

Scheduler Guide

JEUS 9

TMAXSOFT

Copyright

Copyright 2025. TmaxSoft Co., Ltd. All Rights Reserved.

Company Information

TmaxSoft Co., Ltd.

TmaxSoft Tower 10F, 45, Jeongjail-ro, Bundang-gu, Seongnam-si, Gyeonggi-do, South Korea

Website: <https://www.tmaxsoft.com/en/>

Restricted Rights Legend

All TmaxSoft Software (JEUS®) and documents are protected by copyright laws and international convention. TmaxSoft software and documents are made available under the terms of the TmaxSoft License Agreement and this document may only be distributed or copied in accordance with the terms of this agreement. No part of this document may be transmitted, copied, deployed, or reproduced in any form or by any means, electronic, mechanical, or optical, without the prior written consent of TmaxSoft Co., Ltd. Nothing in this software document and agreement constitutes a transfer of intellectual property rights regardless of whether or not such rights are registered) or any rights to TmaxSoft trademarks, logos, or any other brand features.

This document is for information purposes only. The company assumes no direct or indirect responsibilities for the contents of this document, and does not guarantee that the information contained in this document satisfies certain legal or commercial conditions. The information contained in this document is subject to change without prior notice due to product upgrades or updates. The company assumes no liability for any errors in this document.

Trademarks

JEUS® is registered trademark of TmaxSoft Co., Ltd.

Java, Solaris are registered trademarks of Oracle Corporation and its subsidiaries and affiliates.

Microsoft, Windows, Windows NT are registered trademarks or trademarks of Microsoft Corporation.

HP-UX is a registered trademark of Hewlett Packard Enterprise Company.

AIX is a registered trademark of International Business Machines Corporation.

UNIX is a registered trademark of X/Open Company, Ltd.

Linux is a registered trademark of Linus Torvalds.

Noto is a trademark of Google Inc. Noto fonts are open source. All Noto fonts are published under the SIL Open Font License, Version 1.1. (<https://www.google.com/get/noto/>)

Other products and company names are trademarks or registered trademarks of their respective

owners.

The names of companies, systems, and products mentioned in this manual may not necessarily be indicated with a trademark symbol (™, ®).

Open Source Software Notice

Some modules or files of this product are subject to the terms of the following licenses: APACHE2.0, CDDL1.0, EDL1.0, OPEN SYMPHONY SOFTWARE1.1, TRILEAD-SSH2, Bouncy Castle, BSD, MIT, SIL OPEN FONT1.1

Detailed Information related to the license can be found in the following directory:
\${INSTALL_PATH}/license/oss_licenses

Document History

Product Version	Guide Version	Date	Remarks
JEUS 9	3.1.2	2025-03-24	-
JEUS 9	3.1.1	2024-12-24	-

Contents

1. Introduction	1
1.1. Overview	1
1.2. Scheduler Component Structure	1
1.3. Scheduler Server	2
2. Scheduling Tasks	3
2.1. Overview	3
2.2. JEUS Scheduler Class	4
2.3. Defining Tasks	5
2.3.1. Implementing ScheduleListener Interface	5
2.3.2. Schedule Class Inheritance	6
2.3.3. Inheriting RemoteSchedule Class	6
2.4. Obtaining Scheduler Object	7
2.4.1. Obtaining Scheduler Object in Local Environment	7
2.4.2. Obtaining a Scheduler Object in Remote Environment	8
2.5. Registering Tasks	8
2.5.1. Defining One-Time Tasks	9
2.5.2. Defining Recurring Tasks	9
2.5.3. Registering Schedule Task Objects	11
2.6. Controlling Tasks	11
2.7. Using Scheduler	12
2.7.1. Using In a Standalone Environment	12
2.7.2. Using on JEUS Server	13
2.7.3. Using in Jakarta EE Components	15
2.8. Using Job-list	16
3. Configuring JEUS Scheduler	17
3.1. Overview	17
3.2. Configuring JEUS Scheduler on the Server	17
3.3. Configuring Job-list	18
3.4. Configuring a Thread Pool	20
3.4.1. Shared Thread Pool	20
3.4.2. Dedicated Thread Pool	21
3.5. Configuring a Client Container	21

1. Introduction

This chapter describes how to use JEUS Scheduler.

1.1. Overview

JEUS Scheduler is used to schedule a one-time or recurring task.

JEUS Scheduler is an extended feature of JEUS. Although JEUS Scheduler is similar to the timer service of EJB, they are different. While an EJB Timer Service can be used only in the EJB environment, JEUS Scheduler can be used in any environment including the EJB environment and in Java SE applications in general. For example, you can use JEUS Scheduler to periodically delete temporary files and monitor database connections for system management.

Because you cannot use Java SE Timer (`java.util.Timer`) directly in the Jakarta EE environment, you need to use EJB Timer Service or JEUS Scheduler instead. While EJB Timer Service can be used only in an EJB environment, JEUS Scheduler can be used in all Jakarta EE environments as well as general Java SE applications.

If you are familiar with Java SE Timer (`java.util.Timer`), you can easily learn how to use JEUS Scheduler because of their similarities. Note that JEUS Scheduler provides end time and max count settings that are not offered by Java SE Timer.

