

Oracle WebLogic Server 12.1.3

Developing with WebLogic Server

ORACLE WHITE PAPER | JUNE 2014







Table of Contents

Introduction	4
Enabling Development of Modern Applications with Java Standard APIs	5
Java API for WebSocket 1.0: Enabling Highly Interactive Applications	6
A Brief Look at the WebSocket Protocol	6
Developing WebSocket Applications	8
WebSocket Encoders and Decoders	11
WebSocket Clients	11
Working with the WebSocket Protocol in WebLogic Server 12.1.3	14
WebSocket Client Emulation	14
Java API for JSON Processing 1.0: Simplifying JSON Parsing and Processing	15
The Object Model API	15
The Streaming API	16
Working with JSON in Oracle WebLogic Server 12.1.3	16
JAX-RS 2.0: Java API for RESTful Web Services	17
Client API	18
Interceptors and Filters	19
Asynchronous Support	22
Working with JAX-RS 2.0 in Oracle WebLogic Server 12.1.3	23
Jersey 2 Server-Sent Events	25
JPA 2.1: Evolution of Java Persistence	27
Using JPA 2.1 on Oracle WebLogic Server 12.1.3.	28



Innovation Features	29
Oracle TopLink and TopLink Data Services	29
Spring	29
OSGI	29
Developer Ease of Use	30
Zip File Developer Distribution	30
Development Tooling Support	33
Classloading Analysis and Configuration	34
Oracle Apache Maven Support	35
Overview of Working with Oracle Maven Support	35
Oracle Maven Artifacts	35
Oracle Maven Synchronization Plugin	36
Oracle WebLogic Maven Plugin	37
Installation and Lifecycle Goals	38
Deployment and Resource Management	39
Development and Pre-Compilation	40
Oracle Maven Archetypes	40
Continuous Integration Practices	40
Conclusion	41
References	42



Introduction

For developers, Oracle WebLogic Server has always been an efficient and compelling development platform through its developer-oriented features, including the following:

- » Providing a high quality Java EE implementation;
- » Offering a smaller zip based distribution that is faster to download, setup and use than the standard GUI based installers;
- » Establishing strong integration points with leading IDEs such as Eclipse, JDeveloper and NetBeans enabling developers to easily develop, deploy, test and debug applications using WebLogic Server;
- » The FastSwap capability which reduces the impact of redeployment operations on development and test cycles through dynamically reloading classes that change during development without requiring redeployment operations to be performed;
- » Building in support for Apache Maven, initially through the inclusion of a plugin that enables deployment operations to be performed from the Maven lifecycle and extended in following releases to support a greater set of configuration and operational tasks;
- » Introducing innovations such as the early use of annotations for development, enabling support for development of rich Web applications with the Http PubSub Server implementation and providing integration points with popular open source frameworks such as Spring;

With the release of Oracle WebLogic Server 12.1.3 a significant set of new features for developers have been provided, including:

- » Additions of several key new Java standards for developing applications that expose and consume REST based web services, use WebSocket connections to send data between clients and servers in a bi-directional manner, use JSON to represent data payloads, and make use of JPA to access data;
- » Defining a set of Oracle specific Maven artifacts for common APIs and libraries, which cover the WebLogic Server, Coherence and TopLink components;
- » Bundling a recent version of Apache Maven as part of the standard product distribution;
- » Providing a new Apache Maven plugin, oracle-maven-sync, which manages the installation of the Oracle defined artifacts into a Maven repository;
- » Providing a new version of the WebLogic Maven Plugin that executes all of its operational goals without requiring a local installation of WebLogic Server, enabling it to more easily integrate into continuous testing environments;
- » Supplying a set of Maven Archetypes for WebLogic Server and Coherence that can be used by developers to generate starter projects for working with common application patterns;
- » Value add features that support the development innovative new applications of to be developed, such as support for the WebSocket Protocol (RFC 6455) for developing applications that communicate with rich clients in a bi-directional, network efficient manner; TopLink Data Services that provides a no-programming approach for enabling REST access to data including support for



the emission of live-data notification events as data changes; support for the use of OSGi in applications through the ability to define and run OSGi frameworks that provide common OSGi services and the use of OSGi bundles within standard Java EE applications.

As businesses adapt and embrace the use of mobilized applications, developers are increasingly looking for ways to provide portable services and APIs utilizing portable data formats that can be accessed from a variety of different browsers and devices. They are also looking to optimize the way in which applications communicate with these back-end services to send and receive data in a timely and efficient manner. Oracle WebLogic Server 12.1.3 provides features that directly support and enable the development of modern, mobilized applications with updates of Java standard APIs that compliment its existing Java EE 6 implementation base.

New and Updated Java Standard APIs in WebLogic Server 12.1.3:

- » **Java API for WebSocket 1.0. New.** Java standard API for the development of applications that use the WebSocket protocol for creating bi-directional, persistent connections between client and servers, enabling applications to exchange data in a highly efficient, low latency manner.
- » **Java API for JSON Processing (JSON-P) 1.0. New.** Java standard APIs for creating and parsing JSON (JavaScript Object Notation) objects, enabling applications to more easily and efficiently exchange data in a portable form;
- » **Java API for RESTful Services (JAX-RS) 2.0. Update.** Java standard API for developing REST based services, updated in this release to provide a standard Client API for calling REST services, adding Filters and Interceptors to manipulate requests and responses. The included Jersey implementation of the JAX-RS 2.0 specification further provides support for working with the Server-Sent Event communication mechanism using a familiar JAX-RS programming model;
- » **Java Persistence API (JPA) 2.1. Update.** Java standard API for managing the persistence of data, with new features such as supporting bulk update changes via the Criteria API, support for invocation of database stored procedures, standardization of schema and database artifact generation and more.

Enabling Development of Modern Applications with Java Standard APIs

The broadening adoption of HTML5 is allowing developers to build highly interactive and dynamic applications on a scale never seen before. These new breeds of applications are not only incredibly visually appealing and responsive to different device form factors, they are able to incorporate information in real time from many different sources and display it in a highly interactive and usable manner. The growth in the use and acceptance of HTML5 and its counterpart technologies in JavaScript and CSS enables applications to be written once and accessed from a range of devices such as smart phones, tables, and desktops where it will render and respond correctly.



With this rapid move into a world of mobile based highly dynamic applications, coupled with a usage pattern on a 24x7 global nature, there is a corresponding need to develop and efficiently scale the back end services that interact with these clients, moving away from some of existing models that allowing servers to send data to clients.

Oracle WebLogic Server 12.1.3 provides a strong foundation for the rollout of modern dynamic applications through the addition of new Java standards that have been specifically provided to support for bi-directional network connections using the WebSocket Protocol, for working with data payloads expressed in the lingua franca of HTML5 applications in JSON. Updates have been provided for existing popular APIs such as JAX-RS 2.0 and JPA 2.1 that add even more capabilities for development of REST based web services and for efficient data access.

Java API for WebSocket 1.0: Enabling Highly Interactive Applications

Oracle WebLogic Server 12.1.3 supports the use of the WebSocket protocol (RFC 6455) which enables a client to create a lightweight, bi-directional, persistent connection with a server that supports the sending of messages from either connected peer at any time. It does not require the use of the inefficient techniques commonly employed today, such as polling or long-polling mechanisms, for messages sent from the server to be received by clients.

The persistent, bi-directional connection the WebSocket Protocol enables the client or the server to send messages to each other any time to each other. This makes it an ideal solution for creating modern applications that need to receive and exchange data from various services to provide the highly interactive and dynamic experience for users.

A good analogy for WebSockets is a phone call. When making a phone call, you initiate the call by dialling a number. If the party you are trying to call accepts the call by picking up the receiver, the connection is established. While the connection is active, both parties can speak at the same time if they want to (although it's not recommended for a free flowing conversation!) and both parties can hear what is being said even as they are talking themselves. This is what is meant by full-duplex communication. The connection remains active, whether or not anyone is talking, until either one of the parties decides to hang up.

Danny Coward, Java WebSocket Programming, Oracle Press 2013

A Brief Look at the WebSocket Protocol

With the WebSocket protocol, a WebSocket connection is first started with a client making a standard HTTP request to a WebSocket end point containing a request to upgrade the connection to use the WebSocket protocol. Within the request it passes along some additional information that indicates what WebSocket capabilities it supports. This is called the opening handshake.

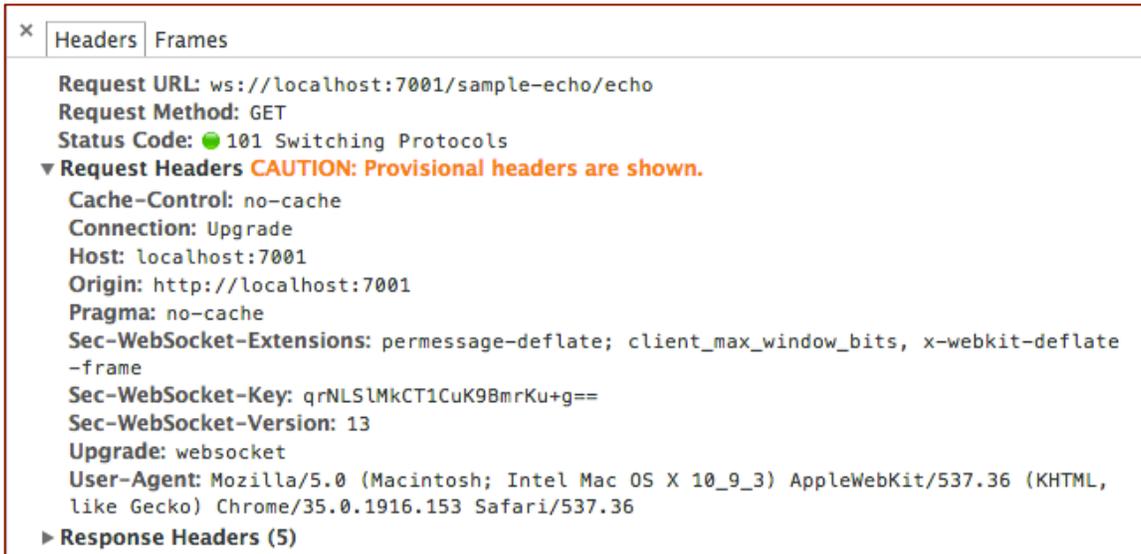


Figure 1 WebSocket Opening Handshake

If the server is willing and able to accept the connection request, it returns a HTTP response indicating that it is able to upgrade the connection, the protocol it will upgrade to and some WebSocket specific information. This response is called the opening handshake response.



Figure 2 WebSocket Opening Handshake Response

After the successful completion of this request/response exchange, the HTTP connection is replaced with a WebSocket connection, using the same network address and ports. Once the WebSocket connection has been established, the client and the server can then freely send WebSocket back and forth. The connection stays alive until either peer decides to explicitly close the connection or something external such as a network timeout due to inactivity or a problem with the underlying network causes the connection to be closed.

Data is sent over a WebSocket connection as a series of WebSocket messages. Each WebSocket message is sent within a data frame, with long messages being broken up into multiple frames that are sent in order, with the recipient reassembling them to reproduce the original message.

