Concurrency Utilities for Java EE

Early Draft Preview

Document Version: Early Draft V. 0.1

Document Date: Apr. 4, 2006

Specification Leads:

Chris D. Johnson, IBM Corporation

Revanuru Naresh, BEA Systems, Inc

See the Concurreny Utilities for Java EE interest site (http://gee.cs.oswego.edu/dl/concurrencyee-interest/index.html) for instructions on how to comment on this draft.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Overview

Java™ Platform, Enterprise Edition (Java EE and formally known as J2EE™) server containers such as the enterprise bean or web component container do not allow using common Java SE concurrency APIs such as `java.lang.Thread` or `java.util.Timer` directly.

The server containers provide runtime support for Java EE application components (such as servlets and Enterprise JavaBeans™ (EJB™)). They provide a layer between application component code and platform services and resources. All application component code is run on a thread managed by a container and each container typically expects all access to container-supplied objects to occur on the same thread.

It is because of this behavior that application components are typically unable to reliably use other Java EE platform services from a thread that is not managed by the container. Java EE Product Providers (see chapter 2.10 of the Java 2 Enterprise Edition Specification) also discourage the use of resources in a non-managed way, because it can potentially undermine the enterprise features that the platform is designed to provide such as availability, security, and reliability and scalability.

This specification provides a simple, standardized API for using concurrency from Java EE application components without compromising the integrity of the container while still preserving the fundamental benefits of the Java EE platform.

1.2 Goals of this specification

This specification was developed with the following goals in mind:

- Preserve the fundamental capabilities and integrity of the Java EE platform.
- Utilize existing applicable Java EE platform services. Provide a simple yet flexible API for application component providers to design applications using concurrency design principles.
- Allow Java SE developers a simple migration path to the Java EE platform by providing consistency between the Java SE and Java EE platforms.
- Allow application component providers to easily add concurrency to existing Java EE applications.
- Support simple (common) and advanced concurrency patterns without sacrificing usability.

1.3 Other Java Platform Specifications

The following Java Platform specifications are referenced in this document:

- Concurrency Utilities Specification (JSR-166)
- J2EEConnector Architecture 1.5 (JSR-112)

- Java Platform Standard Edition
- Java Platform Standard Edition
- Java2 Platform, Enterprise Edition, Management Specification (JSR-77)
- Java Naming and Directory Interface™
- Java Transaction API
- Java Transaction Service
- JDBC™ API
- Java Message Service (JMS)

These specifications may be found at the Java web site: http://java.sun.com.

1.4 Concurrency Utilities for Java EE Expert Group

This specification is the result of the collaborative work of the members of the Concurrency Utilities for Java EE Expert Group. These include the following present and former expert group members; IBM Corporation: Chris D. Johnson, Billy Newport; BEA Systems, Inc: Revanuru Naresh, Stephan Zachwiega; Doug Lea; Redwood Software: Andrew Evers; Tangosol: Cameron Purdy and Gene Gleyzer; Cyril Bouteille; Pierre Vignéras

1.5 Document Conventions

The regular Times font is used for information that is prescriptive to this specification.

The italic Times font is used for paragraphs that contain descriptive information, such as notes describing typical use, or notes clarifying the text with prescriptive specification.

```
The Courier font is used for code examples.
```

2 Overview

The focus of this specification is on providing asynchronous capabilities to Java EE application components. This is largely achieved through extending the Concurrency Utilities API developed under JSR-166 and found in Java 2 Platform, Standard Edition 5 (J2SE™ 5) in the `java.util.concurrency` package.

The J2SE concurrency utilities provide an API that can be extended to support the majority of the goals defined in section 1.2. Application developers familiar with this API in the Java SE platform can leverage existing code libraries and usage patterns with little modification.

This specification has several aspects:

- Definition and usage of centralized, manageable `java.util.concurrent.ExecutorService` objects in a Java EE application server.
- Usage of J2SE Concurrency Utilities in a Java EE application.
- Propagation of the Java EE container's runtime contextual information to other threads.
- Managing and monitoring the lifecycle of asynchronous operations in a Java EE Application Component.
- Preserving application integrity.

