.cs.validator;

import java.util.Locale;
import java.util.Map;

import org.bonitasoft.forms.client.model.FormFieldValue;
import org.bonitasoft.forms.server.validator.IFormPageValidator;

public class MyValidator2 implements IFormPageValidator {

        @Override
        public String getDisplayName() {
                // TODO Auto-generated method stub
                return null;
        }

        @Override
        public boolean validate(Map<String, FormFieldValue> fieldValues,
                        Locale locale) {
                // TODO Auto-generated method stub
                return false;
        }
}
```

## RESTful egyszerű HTTP programozással

Ezen pont érdekessége, hogy kicsit többet látunk a RESTful API hátteréből, mert alacsonyabb szinten fogjuk programozni a HTTP protokollt. A példa azt fogja megmutatni, hogyan tudjuk elérni a saját TASK listánkat. A kód kizárólag Java SDK-ra épülve kezeli a REST hívást. A működés főbb pontjai megjegyzéssel vannak ellátva, ezért további magyarázatot nem igényel, de tanulmányozása hasznos lehet.

14-5. Programlista: GetTaskListHTTP - A saját TASK lista lekérése

```
package org.bonitasoft.engine.external;

import java.io.BufferedReader;
```

```
import java.io.DataOutputStream;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

import org.ow2.bonita.util.Base64;

public class GetTaskListHTTP {
    public static void main(final String[] args) throws Exception {

        final String technicalUser = args[0];
        final String technicalPassword = args[1];
        final String endUserLogin = args[2];
        final String serverAddress = args[3];

        // define URL and content
        final String url = serverAddress + "/API/queryRuntimeAPI/getLightTaskListByActivityState➥
            /READY";
        final String resquestConent = "options=user:" + endUserLogin + ",␣queryList:➥
            journalQueryList";

        // execute request
        HttpURLConnection connection = null;
        DataOutputStream output = null;
        try {
            connection = (HttpURLConnection) new URL(url).openConnection();
            connection.setUseCaches(false);
            connection.setDoInput(true);
            connection.setDoOutput(true);
            connection.setInstanceFollowRedirects(false);
            connection.setRequestMethod("POST");
            connection.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");

            final String technicalUserAndPassword = technicalUser + ":" + technicalPassword;
            connection.setRequestProperty("Authorization", "Basic␣" + Base64.encodeBytes(➥
                technicalUserAndPassword.getBytes()));

            output = new DataOutputStream(connection.getOutputStream());
            output.writeBytes(resquestConent);
            output.flush();
        } finally {
            if (output != null) {
                output.close();
            }
            if (connection != null) {
                connection.disconnect();
            }
        }

        // process response
        if (connection != null) {
            final int responseCode = connection.getResponseCode();
            if (responseCode != HttpURLConnection.HTTP_OK) {
                System.out.println("Request␣failed:␣" + responseCode);
            } else {
                System.out.println("Tasks␣for␣user␣" + endUserLogin);
                final InputStream is = connection.getInputStream();
                final BufferedReader reader = new BufferedReader(new InputStreamReader(is));
                String line;
                final StringBuffer response = new StringBuffer();
                try {
                    while ((line = reader.readLine()) != null) {
                        response.append(line);
                        response.append('\n');
```

```
                }
            } finally {
                reader.close();
                is.close();
            }
            System.out.println(response.toString().trim());
        }

    }
  }
}
```

## Listázás és keresés

A Bonita BPM Engine API számos listázó és kereső metódust tartalmaz. Ennek használatát az API bemutatásakor már megismertük, de érdemes külön is újra rápillantani erre a témára. Tegyük fel, hogy szükségünk van az archivált taszkok egy halmazára, amit egy listába tudunk lekérdezni az alábbi módon:

```
List<ArchivedActivityInstance> archivedActivityInstances = TenantAPIAccessor.getProcessAPI(➥
    session).getArchivedActivityInstances(instance.getId(), 0, 100, ActivityInstanceCriterion.➥
    NAME_ASC);
```

A fentiekhez hasonlóan listázhatjuk az egyéb Bonita objektumokat is, de értelemszerűen az *ArchivedActivityInstance* class helyett mindig a megfelelőt kell használnunk. A következő példa ugyanezen típusú objektumokban való keresést mutatja be, aminél a már megismert *SearchOptionsBuilder* class építi fel a keresési feltételt:

```
SearchOptionsBuilder searchBuilder = new SearchOptionsBuilder(0, 100);
searchBuilder.sort(ArchivedActivityInstanceSearchDescriptor.NAME, Order.ASC);

SearchResult<ArchivedActivityInstance> archActivitResult = TenantAPIAccessor.getProcessAPI(➥
    session).searchArchivedActivities(searchBuilder.done());
```

Ez a példa szintén egy keresést mutat be, de a rendezés mellett már egy filtert is használ:

```
SearchOptionsBuilder searchBuilder = new SearchOptionsBuilder(0, 100);
searchBuilder.filter(ArchivedActivityInstanceSearchDescriptor.ROOT_PROCESS_INSTANCE_ID, ➥
    processInstance.getId());
searchBuilder.sort(ArchivedActivityInstanceSearchDescriptor.NAME, Order.ASC);
SearchResult<ArchivedActivityInstance> archActivitResult = TenantAPIAccessor.getProcessAPI(➥
    session).searchArchivedActivities(searchBuilder.done());
```

A filter ez esetben azt jelenti, hogy a megadott process instance archivált taszkjait adja vissza.

## Pár gondolat a FORM kinézetének beállításáról

Amikor a form készítésről írtunk, akkor annak logikai használatán volt a hangsúly, így nem említettük meg azokat a formázási lehetőségeket, amik a kinézetet szabják testre. Ezen lehetőségek beállítását a FORM details panel *Appearance* fülén tehetjük meg. A mezők (widget-ek) megjelenítésének beállításánál a következő lehetőségeket beállító tabok vannak:

- *Grid*: A szélesség és magasság beállítása.

- *Cell*: A szélesség és magasság, valamint egyedi CSS stílus beállítása.

- *Label*: A szélesség és magasság, valamint egyedi CSS stílus beállítása. Megadhatjuk a betű típusát is, illetve rendelkezhetünk arról, hogy a címke hol helyezkedjen el (bal vagy jobb oldal, alul vagy felül).

- *Field*: A beállítási lehetőségek a Label-lel egyeznek meg, de itt még megadhatóak további adatok is (például a maximálisan bevitt karakterek száma).

### A Grid kinézetének módosítása

A grid cellái tartalmazzák a widget-eket. A szélessége és magassága pixelben vagy százalékban (példa: *50%*) adható meg. Amennyiben nem adunk meg semmilyen értéket egyetlen sorra vagy oszlopra sem, úgy egyenlő arányban osztódik szét a hely a grid cellái számára. Amikor megadunk egy pixel vagy százalék értéket egy oszlopra, akkor az a következő 3 dolgot jelenti egyszerre:

1. az oszlop számára elfoglal akkora helyet (amennyiben százalékot adtunk meg, úgy a fennmaradt hely akkora %-a lesz az oszlop szélessége)

2. észrevehetjük azt is, hogy az oszlopban mindegyik field felveszi ezt az értéket automatikusan, mert csak ennek van értelme.

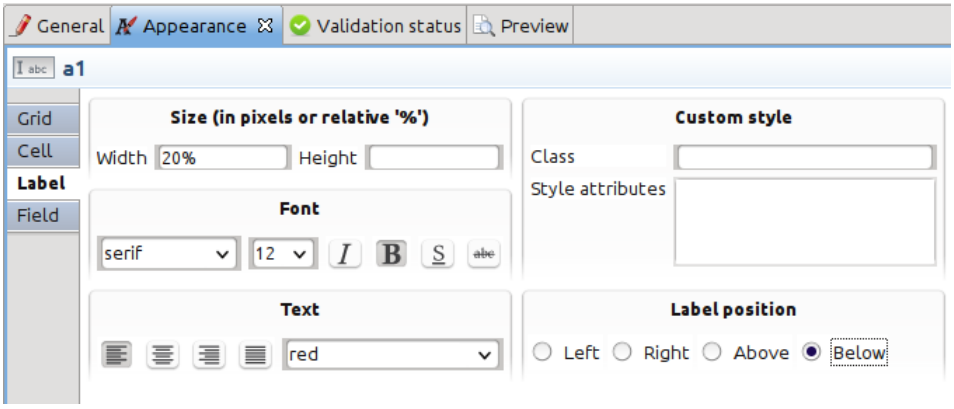
3. A további oszlopok – amennyiben nekik nem adunk értéket – egyenlő arányban osztoznak a megmaradt helyen.

### A Cella kinézetének módosítása

Az itt beállított szélesség és magasság a field körüli üres helyet tudja növelni vagy csökkenteni. Ennek azonban határt szab a grid adta méret lehetőség, így vigyázzunk erre a beállításra, mert a mező egy része eltűnhet, ha az üres hely túl nagy.

### A Label kinézetének módosítása

A lehetőségeket a 14.4. ábra mutatja.




14.4. ábra. A Label kinézetének beállítása

### A Field kinézetének módosítása

A Field fül is hasonló, mint a 14.4. ábra, de itt még megtalálhatóak további mezők is (például az említett *Max char*)

### A FORM változtatása és a CSS használata

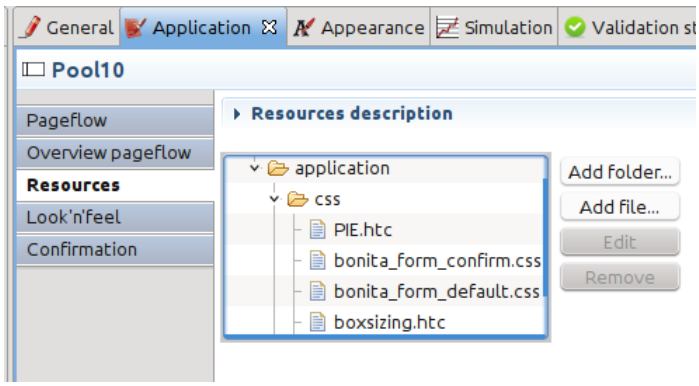
Lehetőség van a generált formra saját HTML kód szintű változtatásokat tenni. Ehhez a form *General→General* fülét jelöljük ki és nyomjuk meg a *Use layout generated from design* feliratú gombot. Ekkor az Edit gombra már lehet is szerkeszteni a HTML lapot, ami például esetünkben így nézett ki, ahohy a 14-6. Programlista mutatja.

14-6. Programlista: A Bonita Stúdió form szerkesztése

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html dir="ltr" xml:lang="en" xmlns="http://www.w3.org/1999/xhtml" lang="en">
        <head>
                <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
                <meta name="description" content="Bonita Forms Application"/>
                <link href="css/bonita_form_default.css" rel="stylesheet" type="text/css"/>

                        <link href="css/generatedcss.css" rel="stylesheet" type="text/css"/>
</head>
        <body>
                <div id="bonita_form_page_label" class="bonita_form_page_label"></div>
                <div id="bonita_form_step_header">
                        <div id="bonita-object-area-top"></div>
                        <div id="bonita-object-area-middle">
                                <div id="bonita-object-area-form">$label.bonita_step_reachedStateDate ➥
                                    $bonita_step_reachedStateDate</div>
                                <div id="bonita-object-area-to">$label.bonita_step_expectedEndDate ➥
                                    $bonita_step_expectedEndDate</div>
                                <div id="bonita-object-area-priority">$label.bonita_step_priority <span class="➥
                                    bonita-priority">$bonita_step_priority</span></div>
                                <div id="bonita-object-area-description">$bonita_step_description</div>
                        </div>
                        <div id="bonita-object-area-bottom"></div>
                </div>
                <div class="bonita_form_container">
    <div>
       <div>
       </div>
    </div>
    <table class="bonita_table_container" width="100%">
      <colgroup>
        <col class="_vlkasHPxEeOkL9XxkuCRhw__3pfN0HPxEeOkL9XxkuCRhw_Column0"/>
        <col class="_vlkasHPxEeOkL9XxkuCRhw__3pfN0HPxEeOkL9XxkuCRhw_Column1"/>
        <col class="_vlkasHPxEeOkL9XxkuCRhw__3pfN0HPxEeOkL9XxkuCRhw_Column2"/>
      </colgroup>
```

