**UltraESB documentation**

By Aditya Jain

**CONTENT**

1. **Overview of UltraESB**
   1. **What is ESB?**
   2. **UltraESB**
2. **Architecture of UltraESB**
   1. **Deployment Architecture**
   2. **Product Architecture**
   3. **Proxy System**
      1. **In-sequence**
      2. **In-destination**
      3. **Out-Sequence**
      4. **Out-destination**
      5. **Error-sequence**
   4. **End Points**
      1. **Address Types**
      2. **End Point Types**
3. **Configuration of UltraESB Server**
   1. **Configuration details**
      1. **Configuring the proxy target**
      2. **Configuring the proxy transport**
      3. **Configuring the endpoints**
      4. **Configuration files**
4. **Installation and Running UltraESB Server**
   1. **Installation**
   2. **Running project on UltraESB**
5. **Overview of UltraESB**
   1. **What is an ESB**

An enterprise service bus (ESB) is a software architecture for middleware that provides fundamental services for more complex architectures. In simple language, an ESB is a gateway that provides interaction between different software modules.

As different services have been provided as web-services, utilization of all of them may require intercommunication. This intercommunication between different modules or services is very essential for the proper working of the entire project. If we get down to write a java or any other language code to interact with and between two different services or software, it would take a lot of effort. Also the code would be needed to be modified for every new service that is introduced in the project. A better way to deal with the problem is to create a software bus, which knows how to configure and interact with different software services. This bus in turn can help establish proper communication between differently operated softwares.

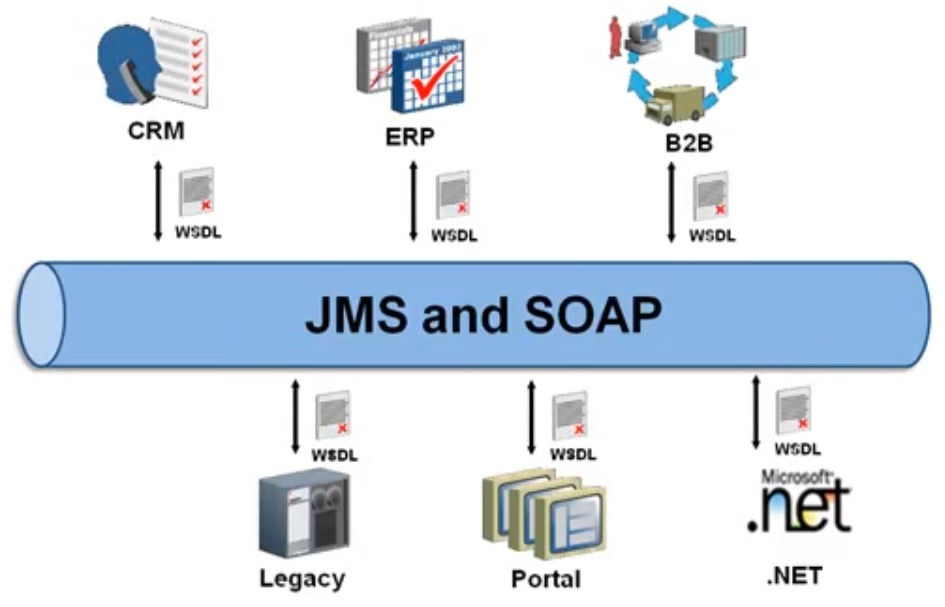
 Such software buses for communication between service software, which are mostly used on enterprise level are commonly known as Enterprise service bus or simply ESB.

Fig 1.1

Figure 1.1 shows a simple diagram of how an ESB interacts with different softwares. WSDL, i.e. web service descriptive language, is a standard language that is used for the description of a web service to the client it is exposed to. WSDL helps client understand how the web service is designed and also tells on how to use it. WSDL basically specifies the location of the service, and the methods of the service. For more information on WSDL, refer to the following link,

<https://www.w3.org/TR/wsdl>

Since an ESB acts as a interlink between different services, services which may not be able to communicate directly due to the difference of their architecture and the programming language understood by them, their needs to be a way they can exchange information. That’s where the use of transport protocol comes in handy. There are different transport protocols which can be used for the message transfer such JMS(Java Messaging Service) as shown in figure 1.1. Other transport protocols such as HTTP, Email, TCP, FTP etc. can also be used.

These transport protocols defines a set of rules and regulation for the message to be transferred, which may contain headers and a payload. A payload is the most important part of the message as it contains the actual meaningful information or data that is to be transferred. Headers are additional information set which are sent along with the payload. They basically help in for the identification and security purpose of the message sent and may also contain useful information about the source and destination addresses of the message. In simple words, without headers a message would be just a chunk of data without a name and address.