2.1 Container-Managed vs. Unmanaged Threads

Java EE application servers require resource management in order to centralize administration and protect application components from consuming unneeded resources. This can be achieved through the pooling of resources and managing a resource's lifecycle. Using Java SE concurrency utilities such as the `java.util.concurrency` API, `java.lang.Thread` and `java.util.Timer` in a server application component such as a servlet or EJB are problematic since the container and server have no knowledge of these resources.

By extending the `java.util.concurrent` API, application servers and Java EE containers can become aware of the resources that are used and provide the proper execution context for the asynchronous operations to run with.

This is largely achieved by providing managed versions of the predominant `java.util.concurrent.ExecutorService` interfaces.

2.2 Application Integrity

Managed environments allow applications to coexist without causing harm to the overall system and isolate application components from one another. Administrators can adjust deployment and runtime settings to provide different qualities of service, provisioning of resources, scheduling of tasks, etc. Java EE containers also provide runtime context services to the application component. When using concurrency utilities such as those in `java.util.concurrent`, these context services need to be available.

2.3 Container Thread Context

Java EE depends on various context information to be available on the thread when interacting with other Java EE services such as JDBC data sources, JMS providers and EJBs. When using Java EE services from a non-container thread, the following behaviors are required:

- Saving the application component thread's container context.
- Identifying which container contexts to save and propagate.
- Applying a container context to the current thread.
- Restoring original thread's context.

The contexts to be propagated are governed by the container but typically include naming context, class loader and security information.

The relationships between the various Java EE architectural elements, containers and concurrency constructs are shown in Figure 1. This diagram is an extension of the Figure J2EE.2-1 in the J2EE 1.4 specification.

Containers (represented here in a single rectangle) provide environments for application components to safely interact with Java EE Standard Services (represented in the rectangles directly below the EJB/Web Container rectangle). Three new concurrency services (represented by three dark-gray rectangles) allow application components to run asynchronous tasks and Java EE Standard Services to call application components without violating container contracts.

The arrows in the diagram illustrate various flows from one part of the Java EE platform to another.

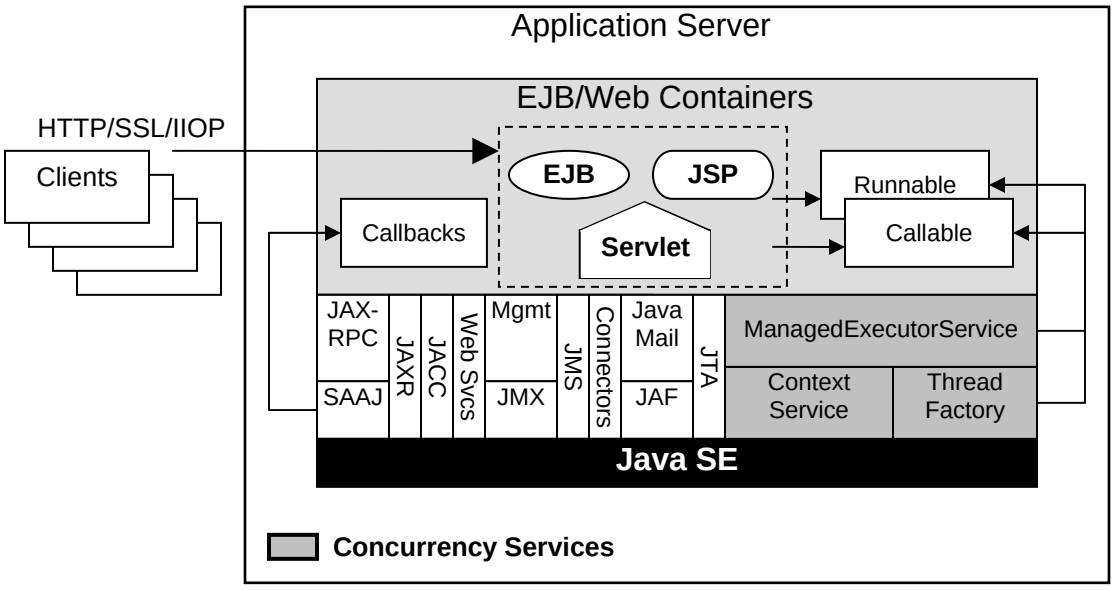

Figure 1 - Concurrency Utilities for Java EE Architecture Diagram

2.3.1 Contextual Invocation Points

Container context and management constructs are propagated to component business logic at runtime using various invocation points on well known interfaces. These invocation points or callback methods, here-by known as "tasks" will be referred to throughout the specification:

- `java.util.concurrent.Callable`
  - `call()`
- `java.lang.Runnable`
  - `run()`

The following callback methods are also contextual invocation points:

- `javax.util.concurrent.ManagedTaskListener`
  - `taskAborted()`
  - `taskSubmitted()`
  - `taskStarting()`
- `javax.util.concurrent.Trigger`
  - `getNextRunTime()`
  - `skipRun()`

Other callback methods run with an unspecified context, but can be made contextual through the `ContextService` (see following sections), which can make any Java object contextual.