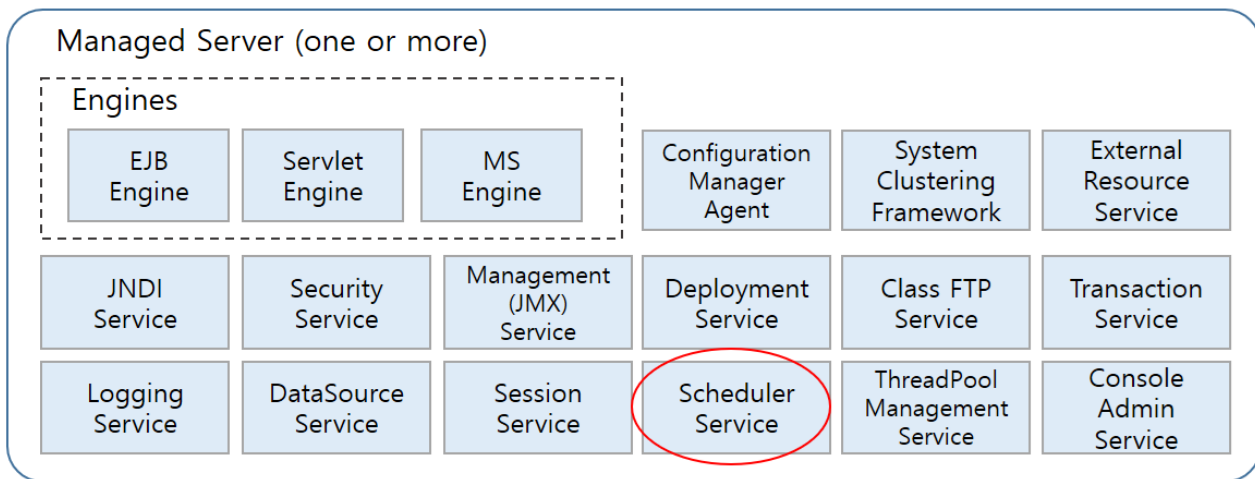
The following are supported conditions for using JEUS Scheduler and its features:

- As a standalone scheduler in Java SE applications.
- As a standalone scheduler in Jakarta EE application clients.
- Connect to JEUS server scheduler service from remote environments.
- Register jobs in the JEUS server configuration file for use in the JEUS server.
- Use JEUS server scheduler service from Jakarta EE components such as servlet, JSP, and EJB.

1.2. Scheduler Component Structure

JEUS Scheduler can be used both in user applications and on a JEUS server. A JEUS server can start the scheduler service and schedule tasks remotely using the configuration file.

The following figure shows JEUS Scheduler as a component of the JEUS server.

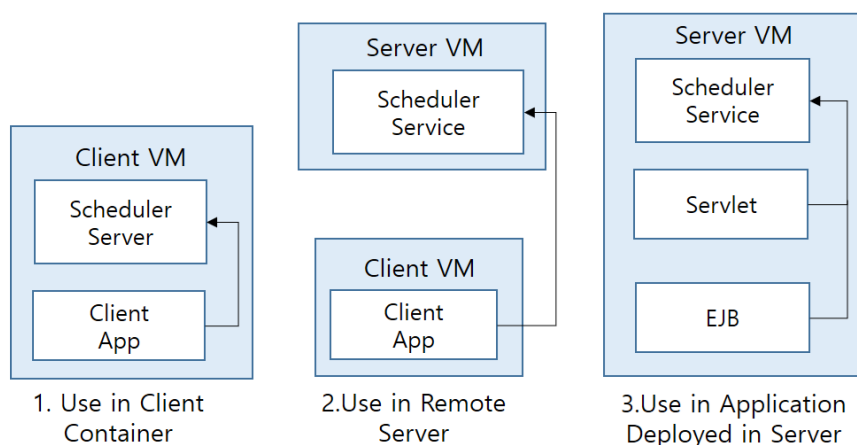


The Scheduler Component in JEUS

1.3. Scheduler Server

Scheduler's service and execution method vary according to the server type.

The following figure shows the execution method by Scheduler Server.



Execution Methods by Scheduler Server

The following describes each Scheduler Service type.

- Server Scheduler Service

It is used for a recurring task. A recurring task on the server may be registered in the Job-list and executed when the server starts.

It is usually used for tasks that periodically execute Jakarta EE components such as a servlet, JSP, and EJB. A task can be registered in the JNDI repository for remote client use.

- Client Scheduler Service

The JEUS Scheduler Service runs in a standalone mode in the application client. It is normally used to periodically execute a task in an application client. It is also used for tasks that no longer need to be executed after the client is terminated.

2. Scheduling Tasks

This chapter outlines basic concepts related to JEUS Scheduler and instructions for JEUS Scheduler programming.

2.1. Overview

For JEUS 5 and later, the following features have been added or changed. The JEUS 9 Scheduler is compatible with its earlier versions, and programs written with previous versions can still be used without modification.

- Add SchedulerListener Interface

The ScheduleListener interface defines a task.

This is the top level interface that must be implemented by all tasks. It also implements the existing schedule task classes. Therefore, tasks can be defined by implementing ScheduleListener without inheriting a schedule task class.

- Add SchedulerFactory Class

The existing SchedulerManager class for client and server is replaced by the SchedulerFactory class. You can use SchedulerFactory to get Scheduler objects, and register tasks regardless of environment (client, server, and remote client environments).

- Add various methods

Various methods for adding a task have been added. You can specify a start time, cycle, termination time, and maximum execution count properties when adding a task.

- Modify into a multi-thread execution method that uses a thread pool

While the existing JEUS Scheduler used a single thread to execute tasks, the new JEUS Scheduler executes each task with a separate thread using a thread pool. This prevents a blocked task from affecting other tasks.