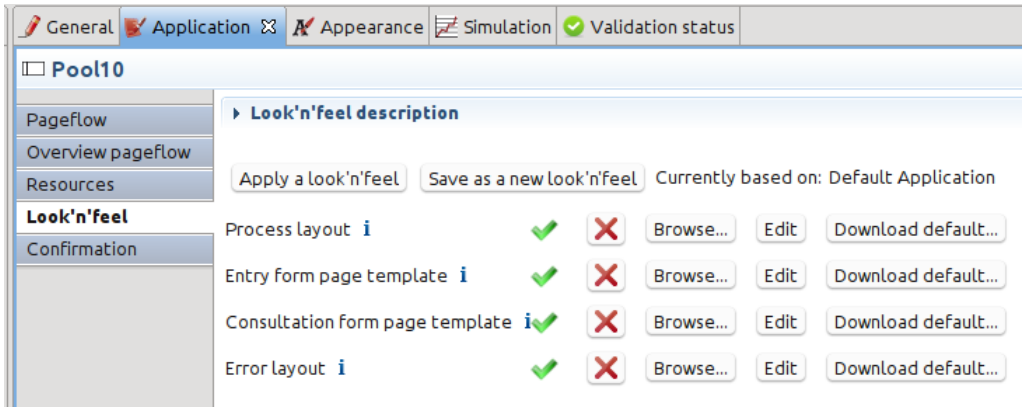
```
      <tr class="_vlkasHPxEeOkL9XxkuCRhw__3pfN0HPxEeOkL9XxkuCRhw_Row0">
        <td class="widget__vlkasHPxEeOkL9XxkuCRhw__3pfN0HPxEeOkL9XxkuCRhw_a1" colspan="1" rowspan="1">
          <div id="a1">
           </div>
          <div id="a1_default_validator">
           </div>
         </td>
        <td class="widget__vlkasHPxEeOkL9XxkuCRhw__3pfN0HPxEeOkL9XxkuCRhw_Date1" colspan="1" rowspan="1">
          <div id="Date1">
           </div>
          <div id="Date1_default_validator">
           </div>
         </td>
        <td class="widget__vlkasHPxEeOkL9XxkuCRhw__3pfN0HPxEeOkL9XxkuCRhw_Date2" colspan="1" rowspan="1">
          <div id="Date2">
           </div>
          <div id="Date2_default_validator">
           </div>
         </td>
      </tr>
      <tr class="_vlkasHPxEeOkL9XxkuCRhw__3pfN0HPxEeOkL9XxkuCRhw_Row1">
        <td class="widget__vlkasHPxEeOkL9XxkuCRhw__3pfN0HPxEeOkL9XxkuCRhw_b1" colspan="1" rowspan="1">
          <div id="b1">
           </div>
          <div id="b1_default_validator">
           </div>
         </td>
        <td class="widget__vlkasHPxEeOkL9XxkuCRhw__3pfN0HPxEeOkL9XxkuCRhw_Message1" colspan="1" rowspan="1">
          <div id="Message1">
           </div>
         </td>
        <td>
         </td>
      </tr>
      <tr class="_vlkasHPxEeOkL9XxkuCRhw__3pfN0HPxEeOkL9XxkuCRhw_Row2">
        <td class="widget__vlkasHPxEeOkL9XxkuCRhw__3pfN0HPxEeOkL9XxkuCRhw_Submit1
            _vlkasHPxEeOkL9XxkuCRhw__3pfN0HPxEeOkL9XxkuCRhw_td_Submit1" colspan="1" rowspan="1">
          <div id="Submit1">
           </div>
         </td>
        <td>
         </td>
        <td>
         </td>
      </tr>
    </table>
    <div>
       <div>
       </div>
    </div>


              </div>
        </body>
</html>
```

Itt látható, hogy hol van az alapértelmezett CSS fájl, de mi is tehetünk mellé egy másikat. A hivatkozott erőforrásokat a 14.5. ábrán mutatott helyről is menedzselhetjük, itt most ez a CSS fájl kapcsán érdekes.