2.3.2 Contextual Objects and Tasks

Tasks are concrete implementations of the Java SE `java.util.concurrent.Callable` and `java.lang.Runnable` interfaces (see the Javadoc for `java.util.concurrent.ExecutorService`). Tasks are units of work that represent a computation or some business logic.

A contextual object is any Java object instance that has a particular application component's thread context associated with it (for example; user identity).

When a task instance is submitted to a managed instance of an `ExecutorService`, the task becomes a contextual task. When the contextual task runs, the task behaves as if it were still running in the container it was submitted with.

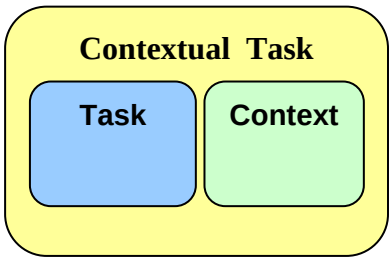

Figure 2 - Contextual Task

2.4 Usage with J2EE Connector Architecture

J2EE Connector Architecture (JCA) 1.5 allows creating resource adapters which can plug into any compatible Java EE application server. The JCA specification provides a `WorkManager` interface that allows asynchronous processing for the resource adapter. It does not provide a mechanism for Java EE applications to interact with an adapter's `WorkManager`.

This specification addresses the need for Java EE applications to run application business logic asynchronously using a `javax.util.concurrent.ManagedExecutorService` or `java.util.concurrent.ExecutorService` with a `javax.util.concurrent.ManagedThreadFactory`. It is the intent that JCA 1.5 `WorkManager` implementations may choose to utilize or wrap the `java.util.concurrent.ExecutorService` or other functionalities within this specification when appropriate.

Resource Adapters can access each of the Administered Objects described in the following sections by looking them up in the JNDI global namespace.

2.5 Security

This specification largely defers most security decisions to the container and Java EE Product Provider as defined in the Java 2 Platform Enterprise Edition Specification.

If the container supports a security context, the Java EE Product Provider must propagate that security context to the thread of execution.

Application Component Providers should use the interfaces provided in this specification when interacting with threads. If the Java EE Product Provider has implemented a security manager, some operations may not be allowed.

3 Administered Objects

This section introduces four programming interfaces for Java EE Product Providers to implement (see J2EE.2.10 for a detailed definition of each of the roles described here). Instances of these interfaces must be made available to application components through containers as administered objects:

- Section 3.1, "ManagedExecutorService" –The interface for submitting asynchronous tasks from a container.
- Section 3.2, "ManagedScheduledExecutorService" – The interface for scheduling tasks to run after a given delay or execute periodically.
- Section 3.3, "ContextService" – The interface for creating contextual objects.
- Section 3.4, "ManagedThreadFactory" – The interface for creating managed threads.

3.1 ManagedExecutorService

The `javax.util.concurrent.ManagedExecutorService` is an interface that extends the `java.util.concurrent.ExecutorService` interface. Java EE Product Providers provide implementations of this interface to allow application components to run tasks asynchronously.

3.1.1 Application Component Provider's Responsibilities

Application Component Providers (application developers) (J2EE2.10.2) use a `ManagedExecutorService` instance and associated interfaces to develop application components that utilize the concurrency functions that these interfaces provide. Instances for these objects are retrieved using the Java Naming and Directory Interface (JNDI) Naming Context (J2EE.5).

The Application Component Provider may use resource environment references to obtain references to a `ManagedExecutorService` instance as follows:

- Assign an entry in the application component's environment to the reference using the reference type of: `javax.util.concurrent.ManagedExecutorService`. (See J2EE.5.5.1.2 for information on how resource environment references are declared in the deployment descriptor.)
- This specification recommends, but does not require, that all resource environment references be organized in the appropriate subcontext of the component's environment for the resource type. For Example, all `ManagedExecutorService` references should be declared in the `java:comp/env/concurrent` subcontext.
- Look up the administered object in the application component's environment using JNDI (J2EE.5).