WebSocket Protocol

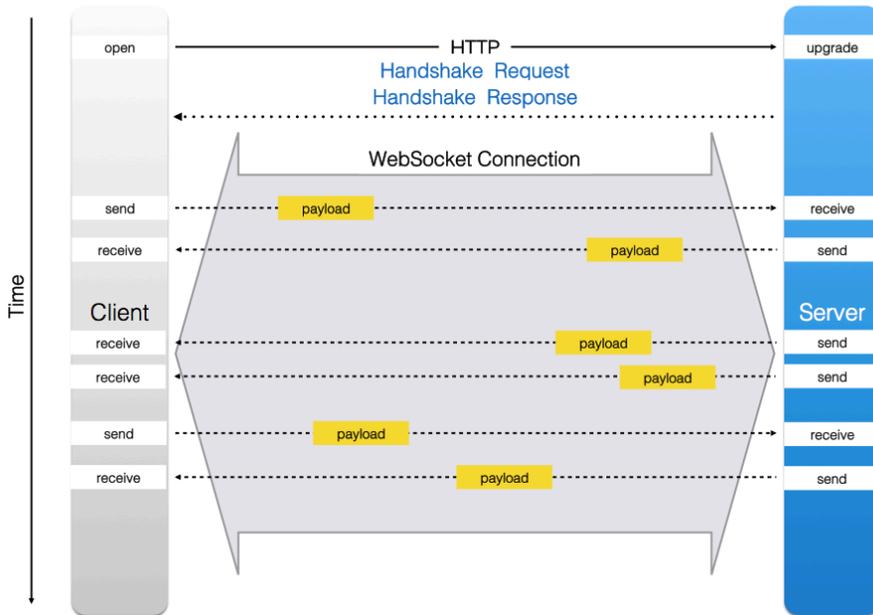


Figure 3 WebSocket Interactions

Developing WebSocket Applications

The JSR-356 Java API for WebSocket 1.0 specification introduces a standard programming API for creating WebSocket applications in Java. The API introduces the concept of a WebSocket Endpoint, which is responsible to handling one side of a conversation, sending and receiving messages as required and responding to the lifecycle activities that are imposed on it.

The server-side of the WebSocket connection is known as a `ServerEndpoint`. There are two approaches defined in the specification for creating `ServerEndpoint` implementations – an annotation based approach, where the configuration of the Endpoint itself and the appropriately defined lifecycle methods are specified using annotations, such as `@OnOpen`, `@OnMessage` and so forth. The second approach is with its programmatic API that provides developers with base classes and interfaces that are used to provide the necessary lifecycle methods and to dynamically define and configure Endpoint instances.

The core lifecycle events that are sent to an application that exposes WebSocket endpoint are the following:

- » **Connection Opened:** this event occurs when the WebSocket endpoint is being asked to open a connection a peer, offering developers the opportunity to interact with the Session and possibly perform any initialization tasks the WebSocket endpoint will use;
- » **Message Received:** a message has been received from a peer. The message can be either a text message, a binary message or a pong message and may be received in either full or partial form;
- » **Error Occurred:** an error has occurred on the WebSocket connection, enabling developers the opportunity
- » **Connection Closed:** the connection has been closed by the peer, offering developer

Using the annotation based programming model, an application that presents itself as a WebSocket endpoint and participates in the lifecycle to receive and send messages can be expressed with a very simple class of only a few lines of code.

For example, as a slight variant on the usual echo WebSocket, the following class receives a message from a WebSocket client and sends it back in a reversed form.

```
import javax.websocket.OnMessage;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/echoserver")
public class EchoServer {

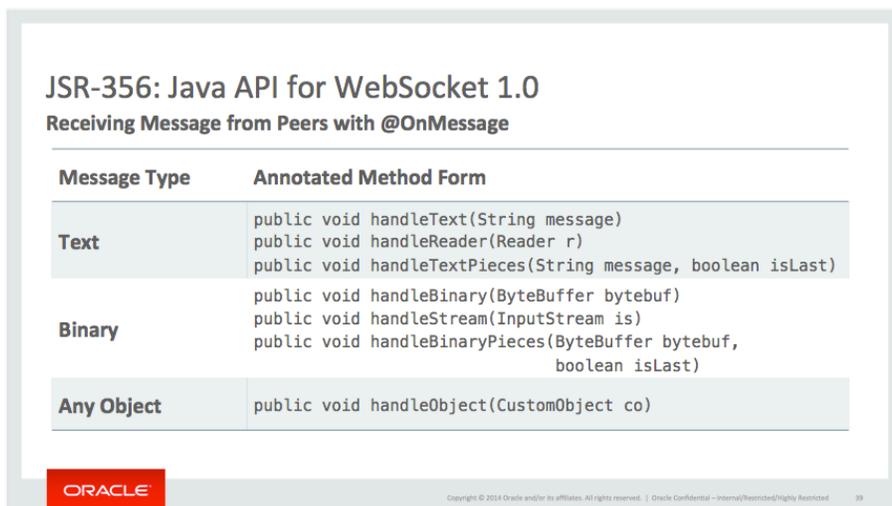
    private String returnMessage = "I got this message '%s' and am sending it back in reverse '%s'";

    @OnMessage
    public String echo(String message) {
        return String.format(returnMessage,
            message,
            new StringBuilder(message).reverse());
    }
}
```

Figure 4 WebSocket Annotation Example

The **@ServerEndpoint** annotation denotes the class as being a `ServerEndpoint`, which the WebSocket implementation detects and instantiates using the supplied value as the URI to access it.

The **@OnMessage** annotation denotes the method used to handle incoming WebSocket messages. In this simple case the method receives a whole text message. Additional method definitions can be used to receive binary messages as well as handling partial forms for both message types.



The slide content is as follows:

JSR-356: Java API for WebSocket 1.0

Receiving Message from Peers with @OnMessage

Message Type	Annotated Method Form
Text	<code>public void handleText(String message)</code> <code>public void handleReader(Reader r)</code> <code>public void handleTextPieces(String message, boolean isLast)</code>
Binary	<code>public void handleBinary(ByteBuffer bytebuf)</code> <code>public void handleStream(InputStream is)</code> <code>public void handleBinaryPieces(ByteBuffer bytebuf, boolean isLast)</code>
Any Object	<code>public void handleObject(CustomObject co)</code>

ORACLE
Copyright © 2014 Oracle and/or its affiliates. All rights reserved. | Oracle Confidential - Internal/Restricted/Highly Restricted 39

Figure 5 Method Forms for @OnMessage

To use the programmatic API the Endpoint class can be extended to define a WebSocket endpoint. The Endpoint class defines an equivalent set of lifecycle methods that can be overridden to handle WebSocket open, error and close events.

To receive messages from a peer, a `MessageHandler` class is created which provides an `onMessage` method. This method is invoked when messages are received by the `Endpoint`. As with the annotated model, the `MessageHandler` and requisite `onMessage` method can take one of several forms to allow it to handle text and binary messages that are received as either whole or partial messages.

```
import java.io.IOException;
import javax.websocket.Endpoint;
import javax.websocket.EndpointConfig;
import javax.websocket.MessageHandler;
import javax.websocket.Session;

public class ProgrammaticEchoServer extends Endpoint {

    private String returnMessage = "I got this message '%s' and am programmatically sending it back
in upper-case reverse '%S'";

    @Override
    public void onOpen(Session newSession, EndpointConfig endpointConfig) {
        final Session session = newSession;

        session.addMessageHandler(new MessageHandler.Whole<String>() {
            @Override
            public void onMessage(String msg) {
                try {
                    session.getBasicRemote().sendText(
                        String.format(returnMessage, msg, new StringBuilder(msg).reverse()));
                } catch (IOException ioe) {
                    ioe.printStackTrace();
                }
            }
        });
    }
}
```

Figure 6 Simple Example of Programmatic Endpoint

When the programmatic API is used the configuration for the `Endpoint` is provided through an implementation of the `ServerApplicationConfig` interface. In this implementation a `ServerEndpointConfig` instance is created that contains the `Endpoint` class(es) and the URI(s) for clients to access it. The `ServerEndpointConfig` class can also handle more complex tasks such as sub-protocol negotiation, origin server checking and so forth.

```

import java.util.HashSet;
import java.util.Set;
import javax.websocket.Endpoint;
import javax.websocket.server.ServerApplicationConfig;
import javax.websocket.server.ServerEndpointConfig;

public class ProgrammaticEchoServerConfig implements ServerApplicationConfig {

    @Override
    public Set<ServerEndpointConfig> getEndpointConfigs(Set<Class<? extends Endpoint>> set) {
        // throw new UnsupportedOperationException("Not supported yet.");
        Set<ServerEndpointConfig> config = new HashSet<>();
        ServerEndpointConfig endpoint =
            ServerEndpointConfig.Builder
                .create(ProgrammaticEchoServer.class, "/programmatischechoserver")
                .build();
        config.add(endpoint);
        return config;
    }

    @Override
    public Set<Class<?>> getAnnotatedEndpointClasses(Set<Class<?>> set) {
        // throw new UnsupportedOperationException("Not supported yet.");
        return set;
    }
}

```

Figure 7 Programmatic WebSocket Endpoint

WebSocket Encoders and Decoders

The JSR-356 Java API for WebSocket 1.0 specification specifies type types of Java objects that can be sent and received as WebSocket messages, covering String, the primitives (and their Object counterparts) and binary message types with a byte array. For working with custom Java objects, the specification defines the concept of Encoder and Decoder components, which can be implemented to transform custom Java objects to and from a format that can be sent as a WebSocket message.

See the Java API for JSON Programming section for examples of Encoder and Decoders that use that API to transform Java objects to and from JSON forms.

WebSocket Clients

The most typical WebSocket clients in use today are those created with web applications, where a HTML page is combined with JavaScript to create active pages for users. The WebSocket clients in these cases are developed using the W3C WebSocket JavaScript API. This small useful JavaScript API provides client side developers with a means to open a WebSocket connection with a server, send messages over the connection and to respond to events that occur on the WebSocket connection. These events cover cases for when data is received on the connection; when the connection is closed and when errors are detected.

This event based programming model enables developers to create applications that respond to data when it is received instead of issuing a request and waiting for the response. For developers that have been using the asynchronous Ajax programming style where HTTP requests are made and callback functions used to handle the asynchronous responses when they are received, the WebSocket API has a very familiar feel, but offers the benefits of using the more efficient and powerful WebSocket Protocol.

For example, a simple WebSocket client application to call the reserving echo WebSocket endpoint would make calls as shown below, to open the connection, send a message and receive a message back via the onmessage callback function.

```

function initWebSocket() {
    wsUri = ((window.location.protocol == "https:") ? "wss://" : "ws://")
        + window.location.host
        + "/websocket-356-echo/echoserver";

    writeToScreen("Connecting to " + wsUri);

    echo_websocket = new WebSocket(wsUri);

    echo_websocket.onopen = function(evt) {
        writeToScreen("Connected!");
    };

    echo_websocket.onmessage = function(evt) {
        writeToScreen("Received message: " + evt.data);
    };

    echo_websocket.onerror = function(evt) {
        writeToScreen('<span style="color: red;">ERROR:</span> ' + evt.data);
    };
}

function doSend() {
    echo_websocket.send(msgid.value);
    writeToScreen("<span class='sendmsg'>Sent message: " + message + "</span>");
}

```

Figure 8 Example of Using JavaScript WebSocket API

When a HTML client is used that makes calls to the WebSocket API of this type, connections can be opened and messages sent back and forth between the peers.