- Add Job-list feature

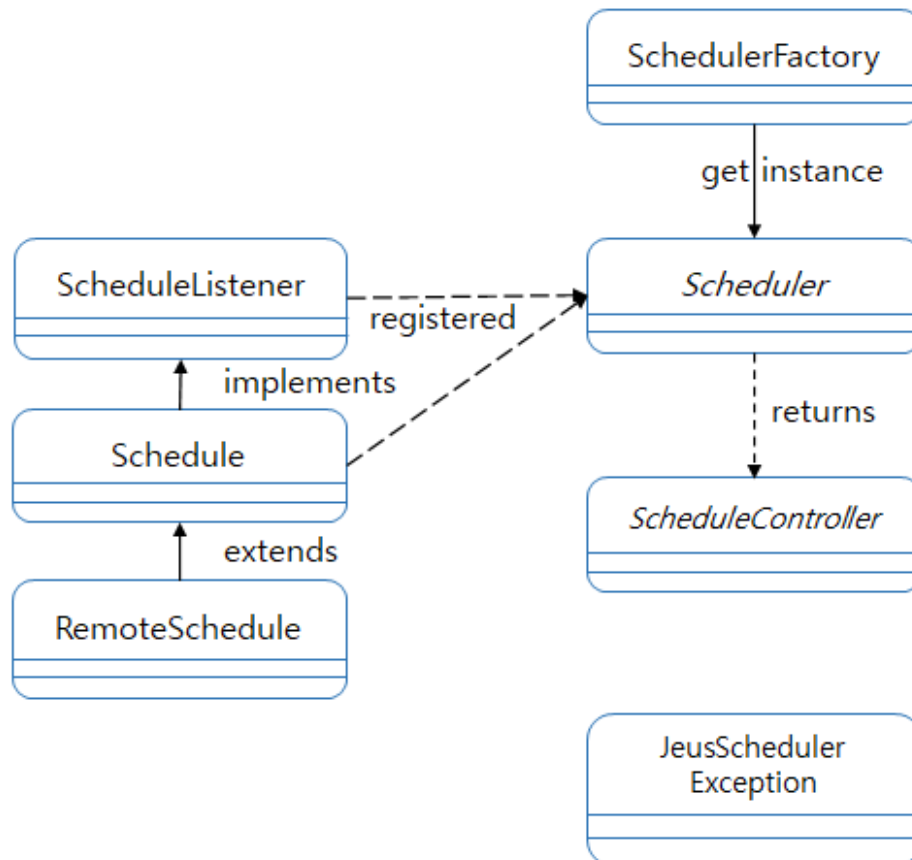
Job-list feature allows tasks to be registered on JEUS server configuration instead of through programming.

The next section describes the classes that make up the JEUS Scheduler, and how to define, register, and control a job schedule. All examples explained in this chapter can be found in the `JEUS_HOME/samples/scheduler` directory.

2.2. JEUS Scheduler Class

JEUS Scheduler is conceptually identical to and contains similar interfaces of J2SE Timer. The `java.util.TimerTask` class that represents a task corresponds to the `jeus.schedule.ScheduleListener` interface, and the `java.util.Timer` class that registers a task corresponds to the `jeus.schedule.Scheduler` interface. So if you are familiar with J2SE Timer interface, you can use JEUS Scheduler without difficulty.

The following figure shows each Scheduler API classes.



Scheduler API Classes

API Class	Description
package <code>jeus.schedule</code>	All classes and interfaces of JEUS Scheduler belong to <code>jeus.schedule</code> package and its sub-packages.
interface <code>ScheduleListener</code>	A task that needs to execute at a set time is defined as a class by implementing the <code>ScheduleListener</code> interface. <code>ScheduleListener</code> contains a callback method called <code>onTime()</code> that is invoked at the scheduled time.
abstract class <code>Schedule</code>	Abstract class that implements <code>ScheduleListener</code> which was used in versions earlier than JEUS 5. This class contains the <code>nextTime()</code> and <code>onTime()</code> callback methods. The <code>nextTime()</code> method is used dynamically to get the time to execute the task after it is scheduled
abstract class <code>RemoteSchedule</code>	Special abstract class that contains the <code>initialize()</code> callback method and receives initialization parameters when creating an object.
class <code>SchedulerFactory</code>	Calls a concrete <code>Scheduler</code> object.

API Class	Description
interface Scheduler	Core interface for registering tasks in JEUS Scheduler. It defines various registerSchedule() methods.
interface ScheduleController	When a task is registered in the Scheduler, the object that implements the ScheduleController interface is returned. This handler object is used to get information about or cancel the task.
exception JeusSchedulerException	JeusSchedulerException may be thrown if an internal error occurs while registering or canceling a task in JEUS Scheduler.



For more information about each interface or class, refer to the JEUS Scheduler Javadoc API.

2.3. Defining Tasks

To execute a task, you must first define its class and then the task itself by specifying its execution time and cycle.

2.3.1. Implementing ScheduleListener Interface

A task that executes at a set time is defined by first implementing the ScheduleListener interface.

ScheduleListener contains the onTime() callback method that is invoked at the set time. A task class must implement the onTime() method that contains the task implementation.

Task object example

```
public class SimpleTask implements ScheduleListener {
    private String name;
    private int count;

    // no-arg constructor is required if classname is used for task registration
    public SimpleTask() {
    }

    public SimpleTask(String name) {
        this.name = name;
    }

    public void onTime() {
        count++;
        echo("##### " + name + " is waked on " + new Date());
    }

    ...
}
```

2.3.2. Schedule Class Inheritance

Instead of directly implementing a `ScheduleListener`, you can define it by inheriting the `Schedule` class or `RemoteSchedule` class. The `Schedule` and `RemoteSchedule` classes were used in earlier versions prior to JEUS 5. Starting from JEUS 5, `ScheduleListener` must be implemented to define tasks in general, but the previous task classes are also provided to ensure backward compatibility.

The `Schedule` abstract class contains the `nextTime()` callback method in addition to the `onTime()` method. The `nextTime()` method does not schedule the call time when registering tasks, but instead the next call time is scheduled in the task class. Therefore, it is more efficient to use the method for tasks with a variable time cycle than a fixed time cycle.

