ВИСОКА ШКОЛА ЕЛЕКТРОТЕХНИКЕ И РАЧУНАРСТВА СТРУКОВНИХ СТУДИЈА

**Живојиновић Марко РИН-26/19**

**Размена података у оквиру Oracle BPM архитектуре**

**- примењени истраживачки рад -**



Београд, јун 2021.

САДРЖАЈ:

[1. Увод 2](#_Toc74601445)

[2. *BPM* системи 3](#_Toc74601446)

[2.1. Пословни процеси 3](#_Toc74601447)

[2.2. Компоненте пословног процеса 4](#_Toc74601448)

[3. Софтверске платформе за *BPM* 6](#_Toc74601449)

[4. *Java Enterprise Edition* (*Java EE*) 7](#_Toc74601450)

[4.1. *Java Servlet* 7](#_Toc74601451)

[4.2. *JSF* странице и *Bean* објекти 9](#_Toc74601452)

[4.2.1. *JSF Managed Bean* 9](#_Toc74601453)

[4.2.2. *Enterprise Java Bean* (*EJB*) 11](#_Toc74601454)

[4.2.3. *CDI* бинови (*Context and Dependency Injection*) 12](#_Toc74601455)

[5. *ORACLE ADF* 14](#_Toc74601456)

[5.1. *Controller* слој 15](#_Toc74601457)

[5.2. *View* слој 17](#_Toc74601458)

[5.3. *Model* слој 18](#_Toc74601459)

[*6.* *Oracle BPM* 21](#_Toc74601460)

[6.1. *Oracle Weblogic* сервер 22](#_Toc74601461)

[6.2. Моделовање пословних процеса 22](#_Toc74601462)

[7. Oracle Business Process Workspace 25](#_Toc74601463)

[8. Развој прототипа за размену података 26](#_Toc74601464)

[8.1. Структура апликцаије 26](#_Toc74601465)

[8.2. Структура процеса 26](#_Toc74601466)

[8.3. Дефинисање структуре података 29](#_Toc74601467)

[9. Закључак 32](#_Toc74601468)

[10. Литература 33](#_Toc74601469)

# Увод

Када се говори о имплементацији сложених решења намењеним за функционисање великих пословних система, постоје разне стратегије којима се може доћи до жељеног решења оваквих система. Пре свега, питање је за коју платформу је намењено софтверско решење за рад оваквих система. Данас се грубо према платформи за коју су намњене, апликације могу поделити на стандардне десктоп апликације, које се затим морају инсталирати на сваки кориснички уређај да би се омогућио рад крајњим корисницима, андроид апликације које су намењене првенствено мобилним уређајима и веб апликације. У данашње време све чешће се прибегава изради веб апликација које имају задатак да омогуће рад у датом пословном систему. Један од главних разлога зашто су веб апликације све популарније за коришћење јесте независност оперативног система корисничког уређаја од самих апликација. Све што је неопходно да би се приступило апликацији и користиле њене функционалности јесте веб претраживач који поседује готово сваки кориснички уређај данас.

На почетку развоја веб апликација, потребно је размислити о избору програмског језика, односно технологија (*framework*) које ће бити коришћене за израду апликације. Поред овога, битно је размислити о начину организације и чувања података, најчешће у оквиру одређене релационе базе података. Уколико се одабере овакав правац развоја, програмери имају потпуну слободу у коришћењу одабраних технологија и њиховог комбиновања у циљу израде веб апликација различитих намена. Међутим, уочено је да се често многи пословни системи могу свести на опис активности које корисници изводе у оквиру одређених процеса. Због ове чињенице, многе софтверске компаније су развиле своје BPM(*Business Process Management*) софтверске имплементације које имају намену да олакшају развој решења за разне пословне системе. У оквиру овог рада ће бити описан начин рада у оквиру OracleBPM платформе и поређење са другим BPMсистемима.

# *BPM* системи

Типичан начин рада у оквиру готово било које фирме, без обзира на делатност којом се бави се може описати кроз пословне процесе који се одвијају у оквиру те фирме. BPM служи како би се ови процеси што прецизније описали, и уочила њихова евентуална повезаност, као и комплетан опис свих активности које се могу изводити у оквиру њих. Дакле, BPM као појам није конкретно везан за развој софтверских решења која служе за израду пословних процеса који се одвијају у оквиру неке фирме. BPM представља технику која има своја правила и принципе помоћу којих се дефинишу пословни процеси. Када се говори о побољшању пословања одређене фирме, кроз BPM се не може утицати на побољшање конкретних активности који запослени изводе у оквиру неког процеса. Заправо, кроз BPM се дефинише целокупни пословни систем и његови међусобно повезани елементи. Елементи у оквиру BPM су дакле одређени догађаји који се могу уочити, активности и одлуке. Управо ови елементи су кључни фактори у сваком пословном процесу који одређују његов ток.

## Пословни процеси

У сваком привредном субјекту, фирми, организацији се одвијају процеси који дефинишу пословање. Типични примери процеса који се могу видети у оквиру фирми могу бити:

* Наруџбина – плаћање: Процес започиње трговац, када клијент поручи одређени производ, или закаже услугу, а завршава се када се дати производ испоручи и обави плаћање. Овај тип процеса садржи активности које су везане за верификацију наруџбине, слање нарученог производа, испоруку и уручење рачуна и потврду од стране клијента.
* Жалба клијента – решење проблема: Овај процес започиње у моменту када клијент покрене жалбу због незадовољства пруженом услугом, односно купљеним производом. Процес се не завршава док се клијент, фирма, а у најбољем случају обоје не сложе да је проблем решен.
* Подношење захтева – одобрење: Још једна доста честа врста процеса, који почиње када особа подноси пријаву везану за одређену потребу, који се може завршити одбијањем или одобрењем. Ова врста процеса се може срести у оквиру разних државних служби где грађани подносе захтеве, као што је издавање грађевинске дозволе, на пример.

Како видимо у горе наведеним примерима, пословни процеси су управо оно што се одвија у оквиру фирми и организација када је потребно извршити интеракцију са крајњим корисницима. Начин на који су ови процеси организовани и изведени одређује квалитет услуге који се пружа клијентима, као и ефикасност целокупног пословања једне фирме.

## Компоненте пословног процеса

Како се види у претходном поглављу, пословни процеси се најчешће састоје из више догађаја (*event*) и активности (*activity*). Догађаји се односе на ствари које се у неком моменту догоде, и немају предвиђено временско трајање. Догађаји могу покренути серију активности које следе након њиховог дешавања. На пример, када се деси догађај да клијент поднесе захтев за повраћај новца, запослени има задатак да спроведе активност прегледа описа захтева који је клијент упутио.

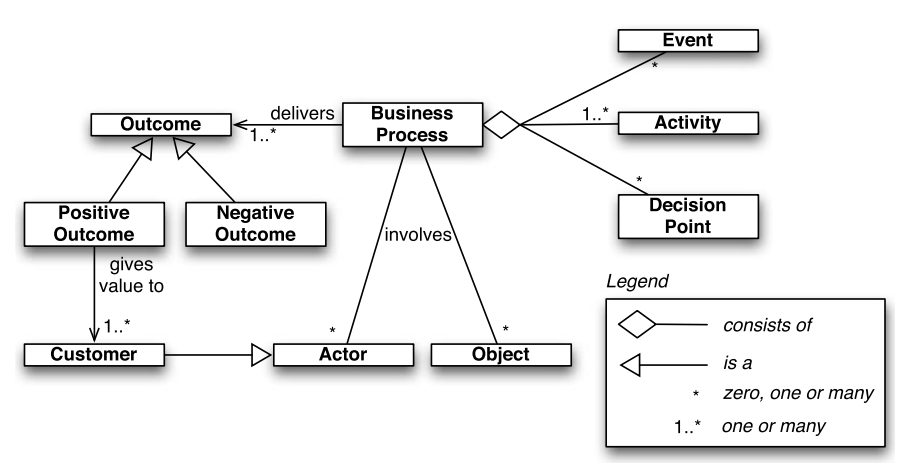
Када је потребно да човек мануелно изврши одређени задатак у оквиру активности, најчешће у смислу одређивања садржаја података који му долазе у датој активности, онда се та активност назива задатак (*task*). На нивоу задатака, потребно је одредити који запослени ће извршити конкретан задатак. Одређени задатак се не мора увек доделити конкретном запосленом, већ се може доделити одређеној корисничкој групи која постоји у оквиру фирме.

Поред догађаја и активности, процеси у главном садрже и тачке одлучивања – *decision points*. Тачке одлучивања представљају „раскрсницу“ у оквиру процеса, односно којим путем ће се наставити извршење датог процеса, најчешће у зависности од вредности неког податка. Често вредност податка који одређује даљи ток процеса бива одређена од стране запосленог који извршава одређену активност.

Процес садржи и одређене чиниоце поред претходно наведених фактора. Ови чиниоци могу бити људи, организације, или софтвер који аутоматизовано извршава задатке постављене пред њега. Процеси дакле садрже најчешће предмете пословања којима се одређена фирма бави, односно њихове описе у виду података, а такође се кроз њих могу размењивати документи који су претходно направљени.

Завршетак процеса у може имати један или више исхода. Уколико се ради о процесу као што је захтев за повраћајем новца од стране клијента, може се десити да се процес успешно заврши (гледано са стране клијента) тако што ће му бити повраћен новац. Могућ исход је и да се деси одбијање од стране фирме, тако да новац неће бити враћен клијенту. Међутим, може се десити и да завршетак једног процеса представља услов за започињање неког новог који је условљен претходним. Ово значи, на датом примеру да се клијент може у неком моменту поново пожалити, а компанија ће имати забележен претходни случај жалбе клијента, па на основу њега у неком моменту одредити ток процеса.

На слици 1. је приказана типична структура једног пословног процеса. Као што је већ речено, један процес се може састојати од неколико догађаја, тачака одлучивања, а макар једне активности. Процес унутар себе укључује чиниоце (људи, групе људи, софтвер...), објекте (документа, подаци...), а на излазу даје неки од могућих исхода који даје одговор крајњем клијенту. Овај скуп компонената представља основну структуру која се може применити на многе видове пословања, односно делатности и помоћу које се могу описати сви различити пословни процеси који се одвијају у оквиру ових делатности.



Слика 1. Компоненте пословног процеса [1]

# Софтверске платформе за *BPM*

Како се види у претходном поглављу, само моделовање пословних процеса је заправо идејно решење како треба пословање једне компаније да се одвија у циљу успешног пословања. Данас постоје бројна софтверска решења за моделовање пословних процеса. У основи, сва ова решења садрже неки алат за дефинисање процеса који ће се одвијати. Њихове генералне карактеристике су на први поглед сличне, али је једно од кључних питања да ли ће одређен систем бити компатибилан са постојећим окружењем. Опште је познато да различити софтверски производи и решења захтевају различиту софтверску и хардверску подршку. Под софтвером се најчешће подразумева оперативни систем, или сервер уколико се ради о веб апликацијама. Свако софтверско решење се базира на одговарајућој технологији односно програмском језику. Фирмама у којима се постојећа решења базирају на *PHP* технологији и *Apache* серверу погодује имплементација BPMрешења које подржава ову врсту технологија. Због овога је битно препознати који тип технологија се тренутно имплементира у оквиру компаније и која BPM платформа је најпогоднија за примену у постојећу инфраструктуру. Такође, BPM платформе се ослањају на одређену релациону базу података у којој се чувају сви подаци који се тичу процеса и других активности. Нека BPM решења подржавају MySQL, нека Oracle, а нека SQL Server базе података. Најчешће се унутар фирми примењује BPM решење које подржава постојећи тип базе података.

Модерна решења BPM платформи се најчешће базирају на сервисно оријентисаној архиви. SOA (*Service Oriented Architecture*) подразумевају концепт развоја сервиса који се могу користити од стране различитих апликација. Ово значи да се пословни процес у оквиру BPM платформе може иницирати, односно контролисати чак из апликација које су независно инсталиране у постојећем окружењу.

Такође, поред алата за моделовање пословног процеса који је потребно одабрати у складу са жељеном BPM платформом, потребно је одабрати језик за моделовање процеса (*modeling language*). Овај језик, слично било ком програмском језику садржи сопствену семантику и синтаксу. Једна од кључних карактеристика BPM алата за моделовање јесте да подржи неки од популарних језика за моделовање процеса, као што су BPMN 2.0, BPQL, BPEL, EPC, UML, XML… Неке BPM платформе подржавају више језика за моделовања, док неке подржавају само један.

