System Architecture Document
Grants.gov System

V 2.0

April 27, 2007

Table of Contents

List of Figures

1. Overall System Architecture

The extended capabilities Grants.gov system architecture is based on the previously existing system architecture. Our approach has been to preserve the healthy components and remove the components that negatively affect performance and scalability. Legacy opportunity data and application data are supported through integration at the Data Access layer. In the previously existing System, the Data Access design continues to allow connectivity to only a single database. The underlying implementation has been modified to include queries to the legacy Database and the extended capabilities Database. The Mechanisms are described in more detail in the Design Document.

The schematic below shows the revised Grants.gov system architecture with extended capabilities. It includes, new, as well as reengineered components. The legend in the schematic indicates the color-coding that distinguishes the new, reused, and reengineered components. The changes on the client side include the Adobe Acrobat Reader and the ability to search attachments. On the Server side, the changes include the introduction of the Adobe LiveCycle forms server, Google Search Appliance, and the reengineered Apply process.

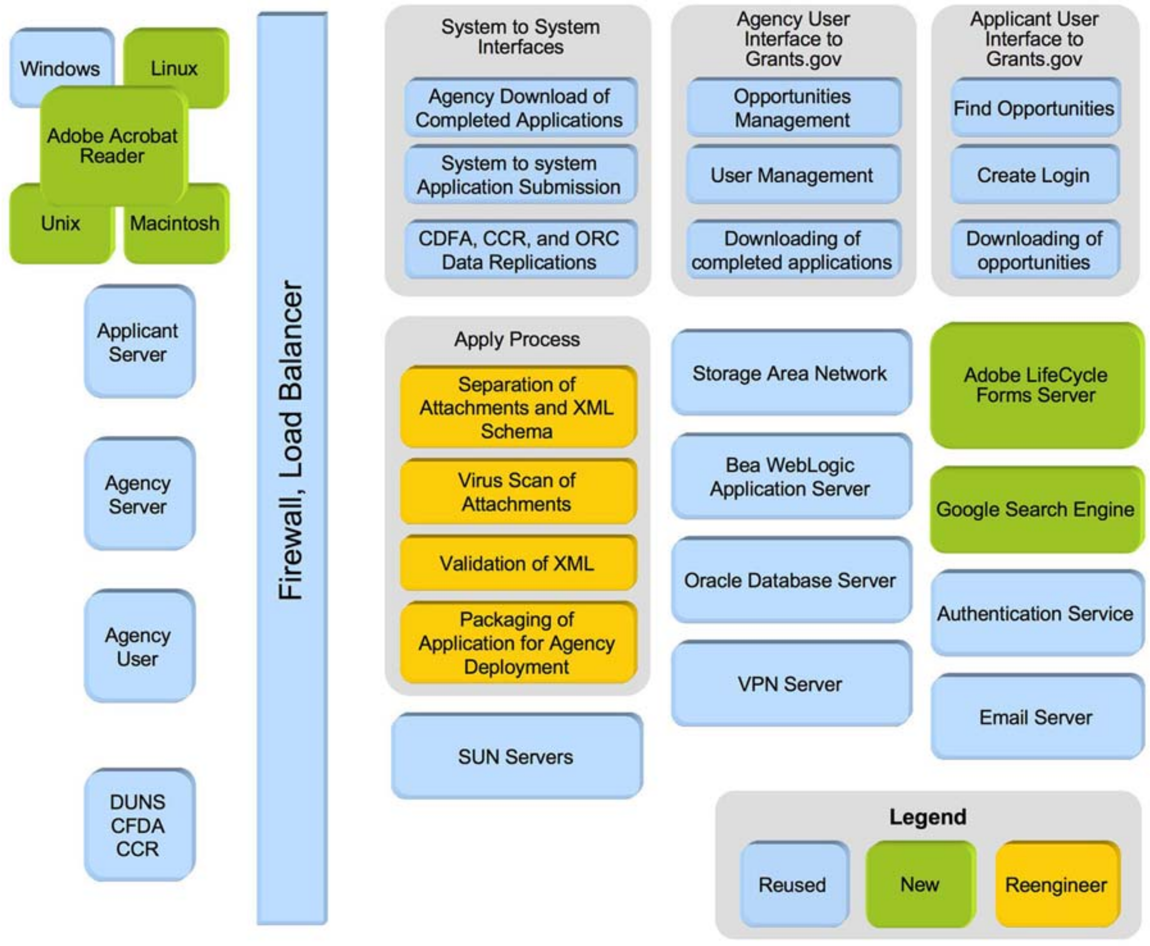

Windows
Linux
Adobe Acrobat Reader
Unix
Macintosh
Applicant Server
Agency Server
Agency User
DUNS CFDA CCR
Firewall, Load Balancer
System to System Interfaces
Agency Download of Completed Applications
System to system Application Submission
CDFA, CCR, and ORC Data Replications
Agency User Interface to Grants.gov
Opportunities Management
User Management
Downloading of completed applications
Applicant User Interface to Grants.gov
Find Opportunities
Create Login
Downloading of opportunities
Apply Process
Separation of Attachments and XML Schema
Virus Scan of Attachments
Validation of XML
Packaging of Application for Agency Deployment
SUN Servers
Storage Area Network
Bea WebLogic Application Server
Oracle Database Server
VPN Server
Adobe LifeCycle Forms Server
Google Search Engine
Authentication Service
Email Server
Legend
Reused
New
Reengineer

Figure 1: 2007 Grants.gov System Architecture

2. Architecture Summary

The extended capabilities Grants.gov system architecture is the result of some **strategic** and **tactical** architectural decisions. The **strategic** architectural decisions include the following:

- Replacing PureEdge based forms with Adobe based forms.
- Introducing the Google Search Appliance.
- Replacing iPlanet based Queues; Queue Listeners with Plain Old Java Objects (POJO) based Services.
- Replacing InFlowSuite rule sets and rules with POJO based rules.
- Retaining the current database schema.
- More conscious use of Spring and Hibernate frameworks.
- Streamlining error processing.
- Using the capabilities of the application servers.

The **tactical** architectural decisions include the following:

- Retaining the current set of Java Service Pages (JSP) and servlets.
- Retaining the current Data Access Objects (DAO) and Hibernate for database interactions.
- Removing embedded Standard Query Language (SQL) and converting to Hibernate objects.
- Removing InFlowSuite dependency.

3. Technology Stack

The revised extended capabilities system is being deployed under Weblogic 8.1.5 and JDK 1.4.1. Adobe 7.0.8 is being used. Open source frameworks such as *Spring* and *Hibernate* form the foundation on which new components and services are being developed.

4. Software Architecture Layers

The existing system software architecture had no clear separation of layers. Even though several popular frameworks like *struts* and *Hibernate* had been used, there was no clear delineation of responsibilities between presentation, business/services and data layers. This made the task of refactoring and reengineering very difficult. The existing code base was quite fragile so that making large-scale changes jeopardized the delivery of a stable, industrial strength solution in April 2007.

The extended capabilities Grants.gov system architecture includes defined presentation, business, and data access layers. The schematic shown in Figure 2 provides a simplified view of the new software architecture layers and the components and services that are part of each layer. Following the Model View Controller (MVC) paradigm, the reengineered architecture consists of presentation, business, and data access layers.

The presentation layer will include *Spring*, *MVC*, and struts. The search application initially uses the struts framework at a very rudimentary level. Future releases will take advantage of the frameworks more comprehensively.

The business layer includes POJO Business Services and infrastructure services. The infrastructure services include Adobe product API's, existing InFlowSuite services that are reused and re-factored, and support for Google search.

The data access layer will include custom data access components built using the InFlowSuite utilities including Hibernate's base data access components. Our goal is to eventually convert all custom data access components to Hibernate-based data access components.

The extended capabilities system of April 2007 is configured to process only the opportunities created in the April 2007 extended capabilities system. Figure 2 addresses only the extended capabilities Architecture. The Architecture for processing IBM Workplace Forms remains intact. The previously existing Architecture will be used for processing legacy opportunities. Modifications have been made to the new Search module for users to download the packages created for old opportunities.

Modifications have also been made to the Grantor User Interface to retrieve old, as well as new, opportunities. These changes are also discussed in the Design Document.

There is no direct relationship between Spring and Inflowsuite frameworks. Figure 2 shows that Spring and Hibernate are Infrastructure Frameworks. Currently, the Grantor and Grantee User Interface components use the Inflowsuite services for implementation. Once the User Interface is redesigned, this dependency will be removed.