14.5. ábra. Az alkalmazás webes erőforrásai

A CSS fájlba felvehetünk új stílus osztályokat is, aminek a nevét a a 14.4. ábrán is látható *Custom style* helyen adhatjuk meg. A 14.6. ábra az általános alkalmazás kinézetének konfigurálási lehetőségeit mutatja.




14.6. ábra. Az általános alkalmazás kinézet konfigurálása

### Az opciók

A Form *General→Options* füle arra szolgál, hogy szabályozzuk a következő dolgokat:

1. *Html attributes*: HTML attribútumok megadását teszi lehetővé (példa: class="popup").
2. *Insert widget if*: Ide egy Groovy kifejezés kerülhet, amely scriptnek egy logikai értékkel kell visszatérnie.
3. *Is mandatory*: Ez egy checkbox, amivel megadhatjuk, hogy ez a mező kötelező-e. Amennyiben igen, úgy megadhatjuk az üzenet szövegét is.
4. *Read only*: Ez egy checkbox, amivel megadhatjuk, hogy ez a mező csak olvasható-e.

## Egy másik apró példa a Contingency használatára

Gyakori feladat, hogy egy Field group mezőinek az értékét össze kell adnunk, az eredményt pedig egy csoporton kívüli helyre kell írnunk. Tegyük fel, hogy van egy *Group1* nevű group-unk, amire beállítottuk a *Multiple* pipát. Ennek a group-nak van egy *miles* nevű mezője is, amibe a mérföld értékeket írjuk be. A feladat az, hogy egy group-on kívüli mezőnek adjuk vissza contingency segítségével a mérföldek összegét. Ekkor a külső mező *Contingencies* fülénél készítsünk az *Update value* helyen egy ilyen scriptet, ahol a visszatérési érték *Float* lesz:

```
Float result = 0.0f;
if (field_Group1 != null) {
        for (def item : field_Group1) {
                result += item.get("miles");
        }
}
return result;
```

## Egy HTML widget tartalmának kezelése JQuery-vel

Tegyünk ki a formra egy HTML vezérlőt és az *Initial value* értékét egy konstans szöveggel töltsük fel, ami egy *JQuery* script lesz:

```
<div id="dlg">
<input type='text' name='value' value='Alma'/>
</div>

<script>
        var currentInputItem = null;

        jQuery("div#dlg").dialog({
                modal : true,
                title : "test dialog",
                autoOpen : false,
                buttons : {
                        'OK' : function() {
                                jQuery("div#dlg").dialog("close");
                                jQuery(currentInputItem).val(jQuery("input[name='value']").val()
                                    );
                        },
                        'Cancel' : function() {
                                jQuery("div#dlg").dialog("close");
                        }
                }
        });

jQuery(document).ready(function() {
        jQuery(".popup").click(function() {
                openPopup(jQuery(this));
        });

});


function openPopup(inputItem) {
        currentInputItem = inputItem;
        jQuery("div#dlg").dialog("open");
}

</script>
```

Ennek hatására a bekerült *input* html text field az *Alma* értékkel fog megjelenni.

## AD/LDAP authentication

Ez egy új gyári lehetőség, a Bonita 6.1.0 verzióban jelent meg. Az előfeltétel az, hogy amikor bejelentkezünk a Bonita Portálra, akkor a felhasználónak léteznie kell az LDAP címtárban és a Bonita saját adatbázisában is. A jelszó az LDAP adatbázissal szemben lesz leellenőrizve.

### Az Bonita alapértelmezett LoginManager működésének megadása

Van ez az XML, ahol meg lehet mondani, hogy milyen módon történjen az alapértelmezett authentication:

```
<bonita.home>/server/tenants/1/conf/services/cfg-bonita-authentication-impl.xml
```

Ez a tartalma:

```
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/➥
    XMLSchema-instance" xmlns:p="http://www.springframe
        xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.➥
    springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="authenticationService" class="org.bonitasoft.engine.authentication.impl.➥
    AuthenticationServiceImpl">
        <constructor-arg name="identityService" ref="identityService" />
        <constructor-arg name="logger" ref="technicalLoggerService" />
    </bean>

</beans>
```

### Az LDAP authentication konfigurálása

**Változtatás a bonita home-ban** A *cfg-bonita-authentication-impl.xml* fájlban az alapértelmezett *AuthenticationServiceImpl* class-t cseréljük le ezzel:

```
com.bonitasoft.engine.authentication.impl.JAASAuthenticationServiceImpl
```

Ekkor az új XML konfigurációs fájl így fog kinézni:

```
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/➥
    XMLSchema-instance" xmlns:p="http://www.springframe
        xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.➥
    springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="authenticationService" class="com.bonitasoft.engine.authentication.impl.➥
    JAASAuthenticationServiceImpl">
        <constructor-arg name="identityService" ref="identityService" />
        <constructor-arg name="logger" ref="technicalLoggerService" />
    </bean>

</beans>
```

**A JAAS login context konfigurálása** A konfiguráció célja megadni az LDAP/AD szerver elérhetőségét. A login context kötelező neve ez: *BonitaAuthentication-tenant_id*. Amennyiben egyszerű Java klienst vagy Tomcat-et használunk, akkor hozzunk létre egy *bonita.cfg* fájlt és ez legyen a tartalma:

```
BonitaAuthentication-1 {
        com.sun.security.auth.module.LdapLoginModule sufficient
              userProvider="ldap://localhost:389/ou=People,dc=example,dc=com"
              userFilter="(&(uid={USERNAME})(objectClass=inetOrgPerson))"
              authzIdentity="{USERNAME}"
              debug=true
              useSSL=false;
};
```