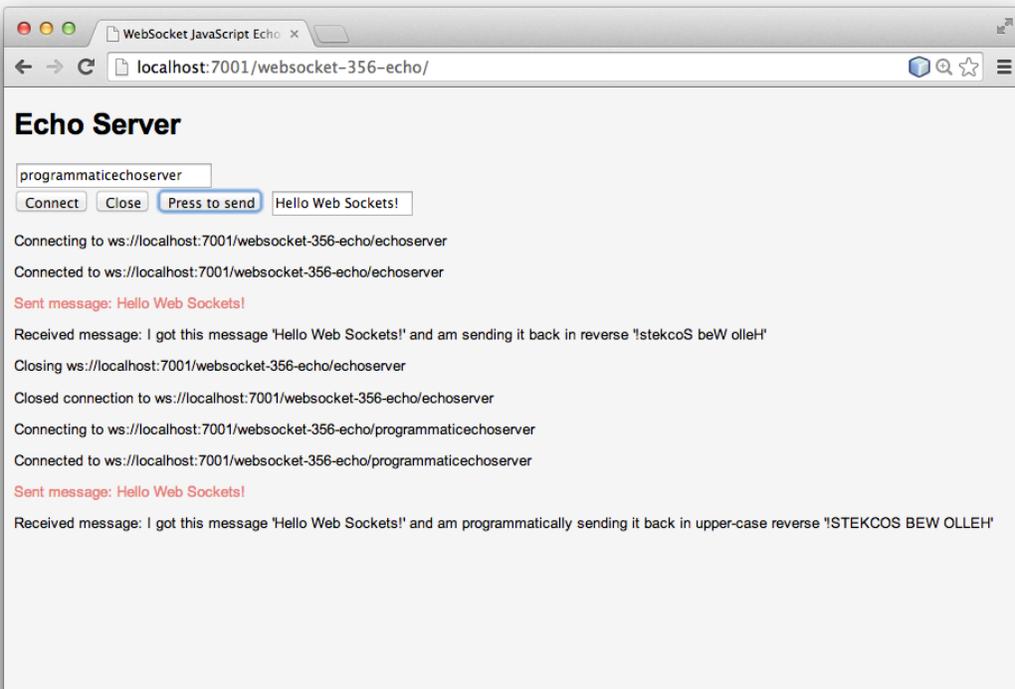


Figure 9 HTML WebSocket Client

With the development of the Java API for WebSocket Protocol 1.0 specification, a level of importance was also placed on providing support for programming WebSocket clients directly in Java. To that end the JSR-356 specification, in addition to providing a server-side programming model, also addresses the area of creating the client-side of a WebSocket application. In a cleverly done way, the specification doesn't introduce any significant differences in developing the different sides of connection other than generally considering that a `ServerEndpoint` is a WebSocket endpoint that is ready and waiting for connection requests and which can host multiple connections at a time; whereas a `ClientEndpoint` will create connections to its peer and connect to at most one peer at a time.

A `ClientEndpoint` is subject to the same lifecycle and event model of the general WebSocket Endpoint and can be defined using the annotation or programmatic model.

```
...
import javax.websocket.WebSocketContainer;
import javax.websocket.ClientEndpoint;
import javax.websocket.ContainerProvider;
import javax.websocket.DeploymentException;
import javax.websocket.OnMessage;
...

@ClientEndpoint
public class EchoClient {

    Session session = null;
    public static void main(String[] args) {
        EchoClient ec = new EchoClient();
        ec.connect("ws://localhost:7001/websocket-356-echo/echoserver");
    }

    @OnOpen
    public void init(Session session) {
        this.session = session;
        try {
            System.out.println("OnOpen: sending message \"Hello!\");
            session.getBasicRemote().sendText("Hello!");
        } catch (IOException ex) {
            Logger.getLogger(EchoClient.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    @OnMessage
    public void message(String msg) {
        System.out.println("OnMessage: " + msg);
    }

    @OnClose
    public void close() {
        System.out.println("@OnClose: Closing time ..");
        System.exit(-1);
    }

    private void connect(String path) {
        WebSocketContainer wsc = ContainerProvider.getWebSocketContainer();
        EchoClient echoclient = new EchoClient();
        try {
            wsc.connectToServer(this, new URI(path));
        } catch (Exception e) {
            System.out.println("Error Connecting: " + e.getMessage());
        }
    }
}
```

Figure 10 Simple Example of a Java WebSocket Client

When the client executes and connects to its server peer, it adheres to the lifecycle events for Endpoints, invoking the annotated methods as appropriate when interacting with its server peer.

```
[INFO] --- exec-maven-plugin:1.2:java (default-cli) @ websocket-356-echo ---
OnOpen: sending message "Hello!"
OnMessage: I got this message 'Hello!' and am sending it back in reverse '!olleH'
```

Figure 11 WebSocket Client Execution

Working with the WebSocket Protocol in WebLogic Server 12.1.3

With the advent of JSR-353 and the Java API for WebSocket 1.0 specification, the previous WebSocket implementation provided in Oracle WebLogic Server 12.1.2 that supported the WebSocket Protocol (RFC 6445) with a WebLogic Server specific API has been deprecated in this release. All new development activity should be focused on the Java standard API – `javax.websocket.*`.

The WebSocket implementation in Oracle WebLogic Server 12.1.3 is based on JSR-353 Java API for WebSocket 1.0 specification, using the Tyrus reference implementation as the engine. Through integration with WebLogic networking and threading services, high levels of performance and scalability have been measured in internal benchmarks, outperforming the previous implementation across the set of metrics used when the same benchmark test was performed.

WebSocket Client Emulation

With the WebSocket Protocol expected to provide a cornerstone of HTML5 by enabling the server push of data to browser clients, its use will become a common element of the many modern web applications deployed using WebLogic Server. However there is also expected to be some situations where the use of WebSocket connections may be problematic because either there are network intermediaries such as firewalls and proxies that do not allow WebSocket connections, or because the browser platform being used does not support WebSockets. In which case developers are forced to detect whether their particular deployment environment will support WebSocket connections, and if not, employ some other HTTP-based technique such as long polling to emulate server push operations. This kind of accommodation of differences in environment and browser support for WebSockets leads to additional complexity for developers in selecting and programming for such fallback situations.

To support this use case, Oracle WebLogic Server 12.1.3 provides a WebSocket Fallback feature that enables the use of WebSocket protocol by developers on both the server side via JSR-356 and the client side via the WebSocket JavaScript API, but where the actual transport mechanism used to send WebSocket messages may be transparently negotiated down from using native WebSocket connection to using a HTTP based long-polling mechanism instead. This is transparent to the developer, who is able to build applications that utilize the features of the WebSocket protocol and related programming models, while removing the need to provide explicit solutions that handle situations where WebSocket connections can not be regularly or reliably established.

The WebSocket Fallback feature consists of two components: a client side JavaScript library called `orasocket.js`, which handles the client side interactions of the WebSocket protocol over a HTTP long-polling based connection and a server side adapter that is integrated into WebLogic Server to handle the conversion of HTTP based data frames and routes them into the WebSocket engine as standard WebSocket messages. To enable the WebSocket Fallback feature the `orasocket.min.js` JavaScript library needs to be included in application and included in the HTML5 client page so that it is loaded. Secondly the web application containing the WebSocket application needs to be provided with a context parameter that enables the use of the fallback mode if needed.

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd">
  <context-param>
    <description>Enable fallback mechanism</description>
    <param-name>com.oracle.tyrus.fallback.enabled</param-name>
    <param-value>true</param-value>
  </context-param>
</web-app>
```

Figure 12 Enabling WebSocket Fallback in web.xml

Java API for JSON Processing 1.0: Simplifying JSON Parsing and Processing

Oracle WebLogic Server 12.1.3 provides full support for the Java API for JSON Processing 1.0 (JSR 353) specification through the inclusion of the JSR-353 reference implementation. The JSON Processing API and implementation is available to all applications deployed to a WebLogic Server instance.

JSON (JavaScript Object Notation) is a lightweight data-interchange format that is widely used as a common format to serialize and de-serialize data in applications that communicate with each other over the Internet. These applications are often created using different programming languages and run in very different environments. JSON is suited to this scenario because it is an open standard, it is easy to read and write, and it is more compact than other representations. RESTful web services typically make extensive use of JSON as the format for the data inside requests and responses, with the JSON representations usually being more compact than the counterpart XML representations since JSON does not have closing tags.

The Java API for JSON Processing provides a convenient way to process (parse, generate, transform, and query) JSON text. For generating and parsing JSON data, there are two programming models, which are similar to those used for XML documents.

- » The Object Model API: the object model API creates a tree that represents the JSON data in memory. The tree can then be navigated, analyzed, or modified. This approach is the most flexible and allows for processing that requires access to the complete contents of the tree. However, it is often slower than the streaming model and requires more memory. The object model generates JSON output by navigating the entire tree at once.
- » The Streaming API: the streaming API uses an event-based parser that reads JSON data one element at a time. The parser generates events and stops for processing when an object or an array begins or ends, when it finds a key, or when it finds a value. Each element can be processed or discarded by the application code, and then the parser proceeds to the next event. This approach is adequate for local processing, in which the processing of an element does not require information from the rest of the data. The streaming model generates JSON output to a given stream by making a function call with one element at a time.

The Object Model API

The object model API is a high-level API that provides immutable object models for JSON object and array structures. These JSON structures are represented as object models using the Java types `JsonObject` and `JsonArray`. The interface `javax.json.JsonObject` provides a `Map` view to access the unordered collection of zero or more name/value pairs from the model. Similarly, the interface `JsonArray` provides a `List` view to access the ordered sequence of zero or more values from the model.

The object model API uses builder patterns to create these object models. The classes `JsonObjectBuilder` and `JsonArrayBuilder` provide methods to create models of type `JsonObject` and `JsonArray` respectively. These object models can also be created from an input source using the class `JsonReader`. Similarly, these object models can be written to an output source using the class `JsonWriter`.

The Streaming API

The streaming API consists of the interfaces `JsonParser` and `JsonGenerator`. The interface `JsonParser` contains methods to parse JSON in a streaming way. The interface `JsonGenerator` contains methods to write JSON to an output source in a streaming way.

`JsonParser` provides forward, read-only access to JSON data using the pull parsing programming model. In this model the application code controls the thread and calls methods in the parser interface to move the parser forward or to obtain JSON data from the current state of the parser. `JsonGenerator` provides methods to write JSON to an output source. The generator writes name/value pairs in JSON objects and values in JSON arrays.

The streaming API is a low-level API designed to process large amounts of JSON data efficiently.

Working with JSON in Oracle WebLogic Server 12.1.3

Within the context of Oracle WebLogic Server 12.1.3 the Java API for JSON Parsing provides a standard, simple and convenient API for working with JSON payloads. This has specific applicability for WebSocket enabled applications where the exchange of messages with browser based WebSocket clients typically makes use of JSON.

```
import javax.json.Json;
import javax.json.JsonObject;
import javax.json.JsonReader;
...

public class AuctionMessageDecoder implements Decoder.Text<AuctionMessage> {

    public AuctionMessage decode(String s) {
        JsonReader jsonReader = Json.createReader(new StringReader(s));
        JsonObject json = jsonReader.readObject();
        return new AuctionMessage(json.getString("type"),
            json.getString("communicationId"), json.getString("data"));
    }
    ...
}
```

Figure 13 WebSocket Decoder Example using Java API for JSON Processing

The counterpart `Encoder` component is responsible for converting an `AuctionMessage` objects into a form that can be transmitted as a WebSocket message. In this case, JSON is being used as the payload format.