Сва модерна BPM решења најчешће садрже и командну таблу – *dashboard*. Овде администратори система могу да виде како се пословни процеси одвијају, као и да извуку одогварајуће анализе. Командне табле имају задатак да омогуће надгледање процеса у реалном времену, као и генерисање извештаја и графикона како би се приказао ток текућих пословних процеса. Командна табла је једна од кључних ствари како би се видела ефикасност обављања пословних процеса, као и како би се размотриле могућности даљег унапређења њихове имплементације и организације.

# *Java Enterprise Edition* (*Java EE*)

Као што је већ речено, данас су софтверске компаније развиле бројна *BPM* решења за дизајнирање односно моделовање пословних процеса. Један од могућих избора је свакако *Oracle BPM* платформа. Ова платформа омогућава развој велике палете различитих апликација које заједно функционишу и размењују податке кроз њихов централни део – пословни процес. Oracle као светски позната компанија већ поседује доста признатих решења у оквиру рачунарских система и информационих технологија. Једна од најпознатијих релационих база података *јесте* Oracle база података, на којој се управо базира и њихова *BPM* архитектура пословних система. Целокупан систем се базира на *Java* програмском језику и пратећим *Java EE* (*Java Enterprise Edition*) веб технологијама.

Да би се ушло у дубља разматрања функционисања овог система неопходно је разумети основе стандардних *Java EE* веб апликација. Када се говори о развоју стандардних *Java EE* веб апликација, могуће је одабрати неки од актуелних сервера који подржавају инсталацију апликација овог типа на њих. Као један од најпознатији свакако јесте *Apache Tomcat* сервер, који се вероватно најчешће и помиње у стручној литератури. У основи, сваки сервер (гледано са софтверске стране) поседује одређена конфигурациона подешавања и језгро које омогућава саму инсталацију веб апликација на њега. *Java EE* технологије су главним делом сконцентрисане на рад са *Java Servlet* класама и *Java Bean* објектима.

## *Java Servlet*

*Servlet* представља класу која омогућава имплементацију комуникације на серверској страни веб апликација. Под овим се подразумева да се у оквиру ових класа дефинишу одговори сервера на захтев клијената. Класе које се користе су дефинисане унутар стандардних *javax.servlet* и *javax.servlet.http* пакета. Као што је већ познато, када се говори о комуникацији преко *http* протокола, најчешће се комуникација реализује преко два типа захтева који су типа *get* и *post*. Да би се ова два захтева имплементирала у оквиру веб апликација реализованих кроз *Java Servlet* технологије, најчешће се дефинишу класе које наслеђују *HttpServlet* класу.

Када желимо да дефинишемо могућност прихватања неког од захтева са клијентске стране, у оквиру класе која наслеђује *HttpServlet* класу, потребно је преклопити жељене методе (у случају *get* захтева то је *doGet* метода, а у случају *post* захтева *doPost*). Ове две методе примају као улазне аргументе два параметра, типа *HttpServletRequest* и *HttpServletResponse*. Ови параметри служе како би се у случају *HttpServletRequest* параметра извукли евентуални подаци који се шаљу, односно у случају *HttpServeltResponse* припремио одговарајући одговор сервера ка клијентској страни. У најпростијем случају, могуће је формирати одговарајући одговор кроз обичан *String* који представља *html* структуру странице која ће се генерисати на клијентској страни.



Слика 2. Основна структура типичне *Java Servlet* класе [10]

Међутим, *Java Servlet* представља једно од првобитних решења компаније *Oracle* у смислу развоја веб апликација и комуникације клијента и сервера. Такође, када се говори о *Java Servlet* технологији, незаобилазно је споменути и .*jsp* (*Java Servlet Pages*) стране. Ове стране могућност генерисања динамичке структуре странице програмски, са стране сервера. У оквиру ових страна, могуће је писати све наредбе које постоје у стандардном програмском језику *Java*. Поред овога, могуће је мешати *html* елементе који ће бити генерисани на страни и *Java* код. На овај начин се *.jsp* страна генерише на серверској страни и враћа клијенту као чист *html* код, који претраживачи једино разумеју.

*Java Servlet* технологије, као и *.jsp* стране се данас користе на веома великом броју веб апликација иако су једне од првих веб технологија које су развијене. Оне нису нестале развијањем нових веб технологија и *framework* решења других компанија односно програмских језика, а и развијањем самих *Java EE*  технологија. Модерније решење, развијено у оквиру *Java EE* платформе јесу *.jsf* стране, односно *Java Bean* објекти, о којима ће бити речи у наредном поглављу.

## *JSF* странице и *Bean* објекти

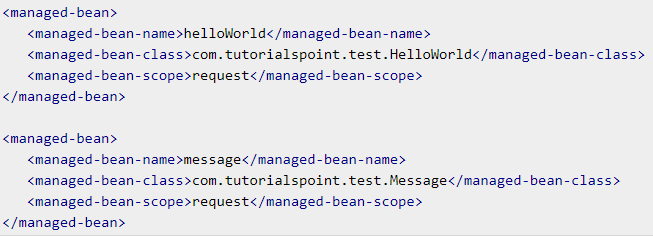
*JSF (Java Server Faces)* странице представљају модерније решење развијено такође у оквиру *Java EE* платформе компаније Oracle*.* Главна разлика у односу на претходне *.jsp* стране јесте што се сада избегава писање *Java* кода у оквиру самих страница, већ се одваја слој логичке обраде података у оквиру *Bean* објеката, а на страници је задатак да обрађене вредности укључи у оквиру своје структуре приликом приказа на клијентској страни.

Генерално, *Java EE* управо уводи појам *bean* – а (у даљем тексту **бин**) када се говори о развоју веб апликациј у оквиру ових технологија. Бин заправо представља објекат за чије инстанцирање је одговоран веб сервер на кога се инсталира веб апликација која има унутар себе декларисане класе које су конфигурисане да буду бин објекти. Бин класе по правилу има атрибуте који су *private* досега видљивости, и сваки од њих има свој пар *geter* и *seter* метода. Има више врста бин објеката који се инстанцирају на серверу:

* *JSF Managed Bean*
* *Enterprise Java Bean (EJB)*
* *CDI Bean (Context and Dependency Injection)*

### *JSF Managed Bean*

*Managed Bean* представља стандардну *Java* класу која је преко одговарајућих анотација, или чешће преко конфигурационе *faces-config.xml* датотеке подешена да има својства *Managed Bean* објекта. У верзији *JSF* 1.2, било је неопходно регистровање *bean* објеката искључиво у оквиру конфигурационог фајла, док су од верзије 2.0 уведене анотације над класама.



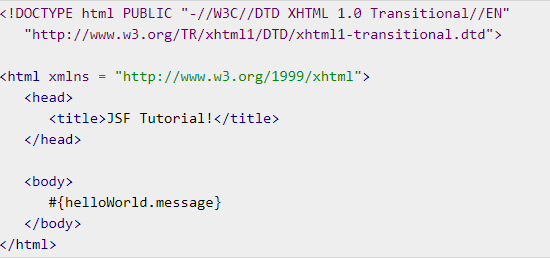
Слика 3. Подешавање *Managed Bean* у оквиру конфигурационе датотеке [11]



Слика 4. Подешавање *Managed Bean* преко анотација [11]

Што се тиче подешавања у оквиру конфигурационе датотеке, ради се о *.xml* подацима где се наводи име *Bean* објекта, класа на основу које ће се инстанцирати, односно његов тип, као и досег (*scope*) важења. Такође, подразумевано, сваки *Bean* има својство *eager* подешено на *false* вредност. Ово својство говори серверу да ли је неопходно да се инстанцира *Bean* на серверској страни пре него што се на њега позове (у оквиру неке странице).

Преко анотација су ова подешавања читљивија, и одмах се виде сва својства једне класе која ће заправо бити репрезентација *bean* објекта. Једно од важних подешавања сваког *Managed Bean* објекта јесте његов досег важења. Овај досег може бити *RequestScoped* – што значи да се *bean* рекреира на сваки *http* захтев упућен са клијентске стране ка серверу. *NoneScoped –* само при првој евалуацији стране се креира, и одмах се уништава. *ViewScoped*  - траје док траје преглед клијента у текућем прозору, при редирекцији на неку другу страну, *bean* се уништава. *SessionScoped* – траје колико год да траје сесија успостављена између клијента и сервера. *ApplicationScoped* – траје док је инсталирана апликација у оквиру које је подешен овај *bean* на сервер.



Слика 5. Пример *.jsf* странице [11]

Што се тиче структуре .*jsf* страница, оне могу садржати стандардане *html* елементе, као и *xml* елементе. Ове технологије се убрајају под *.xhtml* странице, па се могу срести и под овом екстензијом. Најчешће се срећу под екстензијом *.jsf* и представљају структуру *.xml* елемената, који приликом поступа који се назива рендеровање (*render*) генеришу *html* садржај који ће у одговору бити приказан на клијентској страни. У оквиру самог *Java EE* пакета су дефинисани *xml* елементи који ће имати одговарајућу репрезентацију у виду *html* садржаја на клијентској страници.

Неки од типичних елеманта јесу елементи који се тичу комуникације са крајњим корисником, као што су дугмићи – *commandButton*, који морају бити угњеждени у оквиру *form* елемента како би испунили своју функционалност. Дефинисање одговарајућих акција ових дугмића се спроводи кроз атрибуте над њима, који директно указују на одговарајуће методе у оквиру *Managed Bean* објеката на које су повезани. Приликом позива одговарајуће методе, почиње *request* фаза, а *response*, односно одговор сервера се догађа на крају последње линије дате методе. Уколико се дефинише *action* атрибут над *commandButton* компонентом, онда је неопходно да метода врати *String* вредност, која може представљати назив неке друге *.jsf* странице. На овај начин се врши редирекција између страна помоћу ових технологија.

### *Enterprise Java Bean* (*EJB*)

У основи, *EJB* бинови се могу поделити на сесијске (*Session beans*) и оне које контролишу извесни догађаји и поруке (*Message – driven beans*). Оба типа имају своје специфичне намене у оквиру *Java EE* архитектуре. Коришћење *EJB* у оквиру апликација се првенствено своди на слој који се бави радом са подацима. Под овим се подразумева најчешће рад са базом података, који се тиче дохватања, брисања и ажурирања података из одговарајућих табела. Поред ових функционалности, могуће је увек писати делове кода који ће имати задатак да обраде неки специфичан захтев.

*EJB* сесијског типа се користи како би се обавила одређена операција, или дохватили одређени подаци. Захтев за обављањем ових задатака се увек упућује са клијентске стране. У литератури која се бави *Java EE* архитектуром се често среће назив пословна логика (*business logic)*, када се говори о томе шта је задатак једног *EJB* објекта. Управо се под овим подразумевају неке од честих операција које један задатак треба да изврши.

Сами сесијски бинови се могу поделити према дужини трајања на: *stateful, stateless* и *singleton* бинове. *Stateful* сесијски бинови аутоматски чувају тренутно стање података у виду *Java* објеката на страни сервера непосредно након инстанцирања бина. У случају да апликација која позива *stateful* бин нема више потребе за његовим коришћењем, он се уклања. Због овога се овај тип бинова и назива *stateful*, јер има могућност чувања свог тренутног стања, привремено. За разлику од њих, *stateless* бинови немају могућност привременог чувања свог стања, односно статуса података на страни сервера. У моменту позива одређене методе из *stateful* бина, он се креира, извршава се дата пословна логика која је дефинисана у оквиру дате методе, и цео бин се уклања, укључујући све референце ка подацима које је садржао. Уопштено говорећи, када се у литератури објектно оријентисаних програмских језика каже да је одређена класа *singleton* типа, ово подразумева да се увек користи само једна инстанца те класе за остварење потребних функционалности. Због овога су и *singleton* бинови добили управо тај назив. То значи да ће *EЈB* који је проглашен за *singleton* увек имати једну инстанцу, која је задужена за опслуживање свих захтева клијената. Животни век ових бинова је практично док год је постављена апликација са датим бином на серверу.

На слици 6. је приказан основни начин декларисања сесијског *EJB* *stateful* типа. Када се генерално говори о сесијским *EJB* биновима, по правилу је неопходно да сваки бин имплементира два интерфејса, један који ће бити реализован као локални, и он се означава са *@Local* анотацијом. Други интерфејс треба да буде анотиран са *@Remote* анотацијом. У оквиру сваког од њих је потребно дефинисати методе које желимо да позивамо кроз њих. Управо се најчешће сами интерфејси користе у оквиру апликација које желе да користе одређене функционалности једног *EJB* бина. Интерфејс који је означен *@Local* анотацијом, користи се у оквиру исте апликације у оквиру које је сам бин декларисан, док се *@Remote* интерфејс користи у свим другим апликацијама. Један од разлога овог концепта јесте зато што је управо могуће ограничити приступ одређеним методама самог *EJB* бина у зависности да ли се приступа из исте апликације, или друге.



