JMS Adapter

Description	JMS Adapter documentation and tutorial.
Id	167
Last modified	Thu Mar 20 23:43:50 EDT 2008
GUID	709c44b3553ae16737b8cd0756136441b32ccc34

JMS Adapter abstracts Java applications from JMS API's. It can be used by Java applications to

- Send JMS messages and receive replies. Java application acts in this case as a JMS service consumer.
- Listen for JMS messages and send replies. In this case Java application acts as a JMS service provider.

Using JMS Adapter Java application can act as service consumer and provider at the same time. Also JMS Adapter itself can be started as a standalone Java applicaition and act as a service orchestrator.

The Hammurapi Group JMS Adapter is a *true* adapter because it adapts to the client, whether the client produces or consumes messages, instead of the client having to adopt adapter-provided API's.

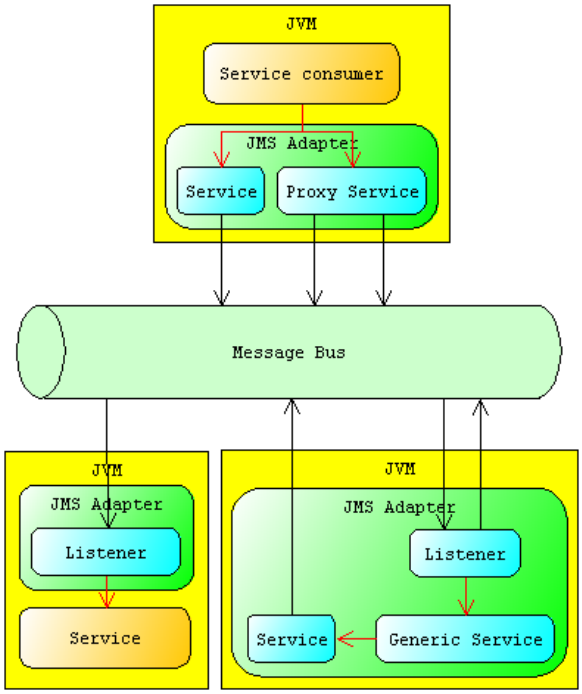

Figure 1. JMS Adapter scenarios.

Features

- Java/JMS binding.
- Three binding components to work with XML JMS messages.
- XSLT transformations of incoming/outcoming messages.
- Connection pooling.
- Session pooling.
- Automatic recovery of failed connections.
- Asynchronous invocations.
- Metrics collection.
- Supports JCA 1.5

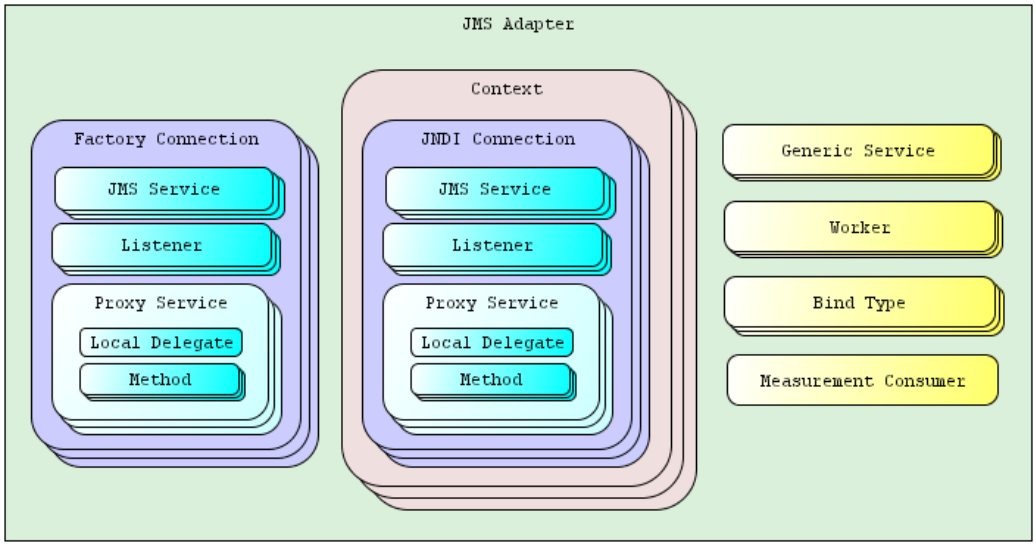

Figure 2. Adapter architecture.