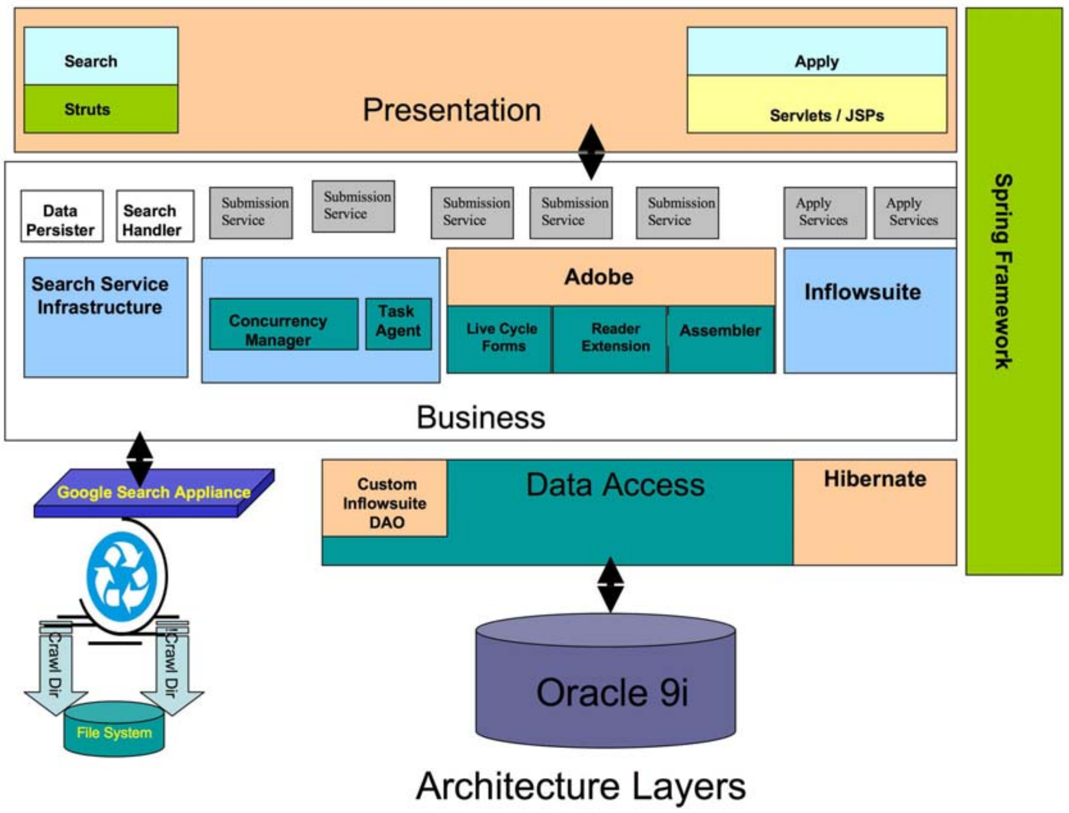


Figure 2: Simplified View of Software Architecture Layers

4.1. Security Architecture

The Security Architecture of the extended capabilities system has not changed from the earlier system implementation. Changing the Security Architecture was not part of the scope for this release. When the User Interface is redesigned, the authorization component will be refactored and redesigned. Standard Java Authentication and Authorization Service (JAAS) API will be reviewed for implementation. The implementation of the left navigation bar uses a few Inflowsuite core classes as well as custom classes. Changes have to be done to remove the core classes.

5. Primary Features of Extended Capabilities Architecture

One of the key features of the extended capabilities Grants.gov system architecture is the utilization of the Google Search Appliance to replace the existing "find" functionality. The current *find* functionality is limited to searching the metadata associated with *Synopsis* and *Synopsis title*. There was no support in the earlier system for searching the attachments or other website content. Google Search Appliance offers a more comprehensive solution for implementing search functionality. The Google Search Appliance is an integrated hardware and software solution that provides a simple HTTP based GET or POST interface. The Google Search Appliance provides the ability to search 300,000 documents.

The Google Search Appliance-based search has been integrated with the already existing Grants.gov application using the universally accepted Google User Interface.

5.1. Google-Based Search

There are two parts to the integration of Google-based search into Grants.gov. The first part involved incorporating the Google-based search solution involved persisting synopsis and attachments data in a File System (Google Search Appliance is File System based; Google Search Appliance can search in databases.). This has enabled the Google search engine to crawl and index the contents. The legacy system was modified and a new component was added to create a new HTML file for *synopsis* and save it to the File System. Figure 3 shows the new component as the *File System Persistence Engine*. The Google Search Appliance has been configured to "crawl" the file system and index the contents. The contents in the File System will be arranged in a directory.

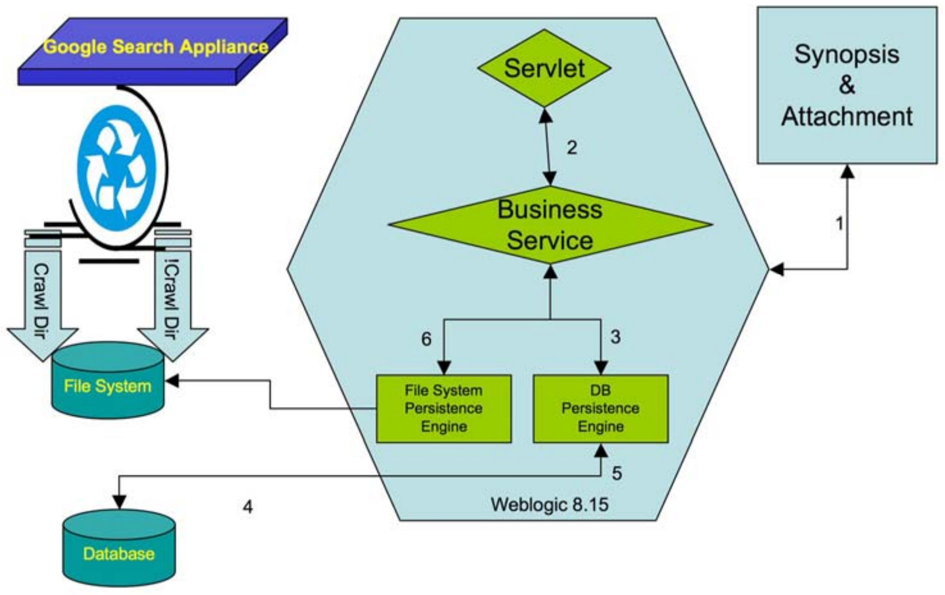
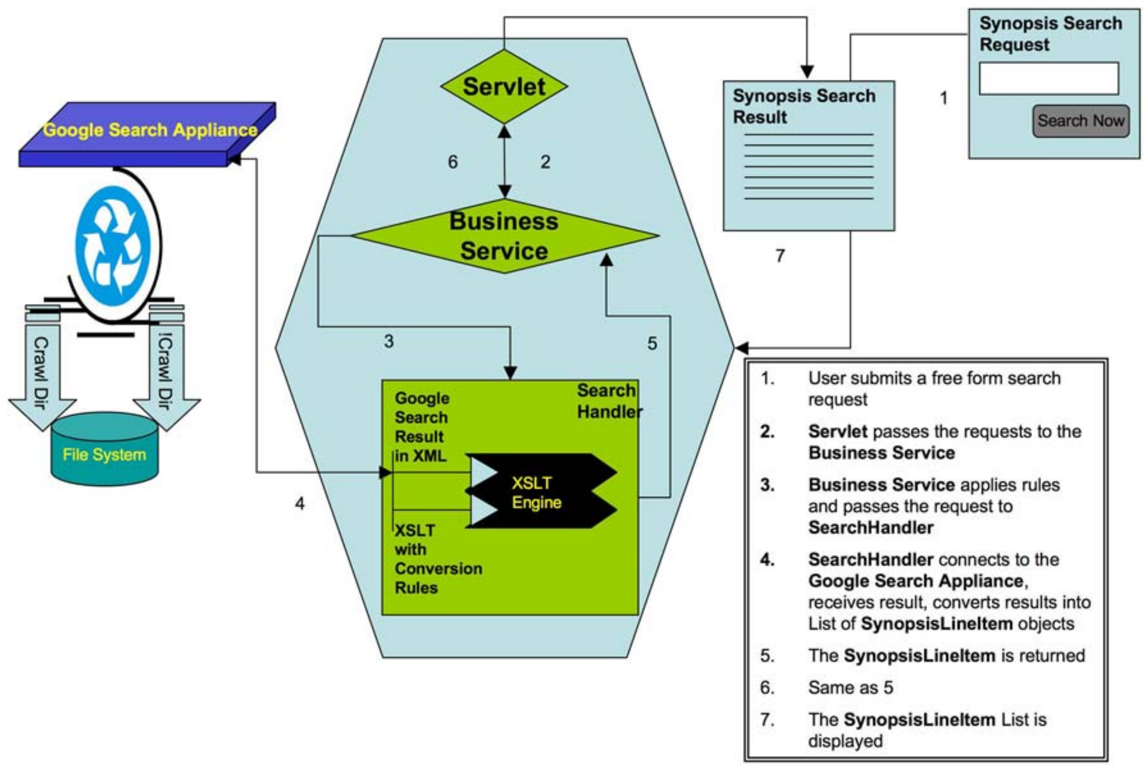


Figure 3: Conceptual Architecture

The second part of integrating Google-based search into Grants.gov involves ***rendering the search results***. Figure 4 shows the conceptual flow for ***rendering the synopsis search results***.



1. User submits a free form search request
2. **Servlet** passes the requests to the **Business Service**
3. **Business Service** applies rules and passes the request to **SearchHandler**
4. **SearchHandler** connects to the **Google Search Appliance**, receives result, converts results into List of **SynopsisLineItem** objects
5. The **SynopsisLineItem** is returned
6. Same as 5
7. The **SynopsisLineItem** List is displayed

Figure 4: Rendering Synopsis

The legacy Search module code has been modified and a new Search Handler has been added to pass search requests to the Google Search Appliance. The same information that is currently used in the existing system for display is extracted from Google Search Appliance using GSA-JAPI: Java API for interacting with the Google Search Appliance™. The same Search Result Collection that the previous user Interface code used to display the result of a search is used to render the search data returned by Google Search Appliance. The extended capabilities Architecture also provides enhancements in the form of searching through synopsis attachments.

