Using the Celtix management

Table of Contents

Overview

Celtix provides management facilities which bases on Java Management Extensions (JMX) to instrument its core runtime. Several key runtime components are exposed as JMX Managed Beans (MBeans). This lets an Celtix runtime be monitored and managed either in process or remotely with the help of JMXRemote API.

A support for registering custom MBeans is avalable in Celtix 1.0. Java developers can create their own Mbeans statically or dynamically and register them either with their MBeanServer of choice or with a default MBeanServer created by Celtix.

What is special about Celtix 1.0 though is that some of its key runtime components can now be exposed as JMX MBeans dynamcially .

Default instrumentation in Celtix

As noted above, the Celtix servers can have their runtime components exposed as JMX MBeans. At the moment, the following components can be managed :

- CeltixBus
- WorkQueue
- WSDLManager
- Endpoint
- HTTPServerTransport
- JMXServerTransport

All runtime components are registered with an MBeanServer as Model Dynamic MBeans. This ensures that they can be viewed by third-party management consoles without any additional client-side support libraries.

These components follow an advice from a JMX Best Practices document on how to name MBeans. All Celtix runtime MBeans use "`org.objetweb.celtix.instrumentation`" as their domain name when creating ObjectName.

Component	Instance name	Object Name Value
CeltixBus	Bus name	`org.objectweb.celtix.instrumentation:type=Bus,name=celtix`
WorkQueue	Bus name and string constant	`org.objectweb.celtix.instrumentation:type=Bus.WorkQueue,Bus=celtix,name=WorkQueue`
WSDLManager	Bus name	`org.objectweb.celtix.instrumentation:type=Bus.WSDLManager,Bus=celtix,name=WSDLManager`
Endpoint	Bus name, QName of service, name of port, and Endpoint	`org.objectweb.celtix.instrumentation:type=Bus.Endpoint,Bus=celtix,Bus.Service={http://objectweb.org/hello_world}SOAPService",Bus.Port=SoapPort,name=Endpoint`
HTTP Server	Bus name,	`org.objectweb.celtix.instrumentation:type=Bus.Service.P`

Component	Instance name	Object Name Value
Transport	QName of service, name of port, and HTTPServerTransport	`ort.HTTPServerTransport,Bus=celtix,Bus.Service={http://objectweb.org/hello_world}SOAPService",Bus.Port=SoapPort,name=HTTPServerTransport"`
JMS Server Transport	Bus name, QName of service, name of port, and JMSServer Transport	`org.objectweb.celtix.instrumentation:type=Bus.Service.Port.JMSServerTransport,Bus=celtix,Bus.Service={http://objectweb.org/hello_world}SOAPService",Bus.Port=SoapPort,name=JMSServerTransport"`

CeltixBus

The following attributes are currently supported :

Name	Description	Type	Read/Write
TransportFactories	Bus Transport Factories name	String[]	R
BindingFactories	Bus Binding Factories name	String[]	R
ServiceMonitoring	Used to enable/disable transport performance counters monitoring	Boolean	RW

WorkQueue

The following attributes are currently supported :

Name	Description	Type	Read/Write
ThreadingModel	Threading Model can be SINGLE_THREADED or MULTI_THREADED	String	R
WorkQueueSize	The thread pool size	Integer	R
Empty	The flag that indicate the WorkQueue is empty	Boolean	R
HighWaterMark	A number marking the highest busy level of WorkQueue reached.	Integer	RW
LowWaterMark	A number marking the lowest busy level of	Integer	RW

	WorkQueue reached.		
Full	The flag that indicate the WorkQueue is full	Boolean	R

WSDLManager

The following attributes are currently supported :

Name	Description	Type	Read/Write
Services	A list of Services QNames that the celtix wsdl manager provides	String[]	R
Ports	A list of ports name that the celtix wsdl manager provide	String[]	R
Bindings	A list of bindings name that the celtix wsdl manager provide	String[]	R

the following operation is currently supported :

Name	Description	Parameters	Return Type
GetOperation	Get the specified Service and Port's operation name	ServiceName, PortTypeName	String[]

Endpoint

The following attributes are currently supported :

Name	*Description*	*Type*	*ReadWrite*
ServiceName	Service QName in expanded form	String	R
PortName	The port name which registered with the Endpoint	String	R
HandlerChains	A list of classes name that Endpoint's ServiceBinding customed	String[]	R
State	Endpoint Service state	String	R

The following operations are currently supported :

Name	Description	Parameters	Return Type
start	Start (activate) a service	None	Void
stop	Stop (deactivate) a service	None	Void

HTTPServerTransport

The following attributes are currently supported :