Note *– Future versions of Java EE may have alternatives to JNDI such as annotations and resource injection. These alternatives may be added once this specification is incorporated into an umbrella JSR.*

- Components create task classes by implementing the `java.lang.Runnable` or `java.util.concurrent.Callable` interfaces. These task classes are typically stored with the Java EE application component.
- Task instances are submitted to a `ManagedExecutorService` instance using any of the defined `submit()` or `execute()` methods. Task instances will run as an extension of the Java EE container instance that submitted the task as and may interact with Java EE resources as defined in other sections of this specification.

Note *– The use of resource environment references is best practice. Using the JNDI name of the `ManagedExecutorService` directly is allowed.*

It is important for Application Component Providers to identify and document the required behaviors and service-level agreements for each required `ManagedExecutorService`. See section 3.1.3.1 for a listing of required configuration attributes. The following example illustrates how the component can describe and utilize multiple executors.

3.1.1.1 Usage Example

In this example, an application component is performing two asynchronous operations from a servlet. One operation (reporter) is starting a task to generate a long running report. The other operations are short-running tasks that parallelize access to different back-end databases (builders).

Since each type of task has a completely different run profile, it makes sense to use two different `ManagedExecutorService` resource environment references. The attributes of each reference are documented using the `<description>` tag within the deployment descriptor of the application component and later mapped by the Deployer.

3.1.1.1.1 Reporter Task

The Reporter Task is a long-running task that communicates with a database to generate a report. The task is run asynchronously using a `ManagedExecutorService`. The client can then poll the server for the results.

3.1.1.1.1.1 Resource Environment Reference

The following resource environment reference is added to the web.xml file for the web component. The description reflects the desired configuration attributes (see 3.1.3.1):

Note *– Using the description for documenting the configuration attributes of the administered object is optional. The format used here is only an example. Future revisions of Java EE specifications may formalize usages such as this.*

```
<resource-env-ref>
   <description>
      This executor is used for the application's reporter task.
      This executor has the following requirements:
         Run Location:    NA
```

```
          Comp/Server:     Server-Administered
          Context Info:    Local Namespace
   </description>
   <resource-env-ref-name>
      concurrent/BatchExecutor
   </resource-env-ref-name>
   <resource-env-ref-type>
      javax.util.concurrent.ManagedExecutorService
   </resource-env-ref-type>
</resource-env-ref>
```

3.1.1.1.1.2 Task Definition

The task itself simply uses a resource-reference to a JDBC data source, and uses a connect/use/close pattern when invoking the Datasource.

```
public class ReporterTask implements Runnable {
   String reportName;

   public ReporterTask(String reportName) {
      this.reportName = reportName;
   }

   public void run() {
      // Run the named report
      if("TransactionReport".equals(reportName)) {
         runTransactionReport();
      else if("SummaryReport".equals(reportName)) {
         runSummaryReport();
      }
   }
}

void runTransactionReport() {

      // Lookup the DataSource (the local naming
      // environment is inherited from the parent
      // component (servlet).
      InitialContext ctx = new InitialContext();
      DataSource ds = (DataSource)
         ctx.lookup("java:comp/env/jdbc/Reporter");

      Connection con = ds.getConnection();

      // Read/Write the data using our connection.
      ...

      // Commit and close.
      con.commit();
      con.close();
   }
}
```

3.1.1.1.1.3 Task Submission

The task is started by a HTTP client connecting to a servlet. The client specifies the report name and other parameters to run. The handle to the task (the `Future`) is cached so that the client can query the results of the report. The `Future` will contain the results once the task has completed.

```
public class AppServlet extends HTTPServlet implements Servlet {
   // Cache our executor instance
```

```
   ManagedExecutorService mes = null;


   public void init(){
      // Lookup the executor and bean homes
      InitialContext ctx = new InitialContext();
      mes = (ManagedExecutorService)
         ctx.lookup("java:comp/env/concurrent/ReporterExecutor");

   }

   protected void doPost(HttpServletRequest req, HttpServletResponse
      resp) throws ServletException, IOException {
      // Get the name of the report to run from the input params…

      // Assemble the header for the response.

      // Create a task instance
      ReporterTask reporterTask = new ReporterTask(reportName);

      // Submit the task to the ManagedExecutorService
      Future reportFuture = mes.submit(reporterTask);

      // Cache the future somewhere (like the client's session)
      // The client can then poll the servlet to determine
      // the status of the report.
      ...

      // Tell the user that the report has been submitted.
      ...
   }
}
```