The class diagram shows the details of the *File System Persistence Engine* and the **Search Handler.** The *File System Persistence Engine* is implemented as *HTMLGoogleableDataPersister* and the **Search Handler** is implemented as *GoogleSearchHandler*. The class diagram (Figure 5) also shows key classes such as *OppSynopsisService* and *SearchAction* that have been modified.

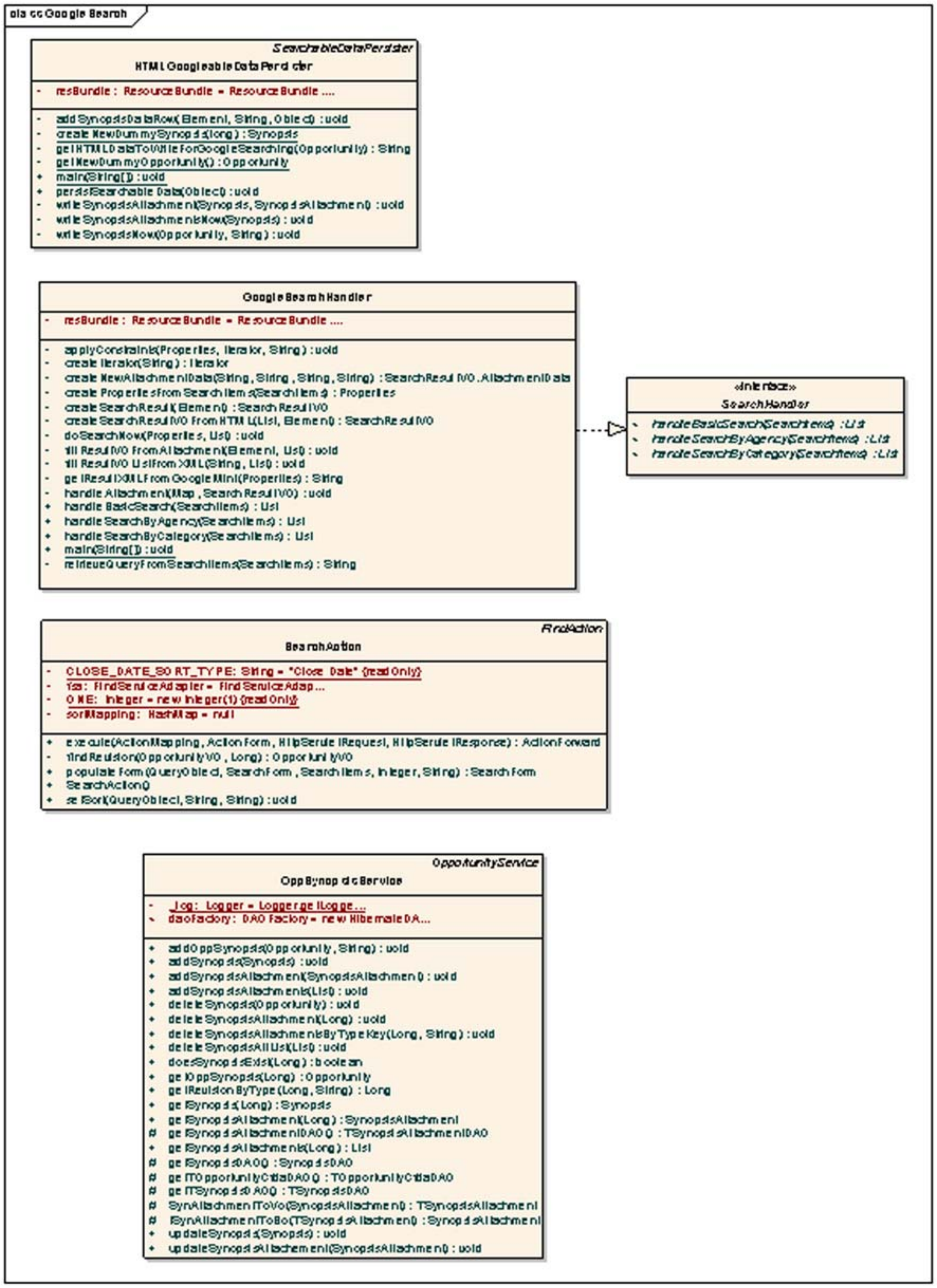

class Google Search
SearchableDataPersister
HTMLGoogleableDataPersister
- resBundle : ResourceBundle = ResourceBundle ....
- addSynopsisDataRow(Element, String, Object) : void
- createNewDummySynopsis(long) : Synopsis
- getHTMLDataToWriteForGoogleSearching(Opportunity) : String
- getNewDummyOpportunity() : Opportunity
+ main(String[]) : void
+ persistSearchableData(Object) : void
- writeSynopsisAttachment(Synopsis, SynopsisAttachment) : void
- writeSynopsisAttachmentsNow(Synopsis) : void
- writeSynopsisNow(Opportunity, String) : void
GoogleSearchHandler
- resBundle : ResourceBundle = ResourceBundle ....
- applyConstraints(Properties, Iterator, String) : void
- createIterator(String) : Iterator
- createNewAttachmentData(String, String, String, String) : SearchResultVO.AttachmentData
- createPropertiesFromSearchItems(SearchItems) : Properties
- createSearchResult(Element) : SearchResultVO
- createSearchResultVOFromHTML(List, Element) : SearchResultVO
- doSearchNow(Properties, List) : void
- fillResultVOFromAttachment(Element, List) : void
- fillResultVOListFromXML(String, List) : void
- getResultXMLFromGoogleMini(Properties) : String
- handleAttachment(Map, SearchResultVO) : void
+ handleBasicSearch(SearchItems) : List
+ handleSearchByAgency(SearchItems) : List
+ handleSearchByCategory(SearchItems) : List
+ main(String[]) : void
- retrieveQueryFromSearchItems(SearchItems) : String
«interface»
SearchHandler
+ handleBasicSearch(SearchItems) : List
+ handleSearchByAgency(SearchItems) : List
+ handleSearchByCategory(SearchItems) : List
FindAction
SearchAction
- CLOSE_DATE_SORT_TYPE: String = "Close Date" {readOnly}
- fsa: FindServiceAdapter = FindServiceAdap...
- ONE: Integer = new Integer(1) {readOnly}
- sortMapping: HashMap = null
+ execute(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse) : ActionForward
- findRevision(OpportunityVO, Long) : OpportunityVO
+ populateForm(QueryObject, SearchForm, SearchItems, Integer, String) : SearchForm
+ SearchAction()
+ setSort(QueryObject, String, String) : void
OpportunityService
OppSynopsisService
- _log: Logger = Logger.getLogge...
- daoFactory: DAOFactory = new HibernateDA...
+ addOppSynopsis(Opportunity, String) : void
+ addSynopsis(Synopsis) : void
+ addSynopsisAttachment(SynopsisAttachment) : void
+ addSynopsisAttachments(List) : void
+ deleteSynopsis(Opportunity) : void
+ deleteSynopsisAttachment(Long) : void
+ deleteSynopsisAttachmentsByTypeKey(Long, String) : void
+ deleteSynopsisAllList(List) : void
+ doesSynopsisExist(Long) : boolean
+ getOppSynopsis(Long) : Opportunity
+ getRevisionByType(Long, String) : Long
+ getSynopsis(Long) : Synopsis
+ getSynopsisAttachment(Long) : SynopsisAttachment
# getSynopsisAttachmentDAO() : TSynopsisAttachmentDAO
+ getSynopsisAttachments(Long) : List
# getSynopsisDAO() : SynopsisDAO
# getTOpportunityCfdaDAO() : TOpportunityCfdaDAO
# getTSynopsisDAO() : TSynopsisDAO
# SynAttachmentToVo(SynopsisAttachment) : TSynopsisAttachment
# tSynAttachmentToBo(TSynopsisAttachment) : SynopsisAttachment
+ updateSynopsis(Synopsis) : void
+ updateSynopsisAttachment(SynopsisAttachment) : void

Figure 5: Class Google Search

5.2. Adobe Based Forms Processing

In the extended capabilities Grants.gov system, Adobe PDF forms technology replaces the PureEdge based forms. This necessitated changes throughout the previous architecture and code base. PDF Forms replace the PureEdge forms. The Schematic in Figure 6 shows an overview of the business process flow associated with **Forms Processing**.