```

import javax.json.Json;
import javax.json.JsonArrayBuilder;
import javax.json.JsonObject;
...

public class AuctionMessageEncoder implements Encoder.Text<AuctionMessage> {

    public String encode(AuctionMessage object) throws EncodeException {
        JsonObject json = null;
        switch (object.getType()) {
            case AuctionMessage.AUCTION_LIST_RESPONSE:
                JsonArrayBuilder arrayBuilder = Json.createArrayBuilder();
                ArrayList<String> nameList = (ArrayList<String>) object.getData();
                for (int i = 0; i < nameList.size(); i += 2) {
                    arrayBuilder.add(Json.createObjectBuilder()
                        .add("id", nameList.get(i))
                        .add("name", nameList.get(i + 1)));
                }
                json = Json.createObjectBuilder()
                    .add("type", object.getType())
                    .add("communicationId", object.getCommunicationId())
                    .add("data", arrayBuilder.build());
                break;
            case AuctionMessage.LOGIN_RESPONSE:
                json = Json.createObjectBuilder()
                    .add("type", object.getType())
                    .add("communicationId", object.getCommunicationId())
                    .add("data", Json.createObjectBuilder()
                        .add("name", ((AuctionItem) object.getData()).getName())
                        .add("description", ((AuctionItem) object.getData()).getDescription())
                        .add("price", ((AuctionItem) object.getData()).getPrice())
                        .add("bidTimeoutS", ((AuctionItem) object.getData()).getBidTimeoutS()))
                    .build();
                break;
            default:
                json = Json.createObjectBuilder()
                    .add("type", object.getType())
                    .add("communicationId", object.getCommunicationId())
                    .add("data", (String) object.getData()).build();
                break;
        }
        return json.toString();
    }
}

```

Figure 14 WebSocket Encoder Example using Java API for JSON Processing

While the inclusion of the support for working with JSON with the Java API for JSON specification provides developers with a standard and relatively simple API for working with JSON payloads, another approach for working with JSON is to utilize the Oracle TopLink MOXy feature, which in conjunction specifying declarative, standard JAXB annotations on the classes being used for messages, can automatically transform Java objects to and from a JSON representation without the need to write manual parsing and generating code.

For more information, see the blog entry by Blaise Doughan at <http://blog.bdoughan.com/2013/07/eclipselink-moxy-and-java-api-for-json.html>

JAX-RS 2.0: Java API for RESTful Web Services

With the expansion in both the ease of access to and corresponding use of mobile applications that provide users with access to a vast array of information from sources such as social media posts, photographs, news stories, sports scores, stock prices, currency valuations, online gaming and the like, the interest in providing remote APIs for applications and services is following suit. Not unsurprisingly, given the communication channel is the Internet, the growth of web services that follow the REST architectural style.



RESTful web services are services that are built according to REST principles and are designed to work well on the Internet. RESTful web services are those that conform to architectural style constraints such as stating that resources must be fully addressable via URIs and interacted with through a uniform interface. RESTful web services are (generally) built on the HTTP protocol and implement operations that map to the common HTTP methods, such as GET, POST, PUT, and DELETE to create, retrieve, update, and delete resources, respectively. With this HTTP centric approach and the ensuing stateless interaction model, combined with the data format independence, its clear why REST based web services have become a widely adopted approach

The Java API for RESTful Web Services specification provides a standard based approach to developing RESTful web services for the Java platform.

Oracle WebLogic Server 12.1.3 provides full support for JAX-RS 1.1 as required by the Java EE 6 specification. The JAX-RS 1.1 specification offers developers the ability to easily create REST web services using POJOs and EJB Stateless Session Beans through a decoration based approach, specifying annotations such as `@Path`, `@GET`, `@POST` on classes which become Resources at runtime. Integration with the Java EE platform enables applications containing these Resources to be deployed and exposed as REST web services for clients to call and make use of. The JAX-RS 1.1 specification provides support for also defining how relevant information is automatically extracted from HTTP requests and passed into the Resource through using annotations such as `@PathParam`, `@QueryParam` and `@FormParam`, as well as prescribing what data formats (MediaTypes) the Resource will produce and consume using `@Produces` and `@Consumes`.

Oracle WebLogic Server 12.1.3 further enhances its support for REST based web services by providing developers with the ability to use the latest JAX-RS 2.0 specification in preference to the default JAX-RS 1.1 implementation. This is provided through the introduction of shared-library that can be deployed and used by deployed applications.

The `$ORACLE_HOME/wlserver/common/deployable-libraries/jax-rs-2.0.war` – which contains the JAX-RS 2.0 API library, a Jersey 2.x implementation and a number of other Jersey Features – can be deployed to a WebLogic Server environment and referenced by applications that wish to make use of it. This enables developers to use latest features for REST web services as defined in JAX-RS 2.0 in applications that will deploy and run on Oracle WebLogic Server.

Client API

One of the most significant new features in the JAX-RS 2.0 specification is the JAX-RS Client API, which is a fluent Java based API for communication with REST web services. This new standard Client API is designed to make it easy and straightforward to consume a web service exposed over the HTTP protocol and enables developers to concisely and efficiently implement client-side solutions that are now portable across vendors and environments.

The JAX-RS Client API can be used to consume any web service exposed through the HTTP protocol and is not restricted to services implemented using JAX-RS. Developers familiar with JAX-RS will find the Client API to be largely complementary to the services they expose, especially if the Client API is utilized by those services themselves, or to test those services.

One of the direct benefits of the Client API is that it allows developers to design and work at the level of the Resources they are having interactions with, instead of needing to work at the level of the HTTP protocol itself and constructing and parsing the low level interactions of HTTP needed to work with the Resource.

For example, with a low-level HTTP client library, sending a POST request with a bunch of typed HTML form parameters and receiving a response de-serialized into a JAXB bean is not straightforward at all. With the new JAX-RS Client API supported by Jersey this task is very easy ... If the code had to be written using `URLConnection`, the developer would have to write custom code to serialize the form data that are sent within the POST request and de-serialize the response input stream into a JAXB bean.

Jersey 2.5 User Guide

The Client API is designed to be fluent, with method invocations chained together to configure and submit a request to a REST resource in only a few lines of code. The Client API supports the full set of HTTP requests to allow clients to interact with REST web services to get, put, post and delete resources. Adding properties, parameters and objects for the call is done using the same style of fluent API. For example, to obtain a random quote from a REST service takes only several lines of code.

```
Client client = ClientBuilder.newClient();
String quote = client.target("http://localhost:7001/jaxrs20-client-server/api/quotemaker")
    .request(MediaType.TEXT_PLAIN)
    .get(String.class);
client.close();
```

Figure 15 Client API Example

The Client API can be used to interact with any REST web service and is not constrained to working with JAX-RS based web services. The Client API can be used to interact with REST web services from any Java client, be it a command line client, a Java EE component such as a Servlet or EJB or even a JAX-RS web service itself, providing those components with an easy and portable solution for using REST web services.

Interceptors and Filters

To support requirements developers have which involve participating in the actual interaction sequence between a request and a response for a JAX-RS Resource, the concepts of Filters and Interceptors have been added to the JAX-RS 2.0 specification.

Filters can modify inbound and outbound requests and responses including modification of headers, entity and other request and response parameters.

Filters can be used when you want to modify any request or response parameters like headers. For example you would like to add a response header "X-Powered-By" to each generated response. Instead of adding this header in each resource method you would use a response filter to add this header.

Jersey User Guide, Filters and Interceptors

An example of a Filter that logs and records every request and response that is made for a Resource is provided with the sample set in Oracle WebLogic Server 12.1.3.

```

import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.container.ContainerResponseContext;
import javax.ws.rs.container.ContainerResponseFilter;
import javax.ws.rs.ext.Provider;

@Provider
public class TrafficLogger implements ContainerRequestFilter, ContainerResponseFilter {
    private static final Logger LOGGER = Logger.getLogger(TrafficLogger.class.getName());

    @Override
    public void filter(ContainerRequestContext req) throws IOException {
        LOGGER.log(Level.INFO, "Request, from: {0}, by: {1}, with: {2}.",
            new Object[] { req.getUriInfo().getBaseUrl(), req.getMethod(),
                req.getRequest().toString() });
    }
    @Override
    public void filter(ContainerRequestContext req, ContainerResponseContext res) throws IOException {
        LOGGER.log(Level.INFO, "Response, to: {0}, status: {1}, with: {2}",
            new Object[] { req.getUriInfo().getBaseUrl(), resContext.getStatus(),
                res.getHeaders().toString() });
    }
}

```

Figure 16 JAX-RS Traffic Logging Filter

Interceptors are different from Filters and are designed to enable the modification of entities through the manipulation of entity input and output streams. This is accomplished using implementations of the `ReaderInterceptor` and/or `WriterInterceptor` interfaces to perform the necessary changes to entities.

An example of an Interceptor shipped with the sample set from Oracle WebLogic Server 12.1.3 demonstrates the lightweight encoding of entity streams using an XOR encoding mechanism so they are not readily visible to 3rd parties, performed using a class that implements both the `ReaderInterceptor` and `WriterInterceptor` interfaces.

```

import java.io.*;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.ext.Provider;
import javax.ws.rs.ext.ReaderInterceptor;
import javax.ws.rs.ext.ReaderInterceptorContext;
import javax.ws.rs.ext.WriterInterceptor;
import javax.ws.rs.ext.WriterInterceptorContext;

@Provider
public class DataEncryptionInterceptor implements ReaderInterceptor, WriterInterceptor {

    public static final int KEY = 228;

    @Override
    public Object aroundReadFrom(ReaderInterceptorContext rContext)
        throws IOException, WebApplicationException {
        byte[] data = fetchBytes(rContext.getInputStream());
        // Decryption
        crypt(data, 0, data.length);
        ByteArrayInputStream buffer = new ByteArrayInputStream(data);
        rContext.setInputStream(buffer);
        return rContext.proceed();
    }

    @Override
    public void aroundWriteTo(WriterInterceptorContext wContext)
        throws IOException, WebApplicationException {
        OutputStream currentOutput = wContext.getOutputStream();
        // Encryption
        ByteArrayOutputStream nextOutput = new ByteArrayOutputStream() {

            @Override
            public synchronized void write(int b) {
                super.write(b ^ KEY);
            }

            @Override
            public synchronized void write(byte[] b, int off, int len) {
                crypt(b, off, len);
                super.write(b, off, len);
            }

            @Override
            public void write(byte[] b) throws IOException {
                crypt(b, 0, b.length);
                super.write(b);
            }

        };
        wContext.setOutputStream(nextOutput);
        wContext.proceed();
        currentOutput.write(nextOutput.toByteArray());
    }

    private void crypt(byte[] data, int off, int len) {
        for (int i = 0; i < len; i++) {
            data[off + i] = (byte) (data[off + i] ^ KEY);
        }
    }

    private byte[] fetchBytes(InputStream is) throws IOException {
        ByteArrayOutputStream buffer = new ByteArrayOutputStream();
        int read = 0;
        byte[] data = new byte[1024];
        while ((read = is.read(data, 0, data.length)) != -1) {
            buffer.write(data, 0, read);
        }
        buffer.flush();
        return buffer.toByteArray();
    }
}

```

Figure 17 JAX-RS Interceptor Example

Asynchronous Support

With JAX-RS 1.1 the interaction model between the client and a server was based on a synchronous model – as the client makes a request to the server, the server processes the request with the target Resource using the same thread with which it received the initial connection, ultimately releasing it once the response is complete and has been written.

In JAX-RS 2.0 the concept of an asynchronous interaction has been added which allows a Resource to suspend itself, which has the effect of disassociating the client connection handling from the Resource as it is executed. For developers familiar with the Servlet 3.0 specification, the asynchronous model supported JAX-RS 2.0 will appear very familiar.

Asynchronous processing is particularly relevant for situations where Resource requests occur that are known to take some non-insignificant amount of time to produce a result, or for this situations in which a response will always be produced some time into the future. For these cases, the use of asynchronous programming can help provide improved scalability and overall throughput of the system by freeing up threads to handle incoming requests and allowing Resources to complete their work independently.