3.1.1.1.2 Builder Tasks

This servlet accesses two different data sources and aggregates the results before returning the page contents to the user. Instead of accessing the data synchronously, it is instead done in parallel using two different tasks. Each task accesses a local entity EJB. Since performance is most critical, the ManagedExecutorService is component-managed (see section 3.1.6.2).

3.1.1.1.2.1 Resource Environment Reference

The following resource environment reference is added to the web.xml file for the web component. The description reflects the desired configuration attributes (see 3.1.3.1):

> ***Note*** *– Using the description for documenting the configuration attributes of the administered object is optional. The format used here is only an example. Future revisions of Java EE specifications may formalize usages such as this.*

```
<resource-env-ref>
   <description>
      This executor is used for the application’s builder tasks.
      This executor has the following requirements:
         Run Location:    Local
         Comp/Server: Component-Administered
         Context Info:    Local Namespace, Security
   </description>
   <resource-env-ref-name>
      concurrent/BuilderExecutor
   </resource-env-ref-name>
```

```
   <resource-env-ref-type>
      javax.util.concurrent.ManagedExecutorService
   </resource-env-ref-type>
</resource-env-ref>
```

3.1.1.1.2.2 Task Definition

The task itself simply uses a local EJB reference to retrieve the data from the persistent store. The task implements the `javax.util.concurrent.Identifiable` interface to allow system administrators to diagnose problems.

```
public class AccountTask implements Callable<AccountInfo>, Identifiable {
   // The ID of the request to report on demand.
   String reqID;
   String accountID;
   AccountInfoHome acctInfoHome;

   public AccountTask(String reqID, String accountID, AccountInfoHome
      acctInfoHome) {
      this.reqID=reqID;
      this.accountID=accountID;
      this.acctInfoHome = acctInfoHome;
   }

   public AccountInfo call() {
      // Find the entity bean and return it to the client.
      return acctInfoHome.find(accountID);
   }

   public String getIdentityName() {
      return "AccountTask:  ReqID=" + reqID + ", Acct=" +
         accountID;
   }

   public String getIdentityDescription(Locale locale) {
      // Use a resource bundle…
      return "AccountTask asynchronous EJB invoker";
   }
}

public class InsuranceTask implements Callable<Long>, Identifiable {
   // The ID of the request to report on demand.
   String reqID;
   String accountID;
   InsuranceInfoHome insInfoHome;

   public InsuranceTask (String reqID, String accountID,
      InsuranceInfoHome insInfoHome) {

      this.reqID=reqID;
      this.accountID=accountID;
      this.insInfoHome = insInfoHome;
   }

   public InsuranceInfo call() {
      // Retreive the insurance info for the account.
      return insInfoHome.find(accountID);
   }

   public String getIdentityName() {
      return "InsuranceTask:  ReqID=" + reqID + ", Acct=" +
         accountID;
   }
```

```
   public String getIdentityDescription(Locale locale) {
      // Use a resource bundle…
      return " InsuranceTask asynchronous EJB invoker";
   }
}
```