Tutorial

Installation

- Download hgee-2.7.0.2.zip
- Extract files to some directory, say, `C:\JmsTutorial`
- Download and intall Apache ActiveMQ
- Start ActiveMQ and create two queues: `requestQueue` and `replyQueue`.
- Copy `apache-activemq-x.x.x.jar` from ActiveMQ installation directory to `C:\JmsTutorial\lib`.
- Download jms-adapter-tutorial.zip
- Extract files to a temporary directory
- Move `jms-adapter-tutorial.jar` to `C:\JmsTutorial\lib`
- Rename `src` directory to `jms-adapter-tutorial-src`
- Move all files from the temporary directory, including `jms-adapter-tutorial-src`, to `C:\JmsTutorial\lib`
- Set `x` flag on `jms-adapter.sh` on *x platforms.

How to start and stop service or listener

On Windows start

```
jms-adapter.bat
```

On *x systems

```
./jms-adapter.sh ./
```

Use `listener.xml` to start listener and `service.xml` to start service. To stop service or listener simply terminate JVM with Ctrl-C. Adapter's shutdown hook will properly stop the adapter.

Step 1: Listener

Listener adapter configuration is stored in listener.xml. JMS text message with XML payload is converted to a Java object by the simple xml converter. The java object then is processed by `biz.hammurapi.jms.adapter.tutorial.SampleProcessor` Exceptions are also handled by this class.

```
<ns:jms-adapter xmlns:ns="http://www.hammurapi.biz/jms/adapter/definition"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

   <ns:name>Listener</ns:name>
   <ns:description>Sample service</ns:description>
   <ns:connection>
       <ns:name>active-mq</ns:name>
       <ns:description>Active MQ connection</ns:description>
       <ns:reuse-thread-session>true</ns:reuse-thread-session>

       <ns:listener>
           <ns:name>sample-listener</ns:name>
           <ns:description>Sample listener</ns:description>
           <ns:destination>requestQueue</ns:destination>
           <ns:queue-from-session>true</ns:queue-from-session>
           <ns:bind-type>xml-simple</ns:bind-type>
           <ns:processor type="biz.hammurapi.jms.adapter.tutorial.SampleProcessor"/>
```

```
        </ns:listener>

        <ns:factory type="org.apache.activemq.ActiveMQConnectionFactory">
            <ns:property name="brokerURL">tcp://localhost:61616</ns:property>
        </ns:factory>
    </ns:connection>

    <ns:bind-type type="biz.hammurapi.jms.adapter.converters.SimpleXmlConverter">
        <ns:name>xml-simple</ns:name>
        <ns:description>Simple XML converter</ns:description>
    </ns:bind-type>
</ns:jms-adapter>
```

Listing 1. listener.xml

Listing 1 shows XML configuration of JMS listener. This is a very basic configuration with many elements omitted. Please consult `jms-adapter.xsd` in HGee distribution for the full list of supported elements and attributes.

```
package biz.hammurapi.jms.adapter.tutorial;

import biz.hammurapi.jms.adapter.Processor;
import biz.hammurapi.util.ExceptionSink;

/**
 * Receives object and prints it to console
 */
public class SampleProcessor implements Processor, ExceptionSink {

    public Object process(Object obj) {
        System.out.println(obj);
        return null;
    }

    public void consume(Object source, Exception e) {
        e.printStackTrace();
    }
}
```

Listing 2. SampleProcessor.

Start listener and then send messages to the requestQueue from ActiveMQ JMX console. You will see that messages in plain text cause exceptions, XML messages with a single text element and no attributes get successfully processed. If you add attribute type="java.lang.Integer" you will observe an exception if element content is not a number.

Examples

```
Hello!
```

Invalid payload - not XML.

```
<greeting>Hello</greeting>
```

Valid payload - default type is String.

```
<greeting type="java.lang.String">Hello!</greeting>
```

Valid payload - explicit type specification.

```
<number>323</number>
```

Valid payload - String.

```
<number type="java.lang.Integer">323</number>
```

Valid payload - Integer.

```
<number type="java.lang.Integer">Hello!</number>
```

Invalid payload - Hello! cannot be parsed to java.lang.Integer.

