Clustering Awareness of Java Applications in a SAP NetWeaver Cluster

SAP Platform Ecosystem

Copyright

Icons in Body Text

Icon	Meaning
	Caution
	Example
	Note
	Recommendation
	Syntax

Additional icons are used in SAP Library documentation to help you identify different types of information at a glance. For more information, see *Help on Help* → *General Information Classes and Information Classes for Business Information Warehouse* on the first page of any version of *SAP Library*.

Typographic Conventions

Type Style	Description
Example text	Words or characters quoted from the screen. These include field names, screen titles, pushbuttons labels, menu names, menu paths, and menu options. Cross-references to other documentation.
Example text	Emphasized words or phrases in body text, graphic titles, and table titles.
EXAMPLE TEXT	Technical names of system objects. These include report names, program names, transaction codes, table names, and key concepts of a programming language when they are surrounded by body text, for example, SELECT and INCLUDE.
`Example text`	Output on the screen. This includes file and directory names and their paths, messages, names of variables and parameters, source text, and names of installation, upgrade and database tools.
`Example text`	Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation.
`<Example text>`	Variable user entry. Angle brackets indicate that you replace these words and characters with appropriate entries to make entries in the system.
`EXAMPLE TEXT`	Keys on the keyboard, for example, F2 or ENTER.

Table of Contents

Brief Introduction to SAP Web AS Cluster Architecture

Overview

Unfortunately, the behaviour of a J2EE application server cluster was not fixed in the J2EE specification. Therefore, the implementation of cluster behaviour is completely up to the vendor of the J2EE application server.

Performance, failure safety, and scalability of the J2EE applications running on the cluster depend for the most part on the quality of the implementation by the application server vendor as well.

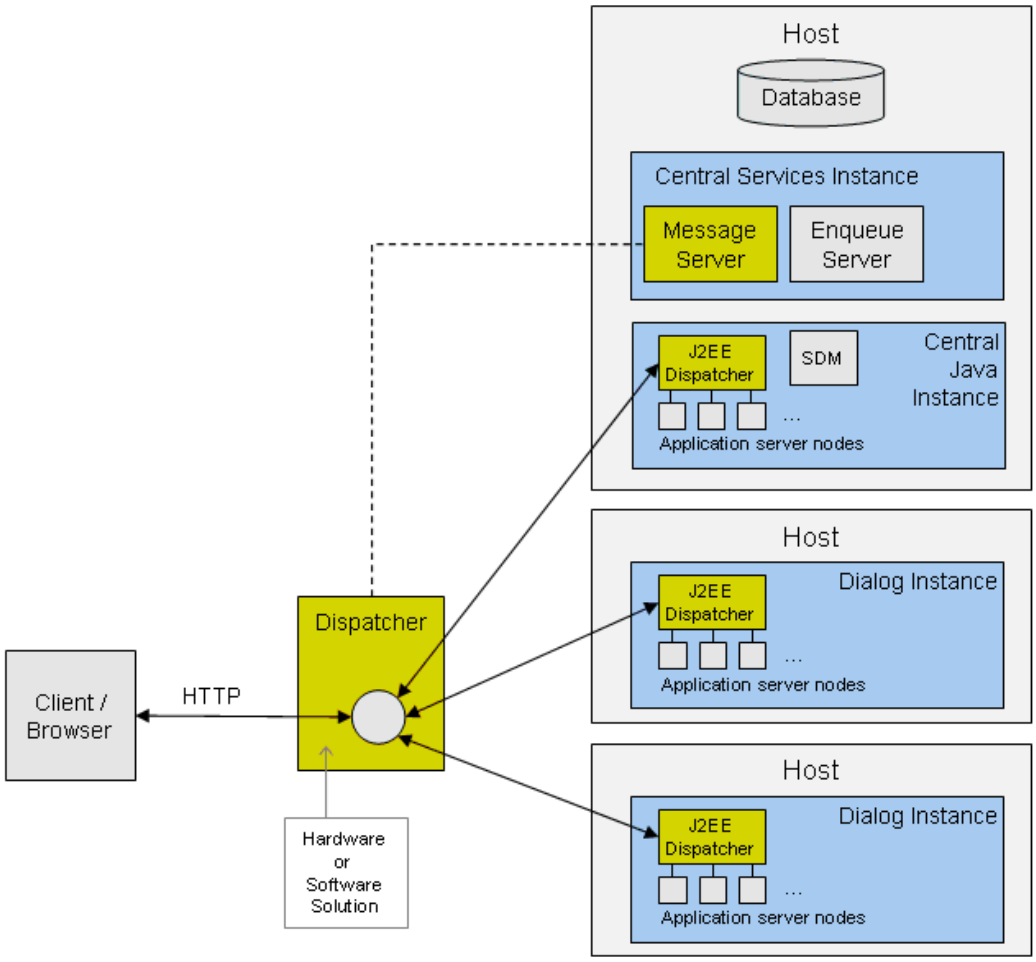
SAP Web Application Server Java (in the following shortened to SAP Web AS) has a mature design, which allows the operation of numerous clustered application server instances and to efficiently distribute the load of applications across them. In addition to this, SAP Web AS offers sufficient scalability and administrational functions.

