Oracle® Fusion Middleware

Developing CommonJ Applications for Oracle WebLogic Server

F18336-02
November 2022

Oracle Fusion Middleware Developing CommonJ Applications for Oracle WebLogic Server,

F18336-02

Contents

Preface

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

Audience

This document is a resource for system administrators and operators responsible for monitoring and managing a WebLogic Server installation. It is relevant to all phases of a software project, from development through test and production phases.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Accessible Access to Oracle Support

Oracle customers who have purchased support have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Resources

New and Changed WebLogic Server Features

For a comprehensive listing of the new WebLogic Server features introduced in this release, see *What's New in Oracle WebLogic Server*.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 Using the Timer and Work Manager API

This chapter provides an overview of the Timer and Work Manager API and demonstrates how to implement it within an application.
This chapter includes the following sections:

Overview

The Timer and Work Manager API is defined in a specification created jointly by Oracle 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 Timer API Overview.

  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 Work Manager API.

  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.

Timer API Overview

The Timer API consist of three interfaces:

- `TimerManager`
- `TimerListener`
- `Timer`

The `TimerManager` interface provides the framework for creating and using timers within a managed environment. The `TimerListener` 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.

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 TimerManager

A `TimerManager` is configured during deployment by means of deployment descriptors. The `TimerManager` definition may also contain additional implementation-specific configuration information.

Once a `TimerManager` is 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()` on a `TimerManager` returns a new logical instance of a `TimerManager`.

The `TimerManager` interface is thread-safe.

For more information about using JNDI, see *Developing JNDI Applications for Oracle WebLogic Server*.

Suspending a TimerManager

You can suspend and resume a `TimerManager` 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` using the `stop` method. After the `stop` method is invoked, all active Timers are stopped and the `TimerManager` instance stops 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 explains the steps required for using 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 using JNDI. For more information about JNDI lookup, see *Developing JNDI Applications for Oracle WebLogic Server*.

Implementing the Timer API

To implement the Timer API, complete the following steps:

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

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

   See *Developing JNDI Applications for Oracle WebLogic Server* for more information about JNDI lookup.
3. Create a new `TimerManager` based on the JNDI lookup of the `TimerManager`. For example:

   ```
   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. For example:

   ```
   TimerListener listener = new StockQuoteTimerListener('abc', 'example');
   ```
5. Invoke the `TimerManager.schedule` method. For example:

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

   The `schedule` method returns a `Timer` object.
6. Implement the `timerExpired` method. For example:

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

Implementing the Timer API for cluster-wide timers has additional requirements, described in Life Cycle of Timers.

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 explains 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 includes the following topics:

- Life Cycle of Timers
- Implementing and Configuring Job Schedulers
- Unsupported Methods and Interfaces

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 timer

  A local timer is scheduled within a single server JVM and is handled within this JVM throughout its 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 the `commonj.timers` package.

- Cluster-wide timer

  A cluster-wide timer is aware of the other JVMs containing each server within the cluster and is therefore 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. If at least one cluster member is alive, the timer continues to function. This functionality is referred to as the Job Scheduler.

  Implementing the Timer API for a Job Scheduler has the following requirements in addition to those listed in Implementing the Timer API:

  – The Timer Listener class must be serializable.

  – The Timer Listener class must be present in the server system classpath.

  – The minimum time for recurring execution of a timer is 30 seconds because Job Schedulers pick up timers for execution every 30 seconds.

Each timer has its own advantages and disadvantages. Local timers can 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. On the other hand, Job Schedulers 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 section describes the basic procedure for implementing Job Schedulers within an application and for configuring your WebLogic Server environment to utilize them. The following topics are included:

- Database Configuration
- Data Source Configuration
- Leasing

- JNDI Access within a Job Scheduler
- Canceling Jobs
- Debugging

Database Configuration

To maintain persistence and make timers cluster-aware, Job Schedulers require a database connection. The Job Scheduler functionality supports the same databases that are supported by server migration.

For convenience, you can use the same database used for session persistence, server migration, and so on. For example, see Configure server migration in a cluster in the *Oracle WebLogic Server Administration Console Online Help* for information about how to create or select a data source for server migration.

In the database, you create a table named `WEBLOGIC_TIMERS`. Schemas for creating this table are in the following location:

```
WL_HOME/server/db/dbname/scheduler.ddl
```

In the preceding path, *`dbname`* represents the name of the database.

Note:

`WEBLOGIC_TIMERS` table can also be configured by using ClusterMBean attribute `jobSchedulerTableName`.

Data Source Configuration

After you create a table with the required schema, you must define a data source that is referenced from within the cluster configuration. Job Scheduler functionality is available only if a valid data source is defined in the `ClusterMBean.DataSourceForJobScheduler` attribute. For information about how to use the WebLogic Server Administration Console to configure this attribute, see Configure a data source for a job scheduler in the *Oracle WebLogic Server Administration Console Online Help*.

The following `config.xml` excerpt shows 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>
```

Leasing

Leasing must be enabled for Job Schedulers. You can use either high-availability database leasing or non-database consensus leasing. When using high-availability database leasing, you must create the leasing table in the database.