Name	Description	Type	ReadWrite
ServiceName	Service QName in expanded form	String	R
PortName	The PortName of the HttpServerTransport	String	R
Url	The url which HTTPServerTransport listens to	String	R
TotalError	Total number of request-processing errors	Integer	R
RequestTotal	Total number of requests (including oneway) to this service	Integer	R
RequestOneWay	Total number of requests only oneway to this service	Integer	R

JMSServerTransport

The following attributes are currently supported :

Name	Description	Type	ReadWrite
ServiceName	Service QName in expanded form	String	R
PortName	The PortName of the HttpServerTransport	String	R
Url	The url which HTTPServerTransport listens to	String	R
TotalError	Total number of request-processing errors	Integer	R
RequestTotal	Total number of requests (including oneway) to this	Integer	R

	service		
RequestOneWay	Total number of requests only oneway to this service	Integer	R

Celtix Instrumentation Configuration

To have the Celtix runtime exposed as JMX MBeans one needs to enable instrumentation and JMX facilities to work. The other needs to know how to establish a connector to the JMX MBServer. These information can be provided in Celtix Configuration:

Celtix management configuration is specified using the `org.objectweb.celtix.bus.instrumentation.instrumentation_config.spring.InstrumentationConfigBean` class for the configuration bean. Using this configuration bean, you specify the Celtix instrumentation event listener behaviors by using the `instrumentationControl` property. It has two values , one is `InstrumentationEnabled` ,the other is `JMXEnabled` .

`InstrumentationEnable`	Used to enable/disable Celtix runtime component's created and removed related event handling
`JMXEnable`	Used to enable/disable Celtix JMX related MBean register and unregister event handling

Using the management configuration bean, you also specify the MBeanServer behaviors using the `MBServer` property. It has one value which is `JMXConnector`

`JMXConnector`	Specifies how to setup the JMXConnectorServer which provider the remote connection to JMXServer. This value contains three subvalues. `Threaded` ,if it is setted to be true means that JMXConnectorServer run in a new created thread . `Daemon`,if it is setted to be true and the `Threaded` set to be true means that the JMXConnectorServer thread which run the MBeanServer will be set to Daemon mode. `JMXSerivceURL`, means the Remote access is done through JMX Remote, using the JMXServiceURL. See javax.management.remote.rmi package details for more details on JMXServiceURLs.

```
<bean id="celtix.Instrumentation"
    class="org.objectweb.celtix.bus.instrumentation.instrumentation_config.spring.InstrumentationConfigBean">
   <property name="instrumentationControl">
     <value>
         <im:instrumentation>
              <im:InstrumentationEnabled>true</im:InstrumentationEnabled>
              <im:JMXEnabled>true</im:JMXEnabled>
         </im:instrumentation>
     </value>
   </property>
   <property name="MBServer">
     <value>
       <im:MBServer>
         <im:JMXConnector>
           <im:Threaded>true</im:Threaded>
           <im:Daemon>false</im:Daemon>
           <im:JMXServiceURL>service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi/server</im:JMXServiceURL>
          </im:JMXConnector>
        </im:MBServer>
     </value>
   </property>
 </bean>
```

Text 1: management setup Information in Celtix Configuration

More information

For more information on using Celtix configuration see the Celtix Configuration Guide.

Accessing Celtix MBeans from management consoles

Celtix runtime MBeans can be accessed remotely using JMX Remote. As such, any third party consoles supporting JMXRemote can be used to monitor and manage celtix servers.

We can recommend a jconsole tool shipped with JDK 1.5. Just launch a <JDK_HOME>/bin/jconsole, select 'Advanced' tab and enter or paste a JMXServiceURL (default one or the one copied from instrumentation configuration `JMXServiceURL`), and that's it.

Instrumentation Celtix Service

As the abover overview has mentioned, the Celtix application developers can create their own MBean and registed to the Celtix MBeanServer. Celtix allows the registration of additional MBeans with the Celtix MBean server. This makes it possible for you to add custom instrumentation to your service implementations and mange it through the same management console as the other Celtix components.

There are two methods of instrumenting your service implementations:

- implement one of the JMX MBean interfaces and register it with Celtix's MBean server.
- implement an Celtix `instrumentation` interface.

Functionally there is no different between the two approaches. The decision on which to use depends on ease of development, maintainability, and portability.

Using the JMX APIs

The Celtix MBean server can be accessed through the Celtix bus and allows for the registration of user developed MBeans. This allows you to instrument your service implementation by developing a custom MBean using one of the JMX MBean interfaces and registering it with the Celtix MBean server. Your custom instrumentation will then be accessible through the same JMX connection as the Celtix internal components used by your service.

In order to access Celtix runtime MBeans, one needs to get a handle to the MBeanServer first.