The following features of SAP Web AS guarantee the performance, failure safety, scalability, and also simple handling in big cluster environments:

- SAP Web AS cluster achieves very low downtimes and considerably reduced maintenance times due to the functions of the SAP Cluster infrastructure.
- The administration of a dynamic cluster is very simple because of the support of self-organizing functions as part of the cluster infrastructure. All essential functions, such as adding or removing cluster nodes, are automatically handled internally by the cluster infrastructure. Therefore the consistency and distribution of deployed applications is guaranteed to all cluster components.
- The minimization of the number of internal connections with simultaneous avoidance of communication bottlenecks is guaranteed by the cluster architecture. (**`Star-cluster-architecture`** with peer-to-peer support).
- The central database that stores all the binary and configuration data guarantees the consistency of the deployed applications.
- SAP Web AS has a homogeneous cluster infrastructure that requires only low administration. In addition to this, the cluster is enabled for remote administration and offers a **`StartUp- and Control-framework`**, for a very comfortable central lifecycle management.
- Debugging in the SAP Web AS cluster is conceived in a way that productive applications are fully debugging-enabled (remote debugging as well) without disturbing active user sessions on the productive application.

Components of the SAP Web AS Cluster Environment

The SAP Web AS cluster consists of a global **`Dispatcher`**, a **`System Database`**, a **`Central Instance`** where central services like the **`Message Server`**, the **`Enqueue Server,`** and the **`Software Deployment Manager (SDM)`** are located, as well as one or several **`Dialog Instances`** with two types of nodes, the **`J2EE Dispatcher`** and the **`Application Server`**.



Dispatcher

This abstract component depicts the central HTTP load dispatching station and the central entry point for all incoming client HTTP requests.

As definite components, several solutions are possible:

- *Network Address Translater (NAT)*
- *Reverse Proxy*
- *Physical Web Switch*
- *Software Web Switch*

All these solutions have advantages and disadvantages that are justified primarily in the configuration possibilities of the components. SAP recommends the use of its own software web switch called "SAP `Web Dispatcher`."

*Please do not confuse the **`SAP Web Dispatcher`** with the **`J2EE Dispatcher`** of the SAP Web AS discussed in the following section. Please have a look at the illustration above to identify the components and their location.*

J2EE Dispatcher and Application Server Node

The **J2EE Dispatcher** is a local entry point for all incoming client requests (not only HTTP). These requests are analyzed by the **J2EE Dispatcher**, locally distributed to the application server nodes and with responses returned to the client. The application server node:

- processes the client requests,
- provides the J2EE container functions, and
- establishes connections to external systems.

The communication between the **J2EE Dispatcher** and the J2EE server nodes, concerning the client request/response, always works in a direct way, i.e. it works with peer-to-peer communication. **J2EE Dispatcher** and the J2EE server nodes are Java VM processes.

System Database

The **System Database** is the central storage for all kinds of binary and configuration data. This database enables:

- the central administration of the SAP Web AS cluster,
- the central deployment of applications, and
- the central installation and execution of upgrades.

Software Deployment Manager (SDM)

The **SDM** is a central component that consists of a server part and a client part. The server part is responsible for the consistency of the deployed content. The client part is responsible for the deployment of software development archives **(sda)** and is responsible for the execution of system upgrades as well.