```
@Path("/chat")
@Produces(MediaType.TEXT_PLAIN)
@Consumes(MediaType.TEXT_PLAIN)
public class AsyncChatResource {
    private static final Logger LOGGER = Logger.getLogger(AsyncChatResource.class.getName());
    private static final String POST_NOTIFICATION_RESPONSE = "Message stored.";
    private static final ExecutorService QUEUE_EXECUTOR = Executors.newCachedThreadPool();
    private static final BlockingQueue<String> messages = new ArrayBlockingQueue<String>(5);
    @GET
    public void pickUpMessage(@Suspended final AsyncResponse ar,
        @QueryParam("id") final String messageId) {
        LOGGER.log(Level.INFO,
            "Received GET <{0}> with context {1} on thread {2}.",
            new Object[] { messageId, ar.toString(), Thread.currentThread().getName() });
        QUEUE_EXECUTOR.submit(new Runnable() {
            public void run() {
                try {
                    final String message = messages.take(); // block-wait for message to be available
                    LOGGER.log(Level.INFO,
                        "Resuming GET <{0}> context {1} with a message {2} on thread {3}.",
                        new Object[] { messageId, ar.toString(), message,
                            Thread.currentThread().getName() });
                    ar.resume(message); // resume and send message to client
                } catch (InterruptedException ex) {
                    LOGGER.log(Level.SEVERE,
                        "Waiting for a message pick-up interrupted, cancelling " + ar.toString(), ex);
                    ar.cancel(); // close the open connection
                }
            }
        });
    }
}
```

Figure 18 Example Async Method

Working with JAX-RS 2.0 in Oracle WebLogic Server 12.1.3

The JAX-RS 2.0 support within Oracle WebLogic Server 12.1.3 is an optional feature. In order to make use of JAX-RS 2.0, it requires the deployment of a shared-library to the desired WebLogic Server targets, and at an individual application level the availability of the relevant WebLogic deployment descriptor in which a reference is made to include the shared-library.

The shared-library that provides the JAX-RS 2.0 support for applications to use is supplied as the `$ORACLE_HOME/wlserver/common/deployable-libraries/jax-rs-2.0.war`. This library can be deployed using any of

the available deployment mechanisms such as the Administration Console, the command line utility, the WebLogic Maven Plugin or via a WLST script.

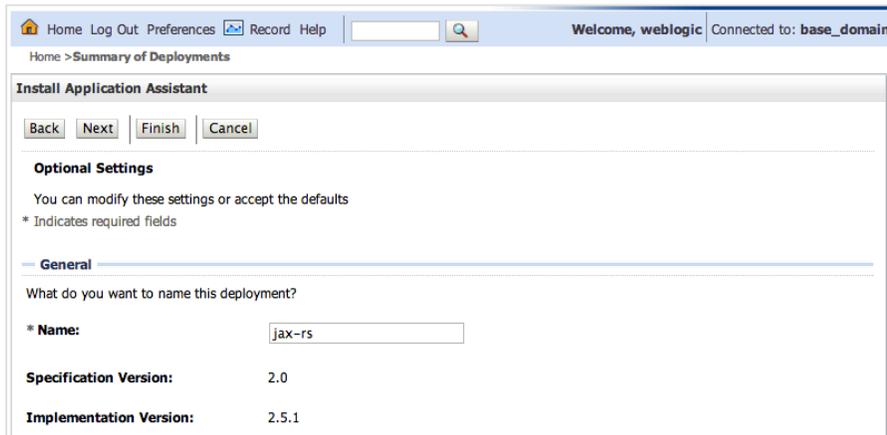


Figure 19 Deploying JAX-RS 2.0 Shared Library

The `jax-rs-2.0.war` shared library itself contains a number of libraries that are used to enable the JAX-RS 2.0 functionality, including the JAX-RS 2.0 API library, the Jersey implementation and some feature extensions, some integration libraries for WebLogic Server and a `weblogic.xml` deployment descriptor that configures the library for use with relevant `filtering-classloader` definitions.

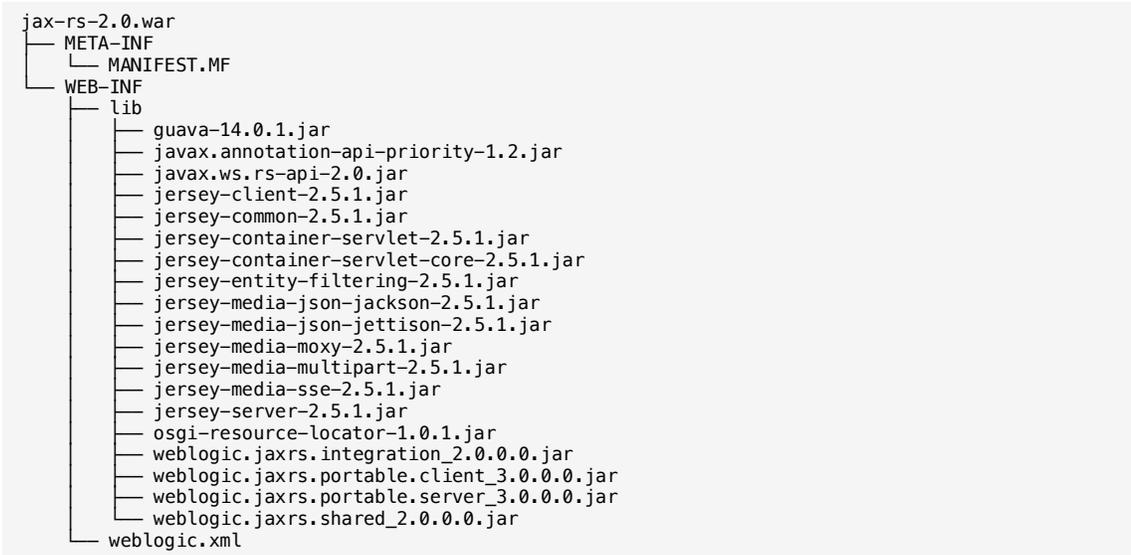


Figure 20 JAX-RS 2.0 Shared Library Contents

To use the JAX-RS 2.0 feature when deployed on WebLogic Server, applications must supply the relevant `weblogic` descriptor file in which a `library-ref` entry is defined for the `jax-rs:2.0` shared-library.

```

<?xml version="1.0" encoding="UTF-8"?>
<weblogic-web-app xmlns=http://xmlns.oracle.com/weblogic/weblogic-web-app
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-web-app
  http://xmlns.oracle.com/weblogic/weblogic-web-app/1.0/weblogic-web-app.xsd">
  <library-ref>
    <library-name>jax-rs</library-name>
    <specification-version>2.0</specification-version>
  </library-ref>
</weblogic-web-app>

```

Figure 21 Example weblogic.xml for using JAX-RS 2.0

Jersey 2 Server-Sent Events

Through the inclusion of the Jersey 2.x implementation to support the JAX-RS 2.0 API, an additional feature for working with mobile and rich clients is available for developers to use. This useful feature provides a JAX-Rs style approach for using the Server-Sent Event model, which allows an application deployed on a server to asynchronously push data from the server to a client. With Server-Sent Events upon a client establishing a connection to a server using a specific MIME type, the server can send data to the client at any time without needing any form of request or subsequent interaction from the client. Whenever a new data event occurs on the server, the server sends the data directly to the client. In effect, the Server-Sent Event model provides a solution for a one-way publish-subscribe model between a client and server.

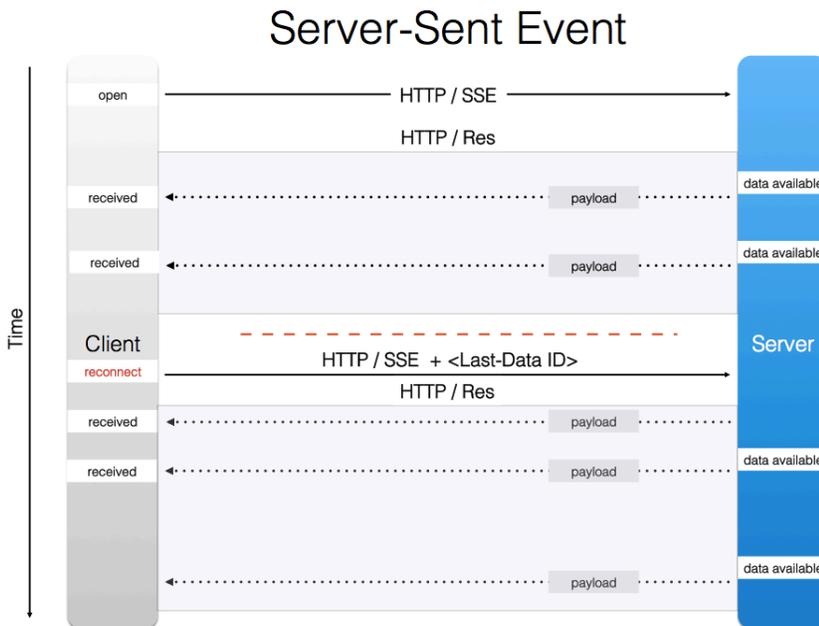


Figure 22 Server-Sent Event Interaction Example

Most modern browsers support the concept of Server-Sent Events through the W3C EventSource JavaScript API. More details of the API are available at <http://www.w3.org/TR/eventsource>.

The typical pattern for the use of Server-Sent Events are where browser clients use the JavaScript EventSource object to open a connection to a server and register callback functions for various events that can occur on the connection. This establishes the client to server connection and from this point onwards the server is able to send



data to the client at any time. When data arrives on the client, the EventSource raises an onmessage event and the callback is able to receive the data. If the event data is in JSON form, the JSON parser available within the browser can quickly encode the payload into its JavaScript form.

The JAX-RS Client API also supports the Server-Sent Event model through the use of the Sse-Feature that can be registered with the ClientBuilder to use the functionality. Events delivered from the server can be obtained by pulling InboundEvents from an EventInput object or by using an EventListener on an EventSource object.

The programming model for Server-Sent Events through the Jersey 2.x implementation supplied with Oracle WebLogic Server 12.1.3 is based on the JAX-RS styled programming model. Annotations are used to expose the Server-Sent Event URI and declare the @Produces type as being the required Server-Sent Event MIME type. A client requesting this resource will be returned an EventOutput Resource, which is treated as a special case by Jersey such that it doesn't close the connection to the client and instead, writes the necessary HTTP headers to return the response the client to keep the connection open and then waits for Server-Sent Event data to send over the connection.

Comparing Server-Sent Events and WebSocket Technologies

Server-Sent Events are part of the [HTML 5 specification](#), which also includes WebSocket technology. Both communication models enable servers to send data to clients in an unsolicited manner. However, server-sent events establish one-way communication from server to clients, while a WebSocket connection provides a bi-directional, full-duplex communication channel between servers and clients, promoting user interaction through two-way communication.

The following key differences exist between WebSocket and Server-Sent Events technologies:

- 1. Server-sent events can only push data to the client, while WebSocket technology can both send and receive data from a client.*
- 2. The simpler server-sent events communication model is better suited for server-only updates, while WebSocket technology requires additional programming for server-only updates.*
- 3. Server-sent events are sent over standard HTTP and therefore do not require any special protocol or server implementation to work. WebSocket technology requires the server to understand the WebSocket protocol to successfully upgrade an HTTP connection to a WebSocket connection.*

Developing and Securing RESTful Web Services for Oracle WebLogic Server, Oracle WebLogic Server 12.1.3

The Jersey 2.x implementation also provides support for the concept of broadcasting, where a server can send messages to multiple clients at a time. In this case, the Resource can store client connection requests in a special class, SseBroadcaster, which can expose a broadcast operation from which it will send the specified message to all currently connected clients stored within itself.

Developing applications that use the Jersey Server-Sent Events feature with Oracle WebLogic Server 12.1.3 is the same process as using any of the JAX-RS 2.0 features, requiring the deployment and referencing of the jax-rs-2.0.war shared-library. The shared-library contains the Jersey specific libraries to support the use of the Server-Sent Event model.

JPA 2.1: Evolution of Java Persistence

The JPA 2.1 specification continues to evolve based on community feedback, providing support for requirements that developers observe when using the very popular API on real projects. Oracle continues to lead the JPA specification and supply the reference implementation through the open-source EclipseLink project.

In the JPA 2.1 release additions were made to provide support for select new capabilities such as:

- » Standard Portable Schema Generation: defines a standardized and portable set of properties to define how schema generation is performed for a persistence-unit, including execution types of DDL actions, script file generation and execution.
- » Stored Procedure Invocation: introduced support for executing database stored procedure calls through the new `StoredProcedureQuery` API and `Named Stored Procedure Queries`, allowing code that executes directly within the database to be called using a formal API.
- » Bulk Updates and Deletes with the Criteria API: provided new `CriteriaUpdate` and `CriteriaDelete` classes to support the execution of bulk update and bulk delete queries using the Criteria API.
- » Dynamic Named Queries: provides support for creating and adding named queries in a dynamic manner to a persistence-unit
- » Converters: adds the ability to control the conversion from an attribute type and value to the corresponding database type and value

```
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.NamedStoredProcedureQuery;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;

@Entity
@NamedStoredProcedureQuery(name="querypersons", procedureName="querypersons",
    resultClasses=Person.class)
@Table(name="JVAEE_PERSON")
public class Person implements Serializable {
    ...
}
```

Figure 23 Stored Procedure Named Query

