GLASSFISH™ ENTERPRISE SERVICE BUS (ESB)
HIGH AVAILABILITY AND CLUSTERING

White Paper
December 2009

Authors:

Mike Somekh
Mark Foster
Rastislav Kanocz

Acknowledgments:

The authors would like to thank Andreas Egloff and Gabor Puhalla from Sun Microsystems for their thorough reviews and contributions during the whitepaper creation.

Table of Contents

Executive Summary

GlassFish Enterprise Service Bus (ESB) is Sun Microsystem's lightweight and agile integration product suite for services-based and composite application development. The ESB is typically at the heart of an enterprise-level business application platform. Most deployments require serious consideration of non-functional requirements, including performance, reliability and availability, and for most mission-critical applications High Availability (HA) is a key requirement. HA is employed to ensure a certain degree of operational continuity within a given time period. Although the ideal scenario is one where all systems would be implemented to be available to the "9 nines" standard with little latency and no single point of failure — anywhere in the system, other factors outweigh this ideal.

While there is no single solution or strategy that can be employed for all deployments, there are multiple options and techniques that can be used to make GlassFish ESB applications highly available, which are discussed in this whitepaper. Enterprise Middleware and Integration Architects as well as GlassFish ESB Developers can use this information about clustering and other techniques to architect highly available integration solutions using GlassFish ESB.

Clustering is a way of deploying a number of components, allowing them to work together to improve availability and performance. Clustering can be implemented at the operating-system level, application-server level and ESB level. Virtualization and Disaster Recovery can also be used to further augment the availability measures and these are also covered in brief.

The scoping and initial analysis of availability requirements is a crucial first step to the deployment of an HA solution. This whitepaper follows the analysis of key criteria to architect a highly available GlassFish ESB solution right from the planning stage. This analysis includes defining acceptable levels of downtime, the degrees of latency allowable, divisions between semi-static data and time-critical data and options for recovery. All of these requirements must be considered together to arrive at a solution; these need not apply only to the monolithic final solution and can be used at the subsystem level as well. Also in scope is an examination of architecture and implementation considerations that may affect not only the initial implementation, but future developmental iterations as well, including Transaction Management (XA), asynchronous message correlation, compensation and Quality of Service attributes.

Following this initial analysis, the whitepaper details a reference architecture for a very typical deployment solution based on a real-world scenario. The scenario focuses on a system that manages a customer's business processes. The solution accepts requests via web services-based interfaces, manages corresponding business processes, handles faults correctly and integrates with existing backend systems. It includes a web-based application and requires asynchronous message correlation. The scenario was built and tested in a developer environment without HA, and then designed and deployed to become highly available. Products covered include the HTTP Binding Component, BPEL Service Engine, the Database Binding Component, EJBs (Java EE Service Engine) and an ICEFaces web application. JMS is not used at this time but is briefly covered in this discussion.

The deployment architecture is described in detail, including the architecture tiers and various scalability and availability strategies employed. Some of the implementation patterns (related to GlassFish ESB and used in deployment architecture), certain limitations and possible workaround are also discussed. The primary purpose of this whitepaper is to make this overview and referential information available to architects and developers so they can extend these concepts by example to their own project requirements.

Chapter 1

Introduction

GlassFish ESB, Sun Microsystem's integration product suite, is based on OpenESB. OpenESB is an Open Source project that delivers a platform for business integration, Enterprise Application Integration (EAI), and Services-Oriented Architecture (SOA). Based on a large number of standards, such as JBI, Java EE, SOAP, WS-* and so on, OpenESB allows enterprises to build flexible, robust solutions for integrating their systems using a large number of components including binding components (adaptors) and service engines (processors).

GlassFish ESB is a *binary distribution* of OpenESB components for which commercial support is available from Sun Microsystems. It relies on OpenESB, Netbeans™ IDE and the GlassFish Application Server. Its components can leverage existing POJOs and standard EJBs to compose solutions, can be used for development with common version control systems such as Subversion or CVS, and managed and monitored through standard interfaces such as JMX and so on.

Integration solutions based on GlassFish ESB can be deployed in multiple ways, depending on business requirements and network and hardware topologies. The broad spectrum of varying project requirements makes it impossible to offer a standard optimal solution for all GlassFish ESB business integration projects. For example, some human workflow web applications requiring near-instantaneous failover may utilize application-server clustering as well as load balancers. On the other hand, long-running batch-processing applications that can accept comparatively longer periods of downtime can be implemented using applications running in an active/passive configuration.

This whitepaper discusses the different options available to Enterprise Middleware and Integration Architects and also GlassFish ESB Developers for building highly available and scalable applications with GlassFish ESB. It is safe to assume that although the reference architecture depicted in this whitepaper demonstrates an industry-specific scenario by example, this paper aims to be generic to the technology and that the recommended techniques can be extended to any industry.

Chapter 2

The Need for High Availability and Clustering

Enterprise integration architects designing ESB systems have a number of important considerations to take into account. Functional requirements dictate that the ESB provide a set of typical services such as message routing, data transformation and application adaptors. However, the ESB is typically at the heart of an enterprise's business application platform and successful deployments require deeper consideration for other non-functional requirements, including *performance, reliability* and *availability.*

The term, Highly Available, is therefore applicable to systems that are designed to ensure a certain degree of operational continuity within a given timeframe. This is often measured in terms of a number of nines, representing the percentage uptime of the ESB. For example, over the period of a year:

- 2 nines or 99% represents 3.65 days of acceptable downtime per year
- 5 nines or 99.999% represents 5 minutes of acceptable downtime per year
- 7 nines or 99.99999% represents just 2 seconds of acceptable downtime per year!

Applications built around an ESB consist of multiple parts; therefore any business application may have multiple points of failure. This is especially true of an ESB that provides many different services itself, and by definition, has disparate touch-points or interfaces with multiple external systems. For example, a web application that uses GlassFish ESB to interface with multiple databases could be implemented using EJBs, JBI binding components, BPEL processes, JMS and so on. Failure of any component may have a serious affect on the business application's availability. Therefore one important consideration is to identify and manage all single points of failure.

Clustering is a way of deploying a number of components, allowing them to work together to improve availability and performance, compared to what can be achieved by deploying just one component. Clusters can provide added reliability using failover capabilities, which means that when a component fails, another takes over and provides the same set of services. This is often done transparently so that clients are not aware of the failover, be it a software or hardware component that goes down. Components often need to be cluster aware to participate in a cluster.

Availability can often be broken down into service availability and data availability. When a component fails (business process, an application server, a machine or a data storage device), ideally both the services should be failed over (so that processing can continue) as well as state (so that the latest data is available).

A related aspect is known as fault tolerance. This is the ability of a system (application architecture) to resume operation in the event of a hardware or software failure. Systems that are designed with fault tolerance in mind often employ redundant components that take over processing when another goes down. These redundant components may themselves be active, where they are running in tandem with the primary deployed components, or passive, where they are engaged only when their peer component fails.

Chapter 3

Scope

Enterprise architects can meet disparate reliability and fault tolerance requirements for ESB-based applications using some or all of the clustering approaches discussed in this whitepaper.

In scope

Clustering options for building highly available solutions using GlassFish ESB include:

- Hardware and operating system-level clustering
- Application server-level clustering
- ESB-level clustering (GlassFish ESB)

Each of these options will be discussed in detail along with how they can be used in isolation or in tandem and the pros and cons of different solutions.

Out of scope

High availability and clustering, especially when applied to integration solutions are large topics. While much of the information here is applicable elsewhere, the following subjects are not in scope for this whitepaper:

- Database-level clustering
- Disaster Recovery

  Used for deployments where high levels of downtime are acceptable.
- Virtualization technologies

  These do not fit strictly within the scope of this paper. However, the reader can use the following references for more information on the subject:
- Solaris Containers

  http://blogs.sun.com/JeffV/entry/high_availability_networking_for_solaris
- OracleVM

  http://www.oracle.com/us/technologies/virtualization/index.htm
- LDoms

  http://whitepapers.techrepublic.com.com/abstract.aspx?docid=1105085
- Hyper-V

  http://blogs.msdn.com/clustering/archive/2008/06/21/8628515.aspx
- VMware

  http://www.vmware.com/products/high-availability/overview.html

Chapter 4

Clustering Options Applicable to GlassFish ESB

As discussed earlier, even though various different types of clustering options are available for implementing highly available solutions using GlassFish ESB, it is important to note that it is rare for any single one of these to be considered a complete solution and they should be considered in parallel.

OS-level and hardware-level clustering (Solaris and HA hardware)

Sun's OpenSolaris™ provides cluster capabilities at the operating system level. This means that a process running on one machine can be transparently failed over to another machine without loss of service. It is often used as a part of a hardware cluster (such as Solaris Cluster) that provides features such as shared storage, cluster communication and administration.

Sun's Open HA Cluster provides a simple, thoroughly tested solution for open software components, such as MySQL™ and Apache, as well as customer applications. OpenSolaris™ has several features designed to avert downtime or to automate the recovery process to minimize both the downtime and cost. Predictive Self Healing, featuring Solaris Fault Manager and Service Management Facility, provides intrinsic availability features. Open HA Cluster uses hardware redundancy and advanced monitoring of all components of the systems to protect against hardware faults. See http://www.OpenSolaris™.com/learn/features/availability for more information.

Application server-level clustering (GlassFish Enterprise Server)

Sun's GlassFish Enterprise Application Server, which provides the run-time for JBI components and the deployed GlassFish ESB projects, supports clustering by grouping application server instances. Each cluster is considered to be a logical managed unit and instances share a common configuration, host identical applications and can be managed and monitored centrally.