Locking Server (Enqueue Server)

The **Enqueue Server** is used for the synchronization of simultaneous access to common resources (logical locking) in a SAP Web AS cluster. This is valid across host boundaries. All J2EE Dispatcher and J2EE server nodes in a cluster are connected to the **Enqueue Server.**

Possible reasons for setting locks in a cluster are:

- Starting/stopping of a server or dispatcher node;
- Working with configuration data in the central database;
- Working with the EJB CMP container.

Message Server

The **Message Server** is the central component for information exchange in the cluster. The complete communication between the cluster nodes goes through the **Message Server**. Information about the cluster state and state information of all essential components are deposited on the Message server. It is used by all SAP load-balancing mechanisms to detect the available J2EE dispatchers.

Cluster Lifecycle Management

SAP Web AS has a framework for its lifecycle management. This framework is ready to integrate SAP Web AS instances into the already existing SAP Instance Management Solution. This lifecycle management framework is based on the **StartUp and Control framework**, which guarantees a very comfortable central lifecycle management. One possible Microsoft Windows-based GUI for administrating SAP Web AS cluster nodes is the Microsoft Management Console.

Load-balancing

HTTP load-balancing

As visualized in the previous overview figure, there are two load-balancing types: **`Cluster load-balancing`** and **`Instance load-balancing`**

The cluster load-balancing is recommended by SAP to be done by the SAP Web Dispatcher. The SAP Web Dispatcher receives its information about the available Java instances (particularly the J2EE dispatcher) in the cluster from the Message Server.

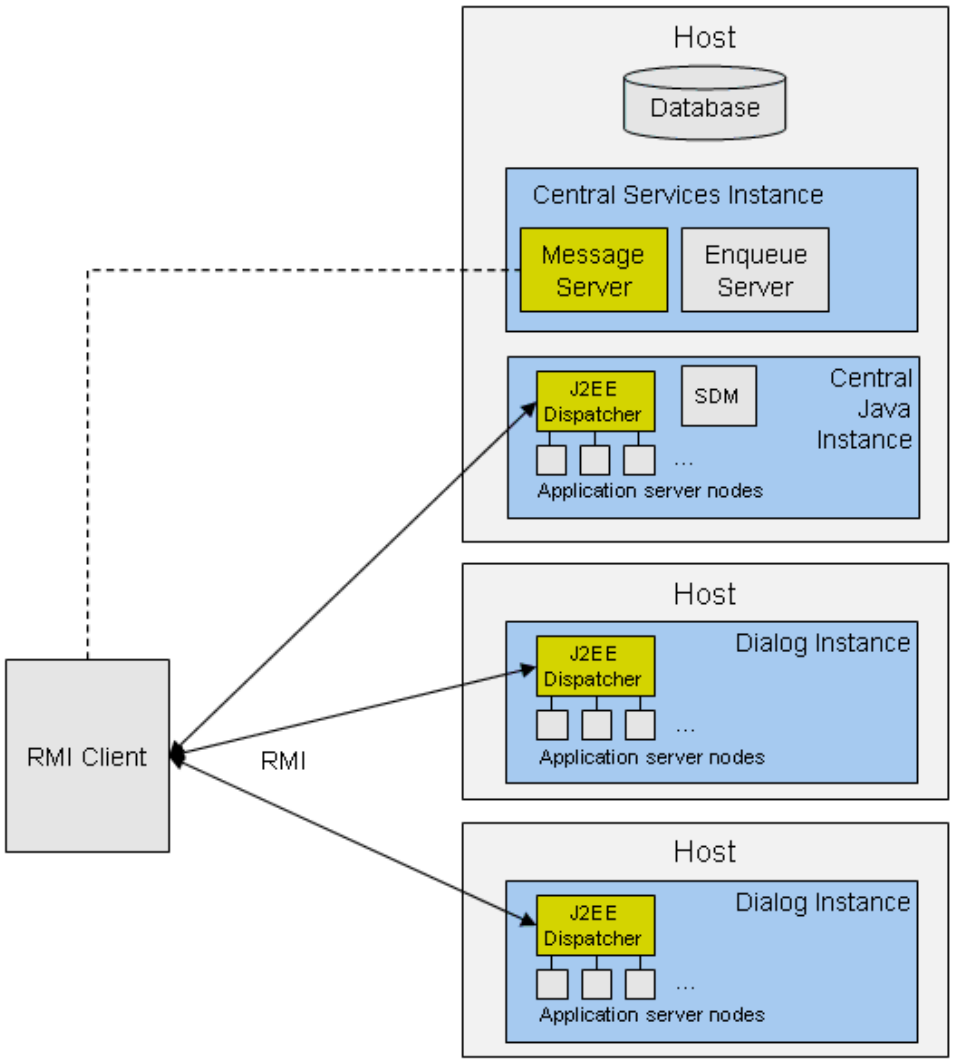
There are two load-balancing strategies available:

- Weighted round robin
- Instance load-balancing

RMI load-balancing

The RMI load-balancing works similar concerning the cluster information retrieval but the actual dispatching process is implemented on client-side, though transparently.

The client obtains an instance of javax.naming.InitialContext from the Message Server. The InitialContext represents an RMI-P4 connection to a dedicated J2EE Dispatcher in the cluster which has been selected by the message server using a weighted round robin load-balancing algorithm. The dispatcher then load balances the request to a server process within the same instance using the round robin algorithm.



The code for attaching to this load-balancing mechanism looks as follows:

```
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sap.engine.services.jndi.InitialContextFactoryImpl");
p.put(Context.PROVIDER_URL, "sapms://<msgserver_host>:<msgserver_HTTP_port>");
p.put(Context.SECURITY_PRINCIPAL, "<username>");
p.put(Context.SECURITY_CREDENTIALS, "<password>");
ctx = new InitialContext(p);
```

You can obtain the Message Server's HTTP port from the instance profile.

In order to handle situations when a server node crashes while executing an operation on the remote object, clients must modify their code as a **`com.sap.engine.services.rmi_p4.P4ConnectionException`** will be thrown. In this case the state of your remote object is lost so you will have to create a new instance to restart the process.

To add state failover capabilities to your RMI/P4 application, SAP recommends using EJBs for the implementation of the remote objects (see following chapter).

Failover

Overview

SAP Web AS provides a high degree of fault tolerance for J2EE Web applications to provide high and continuous availability to the applications running on it. At the end of each request-response cycle, the **HTTP Session** is replicated to a persistent storage. If the system crashes, the storage is used to retrieve all lost session states. This is done by standard Java serialization.

In case of a server failure, the failover system performs a failover that is transparent for the user, which provides a stable and secure environment for the user applications. The J2EE Engine Session Failover Service provides the functions of the failover system and provides **HTTP failover** and **stateful session failover**.

- **HTTP failover** can be enabled for applications that include Web components.
- Stateful session failover can be enabled for specific stateful session beans only if the beans are not involved in **HTTP Sessions** and are accessed through **RMI-P4**.
- The Session Failover Service is used by the Web Container for **HTTP failover** and the EJB Container for **stateful session failover**.

HTTP Failover

The Session Failover Service serializes **HTTP Sessions** and stores all information related to these **HTTP Sessions** in files or in the database. When the server process that processes the HTTP request crashes the dispatcher redirects the request to another server node. The new server process reads the information about the requested session from the persistent storage that holds the state after the last successfully ended request and continues processing the new request.

When the first server process is again up and running, the dispatcher starts redirecting the requests to it. The whole process is transparent to the user.

Enterprise beans that are involved in an **HTTP Session** are serialized by the EJB Container when the **HTTP Session** is serialized. Stateful session beans, which are accessed through **RMI/P4**, can use the **stateful session failover** provided by the Session Failover Service.

The HTTP requests are processed on the server processes. The **HTTP Sessions** are kept on the server processes, too. Therefore the **HTTP failover** is performed on server processes only.

Dispatcher or server failures in the cluster are detected and managed by the Cluster Manager.

Stateful Session Failover for Enterprise Java Beans

This feature is intended only for stateful session beans that are not invoked by Web components (RMI/P4 clients). It provides failover for stateful session EJB after a server process failure. The state of the stateful session EJB is represented in a stateful session instance, which is serialized to a persistent storage.

If the server hosting a bean instance is already crashed when a request to the bean instance reaches the J2EE dispatcher, the request will be transparently dispatched to another server node connected to the J2EE dispatcher. The new server node process will deserialize the data from the persistent storage and the bean state will be restored.