```

@Stateless
public class AssetService {

    @PersistenceContext(unitName="serviceUnit")
    private EntityManager manager;

    /*
     * Using Criteria Delete API to construct dynamic queries
     */
    public void doDelete(String teamname, int price) {
        CriteriaBuilder builder = manager.getCriteriaBuilder();
        CriteriaDelete<Asset> delete = builder.createCriteriaDelete(Asset.class);
        Root<Asset> asset = delete.from(Asset.class);
        delete.where(builder.equal(asset.get(Asset_.teamname), teamname),
            builder.ge(asset.get(Asset_.price), price));
        Query criteriaUpd = manager.createQuery(delete);
        criteriaUpd.executeUpdate();
    }

    ...
}

```

Figure 24 Bulk Criteria API

Using JPA 2.1 on Oracle WebLogic Server 12.1.3.

Oracle WebLogic Server 12.1.3 supports JPA 2.0 as required by Java EE 6 as its default JPA implementation.

The JPA 2.1 support within Oracle WebLogic Server 12.1.3 is an optional feature that can be explicitly enabled through either the application of a formal Oracle Patch downloaded from the My Oracle Support web site or by manually changing the setting of the CLASSPATH used to launch the server so the JPA 2.1 implementation is accessible.

The files required for supporting JPA 2.1 are included with a default WebLogic Server installation however they are not enabled by default.

The files supplied to support and enable JPA 2.1 are:

- » The **javax.persistence_2.1.jar** file that contains the JPA 2.1 API. This file is supplied in the \$ORACLE_HOME/oracle_common/modules directory.
- » The **com.oracle.weblogic.jpa21support_1.0.0.0_2-1.jar** file that contains the files for integrating and enabling JPA 2.1 support in WebLogic Server. This file is installed in the \$ORACLE_HOME/wlserver/modules directory.

To enable the JPA 2.1 feature, define a `PRE_CLASSPATH` environment variable, which contains these libraries before starting WebLogic Server.

Innovation Features

Oracle TopLink and TopLink Data Services

Oracle TopLink 12.1.2 is fully integrated into the Oracle WebLogic 12.1.2 infrastructure. Oracle TopLink, based on the open source Eclipse Persistence Services Project (EclipseLink), is an advanced object-persistence framework that provides runtime capabilities that reduce development and maintenance efforts and increase enterprise application functionality. TopLink is designed for use in wide range of Java EE and Java SE architectures. TopLink's most popular persistence services include:

- » **Relational:** For persistence of Java objects using the standard Java Persistence API (JPA) specification to a relational database accessed using Java Database Connectivity (JDBC). TopLink provides advanced features for all leading databases with specialized support for Oracle Virtual Private Database, XML DB XMLType, flashback, and stored procedures and functions with Oracle Database.

- » TopLink Grid: For integration with Oracle Coherence that supports scaling JPA applications up onto large clusters and leveraging the processing power of the grid to parallelize queries for Oracle Coherence cached objects.
- » JSON and XML Binding: Oracle TopLink MOxY is a powerful capability provided by TopLink for handling the conversion between Java objects and JSON/XML documents using the Java Architecture for XML Binding (JAXB) approach through annotations on domain classes.

Most significant for development of applications servicing HTML5 or mobile clients, TopLink Data Services support includes the use of RESTful Data Services. TopLink Data Services makes it simple to automatically expose JPA entities with predictable and consistent REST services so that HTML5 clients using JavaScript, or mobile clients can easily retrieve and manipulate data stored in databases via Oracle WebLogic Server using REST. JPA application developers can expose RESTful interfaces to existing entities declaratively, with no additional programming required. Customizable bindings to for both JSON and XML are supported. Powerful capabilities, such as listening on Oracle Database notifications, and updating RESTful interfaces automatically, are also available.

Spring

Oracle WebLogic Server continues to support applications developed using the Spring Framework, either through the embedding and use of Spring as a 3rd party library within the application itself or through the explicit use of the WebLogic Server Spring Integration features that offer additional capabilities to Spring applications when running on WebLogic Server.

Oracle WebLogic Server 12.1.3 has added certification for the using the Spring 4.0.x release with the WebLogic Spring Integration features.

OSGI

The use of OSGI within Oracle WebLogic Server 12.1.3 is supported through the ability to register and instantiate an OSGI framework as part of a WebLogic Server runtime. The Apache Felix OSGI implementation is provided as the default OSGI framework. With an OSGI framework available, OSGI bundles can be deployed, either as server level bundles or as bundles embedded within standard Java EE archives (war, ear, jar) and are made available via JNDI bindings for applications locate and use. The OSGI bundles deployed on WebLogic Server can access and use the OSGI Common Services, including the logging services. The OSGI bundles can also access the WebLogic Server data sources and work managers that are available on the runtime server.

Developer Ease of Use

Zip File Developer Distribution

Recognizing that developers are increasingly looking for small-to-download and quick-start offerings, for the last few releases WebLogic Server has been made available in the form of a zip file. The zip file distribution contains the full featured WebLogic Server product in simple zip file form, removing non-essential components included usually in the full installation such as bundled JDKs, Eclipse based development tooling, native libraries and so forth. This reduction and packaging approach has reduced the size of the download to approximately 180MB, which takes less than 5 minutes to download on a moderately decent network.

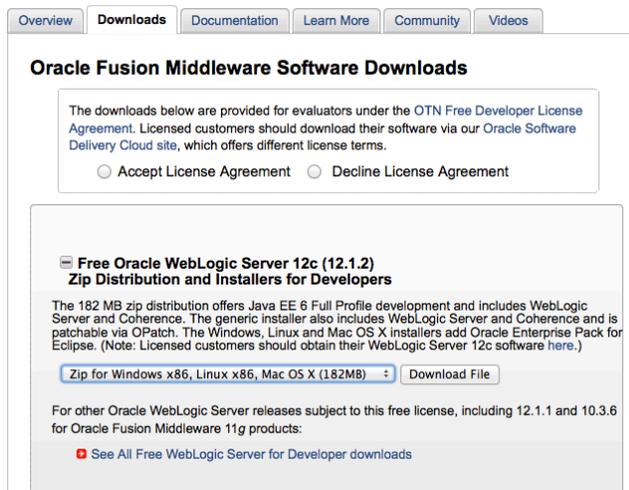


Figure 25 Zip File Download from Oracle Website

Once the zip file has been downloaded from the Oracle website, an instance of WebLogic Server can be quickly created and started by extracting the contents of the zip, running a single configuration script and launching the server to create a new domain.

```
$ unzip wls1213_dev.zip
Archive:  wls1213_dev.zip
creating:  wls12130/
...
creating:  wls12130/oracle_common/modules/
inflating: wls12130/oracle_common/modules/javax.annotation_1.2.0.0_1-1.jar.pack
inflating: wls12130/oracle_common/modules/javax.ejb_3.3.0.jar.pack
inflating: wls12130/oracle_common/modules/javax.jms_1.1.4.jar.pack
inflating: wls12130/oracle_common/modules/javax.jsp_4.0.0.0_2-2.jar.pack
inflating: wls12130/oracle_common/modules/javax.mail_2.0.0.0_1-4-4.jar.pack
...
inflating: wls12130/oracle_common/modules/oracle.jdbc_12.1.0/ojdbc7.jar.pack
inflating: wls12130/oracle_common/modules/oracle.jdbc_12.1.0/ojdbc7_g.jar.pack
inflating: wls12130/oracle_common/modules/oracle.jdbc_12.1.0/ojdbc7dms.jar.pack
...
inflating: wls12130/wlserver/common/bin/commEnv.cmd
inflating: wls12130/wlserver/common/bin/config.cmd
inflating: wls12130/wlserver/common/bin/config_builder.cmd
inflating: wls12130/wlserver/common/bin/pack.cmd
...
```

Figure 26 Unzip of Developer Zip Distribution

The supplied config.sh or config.cmd script is used to prepare the installation for use and to optionally create a new domain as part of the configuration process.

```

wls12130 $ ./configure.sh

*****
WebLogic Server 12g (12.1.3.0) Zip Configuration
MW_HOME: /Users/sbutton/Java/wls12130
JAVA_HOME: /Library/Java/JavaVirtualMachines/jdk1.7.0_45.jdk/Contents/Home
Note: MW_HOME not supplied, default used
*****
Please wait while 740 jars are unpacked ...
...
Unpacking weblogic.server.merged.jar 215 to go
...Unpacking done 215 to go