GlassFish therefore supports both high-service and high-data availability using techniques such as in-memory session replication (where state in one user session is available in multiple GlassFish instances even if one or more instances go down). By using multiple instances available for processing requests, applications deployed to GlassFish can be made more scalable and performance can be improved.

Components that are deployed to a GlassFish cluster must be cluster-aware, which means they must be designed to work in a cluster as well as a single instance. It is therefore important to ensure that the GlassFish ESB components used in the deployment can support clustering and are configured correctly. For example, in order for a business process running on an application server instance to be completed transparently when that instance goes down, the BPEL engine was developed to store process state into a persistent store (in this case, a database). If an application server that 'owns' the instance fails, another takes ownership to complete processing it from the point of failure by reading its state from the database. All BPEL service engines must therefore point to the same database tables for this to work. Clearly, the database needs to be highly available to ensure that it does not become a single point of failure itself.

Note: The GlassFish Enterprise Server has the option of an enterprise profile which provides HADB and allows it to have even greater level of HA (99.999%). Although HADB is not a part of GlassFish ESB, GlassFish ESB does provide an in-memory session replication option for high data availability as explained above. The memory replication feature takes advantage of the clustering feature of GlassFish to provide most of the advantages of the HADB strategy with much less installation and administrative overhead. In addition, using HADB increases the availability of state for EJBs and web applications, it does not affect the availability of other GlassFish ESB components (or stateless Java EE components). It is also possible to combine a web tier utilizing HADB.

ESB level clustering (GlassFish ESB Components)

All GlassFish ESB components are supported in a clustered environment. Some components work inherently in a clustered environment and some require special configuration, notably in the areas of load balancing and failover. Stateful GlassFish ESB service engines (SEs), such as BPEL SE and IEP SE, use a persistence database to track cluster instances and distribute work. Other components, such as the Scheduler Binding Component (BC), Data Mashup SE and LDAP BC, work inherently in a clustered environment without needing to know about the different instances.

The BPEL SE and IEP SE provide additional support for clustering at the component level; this means using multiple stand-alone GlassFish instances that operate at the ESB level as a cluster. This is not a question of backwards compatibility, but can be easier to manage in environments that require dynamic sizing (spinning up new equivalent instances on demand, for example on virtualized platforms), setting up active-passive systems, and so on. This allows for persistence, high-availability, and failover without using a GlassFish server cluster for all components.

Load balancing and failover functions depend on the individual components in the cluster. For example, load balancing is not applicable to service engines and is protocol-specific for some of the binding components. The File BC, JMS BC and FTP BC all have built-in load balancing, while any hardware or software HTTP load balancer can be used for distributing SOAP transactions (for example, the HTTP Load Balancer plug-in or Sun Java Web Server's Reverse Proxy feature). The HTTP Load Balancer distributes incoming HTTP and HTTPS transactions among the instances in a cluster and also fails over requests to another server instance if the original server instance becomes unavailable. The HTTP BC in GlassFish ESB does not have its own load balancing capabilities, making the HTTP Load Balancer plug-in a useful tool when implementing the HTTP BC in a clustered environment. Load balancing and failover for the JMS BC is provided through a JMS broker cluster.

A detailed discussion of how GlassFish ESB components support ESB-level clustering is beyond the scope of this whitepaper. However, more information and implementation-level details are available at the following link: JBI Component Cluster Support

Database-level clustering

The specific details of database-level clustering are beyond the scope of this document. Although a typical HA architecture will certainly include a highly available database, the implementation details of this is not really relevant to how a highly available GlassFish ESB will function. Suffice to say that there are many options available, such as MySQL™ HA, Oracle RAC and so on. These solutions are used, for example, as the persistence layer for the BPEL SE. These highly available DB solutions are not to be confused with HADB; both are essentially different things.

Java Message Service (JMS)

Java Message Service (JMS) allows application architects and developers to use a common messaging interface for components to communicate. JMS can be used to implement various patterns efficiently, for example, asynchronous publish and subscribe, request reply or broadcast. JMS is particularly useful within the context of a highly available solution. For example, it can allow producers to continue functioning even when consumers are down and vice versa. It can ensure that messages delivered to a component when it was down are delivered when the component recovers. Other features such as message 'throttling' can also help to keep the system running.

However, JMS providers (implementations) also comprise running components and are subject to the same level of consideration as other parts of the architecture. There is little point in clustering other components if the JMS messaging layer itself is a single point of failure, therefore many implementations now make their JMS provider highly available. Data availability can be ensured by persisting messages to an underlying highly available message store or alternatively, replicated across multiple message stores that are kept in sync. Service availability is often provided by having a cluster of JMS nodes, any of which can service a request, and an API that allows clients to transparently fail over to other nodes if they lose a connection.

Sun's Java MQ offers multiple clustering modes, five nines availability using MSQL cluster or HADB, wildcard destinations, schema validation, fine management and tuning capabilities and a growing list of features. A cluster of brokers services requests using a single message data store.

Chapter 5

Analysis — Where to Start?

High Availability is employed to ensure a certain degree of operational continuity within a given time period. Ideally, all systems would be implemented to be available to the 9 nines standard with little or no latency and no single point of failure — anywhere. In reality, other factors outweigh this ideal, especially the cost of developing and maintaining such a solution when it is not needed or when other, simpler solutions will suffice. Determining the degree of availability required for a predefined time frame is therefore a pre-requisite and an appropriate starting point for the analysis and planning.

What is an acceptable level of downtime?

Consider a bank's batch processing system that number-crunches a large amount of data overnight to generate reports using GlassFish ESB. If this long-running batch process is not available for 10 minutes to a couple of hours because it takes this long for a backup to come online and to be populated with the correct data, then this delay may be completely acceptable. Reports will be generated in time by the following morning.

However, the same bank has a completely different tolerance to downtime for their mission-critical intra-day trading system. If the service for placing trades goes down, all business stops and it would be an unacceptable risk if this happens for too long a period. In this case, the downtime of 15 seconds may be the limit — any more than that and the bank risks being over-exposed. This is also true for the sub-systems of the business solution. So the GlassFish ESB application server would need to be clustered so that trading can continue transparently without need for manual intervention within seconds.

However, if these systems can use what is called *semi-static data* (for example, data that rarely changes from day to day such as a company name, daily historical records or even interest rates), then the acceptable downtime of those sub-systems could indeed be different. In this case, if the database goes down, running GlassFish ESB applications could still function correctly with the *comparatively stale* semi-static data, for hours or even the entire day until the database is brought back online.

Asking the right questions about this application uncovers key criteria to arrive at the availability requirements:

- One sub-system (such as the application running on the server) has a short acceptable downtime for service availability requiring clustering.
- Others (such as a database) have a much longer acceptable downtime and will *not* need to be clustered or made highly available.
- Or, all sub-systems may need to be highly available in unison.

What about options for recovery?

Another important pragmatic consideration is how the system or sub-systems must recover from the failure of their own component or a dependent component. Data recovery involves salvaging the data that is being managed by a failed component (for example, a database), and service recovery similarly deals with the ability to restart a failed service. Either can be automatic or manual and both methods are valid, although automatic recovery, which is the primary focus of this document, is preferred.

Mission-critical systems require automatic system recovery where software is configured to detect a failure, determine the nature of the problem and take appropriate action. A good example would be a GlassFish ESB application (service assembly) deployed to a cluster where the processing of BPEL business processes continues when one node fails because the BPEL Service Engines coordinate and take ownership of processes that are stuck. Another example is a load balancer that automatically redirects HTTP requests to different servers when communication with one times out.

However, probably for most deployments, some level of manual recovery is also acceptable. Disaster recovery sites are often created to handle *catastrophic, uncontrolled or unpredictable* scenarios such as a data centre being taken offline. By definition, the events that cause the disaster recovery site to come into operational use are rare. Severe flooding, acts of terrorism or power blackouts often require more than just technical system recovery. Semi-manual procedure such as daily data replication, and manual procedures, such as transferring equipment or even personnel, are often just as important.

This does not imply that a fully automated highly available solution cannot be deployed between a primary site and a disaster recovery site. Dedicated private networks with high bandwidth, software architecture techniques such as HA applications servers and solutions based on HA JMS, replicated data stores and so on can all be implemented across different geographic and time zones.

However, it is essential to evaluate the benefits of implementing and maintaining such a solution. If an online account system for electricity users goes down, the high cost of a fully automated solution may not necessarily prove cost-effective compared to a manual one where support and operational staff are called in to help.

Performance is also a consideration. How much data is it feasible to transfer and store solely to provide automatic recovery? And what effect does this have on performance, throughput and latency? Your analysis should therefore determine which points of failure must be recovered and to what degree.

Should we consider combinations?

As mentioned in the previous section, there are multiple levels at which HA can be applied, from the coarse-grained OS-level to the finer-grained ESB level. Most HA architectures will necessarily consist of a combination of all of these levels.

Granularity of the clustering is an important consideration. In general:

- OS-level clustering should be applied to the failure of large sub-systems, such as a network, storage or a large application (such as a CRM).
- Application server-level clustering should be applied to critical failures of the application server itself (for example when it runs out of memory) or when an application fails to respond.
- GlassFish ESB provides fine-grained control over service units deployed to a JBI container. In this case, individual business processes or database binding component service units can handle failures themselves.

What are the architectural and implementation-level considerations?

There are certain constraints that some requirements impose on a highly available solution. Even though some of these requirements may not be evident in the initial project phases, it's important to consider whether or not they will become a requirement in a future iteration of the project, as it is difficult to retrofit a new architecture pattern down the line.

