

Weblogic Clustering

Clustering creates an illusion — it permits the deployment of application components and services to several machines while presenting only a single face to the client. There are good reasons to support this illusion. When a client requests a service, it should make no difference if the service runs on a single server or across a number of servers. The clustering abstraction provides you with a clear route to improving the performance and scalability of your applications, albeit with increased administration of hardware and network resources. WebLogic's clustering offers three important benefits:

Scalability

A solution that allows you to create additional capacity by introducing more servers to the cluster, thereby reducing the load on existing servers.

Load balancing

The ability to distribute requests across all members of the cluster, according to the workload on each server.

High availability

A mix of features that ensure applications and services are available even if a server or machine fails. Clients can continue to work with little or no disruption in a highly available environment. WebLogic achieves high availability using a combination of features: replication, failover, and migratable services.

Clustering Overview

A WebLogic domain can be composed of a number of WebLogic instances, and several of these servers may be grouped into clusters. For instance, you could set up a cluster of WebLogic servers, all of which host your web application and related resources. This cluster of servers could be fronted by a load balancer that distributes requests evenly across all the members of the cluster. The load balancer could itself be another WebLogic instance. All server instances must belong to the same WebLogic domain. Thus, a WebLogic cluster is a group of servers working together with services, such as clustered JNDI, to provide support for failover and load balancing. A domain may in turn have a number of WebLogic instances, several groups of which can be placed into different clusters.

Tiers

Layered architecture is a common design methodology used in software engineering. The software is decomposed logically into a number of layers (tiers), each layer providing a well-defined set of services. J2EE applications can be partitioned in the same way. A typical J2EE application includes multiple web components (servlets, JSPs, and filters), a number of EJBs that implement the business logic, and data access classes or EJBs that interface with the underlying data store. WebLogic's support for clustering enables you to physically partition these services across well-defined application tiers. In this context, a tier represents a number of WebLogic instances that behave in a similar way, each hosting the same set of applications and services. Clearly, it makes sense to map each tier to a different WebLogic cluster. A typical application setup would define at least one of the following tiers:

Web tier

Refers to the bank of servers that provide static content, such as HTML and images, to clients. The web tier is usually the first point of contact for clients, although client requests also may be directed transparently through firewalls and then distributed to these servers via a load balancer. The web tier usually is composed of a number of web servers.

Presentation tier

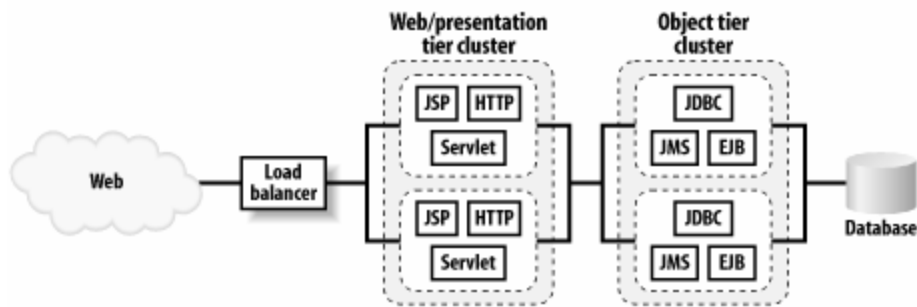
Refers to WebLogic instances that provide dynamic content. In our context, this applies to servers hosting the JSP pages and servlets. The presentation tier often is combined with the web tier, though this isn't necessary.

Object tier

Refers to WebLogic servers that host the RMI and EJB objects, which encapsulate the business logic of your application. If your application is a pure web application not utilizing any EJB or RMI objects, the object tier may not even be needed.

Data tier

Refers to the servers that provide access to the actual data store(s). This includes servers hosting your DBMS, LDAP store, and more.