A következő Java system változó mutasson a fenti fájlra, teljes útvonallal:

```
java.security.auth.login.config
```

Amennyiben JBOSS szervert használunk, úgy ezt a fájlt kell módosítani és hozzáadni egy új login context-et a JBOSS szintaxis használatával:

```
JBoss_home/server/default/conf/login-config.xml
```

A beszúrt rész így néz ki:

```
<application-policy name="BonitaAuthentication-1">
    <authentication>
      <login-module code="com.sun.security.auth.module.LdapLoginModule"
         flag="required">
         <module-option name="userProvider">ldap://localhost:389/ou=People,dc=example,dc=com</➥
             module-option>
         <module-option name="userFilter">(&(uid={USERNAME})(objectClass=inetOrgPerson))</module-➥
             option>
         <module-option name="authzIdentity">{USERNAME}</module-option>
         <module-option name="debug">true</module-option>login

         <module-option name="useSSL">false</module-option>
       </login-module>
    </authentication>
 </application-policy>
```

A fenti beszúrt rész AD esetén így alakul:

```
<application-policy name="BonitaAuthentication-1">
<authentication>    <login-module code="com.sun.security.auth.module.LdapLoginModule" flag="➥
    required">
<module-option name="userProvider">ldap://localhost:389/cn=users,dc=example,dc=com</module-➥
    option>
<module-option name="authIdentity">{USERNAME}@example.com</module-option>
<module-option name="userFilter">(&(|(samAccountName={USERNAME})(userPrincipalName={USERNAME})(➥
    cn={USERNAME}))(objectClass=user))</module-option>
<module-option name="debug">true</module-option>
<module-option name="useSSL">false</module-option>
</login-module>
</authentication>
</application-policy>
```

**A java.security.auth.login.config beállítás** Említettük, hogy ezt be kell állítani, de nem mondtuk eddig meg azt, hogy miképpen tehetjük meg. Tomcat alatt a *<TOMCAT_HOME>/bin* könyvtárban található *setenv.sh* fájlba azt a sort beírhatjuk:

```
SECURITY_OPTS=<-Djava.security.auth.login.config=<path-to-file-jaas-bonita.cfg>
```

És alatta ezt a sort

```
CATALINA_OPTS="${CATALINA_OPTS} ${BONITA_HOME} ${DB_OPTS} ${BTM_OPTS} -Dfile.encoding=UTF-8➥
    Xshare:auto -Xms512m -Xmx1024m -XX:MaxPermSize=256m -XX:+HeapDumpOnOutOfMemoryError"
```

cseréljük ki ezzel:

```
CATALINA_OPTS="${CATALINA_OPTS} ${BONITA_HOME} ${DB_OPTS} ${BTM_OPTS} ${SECURITY_OPTS} -Dfile.➥
    encoding=UTF-8 -Xshare:auto -Xms512m -Xmx1024m -XX:MaxPermSize=256m -XX:+➥
    HeapDumpOnOutOfMemoryError"
```

## Fontosabb Bonita Stúdió könyvtárak

A Bonita Stúdió */opt/BonitaBPMSubscription-6.1.2/workspace/default* könyvtára alatt van néhány olyan mappa, amit feltétlenül ismernünk kell:

- *diagrams*: Itt vannak a *proc* kiterjesztésű XML fájlok, amik egy-egy BPMN diagram tartalmát (BMPM, formok, validátorok, ...) tartalmazzák.

- *customTypes*: Ide kerülnek azok a segédfájlok, amik segítik a stúdióval létrehozott custom data type-ok létrehozását.
- *environements*: Az egyes környezetek XML leírása
- *forms*: A form template-ek helye
- *lib*: A hozzáadott Java *jar* fájlok helye (ide kerül az adattípusból generált *jar* is).
- *organizations*: Például alapértelmezésben itt található az *ACME.organization* XML fájl is.
- *src-connectors*: A saját konnektorok Java kódja ide kerül.
- *src-customTypes*: A saját adattípusok Java kódja ide kerül.
- *src-filters*: A saját filterek Java kódja ide kerül.
- *src-validators*: A saját validátorok Java kódja ide kerül.
- *xsd*: A saját adattípusokból generált XSD ide kerül.

