FASRBPS 4.0 Technical Specification Document

Revision History

Note: The revision history cycle begins once changes or enhancements are requested after the document has been baselined.

| Date | Version | Description | Author |
| --- | --- | --- | --- |
| 04/02/2014 | 1.0 | Documented created for Release 4.0 | vafscgopalk |
|  |  |  |  |

# Purpose:

This document provides high level description of changes made for RBPS 4.0 for Process Improvement, CRs 531, 537, 553, 555 and 558.

# Components changed:

Logical changes for Process Improvement were mainly in RbpsWS, RbpsRepository, UserInformationWSHandler and RbpsServiceFacade. Remaining Process Improvement changes were correcting hundreds of reference issues with RbpsRepository, CommonUtils, LogUtils and SimpleDateUtils.

* **RbpsWS** 
  + Modify to exit the process after processing x number or claims. This threshold will be saved in rbps.properties file. Method processRbpsAmendDependency accepts currentProcess # and totalProcessCount as parameters. Creates a file named RBPSBatchProcess\_x (where x is currentProcess #) if ti does not exists and continue with claim process. If file exists, exits out of RBPS. Once claims processing is complete for the processor, file RBPSBatchProcess\_x will be deleted before exiting RBPS.
  + Methods writeCurrentProcessFile and deleteCurrentProcessFile were added to add/delete process file.
* **RbpsServiceFacade.java**
  + Modified executeAll() method to accept currentProcess, totalProcesses, and RbpsRepository as parameters.
  + getAndProcessClaim() method was modified to call userInformation service and process claim for numberOfClaimsPerBatch times. UserInformation service will be called to get the next claim until count equals numberOfClaimsPerBatch or there are no more claims in the queue. This is done for each processor (thread).
  + numberOfClaimsPerBatch is the maximum number of claims that will be processed by a processor.
  + Method addJournalEntry was modified to display correct time stamp for each claim processed.
* **UserInformationWSHandler:** 
  + Modified to pass the processor information (ex. x number of total\_processor). This will be used in logical grouping of claims by file number to be returned for processing. The veteran’s participant id will be used in the group logic.
* **RbpsRepository**
  + Inversion of Control was removed for RbpsRepository and is now being passed as a parameter for all method calls to ensure thread safety.
  + CommonUtils reference was removed from RbpsRepository.
* All classes that referenced RbpsRepository were modified to accept repository as a parameter to the respective method call. RbpsRepository is not injected into classes any more.
* **CommonUtils.**
  + CommonUtils was made static.
* **LogUtils**
  + LogUtils was made static.
* **SimpleDateUtils.**
  + SimpleDateUtils was made static.
* All classes that referenced CommonUtils, LogUtils and SimpleDateUtils were changed to make them static references.
* **rbps-core-spring-config.xml**
  + Bean definitions scope in rbps-core-spring-config.xml were changed to prototype except for wsClientUtil and rbpsApplicationDetails.
* **rbps-ws-client-spring-config.xml**
  + Bean definition scope in rbps-ws-client-spring-config.xml were changed to prototype.
* **BEPlog4j.properties**
  + log4j.appender.rbps = org.apache.log4j.DailyRollingFileAppender
  + log4j.logger.gov.va.vba.rbps=DEBUG,rbps
* **FCDRPopulator:** 
  + New method reviewRatingDate was added to check if promulgation date returned was within one year of claim date, when no FCDR could not be determined.
* **GeneratePdf:**
  + getDependentFromMap and responseToSummary were modified to handle “No matching Award Line/Reason grantDenial.”
* **AwardSummaryBuilder**
  + Method assignDependentInfo was modified to handle No matching Award Line/Reason grantDenial.
* **GenericLetterGeneration**
  + Method generateRbpsLettter was modified to throw exception if letter could not be generated after creating Award Print.