3.1.1.1.2.3 Task Invocation

The `ManagedExecutorService` is component-managed and is therefore managed by the servlet instance lifecycle. Tasks are created on demand by a request to the servlet from an HTTP client.

```
public class AppServlet extends HTTPServlet implements Servlet {
   // Cache our executor instance and bean homes.
   ManagedExecutorService mes = null;
   AccountInfoHome acctInfoHome = null;
   InsuranceInfoHome insInfoHome = null;

   public void init(){
      // Lookup the executor and bean homes
      InitialContext ctx = new InitialContext();
      mes = (ManagedExecutorService)
         ctx.lookup("java:comp/env/concurrent/BuilderExecutor");
      acctInfoHome = (AccountInfoHome)
         ctx.lookup("java:comp/env/ejb/AccountInfoLocalHome");
      insInfoHome = (InsuranceInfoHome)
         ctx.lookup("java:comp/env/ejb/InsuranceInfoLocalHome");

   }

   public void destroy() {
      if(mes!=null) {
         // Stop the executor (this is a component-managed executor.)
         // Note:  This would throw an IllegalStateException if
         // server-managed instead of component-managed.
         mes.shutdownNow();
      }
   }

   protected void doPost(HttpServletRequest req, HttpServletResponse
      resp) throws ServletException, IOException {
      // Get our arguments from the request (accountNumber and
      // requestID, in this case.

      // Assemble the header for the response.

      // Create the task instances
      ArrayList<Callable> builderTasks = new ArrayList<Callable>();
      builderTasks.add(new AccountTask(reqID, accountID, acctInfoHome));
      builderTasks.add(new InsuranceTask(reqID, accountID,insInfoHome));

      // Submit the tasks and wait.
      List<Future<Object>> results = mes.invokeAll(builderTasks);
      AccountInfo accountInfo = (AccountInfo) results.get(0).get();
      InsuranceInfo insInfo = (InsuranceInfo) results.get(1).get();

      // Process the results
   }
}
```

3.1.2 Application Assembler's Responsibilities

The Application Assembler (J2EE.2.10.3) is responsible for assembling the application components into an Enterprise Archive (.ear) and providing assembly instructions that describe the dependencies to the administered objects.

3.1.3 Deployer's Responsibilities

The Deployer (J2EE.2.10.4) is responsible for deploying the application components into a specific operational environment. In the terms of this specification, the Deployer installs the application components and maps the dependencies defined by the Application Component Provider and Application Assembler to administered objects with the properly defined attributes. See J2EE.5.5.2 for details.

3.1.3.1 ManagedExecutorService Configuration Attributes

Each `ManagedExecutorService` may support one or more runtime behaviors as specified by the attributes in this section. `ManagedExecutorService` configuration attributes that are a requirement of this specification are as indicated. If a configuration attribute is not required, it is considered optional. Implementations that declare support for of the optional behaviors must pass the Technology Compatibility Kit (TCK) for that configuration attribute.

3.1.3.1.1 Run Location

Identifies where the tasks will run.

Local	The task is run in the same process that submitted the task. (Required)
Distributable	The task may be run in any process including the process that submitted the task. See section 3.5.
Distributable with Affinity	The task may be run in any one process including the process that submitted the task. All tasks will run on the same process. See section 3.5.

3.1.3.1.2 Other Container Contexts

Java EE Product Providers may include other contexts that may be propagated to a task or `javax.util.concurrent.ManagedTaskListener` thread (e.g. Locale). `ManagedExecutorService` implementations may add any additional contexts and provide the means for configuration those contexts in any way so long as these contexts do not violate the required aspects of this specification.

3.1.4 Java EE Product Provider's Responsibilities

The Java EE Product Provider's responsibilities are as defined in J2EE.5.5.3 and must provide implementation that provide the behaviors defined in section 3.1.3.1.

The following section illustrates some possible configuration options that a Java EE Product Provider may want to provide.

3.1.4.1 Configuration Examples

This section and subsections illustrate some examples how a Java EE Product Provider could configure a `ManagedExecutorService` and the possible options that such service could provide.

Providers may choose a more simplistic approach, or may choose to add more functionality, such as a higher, quality-of-service, persistence, task partitioning or shared thread pools.

Each of the examples has the following attributes:

- **Name**: An arbitrary name of the service for the deployer to use as a reference.
- **JNDI name**: The arbitrary, but required, name to identify the service instance. The deployer uses this value to map the service to the component's resource environment reference.
- **Context**: A reference to a `ContextService` instance (see section 3.3). The context service can be used to define the context to propagate to the threads when running tasks. Having more than one `ContextService`, each with a different policy may be desirable for some implementations.
- **ThreadFactory**: A reference to a Managed ThreadFactory instance (see section 3.4). The managed ThreadFactory instance can create threads with different attributes (such as priority).
- **Management**: Section 3.1.6 describes server-managed and component-managed executors.
- **Hung Task Threshold**: The amount of time in milliseconds that a task can execute before it is considered hung.
- **Pool Info**: If the executor is a thread pool, then the various thread pool attributes can be defined (this is based on the attributes for the Java `java.util.concurrent.ThreadPoolExecutor` class):
  - **Core Size**: The number of threads to keep in the pool, even if they are idle.
  - **Maximum Size**: The maximum number of threads to allow in the pool (could be unbounded).
  - **Keep Alive**: The time to allow threads to remain idle when the number of threads is greater than the core size.
  - **Work Queue Capacity**: The number of tasks that can be stored in the input bounded buffer (could be unbounded).
- **Reject Policy**: The policy to use when a task is to be rejected by the executor. In this example, two policies are available:
  - **Abort**: Throw an exception when rejected.
  - **Retry and Abort**: Automatically resubmit to another instance and abort if it fails.