Stop listener when you are done with experiments.

Step 2: Service and consumer

In this step we will start an adapter intance with a service which calculates string length. Then this services will be consumed by a Java application through an adapter instance created by the application.

```
<?xml version="1.0" encoding="UTF-8"?>
<ns:jms-adapter
    xmlns:ns="http://www.hammurapi.biz/jms/adapter/definition"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <ns:name>Service container</ns:name>
    <ns:description>Hosts sample service</ns:description>
    <ns:connection>
        <ns:name>active-mq</ns:name>
        <ns:description>Active MQ connection</ns:description>
        <ns:reuse-thread-session>true</ns:reuse-thread-session>

        <ns:listener>
            <ns:name>service</ns:name>
            <ns:description>String length service</ns:description>
            <ns:destination>requestQueue</ns:destination>
```

```
            <ns:reply-destination>replyQueue</ns:reply-destination>
            <ns:queue-from-session>true</ns:queue-from-session>
            <ns:bind-type>xml-simple</ns:bind-type>
            <ns:processor type="biz.hammurapi.jms.adapter.tutorial.ServiceProcessor"/>
        </ns:listener>

        <ns:factory type="org.apache.activemq.ActiveMQConnectionFactory">
            <ns:property name="brokerURL">tcp://localhost:61616</ns:property>
        </ns:factory>
    </ns:connection>

    <ns:bind-type type="biz.hammurapi.jms.adapter.converters.SimpleXmlConverter">
        <ns:name>xml-simple</ns:name>
        <ns:description>Simple XML converter</ns:description>
    </ns:bind-type>

    <ns:measurement-consumer type="biz.hammurapi.metrics.SlicingMeasurementConsumer"/>
</ns:jms-adapter>
```

Listing 3. service.xml

```
package biz.hammurapi.jms.adapter.tutorial;

import biz.hammurapi.jms.adapter.Processor;

/**
 * Receives object and prints it to console. Returns object length if
 * object is string. Throws exception otherwise.
 */
public class ServiceProcessor implements Processor {

    public Object process(Object obj) throws Exception {
        System.out.println(obj);
        if (obj instanceof String) {
            return new Integer(((String) obj).length());
        } else {
            throw new IllegalArgumentException(obj==null ? "Request is null" : "Unexpected request type: "+obj.getClass());
        }
    }

}
```

Listing 4. ServiceProcessor.java

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

<ns:jms-adapter
    xmlns:ns="http://www.hammurapi.biz/jms/adapter/definition"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <ns:name>Service consumer</ns:name>
    <ns:description>Provides access to the sample service</ns:description>

    <ns:connection>
        <ns:name>active-mq</ns:name>
        <ns:description>Active MQ connection</ns:description>
        <ns:reuse-thread-session>true</ns:reuse-thread-session>

        <ns:service>
            <ns:name>string-length</ns:name>
            <ns:description>Invokes the string length service</ns:description>
            <ns:request-destination>requestQueue</ns:request-destination>
            <ns:reply-destination>replyQueue</ns:reply-destination>
            <ns:queue-from-session>true</ns:queue-from-session>
            <ns:bind-type>xml-simple</ns:bind-type>
        </ns:service>

        <ns:proxy-service>
            <ns:name>proxy-string-length</ns:name>
            <ns:description>Invokes string length service through interface.</ns:description>
            <ns:interface>biz.hammurapi.jms.adapter.tutorial.StringLengthService</ns:interface>
            <ns:alias>proxy</ns:alias>
            <ns:method>
                <ns:name>length</ns:name>
                <ns:service>
                    <ns:name>Invocation service</ns:name>
                    <ns:request-destination>requestQueue</ns:request-destination>
                    <ns:queue-from-session>true</ns:queue-from-session>
                    <ns:timeout>3000</ns:timeout>
                    <ns:bind-type>xml-simple</ns:bind-type>
                    <ns:property name="to-xml-style">file:invocation.xsl</ns:property>
                </ns:service>
            </ns:method>
        </ns:proxy-service>

        <ns:factory type="org.apache.activemq.ActiveMQConnectionFactory">
            <ns:property name="brokerURL">tcp://localhost:61616</ns:property>
        </ns:factory>
    </ns:connection>

    <ns:bind-type type="biz.hammurapi.jms.adapter.converters.SimpleXmlConverter">
        <ns:name>xml-simple</ns:name>
        <ns:description>Simple XML converter</ns:description>
    </ns:bind-type>

</ns:jms-adapter>
```

Listing 5. service-consumer.xml

service-consumer.xml contains definitions of two types of services - a simple Service and a Proxy Service.