**Code Changes**

* **RbpsWS**

@WebMethod(operationName = "processRbpsAmendDependency")

**public** String processRbpsAmendDependency( @WebParam( name = "currentProcess" ) **final** String currentProcess, @WebParam( name = "totalProcessCount" ) **final** String totalProcessCount ) **throws** RbpsWsException{

RbpsRepository repo = **new** RbpsRepository();

Thread.*currentThread*().setName("currentProcess: " + currentProcess );

startTime = **new** Date();

String retVal = "";

String className = **this**.getClass().getSimpleName();

String methodName = **new** Exception().getStackTrace()[0].getMethodName();

**boolean** currentlyProcessing = **false**;

CommonUtils.*log*( *logger*, "\n" + RbpsConstants.*BAR\_FORMAT* + "\n\n \*\*\* Entering " + className + "." + methodName + " : " + RbpsConstants.*VERSION* + "/" + Thread.*currentThread*().getName() + "\n\n" + RbpsConstants.*BAR\_FORMAT* + "\n");

CommonUtils.*log*( *logger*, String.*format*("currentProcess %s, totalProcessCount %s ", currentProcess, totalProcessCount ) );

**boolean** inDelete = **false**;

**try** {

// if ( ! safeToStartProcessing() ) {

//

// utils.log( logger, "Some other thread is processing claims, returning" );

// return "RBPS is already running, so exiting..";

// }

String fileWriteStatus = writeCurrentProcessFile( currentProcess, totalProcessCount);

**if** ( fileWriteStatus.trim().startsWith("Error") ) {

**return** fileWriteStatus;

}

currentlyProcessing = **true**;

// ToStringBuilder.setDefaultStyle not thread safe - call only once

**if** (!*isSbDefaultSet*()){

**synchronized** (ToStringBuilder.**class**) {

*setSbDefaultSet*(**true**);

ToStringBuilder.*setDefaultStyle*( **new** SaiToStringStyle() );

}

}

rbpsService.executeAll(currentProcess, totalProcessCount, repo);

retVal = repo.getJournalStr();

inDelete = deleteCurrentProcessFile( currentProcess);

**return** retVal;

}

**catch** (Exception ex) {

**if** (! inDelete ) {

deleteCurrentProcessFile( currentProcess);

}

retVal = repo.getJournalStr();

RbpsFaultBean faultInfo = **new** RbpsFaultBean();

faultInfo.setMessage(retVal);

**throw** **new** RbpsWsException( "RbpsWS runtime exception:", faultInfo, ex );

}

**finally** {

displayElapsed( className, methodName, **null**);

*markSafeToStartProcessing*( currentlyProcessing);

}

}

**private** String writeCurrentProcessFile( **final** String currentProcess, **final** String totalProcessCount) **throws** RbpsWsException{

String fileWriteStatus = "";

String fileName = currentProcessFilePath + currentProcess;

CommonUtils.*log*( *logger*, "Current Process File name " + fileName );

**try** {

File file = **new** File(fileName);

**if** ( file.exists() ) {

fileWriteStatus = String.*format*( "Error: Current Process File name %s exists. Exiting from current process %s", fileName, currentProcess );

CommonUtils.*log*( *logger*, fileWriteStatus );

**return** fileWriteStatus;

}

CommonUtils.*log*( *logger*, "About to write Current Process File" );

FileWriter writer = **new** FileWriter( fileName, **true** );

String datText = String.*format*("Process %s of %s", currentProcess, totalProcessCount );

writer.write(datText);

writer.close();

CommonUtils.*log*( *logger*, "Created Current Process File" );

fileWriteStatus = "Created Current Process File";

} **catch** (Exception ex) {

RbpsFaultBean faultInfo = **new** RbpsFaultBean();

faultInfo.setMessage("");

fileWriteStatus = String.*format*( "Error writing to current Process file %s. Exiting from current process %s", fileName, currentProcess );

//throw new RbpsWsException( String.format("Error writing to current Process file %s: ", fileName ), faultInfo, ex );

CommonUtils.*log*( *logger*, fileWriteStatus );

**return** fileWriteStatus;

}

**return** fileWriteStatus;

}

**private** **boolean** deleteCurrentProcessFile( **final** String currentProcess) **throws** RbpsWsException{

**boolean** inDelete = **true**;

String fileName = currentProcessFilePath + currentProcess;

**try**{

File file = **new** File(fileName);