Load Balancing

Load balancing is about distributing work across multiple servers within a cluster. Ideally you want this work to be distributed evenly, to the capabilities and load on each server within the cluster. WebLogic is able to load-balance requests to members within a cluster when certain conditions are satisfied:

- First, WebLogic requires that (most) components be deployed homogeneously to each server in the cluster. In the multi-tier application setup, the web application is targeted to all servers that belong to the presentation tier cluster. Because of this, incoming requests can be distributed evenly across the different members of the presentation tier. Similarly, the EJB components are available on all members of the object tier cluster. Because of this, incoming EJB calls from the presentation tier (or any external client) can be distributed across the different servers in the object tier.
- When a component is deployed to a cluster, multiple copies of the component will be sitting on the different servers comprising the cluster. WebLogic uses the cluster-wide JNDI tree to record the availability of these replicas in the object tier, whereas a proxy plug-in typically holds the information about the availability of servers hosting the servlets and JSP pages.
- WebLogic also must be aware of the status of the components that have been deployed to the cluster. This information forms the basis on which a load balancer can decide which server instances should be used when a request is made to the cluster. WebLogic uses a variety of methods to determine the health status of servers, including multicast heartbeats. By knowing when a server instance is under heavy load while the others aren't, a load balancer is able to direct new requests to the other members of the cluster. Hardware load balancers often have this capability.

Failover

High availability is another important benefit of a WebLogic cluster. By deploying components and installing services homogeneously across all members of a cluster, you incorporate a certain degree of redundancy into your application setup. This ensures that you can continue servicing users, even if one of the servers in the cluster goes down. WebLogic supports a number of clusterable services and provides high availability by including failover features in many of these services. For nonclusterable services, WebLogic still can support high availability by enabling you (the Administrator) to migrate the service onto another running instance within the cluster.

Working with Clusters

Addressing a cluster

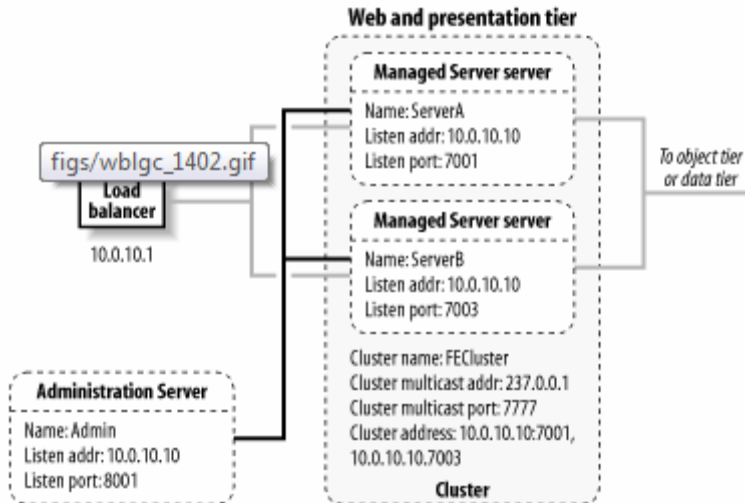
A cluster is composed of a number of individual WebLogic instances. Without clustering, if you needed to address a server, you simply supplied the hostname and port of the server instance. This still will work if the server instance is part of a cluster, but it usually is not what you want. For example, if a servlet needs to make a call to an EJB that is hosted on a separate cluster, the servlet should not need to know which server it should contact. Clearly, a more generalized addressing scheme is required that will allow you to point to a cluster (all servers within the cluster, really), and not just to a single server. We will call this address the cluster address. WebLogic lets you specify the cluster address either as a list of IP addresses, or as a DNS name that is mapped externally to the addresses of all servers that belong to the WebLogic cluster.

You can specify the cluster address using a comma-separated list of hostnames and port numbers. Both of these are valid examples of the cluster address for a WebLogic cluster:

```
10.0.10.10:7001,10.0.10.10:7002,10.0.10.11:7001  
DNS1:7001,DNS2:7001
```

Remember that any address/port combination in the cluster address must be unique. This means that if you need to run a WebLogic cluster on a single machine, you must ensure all server instances participating in the cluster are assigned a unique port number.