## Egy saját Bonita hitelesítő szerviz készítése

Az előző pontban leírt LDAP hitelesítő modulhoz hasonlóan mi is készíthetünk egy sajátot, ami úgy hitelesít, ahogy mi szeretnénk. Ennek a kódja nagyon egyszerű, csak az *AuthenticationService* interface-t kell implementálni, ahogy a példa is mutatja:

```
package org.bonitasoft.engine.authentication.impl;

import org.bonitasoft.engine.authentication.AuthenticationException;
import org.bonitasoft.engine.authentication.AuthenticationService;
import org.bonitasoft.engine.commons.LogUtil;
import org.bonitasoft.engine.identity.IdentityService;
import org.bonitasoft.engine.identity.SUserNotFoundException;
import org.bonitasoft.engine.identity.model.SUser;
import org.bonitasoft.engine.log.technical.TechnicalLogSeverity;
import org.bonitasoft.engine.log.technical.TechnicalLoggerService;

/**
 * @author Elias Ricken de Medeiros
 * @author Matthieu Chaffotte
 * @author Hongwen Zang
 */
public class AuthenticationServiceImpl implements AuthenticationService {

    private final IdentityService identityService;

    private final TechnicalLoggerService logger;

    public AuthenticationServiceImpl(final IdentityService identityService, final ➥
        TechnicalLoggerService logger) {
        this.identityService = identityService;
        this.logger = logger;
    }

    @Override
```

```
    public SUser checkUserCredentials(final String userName, final String password) throws ➥
        AuthenticationException {
        try {
            if (logger.isLoggable(this.getClass(), TechnicalLogSeverity.TRACE)) {
                logger.log(this.getClass(), TechnicalLogSeverity.TRACE, LogUtil.➥
                    getLogBeforeMethod(this.getClass(), "checkUserCredentials"));
            }
            final SUser user = identityService.getUserByUserName(userName);
            final boolean valid = identityService.chechCredentials(user, password);
            if (!valid) {
                throw new AuthenticationException();
            }
            if (logger.isLoggable(this.getClass(), TechnicalLogSeverity.TRACE)) {
                logger.log(this.getClass(), TechnicalLogSeverity.TRACE, LogUtil.➥
                    getLogAfterMethod(this.getClass(), "checkUserCredentials"));
            }
            return user;
        } catch (final SUserNotFoundException sunfe) {
            if (logger.isLoggable(this.getClass(), TechnicalLogSeverity.TRACE)) {
                logger.log(this.getClass(), TechnicalLogSeverity.TRACE, LogUtil.➥
                    getLogOnExceptionMethod(this.getClass(), "checkUserCredentials", sunfe));
            }
            throw new AuthenticationException();
        }
    }
}
```

A metódusokat a beépített *LoginManager* hívja, így a *checkUserCredentials()* hívást is megteszi, átadva neki a user és password információt. A metódus persze bármilyen összetett is lehet, de siker esetén visszaad egy *SUser* objektumot. Ekkor a *cfg-bonita-authentication-impl.xml* fájl így alakul:

```
beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/➥
    XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"
        xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.➥
            springframework.org/schema/beans/spring-beans-3.0.xsd">

        <bean id="authenticationService" class="org.bonitasoft.engine.authentication.impl.➥
            AuthenticationServiceImpl">
                <constructor-arg name="identityService" ref="identityService" />
                <constructor-arg name="logger" ref="technicalLoggerService" />
        </bean>

</beans>
```

## Az API részletes Javadoc alapú leírása

A mindenkori Bonita dokumentáció a teljes API-ról tartalmat egy Javadoc dokumentációt, ami elengedhetetlen eszköz akkor, amikor API szinten használjuk. Amennyiben valamilyen belső részletet szeretnénk megismerni, ezt mindig jó szolgálatot tehet.

## Egy saját LoginManager készítése

A 14-7. Programlista egy saját Bonita LoginManager implementáció, amit a telepítésnél említett fájlban így kell bekonfigurálni:

```
#login.LoginManager = org.bonitasoft.console.common.server.login.impl.standard.➥
    StandardLoginManagerImplExt
login.LoginManager = org.cs.auth.ad.MyLoginManagerImpl
```

Az implementáció az *org.bonitasoft.console.common.server.login.LoginManager* interface megvalósítását jelenti.