**if**(file.delete()) {

CommonUtils.*log*( *logger*, fileName + " is deleted." );

}

**else**{

CommonUtils.*log*( *logger*, fileName + "delete operation failed.");

**return** inDelete;

}

} **catch**(Exception ex ){

RbpsFaultBean faultInfo = **new** RbpsFaultBean();

faultInfo.setMessage("");

CommonUtils.*log*( *logger*, String.*format*("Error deleting current Process file %s: ", fileName ) );

**return** inDelete;

}

**return** inDelete;

}

* **RbpsServiceFacade.java**

@Override

**public** **void** executeAll(**final** String currentProcess,

**final** String totalProcesses, RbpsRepository repo)

**throws** RbpsServiceException {

/\*\*

\* We will be getting data from the DB via the UserInformation web

\* service. We will be getting only one Claim object (a set of data

\* related to one Claim) at a time. This Claim should be entirely

\* processed and then RBPS updates on the DB this Claim's status and

\* label to remove it from the FIFO Queue.

\*

\* We will be looping this way until there is no more Claim on the Queue

\* or numberOfClaimsPerBatch has been processed. Exit condition will be

\* UserInformation web service returning no Claim or claimCount exceeds

\* numberOfClaimsPerBatch.

\*/

**int** claimCount = 0;

startUp(repo);

CommonUtils.*log*(*logger*, "In executeAll currentProcess: " + currentProcess);

**while** ( claimCount >= 0 ) {

claimCount = getAndProcessClaim(currentProcess, totalProcesses, repo, claimCount );

}

}

**private** **int** getAndProcessClaim(**final** String currentProcess,

**final** String totalProcesses, RbpsRepository repo, **int** claimCount) {

startTime = System.*currentTimeMillis*();

String exceptionInfo = "";

claimCount++;

**if** (claimCount > numberOfClaimsPerBatch) {

repo.setVeteran(**null**);

// decrement claimcount hidden in next line

CommonUtils.*log*(*logger*, String.*format*("Processed %d claims for process %s.. exiting RBPS", --claimCount, currentProcess));

**return** *MINUS\_ONE*;

}

CommonUtils.*log*(*logger*, String.*format*("Claim Count %d for process %s", claimCount, currentProcess));

**try** {

String threadName = Thread.*currentThread*().getName();

**if** ( threadName.contains("Claim") ) {

**int** index = threadName.indexOf( "Claim" );

threadName = threadName.substring(0, index).trim();

}

Thread.*currentThread*().setName(threadName + " Claim Count: " + claimCount );

// Get an eClaim Object from the VONAPP eClaims Queue

FindByDataSuppliedResponse newVonnappClaim = getNewClaim(**null**, currentProcess, totalProcesses, repo);

// Do we have an eClaim object?

**if** (*isUserInformationNull*(newVonnappClaim)) {

repo.setVeteran(**null**);

CommonUtils.*log*(*logger*,"Reached end of Queue, no more eClaim objects");

**return** *MINUS\_ONE*;

}

processClaim(newVonnappClaim, repo);

} **catch** (SoapFaultClientException ex) {

exceptionHandler.handleSoapFaultException(ex, repo);

} **catch** (Throwable ex) {

exceptionInfo = exceptionHandler.handleException(ex, repo);

} **finally** {

addJournalEntry(exceptionInfo, currentProcess, repo);

*cleanUp*(repo);

}

**return** claimCount;

}

**public** FindByDataSuppliedResponse getNewClaim(**final** String procId,

**final** String currentProcess, **final** String totalProcesses, RbpsRepository repo) {

**try** {

//userInformationWSHandler.setCommonUtils(utils);

**return** userInformationWSHandler.getFindByDataSuppliedResponse(

repo, procId, currentProcess,

totalProcesses);

} **catch** (Throwable ex) {

//

// Returning null so that it will exit the infinite loop we'd get

// by calling user info for the same claim over and over. Null will

// trigger the "isUserInformationNull" to return true, which will

// cause

// getAndProcess to return 0, which will cause the loop to exit.