After registering a task object, JEUS Scheduler determines the initial execution time by calling `nextTime()`. At the set time, `onTime()` is called and the task is executed when `onTime()` call terminates. The `nextTime()` method must return the next scheduled execution interval in milliseconds. If it returns 0, the task will no longer be executed.

Because `onTime()` must be finished before the `Schedule` task object calls `nextTime()`, while `onTime()` is executing, the `nextTime()` call is delayed. But it is difficult to write programs to execute the task at exact time intervals. Thus, when you are registering a task, it is recommended to set the execution cycle instead of using the `nextTime()` method.

Schedule object example

```
public class SimpleSchedule extends Schedule {
    private String name;
    private int count;
    private long period = 2000; // 2 seconds

    // no-arg constructor is required if classname is used for task registration
    public SimpleSchedule() {
    }

    public SimpleSchedule(String name) {
        this.name = name;
    }

    public void onTime() {
        count++;
        echo("##### " + name + " is waked on " + new Date());
    }

    public long nextTime(long currentTime) {
        return currentTime + period;
    }

    ...
}
```

2.3.3. Inheriting RemoteSchedule Class

The `RemoteSchedule` class is a `Schedule` object that can set the initialization parameters when registering tasks remotely. You can register a remote task object with the class name. The class

contains the `initialize()` Callback method which is called just once after a task is registered. You can use this method to assign initialization values when using the class name to register a remote task.

Note that the `initialize()` callback can be invoked only when `RemoteSchedule` task object is registered by using the `Scheduler.registerSchedule(classname, hashtable, daemon_flag)` method.

RemoteSchedule object example

```
public class SimpleRemoteSchedule extends RemoteSchedule {
    private String name;
    private int count;
    private long period;

    // no-arg constructor is required if classname is used for task registration
    public SimpleRemoteSchedule() {
    }

    // this is called by scheduler after creation
    public void initialize(Hashtable parameters) {
        name = (String) parameters.get("name");
        Long interval = (Long) parameters.get("interval");
        if (interval != null)
            period = interval.longValue();
        else
            period = 2000;
    }

    public void onTime() {
        count++;
        echo("##### " + name + " is waked on " + new Date());
    }

    public long nextTime(long currentTime) {
        return currentTime + period;
    }
    ...
}
```



When registering a task by using the Job-list or an API that uses the class name, a no-arg (default) constructor is required because the corresponding class is initialized by the container.

2.4. Obtaining Scheduler Object

This section describes how to obtain a Scheduler object. JEUS Scheduler operates in both local and remote environments.

2.4.1. Obtaining Scheduler Object in Local Environment

When JEUS Scheduler runs in a local environment, a Scheduler instance is created on a local Java

Virtual Machine (JVM) and all registered tasks run on the same JVM.

In a local environment, a single instance of JEUS Scheduler is created. This instance is the default scheduler that is shared by all clients on the JVM. JEUS Scheduler in a local environment is used by regular J2SE applications, Jakarta EE application clients, and Jakarta EE components such as servlet, JSP, and EJB. Use SchedulerFactory to enable JEUS Scheduler in a local environment.

The default Scheduler instance can be easily obtained in the following way.

```
// Get the default scheduler
Scheduler scheduler = SchedulerFactory.getDefaultScheduler();
```

2.4.2. Obtaining a Scheduler Object in Remote Environment

When JEUS Scheduler runs in a remote environment, a Scheduler instance is created on a remote JVM, and all registered tasks run on the remote JVM.

Because the remote Scheduler is in the form of an RMI object, JEUS Scheduler can be used through an RMI call. In a JEUS environment, the remote Scheduler service runs on the JEUS server. A remote JEUS Scheduler is used when a remote client registers a task on a JEUS node.

To access the remote JEUS Scheduler on a JEUS node, a JNDI lookup is used.

A JEUS node Scheduler instance (Stub) can be obtained in the following way.

```
// Get the remote scheduler
InitialContext ic = new InitialContext();
Scheduler scheduler = (Scheduler)ic.lookup(Scheduler.SERVER_SCHEDULER_NAME);
```



`jeus.schedule.server.SchedulerManager` and `jeus.schedule.client.SchedulerManager` used in JEUS 5 and earlier are no longer used (deprecated). Instead, it is recommended to use Scheduler objects generated by the SchedulerFactory. The deprecated classes, however, are provided for backward compatibility.

2.5. Registering Tasks

This section describes how to register tasks using the Scheduler interface. The methods for registering tasks are the same in both local and remote Schedulers. The only difference is that for a remote JEUS Scheduler, the task objects are serialized and run remotely.

2.5.1. Defining One-Time Tasks

You can register a task to execute only once by specifying the execution time. The execution time can be set as an absolute time by using the `java.util.Date` object or as a relative time based on the current time in milliseconds.

The following shows how to register a task.

```
registerSchedule(ScheduleListener task, Date time, boolean isDaemon)
registerSchedule(ScheduleListener task, long delay, boolean isDaemon)
```

The following shows how to use the method. For more information on method parameters, refer to [Defining Recurring Tasks](#).

```
SimpleTask task1 = new SimpleTask("task1");
Date firstTime1 = new Date(System.currentTimeMillis() + 2000);
ScheduleController handle1 = scheduler.registerSchedule(task1, firstTime1, false);
```

2.5.2. Defining Recurring Tasks

You can register a recurring task by specifying the initial execution time, cycle, termination time, and the maximum number of executions. Select either the fixed-delay or fixed-rate method according to the task.