Слика 6. Пример декларисања *stateful EJB* бина [12]

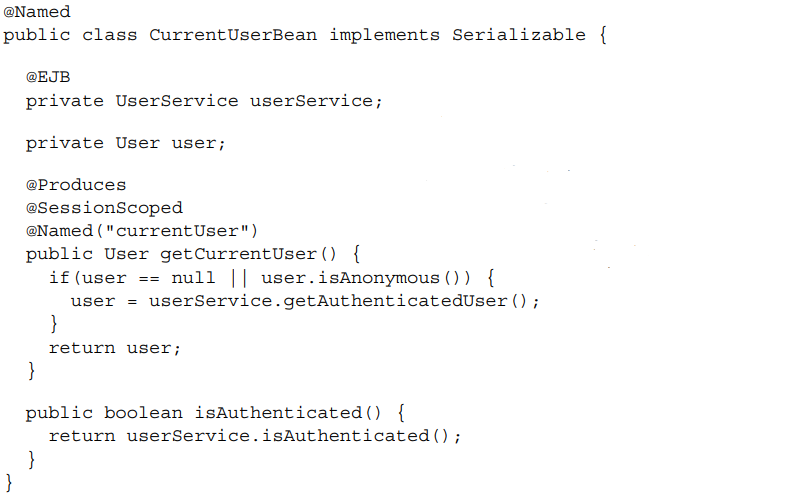
Попут сесијских бинова, тако и *message – driven* бинови имају задатак да изврше одређену пословну логику. Главна разлика у односу на сесијске бинове јесте што се позив метода *message – driven* бинова никада не дешава директно са клијентске стране. Уместо тога, позиви метода ових бинова се догађају када се дешавају одређени догађаји, који онда шаљу поруку преко сервера.

*EJB* бинови представљају једну од кључних компоненти *Java EE* архитектуре управо због тога што имају задатак да се баве самом пословном логиком која заправо диктира сам смисао одређене веб апликације*. EJB* обухвата доста концепата који ће бити обрађени детаљније у следећим деловима рада, на конкретним примерима.

### *CDI* бинови (*Context and Dependency Injection*)

*CDI* бинови практично представљају проширење концепата *EJB* бинова и стандардних *JSF* *Managed* бинова. У оквиру ових бинова, могуће је извршити и одговарајућу пословну логику, али их је такође могуће користити на исти начин као и стандардне *JSF Managed* бинове. *CDI* у погледу рада са подацима не замењује *EJB*, већ управо користи одређени *EJB* који је задужен за дохватање одређених података из базе података и њихово ажурирање. *CDI* бинови такође имају своје векове трајања. За њих, као и за друге претходно наведене бинове, важе анотације *@ApplicationScoped*, *@RequestScoped, @SessionScoped*, међутим поред њих садрже још две могуће опције - *@Dependent* и *@ConversationScoped*.

*ConversationScoped* могућност је уведена када је неопходно да животни век бина траје док има више узастопних захтева са клијентске стране, како би се очували подаци који се тренутно чувају у оквиру њега. *Dependent* опција је подразумевана, и најчешће се и користи, када се врши *Inject* операција овог бина. Да би се постигло његово аутоматско креирање, најчешће се у оквиру неког другог бина наведе анотација *@Inject* над атрибутом типа класе које је *CDI* бин који желимо да користимо на том месту, а сервер изврши инстанцирање објекта ове класе који ће нам онда бити на располагању.

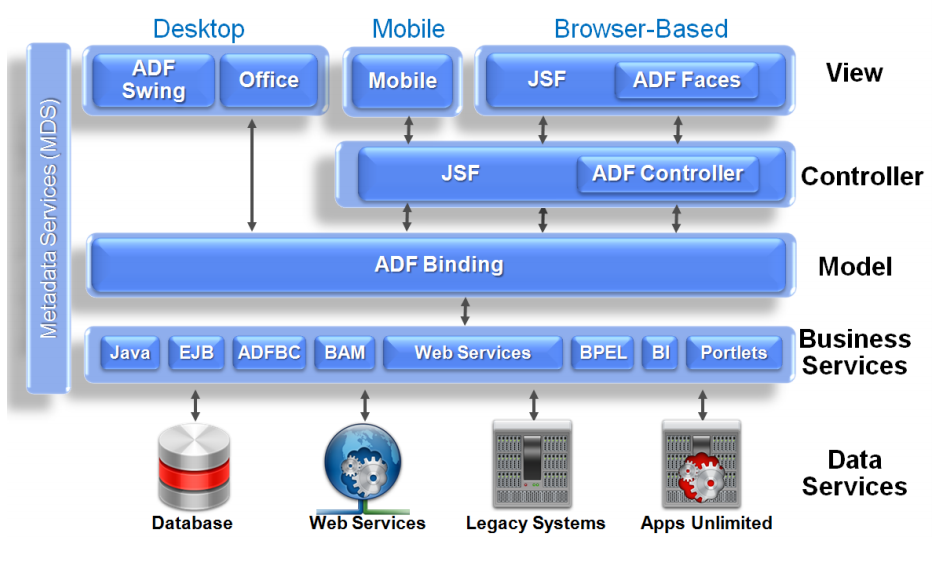


Слика 7. Имплементација *CDI* бина [3]

На слици 7 је приказана једна типична имплементација *CDI* бина који се може инстанцирати преко *@Inject* анотације у оквиру неког другог бина, а такође захваљујући *@Named* анотацији над самом класом, могуће га је користи на начин на који се користе и стандардни *JSF* бинови, директно се реферишући на њега у оквиру неке стране. Овде је приказан такође начин позива методе *getAuthenticatedUser* из *UserService EJB* објекта. Као што се инстанцирају *CDI* бинови преко *@Inject* анотације, на исти начин је могуће постичи инстанцирање *EJB* винова помоћу анотације *@EJB*. На овај начин се *CDI* бинови могу користити као индиректни посредници до података преко *EJB* бинова који се инстанцирају у оквиру њих, а њих саме је могуће користити у другим биновима, као и позивати њихове функционалности директно у оквиру *JSF* страница.

# *ORACLE ADF*

*Oracle ADF* (*Application Development Framework*) представља надоградњу на стандардну *Java EE* архитектуру. Базира са на *Model – View – Controller* парадигми, која је стандардна за већину данашњих веб апликација. Ово значи да се сва логика апликације може структурно поделити на 3 дела: 1) *Model* – одговоран за за дохватање и обраду података који се користе у оквиру апликације, 2) *View* – одговоран за изглед апликације, односно за кориснички интерфејс, 3) *Controller* – представља везу између претходна 2, и одговоран је за ток апликације, односно рутирање захтева клијената. Захваљујући овој архитектури апликација, олакшава се одржавање и поновно коришћење компонената у оквиру апликације. *Oracle ADF* проширује ову парадигму, тако што се изнад постојећег *Model* слоја уводи још један – *Business Services* слој, који представља специфичну имплементацију дохватања података на различите начине, који најчешће обухватају независне веб сервисе, веб сервисе који контролишу ток извршења процеса, *EJB* бинове, итд. Да би било могуће имплементирати све функционалности које нуди *Oracle ADF* радни оквир, потребно је инсталирати развојно окружење *Oracle JDeveloper*, које унутар себе садржи све неопходне библиотеке за развој апликација. О самом окружењу и његовим функционалностима ће бити речи касније.



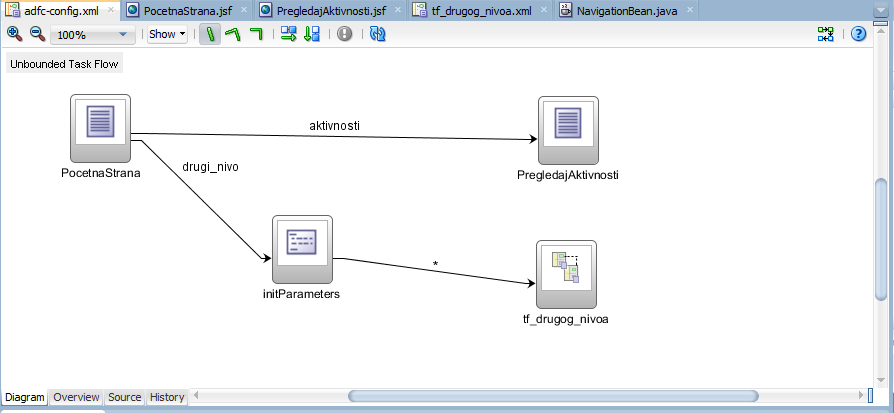
Слика 8. *Oracle ADF архитектура* [\*]

Како се види на дијаграму, *Oracle ADF* омогућава програмерима да одаберу различите технологије у оквиру сваког слоја. Овде се види да *Model* слој представља управо апстракциони слој, и даје флексибилност самом радном оквиру *Oracle ADF*. Заправо, суштина овог слоја јесте да *View* слој, посредством *Controller* слоја може на једнак начин да третира податке који му стижу, без обзира на технологију којом је реализован *Model* слој. У оквиру *Business Services* слоја, подаци се могу прикупљти коришћењем различитих технологија, и овде се одвија објектно – релационо мапирање података, односно трансформисање *Java* објеката, који представљају редове из табеле базе података у саме редове, и обратно. Такође, у зависности од потреба система, подаци се не морају нужно прикљупљати из база података, већ се могу читати из одређених фајлова, као што су *xml, json* или *csv* фајлови, на пример.

## *Controller* слој

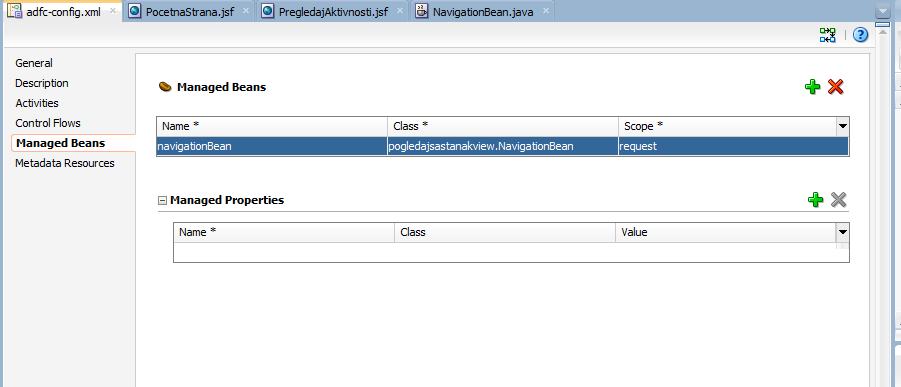
Како је већ речено, *Controller* слој је одговоран за ток извршења апликације, односно на управљање и достављање одговора клијенту, на сваки његов захтев. На пример, када клијент притисне дугме „претражи“ на страници, *Controller* одређује коју акцију ће извршити, и где је потребно да преусмери клијента, односно који одговор ће му бити послат (*View* страница). Када се говори о *Controller* слоју у оквиру *Oracle ADF* радног оквира, могуће је одабрати две опције. Прва опција је стандардан *JSF Controller,* која обухвата рад са стандардним *CDI* и *JSF Managed* биновима. Друга опција је *ADF Controller*, који проширује функционалности стандардног *JSF* контролера. Како се види, код стандардних *JSF* контролера, све стране су практично „на истом нивоу“, односно навигација из једне у другу се одвија експлицитно преко бинова који су декларисани у оквиру *faces-config.xml* конфигурационог фајла, или преко анотација. За разлику од овог принципа, *ADF Controller* одликује присуство такозваних *task – flow* елемената. Ови елементи омогућавају угњеждавање једне или више *JSF* страница у оквиру њих. Поред страница, ови елементи могу да садрже и одређена навигациона правила, као и директне позиве метода из бинова који су декларисани у оквиру њих.

Дакле, за разлику од стандардних *JSF* апликација, где смо имали један конфигурациони фајл у коме су се декларисали сви *Managed* бинови, код *ADF* архитектуре имамо један главни *task – flow*, који се назива *adfc-config.xml* фајл. Подразумевано, он се налази у оквиру *WEB-INF* директоријума. Овај *task – flow* је по својој структури идентичан као и било који други, кориснички дефинисан *task – flow.* Разлика је што је *adfc-config task – flow* увек доступан, на главном нивоу, и он је *unbounded* типа, што значи да му се увек може приступити, без обзира на ком нивоу приступа се тренутни корисник налази. Поред овог типа, остали *task – flow* елементи могу бити и *bounded* типа, што значи да би се приступило компоненатама дефинисаним у њима, неопходно је доћи до њих самих преко неких других страна, односно *task – flow* елемента. Другим речима, за разлику од неких других радних оквира, на пример *Spring* окружења, где се дефинишу руте којима је могуће увек приступити преко експлицитне путање *URL –* а, или стандардних *JSF* апликација где се страницама приступа у формату „ *faces/strana1.jsf“* овде се странице увек угњеждавају у оквиру *task - flow* елемената, где се не може на једноставан начин доћи до сваке странице.