These considerations include Compensation Handling, Asynchronous Message Correlation, Transaction Management (XA) and Quality of Service requirements. These are areas which could be considered a part of availability, if not high-availability, and will be briefly touched upon in this document as they are very scenario-specific. For example, it might not make sense to use XA in a long-running BP, but may make perfect sense in a short-running business process or EJB; the same goes for compensation. Also there could be an impact of using one of these on using another, for example using QoS to redirect a message to a different endpoint (see explanation below) might make XA inapplicable as in effect we are removing the initial failing endpoint from the process. One thing is certain, a knowledgeable enterprise architect can only make decisions about these based on specific scenario requirements.

Message correlation

Consider a typical pattern: a business process exposed as a web service where a number of activities are executed during the lifetime of the business process, and where one of the activities waits for a correlated message. There are various reasons why the business process may be halted unexpectedly. For example, the application server running out of memory, the container stopping unexpectedly, the host machine crashing or the network cable being unplugged so that access to the shared database is lost.

Typically, the business process would be implemented using BPEL. GlassFish ESB provides a level of high availability for processes implemented in BPEL out of the box. Therefore, in this case, the process will be detected as having *stalled* by other BPEL Ses and one will take ownership, ensuring that the process completes as expected. This means that the client application, or indeed the load balancer, which instantiated the business process and ensures that correlated messages are being sent to that BPEL SE instance, does not need to know which engine now owns the business process. A correlated message sent to any BPEL Service Engine will be processed.

However, often there is a requirement to implement (or indeed re-implement) the business process in Java, for example, for improved performance or for more fine-grained control. The same web service could still be exposed but the implementation would rely on the Java EE service engine. In this case, the client application that instantiated the business process will not know which engine now owns the business process.

Another solution would be to use JMS Topics. The client sends a message to one end point (a JMS topic) and all potential consumers of the service subscribe to that topic. Each of them reviews and continue to either process or discard the message depending on ownership.

Compensation handling

Compensation Handling is an infrequently used concept in BPEL. BPEL has no concept of rollback as such, each activity is normally a separate unit-of-work (XA is a special case). In the event of a failure in the process, normally a business failure as opposed to a technical failure, the only way to rollback is to issue compensating activities, e.g. at a certain point in a business process a row is added to a table, later on there is a business failure, a compensating activity could remove this row. Compensation brings with it other problems, again due to transactionality; for example, what if the compensating activity itself fails? Compensation can be powerful but should be used with care and a full understanding of the consequences.

XA transactions

XA is a widely used and understood n-phase-commit protocol for distributed transaction processing, but it brings its own share of considerations that need evaluation. It makes very little sense using XA transactions in a long-running business process, in which case persistence is a better solution. Generally, BPEL processes cannot initiate an XA transaction; some other XA-compliant object must initiate the transaction (such as a JMS receive for example). For more details about initiating XA transactions using BPEL SE, refer to the BPEL SE documentation. There is also debate about whether or not XA transactions are in fact applicable to or usable with BPEL at all, because in exceptional circumstances, it can be almost impossible to configure and use correctly. For a better description of the possible problems with BPEL, see the paper written by Pymma (a GlassFish ESB partner).

The in-memory service composition option of GlassFish ESB allows XA transactions to span the full service composition, meaning that all relevant activities will complete or everything will be rolled back. It does this by receiving from an XA capable resource and propagating the transaction context to multiple processing steps (service providers), before invoking other XA capable resources and committing the transaction. A word of caution on parallel invocations: like most XA transaction managers the GlassFish transaction manager only supports one service to operate on a transaction at one time.

This gives the user the option to choose the appropriate trade-offs between a low overhead in-memory composition or adding more persistence points. The user may choose the in-memory service composition where multiple steps need to be done in an atomic fashion or where it is acceptable that, in the failure case, multiple steps get re-done. Alternatively, the user can explicitly choose the persistence points (and related transaction boundaries) where they are desired by inserting them into the service composition (e.g. adding a persistent JMS queue, or a database, which will act as a de-facto transaction boundary).

While GlassFish ESB supports JMS, it does not rely on persistent JMS in between every service to ensure scaling, HA and no message loss. Please refer to the section on QoS for capabilities such as at-least-once delivery.

QoS — Quality of Service

QoS is a very useful feature within GlassFish ESB. The connection between components can be configured for exceptional conditions, such as when a message cannot be delivered from the HTTP BC to the BPEL SE because the latter is not available. Normally, this would result in an immediate response from the normalized message router (NMR) to the HTTP BC that the endpoint BPEL SE was not available causing the HTTP BC to terminate. QoS allows for the addition of retry and redirect functionality to this connection; for example, in the previous example, it would be possible to tell the HTTP BC to retry the connection “n” times at “m” second intervals. And if this fails as well, to redirect the message to another endpoint, which could be a JMS destination for later processing for example.

Within GlassFish ESB the baseline guarantee is “at-least-once-delivery”; however options exist for “at-most-once-delivery (discard failures)”, or to combine with XA for truly atomic “once-and-only-once-delivery”. Combining unique message identifiers with at-least-once delivery allows customer logic (and specific components such as BPEL) to also identify and remove duplicates.

Chapter 6

Reference Architecture

There are many scenarios that would be useful to demonstrate high availability and clustering with GlassFish ESB. Here we look at a very typical deployment solution, based on a real-world implementation which includes more common GlassFish ESB components. Readers can extend this scenario by example to their own requirements.

Scenario

The scenario focuses on a typical requirement for a system that manages the customer's business processes. The solution accepts requests via web services interfaces, manages corresponding business processes, handles faults correctly and integrates with existing back-end systems.

The scenario was first built and tested in a developer environment without high availabillity. This reference architecture example focuses on how it was designed and deployed to become highly available. Components used include:

- BPEL Service Engine
- HTTP Binding Component
- Database Binding Component
- EJBs (Java EE Service Engine)
- Web application
- JMS (not used in reference architecture)

Functional description

The scenario is a Loan Application process for a Bank. A user or application makes a request for a Loan, providing details such as Name, Salary and the Amount of the Loan requested. If the data is valid, the Bank's business process first archives the record into a database and then responds with a unique Application Number and a Status which can be one of the following, depending on the Loan-to-Salary ratio:

Table 1. Loan application process

Status Action	Loan/Salary ratio	Description
"A" Approved	< 33%	Immediate approval with no review necessary and the business process terminates
"D" Denied	> 66%	Immediate refusal and the business process terminates
"C" Check Required	>= 33% and <= 66%	Business process continues; waits for manual confirmation or denial

A manual check is required if the loan is between 33% and 66% of the applicant's salary. In this case, business process instances continue running, awaiting either a corresponding message, approving or denying the loan application, or timing-out. In other words, the business process waits for an asynchronous message which is correlated on the application number.

The diagram below shows the business process initiated by a Loan Application request and, for relevant cases, completed by a CreditReference request.

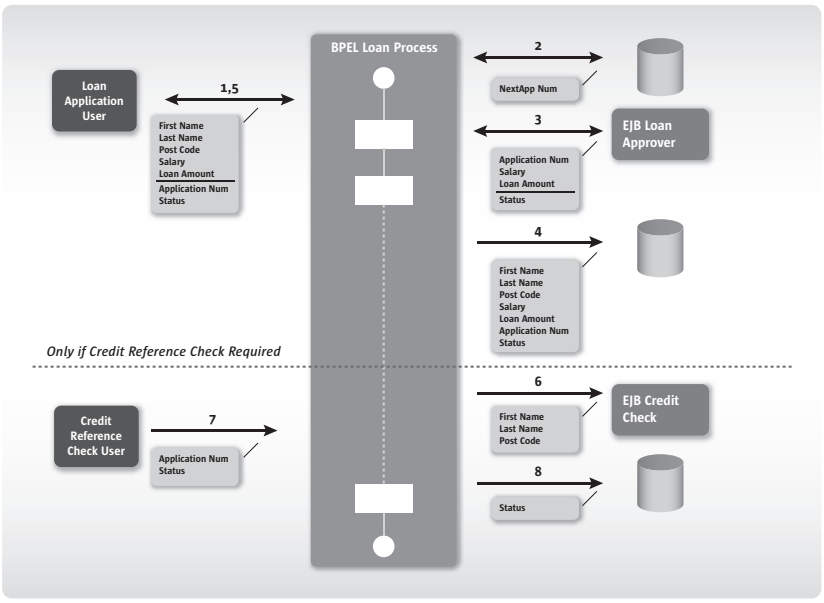


Figure 1.

A detailed explanation of actions done at the component level follows:

1. The User enters the Loan Application details (First Name, Last Name, Post Code, Salary, Loan Amount) using a web application form or SOAP request. The web service call in the form of a SOAP request is accepted and processed by the HTTP BC. The HTTP BC invokes the business process handled by the BPEL SE.
2. The BPEL SE calls the stored procedure used to access the Data Store via the Database BC, which returns the next available Application Number. The Application Number is stored for further correlation if necessary.
3. The BPEL SE calls the EJB Loan Approver handled by the Java EE SE to determine the status of the loan application.
4. The BPEL SE calls the Database BC to insert loan details into the database table.
5. BPEL SE then sends the Application Number and status back to the user over HTTP BC as a SOAP response.

Further steps are only carried out if a credit check is required.

6. If a credit check is required, the BPEL SE calls the EJB Credit Check handled by the Java EE SE — in this phase this is a “dummy” EJB which just logs a message to the GlassFish server log.
7. User enters Credit Check details (Application Number, Status) using the web application form or SOAP request. The web service call in the form of the SOAP request is accepted and processed by the HTTP BC. The HTTP BC invokes the business process handled by the BPEL SE. This business process is correlated on the application number; there may be many process instances awaiting credit checks, the correct instance with the matching application number is restarted, and the remaining instances are left in a *waiting* state.
8. The BPEL SE calls the Database BC to update the application with the new status.
9. The business process ends.