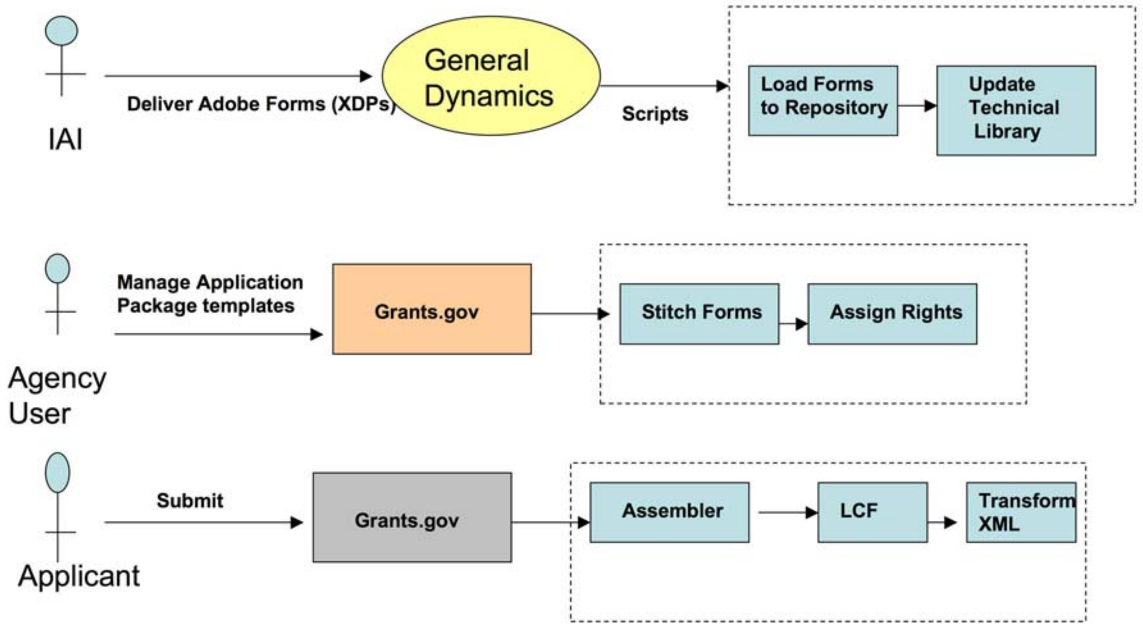


Figure 6: Forms Processing

There are four main aspects to **Forms Processing**:

1. **Stitching**
2. **Reader Extension (Assign Rights)**
3. **Separating Attachments from XML**
4. **Generating XML from PDF**

These are addressed through standard and custom components from Adobe.

Stitching is accomplished through custom Java components. Adobe forms are stored as XDP files; the ***Stitching*** component takes multiple XDP files and produces a single unified XDP file.

Reader Extension allows the *read only* PDF forms to be presented as PDF forms that can be completed (filled). This is accomplished programmatically through API's provided by Adobe.

LiveCycle Forms API is used to extract XML from PDF documents. The LiveCycle Forms Architecture is presented in Figure 7. The RMI based EJB API will be used to convert a given PDF document to XML.

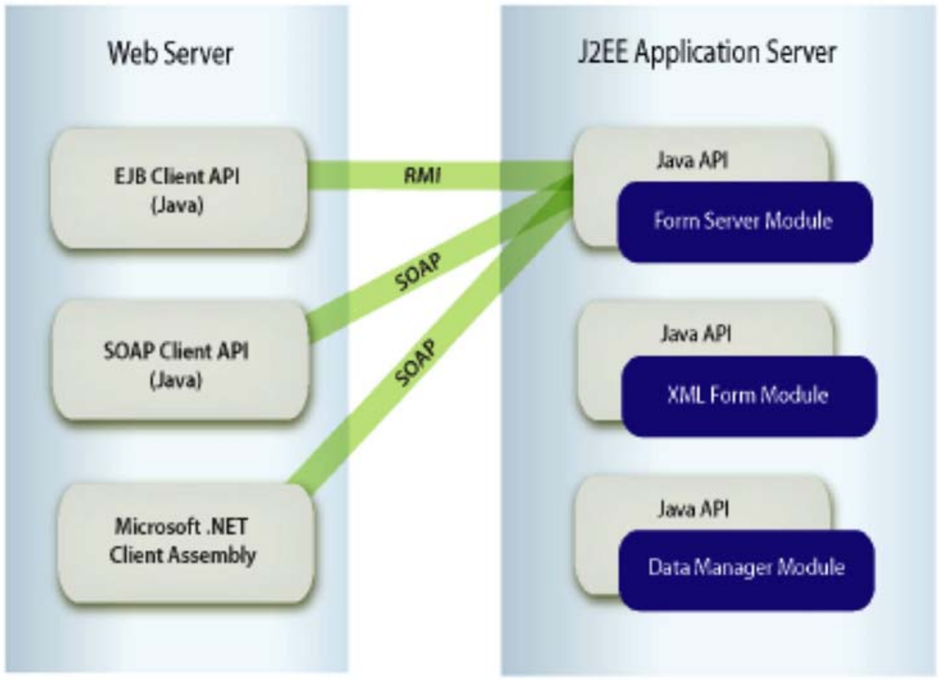


Figure 7: LiveCycle Forms Architecture

LiveCycle Assembler EJB API will be used to separate the attachments from the PDF. The architecture of the LiveCycle assembler is shown in Figure 8.

When an applicant submits a completed application, a PDF containing the forms and the attachments gets transmitted from the client side to the server side. On the Server side, the Assembler product is used to separate the attachments from the forms, and the LiveCycle forms API is used to extract the XML. The XML is validated against the Opportunity Schema.