Example :



In the above example, we've configured two servers to run on the same machine. Each server is assigned a different listen port, so its cluster address is specified using the following comma-separated list of addresses:

```
10.0.10.10:7001,10.0.10.10:7003
```

The cluster address generally is used when a client needs to access some resource bound in the cluster-wide JNDI tree. In our case, an external client would create the initial JNDI context to this cluster using the following code:

```
InitialContext ctx =
    new InitialContext("t3://10.0.10.10:7001,10.0.10.10:7003");
```

This highlights the drawbacks of specifying the cluster address using a comma-separated list of host addresses — it forces you to hardcode the addresses and port numbers of the servers that belong to the cluster. That is, your code is no longer immune from changes in the configuration of your cluster. For instance, if you add or remove physical hardware or alter the IP addresses assigned to your NICs, those same changes need to be applied to your WebLogic configuration and then to your code source. Clearly, this is not pretty.

For this reason, we recommend that in production environments you configure the cluster address using a DNS name. Your DNS server would then be configured to map the DNS name to all of the servers that belong to the WebLogic cluster. By specifying a DNS name for the cluster address, you establish a naming abstraction that shields your source code and cluster configuration from any changes in the hardware configuration.

There are disadvantages to using DNS names. They do not capture port information, so if the DNS name assigned to the WebLogic cluster is mapped to multiple IP addresses, you must assign the same listen port to all Managed Servers in the cluster. For instance, if we configure a WebLogic cluster with two Managed Servers running on separate machines — say, 10.0.10.10 and 10.0.10.11 on port 7001 — you could modify the DNS server for the participating machines to map a DNS name — say, mycluster — to both of these addresses. You then can set the cluster address for the WebLogic cluster to mycluster:7001. If a client needs to interact with the cluster-wide JNDI tree, it would set up the initial JNDI context as follows:

```
InitialContext ctx = new InitialContext("t3://mycluster:7001");
```

Creating a cluster

In WebLogic 8.1, you need to choose the Custom setup. Remember to indicate that your WebLogic configuration should be distributed across a cluster. In WebLogic 7.0, after choosing the WLS Domain template and a name for the WebLogic domain, select the Admin Server with Clustered Managed Servers option. On the Configure Clustered Servers panel, you then can create an entry for each server instance that will participate in the cluster.

For each Managed Server, you must specify the name of the server, and the listen address and listen port on which the server will be available. Each member of the cluster also uses a particular address and port that it needs to send multicast broadcasts. This address/port combination will apply to all of the servers in the cluster. The Domain Configuration Wizard supplies default values for the multicast address and port: 237.0.0.1:7777. All of these settings are independent of the Administration Server, which needs its own name, listen address, and port number.

If you've already created a WebLogic domain, you can simply use the Administration Console to configure a new WebLogic cluster, or to modify the configuration settings of an existing WebLogic cluster. In order to create a new cluster, you need to first configure one or more Managed Servers that will participate in the new cluster. After this, choose the Clusters node from the left pane of the Administration Console and then select the Configure a New Cluster link. Here you should supply values for various configuration settings for the new WebLogic cluster, such as its name and cluster address. Then select the Configuration/Servers tab to add or remove Managed Servers that should belong to the cluster.

Starting and monitoring the domain

Starting a WebLogic domain that is clustered is no different from starting one that isn't. You first need to start the Administration Server, followed by the Managed Servers that belong to the domain. Each Managed Server notifies the Administration Server when it's up and running, and automatically enlists itself with the cluster. The cluster is alive and

healthy when all of its servers are up and running. You can start the Administration Server by running the startWebLogic command. You then can start each Managed Server by using the startManagedWeblogic command:

```
startManagedWebLogic ServerA http://10.0.10.10:8001/  
startManagedWebLogic ServerB http://10.0.10.10:8001/
```

Here, 10.0.10.10:8001 refers to the listen address and port of the Administration Server, and ServerA refers to the name of a Managed Server. Once a Managed Server completes its boot sequence, you may notice an additional log message at the bottom of the console log, similar to the following:

```
<13-Jan-2003 23:01:31 GMT> <Notice> <Cluster> <000102>  
<Joining cluster MyCluster on 237.0.0.1:7777>
```

This indicates that the Managed Server has located the cluster and has joined it successfully.