Слика 9. Дефинисање *Unbounded task – flow* елемента

На слици 9. је приказано формирање основног *adfc-config* елемента, који садржи две *.jsf* странице – *PocetnaStrana.jsf* и *PregledajAktivnosti.jsf*. По номенклатури, све компоненте једног *task – flow* елемента се називају активности, а *Oracle JDeveloper* окружење омогућава коришћење једноставних графичких елемената како би се навигација између ових активности. Могуће је комбиновати више активности, и на програмски начин контролисати *outcome* сваке активности, односно генерисану акцију на њеном излазу. Поред страница, активности у оквиру *task – flow* елемената могу бити и позиви метода из претходно декларисаних бинова у оквиру ових *task – flow* елемената, као и позиви других – *bounded task – flow* елемената.

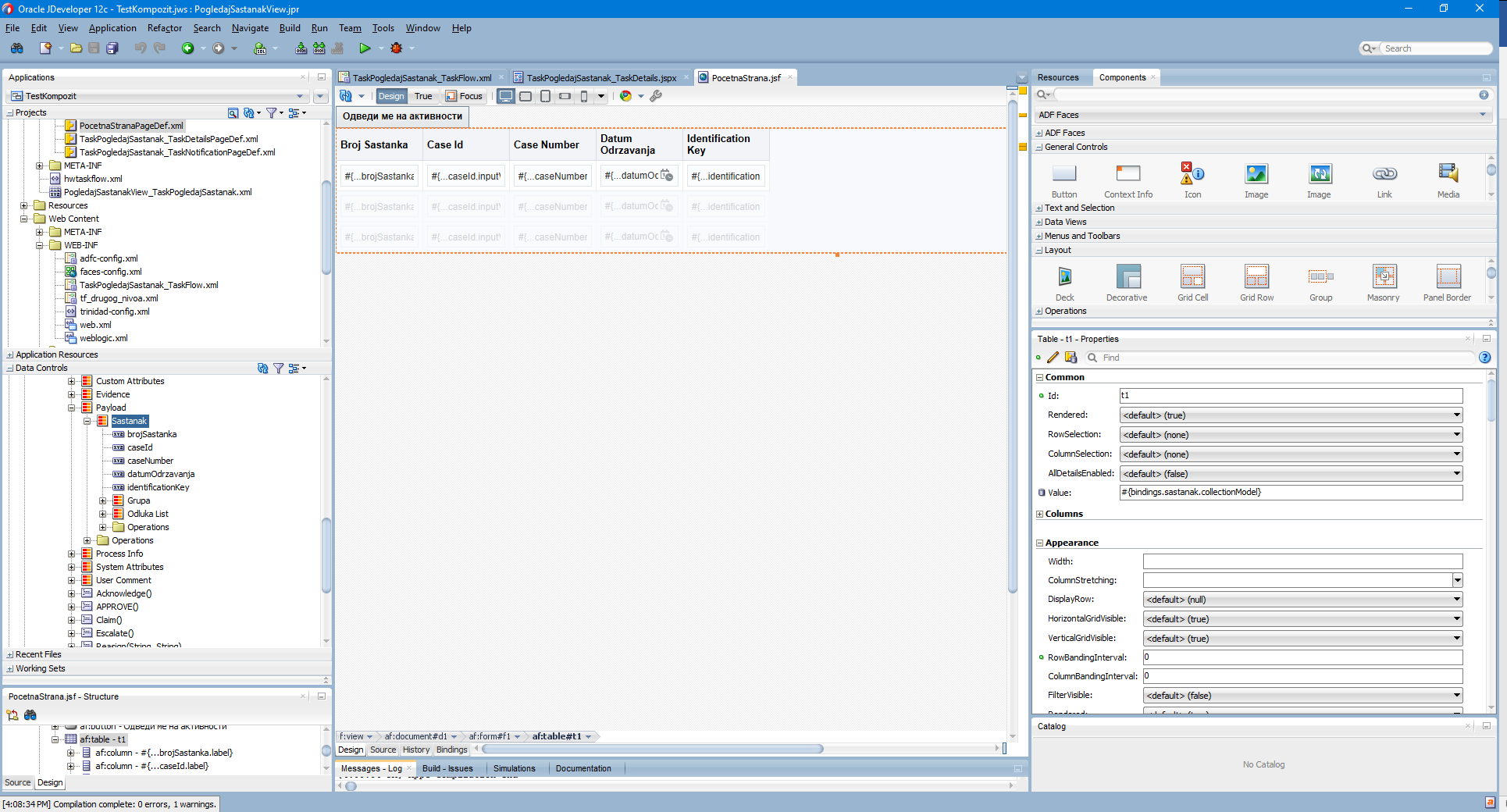


Слика 10. Дефинисање *ADF Managed* бинова

Такође, окружење поред могућности дефинисања тока активности пружа и једноставан интерфејс у коме је могуће декларисати бинове које желимо да користимо у оквиру страница, односно из којих можемо да користимо позиве метода као активности. Дефинише се назив самог бина, односно име преко кога ће бити разрешавано његово позивање, као и класа, односно тип кога ће бити тај бин и наравно његов век трајања. О коришћењу самих *task – flow* елемената ће бити више речи у наставку рада, на конкретним примерима.

## *View* слој

*View* слој представља кориснички интерфејс веб апликације. У претходном поглављу су се спомињале *.jsf* стране. Њихова израда у оквиру *Oracle JDeveloper* развојног окружења је такође максимално олакшана, у том смислу што окружење садржи *WYSIG* (*What You See Is What You Get*) едитор. На овај начин је омогућено једноставно превлачење *ADF Faces* компонената на страну, и њихово слагање у циљу формирања адекватног корисничког прозора. Такође, у оквиру самог окружења, није неопходно увек користити *WYSIG* едиторе (којима ово окружење обилује), већ је могуће увек одабрати и рад над директним изворним кодом, без обзира да ли је у питању израда странице, или чак измену неког *task – flow* елемента.



Слика 11. Имплементација страница

Подразумевано, у оквиру окружења се налази са десне стране палета компонента које су дефинисане за рад у оквиру *Oracle ADF* радног оквира. Компонената које су дефинисане има много, тако да у овом раду неће бити обрађивана свака од њих понаособ. Генерално, палета компоненти садржи готово све стандардне компоненте које могу да се тичу израде корисничких интерфејса. У ово се наравно убравају линкови и дугмад, као и различите врсте панела, табела итд. Оно што чини израду страница олакшаном јесте ослањање на рад са прозором *Data Controls*, који се налази са леве стране у окружењу. Овај прозор представља апстракцију *Model* података који се користе. У главном је то репрезентација *Business Components*, које представљају *ADF* механизам за дохватање и ажурирање података из базе података. Једноставним превлачењем одговарајућег *Data Control* објекта у оквиру странице могуће је генерисати корисничку контролу која ће бити задужена за рад са овим објектом, односно подацима. Најчешће се користе *af:table* компоненте када је у питању рад са вишеструким редовима прочитаним из базе. Сама компонента омогућава регистровање догађаја селекције, измене података, и на једноставан начин је могуће извршити операције као што су *commit* и *rollback* на страни базе података, преко *Data Control* објеката. У описаном случају „превлачења“ *Data Control* објекта на страницу и генерисања одговарајуће компоненте, по аутоматизму ће се одређене датотеке конфигурисати за рад управо са овим објектима. Датотека која конфигурише везу између самих *Data Control* објеката и стране је *DataBindings.cpx* датотека. У њој се кроз *xml* запис описују релације странице и *Data Control* објекта. Поред ње, генерише се за сваку страницу и датотека у формату *naziv\_stranicePageDef.xml*, која садржи детаљнији опис на који се конкретно објекат везује одређен *binding* (веза ка податку). На основу ових датотека, приликом постављања апликације на сервер, разрешавају се везе ка бази података, и интерно се повлаче подаци, без превише напора самих програмера на развоју страница.

Међутим, у случајевима када је неопходно дефинисати сопствену логику повлачења података са различитих извора (није неопходно да се увек подаци читају из базе података), могуће је дефинисати сопствене *EJB* бинове који ће бити одговорни за рад са тим подацима најечешће преко *JPA* (*Java Persistance API*) ентитета. Такође, могуће је дефинисати у оквиру *ADF Managed* бинова позиве на удаљене веб сервисе, са којих ће бити повлачени подаци.

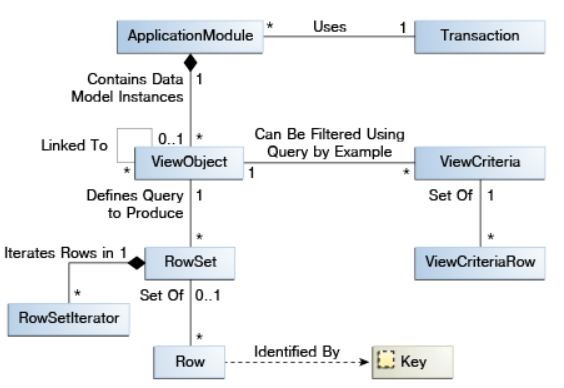
## *Model* слој

Слој *Model* практично представља интерфејс ка *Business Service* слоју. Он има задатак да пружи контролеру преглед на податке на унифициран начин, односно да контролер не мора бити свестан одакле ти подаци долазе, нити којег су типа. Ово значи да практично на нивоу модела подаци увек долазе у виду одређених објеката који су на располагању за даље коришћење, итерацију кроз њихов садржај итд. *ADF Business Components* представља стандардан радни оквир када је реч о раду са базама података у оквиру *Oracle ADF* радног оквира. Није неопходно користити увек овај радни оквир, могуће је повлачити податке и на друге начине, али је концептуално управо овај радни оквир предвиђен за све врсте ажурирања и повлачења података у оквиру *Oracle ADF* радног оквира.

Основне компоненте *ADF Business Components* представљају *Entity* објекти и *View* објекти. *Entity* објекти представљају компоненте које енкапсулирају податке, и другу пословну логику везану за конкретан случај. Што се тиче мапирања табела у бази, то се управо врши кроз *Entity* објекте. Међутим, *Entity* објекти не морају увек бити табеле. Могуће је користити и погледе (*view*) из базе података. Такође, могуће је базирати *Entity* објекте и над *xml* фајловима, међутим онда је потребно прецизно дефинисати структуру који опис нам представља објекат у датом фајлу. Све релације међу табелама у бази се одражавају и на *Entity* објекте који се користе. Релације међу табелама се преводе у асоцијације (*entity associations*), које пресликавају релациона правила која су дефинисана између две или више табела.

Када се говори о *View* објектима, они представљају компоненте које се базирају на *Entity* објектима, и у себи могу да садрже надоградњу информација које садрже *Entity* објекти. Клијенти заправо из *View* објеката увек на крају извлаче конкретне податке. У себи могу да садрже податке како се попуњава одређена табела у оквиру форме, како се обрађује додавање или ажурирање података, прављење листе података која ће да служи као могућ избор код *dropdown* компоненти за избор, итд. Згодна ствар код коришћења *View* објеката је и то што се малопре споменуте асоцијације преводе када се говори о *View* објектима у линкове (*view links*). На овај начин је могуће праћење веза између тренутно одабраног реда у једном *View* објекту и другим, везаним за њега.

Оба типа објеката представљају у суштини *xml* датотеке које заправо дају информацију конкретној апликацији у којој се налазе да је потребно приликом постављања на сервер обезбедити одговарајуће објекте у *runtime* режиму. Ови објекти онда управљају током података, њиховим дохватањем и ажурирањем. Апликациони модул (*Application Module*) представљају имплементацију *ADF Model Data Control* елемената. Они унутар себе садрже *View* објекте и везе међу њима. Заправо крајњи клијент ће видети *Data Control* елемент на онакав начин, какав је дефинисан апликациони модул, који може имати *View* објекте повезане на различите начине, јер *View* објекти не морају нужно увек да представљају идентично стање и релације као што је у бази података на основу које су генерисани.