- **Run Location**: Identifies this executor as a distributable or local type (See section 3.1.3.1.1).

3.1.4.1.1 Typical Thread Pool

The Typical Thread Pool illustrates a common configuration for an application server with few applications. Each application expects to run a small number short-duration tasks in the local process.

ManagedExecutorService	
Name:	Typical Thread Pool
JNDI Name:	concurrent/execsvc/Shared
Context:	concurrent/ctx/AllContexts
Thread Factory:	concurrent/tf/normal
Management:	(•) Server-Managed ( ) Component-Managed
Hung Task Threshold:	60000 ms
Pool Info:	Core Size: 5 Max Size 25 Keep Alive: 5000 ms Work Queue Capacity: 15
Reject Policy:	(•) Abort ( ) Retry and Abort
Run Location:	(•) Local ( ) Distributable ( ) Distrib. with Affinity

Table 1 - Typical Thread Pool Configuration Example

3.1.4.1.2 Batch Executor

The Batch Executor describes a configuration in which the executor is used to run a few long-running tasks in the local process, such as batch jobs. In this example the task can run up to 24 hours before it is considered hung.

ManagedExecutorService	
Name:	Batch Executor
JNDI Name:	concurrent/execsvc/Batch
Context:	concurrent/ctx/AllContexts
Thread Factory:	concurrent/tf/batch
Management:	(•) Server-Managed ( ) Component-Managed
Hung Task Threshold:	24 hours

Pool Info:	Core Size: 0 Max Size 5 Keep Alive: 1000 Work Queue Capacity: 5
Reject Policy:	◉ Abort ○ Retry and Abort
Run Location:	◉ Local ○ Distributable ○ Distrib. with Affinity

Table 2 - Batch Executor Configuration Example

3.1.4.1.3 OLTP Thread Pool

The OLTP (On-Line Transaction Processing)Thread Pool executor uses a thread pool with many more threads and a low hung-task threshold. It also uses a thread factory that creates threads with a slightly higher priority and a `ContextService` with a limited amount of context information.

The executor is component-scoped. This means that the application component has its own instance, which decreases the amount of context propagation that needs to occur.

ManagedExecutorService	
Name:	Shared OLTP Thread Pool
JNDI Name:	concurrent/execsvc/OLTPShared
Context:	concurrent/ctx/OLTPContexts
Thread Factory:	concurrent/tf/oltp
Management:	○ Server-Managed ◉ Component-Managed
Hung Task Threshold:	20000 ms
Pool Info:	Core Size: 100 Max Size 250 Keep Alive: 10000 ms Work Queue Capacity: 100
Reject Policy:	◉ Abort ○ Retry and Abort
Run Location:	◉ Local ○ Distributable ○ Distrib. with Affinity

Table 3 - OLTP Thread Pool Configuration Example

3.1.5 System Administrator's Responsibilities

The System Administrator (J2EE.2.10.5) is responsible for monitoring and overseeing the runtime environment. In the scope of this specification, these duties may include:

- Monitoring for hung tasks.
- Monitoring resource usage (for example, threads and memory).

See section 4, "Managed Objects" for details on how to monitor thread usage.

3.1.6 Server and Component-Managed ManagedExecutorServices

Each `ManagedExecutorService` is either server-managed or component-managed. Server-managed `ManagedExecutorServices` indicate that the instance and lifecycle are centrally managed by the application server. Therefore, the lifecycle server-managed `ManagedExecutorServices` cannot be changed by an application.

Component-managed `ManagedExecutorServices` are local to the component that looks it up. Each look-up of a component-managed `ManagedExecutorService` will result in a new instance. The lifecycle of the `ManagedExecutorService` can be manually controlled by the component and its lifecycle is bound to the component's lifecycle and behaves more like a Java SE `ExecutorService`.