You also can use the Administration Console to monitor the status of the cluster. Select the cluster from under the Clusters node in the left pane. Then, if you select the Monitoring tab from the right pane, you can view the number of servers configured for the cluster, and the number of servers currently participating in the cluster. For the example setup, once both servers have completed their boot sequence successfully, you should expect a value of 2 for both settings.

Deploying to a cluster

In a multi-tier application setup in which each tier is mapped physically to a WebLogic cluster, you need to deploy only those components that must live on that tier. So, in our sample web tier cluster, only the web applications should be deployed. If the application is available as a WAR file, you can deploy and target the WAR to the cluster. If the web application is part of an EAR file, only the web applications ought to be deployed to the web tier cluster. Remember to also deploy any shared classes that may be referenced from the servlets and JSP pages within the web application. You can achieve this via the Administration Console itself. Simply choose the Targets tab for a selected web application, and then choose the name of the cluster that will host the web application.

Servlets and JSPs in a Cluster

The major components that typically are deployed to the presentation tier are servlets and JSP pages. You also could deploy JDBC connection pools and data sources to this

cluster, though in our multi-tier application framework, we will make these available to the object tier cluster only. Let's review the load-balancing and failover features WebLogic provides for the servlets and JSPs.

WebLogic can load-balance the requests to servlets and JSP pages deployed to a cluster. It can distribute the requests across all of the servers in the cluster that host the web application. There is one important caveat. This load balancing occurs only for those requests that are not bound to an HTTP session. As soon as a client is involved in an HTTP session on the server side, session-aware requests to servlets and JSPs are directed to that server while it's available. WebLogic provides various session persistence mechanisms that ensure the HTTP session can be re-created on another member of the cluster in case the primary server fails.

If you choose in-memory session replication for persisting the HTTP session state, WebLogic maintains on a secondary server a copy of the session-state information that is held on the primary server. This means that all web requests to the cluster that are involved in a session are directed to the same server instance holding the primary session state. Only when the primary server fails are the requests redirected to the secondary server holding the replicated session-state information. For this reason, we refer to sessions as "sticky." In this case, failover is provided by replicating the session state onto a secondary server within the cluster.

If you want to deploy JSPs and servlets to a cluster and benefit from WebLogic's load-balancing and failover features, you also should enable session-state persistence for the web application. In our case, we chose in-memory session-state replication for handling session-state failover in the presentation tier cluster. To enable in-memory session-state replication, you need to ensure that the weblogic.xml descriptor file for the deployed web application incorporates the following XML fragment:

```
<!-- weblogic.xml entry -->
<session-descriptor>
  <session-param>
    <param-name>PersistentStoreType</param-name>
    <param-value>replicated</param-value>
  </session-param>
  <!-- other session param's -->
</session-descriptor>
```

Configuring a Load Balancer

A frontend cluster cannot work effectively without a load balancer. The load balancer provides a single unified address that clients can use (ignoring firewalls), and it serves as the main entry point into the cluster. The role of the load balancer is twofold. First, it should balance the load across the available members of the cluster while remaining faithful to the sticky sessions. Second, the load balancer should detect and avoid failed servers in the cluster. WebLogic provides a rudimentary software load balancer, the

HttpClusterServlet, which round-robins HTTP requests through all the available servers in the cluster. A hardware solution typically includes additional logic to monitor the load on individual machines and distribute the requests accordingly.

If we use the HttpClusterServlet, the load balancer represents an additional WebLogic instance that hosts the HttpClusterServlet. This server instance is not part of the cluster — it simply forwards requests to the members of the cluster. The servlet maintains a list of WebLogic instances that host the clustered servlets and JSP pages, and forwards HTTP requests to these servers using a round-robin strategy. If a client has created an HTTP session, the HttpClusterServlet forwards the request to the WebLogic instance that holds the primary state, and fails over to a secondary server in case of a failure. In general, it can do this by creating a cookie that holds the locations of the primary and secondary servers specific to the client's session. This cookie is then passed between the browser and the server on subsequent requests to the cluster. The HttpClusterServlet examines the cookie sent by the client on subsequent session-aware requests, and determines the cluster member it should forward the request to.

```
<!-- web.xml entry -->
<init-param>
  <param-name>WebLogicCluster</param-name>
  <param-value>
    10.0.10.10:7001:7002|10.0.10.10:7003:7004
  </param-value>
</init-param>
```