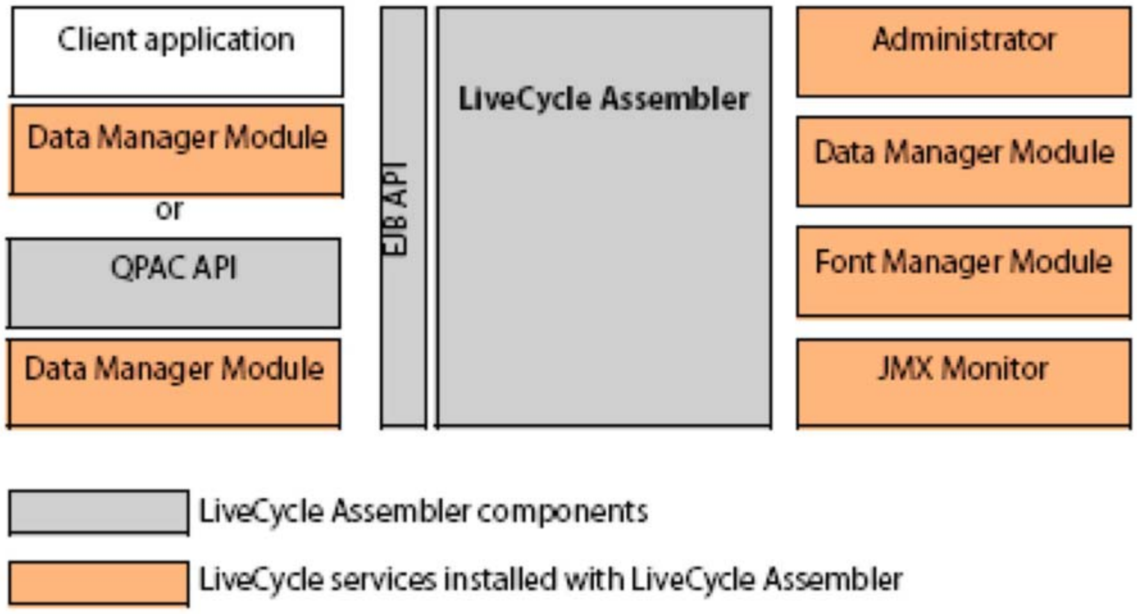


Figure 8: LiveCycle Assembler

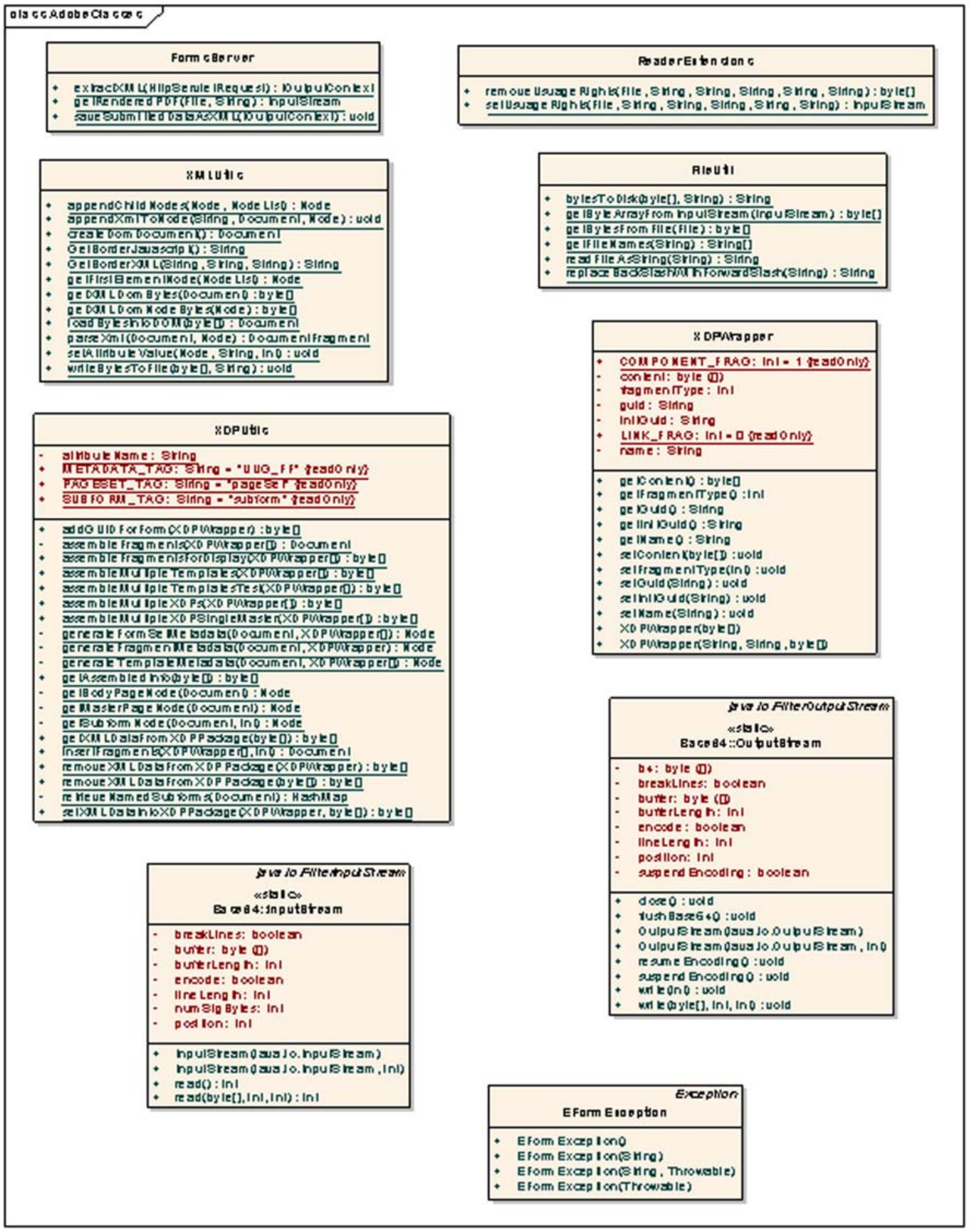

class AdobeClasses
FormsServer
+ extractXML(HttpServletRequest) : OutputContext
+ getRenderedPDF(File, String) : InputStream
+ saveSubmittedDataAsXML(OutputContext) : void
ReaderExtensions
+ removeUsageRights(File, String, String, String, String, String) : byte[]
+ setUsageRights(File, String, String, String, String, String) : InputStream
XMLUtils
+ appendChildNodes(Node, NodeList) : Node
+ appendXmlToNode(String, Document, Node) : void
+ createDomDocument() : Document
+ GetBorderJavascript() : String
+ GetBorderXML(String, String, String) : String
+ getFirstElementNode(NodeList) : Node
+ getXMLDomBytes(Document) : byte[]
+ getXMLDomNodeBytes(Node) : byte[]
+ loadBytesIntoDOM(byte[]) : Document
+ parseXml(Document, Node) : DocumentFragment
+ setAttributeValue(Node, String, int) : void
+ writeBytesToFile(byte[], String) : void
FileUtil
+ bytesToDisk(byte[], String) : String
+ getByteArrayFromInputStream(InputStream) : byte[]
+ getBytesFromFile(File) : byte[]
+ getFileNames(String) : String[]
+ readFileAsString(String) : String
+ replaceBackSlashWithForwardSlash(String) : String
XDPWrapper
+ COMPONENT_FRAG: int = 1 {readOnly}
- content: byte ([])
- fragmentType: int
- guid: String
- initGuid: String
+ LINK_FRAG: int = 0 {readOnly}
- name: String
+ getContent() : byte[]
+ getFragmentType() : int
+ getGuid() : String
+ getInitGuid() : String
+ getName() : String
+ setContent(byte[]) : void
+ setFragmentType(int) : void
+ setGuid(String) : void
+ setInitGuid(String) : void
+ setName(String) : void
+ XDPWrapper(byte[])
+ XDPWrapper(String, String, byte[])
XDPUtils
- attributeName: String
+ METADATA_TAG: String = "UUG_FF" {readOnly}
+ PAGESET_TAG: String = "pageSet" {readOnly}
+ SUBFORM_TAG: String = "subform" {readOnly}
+ addGUIDForForm(XDPWrapper) : byte[]
- assembleFragments(XDPWrapper[]) : Document
+ assembleFragmentsForDisplay(XDPWrapper[]) : byte[]
+ assembleMultipleTemplates(XDPWrapper[]) : byte[]
+ assembleMultipleTemplatesTest(XDPWrapper[]) : byte[]
+ assembleMultipleXDPs(XDPWrapper[]) : byte[]
+ assembleMultipleXDPSingleMaster(XDPWrapper[]) : byte[]
- generateFormSetMetadata(Document, XDPWrapper[]) : Node
- generateFragmentMetadata(Document, XDPWrapper) : Node
- generateTemplateMetadata(Document, XDPWrapper[]) : Node
+ getAssembledInfo(byte[]) : byte[]
- getBodyPageNode(Document) : Node
- getMasterPageNode(Document) : Node
- getSubformNode(Document, int) : Node
+ getXMLDataFromXDPPackage(byte[]) : byte[]
+ insertFragments(XDPWrapper[], int) : Document
+ removeXMLDataFromXDPPackage(XDPWrapper) : byte[]
+ removeXMLDataFromXDPPackage(byte[]) : byte[]
- retrieveNamedSubforms(Document) : HashMap
+ setXMLDataIntoXDPPackage(XDPWrapper, byte[]) : byte[]
java.io.FilterOutputStream
«static»
Base64::OutputStream
- b4: byte ([])
- breakLines: boolean
- buffer: byte ([])
- bufferLength: int
- encode: boolean
- lineLength: int
- position: int
- suspendEncoding: boolean
+ close() : void
+ flushBase64() : void
+ OutputStream(java.io.OutputStream)
+ OutputStream(java.io.OutputStream, int)
+ resumeEncoding() : void
+ suspendEncoding() : void
+ write(int) : void
+ write(byte[], int, int) : void
java.io.FilterInputStream
«static»
Base64::InputStream
- breakLines: boolean
- buffer: byte ([])
- bufferLength: int
- encode: boolean
- lineLength: int
- numSigBytes: int
- position: int
+ InputStream(java.io.InputStream)
+ InputStream(java.io.InputStream, int)
+ read() : int
+ read(byte[], int, int) : int
Exception
EFormException
+ EformException()
+ EformException(String)
+ EformException(String, Throwable)
+ EformException(Throwable)

Figure 9: Class Adobe Classes

5.3. Service Infrastructure

Using the earlier Grants.gov system, submitted applications are processed using a combination of Queues, Queue Listeners, and Queue Rules.

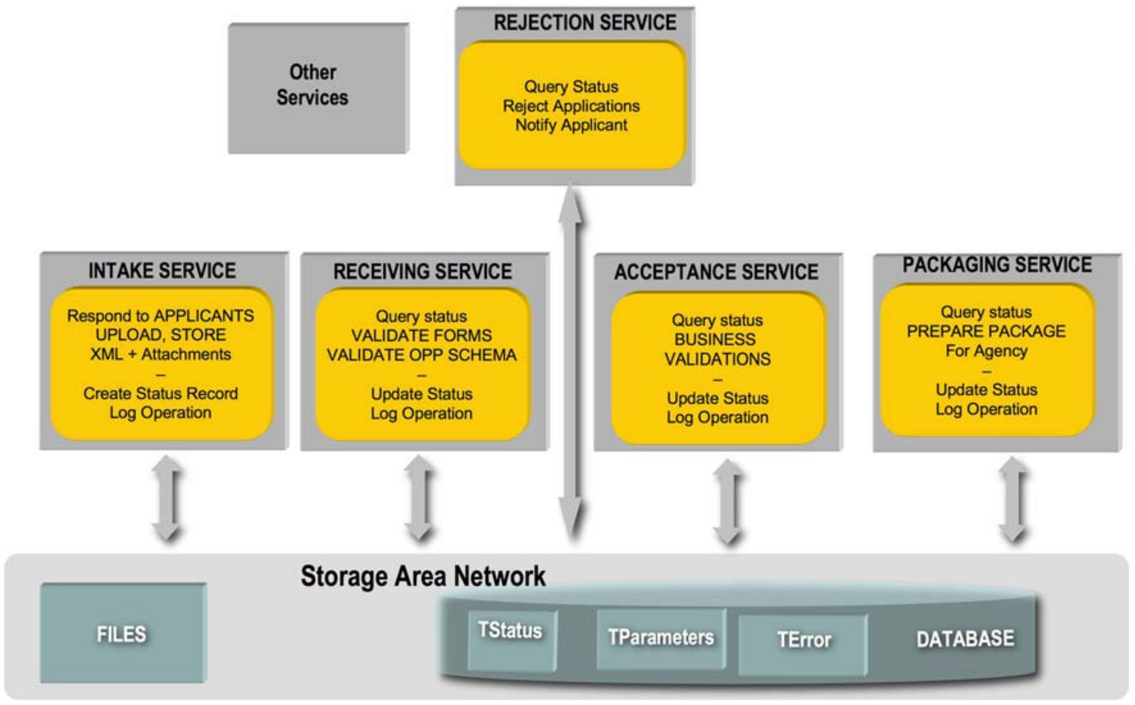


Figure 10: Service Infrastructure

The Service Infrastructure provides a substrate for building POJO-based services that replace the current InFlowSuite Queues and Queue listeners. Service Infrastructure allows the system, as a whole, to be rebalanced or reconfigured dynamically by changing the processing parameters of specific services. It offers the following capabilities:

- Ability to change application processing parameters without stopping or starting the container.
- Ability to change application processing without interrupting a transaction in progress.
- Location transparency - Ability to deploy services to any container.