The following diagram depicts the component interaction for this scenario in the a distributed and HA environment:

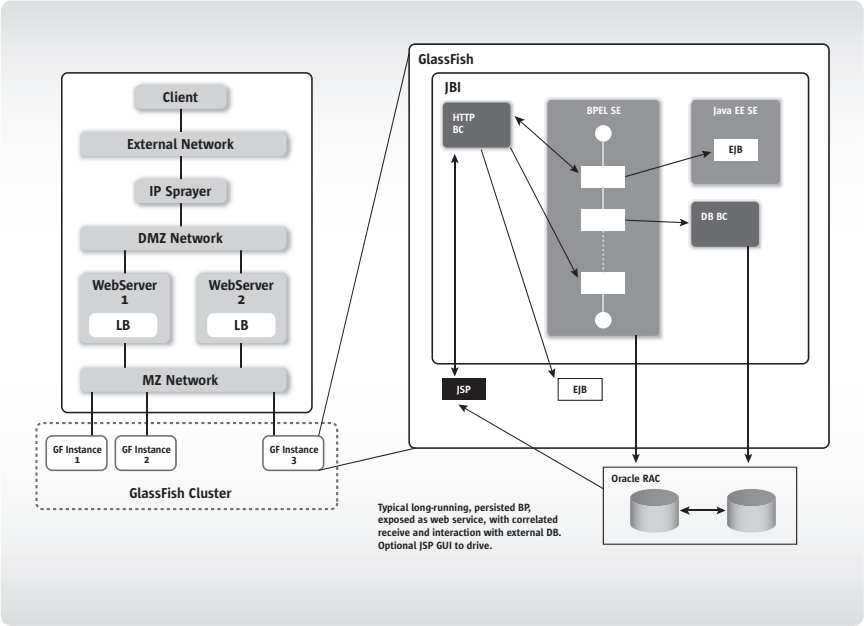


Figure 2.

Deployment architecture

The design of the solution architecture consists of a two-step process: first analyzing and developing the logical architecture and then developing the deployment architecture, as described in the following sections.

Logical architecture diagram

A logical architecture describes and shows the software components and the interactions between them. This is used to arrive at the required set of infrastructure services within a tiered framework.

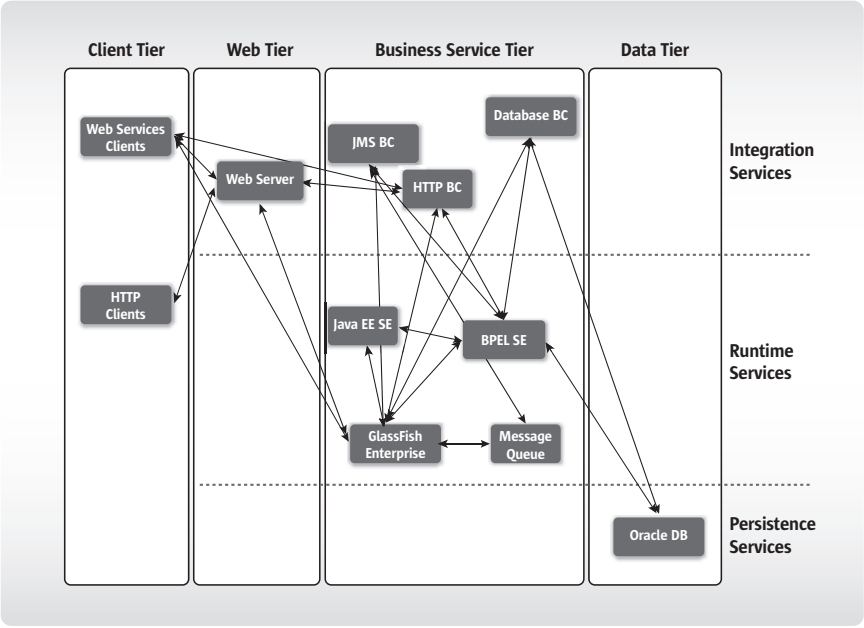


Figure 3.

Service / Component	Service and Role Description
GlassFish Enterprise Server	Provides Java EE 5 runtime environment for GlassFish ESB Suite and serves as a container for JBI components. Provides browser-based and Command Line Interface (CLI) administrative tool to start and stop domains and JBI components, configure the server, and deploy Web applications and Composite Applications (Service Assemblies).
BPEL Service Engine	Enables orchestration of web services using BPEL. Provides JSR 208-compliant JBI runtime component that provides services for executing WS-BPEL 2.0
Java EE Service Engine	JSR 208-compliant JBI runtime component that connects Java EE web services to JBI components.
HTTP Binding Component	Provides connectivity for SOAP over HTTP in JBI 1.0 compliant environment. The HTTP Binding Component is used as both a provider proxy to support connectivity to services in the JBI environment, and as a consumer proxy to invoke services.
Database Binding Component	Provides capabilities for configuring and connecting to databases that support Java Database Connectivity (JDBC) from within the Java Business Integration (JBI) environment.
JMS Binding Component	JMS Binding Component acts as a service provider in an outbound message flow where it converts messages to JMS messages and sends them to a JMS destination. It acts as a proxy consumer for the inbound message flow where it converts the JMS message that it receives from a JMS service to a normalized message, and then sends the normalized message as part of the message exchange to another component as a service request.
Sun Java Message Queue	Provides JMS provider capabilities.
Sun Java System Web Server	Provides web container service for web applications, a Java EE 5 compliant implementation of Java Servlets 2.5, JSP 2.1, JSF 1.2. Provides HTTP Load Balancer service (using integrated Reverse Proxy feature or GlassFish Load Balancer plug-in) for web and web service applications when routing incoming requests to a specific service instance, and, where desired, routing successive requests from the same user to the same service instance.
Oracle DB	Provides Persistent Data Store service for BPEL SE and Data Store service for web application (Loan Application).

Deployment architecture diagram

The application is deployed to a highly available environment as shown in the diagram below, which provides a graphical representation of the deployment architecture for the GlassFish ESB Reference Architecture. It shows the following features for the deployment architecture:

- Representation of the tiered application framework in the Deployment Architecture
- The components that are installed as a part of the Reference Architecture
- The redundancy strategies that are used to achieve scalability and availability

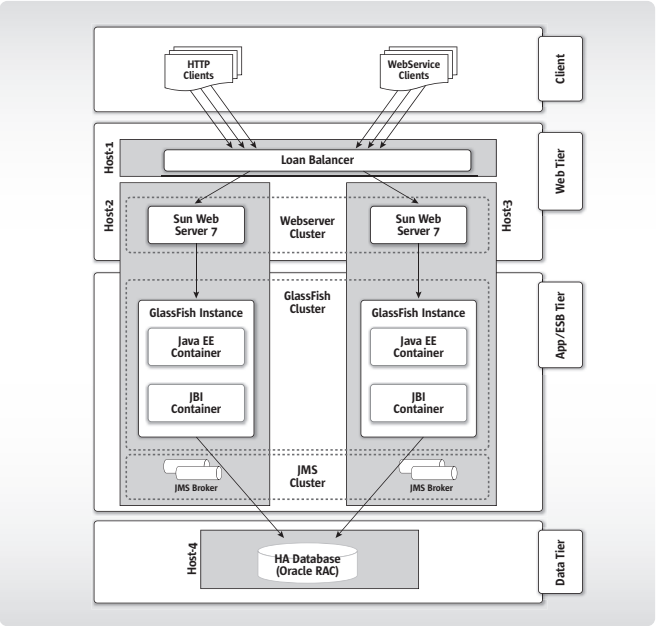


Figure 4.

Client tier

The client tier includes both standalone client programs as well as web browsers. These can be used to access the web application or make web service calls to both initiate or approve a loan application business process.

Web tier

The web tier contains a load-balancer as well as the web servers to which the web application is deployed. The load balancer ensures that requests can be distributed between the two web servers that run the web application servlets. The deployment also uses the latest Sun Java Web Server 7 Update 6 with a built-in Reverse Proxy feature for the web container, which provides clustering and session replication capabilities as well as load balancing. This enables failover between instances and session data replication. Note that Sun Java Web Server does not come bundled with GlassFish ESB but it is available as a free download.

Application and GlassFish ESB tier

This tier contains the GlassFish Enterprise server cluster to which JBI components are installed and the business process project is deployed. The BPEL business process is therefore configured to be persistent for failover between the GlassFish ESB instances. SOAP messages that are accepted and routed for further processing by the HTTP Binding Component (leverages GlassFish Enterprise Server runtime and clustering) are received from the HTTP Load Balancer. Currently the deployment architecture also shows a highly available JMS server. This is a part of a future reference architecture plan; its implementation will not be discussed in this whitepaper.

Data tier

This tier contains the back-end data systems used by the application as well as the BPEL Service Engine for persisting business process instances. BPEL SE uses the JDBC resource created in the GlassFish Enterprise Server to get the database connection required for persistence.

If application data is also stored in the database, then the web application can use the JDBC resource created in the web container on which it is deployed (Sun Java Web Server for our reference architecture).

Oracle DB was chosen for our Deployment Architecture, but was not deployed for high availability as the data-tier availability is not the primary focal point of this paper.

Operating system and virtualization

For demonstrating and testing high availability and clustering with GlassFish ESB, this deployment uses Solaris 10 and the Solaris Zones feature that comes with it. Solaris Zones allows application components to be isolated from one another, even though the zones share a single instance of the operating system. From an application perspective, a zone is a fully functional Solaris OS environment. Multiple zones can be created on a single computer system, each zone serving its own set of applications. Four Solaris Zones are used as virtual hosts for deploying this architecture.

