BEA WebLogic Server

Timer and Work Manager API (CommonJ) Programmer's Guide

Version 10.0
Document Revised: March 30, 2007

Contents

The Timer and Work Manager API

The Timer and Work Manager API

This document provides an overview of the Timer and Work Manager API and demonstrates how to implement it within an application.

Description of the Timer and Work Manager API

The Timer and Work Manager API is defined in a specification created jointly by BEA Systems and IBM. This API enables concurrent programming of EJBs and Servlets within a Java EE application. This API is often referred to as CommonJ.

The CommonJ API contains the following components:

- Timer API

  The Timer API allows applications to schedule and receive timer notification callbacks for a specific listener defined within an application. Timers allow you to schedule and perform work at specific times or intervals. See "Overview of the Timer API" on page 2.

You implement this API by importing the `commonj.timer` package.

- Work Manager API

  The Work Manager API allows an application to prioritize work within an EJB or Servlet. Applications can programmatically execute multiple work items within a container. See "Description of the Work Manager API" on page 9.

  You implement this API by importing the `commonj.work` package.

  In addition to the CommonJ Work Manager API, WebLogic Server includes server-level Work Managers that provide prioritization and thread management. These can be configured globally or for a specific module in an application.

Although `commonj.timer` and `commonj.work` are part of the same API, each provides different functionality. Which one you implement depends on the specific needs of your application. The CommonJ Timer API is ideal for scheduling work at specific intervals; for example, when you know that a certain job should run at a specific time. The CommonJ Work API is ideal for handling work based on priority. For example, you may not be able to predict exactly when a specific job will occur, but when it does you want it to be given a higher (or lower) priority.

The following sections describe the CommonJ APIs in detail.

Overview of the Timer API

The Timer API consist of three interfaces:

- Timer Manager
- Timer Listener
- Timer object

The Timer Manager provides the framework for creating and using timers within a managed environment. The Timer Listener receives timer notifications. The TimerManager.schedule() method is used to schedule the TimerListener to run at a specific time or interval.

For a detailed description of how to implement Timers, see "Using the Timer API" on page 3.

The TimerManager Interface

The TimerManager interface provides the general scheduling framework within an application. A managed environment can support multiple TimerManager instances. Within an application you can have multiple instances of a TimerManager.

Creating and Configuring a Timer Manager

TimerManagers are configured during deployment via deployment descriptors. The TimerManager definition may also contain additional implementation-specific configuration information.

Once a TimerManager has been defined in a deployment descriptor, instances of it can be accessed using a JNDI lookup in the local Java environment Each invocation of the JNDI `lookup()` for a TimerManager returns a new logical instance of a TimerManager.

The TimerManager interface is thread-safe.

For more information on using JNDI, see *Programming WebLogic JNDI*.

Suspending a TimerManager

You can suspend and resume a TimerManager by using the suspend() and resume() methods. When a TimerManager is suspended, all pending timers are deferred until the TimerManager is resumed.

Stopping a TimerManager

You can stop a TimerManager by using the stop() method. After stop() has been called, all active Timers will be stopped and the TimerManager instance will stop monitoring all TimerListener instances.

The TimerListener Interface

All applications using the `commonj.timers` package are required to implement the TimerListener interface.

The Timer Interface

Instances of the Timer interface are returned when timers are scheduled through the TimerManager.

Using the Timer API

This section outlines the steps required to use the Timer API within an application.

Before deploying your application, ensure that you have created a deployment-descriptor that contains a resource reference for the Timer Manager.

This allows the TimerManager to be accessed via JNDI. For more information see See *Programming WebLogic JNDI* for more information on JNDI lookup.

The following procedure describes how to implement the Timer API:

1. Import `commonj.timers.*` packages.
2. Create an InitialContext that allows the TimerManager to be looked up in JNDI

   ```
   IntialContext inctxt = new InitialContext();
   ```

   See *Programming WebLogic JNDI* for more information on JNDI lookup.
3. Create a new TimerManager based on the JNDI lookup of the TimerManager

   ```
   TimerManager mgr =
   (TimerManager)ctx.lookup('java:comp/env/timer/MyTimer');
   ```

   In this statement, the result of the JNDI lookup is cast to a TimerManager.
4. Implement a TimerListener to receive timer notifications.