To achieve these goals, processing parameters are stored in the database and checked periodically. Container-level processing parameters are stored in a system parameter table (TParameters), and checked at startup and when a transaction is completed. Application specific parameters are stored in the application control table (TStatus) and checked for the next transaction to be processed.

We have carefully analyzed the current workflow and mapped it to a set of autonomous services. The different Services and their responsibilities are shown in Figure 10. Each Service maps to a significant step in the Submission Processing workflow, such as Intake, Receiving, Rejection, Acceptance, and Packaging. The Service Architecture uses a shared File System to store incoming applications. Once the applications are processed, contents of the application are also stored in different database tables such as **TFile** and **TAttachments**.

The Architecture uses the concepts of Service Identities and Service Parameters to implement individual services.

5.3.1. Service identities

A Service running in application servers (containers) needs to acquire and know its own identity, then query the processing parameters assigned to that identity.

The service name is hard coded into an application. The code performs only one service. The service identity is acquired by reading the URL of the container in which it is running. Therefore, only one service of a specific type runs in a container. However, multiple containers may perform the same service. The ***ServiceConfigurator*** Component encapsulates the logic associated with starting different services on different containers.

5.3.2. Service parameters

The service reads the **TParameters** table to acquire the concurrency level of the service. When running, the service will launch as many concurrent threads as the parameter specifies. So, even if there is only one service in a container, the level of concurrent processing is fully adjustable.

5.3.3. Application processing parameters, status, and processing information

To facilitate the processing and tracking of grant submissions, the **TStatus** table is used.

Upon creation of a submission in the system by the allocation of a unique id (*process_id*), a row in the **TStatus** table is immediately created, and the status information entered. As the processing progresses, the status is updated by each individual service that is part of the Submission processing. The status table is queried by each service to determine the next submission to be processed. To provide flexibility in processing, the status table may contain several control fields that can be updated by the system or an operator to control the applications to process. To facilitate the tracking of applications by the help desk and reporting activity, the status table also contains a few de-normalized columns that contain submission information stored here redundantly for performance improvement.

Unique status codes are assigned to each event or submission when they enter a service and exit the process. The status code is also a “reservation” in such a way that only one instance of a service grabs the submission and processes it. When the service is completed successfully, the status is set to the service completion status. The status update is its own independent transaction. To change a status, a locking mechanism is used that prevents other applications from being affected. The “Select for update” with the “no wait” clause database technique is used. The select locks the record. Other queries that attempt to get the same record will then fail immediately and not wait for the record to be released (no locking). The “commit” must immediately follow the status update statement. All services have to test for failure when querying a record and a race condition has occurred. This is accomplished through ***LockRow,*** a stored procedure. A ***ServiceUtility*** class encapsulates the stored procedure and provides higher level Java API’s.

Services also implement retry behavior. To implement the retry effectively without blocking a queue, a delay is built in. The status is reset, or set to a specific retry status, and the retry counter is increased reliably to stop processing when the limit of retries is reached. The retry mechanism, combined with the immediate commit of the status update is used to avoid locking the processing queue and race conditions for the same submission. It also guarantees that the record will be processed only once despite multiple instances of services on multiple servers.

The status update and record locking mechanism has been carefully isolated as an independent transaction which does not perform any other business functions, is extremely fast, and is closed by its own commit or rollback operation. Essentially, the independent transaction consists of:

- Open a transaction
- Select one row and lock it
- Update one row

- Commit the transaction
- Or in case of failure, rolling back and explicitly releasing the lock
- End of transaction

Other resources and other applications are not affected and cannot be locked by the status update statement. There are no foreign keys on the status table. Only one row in one table is locked for a brief moment. The entire operation is designed to be completed in a few milliseconds. The rate of transactions, (application submissions) even at peak periods, makes it highly unlikely that two applications reach the same exact state of processing at the same time. If such contention should occur for a specific row, then the second process trying to access a locked record is designed to wait and retry. In the meantime, the first operation will be completed and committed and the status will have changed.

If the database crashes at the exact moment this transaction is opened, the transactions are rolled back and there are no locks when the database is recovered.

There is no anticipated scenario within the system where the status record would remain locked.

If such a locking event were to occur, for an unanticipated cause, a general monitoring process is being put in place to report on transactions that remain in a working status over a certain period of time, (similar to the "stuck" applications in the earlier system). If an application is not being processed for any reason, the system administrator can change the status manually, by updating the status table with an ad-hoc tool. If the record is locked, the administrator can override the locks.

Note: The "Select for Update" mechanism has a poor reputation since inexperienced users and programmers can lock numerous tables and rows with ill-advised transactions or going for lunch before issuing a "commit" statement. The design for the Grants.gov architecture, as implemented, is a very controlled and discrete transaction, limited to one record in one table followed by an immediate commit or rollback. Therefore, the locking of resources is a highly unlikely event and not a concern.

An alternative to the database based approach, unique selection and locking, can be achieved in the J2EE stack and locking of records in memory. However, the locking must occur across multiple independent containers. This is the primary reason to use the database approach.

By updating the status code to a previous status in the processing cycle, the submission is then eligible again for processing at the next step. It will be automatically selected and re-processed in the next iteration.

Resetting a status code may be a manual update through an operator intervention and a custom SQL statement issued, or it can be built into some processes for automatic re-tries when processes fail. In this case, the retry counter and logic must be activated.

5.3.4. Implementing the Service Infrastructure

The Class Diagram in Figure 11 illustrates the primary abstractions involved in the development of the Service Infrastructure. At the heart of each Service is the Concurrency Manager that controls the number of instances of the Services that need to be started or stopped. The Task Agent provides the thread support for each instance. The Process Monitor monitors each service.

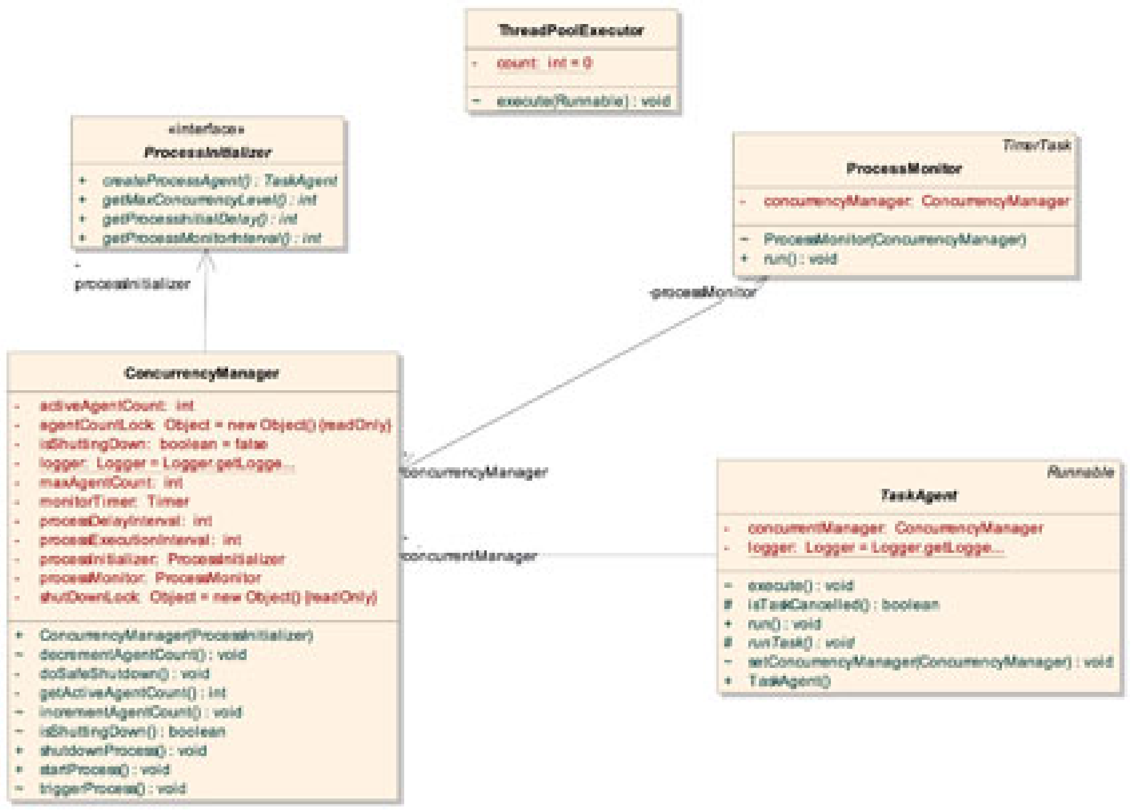

ThreadPoolExecutor
- count: int = 0
~ execute(Runnable) : void
«interface»
ProcessInitializer
+ createProcessAgent() : TaskAgent
+ getMaxConcurrencyLevel() : int
+ getProcessInitialDelay() : int
+ getProcessMonitorInterval() : int
-processInitializer
TimerTask
ProcessMonitor
- concurrencyManager: ConcurrencyManager
~ ProcessMonitor(ConcurrencyManager)
+ run() : void
-processMonitor
ConcurrencyManager
- activeAgentCount: int
- agentCountLock: Object = new Object() {readOnly}
- isShuttingDown: boolean = false
- logger: Logger = Logger.getLogge...
- maxAgentCount: int
- monitorTimer: Timer
- processDelayInterval: int
- processExecutionInterval: int
- processInitializer: ProcessInitializer
- processMonitor: ProcessMonitor
- shutDownLock: Object = new Object() {readOnly}
+ ConcurrencyManager(ProcessInitializer)
~ decrementAgentCount() : void
- doSafeShutdown() : void
- getActiveAgentCount() : int
~ incrementAgentCount() : void
~ isShuttingDown() : boolean
+ shutdownProcess() : void
+ startProcess() : void
~ triggerProcess() : void
-concurrencyManager
-concurrentManager
Runnable
TaskAgent
- concurrentManager: ConcurrencyManager
- logger: Logger = Logger.getLogge...
~ execute() : void
# isTaskCancelled() : boolean
+ run() : void
# runTask() : void
~ setConcurrencyManager(ConcurrencyManager) : void
+ TaskAgent()