The following code snippet shows how to access it locally :

```
Bus bus = Bus.getCurrent();
MBeanServer mbeanServer = bus.getInstrumentationManager().getMBeanServer();
```

And here is how to access it remotely :

```
// The address of the connector server

String url = "service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi/server";

JMXServiceURL address = new JMXServiceURL(url);

// Create the JMXConnectorServer

JMXConnector cntor = JMXConnectorFactory.connect(address, null);

// Obtain a "stub" for the remote MBeanServer

MBeanServerConnection mbsc = cntor.getMBeanServerConnection();
```

When you use the JMX APIs to instrument your service implementation, you follow the design methodology laid out by the JMX specification. This involves the following steps:

1. Decide what type of MBean you wish to use.
    - standard MBeans expose a management interface that is defined at development time.
    - dynamic MBeans expose their management interface at run time.
2. Create the MBean interface to expose the properties and operations used to manage your service implementation.

- standard MBeans use the `MBean` interface.
- dynamic MBeans use the `DynamicMBean` interface.

3. Implement the MBean class.
4. Register MBean class to Celtix MBeanServer

Example: defined a MBean Interface

```
public interface ServerMBean {
   String getServiceName();
   String getAddress();
}
```

Example: Create MBean implementation and register it to Celtix MBeanServer

```
import javax.management.MBeanServer;
import javax.management.ObjectName;
import javax.xml.ws.Endpoint;
import org.objectweb.celtix.Bus;

public class Server implements ServerMBean {

   private static GreeterImpl implementor;
   private String address;

   protected Server() throws Exception {
      System.out.println("Starting Server");
      implementor = new GreeterImpl();
      address = "http://localhost:9000/SoapContext/SoapPort";
      Endpoint.publish(address, implementor);

       //register to the bus MBServer
      Bus bus = Bus.getCurrent();
      MBeanServer mbserver = bus.getInstrumentationManager().getMBeanServer();
      ObjectName name = new
ObjectName("org.objectweb.celtix.instrumentation:type=ServerMBean,Bus="
                + bus.getBusID() + ",name=ServerMBean");
      mbserver.registerMBean(this, name);
   }

   public String getServiceName() {
      return "SoapService";
   }

   public String getAddress() {
      return address;
```

```
  }

  ....

}
```

Using the Celtix Instrumentation Interface

If you do not want to use the JMX interfaces to add custom instrumentation to your service, you can use the Celtix `Instrumentation` interface. This interface wraps the JMX subsystem in proprietary interfaces. You do not need to access the Celtix JMX server to add instrumentation to your service.

To add custom instrumentation to your service using the `ManagedComponent` interface you need to do the following:

1. Implement an instrumentation class that implements the `org.objectweb.celtix.Instrumentation` interface.
2. In the service's initialization routine, instantiate your instrumentation object and register it with the bus.
3. In the service's shutdown routine, unregister your instrumentation object.

Implementing the Instrumentation Class

Like an MBean, a instrumentation class is responsible for providing access to the attributes you want to track and any management operations you want to expose.

Unlike an MBean, you do not need to define an interface for your instrumentation class. Instead, your instrumentation object implements an Celtix management interfaces and defines the operations required to expose the attributes and operations you want.

Celtix management facilities also provide an `MBeanInfoAssembler` which can get instrumented component management interface which is defined with JDK5.0 annotation at the runtime. To enable the use of JDK 5.0 annotations for management interface definition, Celtix provides a set of annotations that mirror the ModelMBean's description, that allows the `MBeanInfoAssembler` to read them and Generate correct ModelMBeanInfo.

To mark a components for export to JMX, you should annotate the instrumentation class with the `ManagedResource` attribute. Each method you wish to expose as an operation should be marked with a `ManagedOperation` attribute and each property you wish to expose should be marked with a `ManagedAttribute` attribute. When marking properties you can omit either the getter or the setter to create a write-only or read-only attribute respectively.