Schemas for creating this table are in the following location:

WL_HOME/server/db/*dbname*/leasing.ddl

In the preceding path, *dbname* represents the name of the database.

See Leasing in *Administering Clusters for Oracle WebLogic Server*.

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 shows 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
```

Canceling Jobs

You can cancel jobs programmatically or by using the WebLogic Server Administration Console.

To cancel a job programmatically, invoke the `cancel` method of the job's corresponding JobRuntimeMBean. You can access a JobRuntimeMBean using either of the following ways:

- Invoke `JobSchedulerRuntimeMBean.getJob(id)` with the ID of a scheduled job. To get the ID, invoke the `JobScheduler.schedule` method to return a Timer object, then use the Timer's `toString` method to return the ID.
- Invoke `JobSchedulerRuntimeMBean.getExecutedJobs()` to return an array of `JobRunTimes` for all jobs that have been executed at least once.

You cannot invoke the `cancel` method to cancel a scheduled job that has not executed at least once.

For information about how to use the WebLogic Server Administration Console to cancel jobs, see Cancel jobs in *Oracle WebLogic Server Administration Console Online Help*.

Debugging

The following debugging flags enable more verbose output:

```
-Dweblogic.debug.DebugSingletonServices=true -Dweblogic.JobScheduler=true
```

Unsupported Methods and Interfaces

The following methods and interfaces in the `commonj.timer` package are not supported in the Job Scheduler environment:

- `CancelTimerListener` interface
- `StopTimerListener` interface
- The following methods of the `TimerManager` interface:
  - `suspend`
  - `resume`
  - `scheduleAtFixedRate`
  - `stop`
  - `waitForStop`
  - `waitForSuspend`

Work Manager API

The Work Manager API, `commonj.work`, provides a set of interfaces that allows an application to execute 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 choice 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 is lost.

Work Manager Interfaces

This section summarizes the interfaces in the Work Manager API. For details about using these interfaces, see `commonj.work` in the *Java API Reference for Oracle WebLogic Server*.

The Work Manager API contains the following interfaces:

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

  A WorkManager is 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. 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 about implementing a `WorkManager` within an application, see `WorkManager` in the *Java API Reference for Oracle WebLogic Server*.

For information about JNDI, see *Developing JNDI Applications for Oracle WebLogic Server*.

- **Work** - 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** - Determines the status of a completed `Work` instance. A `WorkItem` is returned by a `WorkManager` after a `Work` instance has been submitted to that `WorkManager`.

  See `Work` in the *Java API Reference for Oracle WebLogic Server*.
- **WorkListener** - Provides communication between the `WorkManager` and the scheduled work defined within the `Work` instance. `WorkListener` is a callback interface.

  You can use `WorkListener` to determine the current status of the `Work` item. See `WorkListener` in the *Java API Reference for Oracle WebLogic Server*.

  Note:

  `WorkListener` instances are always executed in the same JVM as the original thread used to schedule the `Work` by means of the `WorkManager`.

- **WorkEvent** - A `WorkEvent` is sent to a `WorkListener` as `Work` is processed by a `WorkManager`.

  See `WorkEvent` in the *Java API Reference for Oracle WebLogic Server*.
- **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 of a cluster.

  See `RemoteWorkItem` in the *Java API Reference for Oracle WebLogic Server*.

Work Manager Deployment

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

The following deployment descriptor snippet shows the configuration of 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`.

Automatic Binding of the Default CommonJ Work Manager

Automatic binding of the default CommonJ Work Manager to `java:comp/env/wm/default` has been removed in WebLogic Server 12.2.1.

If you have an application that attempts to use the default CommonJ Work Manager, you can either:

- Add a resource-ref entry for `wm/default` in a deployment descriptor. For example:

```
<resource-ref>
      <res-ref-name>wm/default</res-ref-name>
      <res-type>commonj.work.WorkManager</res-type>
      <res-auth>Container</res-auth>
</resource-ref>
```

- Have the CommonJ Work Manager injected into the application component. For example:

```
@Resource commonj.work.WorkManager myWorkManager;
```

Work Manager Example

The following example shows 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");}

    }
```