Your environment has been set.
Configuring WLS...
Do you want to configure a new domain? [y/n]? Y
...
...
<24/06/2014 9:21:08 PM CST> <Info> <Management> <BEA-140013>
</Users/sbutton/Java/wls12130/user_projects/domains/mydomain/config not found>
<24/06/2014 9:21:08 PM CST> <Info> <Security> <BEA-090065> <Getting boot identity from user.>
Enter username to boot WebLogic server:weblogic
Enter password to boot WebLogic server:
For confirmation, please re-enter password required to boot WebLogic server:
<24/06/2014 9:21:17 PM CST> <Info> <Management> <BEA-141254> <Generating new domain directory in
Users/sbutton/Java/wls12130/user_projects/domains/mydomain.>
<24/06/2014 9:21:59 PM CST> <Info> <Management> <BEA-141255> <Domain generation completed in 42,239
milliseconds.>
<24/06/2014 9:21:59 PM CST> <Info> <Management> <BEA-141107> <Version: WebLogic Server 12.1.3.0.0
Wed May 21 18:53:34 PDT 2014 1604337 >
<24/06/2014 9:22:01 PM CST> <Notice> <WebLogicServer> <BEA-000365> <Server state changed to
STARTING.>
...
...
<Jun 24, 2014 9:22:19 PM CST> <Notice> <WebLogicServer> <BEA-000331> <Started the WebLogic Server
Administration Server "myserver" for domain "mydomain" running in development mode.>
<Jun 24, 2014 9:23:19 PM CST> <Notice> <WebLogicServer> <BEA-000360> <The server started in RUNNING
mode.>
<Jun 24, 2014 9:23:19 PM CST> <Notice> <WebLogicServer> <BEA-000365> <Server state changed to
RUNNING.>

```

Figure 27 Execution of Configuration Script for Zip File Distribution

After the execution of the configuration script completes, the product installation is complete. Optionally a domain may have been created and be available and running for immediate use.

A WebLogic Server installation obtained from the zip file distribution contains the full Java EE API and container set; the advanced WebLogic Services including DataSources and JMS features such as distributed destinations and store-and-forward, the client utilities such as WLST and the Maven artifacts and plugins. A significant component of the zip file distribution is the inclusion of the standard WebLogic Server console for efficiently managing and deploying applications to the server from a browser.

The zip file distribution can be used to rapidly create single server domains for development and testing purposes. Importantly, it can also be used to create domains containing clusters with multiple managed servers, including the ability to use the new Dynamic Cluster model to create clusters with configurable number of managed servers.

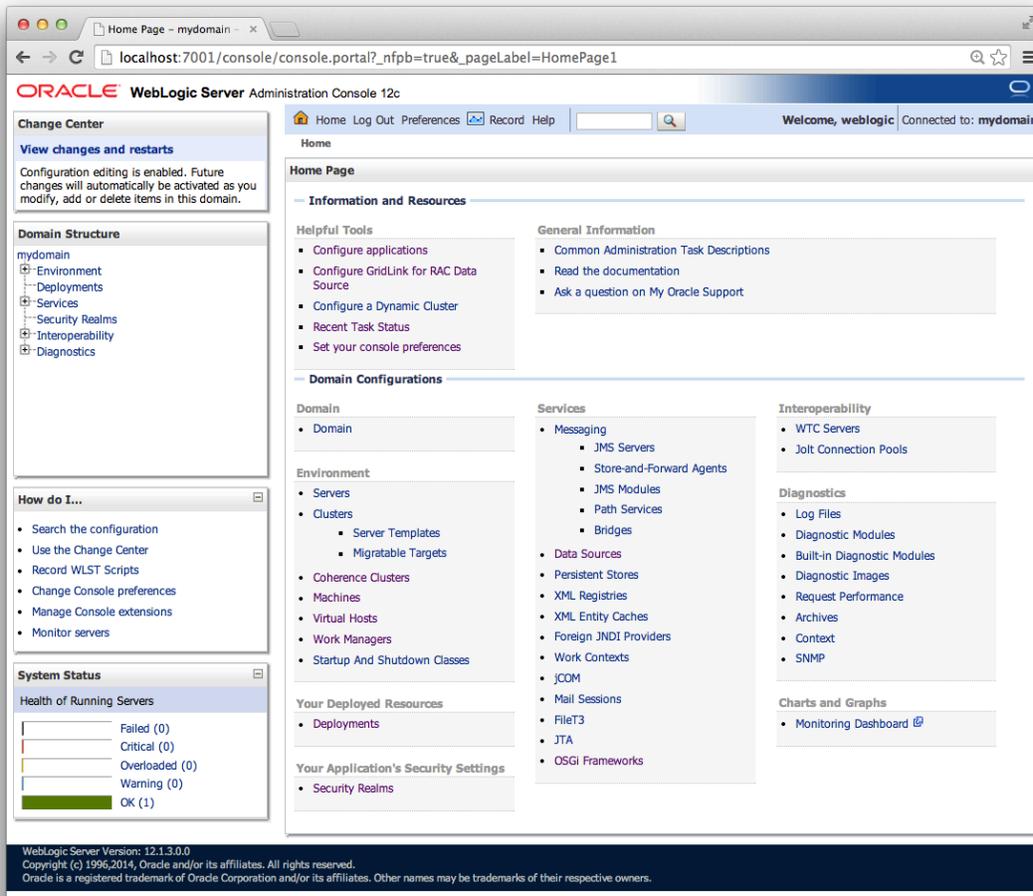


Figure 28 WebLogic Administration Console from Zip File Distribution

For developers continually working in continual build and teardown cycles for testing applications against known environments, an important nature of the zip file distribution is that it can be uninstalled entirely by simply deleting the directory in which it was installed. There are no uninstaller processes that must be run in order to remove registry entries or other meta-data stores typically used by full installer distributions.

Development Tooling Support

Oracle WebLogic Server offers a broad choice of IDEs for application developers to use when building applications.

For Java EE developers that favor the Eclipse integrated development environment, Oracle Enterprise Pack for Eclipse (OEPE) provides all of the tools necessary for Java EE development and deployment on Oracle WebLogic Server. Oracle Enterprise Pack for Eclipse 12.1.3 includes support for full Java EE 6 application development, support for WLST script authoring, and support for development of Oracle Coherence applications, including support for the Grid Archive (GAR) packaging provided with the unique Managed Coherence Servers feature of WebLogic Server.



Oracle NetBeans is an open source IDE implementation that leads the industry in supporting the latest Java EE specifications. It supports both Oracle Glassfish Server, based on the Glassfish open source the Java EE reference implementation, and Oracle WebLogic Server. Developers who have experimented with development of Java EE 6 applications using NetBeans and Glassfish will experience a natural evolution to use of NetBeans for development of Oracle WebLogic Server applications. The latest releases of Oracle NetBeans provide exceptional support for the development of applications using Oracle WebLogic Server 12.1.3 with features that allow NetBeans to start, stop and debug WebLogic Server instances on which to deploy and test applications. Its deploy-on-save feature provides a near seamless development and test cycle where changes made to source code are automatically redeployed on the server and become available for immediate testing. All of this when combined with its extensive support for modern application programming models, including a wide array of HTML5 and JavaScript oriented frameworks, makes Oracle NetBeans a powerful development tool.

Oracle JDeveloper 12.1.3 has been updated to support development of Oracle WebLogic Server 12.1.3 applications. It offers a complete development environment with code editing, testing, debugging, profiling and support the Oracle Application Development Framework (ADF) for rapidly building and deploying enterprise grade applications on Oracle WebLogic Server.

Classloading Analysis and Configuration

Classloading is a complex, and often misunderstood area of application servers. Fortunately, Oracle WebLogic Server provides several mechanisms that simplify configuration of application classloading.

First, Oracle WebLogic Server provides a way of sharing libraries across applications, which simplifies the deployment of such applications. This is useful in cases where the libraries themselves evolve at a different rate than the applications. It also removes the need to deploy the libraries with each application.

Next, Oracle WebLogic Server provides for application-level libraries. These libraries are deployed with each application and loaded through the standard Java EE classloading hierarchy. This means that these libraries are not shared by other applications deployed on the server - each application receives its own copy of the library. This is useful if applications deployed on the same server need to use different versions of the same library.

Oracle WebLogic Server has also simplified classloading through support of a filtering classloader. This classloader does not load classes itself, but prevents classes from being loaded by the parent classloader. The filtering classloader allows applications to override system level classes, which is difficult to accomplish with other application servers. This capability is important for use cases where open source software components used in applications may conflict with different versions of those software components embedded in Oracle WebLogic Server. For instance, your application may need to make use of a different version of Xerces, Spring, Commons-Logging, Log4j and so forth. To accomplish this the filtering classloader would be configured to filter those classes from the system classpath. Instead those classes would be bundled and loaded from the application libraries.

Oracle WebLogic Server provides a Web-based Classloader Analysis Tool for analyzing and resolving classloading issues. In cases like that described above, when open source software used in applications and in Oracle WebLogic Server may conflict, the tool helps to detect such class conflicts, and proposes filtering classloader configurations to resolve them. We recommend that application development projects using open source software libraries use this tool in their development processes for early detection and resolution of classloading issues.

This tool should also be used when upgrading applications, or upgrading existing applications to newer versions of Oracle WebLogic Server, if these applications use open source software. In such upgrade cases, open source software versions used either in the server or in applications may themselves be upgraded. This may cause versions of open source software used in applications to diverge from the versions used in the server, potentially



creating class conflict issues that did not exist before the upgrade. Use of the Classloader Analysis Tool will help to identify and resolve such issues during the upgrade process.

Oracle Apache Maven Support

Oracle WebLogic Server 12.1.3 contains significant functionality to support its use with Apache Maven based projects. The capabilities in this release cover the following areas when working with maven:

- » WebLogic Maven Plugin – an implementation of a maven plugin that performs tasks with WebLogic Server, ranging from installation and domain configuration; start and stop of server instances; execution of WLST scripts to create and configure resources; deployment operations to deploy and manage applications; and application development tasks to help construct web services and pre-compile application archives for distribution.
- » Oracle/WebLogic Maven Artifacts – a set of Oracle defined maven artifacts that can be installed into a maven repository that represent the common APIs and libraries used in development from a WebLogic Server installation.
- » WebLogic Maven Archetypes – a set of maven archetypes that produce several different types of common Java EE projects such as JSF, CDI and MDB applications, complete with configure pom.xml to use the WebLogic artifacts and the WebLogic Maven Plugin.
- » Oracle Maven Synchronization Plugin – a unique solution for populating maven repositories with artifacts from a WebLogic Server installation using an Oracle defined coordinate set.

Overview of Working with Oracle Maven Support

The sequence of activities when working with WebLogic Server and Maven is to install the Oracle Maven Synchronization plugin into the Maven repository being used by the development team. The Oracle Maven Artifacts are installed into a maven repository using the Oracle Maven Synchronization plugin. With the Oracle artifacts installed in the maven repository, the development project can make use of them by declaring dependencies on the relevant artifact containing the required API or implementation library and making use of the various Oracle Maven Plugins to perform specific operations by declaring them as plugins in the relevant project POM file. The Oracle Maven Plugins perform a range of helpful development related tasks when working with WebLogic Server such as installing the product; creating new domains and starting servers; creating required resources such as data sources, JMS destinations and so forth; and performing deployment operations to test the application. The Coherence Maven Plugin provides assistance with packaging operations for Grid Archives (GARs) such as generating the required deployment descriptors, generating POF (Portable Object Format) mapping files for the domain classes and packaging the GAR file ready for deployment.

For developers working on brand new projects, the Oracle Maven Archetypes may be used to create projects containing relevant resources, deployment descriptors, working example code and an appropriately configured Maven POM to package and deploy the project. Archetypes exist for several different types of WebLogic Server projects as well as a Coherence GAR project.

Oracle Maven Artifacts

To support its dependency management mechanism, maven uses the concept of a repository for storing and retrieving artifacts that a project requires. An artifact is a maven packaging construct that combines a standard archive (containing classes and resources) with a set of meta-data (known as coordinates) that ultimately defines its namespace in the Maven repository by which it can be referenced and retrieved. Artifacts are created in a maven repository by performing an install operation, which takes a specified archive file, some meta-data that names and versions the artifact and stores it in the repository. With the artifact in the repository, it can then be accessed by projects through the maven dependency management mechanism.



To embrace the dependency management model, Oracle WebLogic Server 12.1.3 provides a set of maven POM files for a common set of APIs, libraries, client utilities and maven plugins from the WebLogic Server, TopLink and Coherence components that enable them to be installed as artifacts in a maven repository.

Furthermore, given that there are many artifacts available in a WebLogic Server installation, manually performing the task of individually installing each artifact into the repository would be a cumbersome and potentially error prone task. For this reason a specific maven plug-in – oracle-maven-sync – is provided as part of the release, which automates the task of populating a local (or remote) maven repository with artifacts from a given WebLogic Server installation directory.

Oracle Maven Synchronization Plugin

Oracle WebLogic Server 12.1.3 contains a maven plugin that simplifies the process of populating maven repositories with the set of Oracle defined maven artifacts from a specified WebLogic Server installation directory. The synchronization plugin works by searching through the specified directory looking for two types of files it needs to construct an artifact: a standard maven POM file and a corresponding location file. The location file contains a pointer to the actual location of a jar file that corresponds to the POM. The synchronization plugin then performs a maven install operation, using the POM file and the corresponding jar file to create the final artifact in the repository being used.

To use the synchronization plugin, it first must be installed into a maven repository. The installation process for the synchronization plugin uses the maven install goal to manually upload the plugin into the repository. Once installed, the synchronization plugin can be used to install the artifacts from the WebLogic Server installation into the maven repository using its push goal.