- **Fixed-delay Type**

The execution interval is fixed. The next execution time is set according to the previous execution time and cycle. If a task execution is delayed (due to long execution time or external reasons such as garbage collection) and the next execution time passes, the next task is executed immediately, but the following tasks are all delayed. As a result, all task executions are delayed.

- **Fixed-rate Type**

The ratio of task execution is fixed. The next execution time is determined according to the initial execution time and cycle. Even though a task execution is delayed, the next task is executed immediately after the current task to maintain the rate of executions per hour. In the long term, the task execution times are maintained according to the specified cycle.



When executing a fixed-rate task is delayed, JEUS Scheduler calls the task using another thread to maintain the execution time. So even if executing a task is delayed, the task may be executed concurrently by another thread. To ensure this process, you should check if the task objects are thread safe.

The following methods show how to register a task.

```

registerSchedule(ScheduleListener task, Date firstTime, long period, Date endTime,
                long maxcount, boolean isDaemon)

registerSchedule(ScheduleListener task, long delay, long period, Date endTime,
                long maxcount, boolean isDaemon)

registerScheduleAtFixedRate(ScheduleListener task, Date firstTime, long period,
                            Date endTime, long maxcount, boolean isDaemon)

registerScheduleAtFixedRate(ScheduleListener task, long delay, long period,
                            Date endTime, long maxcount, boolean isDaemon)

```

The following describes each parameter.

Parameter	Description
Date firstTime	Initial execution time.
long delay	Next execution time relative to the current time (Unit: milliseconds).
long period	Recurring execution cycle (Unit: milliseconds).
Date endTime	Termination time after which a task is no longer executed. If set to null, the recurring task executes indefinitely.
long maxcount	Maximum number of executions. If set to Scheduler.UNLIMITED, there is no constraint.
boolean isDaemon	Used only if the Schedule is registered remotely. If set to true, the task terminates when connection to a client is lost. A client decides whether a connection is lost according to the distributed garbage collection (DGC) policy of RMI Runtime. Schedule is canceled after 15 minutes from the time the client is disconnected.
boolean isThreaded	Deprecated.

The following example shows how to register tasks.

Defining Recurring Tasks

```

SimpleTask task2 = new SimpleTask("task2");
ScheduleController handle2 = scheduler.registerSchedule(task2, 2000, 2000, null,
                Scheduler.UNLIMITED, false);

SimpleTask task3 = new SimpleTask("task3");
Date firstTime3 = new Date(System.currentTimeMillis() + 2000);
Date endTime3 = new Date(System.currentTimeMillis() + 10 * 1000);
ScheduleController handle3 = scheduler.registerScheduleAtFixedRate(task3,
                firstTime3, 2000, endTime3, 10, false);

```

2.5.3. Registering Schedule Task Objects

The function for registering Schedule or RemoteSchedule task objects is provided for backward compatibility.

Because you can specify the initial and recurring execution times of a task with the `nextTime()` method of the Schedule task object, additional parameters are not required. The following method show how to define a task.

```
registerSchedule(Schedule task, boolean isDaemon)
registerSchedule(String classname, Hashtable params, boolean isDaemon)
```

The following example shows how to register tasks.

Registering Schedule Task Objects

```
Hashtable params = new Hashtable();
params.put("name", "task3");
params.put("interval", new Long(3000));
ScheduleController handle3 = scheduler.registerSchedule(
    "samples.scheduler.SimpleRemoteSchedule", params, true);

SimpleSchedule task4 = new SimpleSchedule("task4");
ScheduleController handle4 = scheduler.registerSchedule(task4, true);
```



Scheduler must be configured correctly since it is a main service provided by a Managed Server (MS). If an MS fails to create the Scheduler, it displays an error and fails to start.

2.6. Controlling Tasks

When you register a task in JEUS Scheduler, the ScheduleController handler object is returned. This object is created for each defined task for you to control the task. You can use the handler to get information about a task or to cancel it.

The following example shows how to cancel a task by calling the `ScheduleController.cancel()` method.

Calling the ScheduleController.cancel Method

```
SimpleTask task2 = new SimpleTask("task2");
ScheduleController handle2 = scheduler.registerSchedule(task2, 2000, 2000, null,
    Scheduler.UNLIMITED, false);

Thread.sleep(10 * 1000);
handle2.cancel();
```

2.7. Using Scheduler

You can use JEUS Scheduler after defining a task, obtaining the Scheduler object, registering a task, and setting the task control handler.

2.7.1. Using In a Standalone Environment

You can use JEUS Scheduler within regular J2SE applications or Jakarta EE application clients. In these cases, JEUS Scheduler and J2SE Timer can be used as libraries. The SchedulerFactory class is used to obtain the Scheduler object. A daemon flag can be set to any value because it is not used when tasks are registered in a local environment

The following is an example of a standalone client:

Using Scheduler on a Standalone Client

```
public class StandAloneClient {
    public static void main(String args[]) {
        try {
            // Get the default scheduler
            Scheduler scheduler = SchedulerFactory.getDefaultScheduler();

            // Register SimpleTask which runs just one time
            echo("Register task1 which runs just one time...");
            SimpleTask task1 = new SimpleTask("task1");
            Date firstTime1 = new Date(System.currentTimeMillis() + 2000);
            ScheduleController handle1 = scheduler.registerSchedule(task1,
                firstTime1, false);

            Thread.sleep(5 * 1000);
            echo("");

            // Register SimpleTask which is repeated
            // with fixed-delay
            echo("Register task2 which is repeated " + "until it is canceled...");
            SimpleTask task2 = new SimpleTask("task2");
            ScheduleController handle2 = scheduler.registerSchedule(task2, 2000,
                2000, null, Scheduler.UNLIMITED, false);

            Thread.sleep(10 * 1000);
            handle2.cancel();
            echo("");

            // Register SimpleTask which is repeated
            // with fixed-rate
            echo("Register task3 which is repeated " + "for 10 seconds...");
            SimpleTask task3 = new SimpleTask("task3");
            Date firstTime3 = new Date(System.currentTimeMillis() + 2000);
            Date endTime3 = new Date(System.currentTimeMillis() + 10 * 1000);
            ScheduleController handle3 = scheduler.registerScheduleAtFixedRate(
                task3, firstTime3, 2000, endTime3, 10, false);

            Thread.sleep(12 * 1000);
            echo("");
        }
    }
}
```



```

// Register SimpleSchedule which is repeated
// every 2 seconds
echo("Register task4 which is repeated " + "every 2 seconds...");
// SimpleSchedule class extends jeus.schedule.Schedule
// and has 2-seconds delayed scheduling logic in "nextTime" method.
SimpleSchedule task4 = new SimpleSchedule("task4");
ScheduleController handle4 = scheduler.registerSchedule(task4, false);

Thread.sleep(10 * 1000);
echo("");

// Cancel all tasks
echo("Cancel all tasks registered on the scheduler...");
scheduler.cancel();
Thread.sleep(5 * 1000);

System.out.println("Program terminated.");
} catch (Exception e) {
    e.printStackTrace();
}
}

private static void echo(String s) {
    System.out.println(s);
}
}

```

If JEUS Scheduler is used within Jakarta EE application clients, JEUS Scheduler's thread pool can be configured in the deployment descriptor (jeus-client-dd.xml).



1. Settings related to the Scheduler thread pool, refer to "Thread Management Commands" in JEUS Reference Guide.
2. To compile or execute source code that uses Scheduler, JEUS related classes such as jeus.jar must be specified in the class path.

2.7.2. Using on JEUS Server

If the Scheduler Service startup is configured on the JEUS server, Scheduler Service running on the server can be accessed by remote clients.

Scheduler Service on JEUS server registers the RMI Scheduler object with JNDI. Therefore, clients can obtain the remote scheduler objects (stub object) using a JNDI lookup. The name of the object is "jeus_service/Scheduler", and it uses the Scheduler.SERVER_SCHEDULER_NAME constant.

After obtaining the Scheduler object, you can register it with Scheduler Service. Note the registered objects are serialized and run from remote Schedulers, which means they run on remote JEUS servers. When you are registering a task, if you set the daemon flag to true, the remote tasks terminate when the remote clients terminate.



For information about configuring the Scheduler Service to start on the JEUS server, refer to "Thread Management Commands" in JEUS Reference Guide.

The following is an example of a remote client:

Using Scheduler on a Remote Client

```
public class RemoteClient {
    public static void main(String args[]) {
        try {
            // Get the remote scheduler
            InitialContext ic = new InitialContext();
            Scheduler scheduler = (Scheduler)ic.lookup(
                Scheduler.SERVER_SCHEDULER_NAME);

            // Register SimpleTask which runs just one time
            echo("Register task1 which runs just one time...");
            SimpleTask task1 = new SimpleTask("task1");
            Date firstTime1 = new Date(System.currentTimeMillis() + 2000);
            ScheduleController handle1 = scheduler.registerSchedule(task1,
                firstTime1, true);

            Thread.sleep(5 * 1000);
            echo("");

            // Register SimpleTask which is repeated
            // with fixed-delay
            echo("Register task2 which is repeated " + "until it is canceled...");
            SimpleTask task2 = new SimpleTask("task2");
            ScheduleController handle2 = scheduler.registerSchedule(task2,
                2000, 2000, null, Scheduler.UNLIMITED, true);

            Thread.sleep(10 * 1000);
            handle2.cancel();
            echo("");

            // Register SimpleRemoteSchedule which is repeated
            // every 3 seconds
            echo("Register task3 which is repeated " + "every 3 seconds...");
            Hashtable params = new Hashtable();
            params.put("name", "task3");
            params.put("interval", new Long(3000));
            // SimpleRemoteSchedule class extends
            // jeus.schedule.RemoteSchedule
            // and has 3-seconds delayed scheduling logic in "nextTime" method.
            ScheduleController handle3 = scheduler.registerSchedule(
                "samples.scheduler.SimpleRemoteSchedule", params, true);

            Thread.sleep(10 * 1000);
            echo("");

            // Cancel all tasks
            echo("Cancel all tasks registered on the scheduler...");
            scheduler.cancel();
            Thread.sleep(5 * 1000);

            System.out.println("Program terminated.");
        }
    }
}
```

```

        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    private static void echo(String s) {
        System.out.println(s);
    }
}

```



To execute the previous example, you must compress relevant task class files into a JAR file and save them in the 'DOMAIN_HOME/lib/application' or 'SERVER_HOME/lib/application' directory. The JEUS server loads the class files to execute the task, and if JEUS is running, it must be restarted.

2.7.3. Using in Jakarta EE Components

JEUS Scheduler can be used in Jakarta EE components such as EJB and servlet. In these environments, JEUS Scheduler runs on the JEUS server.

The EJB 2.1 standard specifications include the EJB Timer Service. Therefore, to comply with the Jakarta EE standard, it is recommended to use the EJB Timer Service for EJB components instead of JEUS Scheduler. But for Jakarta EE components other than EJB components, you must use JEUS Scheduler because EJB Timer Service is not supported.

You can use JEUS Scheduler in Jakarta EE components with the same methods for using Standalone JEUS Scheduler. First, you use the SchedulerFactory class to obtain a Scheduler object running on the server, and then use a method to register the task.



You can configure a thread pool for JEUS Scheduler that runs on JEUS server. For information about the configuration, refer to [Thread Pool Configuration](#).