Availability strategies in deployment architecture

The deployment architecture uses the following strategies to meet the availability requirements:

1. **Service availability**

   Service availability means that a service is available, even when the service provider fails. Service availability is generally achieved by using multiple identically configured service instances (redundancy). Redundancy eliminates single points of failure (assuming that simultaneous failure of all instances is extremely unlikely). If one instance providing a service fails, another instance is available to take over. This mechanism is known as service failover. Service failover is supported in the reference architecture discussed here by using load balancing provided by Sun Java Web Server's integrated Reverse Proxy feature.

2. BPEL SE failover support

BPEL SE is a stateful GlassFish ESB service engine which uses a persistence database to track cluster instances and distribute work. When the BPEL SE is installed and configured in a clustered environment and one engine fails, any in-process business process instances are taken over by one of the remaining engines and the process is completed. When the failed engine recovers, it continues to process new requests. Clustering for the BPEL SE leverages the persistence and recovery features of the service engine, so persistence must be enabled for the BPEL SE on each cluster instance.

Failover is also supported for business processes configured for correlation. When correlated messages are processed in a clustered environment, the load balancer or binding component routes the correlating message to any BPEL SE in the cluster. If the BPEL SE to which the message was routed does not own the correlating business process instance, the instance is routed to the engine that received the correlated message (regardless of which engine began processing the initial message). Processing is then completed on the engine that received the correlated message. The appendixes provide more details about BPEL SE Failover and Correlation support.

3. Session state availability

Session state availability ensures that data associated with a user session is not lost during a service failover. The session state data that is stored by the failed instance is made available to the failover instance. This mechanism is known as session failover. Session data is replicated to other instances of service providers, so the session failover is transparent to users. Session failover is supported in the reference architecture using a Sun Java Web Server cluster and its Session Replication feature.

Scalability strategies in deployment architecture

All service components used in this architecture can be independently scaled up, depending on the kind of traffic that GlassFish ESB receives. Each service component in the deployment architecture is, in reality, composed of two or more service instances running on separate computers behind a load balancer. This architecture allows for horizontal scaling of the service components by adding additional service instances. In particular, the deployment architecture ensures scalability for the following components:

GlassFish ESB service

This uses a GlassFish Enterprise Server two-machine cluster with JBI components installed on the cluster targets, where each instance resides on a different physical or virtual machine. If required, additional GlassFish ESB instances can be added to the cluster, which allows for a cost-effective solution and also helps maintain high availability by distributing the load across additional machines.

Web server

This uses a Sun Java Web Server cluster with two instances on different physical or virtual machines and provides the same benefits as clustering GlassFish ESB.

Java Message Queue

Although the deployment architecture discussed in this document does not implement JMS-level clustering, it can be accomplished by adding additional Sun Java Message Queue brokers configured in a Conventional Broker Cluster or Enhanced Broker Cluster setup.

Data store services

Oracle Real Application Cluster (RAC) is widely used for scaling up data store services in enterprise-level service architectures. However, the deployment architecture discussed in this whitepaper uses a single instance because data store scalability is out of scope for this whitepaper.

Other alternatives for the deployment architecture

Some of the components used for the deployment architecture discussed so far can be substituted depending on requirements and design decisions.

Load balancing

This deployment uses the Sun Java Web Server Reverse Proxy feature to allow simple load balancing, mainly because of the relative ease of set up and simple functionality. However, the GlassFish HTTP Load Balancer plug-in feature, which can be installed with Sun Java Web Server or Apache Web Server, is also commonly used in HA solutions involving GlassFish Enterprise Server and GlassFish ESB. Architects can also use other sophisticated hardware load balancers that can work with GlassFish ESB to create highly available solutions.

Web container

Instead of the Sun Java Web Server used for this deployment, GlassFish Enterprise Server can also be used for web container services. GlassFish Enterprise Server provides in-memory replication for session state availability, which makes it suitable for web-tier deployment and a good option to consider if Sun Java Web Server can not be used for some reason. For example, if the solution requires the usage of portlets or providers that use Java EE APIs that are not supported by Sun Java Web Server, such as the Enterprise JavaBeans (EJB) or Java Connector Architecture (JCA) interfaces, then GlassFish Enterprise Server must be used as the web container.

ESB cluster setup

As discussed earlier on, BPEL and IEP Service Engines also support clustering at the component level. This allows for multiple service engine instances to be deployed on a single GlassFish server in a non-clustered environment. More information on this is available at Configuring Components for Standalone High Availability and Failover.

Architects can also consider deploying multiple instances on the same or across multiple physical/virtual machines in a distributed environment. The Guide to GlassFish High Availability whitepaper for GlassFish Enterprise Server deployment contains more details about such deployment options.

Data store

MySQL™ (with its *MySQL Replication and MySQL Cluster features*) provides support for availability and scalability and is suitable for enterprise production deployments, while JavaDB (Derby) database is more suitable for evaluation and development and does not provide support for high availability.

Operating system and virtualization

GlassFish ESB and other components included in this deployment architecture are supported on multiple operating systems like Solaris, Linux, and Windows. The release notes for GlassFish ESB and other components involved in this deployment contain more details about platform-level support.

GlassFish ESB and other components included in the deployment architecture are tested with and support multiple virtualization technologies. Sun also supports and performs testing of its middleware products on select virtualization system and OS platform combinations to validate that the products continue to function on properly sized and configured virtualized environments, same as they do on non-virtualized systems. For more information on this, refer to System Virtualization Support in Sun Java System Products.

Web application

The web application serves two user roles, a loan applicant (Customer group) and a loan approver (Admin group). Any user is required to login to proceed.

After logging in, the loan applicant fills in their relevant details. On a submit, the web application performs a routine sanity check of the date and then calls the relevant web service.

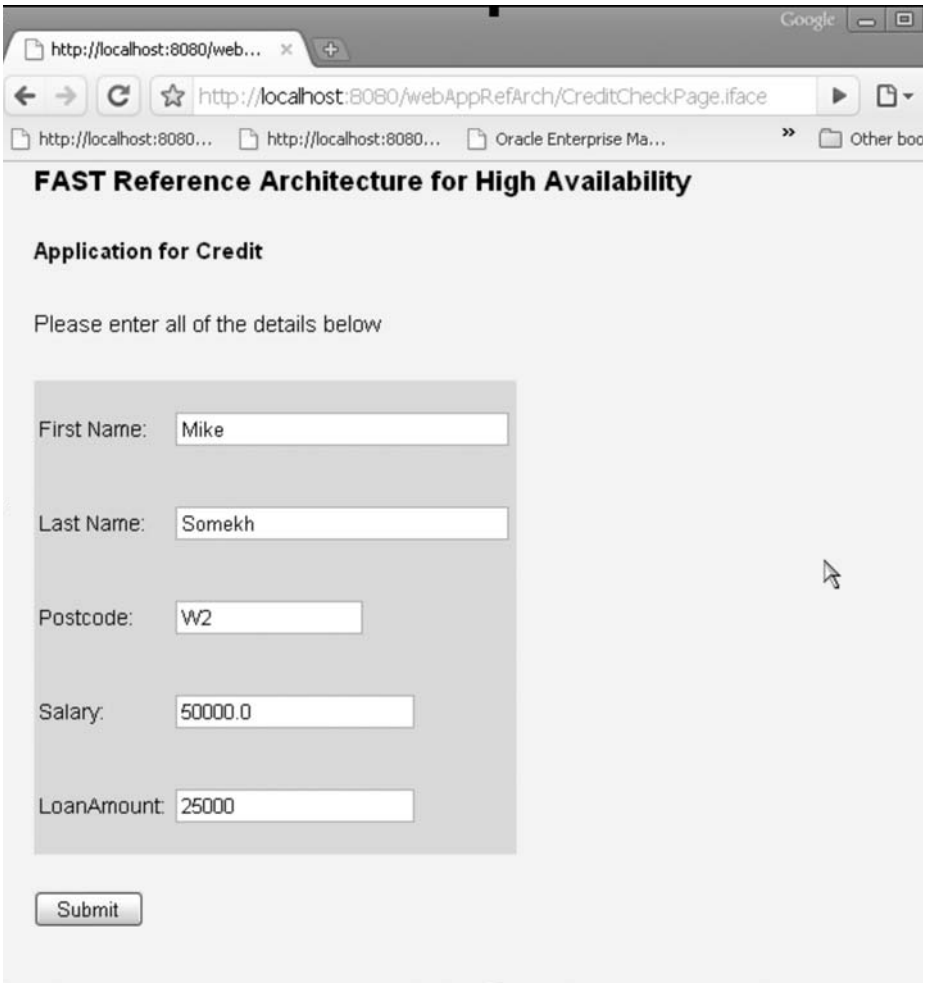
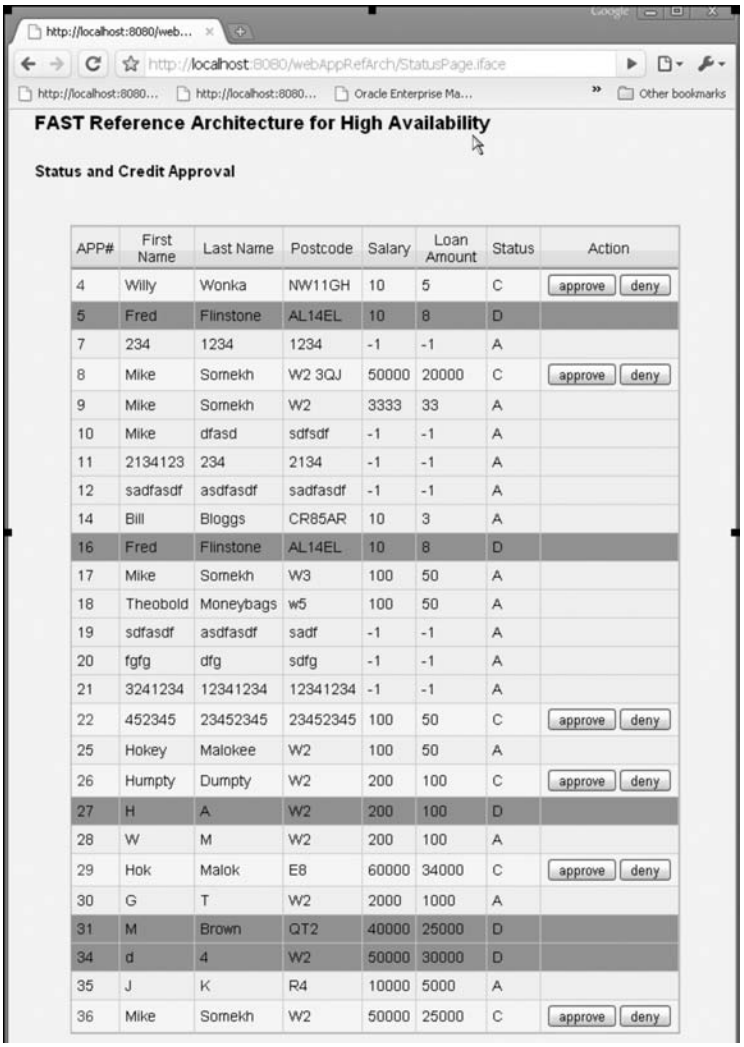