If the server process crashes during the execution of a bean method on the server node, a *com.sap.engine.services.rmi_p4.P4ConnectionException* will be thrown by the J2EE dispatcher. Clients must catch this exception when invoking a bean's business method and try to repeat the invocation. As in the transparent case, the bean state will be restored on the new server node process.

If the repeated call fails again throwing a *com.sap.engine.services.rmi_p4.P4ConnectionException*, it is most likely that the J2EE dispatcher itself is not reachable any more (or all server nodes underneath crashed). In this case, the whole set of operations on this J2EE dispatcher (=Java instance) should be discarded, a new

InitialContext obtained from the message server (see RMI/P4 loadbalancing) and the whole set of operations repeated on the new Java instance. Of course, in this case transactional issues must be strictly observed!

*Both **`HTTP failover`** and **`stateful session failover`** are disabled by default.*

If you need detailed information about how to enable the features, refer to the following sections of help.sap.com:

- Enabling and Disabling HTTP Failover
- Enabling Stateful Session Failover

Applications Running in a Cluster

Introduction

The development of a J2EE application that operates on a SAP Web AS cluster does not show any differences to the development of a J2EE application that operates on a single server, in most cases.

There are only a few cases where changes in the coding or in the configuration by the application developer are necessary.

The following guidelines should be adhered to in order to guarantee that a J2EE application can operate without changes on an SAP Web AS single server as well as on an SAP Web AS cluster.

File I/O

A J2EE application must not access the local file system, for example using the **`java.io`** package. This holds for use of EJBs in particular. If a J2EE application (EJB) relies on the file system, they may behave unpredictably in a cluster in which different servers may hold the same file with different content.

If files such as **`property files`** are needed for read-only access they can be included in a jar file (in the same application archive) and can be loaded via the class loader. Creating a new class loader is not allowed (as described in the EJB specification), but obtaining the current one is not a problem. Use **`getClass().getClassLoader()`** for obtaining the class loader of the application.

Alternatively, read-only data can be placed in the JNDI environment of an EJB. As EJBs are usually transactional components, it does not make sense to write data to non-transactional storage such as a files system. The best way is to store data in the central database.

If the data to be stored is configuration data, then SAP WebAS offers the configuration manager as another alternative for storing and accessing such configuration data. The use of the configuration manager is explained later in this document.

Static Fields/Singletons

A J2EE application should not use read/write static fields. This holds for use of EJBs particularly. The usage of static fields may cause incorrect behavior in a cluster. There is no guarantee that the different VMs in the cluster will have the same values for the static data. Therefore it is recommended to declare all static fields (at least in the EJB tier) as final.

This restriction makes it difficult to implement the singleton design pattern, especially in the EJB tier. Because the EJB specification prevents the use of static fields and synchronization, it is impossible to use singletons in a typical way.

Java Singletons, where the singleton pattern is enforced by a private (protected) constructor and the singleton instance is held in a static field, can be used in the EJB tier of a J2EE application, but limitations seriously restrict their use.

It is possible to avoid violating the restriction on static data by initializing the singleton instance in a static initializer that makes the static field final.

```
public class Singleton {
   static final Singleton instance = null;
   static {
      instance = new Singleton();
   }
   private Singleton(){
   }
   public static Singleton getInstance(){
      return instance;
   }
}
```

The common practice of instantiating the instance only if it is null in the static **`getInstance()`** operation is susceptible to race conditions, which means that it is not necessarily thread-safe (race conditions means the competition of several processes concurrently accessing a system resource).

The static initializer should not attempt to use JNDI as there is no guarantee when the EJB container will instantiate the class.

In the example the singleton instance could have been created when the instance field was declared. Using a static initializer is an option if instantiating the singleton can result in an exception. It makes it possible to catch any exception and throw it in the **`getInstance()`** operation.

As it is not possible to use synchronisation in the singleton's operations and all EJBs (running in the container) will share the same singletone instance, there is no way to limit concurrent access. This restricts the use of singletons only to read-only data caching.

Java Singletons should be used in the EJB tier only with the following restrictions:

- *The singleton instance can be initialized in a static block (or at the point of declaration) to avoid race conditions.*
- *The singleton is not too expensive to initialize.*
- *The singleton does not require JNDI access to initialize.*
- *The action of the singleton does not require synchronization.*

J2EE Pattern Service Locator

The J2EE Pattern Service Locator encapsulates the access to the JNDI API. With this approach clients become independent of the JNDI technology. Only the parameters for the initialization of the **`InitialContext`** must be still submitted to the client.