The following example shows how to use JEUS Scheduler in EJB.

Using JEUS Scheduler in EJB

```

public class HelloEJB implements SessionBean {
    private SimpleTask task;
    private ScheduleController taskHandler;
    private boolean isStarted;

    public HelloEJB() {
    }

    public void ejbCreate() {
        task = new SimpleTask("HelloTask");
        isStarted = false;
    }
}

```

```

public void trigger() throws RemoteException {
    if (!isStarted) {
        Scheduler scheduler = SchedulerFactory.getDefaultScheduler();
        taskHandler = scheduler.registerSchedule(task,
            2000, 2000, null, Scheduler.UNLIMITED, false);
        isStarted = true;
    }
}

public void ejbRemove() throws RemoteException {
    if (isStarted) {
        taskHandler.cancel();
        isStarted = false;
    }
}

public void setSessionContext(SessionContext sc) {
}

public void ejbActivate() {
}

public void ejbPassivate() {
}
}

```

```

public class HelloClient {
    public static void main(String args[]) {
        try {
            InitialContext ctx = new InitialContext();

            HelloHome home = (HelloHome) ctx.lookup("helloApp");
            Hello hello = (Hello) home.create();
            hello.trigger();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

2.8. Using Job-list

You can register a task on the JEUS server by using the Job-list setting in the configuration file instead of coding. The Job-list setting can be configured for JEUS Scheduler. Tasks you register with Job-list are executed at a fixed rate.



The Job-list setting can be configured for JEUS Scheduler that runs on JEUS server. For more information about how to configure the Job-list, refer to [Job-list Configuration](#).

3. Configuring JEUS Scheduler

This chapter describes how to configure JEUS Scheduler in a JEUS configuration file or deployment descriptor (DD).

3.1. Overview

The following configuration is required to use JEUS Scheduler Service.

- **Configuring Scheduler Service on JEUS server**

To remotely access the JEUS server Scheduler Service or execute a recurring task on JEUS server using the Job-list, the Scheduler Service must be activated on the server. To activate the service, you need to add the setting in the domain.xml file. Then, the server starts Scheduler Service at startup. The following is an outline of the configuration required.

- Enable Scheduler Service.
- Configure the thread pool for Scheduler Service.
- Configure Job-list scheduled to execute on the server.

- **Configuring Scheduler Service on the client container**

- Repeat the above settings in the deployment descriptor (DD) file.

Simply put, in order to use the scheduler, you need to activate the Scheduler Service, configure the thread pool, and finally, set the Job-list to execute the actual tasks.

3.2. Configuring JEUS Scheduler on the Server

This section describes how to configure the scheduler on the server by manually editing the domain.xml file.

To remotely access the JEUS server Scheduler Service or execute a recurring task on JEUS server using the Job-list, the Scheduler Service must be activated on the server. To activate the service, you need to add the setting in the domain.xml file. Then, the server starts Scheduler Service at startup.

The following are the steps for adding the job from [Schedule Class Inheritance](#) to the server.

JEUS domain settings

```
...
<server>
  ...
  <scheduler>
    <enabled>true</enabled>
    <pooling>
      <dedicated>
        <min>0</min>
        <max>10</max>
```

```

        <keep-alive-time>60000</keep-alive-time>
        <queue-size>4096</queue-size>
        <stuck-thread-handling>
            <max-stuck-thread-time>3600000</max-stuck-thread-time>
            <action-on-stuck-thread>None</action-on-stuck-thread>
            <stuck-thread-check-period>300000</stuck-thread-check-period>
        </stuck-thread-handling>
    </dedicated>
</pooling>
<job-list>
    <job>
        <name>SimpleTask</name>
        <class-name>java.sample.scheduler.SimpleTask</class-name>
        <begin-time>2024-08-26T10:30:00</begin-time>
        <end-time>2024-08-27T10:30:00</end-time>
        <interval>
            <hourly>1</hourly>
        </interval>
        <count>10</count>
    </job>
</job-list>
</scheduler>
...
</server>
...

```

For more information about the `<pooling>` tag in the `domain.xml` file above, refer to "Thread Pool Configuration" in JEUS Server Guide.

Also, for a description of the `<job-list>` entry, see [Configuring Job-list](#) below.



Scheduler configuration changes are not applied dynamically. The changes take effect after the server restarts.

When the server restarts, the new job executes and the result can be checked in the server logs.

```

##### waked on Mon Aug 26 10:30:00 KST 2024
##### waked on Mon Aug 26 11:30:00 KST 2024

```

3.3. Configuring Job-list

This section describes how to register a job.

You can register a job or schedule it at JEUS server startup in the JEUS configuration file. A job is a single task unit that is scheduled. A job class must implement the `jeus.schedule.ScheduleListener` interface, and the class and its related classes must be compressed into a JAR file and saved in the following location.

- Since the following directory is not synchronized between the Master Server and a Managed

Server (hereafter MS), you must synchronize them manually for every machine to run the scheduler service.

```
DOMAIN_HOME/lib/application
```

- You must create the following directory.