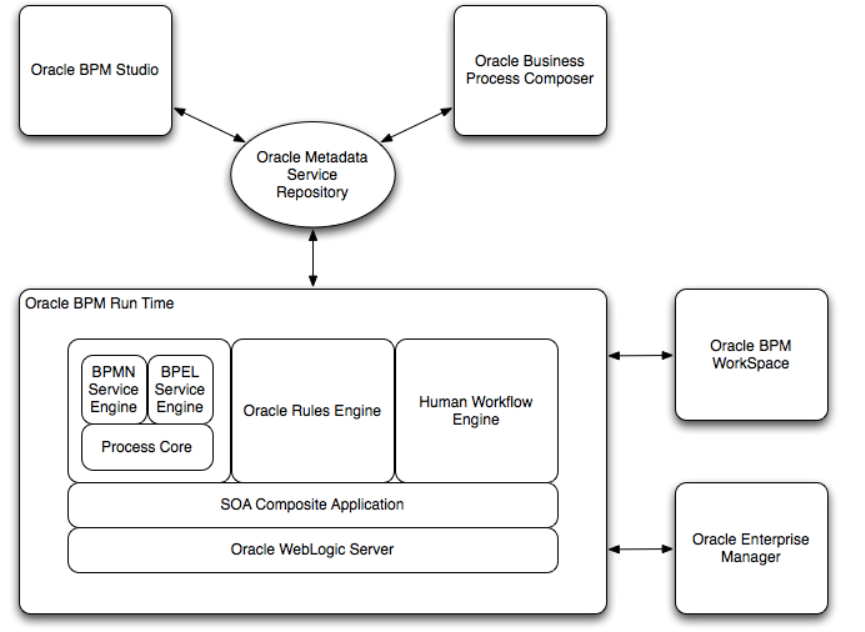


Слика 12. Садржај апликационог модула

Окружење *Oracle JDeveloper* нуди одговарајуће чаробњаке (*wizzard*) за имплементацију компонената *ADF Model* слоја. У овом поступку се наводе подаци о вези на базу података, и наводе се табеле на основу којих ће се извршити генерисање свих објеката. Касније се кроз саме чаробњаке могу дефинисати и садржај и структура *View* објеката, као и апликационог модула. Сви ови генерисани објекти подржавају њихову модификацију у виду програмског допуњавања њиховог садржаја, што значи да је могуће на програмски начин извући податке о сваком генерисаном *View* објекту и повезати тренутан избор реда на начин да одређено дугме на страници посредством *Managed Bean* објекта управља избором реда, на пример.

# *Oracle BPM*

*Oracle BPM Suite* предсавља интегрисану архитектуру намењну за развој, администрацију и коришћење апликација које су вођене пословним процесима. Ова архитектура оомгућава креирање модела процеса базираним на *BPMN* (*Business Process Modeling Notation*)2.0 и *BPEL* (*Business Process Express Language*) језику. Омогућава чак и засебну веб апликацију која омогућава моделовање процеса и пословних правила, као и прављење и модификацију активности које корисници треба да извршавају. Функционисање ових апликација се у великој мери ослања на рад са *Oracle SOA* архитектуром. Да би се моделовали *BPM* процеси, користи се *Oracle BPM Studio*, који је интегрисан у оквиру *Oracle JDeveloper* развојног окружења. У оквиру *Oracle BPM* студија се креирају апликације базиране на процесима. Ове апликације се називају композитне апликације (*SOA composite applications*). Свака оваква апликација садржи известан *Metadata Service* директоријум, у коме се чувају информације о постављеним апликацијама на сервер.



Слика 13. *Oracle BPM* архитектура

На слици 13. су приказане све компоненте које учествују у раду са *BPM* процесима, а на даље у раду ће бити објашњена њихова улога. Централни део ове архитектуре свакако представља сервер на коме се извршава цела архитектура, чији су делови повезани.

## *Oracle Weblogic* сервер

Апликације развијене помоћу *Oracle ADF* радног оквира захтевају *weblogic* сервер како би им било омогућено извршавање. *Oracle* је управо развио *weblogic* сервер као неопходну платформу за рад *BPM* процеса и свих пратећих апликација. Овај сервер у потпуности подржава апликације развијене у оквиру *Java EE* архитектуре, али такође проширује ову архитектуру и подржава све врсте радних оквира које је *Oracle* компанија направила за развој решења за веб апликације. Инсталацијом сервера, подразумевани порт за администрацију самог сервера је 7001. Сама администрација сервера је олакшана кроз интерфејс који представља веб апликација која има подразумевану адресу *http://webogic12c:7001/console*.

Рачунар на коме је инсталиран софтверски сервер *weblogic* не садржи њега самога. Заправо, *weblogic* сервер садржи више софтверских сервера, као што су: *AdminServer, SOA\_Server, UCM\_Server,* итд. Главни сервер који има контролу над другима и преко кога се врши администрација целокупног сервера је свакако *AdminServer*. Преко њега се могу покренути и угасити остали сервери. Основни сервер који је задужен за рад са веб апликацијама, као и композитним апликацијама које садрже пословне порцесе у себи је *SOA\_Server*. Ово значи да би био могућ рад апликација на *weblogic* серверу, минимално морају бити покренути *AdminServer* и *SOA\_Server*.

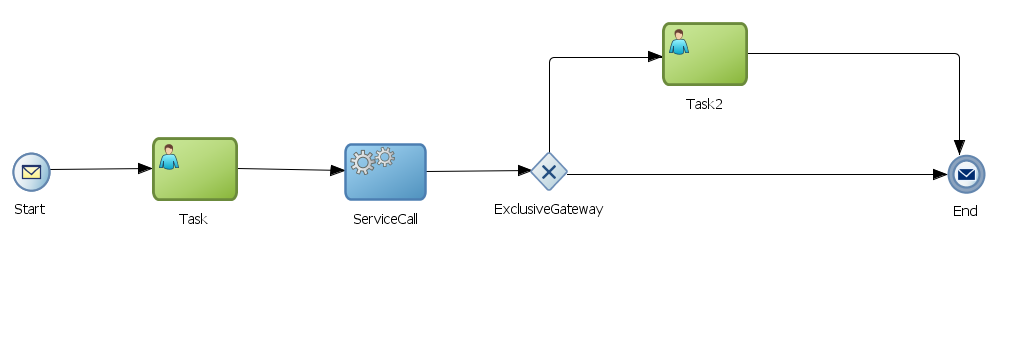
На *SOA* серверу се подразумевано налази инсталирана и апликација неопходна за надгледање управо пословних процеса и осталог дела архитектуре везаног за њих. Ова апликација се назива *Enterprise Manager*, и подразумевано јој се приступа кроз адресу *http://weblogic12c:7001/em*. Кроз ову апликацију је такође могуће пратити и статус осталих сервера, као и вршити одређене администраторске функционалности на целом серверу. Међутим, за разлику од прве наведене апликације, да би *Enterprise Manager* апликација била покренута, неопходно је да буде покренут и *SOA* сервер.

## Моделовање пословних процеса

Како је већ речено, пословни процес се може дефинисати као секвенца повезаних задатака који за свој резултат има неки исход. Како појам „пословни“ сугерише, пословни процеси обично представљају посао који се обавља унутар организација, односно фирми. Управо за ову сврху компанија *Oracle* је развила читаву инфраструктуру за генерализацију пословних апликација које ће бити намењене за најразличитије фирме и организације. Објектима тока (*flow objects*) се називају све компоненте садржане унтуар процеса, а дефинисани су кроз *BPMN*. Језик за моделовање и нотацију процеса садржи унутар себе следеће објекте тока:

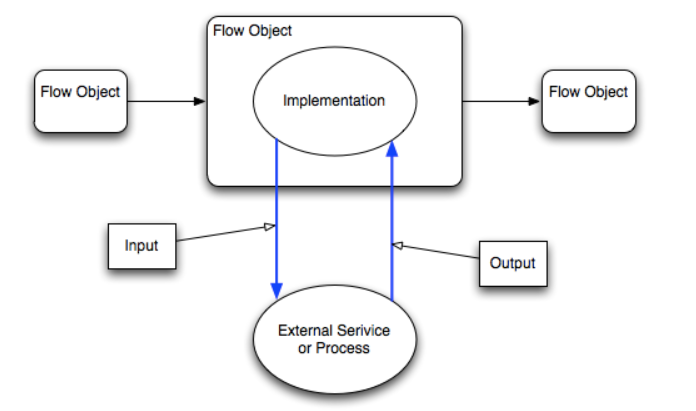
* *Task* (Задатак) - посао унутар процеса који треба да изврши конкретан активиста који се може доделити једном или више корисника
* *Event* (Догађај) – активност која се десила током процеса
* *Gateway* (Рутери) – одређују даљи ток процеса на основу претходног исхода
* *Sequence Flow* (Стрелице) – служе за повезивање осталих објеката

Поред објеката тока, сваки процес садржи податке који се преносе кроз њега, и чији се садржај, односно вредност мења. У оквиру *Oracle BPM* процеса, ови подаци се називају *Data Objects* (*DO*). Док се објекти тока користе како би се дефинисало понашање процеса, објекти података се користе како би се дефинисали и сачували подаци који се користе кроз пословни процес. Дакле, објекти података су променљиве које се дефинишу током поступка моделовања и имплементације процеса. Свака инстанца пословног процеса користи ове променљиве како би сачувала у њима конкретне податке.



Слика 14. Пример пословног процеса

У примеру на слици 14. може се видети како може да изгледа један ток процеса. Идејно решење је да један корисник обавља конкретан задатак, уноси одређене податке, затим се позива конкретан веб сервис, који прихвата све, или неки део података који му се прослеђују од стране корисника, а на основу тих вредности, сервис одређује којим путем ће се процес наставити. Један случај је да оде у горњу грану и да захтева извршење од стране неког другог корисника кроз активност *Task2*. На крају оба правца воде ка крају процеса. Ово је један базичан пример који се може доста усложити, али принцип прослеђивања и обраде података је исти.

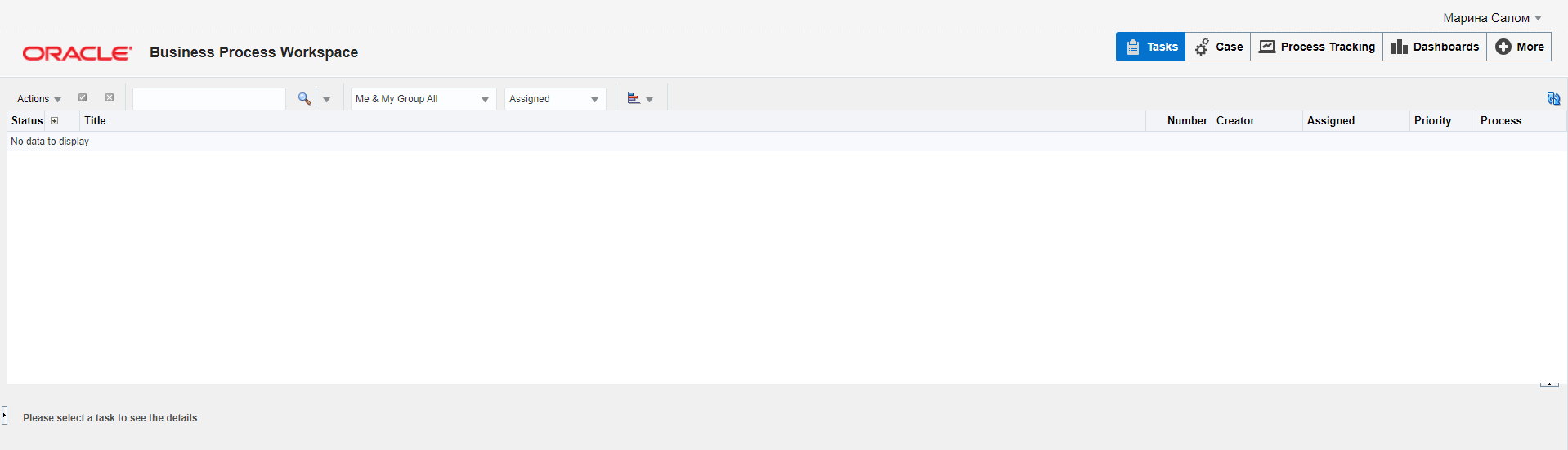


Слика 15. *Data Object* имплементација у оквиру процеса

Што се тиче објеката података који се преносе кроз процесе, они могу бити дефинисани слични као варијабле, у неком од стандардних програмских језика. Потребно је дефинисати њихово име и тип. Већина стандардних типова је подржана, а поред њих је уведен и *Time* тип податка, који највише наличи *java.util.Date* типу података. Могуће је дефинисати и комплексне типове података. Ово се ради угњеждавањем више основних типова у оквиру једног сложеног. Када се све добро дефинише, ови типови се могу користити као такви на даље у току целог процеса. Такође, још један начин за дефинисање комплексних типова који ће се преносити у оквиру процеса јесте преко *xsd* шема података, одакле ће бити препознати комплексан тип са својом структуром, која се може сачињавати из више комплексних типова, као и простих. Генерално, свака активност би требало да има одређен улаз и излаз података. На улазу података се стављају управо објекти података који се шаљу и модификују у току процеса. На излазу се могу добити њихове модификоване вредности, и потребно их је опет повезати са објектима података процеса уколико желимо да се њихове вредности на даље преносе. У процесима је омогућено и поређење и испитивање вредности датих података. Подржани су стандардни изрази проверавања логичких законитости вредности сваког податка. Најчешће се управо на оносву ових вредности одређује даљи ток процеса код *gateway* активности у оквиру процеса.