Mostly the Service Locator is implemented as Singleton so that only one instance of the **`InitialContext`** exists. Although the Service Locator should operate in a cluster, the implementation of the Service Locator as Singleton is acceptable, because the instances stored are intermediately accessed only in reading mode.

The only disadvantage might be the redundant data deposition per cluster node. In case it is not ensured to which node the request will be distributed, an instance should be available in all cluster nodes.

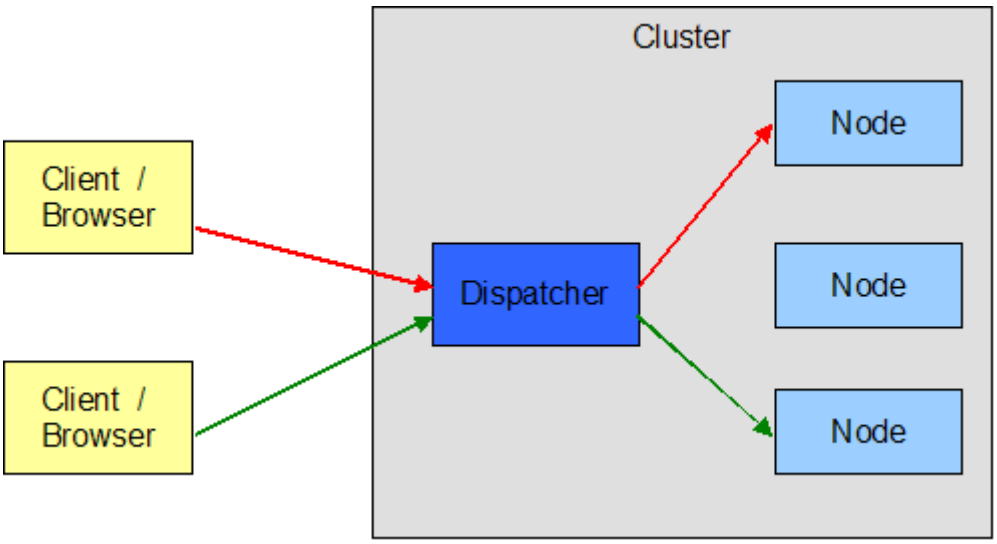
*Since in the SAP Web AS the concept of **`Sticky Sessions`** applies, there is a guarantee that requests will always be distributed to the same node within the same session. In the case that the node will fail, the Session information will be reproduced on a substituted node again.*

Therefore the cache of the Service Locator will be only built upon the node which at last increases the performance of the application.

Sticky Sessions

The concept of **`Sticky Sessions`** is also known as **`Session Affinity of an Application`** Server. In this approach the dispatcher supervises all incoming client requests and routes all requests of one definite client to the same cluster node.

In a fault case the dispatcher is also able to reroute to another cluster node. Therefore the failure safety of this approach can be guaranteed.



HTTP Session and Servlet Context

A current instance of the **`ServletContext`** can be used to attach arbitrary objects with the help of the **`set/getAttribute()`** operations, since the **`ServletContext`** instance is shared between all components in the web tier of a web application. The use of the **`ServletContext`** as an object cache positively affects the performance of a web application because many accesses to deeper layers of the application can be avoided.

The **`ServletContext`** is not replicated in the cluster. Instead, the **`ServletContext`** runs redundantly on all cluster nodes and the **`ServletContext`** instances of the single nodes are not aware of each other. If objects are attached to the ServletContext within a request, the **`ServletContext`** instances of the other cluster nodes will be not informed about this.

Well, one could come onto the idea that if an object is attached to the **`ServletContext`** within a request it can happen that this object is no longer available within a following request because this request could be distributed to another cluster node. Basically this is correct, but since the concept of **`Sticky Sessions`** applies to the SAP Web AS, there is a guarantee that requests will always be distributed to the same node within the same session.

 Exception: In case of **`HTTP session failover`**, the object references stored in the ServletContext are not transferred to a new application node and are thus lost, since the ServletContext itself is not replicated in the cluster!