```
package biz.hammurapi.jms.adapter.tutorial;

import java.io.File;

import org.apache.xmlbeans.XmlObject;
```

```
import biz.hammurapi.jms.adapter.JmsAdapter;
import biz.hammurapi.jms.adapter.JmsService;
import biz.hammurapi.jms.adapter.definition.JmsAdapterDocument;

public class ServiceConsumer {
    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {
        XmlObject document = XmlObject.Factory.parse(new File("service-consumer.xml"));
        if (document instanceof JmsAdapterDocument) {
            biz.hammurapi.jms.adapter.definition.JmsAdapter definition = ((JmsAdapterDocument) document).getJmsAdapter();
            JmsAdapter adapter = new JmsAdapter(definition);
            adapter.start();
            try {
                JmsService service = (JmsService) adapter.get("connections/active-mq/services/string-length");
                System.out.println(service.request("Hello"));
                //System.out.println(service.request(new Integer(385)));
            } finally {
                adapter.stop();
            }
        } else {
            System.err.println("Invalid adapter definition.");
        }
    }
}
```

Listing 6. ServiceConsumer.java

Listing 6 shows to how to invoke our string length service through the simple service.

Proxy services use the same bind types as regular services. In order to do this method invocations are wrapped into instances of Invocation. I our example we use XML payload. Therefore Invocation instance is converted to XML as shown below.

```
<object method-name="length" type="biz.hammurapi.jms.adapter.Invocation">
    <state type="java.util.HashMap">
        <entry>
            <key type="java.lang.String">Value</key>
            <value type="java.lang.String">Hello</value>
        </entry>
    </state>
</object>
```

Listing 7. Invocation.

Our service cannot accept such payload and we have to transform invocation XML to a single string XML. We achieve this by applying a simple stylesheet to the invocation before writing it to a JMS message. Listing 8 shows the stylesheet. Stylesheet name is specified in "to-xml-style" property in Listing 5.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:stylesheet xmlns:xs="http://www.w3.org/1999/XSL/Transform" version="1.0">

    <xs:output method="xml"/>

    <xs:template match="/">
        <xs:for-each select="object/state/entry/value">
            <value type="{@type}"><xs:value-of select="text()"/></value>
        </xs:for-each>
    </xs:template>

</xs:stylesheet>
```

Listing 8. Invocation stylesheet.

```
package biz.hammurapi.jms.adapter.tutorial;

import java.io.File;

import org.apache.xmlbeans.XmlObject;

import biz.hammurapi.jms.adapter.JmsAdapter;
import biz.hammurapi.jms.adapter.ProxyService;
import biz.hammurapi.jms.adapter.definition.JmsAdapterDocument;

public class ProxyServiceConsumer {
    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {
        XmlObject document = XmlObject.Factory.parse(new File("service-consumer.xml"));
        if (document instanceof JmsAdapterDocument) {
            biz.hammurapi.jms.adapter.definition.JmsAdapter definition = ((JmsAdapterDocument) document).getJmsAdapter();
            JmsAdapter adapter = new JmsAdapter(definition);
            adapter.start();
            try {
                StringLengthService proxy = (StringLengthService) adapter.get("proxy");
                proxy.setValue("Hello");
                System.out.println(proxy.getValue());
                System.out.println(proxy.length());
            } finally {
                adapter.stop();
            }
        } else {
            System.err.println("Invalid adapter definition.");
        }
    }
}
```

Listing 8. ProxyServiceConsumer.java

Listing 8 shows how to invoke the string length service through a proxy. Note that string to be sent to the service is stored in locally by invoking setValue() method and then it is shipped to the service as part of invocation payload.