Load-Balancing Schemes

Server-to-Server Routing

Both WebLogic 8.1 and 7.0 support several algorithms for balancing requests to clustered objects: round-robin, weight-based, random, and parameter-based routing. By default, WebLogic uses a round-robin policy for load balancing. The algorithm used to choose between the servers in the object tier cluster depends on two factors:

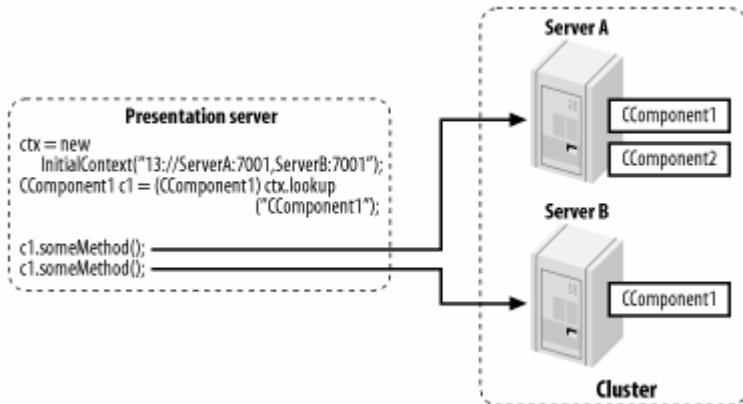
- If the RMI object was compiled with a particular load-balancing scheme, that scheme will be used.
- If no scheme was explicitly configured, the default load-balancing scheme for the cluster will be used.

You can configure the default load-balancing scheme for the cluster by selecting the cluster in the Administration Console and picking a suitable load balancing scheme for the Default Load Algorithm setting in the Configuration/General tab. Let's look at these schemes in more detail.

Round-robin

When the round-robin algorithm is used, requests from a clustered stub are cycled through a list of WebLogic instances hosting the object. A specific order is chosen, and then each server is used in this order until the end of the list is reached, after which the cycle is repeated. A round-robin scheme is simple and predictable. However, this strategy does not react according to the varying loads on the servers. For example, if one server in the cluster is under heavy load, it still will continue to participate in the round-robin scheme like the other members in the cluster, so work may pile up on this server.

Figure below illustrates the round-robin scheme. A server in the presentation tier cluster makes a number of calls on some clusterable component c1 deployed to another cluster. Each method call is directed to one of the servers hosting the component, according to the load-balancing scheme that applies in the situation. Under the round-robin scheme, the method calls simply will alternate between the two servers, ServerA and ServerB.



Weight-based

When using the weight-based load-balancing scheme, the replica-aware stub distributes the requests based on a pre-assigned numeric weight for each server. You can choose a number between 1 and 100 for the Cluster Weight setting for each clustered server under the Configuration/Cluster tab in the right pane. The cluster weight determines what portion of the load a server will bear, relative to other members in the cluster. So, a server with weight 25 will take half as much load as the rest of the servers whose weights are 50. If all members of the cluster are assigned the same weights, they all bear an equal share of the load.

This load-balancing scheme is best suited for clusters with heterogeneous deployments, in which different EJBs are deployed to different sets of servers, or when the processing power of the machines in the cluster varies. You should consider several factors before assigning a weight for the server:

- The number of CPUs dedicated to the particular server
- The speed of the network cards that are used by the machine hosting the clustered server
- The number of non-clustered (pinned) objects or services running on a server

Remember, cluster weights provide only an indication of the "expected" load on a server. A weight-based scheme does not react and respond to the current loads on the cluster servers.

Random

The random load-balancing scheme distributes requests randomly across all members of the cluster. This scheme is recommended for homogenous clusters, in which components are deployed uniformly to all members of the cluster and the servers run on machines with similar configurations. A random load-balancing strategy can distribute loads evenly to all members of the cluster. The longer a WebLogic cluster

remains alive, the closer the distribution is to the "mean." However, each request must incur a slight processing cost of generating a random number. In addition, a random distribution does not account for the differences in the configuration of the machines that are participating in the cluster, and so does not react to different loads on the cluster servers.