//

CommonUtils.*log*(*logger*,

"Unable to get user information from web service", ex, **false**);

**return** **null**;

}

}

**private** **int** getAndProcessSingleClaim(**final** String procId, RbpsRepository repo) {

String exceptionInfo = "";

**try** {

// Get an eClaim Object from the VONAPP eClaims Queue

FindByDataSuppliedResponse newVonnappClaim = getNewClaim(procId,

**null**, **null**, repo);

// Do we have an eClaim object?

**if** (*isUserInformationNull*(newVonnappClaim)) {

repo.setVeteran(**null**);

CommonUtils.*log*(*logger*, "Reached end of Queue, no more eClaim objects");

**return** 0;

}

processClaim(newVonnappClaim, repo);

} **catch** (SoapFaultClientException ex) {

exceptionHandler.handleSoapFaultException(ex, repo);

} **catch** (Throwable ex) {

exceptionInfo = exceptionHandler.handleException(ex, repo);

} **finally** {

addJournalEntry(exceptionInfo, "", repo);

*cleanUp*(repo);

}

**return** 1;

}

**private** **void** processClaim(**final** FindByDataSuppliedResponse newVonnappClaim, RbpsRepository repo) **throws** RbpsClaimDataException {

rbpsRepositoryPopulator.populateFromVonapp(newVonnappClaim.getReturn().getUserInformation().get(0), repo);

String threadName = Thread.*currentThread*().getName();

**if** ( threadName.contains("Proc") ) {

**int** index = threadName.indexOf( "Proc" );

threadName = threadName.substring(0, index).trim();

}

Thread.*currentThread*().setName(threadName + " Proc Id: " + repo.getVnpProcId());

grabCorporateClaimId(repo);

// if ( filterVdc() ) {

//

// repository.setClaimProcessingState( "Rejecting VDC claim" );

// repository.addValidationMessage(

// "Auto Dependency Processing Reject Reason: This is a VDC/Ebenefits claim. RBPS cannot process this claim. Please Review."

// );

// claimProcessorHelper.sendClaimForManualProcessing();

// return;

// }

/\*\*

\* This is an initial data validation: claim data only. The

\* <code>genericUserInformationValidator.validate</code> method will run

\* generic common data validation and decide whether Claim process will

\* go through or log exception

\*

\* Validation will be an on-going task throughout the a Claim process

\* life cycle until RuleApp is called Additional validation will be done

\* from Claim Processor

\*/

genericUserInformationValidator.validate(repo);

claimProcessorHelper.updateClaimStatus(*CLAIM\_STATUS\_PROCESS\_STARTED*,

repo);

// Delegate the VONAPP eClaim processing to a Claim Processor

claimProcessorFactory.getClaimProcessor(

repo.getVeteran().getClaim().getEndProductCode()).processClaim(repo);

CommonUtils.*log*(*logger*, "\*\*\* RBPS sucessfully processed ClaimID ["

+ repo.getVeteran().getClaim().getClaimId()

+ "] with ProcID [" + repo.getVnpProcId()

+ "]");

}

* **UserInformationWSHandler**

**public** FindByDataSuppliedResponse getFindByDataSuppliedResponse( **final** RbpsRepository repo,

**final** String currentProcess,

**final** String totalProcesses ) {

**return** getFindByDataSuppliedResponse( repo, **null**, currentProcess, totalProcesses );

}

**public** FindByDataSuppliedResponse getFindByDataSuppliedResponse( **final** RbpsRepository repo,

**final** String procId,

**final** String currentProcess,

**final** String totalProcesses ) {

FindByDataSupplied request = buildRequest(repo, procId, currentProcess, totalProcesses );

**return** call( repo, request );

}

**private** FindByDataSupplied buildRequest( **final** RbpsRepository repository,

**final** String procId,

**final** String currentProcess,

**final** String totalProcesses ) {

FindByDataSupplied findByDataSupplied = **new** FindByDataSupplied();

UserInformationInput userInformationInput = **new** UserInformationInput();

**if** ( StringUtils.*isBlank*( procId ) ) {

userInformationInput.setVnpProcStateTypeCd(*CLAIM\_STATUS\_READY\_FOR\_RBPS*);

userInformationInput.setUserIdentityType(*USER\_IDENTITY\_TYPE*);

userInformationInput.setUserIdentityValue(*USER\_IDENTITY\_VALUE*);