Payload plays a vital part of any message, as it contains the main information or data of the client’s request or the server’s response. The payload can be written in different formats as well. These formats may include SOAP (Simple Object Access Protocol), XML, HTML, JSON etc.

Hence an ESB may use a combination of different transfer protocol and the payload type depending on the web services. Different web services may support different message formats, hence another job of an ESB is to translate the payload suiting to the web service it is redirected to. For example, web service “A” may support XML, while web service “B” may support JSON, hence it is the job of the ESB to convert the message coming from XML to JSON, when it is directed form “A” to “B”, and vice versa.



Fig 1.2

* 1. **UltraESB**

The UltraESB is a Free and Open Source Enterprise Service Bus provided by the [AdroitLogic](http://adroitlogic.org/). It facilitates the integration of different systems; both within an organization and beyond. Integration is facilitated via "messages" which maybe received or transmitted over one of many transports supported, such as;

* HTTP/S - Hypertext Transfer Protocol / HTTP-SSL
* JMS - Java Messaging Service
* AMQP - Advanced Message Queuing Protocol
* Email - Electronic Mail
* TCP - Transmission Control Protocol
* MLLP/S - Minimal Lower Layer Protocol (used by HL7) / MLLP-Secure
* Files
* FTP/S - File Transfer Protocol / FTP-SSL
* SFTP - SSH (Secure) File Transfer Protocol etc.

Messages may carry different types of payloads such as;

* SOAP - Simple Object Access Protocol
* REST - Representational State Transfer
* XML - Extensible Mark-up Language
* HTML - Hypertext Mark-up Language
* JSON - JavaScript Object Notation
* Text
* Binary
* Protocol Buffers - Google protocol buffer binary format
* CSV - Comma-Separated Values
* EDI - Electronic Data Interchange
* HL7 - Health Level 7
* Hessian
* AS2 - Applicability Statement 2
* Maps etc.,

UltraESB is a powerful tool which can accept messages over one transport in one format, and forward it to another system over another transport and another format.

Messages passing through UltraESB can be “**Mediated**” meaning different functionality can be applied on it. A message may require a several level of validation, after which only, it can proceed for further analysis and response creation; hence the mediation logic provided by UltraESB comes in really handy which will be discussed later in this documentation.

* + 1. **Acting as a gateway between your organization and your partners**

By deploying the UltraESB between your partners, clients and users, it can be used to;

* validate messages received
* verify security
* perform decryption
* filter requests for injection attacks
* throttle or rate/quota limiting
* response caching

Proceeding further, the message can be routed to one or more internal services over different transport.

UltraESB can transform the message payload using any of the following;

* XSLT
* XQuery
* Milyn
* Java
* Groovy or
* any other type of transformation or
* use your favorite external library or
* custom code you’ve developed

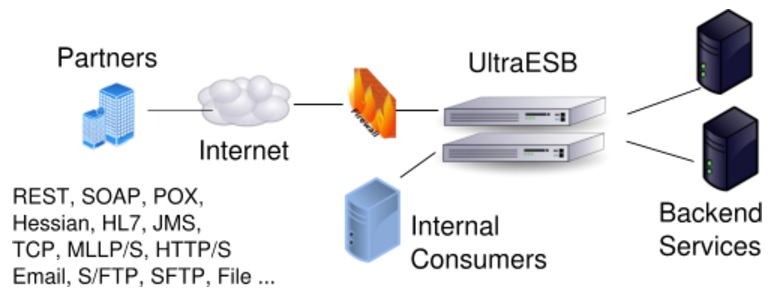
**** Fig 1.3

Figure 1.3 shows a basic diagram of how the UltraESB acts as a gateway for the accessing the backend services provided to the client. As show, the partners can access the services through internet over any transports protocol such as HTTP, JMS, FTP etc. and in any format such as SOAP, REST, simple XML etc.

1. **Configuration of UltraESB Server**

As discussed in section 2 an UltraESB server basically works on the proxy system, where different proxies are used for message handling and redirection. The server should be aware of all the proxy details, the mediation logic and also the different transports that are used in the project. Hence it is needed that the UltraESB server is properly configured. It is done through the use of a “Deployment unit”.

A deployment unit is a file that contains a collection of proxy services, sequences/mediation, endpoints, transports, any custom resources and libraries used by the mediation flows. Deployment units are either directory or a zip/uda archives placed at conf/deployments directory. A spring configuration file named **ultra-unit.xml**, is at the top level of the deployment unit which contains all the dynamic configuration details. These can be loaded, unloaded or updated - without bringing down the ESB with a graceful maintenance shutdown.