Figure 11: Class Service Infrastructure

The interaction of the Service Infrastructure with the Database tables is implemented using Hibernate. The Error, Log, Property, and Status classes represent the Hibernate Data Access Objects. These are shown in the Class Diagram in Figure 12.

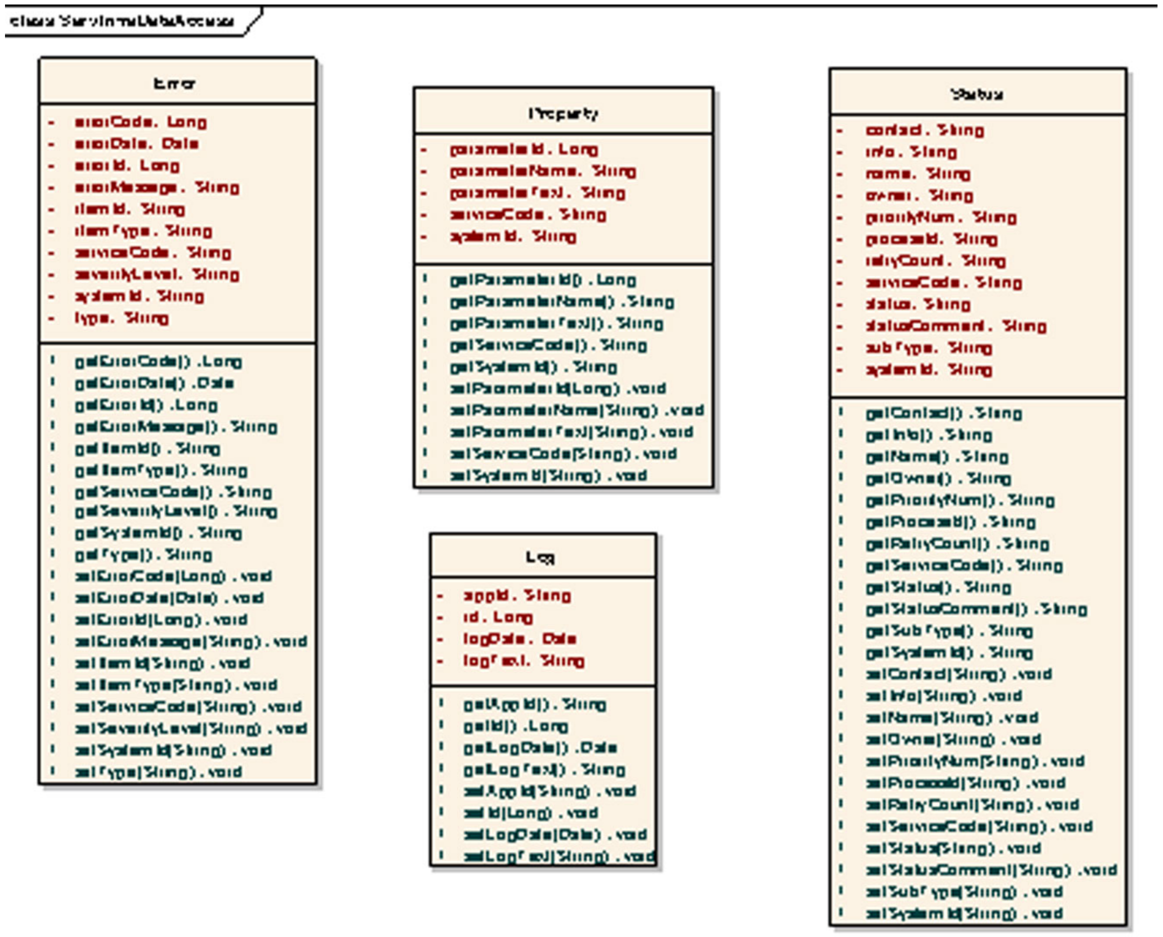

class ServiceInfraDataAccess
Error
- errorCode : Long
- errorDate : Date
- errorId : Long
- errorMessage : String
- itemId : String
- itemType : String
- serviceCode : String
- severityLevel : String
- systemId : String
- type : String
+ getErrorCode() : Long
+ getErrorDate() : Date
+ getErrorId() : Long
+ getErrorMessage() : String
+ getItemId() : String
+ getItemType() : String
+ getServiceCode() : String
+ getSeverityLevel() : String
+ getSystemId() : String
+ getType() : String
+ setErrorCode(Long) : void
+ setErrorDate(Date) : void
+ setErrorId(Long) : void
+ setErrorMessage(String) : void
+ setItemId(String) : void
+ setItemType(String) : void
+ setServiceCode(String) : void
+ setSeverityLevel(String) : void
+ setSystemId(String) : void
+ setType(String) : void
Property
- parameterId : Long
- parameterName : String
- parameterText : String
- serviceCode : String
- systemId : String
+ getParameterId() : Long
+ getParameterName() : String
+ getParameterText() : String
+ getServiceCode() : String
+ getSystemId() : String
+ setParameterId(Long) : void
+ setParameterName(String) : void
+ setParameterText(String) : void
+ setServiceCode(String) : void
+ setSystemId(String) : void
Log
- appId : String
- id : Long
- logDate : Date
- logText : String
+ getAppId() : String
+ getId() : Long
+ getLogDate() : Date
+ getLogText() : String
+ setAppId(String) : void
+ setId(Long) : void
+ setLogDate(Date) : void
+ setLogText(String) : void
Status
- contact : String
- info : String
- name : String
- owner : String
- priorityNum : String
- processId : String
- retryCount : String
- serviceCode : String
- status : String
- statusComment : String
- subType : String
- systemId : String
+ getContact() : String
+ getInfo() : String
+ getName() : String
+ getOwner() : String
+ getPriorityNum() : String
+ getProcessId() : String
+ getRetryCount() : String
+ getServiceCode() : String
+ getStatus() : String
+ getStatusComment() : String
+ getSubType() : String
+ getSystemId() : String
+ setContact(String) : void
+ setInfo(String) : void
+ setName(String) : void
+ setOwner(String) : void
+ setPriorityNum(String) : void
+ setProcessId(String) : void
+ setRetryCount(String) : void
+ setServiceCode(String) : void
+ setStatus(String) : void
+ setStatusComment(String) : void
+ setSubType(String) : void
+ setSystemId(String) : void

Figure 12: Class Diagram

The Class Diagram in Figure 13 shows the Service specific classes that need to be implemented to associate a Service with the Service Infrastructure. A Typical Service requires a *ServiceTaskAgent, ServiceManager, and ServiceProcessIntializer*.