It is worth mentioning how JMS Adapter implements exception propagation. The adapter tries to reproduce the remote exception. E.g. if our service throws IllegalArgumentException with a message "Unexpected argument of type java.lang.Integer", the service will instantiate the same exception type with the same message when it receives error notification message. Stack trace is not propagated. In the case of proxy services it is possible to define a wrapper exception to "wrap" remote exceptions not listed in interface method throws clause. This allows to avoid UnexpectedThrowableException. Runtime exceptions and declared exceptions are not wrapped.

JCA support

JMS Adapter implements JCA 1.5 Resource Adapter. The distribution package contains ra.xml file. Package it with needed jars into a .rar file and deploy to the application server.

The resource adapter wraps JMS Adapter instance. There are two connection modes - stateless and stateful. In the stateful mode services retrieved from the adapter are cached in connection. In the case of proxy services adapter returns proxy instances. If proxy instances maintains state by using local delegates, then this state becomes associated with the connection.

When JMS Adapter is deployed as JCA Resource adapter, default worker uses WorkManager to execute asynchronous tasks.

JMS Adapter implements activation specification and can deliver invocations to message driven beans which implement any Java interface.

Cookbook

Metrics collection

All services collect metrics such as processing time, number of invocations, number of exceptions. Individual measurements are passed to the measurement consumer, if it is present. In the Listing 3 you can see a very simple configuration of measurement consumer. This configuration aggregates measurements over 60 seconds and prints aggregated values to console.

The Common and Enterprise Extensions libraries contain a number of implementations of measurement collectors for more advanced metrics collection.

On a service bus with many servcies it would be quite natural have a metrics collecting topic and metrics collecting service(s). In this case adapter configuration shall contain a difinition of such a service. Measurement consumer would lookup the service using the naming bus and send aggregated metrics to the bus through that service.

Metrics can be used for service monitoring and alerting. For example if level of errors or processing time on some service goes above some level another service can automatically generate alerts.

You can read more about the metrics collection framework here Metrics framework.

If metrics are collected with consumer code "aspect" they can be used for usage-base billing of consumers in service buses where service consumers have to pay for accessing services.

Orchestraction

In the adapter all components can reference each other through the naming bus. Therefore development of orchestrating services is straightforward:

- Define a listener
- Define service or proxy service entries to access services being orchestracted
- Develop listener's processor class.
- Have this class implement `biz.hammurapi.config.Component`. The easiest way is to extend `biz.hammurapi.config.ComponentBase`
- In processor's `start()` method lookup services to be orchestrated through `get()` method.
- Implement orchestraction logic in process() method.

Caching

Transparent caching can be implemted by using Proxy Services with caching Local delegates. You can also implement a Generic Service for caching which caching Local delegates will use.

Rules based XML processing

Rules engine, such as Hammurapi Rules, XmlBeans, `biz.hammurapi.util.BeanVisitable`and JMS Adapter can be combined together to implement rules-based validation, enrichment and transformation (e.g. element-level encryption/decryption) of messages with XML payload. Augmented with XSL transformation of incoming messages this combination can be used as an "edge-service" receiving messages from external clients and converting them to internal "canonical" format.

This is how it can be done:

- Define XML schema.
- Generate Java classes and interfaces from the schema using XmlBeans
- Define listener with pooled processor and XmlBeans bind type.
- Implement processor.
  - It shall use JSR-94 API to obtain rules session, pass objects to it and collect results
  - `biz.hammurapi.util.BeanVisitables` shall be used to break incoming XML bean into individual XML types which will be passed to rules.
- Implement rules.

Scenario

Your organization provides different types of lending services. It works with car dealerships, mortgage brokers, furniture stores, etc. Partners collect credit applications from their customers a submit electronically to your organization. Different partners use different data formats. Also different products require different data, e.g. mortgage application is different from car loan application. Nonetheless they have a lot in common, e.g. Address or Person descriptions can be shared between different application types. In order to optimize application processing you can do the following:

- Define internal or "canonical" XML schema for applications. While mortgage application XML type will most probably be different from car loan application XML type, they will have a lot in common, e.g.
  - Share XML types such as Address or Person
  - Extend a base credit application XML type which contains definitions common for all creadit applications.
- Implement converters which convert partner's formats to the canonical schema. For XML-based formats it can be an XSL stylesheet defined in `from-xml-style` property of a listener.
- Implement rules working on different XML types. E.g.
  - Rule to validate US zip code in Address and expand 5 digits zip to 9 digits,
  - Rule to encrypt customer SSN and mother's maiden name. Encryption of entire application may not be practical for a number of reasons, e.g. performance, key management on all services, not only ones which actually need access to sensitive information.