Apart from the dynamic aspects, some of the static configuration elements that define the configured transports, Work Managers, File Caches, JMX connection semantics and other static aspects of an ESB configuration, is configured using another spring configuration file named **ultra-root.xml**. The user is responsible for selecting the required transports for a solution, from the set of supported transports, and for configuring them as per production deployment requirements. For example, the HTTPS transport will require a user to setup the certificates, while the JMS transport will require the user to configure JMS provider access via standard spring notations.

* 1. **Configuration details**

For the configuration of proxies, the UltraESB namespace <u> needs to be defined in the spring configuration file. This is done using the XML-Schema instance declaration in the <beans> attribute. The following code snippet shows the bare deployment unit configuration,

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:u="http://www.adroitlogic.org/ultraesb"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.2.xsd

http://www.adroitlogic.org/ultraesb http://schemas.adroitlogic.org/ultraesb/v2\_6/ultraesb-artifacts.xsd">

*<!-- Add your proxy service configurations here -->*

</beans>

Now that the IDE is aware of the UltraESB namespace, we can move forward to declaring proxies. A proxy is defined using the “proxy” element of the UltraESB namespace. It may have “id” attribute which defines the name of the proxy.

<u:proxy id="Proxy\_Name">

<u:target></u:target>

</u:proxy>

Under the “proxy” element would be defined two sub elements, “target” and “transport”. The “target” element defines the proxy sequencing such as in-sequence, out-sequence etc. as discussed in the previous section. The “transport” element is used for defining the transport the message should be carried on.

* + 1. **Configuring the proxy target**

Proxy target defines the sequencing of the message and may also include the mediation logic itself.

The “target” element contains five attributes oar sub-elements, namely;

* In-sequence
* In-destination
* Out-sequence
* Out-destination
* Error-sequence

In-sequence specifies the java bean reference where the incoming message should be forwarded to for mediation process. The following code snippet shows how the in-sequence is configured;

<u:proxy id="ProxyName">

<u:target inSequence="reference\_to\_java\_bean"/>

</u:proxy>

The above code shows that, the in-sequence logic has been referred from a java class. It is not necessary to include a java class for sequencing; instead the in-sequencing mediation logic can also be included in line with the proxy declaration as shown in the code snippet below.

<u:proxy id="MyFirstProxy">

<u:target>

<u:inSequence>

*<!-- Sequence definition -->*

</u:inSequence>

</u:target>

</u:proxy>

The other sequences can also be defined in the similar process as show for the in-sequence mediation.

* + 1. **Configuring the proxy transport**

For any proxy service to be reachable over a given transport, the proxy configuration should define the transport on which it is exposed. The other sub-element of the “proxy” element, i.e. “transport”, hence is used for specifying the transport for which the proxy is available. The transport element also has identifier attribute “id”. The “id” specifies which transport it is referring to. Generally all the transport elements are defines in the **ultra-root.xml** configuration file and is imported in the **ultra-unit.xml**. The following code snippet shows the basic configuration of the proxy transport.

<u:proxy id="MyFirstProxy">

<u:transport id="http-8280"/>

<u:target></u:target>

</u:proxy>

* + 1. **Configuring the endpoints**

The endpoints are basically defined for when the message is to be directed to an external web service. Every incoming message needs to be directed to probably other web service after the mediation process is complete. This is done by defining the endpoint either by the in-destination tag elements of the target element, or by defining the endpoint using the “endpoint” tag element of the UltraESB namespace. This endpoint can be invoked later after the mediation logic is implemented.

The “endpoint” element contains an identifier attribute “id” and an element tag “address”. The address tag specifies the address where the message has to be redirected. The below given code snippet shows how an endpoint is configured.

<u:endpoint id="My\_Endpoint" type="round-robin">

<u:address>http://localhost:9000/service/EchoService</u:address>

<u:address>http://localhost:9001/service/EchoService</u:address> <u:address>http://localhost:9002/service/EchoService</u:address>

</u:endpoint>

As shown in the above code snippet, it is possible to specify multiple redirecting addresses to an endpoint. The message will be redirected to all of the specified addresses in a fashion that is specified by the “type” attribute. In the above code, round-robin fashion has been used for the message delivery. Delivery type can also be specified as Fail-over, Round-robin with fail-over, weighted, weighted with fail-over, random and random with fail-over. For more information about the endpoint types, refer to the following address;

[http://docs.adroitlogic.org/ultraesb/v2\_3/architecture-and-design/index.html#EndpointsDestinationsandAddresses-EndpointTypes](http://docs.adroitlogic.org/ultraesb/v2_3/architecture-and-design/index.html%23EndpointsDestinationsandAddresses-EndpointTypes)

* + 1. **Configuration files**

As discussed previously, the configuration is done in the spring configuration files **ultra-unit.xml**, **ultra-root.xml** and maybe **ultra-custom.xml**. The entire proxy configuration as discussed in the previous three sections can be done in the ultra-unit.xml configuration file. The ultra-unit.xml file is at the top level of the deployment unit, hence can be used for all the proxy configuration, but to make the code a little neat, other files such as ultra-root.xml and ultra-custom.xml is used.