```

$ mvn com.oracle.maven:oracle-maven-sync:push -DoracleHome=/Users/sbutton/Oracle/Middleware

[INFO] Scanning for projects...
[INFO] -----
[INFO] Building Maven Stub Project (No POM) 1
[INFO] -----
[INFO] --- oracle-maven-sync:12.1.3-0-0:push (default-cli) @ standalone-pom ---
[INFO] -----
[INFO] ORACLE MAVEN SYNCHRONIZATION PLUGIN - PUSH
[INFO] -----
[INFO] Found 275 location files in /Users/sbutton/Oracle/Middleware/wlserver/plugins/maven
...
[INFO] /Users/sbutton/Oracle/Middleware/wlserver/plugins/maven/com/oracle/weblogic/wls-
api/12.1.3/wls-api.12.1.3.location:
[INFO] com.oracle.weblogic:wls-api:jar:12.1.3-0-0
  Update Time: 20140530044959
  POM File: /Users/sbutton/Oracle/Middleware/wlserver/plugins/maven/com/oracle/weblogic/wls-
api/12.1.3/wls-api.12.1.3.pom
  Real File: /Users/sbutton/Oracle/Middleware/wlserver/server/lib/wls-api.jar
[INFO] Installing /Users/sbutton/Oracle/Middleware/wlserver/server/lib/wls-api.jar to
Users/sbutton/.m2/repository/com/oracle/weblogic/wls-api/12.1.3-0-0/wls-api-12.1.3-0-0.jar
[INFO] Installing /Users/sbutton/Oracle/Middleware/wlserver/plugins/maven/com/oracle/weblogic/wls-
api/12.1.3/wls-api.12.1.3.pom to /Users/sbutton/.m2/repository/com/oracle/weblogic/wls-api/12.1.3-0-
0/wls-api-12.1.3-0-0.pom
...
[INFO]
Users/sbutton/Oracle/Middleware/coherence/plugins/maven/com/oracle/coherence/coherence/12.1.3/cohere
nce.12.1.3.location:
[INFO] com.oracle.coherence:coherence:jar:12.1.3-0-0
  Update Time: 20140515174418
  POM File:
/Users/sbutton/Oracle/Middleware/coherence/plugins/maven/com/oracle/coherence/coherence/12.1.3/coher
ence.12.1.3.pom
  Real File: /Users/sbutton/Oracle/Middleware/coherence/lib/coherence.jar
[INFO] Installing /Users/sbutton/Oracle/Middleware/coherence/lib/coherence.jar to
Users/sbutton/.m2/repository/com/oracle/coherence/coherence/12.1.3-0-0/coherence-12.1.3-0-0.jar
[INFO] Installing
Users/sbutton/Oracle/Middleware/coherence/plugins/maven/com/oracle/coherence/coherence/12.1.3/cohere
nce.12.1.3.pom to /Users/sbutton/.m2/repository/com/oracle/coherence/coherence/12.1.3-0-0/coherence-
12.1.3-0-0.pom
...

```

Figure 29 Installing Oracle Maven Artifacts into a Repository

Oracle WebLogic Maven Plugin

The WebLogic Maven Plugin is a foundational component of the Oracle Maven landscape, enabling WebLogic Server to be integrated into the Maven project lifecycle so that it can be used as a platform on which projects can be tested.

The WebLogic-Maven-Plugin provides a set of goals that allows WebLogic Server to be used within a Maven project for

- » Conducting infrastructure type operations such as installing WebLogic Server versions, creating domains and starting/stopping of servers
- » Executing operational commands such as creation of resources such as JMS Connection Factories and Destinations, DataSources, User and Groups and so forth
- » Deployment related operations that perform the deployment, undeployment and redeployment of applications
- » Development and packaging related operations that cover the pre-compilation of application archives to optimize deployment time and for generating and packaging the specific types of resources required by Web Services

The WebLogic-Maven-Plugin is a standard Maven plugin that uses a set of WebLogic Server libraries – which are declared as dependencies from the Oracle Maven Artifact set – to perform the majority of its goals against local and/or remote WebLogic Server instances using the same configuration information. Through being able to work with local and remote instances the WebLogic Maven Plugin can enable the testing of maven projects against local test installations or against remote targets where tests are conducted against farms of remote WebLogic Server instances. It also makes possible the incorporation of WebLogic Server as a target platform from within Continuous Integration environments where tests are conducted on-demand as project changes are committed to source code repositories.

Installation and Lifecycle Goals

To support situations where servers need to be created on demand on new hardware or virtual machines, or where using new server instances for test suites is necessary in order to ensure tests are executing in a clean environment, the WebLogic Maven Plugin provides support for installing the WebLogic Server product on the target machine. This can be performed using the developer zip file distribution or the generic Java based installer. To use the latter, a response file must be specified that provides the relevant directory details on where the install is to be performed. A useful feature of the install goal is that it allows the installation binary itself to be specified as a local file reference, a HTTP based reference or as maven coordinate to allow it to be pulled out of a maven repository. This latter capability enables WebLogic Server installation binaries to be named and installed into maven repositories for shared common use across development projects. An uninstall goal also exists that supports the de-installation process.

Following the installation of the WebLogic Server product a domain can be created using the create-domain goal. This goal creates a named domain from the default domain template and configures it using the username and password supplied as parameters to the goal. For more complex domain creation requirements, the WLST goal (as described later) can be used to create a domain using the WLST offline goals.

The final goals in this category are the server start and stop goals. These goals can be used to start and stop the AdminServer in a specified domain, allowing WebLogic Server runtime processes to be controlled directly from a maven project.

```
<project>
  ...
  <build>
    <plugins>
      ...
      <plugin>
        <groupId>com.oracle.weblogic</groupId>
        <artifactId>weblogic-maven-plugin</artifactId>
        <version>12.1.3-0-0</version>
        <configuration>
          <user>${oracleUsername}</user>
          <password>${oraclePassword}</password>
          <source>
            ${project.build.directory}/${project.build.finalName}.${project.packaging}
          </source>
          <name>${project.build.finalName}</name>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Figure 30 Minimum POM Definition for WebLogic Maven Plugin

Deployment and Resource Management

The bulk of the WebLogic Maven Plugin goals are oriented around performing deployment and associated resource management operations from a Maven project against a target WebLogic Server domain in order for the project to be tested or even potentially deployed into production for organizations that have embraced Continuous Deployment practices.

The deployment related goals support performing deploy, undeploy and redeployment actions. These goals support the deployment of applications as archives or as exploded directories, with the latter being applicable only to local WebLogic Server installations (or network accessible directory shares). The deployment to remote installations can be controlled through parameters specified on the deploy goal to direct the uploading of the archive to the target server.

For any non-trivial application deployment, the configuration of server resources is a required activity. WebLogic Server has a strong level of scripting capability based around the Jython scripting language which has been used by administrators for many years to create reusable, repeatable scripts that create, configure, test and target server based resources. The WLST environment has full support for working with DataSources and Connection Pools, JMS ConnectionFactories and Destinations, Security related configurations such as users, groups and credential mappings. It also supports creating complete environments consisting of domains, clusters, machines and servers with associated resources. The WebLogic Maven Plugin facilitates the use of the same WLST scripts from within the context of a maven project to allow project related resources to be created, configured and managed as part of the overall maven lifecycle.

With the WebLogic Maven Plugin being a standard maven plugin, its goals can be mapped into the maven lifecycle so they are automatically executed at a specific point in the project execution, ensuring the orderly execution of the goals as required to ensure the required resources are present for the application deployment to successfully complete and tests able to executed.

```
...
  <executions>
    <execution>
      <id>deploy</id>
      <phase>pre-integration-test</phase>
      <goals>
        <goal>deploy</goal>
      </goals>
    </execution>
    <execution>
      <id>cleanup</id>
      <phase>clean</phase>
      <goals>
        <goal>undeploy</goal>
      </goals>
      <configuration>
        <failOnError>false</failOnError>
      </configuration>
    </execution>
  </executions>
...
```

Figure 31 Binding Deployment Goals to Lifecycle

Development and Pre-Compilation

Rounding out the goals provided by the WebLogic Maven Plugin are the Web Services goals that support the execution of common web services development operations that require the generation of resources for use in a development project such as Java skeletons and Web Service clients from WSDL definitions, complimented by a goal that will compile and package a JAX-WS based application.



The WebLogic Server application compiler utility has also been exposed as a goal to support post-packaging operations to be performed on an archive that will pre-compile JSPs and EJBs in the archive, resulting in decreased deployment time for applications that have significant numbers of these types of components.

Oracle Maven Archetypes

Maven Archetypes are a very handy feature maven provides, enabling projects to be created from templates that can represent anything from basic projects with simple configurations and example code through to detailed projects that provide consistent and common foundations for all development projects to be based upon.

Amongst the artifact set provided by the Oracle Maven Artifacts, several archetypes are included. These archetypes can be used to create simple start projects for several common application patterns, which include working code, web pages and a POM file configured with relevant dependencies on the Oracle Maven Artifacts. The POM file is also pre-configured with the WebLogic Maven Plugin to perform any necessary deployment and resource creation operations to allow the project to be deployed and tested.

Continuous Integration Practices

When enterprises develop applications to support their business needs, they typically employ teams of developers who work together, often in small teams, with each team building a part of the application. These parts are then assembled to create the whole application.

Many modern applications are based on modular architectures, which means developers build different aspects of the overall system that are ultimately assembled to fulfil needs of the business application. Some of the types of attributes that make this a popular development approach are:

- » Loose coupling of components of the application, which reduces the impact of change
- » Reuse of services, a long time goal of Information Technology development
- » The flexibility and agility to easily change the application's behaviour as the business need changes

In this new paradigm, many development organizations are also adopting iterative development methodologies to replace the older waterfall-style methodologies. Iterative, agile development methodologies focus on delivering smaller increments of functionality more often than traditional waterfall approaches. Proponents of the new approach claim that the impact is usually less for errors that are found sooner and that the approach is especially suitable to today's environment of constant and rapid change in business requirements.

Many of these techniques also feature the adoption of continuous integration. Organizations have a strong interest in automating their software builds and testing, and continuous integration can help accomplish this.

Continuous integration is a software engineering practice that attempts to improve quality and reduce the time taken to deliver software by applying small and frequent quality control efforts. It is characterized by these key practices:

- » A version control system is used to track changes
- » All developers commit to the main code line, head and trunk, every day
- » The product is built on every commit operation
- » The build must be automated and fast
- » There should be automated deployment to a production-like environment
- » Automated testing should be enabled
- » Results of all builds are published, so that everyone can see if anyone breaks a build
- » Deliverables are easily available for developers, testers, and other invested parties



Through its evolving support for maven and other technologies, Oracle WebLogic Server 12.1.3 provides support for enterprises that adopt continuous integration techniques to develop applications, encompassing:

- » Integration with common version control systems from various development tools, Oracle Enterprise Pack or Eclipse, Oracle NetBeans and Oracle JDeveloper
 - » The ability to build projects from the command line using maven, a build and project management system, so that the build can be scripted and automated
 - » The ability to create new projects from maven archetypes
 - » The ability to download necessary dependencies from maven repositories
 - » The ability to parameterize projects so that builds can be targeted to different environments, such as Test, QA, SIT, and production
 - » The ability to include testing of projects in the maven build life cycle
 - » The ability to populate a maven repository with Oracle-provided dependencies from a WebLogic Server software installation directory
 - » The ability to run maven builds under the control of a continuous integration server like Hudson
- Provision of comprehensive documentation that walks through the process of creating a continuous integration environment to include WebLogic Server

Conclusion

Oracle WebLogic Server 12.1.3 continues to support developers with a proven, reliable Java EE 6 implementation with specific updates to several specifications to support the development of modern web and rich client applications. It also continues its ongoing support for developers through its optimized zip distributions, broad range of support for development tools and its integration into common development lifecycles through its growing set of maven related capabilities.

References

- » Oracle Fusion Middleware 12c (12.1.3) Documentation Library, <http://www.oracle.com/technetwork/middleware/fusion-middleware/documentation/index.html>
- » JSR-339 Java API for RESTful Web Services 2.0, <https://jcp.org/en/jsr/detail?id=339>
- » JSR-338 Java Persistence API 2.1, <https://jcp.org/en/jsr/detail?id=338>
- » JSR-356 Java API for WebSocket 1.0, <https://www.jcp.org/en/jsr/detail?id=356>
- » JSR-353 Java API for JSON Programming 1.0, <https://jcp.org/en/jsr/detail?id=353>
- » Jersey 2.5 User Guide, <https://jersey.java.net/nonav/documentation/2.5>
- » EclipseLink Documentation for JPA 2.1, <http://wiki.eclipse.org/EclipseLink/Release/2.5/JPA21>
- » Java EE 7 Tutorial, <http://docs.oracle.com/javaee/7/tutorial/doc/home.htm>
- » Java WebSocket Programming, Danny Coward, Oracle Press 2013
- » EclipseLink MOxy and the Java API for JSON Processing - Object Model APIs, <http://blog.bdoughan.com/2013/07/eclipselink-moxy-and-java-api-for-json.html>



Oracle Corporation, World Headquarters

500 Oracle Parkway
Redwood Shores, CA 94065, USA

Worldwide Inquiries

Phone: +1.650.506.7000
Fax: +1.650.506.7200



CONNECT WITH US

-  blogs.oracle.com/oracle
-  facebook.com/oracle
-  twitter.com/oracle
-  oracle.com

Hardware and Software, Engineered to Work Together

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. This document is provided for information purposes only, and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document, and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group. 0614

 | Oracle is committed to developing practices and products that help protect the environment