Parameter-based

Parameter-based routing lets you programmatically determine which server should be chosen to handle a method call on a clusterable RMI object. Unlike the other load-balancing schemes, this is not a general scheme that can be applied to any clusterable component. Rather, it is needed only when you want extreme control in routing RMI objects.

Client-Server Routing

Both WebLogic 8.1 and 7.0 support the round-robin, weight-based, and random load-balancing schemes for external client applications that make connections into a cluster. External clients are at a disadvantage because the client eventually makes IP connections to each server in the cluster, as all of these schemes distribute the load across all available servers in the cluster.

WebLogic 8.1 can limit this promiscuous connection behavior for clients. A load-balancing scheme with server affinity attempts to always use connections to servers that are already established, instead of creating new ones. The three load-balancing schemes each have an affinity-based counterpart: round-robin affinity, weight-based affinity, and random affinity. If you set the default load algorithm for a cluster to round-robin affinity, for example, the round-robin scheme will still be used for load balancing server-to-server requests. However, server affinity will cause external clients to simply use servers to which they are already connected. This minimizes the number of IP sockets opened between clients and the clustered servers, but at the cost of eliminating load balancing.

Server affinity for JNDI contexts

If a client or server creates a new context using a cluster address, by default the contexts are distributed on a round-robin basis among the available servers determined by the address. WebLogic lets you disable this round-robinning by inducing the client to create the JNDI context on a server to which it is already connected. In other words, you can enable server affinity for client requests for a JNDI context. You need to supply the `ENABLE_SERVER_AFFINITY` property when creating the initial JNDI context:

```
Hashtable h = new Hashtable();  
h.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFactory");  
h.put(Context.PROVIDER_URL, "t3://server1:7001,server2:7001");
```

```
h.put(weblogic.jndi.WLContext.ENABLE_SERVER_AFFINITY, "true");
Context ctx = new InitialContext(h);
```

Using J2EE Services on the Object Tier

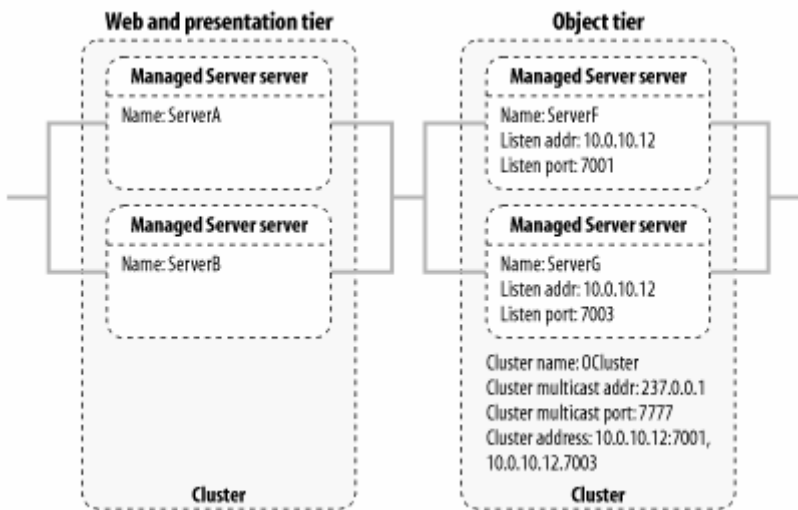
The object tier usually maps to a WebLogic cluster that houses the heavier J2EE services such as EJB and RMI objects, as well as the JMS resources. Because the EJB objects typically need JDBC access to the backend DBMS (data tier), the object tier in our multi-tier application setup also hosts the JDBC resources. Remember, a clustered architecture need not have an object tier. If your application doesn't use EJBs to encapsulate the business logic or doesn't require any asynchronous message handling, you easily can bypass the object tier. In that case, you could deploy the JDBC resources to the presentation tier directly.

Even if your application does make use of EJB objects and/or JMS resources, you still need not construct a separate object tier cluster. You could deploy your entire application (servlets, JSPs, EJBs, JMS destinations, etc.) to a single WebLogic cluster.