# Oracle Business Process Workspace

*Oracle Business Process Workspace* представља веб апликацију која служи за рад са *BPM* пословним процесима. Она подразумевано долази са инсталацијом *weblogic* сервера, поред *Enterprise Manager* апликације, као и *console* апликације, за администрацију самог сервера. *Oracle Business Process Workspace* апликација се ослања на провајдере аутентификације, који могу бити разних типова. Подразумевано сам сервер садржи сопствени *EmbeddedLdap* који се користи за читање корисничких група, и самих корисника. Преко графичког интерфејса саме конзолне апликације за администрацију сервера могуће је вршити додавање, измену и брисање постојећих корисника и корисничких група. Поред овог подразумеваног провајдера за аутентификацију, могуће је користити и друге провајдере, као што је на пример *OpenLdap* провајдер – који може представљати независан *Ldap* (*Lightweight Directory Access Protocol*) са кога ће се читати корисници и корисничке групе.



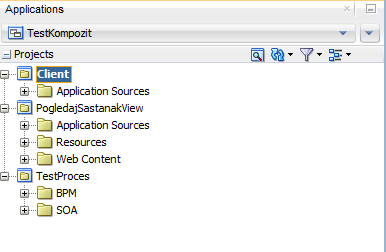
Слика 16. Изглед *Business Process Workspace* апликације

Пријављени корисник у оквиру ове апликације може да види све задатке који су му тренутно додељени. Постоји стандардан мени *Actions* који заправо представља избор могућих акција над самом *HumanTask* активности, која је претходно одабрана у листи задатака. Кликом на одређену *HumanTask* активност, одмах испод листе задатака се отвара *iframe* елемент који садржи форму која је постављена на сервер за одговарајући задатак. Мапирање форме за одговарајући *HumanTask* се врши кроз *hwtaskflow.xml* конфигурациону датотеку, која се налази у оквиру одговарајућег *View* пројекта. Овде се наводи веза ка конкретној *HumanTask* активности у оквиру одређеног процеса. У наставку ће бити приказана конкретна структура ове датотеке на примеру прототипа за тестирање размене података у оквиру *Oracle BPM* архитектуре.

# Развој прототипа за размену података

## Структура апликцаије

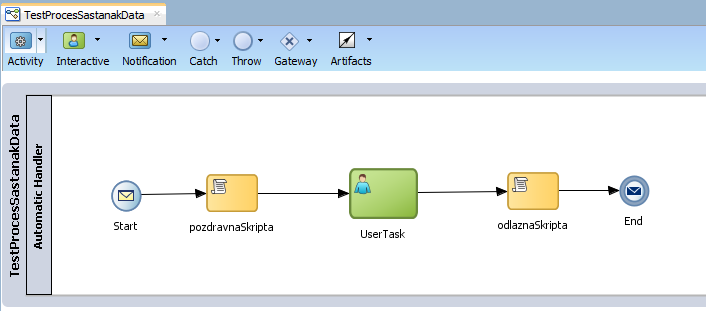
У оквиру *Oracle* *JDeveloper* окружења, могуће је креирати одговарајуће апликације. Када се говори о апликацијама у смислу самог окружења, заправо је реч о радним просторима (*workspace*) у оквиру којих се може наћи један или више пројеката, који су најчешће повезани. У оквиру овог прототипа ће бити реализована три пројекта у оквиру самог радног простора апликације. Први пројекат представља клијент који ће се користи за покретање самог процеса – *Client*. Његова улога је да припремљени скуп података пошаље и позове *SOAP* сервис који врши покретање процеса. Поред овог пројекта, имамо пројекат *PogledajSastanakView*. Овај пројекат служи како би се обезбедила форма за приказ *HumanTask* активности у оквиру *Oracle Business Process Workspace* веб апликације, о којој ће бити више речи касније. Пројекат *PogledajSastanakView* је кључан пројекат, који се бави управљањем током података на *front – end* делу саме апликације. Овај пројекат је одговоран за управљање током извршавања саме апликације, као и за визуелни изглед података добијених кроз процес. Пројекат који се тиче рада са самим процесима представља пројекат *Composite* типа, који у себи садржи све неопходне алате, укључујући и претходно поменути *BPM Studio*, који подржава моделовање процеса. О сваком пројекту и његовој улози ће бити више речи у наредним потпоглављима овог рада.



Слика 17. Структура прототипске апликације

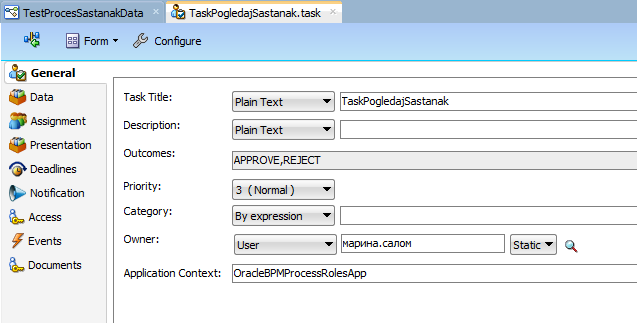
## Структура процеса

У овом раду ће бити приказана могућност рада са подацима у оквиру активности *HumanTask* кроз *Oracle BPM* процесе. Циљ овог рада није удубљивање у поједине активности и објашњавање њихових улога, већ је центар пажње окренут ка подацима који се размењују у оквиру ових процеса. Као што је већ речено, сваки процес има свој почетак и крај. На улазу се најчешће прослеђују одговарајући подаци, који се онда могу иницијализовати на нивоу објеката података (*DataObjects*), који се преносе даље кроз процес. Излаз процеса може диктирати покретање неког другог процеса, или иницирање неких других активности у самом пословном систему. Као прототип намењен за тестирање спремљен је један тест процес који садржи неколико активности. Његова улога је да се прикаже ажурирање података, као и потенцијални ток ових података кроз рад са неком базом података, или позивом одређених веб сервиса. Процес садржи улазну тачку, на коју се прослеђују подаци. Ови подаци се у току даљег рада могу мењати, као и проширивати, у зависности шта је циљ процеса. Тестни процес садржи две активности у виду *Script* *Task* активности. Ове активности су специфичност *Oracle BPM* платформе, и користе *Groovy* програмски језик како би се помоћу њих извршила одређена модификација података. Међутим, могуће је користити их и као тестове који имају улогу да испишу примљене, односно послате податке. Овај тип активности се најчешће поставља пре почетка, односно на крају одговарајуће активности коју желимо да тестирамо.



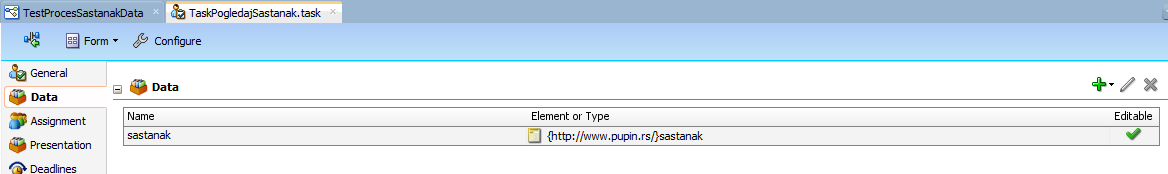
Слика 18. Пример тест процеса

У централном делу тестног процеса се налази *HumanTask*, подразумеваног имена *UserTask*. Ово представља активност коју је неопходно да обави конкретан активиста у оквиру процеса. Додељивање овог задатка корисницима се може реализовати на више начина. Један од начина је свакако фиксно додељивање задатака одређеном кориснику. Управо овај начин ће бити искоришћен у датом прототипу. Поред овог начина, могуће је реализовати доделу конкретном кориснику, односно корисничкој групи преко одређених *Xpath* израза у којима се може на основу вредности тренутно прихваћених података у оквиру процеса одредити додела самом кориснику. Да би се доделио задатак на обављање конкретном кориснику, потребно је унети његов *username*, који се повлачи из репозиторијума корисника са *weblogic* сервера.



Слика 19. Имплементација *HumanTask* активности

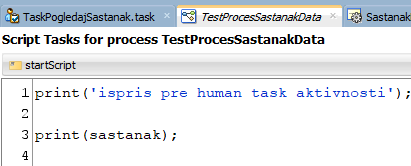
Код имплементације активности у оквиру *HumanTask* активности, у картици *General* је неопходно дефинисати назив конкретног задатка, његов опциони опис, и наравно могуће излазе. Неки од могућих подразумеваних излаза јесу *APPROVE* и *REJECT*. Ово су подразумеване активности због тога што се најчешће у оквиру одређене активности *HumanTask* конкретан човек пита да одреди ток процеса, путем ових *outcome* акција, а тај излаз се може свести управо на одобравање или одбијање, у зависности од примљених података. Такође, *HumanTask* активности се могу категоризовати, али за сада ћемо занемарити ову могућност. Једна од кључних опција јесте власник корисника, односно ко од корисника може да му приступи. Овде ће бити наведен фиксан корисник који има могућност приступу овом задатку.



Слика 20. Подаци коришћени у оквиру *HumanTask* активности

Подаци који се користе у оквиру људског задатка се дефинишу у оквиру картице *Data*. Овде је могуће користити просте типове, као што је *string, integer* итд. Међутим, могуће је дефинисати и комплексну струкутру података, која је претходно дефинисана, најчешће преко одређене шеме. Ова шема мора бити одређеног типа, који мора припадати одређеном именском простору (*namespace*). О дефинисању саме шеме података и могућности њеног креирања ће бити више речи касније. Остале картице неће бити тренутно разматране.

Како је већ речено, *ScriptTask* активности имају задатак да испишу податке на улазу и на излазу података. Њихова имплементација је тривијална, коришћењем *Groovy* програмског језика. Користи се наредба *print* како би се одговарајућа структура података, или фиксни текстуални садржај исписао, најчешће ради тестирања. Испис ових података, као и сваки други испис апликација се чува у *SOA.log* датотеци на *weblogic* серверу. На овај начин је могуће праћење и тестирање развоја апликација постављених на сам сервер.



Слика 21. Коришћење *ScriptTask* активности у оквиру процеса

Након постављања *Composite* апликације на *weblogic* сервер, путем *Enterprise Manager* апликације је могуће праћење стање сваког процеса, и тока података. У оквиру ове апликације се могу видети и *WSDL* опис сервиса који подразумевано покреће сервис. Коришћењем овог *WSDL* податка, могуће је извршити позив из неке друге апликације на њега, и на тај начин проследити одговарајући скуп података приликом покретања процеса. За овај део посла се у оквиру прототипске апликације користи *Client* пројекат који унутар себе садржи логику неопходну за покретање процеса.

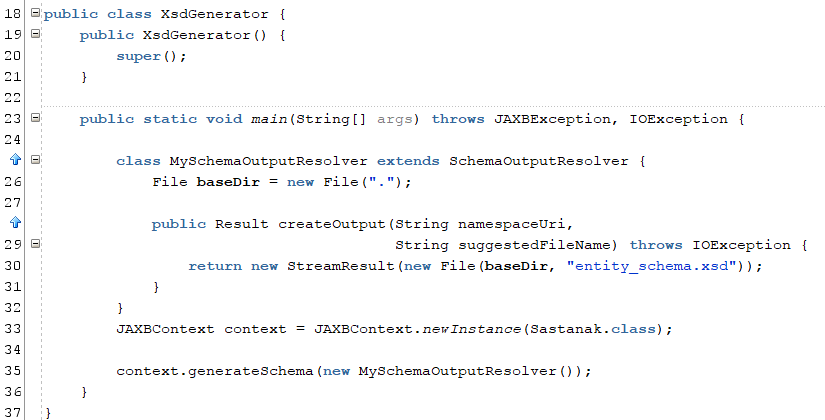
## Дефинисање структуре података

У оквиру самих *Oracle BPM* процеса, могуће је дефинисати *Process Data Objects* – податке који ће бити коришћени на нивоу датог процеса. Ови подаци такође могу бити стандардних простих типова, али могу бити и сложеног типа, најчешће базирани на одређеној *.xsd* шеми. У оквиру шема се дефинише структура сложеног типа, помоћу основних типова, или других угњеждених сложених. Логика рада може бити таква да програмер који моделује одређени процес сам дефинише шеме које ће бити коришћене у току рада. Међутим, у пракси се ово генерисање шема често препушта људима који се баве развојом базом података, због бољег познавања саме структуре података која се може надовезати на структуру базе података. На овај начин члан тима који развија *Model* класе (које се користе и на другим местима у оквиру пројекта), генерише шеме базиране на овим класама како би се омогућило слање и ажурирање ових података у оквиру процеса. Идеја реализације структуре података која ће бити коришћена у оквиру датог процеса јесте генерисање *.xsd* шеме базиране на ентитету у оквиру посебне базе података у којој се чувају подаци везани за апликацију, која се може надовезивати на рад у оквиру пословног процеса. Да би се добила ентитетска класа, коришћен је *JPA* (*Java Persistence API*) радни оквир, који омогућава рад са подацима из базе података на једноставан начин, путем објектно – релационог мапирања. Ово значи да се сви ентитети из базе података преводе у класе у *Java* програмском језику, на основу којих се затим може изгенерисати одговарајућа *.xsd* шема података, која ће бити коришћена у оквиру самих процеса.