`biz.hammurapi.util.BeanVisitable` will pass each XML type in an XML document being visited to rules. It wouldn't matter at what part of a credit application document Address type is encountered and how many addresses the application contains - each instance of Address XML type will be passed to rules working on Address to be validated and enriched.

Based on validation results incoming documents can be published to the internal bus for further automated processing, returned to the sender, or be sent to manual processing. Internal services will consume validated credit applications in canonical format and as such will be able to concentrate on the service logic instead of validation and format transformation.

Database access

HGee library contains BasicDataSourceComponent class, which extends Apache Commons DBCP's BasicDataSource. This class can be used as Generic Service in JMS Adapter.

Working with adapter XML definitions programmatically

HGee distribution contains jms-adapter.jar, which is jms-adapter.xsd compiled with XmlBeans. You can use it to programmatically construct adapter definitions from other sources, e.g. database tables.

Roadmap

We expect the adapter to grow organically through development of Bind Type implementations by the user community and by the core team on as needed basis.

Other areas of improvement of the JMS Adapter include:

- Argun-based Web front-end (service regsitry) for service management, metrics collection and monitoring.
- Integration with Argun web diagrams and service diagrams.
- Service to execute Argun's composite service diagrams.

Glossary

Class
L
Local delegate

Local delegate maintains proxy object state. Local delegate performs proxy invocations not mapped to JMS services. Remote invocations are also filtered through the local delegate so it can modify arguments, return value or prevent remote invocation at all. For example local delegate can cache values returned by remote invocations. Local delegates must implement biz.hammurapi.jms.adapter.LocalDelegate interface.

S

Service

Component

B

Bind Type

Bind types are components which convert JMS messages to Java objects and vice versa. These components shall implement biz.hammurapi.jms.adapter.Converter interface.

C

Context

JNDI context. Contains JNDI connections.

F

Factory Connection

JMS Connection which is created using vendor-specific API's. Both Factory Connection and JNDI Connection implement connection pooling, session pooling, and automatic re-connnection on failure. Also these components can be configured to refresh underlying JMS connections on a regular interval.

G

Generic Service

Non-JMS service. Generic service can act as a helper component for JMS services or as a consumer/coordinator of JMS services.

J

JMS Adapter

Container of other JMS components. It provides naming bus and lifecycle management services for its children. Adapter is configured from an XML document (definition). The definition shall conform to http://www.hammurapi.biz/jms/adapter/definition XML schema defined in jmd-adapter.xsd file in HGee distribution. The schema file contains detailed description of each XML type and element.

JMS Service

JMS Service is a component capable of sending JMS messages. It can send messages in fire-and-forget and request-reply modes. Sending messages can be done synchronously (in the current thread), or asynchronously (by delegating to worker). Reply can be received by the caller as method return value or through a callback interface instance.

JNDI Connection

JMS Connection, which connection factory is obtained through JNDI lookup. Both Factory Connection and JNDI Connection implement connection pooling, session pooling, and automatic re-connnection on failure. Also these components can be configured to refresh underlying JMS connections on a regular interval.

L

Listener

JMS listener. This component uses Bind Type components to convert messages to Java objects. Java objects are then passed to processor instance for further processing.

M

Measurement Consumer

This component collects metrics reported by other components.

Method

Specifies how interface method invocation shall be translated to JMS request/reply or fire-and-forget (if method is void or return value is of no interest for the caller). Methods can be configured to be executed asynchronously and store return value to local proxy state (pre-fetch).

P

Proxy Service

Proxy service maps Java interface method calls to JMS request/reply message exchange. Proxy instances created by proxy service maintain local state and local behavior through Local delegate. Local delegate also acts as a filter for remote calls. In other words, proxy instance is an object with local state and distributed behavior. With proxy services remote services on the service bus can be stateless but appear to be stateful for consumers.

A proxy service can implement any Java interface or collection of interfaces. This feature allows almost transparent migration to message-based communications from other remoting technologies such as RMI.

W

Worker

Workers, typically implemented as thread pools, are used for delegation of work from other components for asynchrhonous execution.