14-7. Programlista: MyLoginManagerImpl - Egy saját LoginManager

```
1  package org.cs.auth.ad;
2
3  import javax.security.auth.callback.CallbackHandler;
4  import javax.security.auth.login.LoginContext;
5  import javax.security.auth.login.LoginException;
6
7  import org.bonitasoft.console.common.server.login.HttpServletRequestAccessor;
8  import org.bonitasoft.console.common.server.login.LoginFailedException;
9  import org.bonitasoft.console.common.server.login.LoginManager;
10 import org.bonitasoft.console.common.server.login.datastore.Credentials;
11 import org.bonitasoft.console.common.server.login.datastore.UserLogger;
12 import org.bonitasoft.console.common.server.login.datastore.UserLoggerExt;
13 import org.bonitasoft.console.common.server.login.impl.jaas.ConsoleCallbackHandler;
14 import org.bonitasoft.console.common.server.preferences.properties.PropertiesFactory;
15 import org.bonitasoft.console.common.server.utils.SessionUtil;
16 import org.bonitasoft.engine.session.APISession;
17 import org.bonitasoft.web.rest.model.user.User;
18 import org.slf4j.Logger;
19 import org.slf4j.LoggerFactory;
20
21 /**
22  * A test Login manager implementation.<br />
23  * Login will be handle by JAAS context. Appropriate context will be chosen
24  * based on user name value. More exactly on based domain name set as user name
25  * prefix.
26  *
27  * @author Antoine Mottier
28  */
29 public class MyLoginManagerImpl implements LoginManager {
30
31    /** Character use to make separation between domain name and user name */
32    private static final String DOMAIN_USERNAME_SEPARATOR = "\\";
33
34    /** JAAS login context name prefix */
35    public static final String JAAS_LOGIN_CONTEXT_PREFIX = "BonitaAuth";
36
37    /** Logger */
38    private static final Logger LOGGER = LoggerFactory
39          .getLogger(MyLoginManagerImpl.class);
40
41    /**
42     * No customization done here compare to the reference module {@inheritDoc}
43     */
44    @Override
45    public String getLoginpageURL(final long tenantId, final String redirectURL) {
46       final StringBuffer url = new StringBuffer();
47       url.append("..").append(LoginManager.LOGIN_PAGE).append("?");
48       if (tenantId != -1L) {
49          url.append(LoginManager.TENANT).append("=").append(tenantId)
50                .append("&");
51       }
52       url.append(LoginManager.REDIRECT_URL).append("=").append(redirectURL);
53       return url.toString();
54    }
55
56    /**
57     * It's in this method that we implement the logic of using user name
58     * without domain for login module. We use the domain name to select the
59     * appropriate login context.
```

```
60      */
61     @Override
62     public void login(final HttpServletRequestAccessor request,
63             final Credentials credentials) throws LoginFailedException {
64        long tenantId = credentials.getTenantId();
65
66        // We have the user name and password both in the request and
67        // credentials objects
68        // In both case, user name include the domain name
69        // We force the user name to lower case as we did the same in LDAP sync
70        // configuration
71
72        String username = request.getUsername();
73        if (username == null) {
74           throw new LoginFailedException("Username is empty");
75        }
76
77        String requestUsernameLowerCase = username.toLowerCase();
78        // String credentialsUsername = credentials.getName();
79        String requestPassword = request.getPassword();
80        if (requestPassword == null) {
81           throw new LoginFailedException("Password is null");
82        }
83
84        // String credentialsPassword = credentials.getPassword();
85        // We extract the user name by removing the domain name
86        String usernameWithoutDomain = extractUserNameFromFullUserName(requestUsernameLowerCase);
87
88        if ((usernameWithoutDomain == null) || usernameWithoutDomain.isEmpty()) {
89           throw new LoginFailedException(
90                 "Username without domain is empty. Full username was: "
91                       + requestUsernameLowerCase);
92        }
93
94        // We also extract the domain name
95        String domainName = extractDomainNameFromFullUserName(requestUsernameLowerCase);
96
97        if ((domainName == null) || domainName.isEmpty()) {
98           throw new LoginFailedException(
99                 "Domain name is empty. Full username was: "
100                      + requestUsernameLowerCase);
101       }
102
103       LOGGER.trace("Username in request: {}", requestUsernameLowerCase);
104       // LOGGER.trace("Username in credentials object: {}",
105       // credentialsUsername);
106       LOGGER.trace("Password in request: {}", requestPassword);
107       // LOGGER.trace("Password in credentials object: {}",
108       // credentialsPassword);
109       LOGGER.trace("Username without domain: {}", usernameWithoutDomain);
110       LOGGER.trace("Domain name: {}", domainName);
111
112       // Create handler with the user name without domain name as stored in
113       // the AD. the handler will be used to provide user name to LDAP login
114       // module. As user name is stored without domain name in each AD we need
115       // to use here the version without domain name.
116       final CallbackHandler handler = createConsoleCallbackHandler(
117             usernameWithoutDomain, requestPassword,
118             String.valueOf(tenantId));
119
120       try {
121          // Build the appropriate login context name based on tenant id and
122          // domain name
123          String loginContextName = buildLoginContextName(tenantId,
124                domainName);
```

```
125         LOGGER.trace("login context name: {}", loginContextName);
126
127         LoginContext loginContext = new LoginContext(loginContextName,
128               handler);
129
130         loginContext.login();
131         loginContext.logout();
132      } catch (final LoginException e) {
133         LOGGER.error("Error while checking user credential", e);
134         throw new LoginFailedException(e.getMessage(), e);
135      }
136
137      String local = DEFAULT_LOCALE;
138      if (request.getParameterMap().get("_l") != null
139            && request.getParameterMap().get("_l").length >= 0) {
140         local = request.getParameterMap().get("_l")[0];
141      }
142
143      // For all the propagation (including to Engine) with use the name that
144      // include the domain
145      final User user = new User(requestUsernameLowerCase, local);
146      final APISession apiSession = getUserLogger().doLogin(credentials);
147      user.setUseCredentialTransmission(useCredentialsTransmission(apiSession));
148      SessionUtil.sessionLogin(user, apiSession, request.getHttpSession());
149   }
150
151   private String extractDomainNameFromFullUserName(String username)
152         throws LoginFailedException {
153      int indexOfDomainUsernameSeparator = username
154            .indexOf(DOMAIN_USERNAME_SEPARATOR);
155      if (indexOfDomainUsernameSeparator == -1) {
156         throw new LoginFailedException(
157               "Invalid username, separator (\\) between domain name and user name was not found in: "
158                  + username);
159      }
160      return username.substring(0, indexOfDomainUsernameSeparator);
161   }
162
163   private String extractUserNameFromFullUserName(String username)
164         throws LoginFailedException {
165      int indexOfDomainUsernameSeparator = username
166            .indexOf(DOMAIN_USERNAME_SEPARATOR);
167      if (indexOfDomainUsernameSeparator == -1) {
168         throw new LoginFailedException(
169               "Invalid username, separator (\\) between domain name and user name was not found in: "
170                  + username);
171      }
172      return username.substring(indexOfDomainUsernameSeparator + 1);
173   }
174
175   private ConsoleCallbackHandler createConsoleCallbackHandler(
176         String username, String password, String tenantId) {
177      return new ConsoleCallbackHandler(username, password, tenantId);
178   }
179
180   /**
181    * Overridden in SP
182    */
183   protected UserLogger getUserLogger() {
184      return new UserLoggerExt();
185   }
186
187   /**
```

```
188      * Add to JAAS login context common prefix:<br />
189      * - tenant id if provided<br />
190      * - domain name
191      *
192      * @param tenantId
193      *             the tenant ID
194      * @param domainName
195      *             the domain name
196      * @return the Login Context name as it should be configured in JAAS
197      *          configuration file (login-config.xml in JBoss)
198      */
199     private String buildLoginContextName(final long tenantId, String domainName) {
200         StringBuilder loginContextName = new StringBuilder(
201                 JAAS_LOGIN_CONTEXT_PREFIX);
202         if (tenantId >= 0) {
203             loginContextName.append("_").append(tenantId);
204         }
205 
206         loginContextName.append("_").append(domainName);
207 
208         return loginContextName.toString();
209     }
210 
211     private boolean useCredentialsTransmission(final APISession apiSession) {
212         return PropertiesFactory
213                 .getSecurityProperties(apiSession.getTenantId())
214                 .useCredentialsTransmission();
215     }
216 
217 }
```