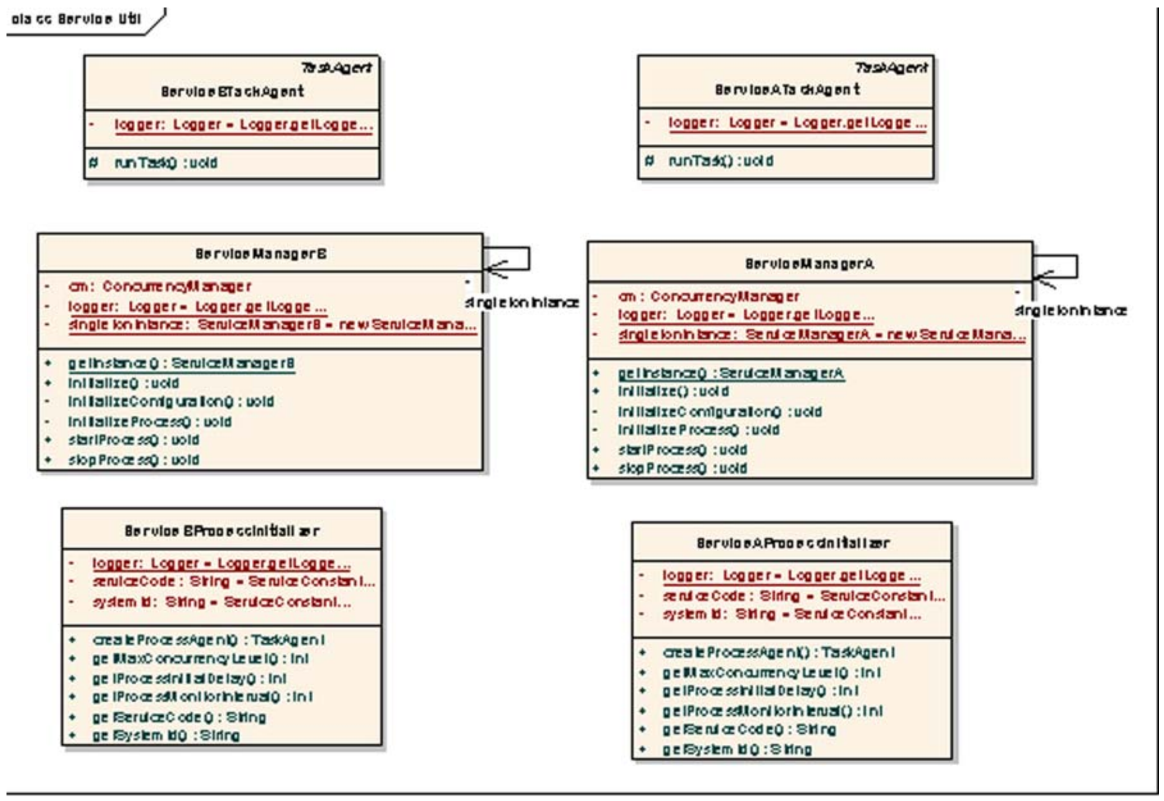

class Service Util
TaskAgent
ServiceBTaskAgent
- logger: Logger = Logger.getLogge...
# runTask() :void
TaskAgent
ServiceATaskAgent
- logger: Logger = Logger.getLogge...
# runTask() :void
ServiceManagerB
- cm : ConcurrencyManager
- logger: Logger = Logger.getLogge...
- singleton Instance: ServiceManagerB = new ServiceMana...
+ getInstance() : ServiceManagerB
+ initialize() :void
- initializeConfiguration() : void
- initializeProcess() : void
+ startProcess() : void
+ stopProcess() : void
-singletonInstance
ServiceManagerA
- cm : ConcurrencyManager
- logger: Logger = Logger.getLogge...
- singletonInstance: ServiceManagerA = new ServiceMana...
+ getInstance() : ServiceManagerA
+ initialize() :void
- initializeConfiguration() :void
- initializeProcess() :void
+ startProcess() :void
+ stopProcess() :void
-singletonInstance
ServiceBProcessInitializer
- logger: Logger = Logger.getLogge...
- serviceCode : String = ServiceConstant...
- system Id: String = ServiceConstant...
+ createProcessAgent() : TaskAgent
+ getMaxConcurrencyLevel() : int
+ getProcessInitialDelay() : int
+ getProcessMonitorInterval() : int
+ getServiceCode() : String
+ getSystemId() : String
ServiceAProcessInitializer
- logger: Logger = Logger.getLogge...
- serviceCode : String = ServiceConstant...
- system Id: String = ServiceConstant...
+ createProcessAgent() : TaskAgent
+ getMaxConcurrencyLevel() : int
+ getProcessInitialDelay() : int
+ getProcessMonitorInterval() : int
+ getServiceCode() : String
+ getSystemId() : String

Figure 13: Class Service Util

6. Data Architecture

The extended capabilities schema (April 2007) is based on the legacy schema. The Architecture goal has been to retain the existing schema and make appropriate minor modifications. Additional columns have been added to some existing tables and a few new tables have been introduced. The one significant change in the extended capabilities Data Architecture is the usage of **BFiles** instead of **BLOBs** in some of the tables. There are ten tables in the legacy database schema that have the **BLOB** data type column. The tables include the following:

- **TATTACHMENTS**
- **TCOMPETITION**
- **TFAMILY_COVER**
- **TFILE**
- **TFORM**
- **TFORM_DIALECT**
- **TINSTRUCTIONS**
- **TPACKAGES**
- **TSYNOPSISATTACHMENT**
- **TSYSDATA**

Among these tables, **TATTACHMENTS** stores the submitted forms and various data files. **TFILE** stores the submitted files in three file types ready for agencies to download.

BFILES are stored in a directory that is SAN mounted. Files on the SAN are backed up. Both the BFILES and the database are independently backed up. We will look into any coordination issues that might arise.

Because of the size of data, database transactions against the two tables had the potential to be constrained and become the potential performance bottleneck of the system during peak application submission time. **BFILE** is an alternative to **BLOB**. It does not store the file itself in the tables, but instead stores a file locator, which points to the file in the file system of the database server, thereby saving the processes for transactions across the database. In order to accomplish the changes, two columns FILE_ID and FILE_LOCATION have been added to the TATTACHMENTS table.

7. Deployment Architecture

The Deployment Architecture for the extended capabilities system is shown below. The current collection of Apply Agency Servers is being used to deploy the extended capabilities system. A Group of Servers was dedicated to handle Adobe API related interaction. Figure 14 also shows the new Google Search Application Cluster. The "BEA Weblogic Cluster" indicated in the Figure 14 indicates a "Grouping or Server Collection". There is no "clustering" proposed for load balancing or transparent fail over.

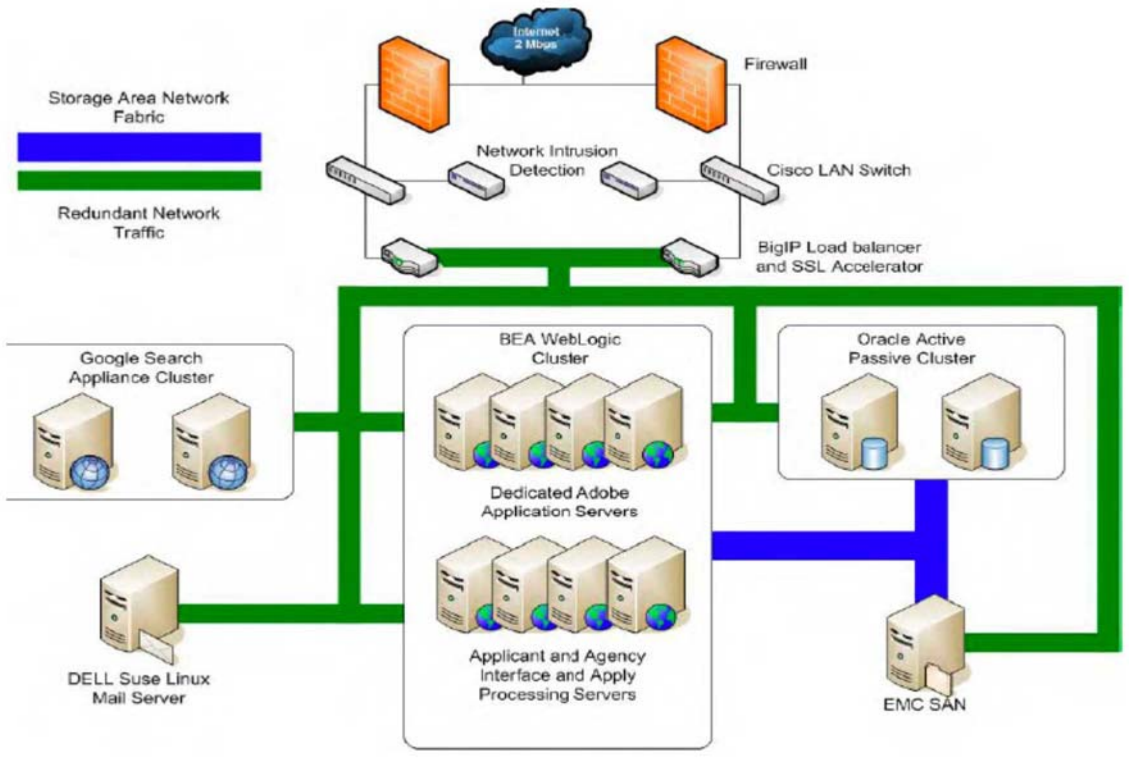

Internet 2 Mbps
Firewall
Storage Area Network Fabric
Redundant Network Traffic
Network Intrusion Detection
Cisco LAN Switch
BigIP Load balancer and SSL Accelerator
Google Search Appliance Cluster
BEA WebLogic Cluster
Oracle Active Passive Cluster
Dedicated Adobe Application Servers
Applicant and Agency Interface and Apply Processing Servers
DELL Suse Linux Mail Server
EMC SAN

Figure 14: Deployment Architecture

Each Weblogic application server will deploy four managed servers corresponding to search, apply, agency system-to-system (Agency S2S), and application system-to-system (Applicant S2S) ***.war*** files.

We recognize that the single Mail Server that is being used from the legacy system is a single point of failure. We will look into providing some redundancy.

8. Acronyms

Term	Definition
API	Application Programming Interface
DAO	Data Access Objects
EJB	Enterprise JavaBeans
HTML	Hyper Text Mark-up Language
HTTP	Hyper Text Transfer Protocol
J2EE	Java Platform Enterprise Edition
JSP	Java Service Page
MVC	Model View Controller
PDF	Portable Document Format
POJO	Plain Old Java Objects
RMI	Remote Method Invocation
S2S	System-to-System
SQL	Standard Query Language
XML	Extensible Markup Language