Figure 5.

A loan approver can then log into the web application to review all loan applications. Approving or denying any application calls the web service with the appropriate parameters. The business process that waits for a correlation message with this information then updates the status of the loan accordingly. This process works even if the server on which the business process was originally running goes down.

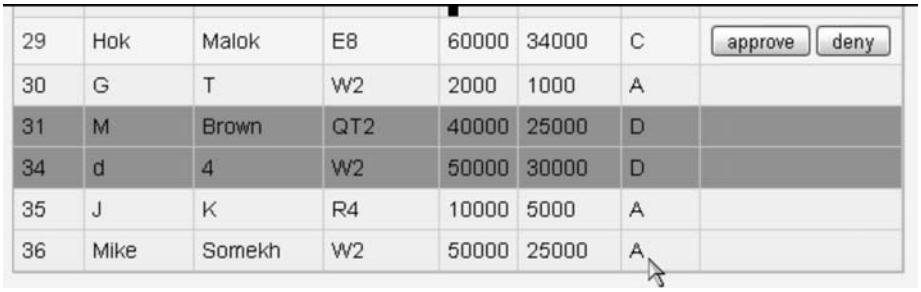


APP#	First Name	Last Name	Postcode	Salary	Loan Amount	Status	Action
4	Willy	Wonka	NW11GH	10	5	C	approve deny
5	Fred	Flinstone	AL14EL	10	8	D	
7	234	1234	1234	-1	-1	A	
8	Mike	Somekh	W2 3QJ	50000	20000	C	approve deny
9	Mike	Somekh	W2	3333	33	A	
10	Mike	dfasd	sdfsdf	-1	-1	A	
11	2134123	234	2134	-1	-1	A	
12	sadfasdf	asdfasdf	sadfasdf	-1	-1	A	
14	Bill	Bloggs	CR85AR	10	3	A	
16	Fred	Flinstone	AL14EL	10	8	D	
17	Mike	Somekh	W3	100	50	A	
18	Theobold	Moneybags	w5	100	50	A	
19	sdfasdf	asdfasdf	sadf	-1	-1	A	
20	fgfg	dfg	sdfg	-1	-1	A	
21	3241234	12341234	12341234	-1	-1	A	
22	452345	23452345	23452345	100	50	C	approve deny
25	Hokey	Malokee	W2	100	50	A	
26	Humpty	Dumpty	W2	200	100	C	approve deny
27	H	A	W2	200	100	D	
28	W	M	W2	200	100	A	
29	Hok	Malok	E8	60000	34000	C	approve deny
30	G	T	W2	2000	1000	A	
31	M	Brown	QT2	40000	25000	D	
34	d	4	W2	50000	30000	D	
35	J	K	R4	10000	5000	A	
36	Mike	Somekh	W2	50000	25000	C	approve deny

Figure 6.

Credit applications approved or denied automatically are highlighted in the web application for quick reference in green and red, respectively. Those applications that require credit checks are highlighted in yellow.

Clicking approve or deny invokes the CreditCheckResponse web service with the appropriate approval or denial information for that application number.



29	Hok	Malok	E8	60000	34000	C	approve deny
30	G	T	W2	2000	1000	A	
31	M	Brown	QT2	40000	25000	D	
34	d	4	W2	50000	30000	D	
35	J	K	R4	10000	5000	A	
36	Mike	Somekh	W2	50000	25000	A	

Figure 7.

Implementation and limitations

The following sections describe some of the implementation patterns used with GlassFish ESB, related issues and limitations, and some possible workarounds.

Implementation patterns

Enabling GlassFish ESB Clustering After Installation

GlassFish ESB installation comes with a GlassFish *developer* profile as default. Before proceeding with the setup for GlassFish ESB clustering, it must be enabled. There are two ways to do this:

- Option 1: In GlassFish Administration Console, click **Add Cluster Support** on the Common Tasks screen. Review the Add Cluster Support page and click **OK**.
- Option 2: Create a default domain with a *cluster* profile with the following command in `<GlassFish ESB install directory>/glassfish` directory:

```
lib/ant/bin/ant -f setup-cluster.xml
```

- The configuration script unpacks the archives and creates a domains subdirectory and a clustering-enabled domain called `domain1`.

GlassFish ESB cluster setup

After clustering is enabled, GlassFish ESB clustering can be set up by performing the following actions:

1. Configuring the GlassFish Cluster: Creation of GlassFish Node Agents, GlassFish cluster instances, and starting up the GlassFish cluster.
2. Adding a Shared Library to the Cluster: This is required by some of the JBI components
3. Adding a JBI Component to the Cluster: Installing JBI components into the GlassFish cluster. BPEL SE and IEP SE components require additional set up.
4. Deploying Service Assembly to the Cluster: In order for any application to run on the cluster, the Service Assembly needs to be deployed to the cluster. Because NetBeans does not support GlassFish clustering, any Service Assemblies to be run on a cluster need to be deployed using the GlassFish Admin Console or from the command line (asadmin tool).

Load balancing of web service calls

There is no difference between handling SOAP over HTTP calls and pure HTTP calls. Additional set up required in addition to using a load balancer is that accessing WSDL files at a load balancer URL requires a schema definition file (.xsd) to be resolved and

accessible over the load balancer too. This is usually deployed in `<compositeAppName> -sun-http-binding` URL). The schema definition file should be accessible via a different URI (in our scenario `/caRefArch-sun-http-binding)` as compared to the WSDL file URI `(/LoanProcessService)`, and it must be configured separately. The following implementation may come handy when using some of the options described earlier for this deployment:

Configuration of HTTP load balancer for web service calls using Sun Java Web Server reverse proxy

The Reverse Proxy feature is integrated in the core Sun Java Web Server product and there is no additional setup required to enable this feature. The following setup configuration was used for the Loan Process Service used in the deployment architecture discussed here:

```
wadm create-reverse-proxy --host=<WS Admin Server host> --port=<WS Admin Server port>
-- config=WebServerCluster --vs WebServerCluster --uri-prefix=/LoanProcessService
--server="http://<glassfish instance 1>:<http bc port> ,http://<glassfish instance
2>:<http bc port>"
```

```
wadm create-reverse-proxy --host=<WS Admin Server host> --port=<WS Admin
Server port> --config=WebServerCluster --vs --uri-prefix=/caRefArch-sun-
http-binding --server="http://<glassfish instance 1>:<http bc port>
,http://<glassfish instance 2>:<http bc port>"

wadm deploy-config --host=<WS Admin Server host> --port=<WS Admin Server
port> WebServerCluster
```

Configuration of HTTP Load balancer for web services calls using GlassFish Load balancer plug-in feature

With additional installation steps, the GlassFish Load Balancer plug-in can be installed with both the Sun Java Web Server or Apache Web Server. Many online resources are available that describe how to do this (for example, http://blogs.sun.com/Prashanth/entry/setting_up_load_balancing_in). After the load balancer plug-in is set up and working, you can set it up to handle web service calls by inserting the following information in the loadbalancer.xml file, which is typically located in the <webserver instance>/config directory:

```
<cluster name="cluster1" policy="round-robin">
<instance disable-timeout-in-minutes="30" enabled="true" listeners="http:
//<glassfish instance 1>:<http bc port>" name="instance-1" weight="100"/>
<instance disable-timeout-in-minutes="30" enabled="true" listeners="http:
//<glassfish instance 2>:<http bc port> name="instance-2" weight="100"/>
<web-module context-root="LoanProcessService" disable-timeout-in-minutes="30"
enabled="true" error-url="sun-http-lberror.html"/>
```

```
<web-module context-root="caRefArch-sun-http-binding" disable-timeout-in-minutes="30"
enabled="true" error-url="sun-http-lberror.html"/>
<health-checker interval-in-seconds="30" timeout-in-seconds="10" url="/"/>
</cluster>
```

If the `loadbalancer.xml` configuration is manually edited, the web server's wadm pull-config command should be used to get the configuration changes into the web server.

```
<ws_install_dir>/bin/wadm pull-config --user=admin --host=<ws_host> --
port=<ws_admin_port> --config=<ws_config> <ws_host>
```

Known issues and limitations

Database BC Issue # 2269: Unable to execute Store Procedure when BPEL persistence is enabled.