**try** {

userInformationInput.setBatchProcessorNbr( Long.*valueOf*(currentProcess) );

userInformationInput.setBatchProcessorTotal(Long.*valueOf*(totalProcesses) );

} **catch**( Exception ex ) {

String detailMessage = "Batch process parameters passed are not numeric";

repository.addValidationMessage( detailMessage );

*logger*.error(" \*\*\*RBPS: [" + detailMessage + "]");

**throw** **new** RbpsWebServiceClientException(detailMessage, ex);

}

}

**else** {

userInformationInput.setVnpProcId(procId);

}

findByDataSupplied.setArg0(userInformationInput);

**return** findByDataSupplied;

}

* **CommonUtils**

All methods were made public static final

* **LogUtils**

All methods were made public static final

* **SimpleDateUtils**

All methods were made public static final

* **rbps-core-spring-config.xml**

sample bean definition:

<bean id=*"vdcClaimFilter"*

class=*"gov.va.vba.rbps.services.impl.VdcClaimFilter"*

autowire=*"byName"*

**scope=*"prototype"*>**

<property name=*"shouldPerformFiltering"* value=*"${rbps.ws.shouldFilterVdcClaims}"* />

</bean>

* **rbps-ws-client-spring-config.xml**

sample bean definition:

<bean id=*"awardsFindMilitaryPayWSHandler"*

class=*"gov.va.vba.rbps.services.ws.client.handler.awards.FindMilitaryPayWSHandler"*

autowire=*"byName"*

**scope=*"prototype"* >**

<!-- <property name="repository" ref="repository" /> -->

<property name=*"webServiceTemplate"* ref=*"findMilitaryPayWebServiceTemplate"* />

<property name=*"findMilitaryPayUri"* value=*"${rbps.ws.awards.uri.findMilitaryPay}"* />

</bean>

* **FCDRPopulator:**

**if** ( promulgationDate.after( repository.getVeteran().getClaim().getReceivedDate() ) ) {

**throw** **new** RbpsRuntimeException( String.*format*( "Rating for Veteran %s %s promulgated after date of claim, please review",

repository.getVeteran().getFirstName(), repository.getVeteran().getLastName() ) );

}

* **GeneratePdf:**

**public** List<AwardSummary> responseToSummary( **final** RbpsRepository repository,

**final** ProcessAwardDependentResponse awardResponse) {

**if** ( awardResponse == **null** ) {

**return** **null**;

}

List<AwardSummary> awardSummaryList = **new** ArrayList<AwardSummary>();

MultiMap dependentMap = createDependentMap( awardResponse, repository );

List<AwardLineSummaryVO> lineSummaries = getLineSummaries( awardResponse );

Date firstChangeDate = *getFirstChangeDate*( awardResponse );

**try** {

*addDenialSummary*( repository, lineSummaries );

**for** ( AwardLineSummaryVO summary : lineSummaries ) {

**if** ( isPriorDecision( firstChangeDate, summary ) ) {

logUtils.log( "Skipping summary: " + summary.getAwardLineReasons().get(0).getReasonSequenceNumber(), repository );

**continue**;

}

List<AwardSummary> summaries = *createAwardSummary*( dependentMap, summary );

awardSummaryList.addAll( summaries );

}

*addToSummaryListForNoMatchingAwardLineReason*(dependentMap, lineSummaries, awardSummaryList);

}

**finally** {

dependentMap = **null**;

lineSummaries = **null**;

}

**return** awardSummaryList;

}

**private** **static** **final** **void** addToSummaryListForNoMatchingAwardLineReason( **final** MultiMap dependentMap,

**final** List<AwardLineSummaryVO> lineSummaries,

List<AwardSummary> awardSummaryList ) {

**for** ( Object dependentObject : dependentMap.values() ) {

DependencyDecisionResultVO dependencyDecisionResultVO = (DependencyDecisionResultVO) dependentObject;

**if** ( dependencyDecisionResultVO.getGrantDenial().toUpperCase().contains( "No matching Award Line/Reason".toUpperCase() ) ) {