The example below shows a class with a JDK 5.0 annotation defined management interface:

```
package demo.hw.server;

import org.objectweb.celtix.bus.management.jmx.export.annotation.ManagedAttribute;
import org.objectweb.celtix.bus.management.jmx.export.annotation.ManagedOperation;
import org.objectweb.celtix.bus.management.jmx.export.annotation.ManagedResourec;
import org.objectweb.celtix.bus.management.Instrumentation;

@ManagedResource(componnetName=”GreeterInstrumentation”,
        decription = "The Celtix Service instrumentation demo component ",
        currencyTimeLimit = 15, persistPolicy = "OnUpdate")
```

```
public class GreeterInstrumentation implements Instrumentation {

  private GreeterImpl greeter;

  public GreeterInstrumentation(GreeterImpl gi) {
      greeter = gi;
  }

  //set up the management component type name
  public String getInstrumentationName() {
    return "GreeterInstrumentation";
  }

  //The instrumentation managemented component reference
  public Object getComponent() {
    return this;
  }

  //The instrumentation unique name for Object Name
  public String getUniqueInstrumentationName() {
    return ",name=Demo.Management"
  }

   @ManagedAttribute(description = "Get the GreetMe call counter")
  public Integer getGreetMeCounter() {
    return greeter.requestCounters[0];
  }

  @ManagedAttribute(description = "Get the GreetMeOneWay call counter")
  public Integer getGreetMeOneWayCounter() {
    return greeter.requestCounters[1];
  }

  @ManagedAttribute(description = "Get the SayHi call counter")
  public Integer getSayHiCounter() {
    return greeter.requestCounters[2];
  }

  @ManagedAttribute(description = "Get the Ping me call counter")
  public Integer getPingMeCounter() {
    return greeter.requestCounters[3];
  }

  @ManagedAttribute(description = "Set the Ping me call counter")
  public void setPingMeCounter(Integer value) {
    greeter.requestCounters[3] = value;
  }

  @ManagedOperation(description = "set the SayHi return name",
          currencyTimeLimit = -1)
  public void setSayHiReturnName(String name) {
    greeter.returnName = name;
  }
}
```

Listing 2: This is an example of using JDK 5.0 annotation to define the management interface.

JMX annotation class

The following annotation shows metadata types are available for use in Celitx JMX:

Celtix JMX annotation Types

Purpose	JDK 5.0 Annotation	Attribute / Annotation Type
Mark all instances of a Class as JMX managed resources	`@ManagedResource`	Class
Mark a method as a JMX operation	`@ManagedOperation`	Method
Mark a getter or setter as one half of a JMX attribute	`@ManagedAttribute`	Method (only getters and setters)
Define descriptions for operation parameters	`@ManagedOperationParameter` and `@ManagedOperationParameters`	Method

The following configuration parameters are available for use on these metadata types:

Parameter	Description	Applies to
`componentName`	Used to set the name description of a managed resource	`ManagedResource`
`description`	Sets the friendly description of the resource, attribute or operation	`ManagedResource`, `ManagedAttribute`, `ManagedOperation`, `ManagedOperationParameter`
`currencyTimeLimit`	Sets the value of the `currencyTimeLimit` descriptor field	`ManagedResource`, `ManagedAttribute`
`defaultValue`	Sets the value of the `defaultValue` descriptor field	`ManagedAttribute`
`log`	Sets the value of the `log` descriptor field	`ManagedResource`
`logFile`	Sets the value of the `logFile` descriptor field	`ManagedResource`
`persistPolicy`	Sets the value of the `persistPolicy` descriptor field	`ManagedResource`
`persistPeriod`	Sets the value of the `persistPeriod` descriptor field	`ManagedResource`
`persistLocation`	Sets the value of the `persistLocation` descriptor field	`ManagedResource`
`persistName`	Sets the value of the `persistName` descriptor field	`ManagedResource`
`name`	Sets the display name of an operation parameter	`ManagedOperationParameter`
`index`	Sets the index of an operation parameter	`ManagedOperationParameter`

Creating and Removing your Instrumentation

To make your custom instrumentation available to management consoles, you must create an instance of your instrumentation class. Then you just need to register your instrumentation to the bus. The bus automatically registers the MBean with the Celtix JMX server.

Unlike JMX-style instrumentation, Celtix API instrumentation must be cleaned up. In your services shutdown() routine you need to tell the bus to remove the MBean created for your instrumentation. This also cleans up any other resources created to support the custom instrumentation.

As with JMX-style instrumentation Celtix API instrumentation is not available until an MBean is created and registered with the Celtix MBean server. However, when you create Celtix API instrumentation you do not directly create an MBean or register it with the MBean server. This is all handled by the bus.

To create an MBean for your instrumentation and register it with the MBean server do the following:

1. Instantiate an instance of your instrumentation class.
2. Register instrumentation class with Instrumentation Manager.

```
Instrumentation in = new GreeterInstrumentation(this);
Bus bus = Bus.getCurrent();
InstrumentationManager im = bus.getInstrumentationManager();
im.register(in);
```

To clean up your custom instrumentation you need to unregister the MBean created to support it and destroy the MBean. This is all done by call the instrumentation manager unregister method.

To remove your custom instrumentation from the JMX server do the following:

Unregist instrumentation object with Instrumentation Manager.

```
Bus bus = Bus.getCurrent();
InstrumentationManager im = bus.getInstrumentationManager();
im.unregister(in);
```