With GlassFish ESB v2.1, there is a known issue with the database BC related to using the Store procedures when persistence is enabled for the BPEL service engine (property PersistenceEnabled=true). If this issue is encountered, the following exception is thrown:

```
BPJBI-6007: Failed to process In Out Message M Ex 249894645338302-58932-134717778673060250
java.lang.RuntimeException: BPJBI-7017: Could not find a business process providing
service for the endpoint specified in the message exchange. Service name:
{http://enterprise.netbeans.org/bpel/bpelRefArch/bpelRefArch}plNEXTAPPNUMFINDER,
endpoint name: jdbcPortTypeRole_partnerRole, operation name: execute
        at
com.sun.jbi.engine.bpel.BPELSEInOutThread.getInComingEventModel(BPELSEInOutThread.java:579)
        at
com.sun.jbi.engine.bpel.BPELSEInOutThread.processRequest(BPELSEInOutThread.java:386)
```

Workaround: The issue has been fixed in GlassFish ESB v2.2 . In addition, the only other workaround possible without using the recommended version is to disable BPEL SE persistence or download the latest Database BC installer available in the OpenESB community.

HTTP binding component issue # 6891581: HTTP BC does not accept requests after restart

If the GlassFish cluster instance on which the HTTP BC is installed crashes, GlassFish Domain Administration Server (DAS) detects the unhealthy cluster instance and automatically restarts it. When the instance comes back online, HTTP BC sometimes fails to start and a message is logged in the server.log file on the cluster instance:

```
WEB0701: Error initializing endpoint
java.net.BindException: Address already in use: <HTTP BC Port>
```

After this point, the HTTP BC is not able to accept further requests.

Workaround 1: Manual and temporary solution

User needs to manually stop and then start the GlassFish server instance once again. The start should be initiated only after HTTP BC has fully shut down (No FIN_WAIT_2, CLOSE_WAIT, TIME_WAIT status on HTTP BC port).

Workaround 2: Permanent solution

The port can be prevented from being in the CLOSE_WAIT, TIME_WAIT or FIN_WAIT_2 states for too long. This is configured on the OS level by setting a low value for the TCP_TIME_WAIT_INTERVAL; this is a Solaris 10 OS parameter. The value for GlassFish ESB clustering can be as low as approximately 10 seconds for HTTP BC to recover after a sudden crash of the GlassFish ESB instance, when DAS is used to automatically restart HTTP BC.

To set the TCP_TIME_WAIT_INTERVAL:

Use the get command to determine the current interval and the set command to specify an interval of 10 seconds:

```
ndd -get /dev/tcp tcp_time_wait_interval
ndd -set /dev/tcp tcp_time_wait_interval 10000
```

```
* Default value: The default time wait interval for a Solaris operating system is
240000 milliseconds, which is equal to 4 minutes.
```

In Solaris Zones deployment, this set up needs to be done in the Global Zone from where this set up information is propagated to local zones.

Java EE Service Engine (6892577): Java EE SE EndPoint disabled after stop and not enabled after start

Occasionally, administration activity required for JBI components in a cluster requires stopping some of the JBI components. A known issue with the Java EE SE component fails to enable End Point for the EJB service after it is stopped and restarted. Exception similar to this one can be thrown:

```
BPJBI-6021: Caught exception when invoking two way external webservice
javax.jbi.messaging.MessagingException: JBIMR0044: Unable to locate activated endpoint
for service connection
{http://enterprise.netbeans.org/bpel/bpelRefArch/bpelRefArch}plLoanApprover
LoanApproverPortTypeRole_partnerRole.
```

Workaround: The Administrator must fully shut-down and start the Java EE SE component. The End Point for EJB service initializes properly after restart. This can be done by using the `asadmin` tool:

```
asadmin stop-jbi-component --target cluster1 sun-javaee-engine
asadmin shut-down-jbi-component --target cluster1 sun-javaee-engine
asadmin start-jbi-component --target cluster1 sun-javaee-engine
```

Web Server core file generated after installing the GlassFish Load Balancer plug-in

When performing the installation of the GlassFish Load Balancer plug-in, Sun Java Web Server configuration files are automatically updated. This may result in the web server producing a core file. This is caused by the insertion of duplicate `loadmodules` rows for the J2EE plug-in in the Web Server `magnus.conf` file.

```
Init fn="load-modules" shlib="libj2eeplugin.so" shlib_flags="(global|now)"
##BEGIN EE LB Plugin Parameters
Init fn="load-modules"
shlib="/opt/SUNWwbsvr7/plugins/lbplugin/bin/libpassthrough.so" funcs="init-
passthrough,service-passthrough,name-trans-passthrough" Thread="no"
Init fn="init-passthrough"
##END EE LB Plugin Parameters
Init fn="load-modules" shlib="/opt/SUNWwbsvr7/lib/libj2eeplugin.so"
shlib_flags="(global|now)
```

Workaround: Removing the duplicate entry for libj2eeplugin.so fixes the core. This issue has been fixed in GlassFish v2.1 patch 3.

Sun Java Web Server 7 limited annotation support

Sun Java Web Server only supports annotations to the extent the bundled JWSDP 2.0 supports them. The server does not support JSR 109 for the `@WebServiceRef` element (required for J2EE compliant containers). Example usage in the web application code:

```
@WebServiceRef(wsdlLocation = "WEB-INF/wsdl/client/LoanProcess/LoanProcess.wsdl")
```

As a result, the `@WebServiceRef` element is not used when the web application is deployed to the web server; the web server uses the Java code in the service that is generated during the web application build process. Deployment to Sun Java Web Server may result in the following exception being thrown:

```
java.io.FileNotFoundException: /root/Temp/WebClientRefArch/src/conf/xml-
resources/web-service-references/LoanProcess/wsdl/jsc-zone-
236.czech.sun.com/LoanProcessService/LoanProcessPort.wsdl (No such file or directory)
```

Workaround: Developers can include a valid URL for the service WSDL file using the `wsdlLocation` attribute for the `wsimport` command used during the build process.

NetBeans IDE Support limitations for integration with GlassFish Enterprise Server and Sun Java Web Server

- **NetBeans IDE does not support managing GlassFish cluster instances,** so JBI components in the GlassFish cluster need to installed and managed using GlassFish administration Tools: GlassFish Administration Console and the `asadmin` CLI tool.
- **Web Server 7.0 NetBeans Plug-in Does not Support JSF Web Applications** JSF application deployment must be done via Web Server administration tools: Web Server Administration Console or `wadm` CLI tool.
- **Web Server 7.0 NetBeans plugin does not work with the native package based Sun Java Web Server installation.**

Chapter 7

Appendix

BPEL service engine failover support test

The following information demonstrates BPEL SE cluster support in a GlassFish Enterprise Server cluster with a test scenario that is used to test Business Process failover support.

1. Web service client sends the SOAP requests.

Figure 8.

2. The requests are processed by the BPEL SE.

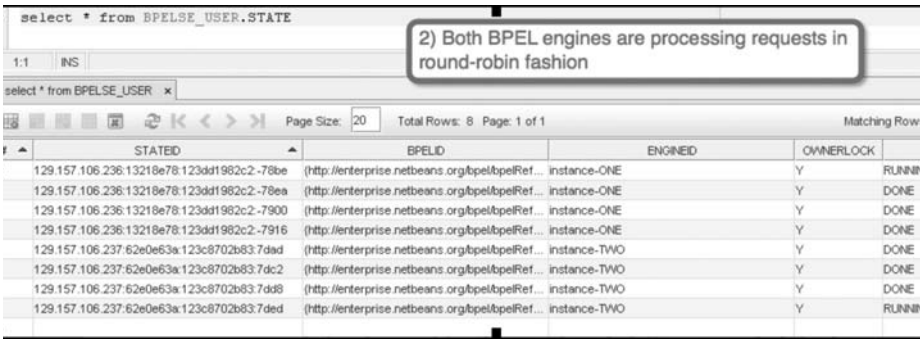


Figure 9.

3. The first engine (instance-ONE) is shutdown using the following command:

```
$GLASSFISH_HOME/bin/asadmin stop-instance instance-ONE
```

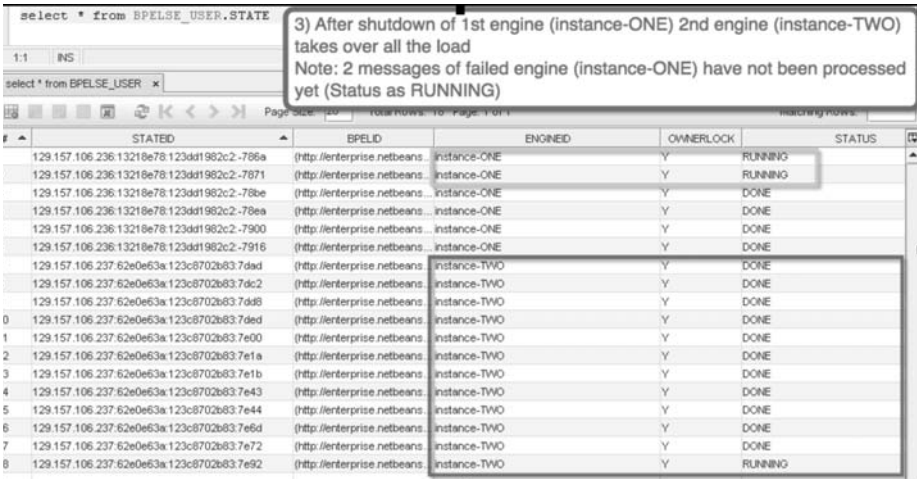


Figure 10.