*createAwardDependentSummary*( dependencyDecisionResultVO, lineSummaries, awardSummaryList );

**int** index = dependencyDecisionResultVO.getGrantDenial().indexOf("-");

dependencyDecisionResultVO.setGrantDenial( dependencyDecisionResultVO.getGrantDenial().substring(0, index).trim() );

}

}

}

**private** **static** **final** List<DependencyDecisionResultVO> getDependentFromMap( **final** Map<Integer, DependencyDecisionResultVO> dependentMap,

**final** AwardLineSummaryVO summary ) {

**if** ( summary == **null** ) {

**return** **new** ArrayList<DependencyDecisionResultVO>();

}

List<DependencyDecisionResultVO> foundDependents = **new** ArrayList<DependencyDecisionResultVO>();

**for** ( AwardReasonSeqNbrVO reason : summary.getAwardLineReasons() ) {

List<DependencyDecisionResultVO> dependents = (List<DependencyDecisionResultVO>) dependentMap.get( reason.getReasonSequenceNumber() );

**if** ( CollectionUtils.*isEmpty*( dependents ) ) {

**continue**;

}

*addToFoundDependentsWithoutNoMatchingAwardLineReason*(foundDependents, dependents);

}

**return** foundDependents;

**private** **static** **void** addToFoundDependentsWithoutNoMatchingAwardLineReason( List<DependencyDecisionResultVO> foundDependents, List<DependencyDecisionResultVO> dependents) {

**for** ( DependencyDecisionResultVO dependencyDecisionResultVO : dependents ) {

**if** ( ! dependencyDecisionResultVO.getGrantDenial().toUpperCase().contains( "No matching Award Line/Reason".toUpperCase() ) ) {

foundDependents.add( dependencyDecisionResultVO );

}

}

}

* } **AwardSummaryBuilder**

**private** **void** assignDependentInfo( **final** AwardReasonSeqNbrVO reason,

**final** DependencyDecisionResultVO dependentInfo,

**final** AwardReason newReason ) {

**if** ( dependentInfo == **null** ) {

**return**;

}

**if** ( dependentInfo.getAlReasonSequenceNumber() != reason.getReasonSequenceNumber() ) {

**return**;

}

newReason.setSequenceNumber( dependentInfo.getAlReasonSequenceNumber() );

newReason.setDependentName( dependentInfo.getFirstName() );

newReason.setDependencyDecisionType( dependentInfo.getDependencyDecisionType() );

newReason.setDependencyDecisionTypeDescription( dependentInfo.getDependencyDecisionTypeDescription() );

newReason.setEffectiveDate( dependentInfo.getAwardEffectiveDate().toGregorianCalendar().getTime() );

newReason.setFirstName( dependentInfo.getFirstName() );

newReason.setFullName( dependentInfo.getFullName() );

newReason.setChild( isChild( dependentInfo ) );

String grantDenial = getGrantDenialValue( dependentInfo.getGrantDenial().toUpperCase() );

newReason.setApprovalType( ApprovalType.*valueOf*( grantDenial ) );

newReason.setClaimId( dependentInfo.getClaimID() );

newReason.setCorpParticipantId( dependentInfo.getPersonID() );

}

* **GenericLetterGeneration**

**private** PdfDocument generateRbpsLettter( **final** RbpsRepository repository,

**final** RbpsApplicationDetails rbpsApplicationDetails,

**final** ByteArrayOutputStream baoStream,

**final** String pdfFileName,

**final** **boolean** hasApprovals,

**final** **boolean** hasDenials,

**final** **boolean** hasMilitaryPay ) {

**if**( ! hasApprovals && ! hasDenials ) {

**throw** **new** RbpsRuntimeException( String.*format*( "Error determining Letter template type with data provided by Awards for Letter >%S<", pdfFileName ) );

}

GenerateLetterFromLiveCycle generator = **new** GenerateLetterFromLiveCycle();

**return** generator.generateLetter( repository, rbpsApplicationDetails, baoStream, pdfFileName, hasApprovals, hasDenials, hasMilitaryPay );

}