   ```
   TimerListener listener = new StockQuoteTimerListener('abc', 'example');
   ```
5. Call TimerManager.schedule()

   ```
   mgr.schedule(listener, 0, 1000*60)
   ```

   The schedule() method returns a Timer object.
6. Implement timerExpired() method

   ```
   public void timerExpired(Timer timer) {
        //Business logic is executed
        //in this method

   }
   ```

Timer Manager Example

```
package examples.servlets;
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.naming.InitialContext;
import javax.naming.NamingException;
```

```
import commonj.timers.*;

/**
* TimerServlet demonstrates a simple use of commonj timers
*/
public class TimerServlet extends HttpServlet {

/**
  * A very simple implementation of the service method,
  * which schedules a commonj timer.
  */
  public void service(HttpServletRequest req, HttpServletResponse res)
    throws IOException
  {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    try {
      InitialContext ic = new InitialContext();
      TimerManager tm = (TimerManager)ic.lookup
        ("java:comp/env/tm/default");
      // Execute timer every 10 seconds starting immediately
      tm.schedule (new MyTimerListener(), 0, 10*1000);
      out.println("<h4>Timer scheduled!</h4>");
    } catch (NamingException ne) {
      ne.printStackTrace();
      out.println("<h4>Timer schedule failed!</h4>");
    }
  }

  private static class MyTimerListener implements TimerListener {
    public void timerExpired(Timer timer) {
      System.out.println("timer expired called on " + timer);
      // some useful work here ...
      // let's just cancel the timer
      System.out.println("cancelling timer ...");
      timer.cancel();
    }
```

```
    }
}
```

Using the Job Scheduler

This section describes how to use the Job Scheduler functionality. The Job Scheduler allows you to implement the commonj.timer API within a clustered environment.

The Job Scheduler is essentially an implementation of the commonj.timer API package that can be used within a cluster. In this context, a job is defined as a commonj.timers.TimerListener instance that is submitted to the Job Scheduler for execution.

This section covers the following topics:

Life Cycle of Timers

When you implement the commonj.timer API within an application, you can configure two possible life cycles for a timer.

- Local timers

  Local timers are scheduled within a single server JVM and are handled within this JVM throughout their life cycle. The timer continues running as long as the JVM is running and fails when the JVM exits. The application is responsible for rescheduling the timer after server startup.

  This is the basic implementation of commonj.timers and is described in "Overview of the Timer API."

- Cluster-wide timers

  A cluster-wide timer is aware of the other JVMs containing each server within the cluster and is therefor able to perform load balancing and failover. The life cycle of a cluster-wide timer is not bound to the server that created it, but continues to function throughout the life cycle of the cluster. As long as at least one cluster member is alive the timer continues to function. This functionality is referred to as the Job Scheduler.

Each timer has its own advantages and disadvantages. Local timers are able to process jobs with much smaller time intervals between jobs. Due to the persistence requirements within a cluster,

Job Schedulers cannot handle jobs with as much precision. Job Schedulers, on the other hand, are better suited for tasks that must be performed even if the initial server that created the task has failed.

Implementing and Configuring Job Schedulers

This sections outlines the basic procedures for implementing Job Schedulers within an application and for configuring your WebLogic Server environment to utilize them.

Database Configuration

In order to maintain persistence and make timers cluster aware, Job Schedulers require a database connection. The Job Scheduler functionality supports the same database vendors and versions that are supported by server migration.

For convenience, you can use the same database used for session persistence, server migration, etc.

In the database, you must create a table called `WEBLOGIC_TIMERS` with the following schema:

```
CREATE TABLE WEBLOGIC_TIMERS (
  TIMER_ID VARCHAR2(100) NOT NULL,
  LISTENER BLOB NOT NULL,
  START_TIME NUMBER NOT NULL,
  INTERVAL NUMBER NOT NULL,
  TIMER_MANAGER_NAME VARCHAR2(100) NOT NULL,
  DOMAIN_NAME VARCHAR2(100) NOT NULL,
  CLUSTER_NAME VARCHAR2(100) NOT NULL,
  CONSTRAINT SYS_C00323062 PRIMARY KEY(TIMER_ID, CLUSTER_NAME, DOMAIN_NAME)
)
```

Data Source Configuration

After you have created a table with the required schema, you must define a data source that is referenced from within the cluster configuration. Job Scheduler functionality is only available if a valid data source is defined in for the DataSourceForJobScheduler attribute of the Cluster MBean. You can configure this from the WebLogic Server Administration Console.

The following excerpt from `config.xml` demonstrates how this is defined:

```
<domain>
...
```

```
 <cluster>
  <name>Cluster-0</name>
  <multicast-address>239.192.0.0</multicast-address>
  <multicast-port>7466</multicast-port>
  <data-source-for-job-scheduler>JDBC Data
    Source-0</data-source-for-job-scheduler>
 </cluster>
...
 <jdbc-system-resource>
  <name>JDBC Data Source-0</name>
  <target>myserver,server-0</target>
  <descriptor-file-name>jdbc/JDBC_Data_Source-0-3407-jdbc.xml</descriptor-
    file-name>
 </jdbc-system-resource>
</domain>
```

JNDI Access within a Job Scheduler

The procedure for performing JNDI lookup within a clustered timer is different from that used in the general commonj.timer API. The following code snippet demonstrates how to cast a JNDI lookup to a TimerManager.

```
InitialContext ic = new InitialContext();
commonj.timers.TimerManager jobScheduler =(common.timers.TimerManager)ic.lookup
   ("weblogic.JobScheduler");
commonj.timers.TimerListener timerListener = new MySerializableTimerListener();
jobScheduler.schedule(timerListener, 0, 30*1000);
// execute this job every 30 seconds
```