4. The second instance of the BPEL service engine takes over the processing and all requests are handled.

   This can be verified by checking the STATE table to where the 18 rows/requests are marked DONE. Notice that only 4 requests were handled by the first BPEL SE (instance-ONE) , which indicates that 2 RUNNING requests were failed over to the second live engine (instance-TWO).

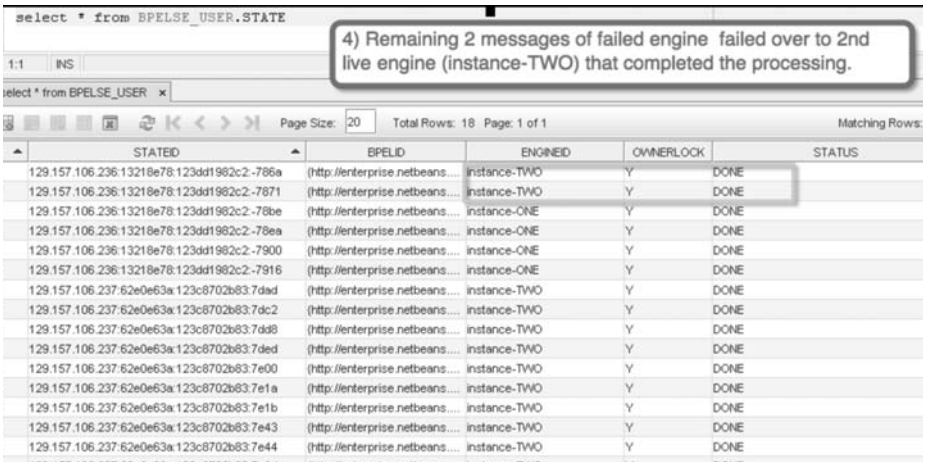


Figure 11.

BPEL SE correlation failover support

This test scenario uses the BPEL SE running on an Glassfish Enterprise Server cluster to demonstrate correlation failover handling. It uses an in-house web application running on a Sun Java Web Server Cluster that calls a business process application deployed on the GlassFish Enterprise Server cluster. This scenario also exercises all the service components of the reference deployment architecture described in this whitepaper.

1. Customer uses the web application to submit a Loan request that requires a manual Credit Check and approval by the Loan Approver.

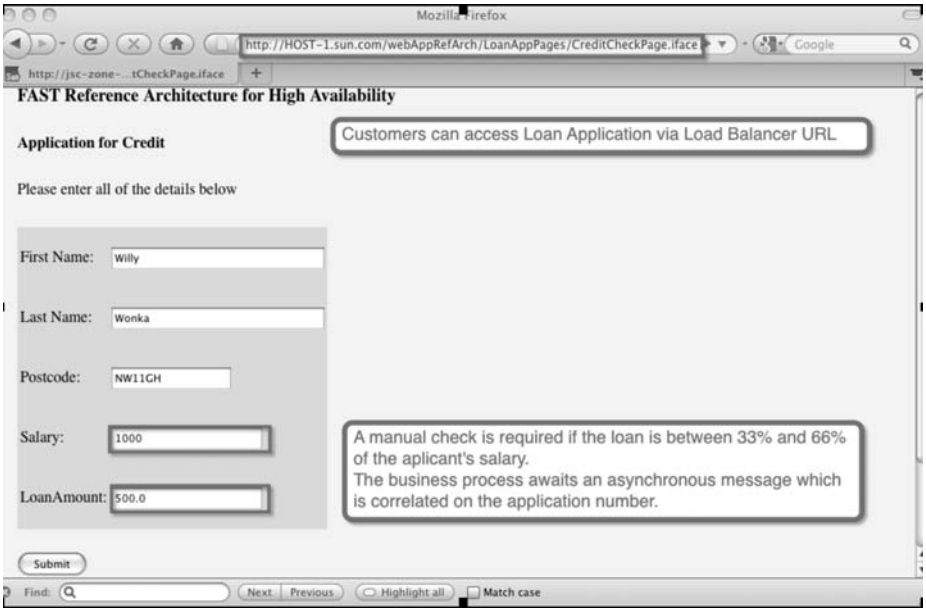


Figure 12.

- Correlation represented in the BPEL Diagram

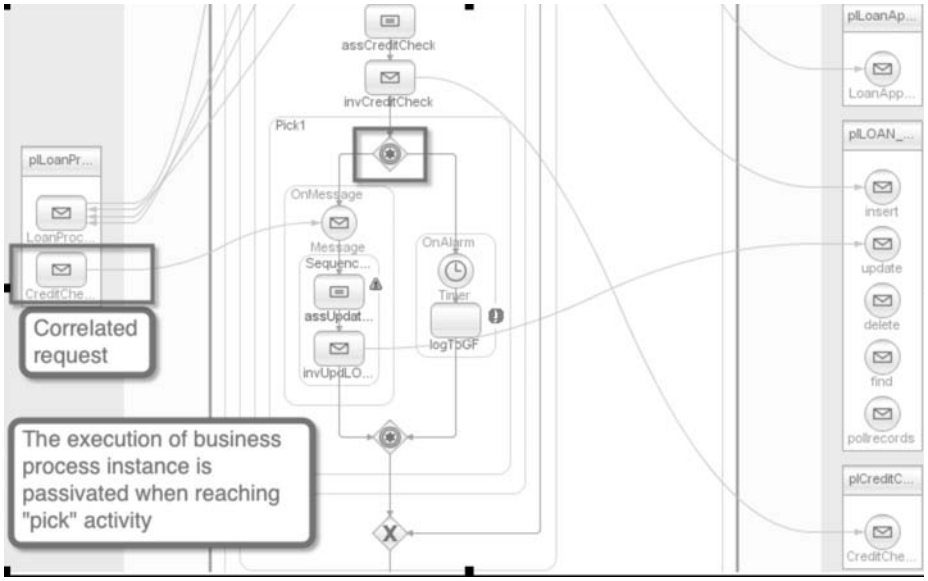


Figure 13.

- Correlation Depicted in the BPEL Persistent Database

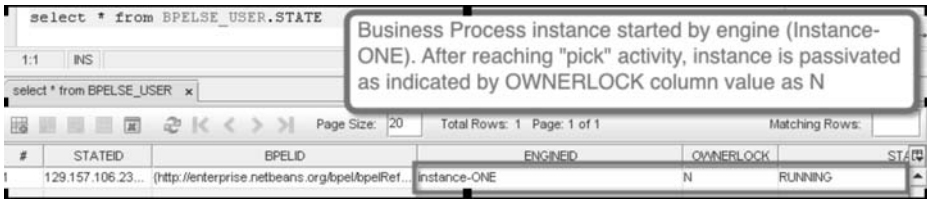


Figure 14.

2. Trigger failover by shutting down the first engine instance (Instance-ONE) with the following command:

```
$GLASSFISH_HOME/bin/asadmin stop-instance instance-ONE
```

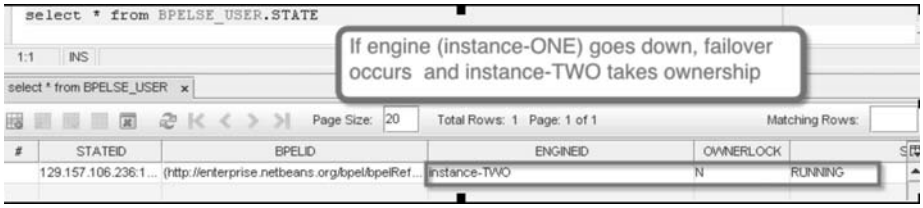


Figure 15.

3. Loan Approver approves (or denies) the loan using the web application.

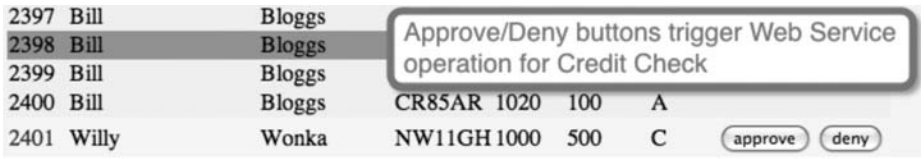


Figure 16.

TWO) and the passive business process instance is re-invoked and taken through to completion by this engine.

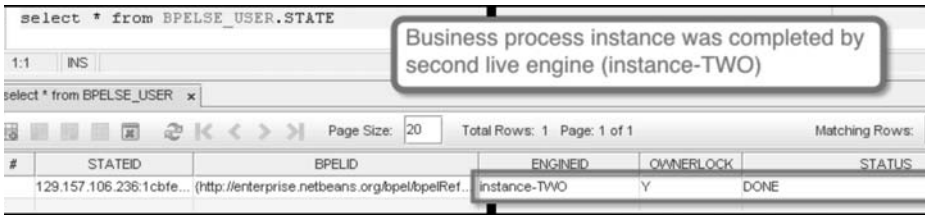


Figure 17.

4. The Loan status indicates approved.

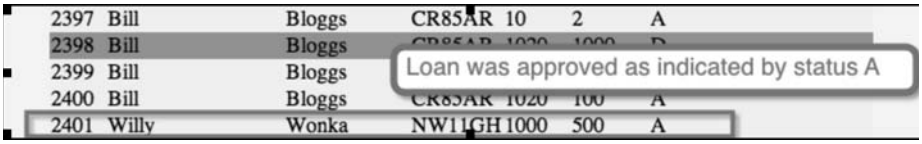


Figure 18.

Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 USA **Phone** 1-650-960-1300 or 1-800-555-9SUN (9786) **Web** sun.com