```
SERVER_HOME/lib/application
```

JEUS Scheduler Service executes jobs that are configured on the server.

The following items must be configured.

Item	Description
Name	Specifies the name of a Job.
Class Name	Specifies the fully qualified name of the class that performs the job.
Description	Specifies the job description.
Begin Time	<p>Initial time to execute the task. If the time is not set, the task executes when the JEUS server starts.</p> <ul style="list-style-type: none">◦ Type: XML dateTime type◦ Format: yyyy-mm-ddThh:mm:ss (e.g. 2024-08-26T10:30:00) <p>If the Begin Time of a registered job has already passed, the initial execution time is adjusted to a time after the current time.</p>
End Time	<p>Time when the task terminates. If the time is not set, the task does not terminate.</p> <ul style="list-style-type: none">◦ Type: XML dateTime type◦ Format: yyyy-mm-ddThh:mm:ss (e.g. 2024-08-27T10:30:00) <p>If End Time has already passed, the task does not execute at all.</p>
Interval	<p>Specifies the period at which the Job is performed. You can select from the options below.</p> <ul style="list-style-type: none">◦ millisecond◦ minutely◦ hourly◦ daily
Count	Maximum number of executions. If Count is not set or set to -1, there is no limit for the number of executions.



If a task is registered in the Job-list, the task executes at a fixed rate and therefore maintains a stable execution cycle. But if executing a task is delayed, the task may be executed concurrently by another thread. To ensure this process, you must check if the task object is thread safe.

3.4. Configuring a Thread Pool

Scheduler Service controls a thread pool by adjusting the number of threads required for service execution. A thread pool can be either a **shared thread pool** that uses the system thread pool or a **dedicated thread pool** that needs to be created individually.

This section describes how to set up a thread pool using the console tool. For more information about each element, refer to "Thread Pool Configuration" in JEUS Server Guide.

3.4.1. Shared Thread Pool

If Scheduler Service uses a shared thread pool, you need to configure the number of threads only.

Using the console tool

The following shows how to set the thread pool of the scheduler using the console tool (jeusadmin).

```
[MASTER]domain1.adminServer>modify-system-thread-pool server1 -service scheduler -r 10
Successfully performed the MODIFY operation for The scheduler thread pool of the server (server1)..,
but all changes were non-dynamic. They will be applied after restarting.
Check the results using "show-system-thread-pool server1 -service scheduler or modify-system-thread-
pool server1 -service scheduler".
```

```
[MASTER]domain1.adminServer>show-system-thread-pool server1 -service scheduler
Shows the current configuration.
the system thread pool of the server (server1)
```

```
=====
+-----+-----+
| Min           | 0           |
| Max           | 100         |
| Keep-Alive Time | 300000      |
| Queue Size    | 4096        |
| Max Stuck Thread Time | 3600000    |
| Action On Stuck Thread | IGNORE_AND_REPLACE |
| Stuck Thread Check Period | 300000    |
| Reserved Threads for the Service transaction | 0          |
| Reserved Threads for the Service scheduler   | 10         |
| Reserved Threads for the Service namingserver | 0          |
+-----+-----+
=====
```


3.4.2. Dedicated Thread Pool

If Scheduler Service uses a dedicated thread pool, the thread pool can be configured using the console tool (jeusadmin).



The thread pool configuration of the Scheduler Service is dynamic and therefore the server does not need to be restarted. However, the server must be restarted when changing from using the shared thread pool to the dedicated thread pool for changes to take effect.

Using the console tool

The following shows how to configure the scheduler's thread pool using the console tool (jeusadmin).

```
[MASTER]domain1.adminServer>show-service-thread-pool server1 -service scheduler
Shows the current configuration.
=====
+-----+-----+
(No data available)
=====

[MASTER]domain1.adminServer>modify-service-thread-pool server1 -service scheduler -min 0 -max 20
Successfully performed the MODIFY operation for The scheduler thread pool of the server (server1).,
but all changes were non-dynamic. They will be applied after restarting.
Check the results using "show-service-thread-pool server1 -service scheduler or modify-service-
thread-pool server1 -service scheduler".

[MASTER]domain1.adminServer>show-service-thread-pool server1 -service scheduler
Shows the current configuration.
=====
+-----+-----+
| Min                | 0      |
| Max                | 20     |
| Keep-Alive Time    | 60000  |
| Queue Size         | 4096   |
| Max Stuck Thread Time | 3600000 |
| Action On Stuck Thread | NONE   |
| Stuck Thread Check Period | 300000 |
+-----+-----+
=====
```

3.5. Configuring a Client Container

This section describes how to configure JEUS Scheduler that is running on a client container when using a Jakarta EE application. Add the following **<scheduler>** element to the JEUS DD file, jeus-client-dd.xml.

Client Container Configuration: <jeus-client-dd.xml>

```
<?xml version="1.0"?>
```

```

<jeus-client-dd>
  <module-info>
    ...
  </module-info>
  ...
  <scheduler>
    <enabled>true</enabled>
    <!-- Scheduler Thread-pool settings -->
    <pooling>
      <dedicated>
        <min>2</min>
        <max>10</max>
        <keep-alive-time>60000</keep-alive-time>
        <queue-size>4096</queue-size>
        <stuck-thread-handler>
          <max-stuck-thread-time>3600000</max-stuck-thread-time>
          <action-on-stuck-thread>None</action-on-stuck-thread>
        </stuck-thread-handler>
      </dedicated>
    </pooling>
  </scheduler>
  ...
</jeus-client-dd>

```

To apply the changes after modifying the previous configuration file, the JEUS server does not need to be restarted, but client modules and client containers must be restarted.



1. A shared thread pool cannot be used when using Scheduler Service on a client container.
2. For client containers and application clients, refer to "JEUS Application Client Guide".

The following shows how to set jobs that run on client containers.

Job-list Configuration: <jeus-client-dd.xml>

```

<scheduler>
  ...
  <job-list>
    <job>
      <class-name>samples.ScheduleJob</class-name>
      <name>ScheduleJob</name>
      <description>This is a sample for scheduler service</description>
      <begin-time>2024-08-26T00:00:00</begin-time>
      <end-time>2024-08-27T00:00:00</end-time>
      <interval>
        <minutely>30</minutely>
      </interval>
      <count>-1</count>
    </job>
  </job-list>
</scheduler>

```