The ultra-root.xml file is used to define the transport related information such as transport listener and transport sender, work managers and file caches. The following is a code snippet of the ultra-root.xml file used in the project.

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:u=*"http://www.adroitlogic.org/ultraesb"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.1.xsd*

*http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.1.xsd*

*http://www.adroitlogic.org/ultraesb http://schemas.ultraesb.org/ultraesb-2.0.xsd"*>

<bean id=*"fileCache"* class=*"org.adroitlogic.ultraesb.core.PooledMessageFileCache"*>

<constructor-arg value=*"200"* type=*"int"* />

</bean>

<bean id=*"http-8280"* class=*"org.adroitlogic.ultraesb.transport.http.HttpNIOListener"*>

<constructor-arg ref=*"fileCache"* />

<property name=*"port"* value=*"8280"* />

</bean>

<bean id=*"http-sender"* class=*"org.adroitlogic.ultraesb.transport.http.HttpNIOSender"*>

<constructor-arg ref=*"fileCache"* />

</bean>

<bean id=*"jms-lsnr"*

class=*"org.adroitlogic.ultraesb.transport.jms.JMSTransportListener"*>

<property name=*"connectionFactory"* ref=*"hornetQJMSConnectionFactory"* />

<property name=*"transactionManager"* ref=*"txManager"* />

<property name=*"sessionTransacted"* value=*"true"* />

</bean>

<bean id=*"jms-sender"*

class=*"org.adroitlogic.ultraesb.transport.jms.JMSTransportSender"*>

<property name=*"template"*>

<bean id=*"jmsProducerTemplate"* class=*"org.springframework.jms.core.JmsTemplate"*>

<property name=*"connectionFactory"* ref=*"hornetQJMSConnectionFactory"* />

<property name=*"pubSubDomain"* value=*"false"* />

<property name=*"sessionAcknowledgeModeName"* value=*"SESSION\_TRANSACTED"* />

</bean>

</property>

</bean>

</beans>

The ultra-custom.xml file is mostly used for the user defined data such as defining the custom beans in the configuration files and other relevant information. The user may or may not use ultra-custom.xml, instead ultra-root.xml or ultra-unit.xml can also be used for custom bean definitions. But it is always better to separate the code logic in an understandable way.

1. **Installation and Running UltraESB Server**

UltraESB is freeware software provided by the adroitlogic.org and its installation is a simple process as it is a standalone Java application.

* 1. **Installation**

Installing UltraESB is simply downloading and extracting (.zip) the binary distribution, hosted at the downloads page of AdroitLogic, into your software installation directory. The following steps will guide you through the installation process;

1. Download the UltraESB software package for the adroitlogic.org website, or if you already have the package in your system, no need to download it again.
2. Now extract the downloaded UltraESB package zip file to the software installation directory, for instance,

C:\Program Files\ultraesb-2.6.0

1. The installation process is now done.
2. Once the installation is done, you can browse through the UltraESB installation folder to study about the structure of the installation.
3. Refer to the following data for information about the UltraESB files,

**/bin**

    ultraesb.sh/bat - starts the UltraESB

    uterm.sh/bat - starts the UltraESB management terminal

    toolbox.sh/bat - starts the SOA ToolBox

    encrypt.sh/bat - encrypts password to be stored on the configuration file

    ultraesb-daemon.sh - starts and manages the UltraESB on production deployments (start | stop | etc…)

    zkServer.sh/cmd - startup script for starting ZooKeeper for UltraESB clustering

    zkCli.sh/cmd - command line client for ZooKeeper

    zkEnv.sh/cmd - helper script to setup the environment for the zkServer and zkCli

    zkCleanup.sh - cleanup old transaction logs, snapshots of ZooKeeper

    zkServer-daemon.sh - starts and manages the Zookeeper on production deployments (start | stop | etc…)