## Az Internet Explorer 8 működés javítása

Amennyiben IE8-cal használjuk a Bonita Portált, úgy hasznos lehet az alábbi egyszerű Filter, ami force-olja azt, hogy az IE a legmagasabb verziónak megfelelő módban szolgálja ki a kérést, függetlenül a beállításaitól. A 14-8. Programlistán látható filter a HTTP header-re ráteszi *X-UA-Compatible* névvel a *IE=edge* értéket.

14-8. Programlista: SetHttpHeadersFilter filter IE8 számára

```
1  package org.cs.ie8.servlets;
2  
3  import java.io.IOException;
4  import javax.servlet.Filter;
5  import javax.servlet.FilterChain;
6  import javax.servlet.FilterConfig;
7  import javax.servlet.ServletException;
8  import javax.servlet.ServletRequest;
9  import javax.servlet.ServletResponse;
10 import javax.servlet.http.HttpServletResponse;
11 
12 public class SetHttpHeadersFilter implements Filter
13 {
14     // ---------------------------------------------------- Instance Variables
15     /**
16      * The default character encoding to set for requests that pass through this
17      * filter.
18      */
19     protected String encoding = null;
20     /**
```

```
21          * The filter configuration object we are associated with. If this value is
22          * null, this filter instance is not currently configured.
23          */
24         protected FilterConfig filterConfig = null;
25         /**
26          * Should a character encoding specified by the client be ignored?
27          */
28         protected boolean ignore = true;
29         /**
30          * The Microsoft X-UA-Compatible hack
31          */
32         protected String xuacompatible = null;
33 // ---------------------------------------------------------- Public Methods
34
35         /**
36          * Take this filter out of service.
37          */
38         public void destroy()
39         {
40             this.encoding = null;
41             this.filterConfig = null;
42             this.xuacompatible = null;
43         }
44
45         /**
46          * Select and set (if specified) the character encoding to be used to
47          * interpret request parameters for this request.
48          *
49          * @param request The servlet request we are processing
50          * @param response The servlet response we are creating
51          * @param chain The filter chain we are processing
52          *
53 * @exception IOException if an input/output error occurs
54          * @exception ServletException if a servlet error occurs
55          */
56         public void doFilter(ServletRequest request, ServletResponse response,
57                 FilterChain chain)
58                 throws IOException, ServletException
59         {
60 // Conditionally select and set the character encoding to be used
61             if (ignore || (request.getCharacterEncoding() == null))
62             {
63                 String selectEncoding = selectEncoding(request);
64                 if (selectEncoding != null)
65                 {
66                     request.setCharacterEncoding(selectEncoding);
67                 }
68             }
69 //ajout du X-UA-Compatible pour IE8+
70 //aussi present dans /inc/head.inc.jsp
71             ((HttpServletResponse) response).setHeader("X-UA-Compatible", this.xuacompatible);
72 // Pass control on to the next filter
73             chain.doFilter(request, response);
74         }
75
76         /**
77          * Place this filter into service.
78          *
79 * @param filterConfig The filter configuration object
80          */
81         public void init(FilterConfig filterConfig) throws ServletException
82         {
83             this.filterConfig = filterConfig;
84             this.encoding = filterConfig.getInitParameter("encoding");
85             String value = filterConfig.getInitParameter("ignore");
```

```
86          this.ignore = (value == null || value.equalsIgnoreCase("true") || value.equalsIgnoreCase➥
                ("yes"));
87          this.xuacompatible = filterConfig.getInitParameter("X-UA-Compatible");
88      }
89  // ------------------------------------------------------ Protected Methods
90
91      /**
92       * Select an appropriate character encoding to be used, based on the
93       * characteristics of the current request and/or filter initialization
94       * parameters. If no character encoding should be set, return
95       * <code>null</code>. <p> The default implementation unconditionally returns
96       * the value configured by the <strong>encoding</strong> initialization
97       * parameter for this filter.
98       *
99  * @param request The servlet request we are processing
100      */
101     protected String selectEncoding(ServletRequest request)
102     {
103         return (this.encoding);
104     }
105 }
```

A filter konfigurációját így kell beírni a *web.xml* fájlba:

```
<!-- Please insert your web.xml file this part (inyiri@mol.hu -->
 <filter>
        <filter-name>SetHttpHeadersFilter</filter-name>
        <filter-class>com.vc.mm.servlets.SetHttpHeadersFilter</filter-class>

        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>

        <init-param>
            <param-name>X-UA-Compatible</param-name>
            <param-value>IE=edge</param-value><!-- IE=8 -->
        </init-param>

    </filter>

    <filter-mapping>
        <filter-name>SetHttpHeadersFilter</filter-name>
        <url-pattern>/*</url-pattern>
 </filter-mapping>
```