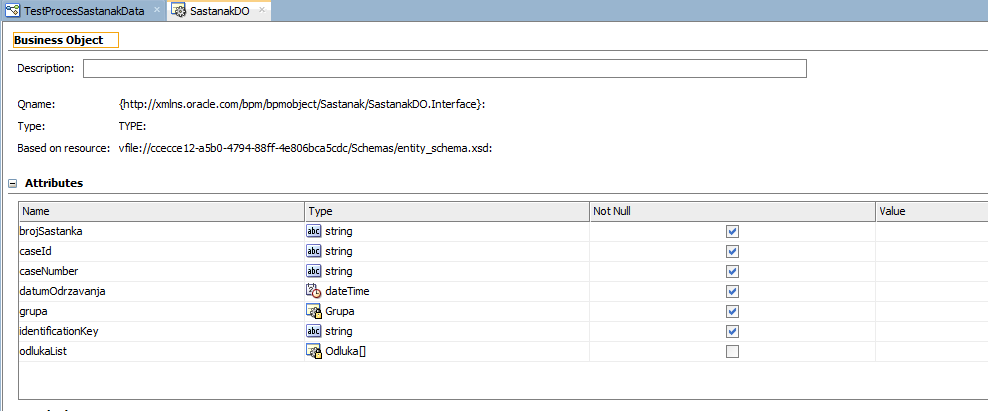
Слика 22. Ентитетска класа базирана на табели у оквиру базе података

*JPA* радни оквир користи анотације како би описао структуру генерисаних података. Овај радни оквир користи *JPQL* (*Java Persistence Query Language*) који омогућава читање података из базе података, коришћењем синтаксе која веома подсећа на стандардан *SQL* језик за рад са базама података. У оквиру ових класа, дефинишу се одређени *NamedQuery* позиви, који се преко имена могу касније позивати. Такође, омогућено је дефинисање сваке колоне табеле на нивоу атрибута класе. *JPA* ентитети се често користе у оквиру *EJB* бинова, који њих користе као објекте кроз које се врши враћање прочитане структуре података из базе података. Захваљујући овој особини, могуће је радити са истом структуром података преко позива самих *EJB* бинова у оквиру разних апликација, али и преко самих пословних процеса кроз *Oracle BPM* архитектуру.



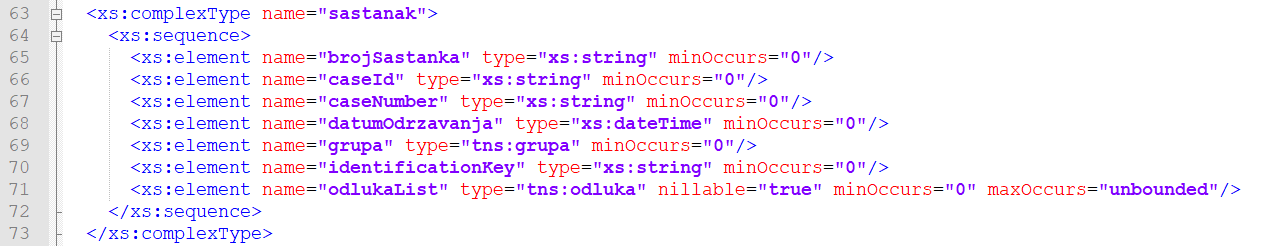
Слика 23. Пример конвертовања општих *Java* класа у *xsd* шеме података

Коришћењем програма за генерисање *.xsd* шема, могуће је добити шеме базиране на класама, које могу бити и *JPA* ентитетске класе. Програм користи *JAXBContext* класу како би на основу прослеђеног типа класе генерисао одговарајућу шему. Затим се ове шеме могу укључити у оквиру *Composite* пројекта који садржи пословне процесе. Да би овај пројекат могао да препозна генерисане *.xsd* шеме, неопходно је поставити их у оквиру директоријума *TestProces\SOA\Schemas.* На овај начин ће ове шеме бити препознате када буде било потребно генерисање *DataObject* података на основу њих.



Слика 24. Преглед *Business Object* податка генерисаног на основу шема

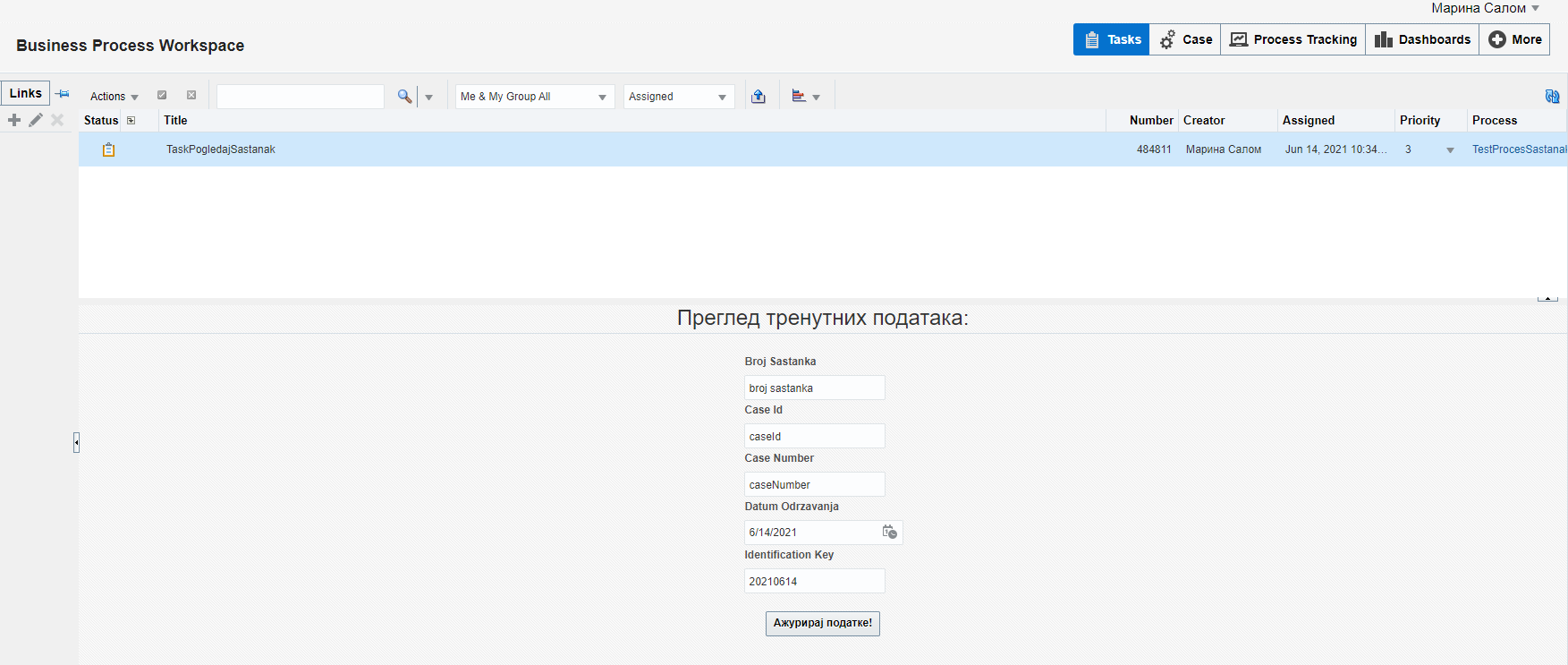
У оквиру окружења, може се проверити генерисани *BusinessObject* на основу шеме. Одговарајући типови података се правилно мапирају, као и *not null* вредности. Овде се види на којој шеми је базиран конкретан *BusinessObject* који ће бити коришћен даље у раду у оквиру самог пословног процеса. На примеру ће бити приказана могућност конверзије овог објекта у стандардан *Java* објекат, над којим је онда могуће вршити позиве свих стандардних *Java* метода и операција. Такође, битно је приметити аутоматску конверзију *dateTime* типа податка, који се извлачи из *xsd* шеме која је генерисана.



Слика 25. Генерисана *xsd* шема података базирана на ентитетској класи

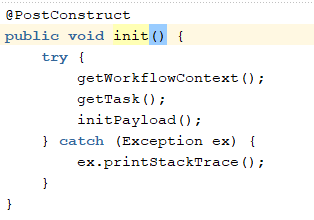
## Реализација *view* пројекта

У оквиру *Oracle JDeveloper* окружења, постоје одговарајући чаробњаци (*wizzards*) који омогућавају аутоматско генерисање *view* пројеката за одговарајући *HumanTask*. На овај начин, *view* пројекат повлачи кроз *Data Control* објекте све информације о подацима који се користе у оквиру процеса, и могуће је користити стандардне *ADF* контроле за рад са овим подацима. Главни део саме предефинисане структуре података која се прослеђује кроз процесе се налази у оквиру *Payload* сложеног елемента у оквиру *Data Control* панела. Он унутар себе садржи све кориснички дефинисане сложене објекте, који су доступни на нивоу рада над *HumanTask* активностима. Поред ових података, овде се налазе и одговарајуће позиве метода, које су стандардне за рад на нивоу *HumanTask* активности, као што су све методе генерисане за позиве *Approve*, или *Reject.* Такође, овде се налази значајан позив методе *update()*, која служи како би се у моменту њеног позива ажурирали подаци дохваћени у тренутном моменту приступа активности *HumanTask*. Практично, све ово значи да нам је потребно да имамо увек један *view* пројекат који ће бити везан за сваку *HumanTask* активност.



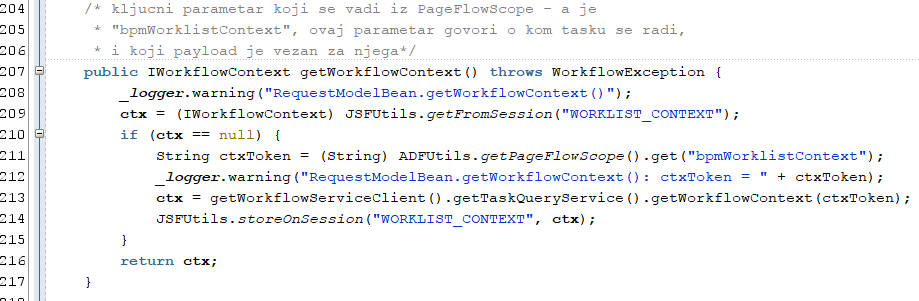
Слика 26. Приказ постављене форме у оквиру *HumanTask* активности

Ажурирање и приказ података у иницјалној верзији ове форме је увек могуће урадити кроз стандардне *Oracle ADF* *binding* методе, које се везују за саме *Data Control* објекте. Међутим, *Oracle ADF*, као радни оквир који је направљен са циљем да се крајњим програмерима максимално олакша рад са подацима има знатна ограничења у виду приказа података на извесне нестандардизоване начине. Ово значи да овај радни оквир није довољно флексибилан када год имамо потребу да изађемо ван стандардних датих контрола и механизама за коришћење података. Због овога се може користити *api* који нам може знатно олакшати рад са подацима који се користе у оквиру процеса. Његова имплементација се тиче параметара који се прослеђују у *task – flow* елемент који представља улаз у овај *view* пројекат. Ови параметри се садрже у *task – flow* компоненти која се аутоматски генерише за овај *HumanTask*. Параметри који се примају у оквиру овог елемента, такође се аутоматски попуњавају приликом његовог генерисања, а значајни су за коришћење и даљи рад у виду превођења дохваћених *DataObject* структура у *Java* објекте са којима је у великој мери олакшан рад. Два кључна параметра која се прослеђују су свакако *bpmWorklistContext* и *bpmWorklistTaskId*. На основу ова два параметра могућ је дохватити све тренутне вредности података које су везане за одређени *HumanTask*, као и ажурирати њихове вредности. Када је ово имплементирано, први корак у конверзији података прихваћених у процесу јесте формирање одговарајућег *ADF Managed* бина, који ће бити одговоран за чување података који су прочитани. Најчешће је овај бин *PageFlow* досега, што значи да ће подаци у оквиру њега бити доступни док год се не напусти *task – flow* у оквиру кога је он декларисан. *ADF Managed* бинови наслеђују од стандардних *JSF* бинова анотацију *@PostConstruct*, којом се може означити одговарајућа метода за коју желимо да се изврши одмах након инстанцирања самог бина на серверској страни. У случају конверзије података у стандардне *Java* објекте, доступне за рад над њима у оквиру странице, могуће је дефинисати *init* методу у којој ће се одвити дохватање самих података.