Interacting with the Object Tier

Establishing the object tier cluster is no different from setting up the web/presentation tier. The object tier cluster, named `oc1uster` here, is composed of two Managed Servers, `serverF` and `serverG`. Both servers coexist on another machine whose IP

address is 10.0.10.12, and listen on ports 7001 and 7003 respectively. The cluster address for `oCluster` is `10.0.10.12:7001,10.0.10.12:7003`.



Objects that live in the presentation tier will need to reach resources deployed on the object tier. For instance, a servlet may need to access EJB objects or JDBC resources deployed to the object tier. Moreover, requests to these resources should be load-balanced across those various members of the object tier cluster. Clustered JNDI and replica-aware RMI stubs provide us with these capabilities.

Suppose you were to deploy a data source to OCluster. If a servlet in the presentation tier needs to use the data source, it should establish an initial context to OCluster, not an individual server within the cluster. In this case, the servlet would set up the JNDI context as follows:

```
// code running on the presentation tier, which establishes
// a context to the object tier using the object tier's cluster address
InitialContext ctx = new InitialContext("t3://10.0.10.12:7001,10.0.10.12:7003");
```

Remember that although we've specified the cluster address, the context eventually will be associated with a particular server in the list, round-robinning between the servers each time we set up the initial JNDI context. By specifying the addresses of all the servers in the cluster, we allow for the Context object itself to fail over. Even though the JNDI context is initially bound to a particular server, because it is aware of the addresses of other servers in the list, it automatically can fail over to the next available server if the original server becomes unavailable.

Since we established a context on a member of the object tier, we now can look up resources bound to the cluster-wide JNDI tree, such as our JDBC data source:

```
DataSource ds = (DataSource) ctx.lookup("MyDomain.DS")
```

Clustering EJB and RMI

Load balancing and failover

Load balancing for EJBs occurs at both the EJB home and EJB object level. When a client looks up the JNDI tree for an EJB home object, it acquires a cluster-aware stub that can locate home objects on each server to which the EJB component was deployed. Likewise, the EJB object itself is represented by a replica-aware stub that automatically can fail over between method calls. Failover during method calls occurs only if the EJB methods are marked as being idempotent.

WebLogic 8.1 and 7.0 support several load-balancing schemes for EJB and RMI objects: round-robin, weight-based, random, and parameter-based routing. The round-robin algorithm is the default load-balancing strategy. WebLogic 8.1 supports three additional schemes: round-robin affinity, weight-based affinity, and random affinity.

EJB load balancing

WebLogic lets you adjust the default load-balancing strategy for a cluster. This default strategy applies to all clusterable services running on the cluster. All clusterable services use a round-robin algorithm as their default load-balancing strategy. The default strategy can be overridden on a per-service basis. For instance, when you compile a clusterable RMI object, you can explicitly select a load-balancing strategy for the generated RMI stub.

Every EJB component type supports load balancing at the home level. In order to enable cluster-aware EJB home stubs, specify `true` for the `home-is-clusterable` element in the EJB's `weblogic-ejb-jar.xml` descriptor file. The load-balancing scheme for the EJB home object will default to the load-balancing strategy you've configured for the cluster to which the EJB component is subsequently deployed. You also can change the load-balancing scheme for a particular EJB by specifying a value for the `home-load-algorithm` element in the `weblogic-ejb-jar.xml` descriptor file. Its value can be set to either `RoundRobin`, `WeightBased`, `Random`, `RoundRobinAffinity`, `WeightBasedAffinity`, or `RandomAffinity`. In addition, you can use the `home-call-router-class-name` element to specify a custom call router class for the home stub of a stateful session EJB or an entity EJB.

Here is a breakdown of other features particular to the different EJB types:

Stateless session beans

Stateless session EJBs support replica-aware EJB objects. To enable replica-aware EJB objects, specify `true` for the `stateless-bean-is-clusterable` element in the `weblogic-ejb-jar.xml` descriptor. To set the load-balancing strategy at the EJB

object level, use the `stateless-bean-load-algorithm` element in the `weblogic-ejb-jar.xml` descriptor file. You also may use the `stateless-bean-call-router-class-name` element to specify a custom call router class for a stateless session EJB.