    /native - directory containing the wrapper native libraries

**/conf**

    ultra-root.xml - the root configuration file that contains the core engine configuration

    ultra-custom.xml - the custom spring beans that are statically included into the root configuration

    ehcache.xml - ehcache configuration for the UltraESB caching implementation, better keep it as it is

    encrypted.properties - phrases for the encrypted information

    log4j2.xml - UltraESB loggers configuration for log4j2

    log4j\_zk.xml - ZooKeeper logger configuration for log4j2

    wrapper.conf - wrapper configuration used by the UltraESB daemon script

    zoo.cfg - ZooKeeper configuration

ldap.conf - ldap configuration

    /deployments - the directory containing the deployment units

        /default - the default deployment unit

            ultra-unit.xml - the configuration of the default deployment unit which holds proxy service definitions

            /lib - the libraries related to the default deployment unit (this is empty by default as there are no libraries required for the default deployment unit)

            /classes - mediation classes for the default deployment unit

            /src - mediation sources for the default deployment unit

    /monitoring - monitoring related configurations

        ultra-metrics.xml - the metrics engine configuration

        alert-configs.json - the alert configurations defined in a JSON format

        default-alert-actions.json - default alert actions defined for all alerts (alert actions can also be defined per alert configuration)

    /uz - Zabbix configuration files

/elasticsearch - elasticsearch configuration files

elasticsearch.yml - embedded ElasticSearch configuration

/management - management related configurations

        uterm.properties - configurations of the UltraESB management terminal

        log4j2\_uterm.properties - UltraESB management terminal loggers configuration for log4j2

/mediation

        /classes - default location where mediation sequences are compiled

        /src - default location for any Java source code to be picked up by sequences

    /keys - the Keystores for private keys and certificates

/license

license.conf.properties - configuration file for license validation

license.key.properties - product license file

**/data** - default data directory used by the UltraESB

    /zookeeper - default data directory used by the ZooKeeper

/elasticsearch - default data directory used by ElasticSearch

**/docs** - documentation assets of UltraESB

/schema - schema file for ultraesb-2.6.0

**/lib** - ultraesb and other core libraries

    /patches - patches to any of the libraries used by the UltraESB - this location occurs first in the classpath

    /custom - libraries developed or third-party libraries added by the user to be used in sequences etc

    /endorsed - Java endorsed directory for UltraESB

    /optional - contains optional JAR files. Files not used in your installation maybe deleted

    /samples - contains JAR files used in samples. Maybe deleted in a production environment

    /test - contains JAR files used for unit tests. Maybe deleted in a production environment

/uterm - contains JAR files required for UTerm

**/logs** - default log directory

/elasticsearch - default log directory for elasticsearch

**/samples**

    /conf - sample configuration files

    /resources - XSD’s, WSDL’s, XSLT’s etc. used by samples

    /src - source code for samples and unit tests

    /webapp - Web application loaded into the sample Jetty server for samples and unit tests

    /bin - script files for load tests

**/tmp** - temporary files

**/webapp** - UltraESB web application, which can be used to install UltraESB on any J2EE container

* 1. **Running project on UltraESB**

As the project is dependent on different software models, hence running them before UltraESB is required. The information below will guide you through the process of deploying the project on the UltraESB server.

* **Running HornetQ**

1. Start the hornetQ by running RUN.BAT from HORNETQ\BIN file directory

* **Running Coherence**

1. Make sure COHERENCE\SCRIPT\GRADLE\SERVER\ gradle.properties has the right coherence version.
2. Start the server using cmd, from COHERENCE\SCRIPT\GRADLE\SERVER folder by typing “gradle start”
3. Now, make sure COHERENCE\SCRIPT\GRADLE\LOADER\dev.properties has the right coherence version.
4. Also, make sure COHERENCE\SCRIPT\GRADLE\LOADER\DATA folder has required csv files.
5. Now, Start loading data when server is up from COHERENCE\SCRIPT\GRADLE\LOADER\ folder using cmd with the command “gradle loadData” to load all data and “gradle load\_%csv filename%” to load a specific file.

* **Running UltraESB**

1. Make sure all the dua deployment files of the project are pasted in ULTRAESB\CONF\DEPLOYMENTS.
2. All the properties related to xmla are present in ULTRAESB\CONF\RESOURCES\XMLADAPTOR\ultra-conf.properties.
3. The jars which are required for XMLA specifically should be in ULTRAESB\LIB\CUSTOM (e.g XMLA depends on xml\_common, cdm, coherence, endeca etc)
4. Start the server from ULTRAESB\BIN

- by running ultraesb.bat directly

- or alternatively open command prompt, and type

“cd ULTRAESB\BIN”, then “ultraesb.bat”

1. To start the server in debugging mode, type

“ultraesb.bat –xdebug” in the cmd.