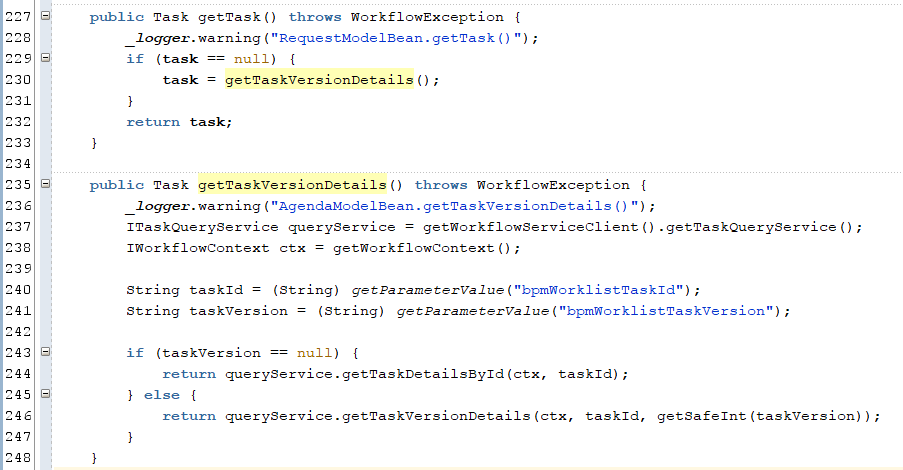


Слика 27. Имплементација *init* методе

Како се види, метода која има задатак да иницијализује податке позива *getWorkflowContext* методу која има задатак да за параметар који се дохвата из *PageFlow* досега самог *task – flow* елемента иницијализује *bpmContext* на основу кога ће се читати подаци везани за *Task*, односно за саме податке касније.

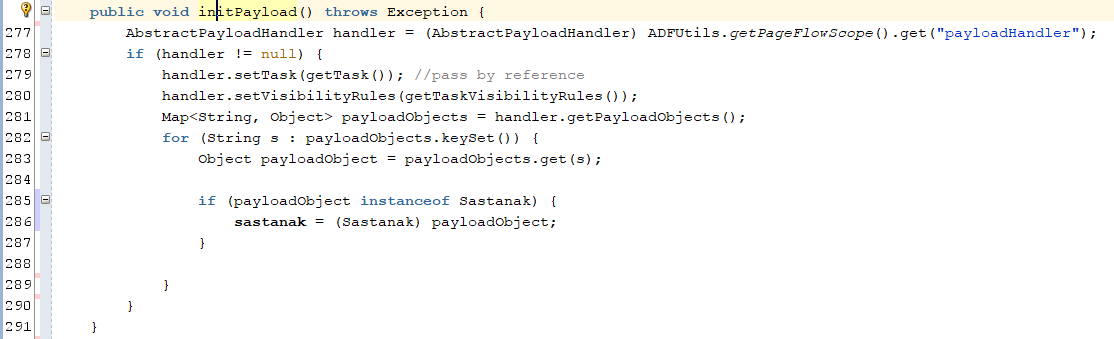


Слика 28. Имплементација *getWorkflowContext* методе



Слика 29. Имплементација *getTask* методе

Након успешне иницијализације контекста, неопходно је дохватити сам *Task* објекат, који је дефинисан у оквиру пакета *oracle.bpel.services.workflow.task.model* и суштински унутар себе садржи све неопходне податке о самој *HumanTask* активности коју је неопходно извршити. Користи се позив метода које враћају за прослеђене параметре такође из *task – flow* компоненте одговарајуће класе за рад са самим *Task* објектом, из кога је могуће онда прочитати податке. На овај начин су обезбеђени сви предуслови за позив методе *initPayload*, која управо има задатак да конкретне податке дохвати и трансформише их у *Java* објекте, ради касније лакше манипилације и рада са њима.



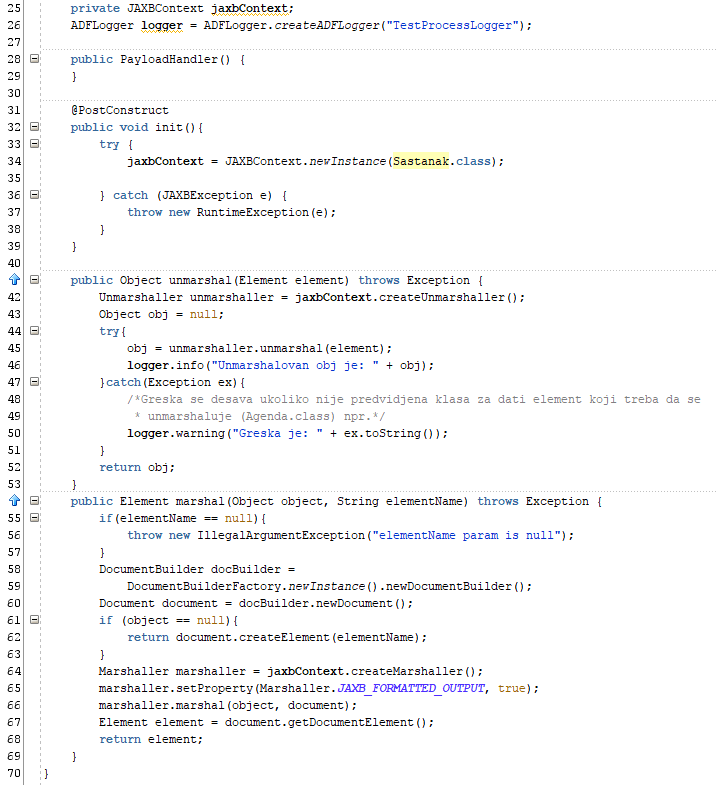
Слика 30. Имплементација *initPayload* методе



Слика 31. Имплементација *getPayloadObjects* методе

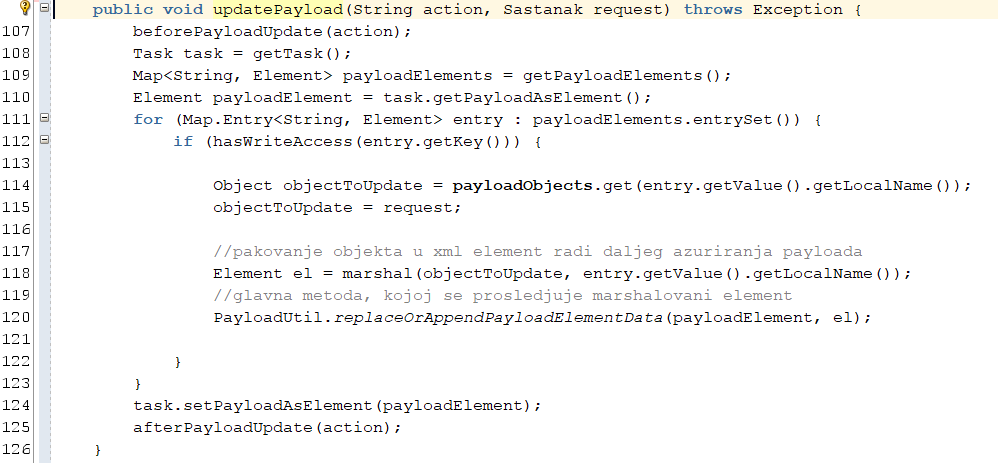
Како се види у коду, у оквиру позива методе *initPayload*, дохвата се претходно декларисани *payloadHandler* бин који унутар себе има кључну методу *getPayloadObjects*, која управо повлачи *Java* објекте, који су претходно добијени. Пошто се на овом месту може наћи више типова објеката, потребно је проверити тип свакога, и онај који желимо сместити у оквиру одговарајуће променљиве, која на крају заправо и може бити атрибут самог бин објекта. С обзиром да тренутна имплементација подразумева коришћење класе *Sastanak*, која се користи као класа демонстрација на прототипу, потребно је предвидети конверзију у њен тип.

У оквиру методе *getPayloadObjects* је могуће предвидети и контролисати дохватање самих објеката кроз *payloadObjects* атрибут, који представља *HashMap<String, Object>* типа. Идеја је да се у оквиру овог објекта чува за назив сваког дохваћеног податка његово име, како је дефинисано у оквиру самих процеса, и његова вредност, овог пута као *Java* објекат. Како се види, ова метода је дефинисана у класи *AbstractPayloadHandler,* међутим класа *PayloadHandler* наслеђује њу и допуњује је са неопходним функционалностима у виду *JAXBContext* механизма конвертовања *xml* садржаја у *Java* објекте и обратно. *PayloadHandler* класа је класа истоименог бина који је одговоран за рад са самим *payload* подацима. Да би била омогућена исправна трансформација *xml* објеката, који у таквом облику путују све време кроз процесе у *Java* објекте, потребно је код *JAXBContext* иницијализатора навести све жељене класе које заправо одговарају коришћеним *xsd* шемама за генерисање података.



Слика 32. Имплементација *PayloadHandler* класе

Кључна класа *PayloadHandler* поред претходно наведеног дефинисања коришћења класа у оквиру процеса конверзије *xml* садржаја у *Java* објекте, и обратно садржи управо методе *unmarshal* и *marshal*. Прва се користи када је потребно од *xml* елемента добити његову репрезентацију у виду *Java* објекта, а друга када је потребно од *Java* објекта направити *xml* елемент, са одговарајућим именом. Позивом ових метода обезбеђује се синхронизација података у моменту потребе ажурирања самих података.



Слика 33. Имплементација *updatePayload* методе из класе *AbstractPayloadHandler*

Метода *updatePayload* као аргументе је дефинисана да прими *action*, у случају да је потребно описати одређену акцију на основу које се врши ажурирање података, међутим ово за исход има само евентуалан испис порука у *log* датотеци на серверу. У општем случају је довољно проследити ажурну верзију објекта над којим су се вршила ажурирања кроз кориснички интерфејс, а то је *request* објекат типа *Sastanak*. Овај објекат се позивом претходно описаних метода претвара у *xml* садржај и шаље на ажурирање даље кроз претходно описане методе. Такође, да би се успешно извршило ажурирање самих података, неопходно је позвати и методу *updateTask*, над *taskService* објектом који је претходно дохваћен, на основу *bpmContext* параметра, како би се у следећем поступку видео ажуриран садржај објекта.

# Закључак

*Hadoop* и његови алати захваљујући својим особинама омогућавају одличан начин за складиштење и обраду великих количина података. Већ су развијена бројна решења на основу *Hadoop* система, а у овом раду је приказана једна могућа варијанта за употребу овог система. Сама анализа сентимената представља веома широку грану за истраживање. Битно је имати довољно добре библиотеке и правилно их употребити у оквиру *MapReduce* функционалности која је обезбеђена у *Hadoop* систему. Могућа надоградња решења приказаног у овом раду би се могла огледати у додавању *HBase* базе података за складиштење резултата анализе, као и у проширивању броја чворова, односно другачијег конфигурисања самог *HDFS* система. Програмски језик *Java* има огромну заједницу програмера који свакодневно развијају нове библиотеке које пружају начине за интеграцију са разним технологијама у области информационих технологија, па тако и *Hadoop* алата. Због тога што је *Hadoop* отвореног кода, постоји читава заједница његових програмера који стално раде на његовом усавршавању и надоградњи постојећих верзија његовог екосистема. Како се количина података свакодневно увећава, потреба за *Hadoop* системом и другим алатима за брзо и ефикасно обрађивање података ће сигурно све више расти у будућности.

# Литература

[1] M. Dumas, M. La Rosa, J. Mendling, H. A. Reijers „Fundamentals of Business Process Management“, 2018.

[2] <https://www.scirp.org/journal/paperinformation.aspx?paperid=88720>, приступано 20.05.2021

[3] D. Panda, R. Rahman, R. Cuprak, M. Remijan „EJB 3 in action“, 2014.

[4] <https://www.tutorialspoint.com/servlets/index.htm>, приступано 20.05.2021.

[5] <https://www.tutorialspoint.com/jsf>, приступано 25.05.2021.

[6] <https://www.tutorialspoint.com/ejb/> , приступано 25.05.2021.