Stateful session beans

Because of the sticky nature of stateful session EJBs, method calls to a stateful session bean always are routed to the same EJB instance.

Entity beans

For read-only entity EJBs, WebLogic supports load balancing and failover on every method call, while read-write entity EJBs are pinned to a particular member of the cluster.

Failover for EJB and RMI objects

If an RMI stub makes a call to a service on one of the servers hosting the object and the call fails, the stub will detect the failure and retry the call on a different server.

As explained earlier, automatic failover occurs only if WebLogic knows that the method call is idempotent. If a method is not marked as idempotent, WebLogic cannot be sure that by retrying the method on a different server it won't duplicate any changes made during the previous call. To avoid this potential mistake, WebLogic errs on the safe side and refuses to automatically failover on non-idempotent EJB methods. By default, all methods of stateless session EJB home objects and read-only entity beans are marked as idempotent.

WebLogic still can failover on nonidempotent methods, but only in two exceptional cases. If a `ConnectException` or a `MarshalException` is thrown when a stub attempts to reach the object on the server side, the stub will fail over to a different server. Both of these exceptions can be thrown only before the EJB method begins its execution, and therefore no changes could have been initiated.

Failover manifests in different ways when it comes to EJB and RMI objects. Depending on the EJB type, an EJB may have replica-aware `EJBHome` and `EJBObject` stubs. As a result, failover and load balancing can occur when a client looks up the EJB's home object, or when it invokes an EJB method using its `EJBObject` stub. The varying types of failover are here:

Stateless session EJBs

As stateless session EJBs do not maintain any server-side state, the EJBObject stub returned by the EJB home object can route a method call to any server hosting the object. Failover occurs only on idempotent methods.

Stateful session EJBs

The EJBObject stub for a clustered stateful session EJB maintains the locations of the servers that hold the EJB's primary and secondary states. The EJB instance exists only on the server hosting the primary state, while its state may be replicated to a secondary server. Calls to the EJB object are routed to the server hosting the primary state. If the primary server fails for some reason, subsequent calls are then routed to the server hosting the secondary state. In this case, the EJB instance is re-created on the secondary server using the replicated session state, and a new server is chosen as the secondary server. Changes to the stateful session EJB instance are replicated either when a transaction commits, or after each method invocation if the client hasn't initiated a transaction.

Entity EJBs

For read-only entity beans, the EJBObject stub load-balances on every method call, and supports failover for idempotent methods. WebLogic avoids database reads by caching read-only beans on every server to which they've been deployed. For read-write entity beans, when an EJB home object finds or creates such an EJB instance, it obtains an instance from the same server and returns an EJBObject stub pinned to that server. Hence, for read-write entity beans, WebLogic supports load balancing and failover only at the home level, and not at the method call level (the EJBObject level).

Using JDBC Resources in a Cluster

WebLogic provides failover and high-availability features through JDBC multipools and cluster-aware data sources. Note that a JDBC connection established with the backend DBMS relies on state tied to the physical connection between the JDBC driver and the DBMS. This implies that WebLogic cannot offer failover for JDBC connections. If a client has acquired a JDBC connection and the DBMS to which it is attached or the server from where the connection was obtained fails, the connection is terminated and the client no longer is able to use that connection object.

WebLogic provides high-availability and load-balancing features at the connection pool level through the use of a multipool. A multipool is simply a pool of connection pools, each connection pool potentially drawing its connections from a different DBMS instance. A multipool may be used if the DBMS supports multiple replicated, synchronized database instances. If the multipool is configured for failover, then a connection always is drawn from the first connection pool until failure, after which the

connection is drawn from the next pool in the list. If the multipool is configured for load balancing, requests for JDBC connections are distributed through the connection pools in a round-robin fashion.

Finally, JDBC DataSource objects deployed to a WebLogic cluster are replica-aware. In order to use connection pools and data sources in a clustered environment so that you can take advantage of these features, ensure that you deploy both the connection pool and the data source to the cluster, and not to individual servers within the cluster.