Therefore, if data failover is not required, no special features must be taken into account for operating in the SAP Web AS cluster, and both **`ServletContext`** and **`HTTP Session`** are suitable for object storage.

If you want to develop a distributed, failover-ready Web application, the **`HTTP Session`** is the suitable object storage and objects which should be attached to the **`HTTP Session`** must be serializable.

 Exceptions to this rule are the following objects:

- References to enterprise beans interfaces
- Reference to **`javax.ejb.SessionContext`**
- Reference to naming contexts
- Reference to **`javax.transaction.UserTransaction`**

In addition to this:

- Do not store database or network connections – they cannot be serialized. If you are using JDBC connections, they must be closed and all references to them must be set to null before the Session Failover Service tries to serialize an **`HTTP Session`** containing JDBC connections
- Do not store JMS objects – they cannot be serialized. If you have attached JMS objects to the **`HTTP Session`**, they must be set to null and all references to them must be set to null as well before the **`HTTP Session`** can be serialized.
- Do not store large objects into the session because the application overhead becomes significant.

Deployment

The deployment process of an application on the SAP Web AS cluster is just like the deployment of an application on a single server due to the central deployment approach. Every application is initially deployed in the central database and it is synchronized with the local file system after successful deployment and during server start – this process is known as **`bootstrap`**. This makes it possible to treat the cluster like a single server concerning the deployment of applications.

Therefore, there are no special deployment-specific issues in the SAP Web AS cluster.

Enqueue Server Locking

The concept of locks is closely related to the concept of transaction isolation. Transaction isolation makes sure that two simultaneously running transactions do not affect each other. The enqueue service has the following features:

- Internally it is used for locking synchronization within the Java cluster.
- Applications can lock objects and release locks again. The enqueue service processes these requests and manages the lock table with the existing locks.

If you are using any of the SAP persistence layers the locking service is in a blackbox and you don't have to worry about clustering awareness concerning the locking service.

But if you want to implement persistence by your own – for example using Bean Managed Persistence or JDBC – you have two possibilities:

- Database locks. With this locks, you apply the locking technique provided by your database vendor.
- Logical locks. With this locks, you use a locking technique provided and managed centrally by the Web AS Java. Logical locks are managed by the **`Enqueue Server`** via a central lock table. J2EE applications use the **`LogicalLocking`** and **`TableLocking`** interfaces provided by the Locking Adapter Service. These interfaces access the Locking Manager, which in turn communicates with the **`Enqueue Server`**. To set locks, you can use the **`table-locking API`**.

 *One important point relating the SAP Web AS cluster and the Enqueue Locking is the locking type* ***`No automatic locking`*** *which can be set in the SAP specific deployment descriptor file* ***`persistent.xml`****. This type of locking is also called* `Local` ***`locking`****. It can only be used when only one server is running because it does not synchronize the locks on multiple servers defining a cluster. That is,* ***`Local locking`*** *synchronizes the concurrent access on one server only, and does not use the Enqueue Server to keep the locks. In this case, a container-managed bean instance can use the* ***`EJBLocking API`*** *to explicitly lock the database entity object representing the bean in the Enqueue Server.*

The default Enqueue Server locking type is **`Table locking`**.

For deeper and more detailed information about the Enqueue Locking please refer to:
http://help.sap.com/saphelp_nw04/helpdata/de/9a/4cdcc80fa4c747a2ccb5859f467412/frameset.htm.

Logging

The logging mechanism works like a singleton so that every cluster node has its own logging directory where node specific logging and tracing information are located. Therefore, from application development side no cluster specific implementations are necessary.

JMS

There is one important aspect concerning a cluster and JMS. Suppose you are configuring a JMS Topic for both directions. If your receiver is a MDB there will be two possible options:

- The MDBs will run on all nodes and if a message comes it will be received by all nodes.
- The MDB will run only on one server node and will receive the messages.

This behaviour can be configured in the deployment descriptors. For deeper and more detailed information please visit:

http://help.sap.com/saphelp_nw04/helpdata/de/37/30c557fad05341a951cfd051bf0b44/content.htm

Look in the section for the property **`topic-on-all-nodes`**.

About the author

Carsten Lange is a solution architect with focus on Java/JEE technology. He is also an experienced project manager in software development and has been leading development teams for several consulting organizations in Germany and USA.