Unsupported Methods and Interfaces

Not all methods and interfaces of the commonj.timer API are supported in the Job Scheduler environment. The following methods and interfaces are not supported:

- `CancelTimerListener` interface
- `StopTimerListener` interface
- The following methods of the `TimerManager` interface:
  - `suspend()`
  - `resume()`

- `scheduleAtFixedRate()`
- `stop()`
- `waitForStop()`
- `waitForSuspend()`

Description of the Work Manager API

The Work Manager (`commonj.work`) API provides is an interface that allows an application to executed multiple work items concurrently within a container.

Essentially, this API provides a container-managed alternative to the java.lang.Thread API. The latter should not be used within applications that are hosted in a managed Java EE environment. In such environments, the Work Manager API is a better alternative because it allows the container to have full visibility and control over all executing threads.

Note: The Work Manager API provides no failover or persistence mechanisms. If the Managed Server environment fails or is shut down, any current work will be lost.

Work Manager Interfaces

This section provides a general overview of the interfaces defined in the Work Manager API.For more detailed information on using these interfaces, see the javadocs for the commonj.work package.

The Work Manager API contains the following interfaces:

- **WorkManager** - This interface provides a set of scheduling methods that are used to schedule work for execution.

  WorkManagers are defined by system administrators at the server level. A WorkManager instance is obtained by performing a JNDI lookup. A managed environment can support multiple WorkManager instances. See *Programming WebLogic JNDI*. You configure WorkManagers during deployment as resource-refs. See "Work Manager Deployment."

  At the application level, each instance of WorkManager returns a WorkItem. For more information on implementing a WorkManager within an application, see the WorkManager javadocs.

- **Work** - This interface allows you to run application code asynchronously. By creating a class that implements this interface, you can create blocks of code that can be scheduled to run at a specific time or at defined intervals. In other words, this is the "work" that is handled within the Work Manager API.

- **WorkItem** - After a `Work` instance has been submitted to a WorkManager, the WorkManager returns a WorkItem. This WorkItem is used determine the status of the completed Work instance.
- **WorkListener** - The WorkListener interface is a callback interface that provides communication between the WorkManager and the scheduled work defined within the Work instance.

  You can use WorkListener to determine the current status of the Work item. For more information, see the WorkListener javadocs.

  Note: WorkListener instances are always executed in the same JVM as the original thread used to schedule the Work via the WorkManager.
- **WorkEvent** - A WorkEvent is sent to a WorkListener as Work is processed by WorkManager.
- **RemoteWorkItem** - The RemoteWorkItem interface is an extension of the WorkItem interface that allows work to be executed remotely. This interface allows serializable work to be executed on any member in a cluster.

Work Manager Deployment

Work Managers are defined at the server level via a resource-ref in the appropriate deployment descriptor. This can be `web.xml` or `ejb-jar.xml` among others.

The following deployment descriptor fragment demonstrates how to configure a WorkManager:

```
...
<resource-ref>
   <res-ref-name>wm/MyWorkManager</res-ref-name>
   <res-type>commonj.work.WorkManager</res-type>
   <res-auth>Container</res-auth>
   <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
...
```

Note: The recommended prefix for the JNDI namespace for WorkManager objects is `java:comp/env/wm`.

Work Manager Example

This section contains a working code example using a CommonJ Work Manager within an HTTP Servlet.

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import weblogic.work.ExecuteThread;
import commonj.work.WorkManager;
import commonj.work.Work;
import commonj.work.WorkException;

public class HelloWorldServlet extends HttpServlet {

   public void service(HttpServletRequest req, HttpServletResponse res)
      throws IOException
   {
      res.setContentType("text/html");
      PrintWriter out = res.getWriter();


      try {
         InitialContext ic = new InitialContext();
         System.out.println("## [servlet] executing in: " +
            ((ExecuteThread)Thread.currentThread()).getWorkManager()
            .getName());
         WorkManager wm = (WorkManager)ic.lookup
            ("java:comp/env/foo-servlet");
         System.out.println("## got Java EE work manager !!!!");
         wm.schedule(new Work(){
         public void run() {
         ExecuteThread th = (ExecuteThread) Thread.currentThread();
         System.out.println("## [servlet] self-tuning workmanager: " +
```

```
            th.getWorkManager().getName());
          }
          public void release() {}

          public boolean isDaemon() {return false;}
          });
    }
    catch (NamingException ne) {
          ne.printStackTrace();}

    catch (WorkException e) {
          e.printStackTrace();
    }

    out.println("<h4>Hello World!</h4>");
    // Do not close the output stream - allow the servlet engine to close it
    // to enable better performance.
    System.out.println("finished execution");}

    }
```