3.1.6.1 Server-Managed ManagedExecutorService

A server-managed `ManagedExecutorService` is intended to be used by multiple components and applications. When the executor runs a task, the context of the thread is changed to match the component instance that submitted the task. The context is then restored when the task is complete.

In Figure 3, a single server-managed `ManagedExecutorService` is used to run tasks (in blue) from multiple application components (each denoted in a different color). Each task, when submitted to the `ManagedExecutorService` automatically retains the context of the submitting component and it becomes a Contextual Task. When the `ManagedExecutorService` runs the task, the thread applies the context to the task such that it behaves as if it were part of the container that submitted the task.

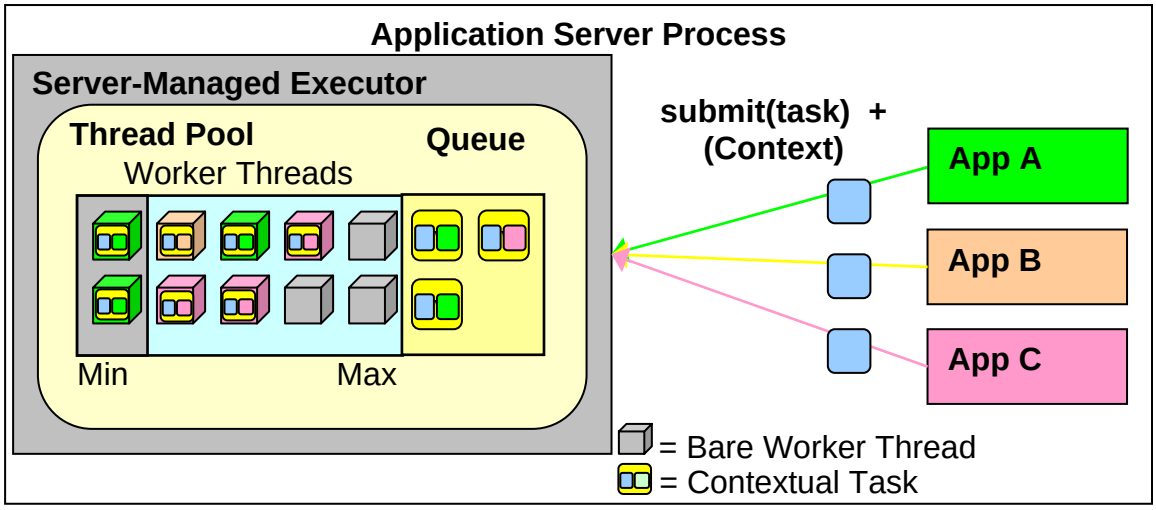

Figure 3 - Server-Managed Thread Pool Executor Component Relationship

3.1.6.1.1 Java EE Product Provider Requirements

This subsection describes additional requirements for server-managed `ManagedExecutorService` providers.

1. All tasks, when executed from the `ManagedExecutorService`, will run with the Java EE component identity of the component that submitted the task.

2. The lifecycle of a `ManagedExecutorService` is managed by an application server. All lifecycle operations on the `ManagedExecutorService` interface will throw a `java.lang.IllegalStateException` exception (see 3.1.6.2.2).
3. All tasks submitted to an executor should not run if task's component is not started.

3.1.6.2 Component-Managed Executor

A component-managed `ManagedExecutorService` is intended to be used by a single component instance or session. All threads that run under this executor will run with the context of the executor's creator. If the threads are pooled, the overhead of reusing a thread disappears since the context does not have to change the thread's context.

In Figure 4, an application (App A) has two different components that are interacting with a component-managed `ManagedExecutorService`. In this scenario, an EJB method (EJB 1) looks-up the component-managed executor and made it publicly available to a servlet (Servlet 1) to use (e.g. using a static field of a shared class). All threads in the executor's thread pool are primed with the context of the EJB method. When both the EJB and Servlet submit tasks to the executor both will run in the EJB method's context. In order for this scenario to work properly, the servlet must be able to tolerate running with the EJB's context.

Note *– This scenario is not typical and is described for illustration purposes. A component-managed executor is to be used from a single application component and not shared with other components.*

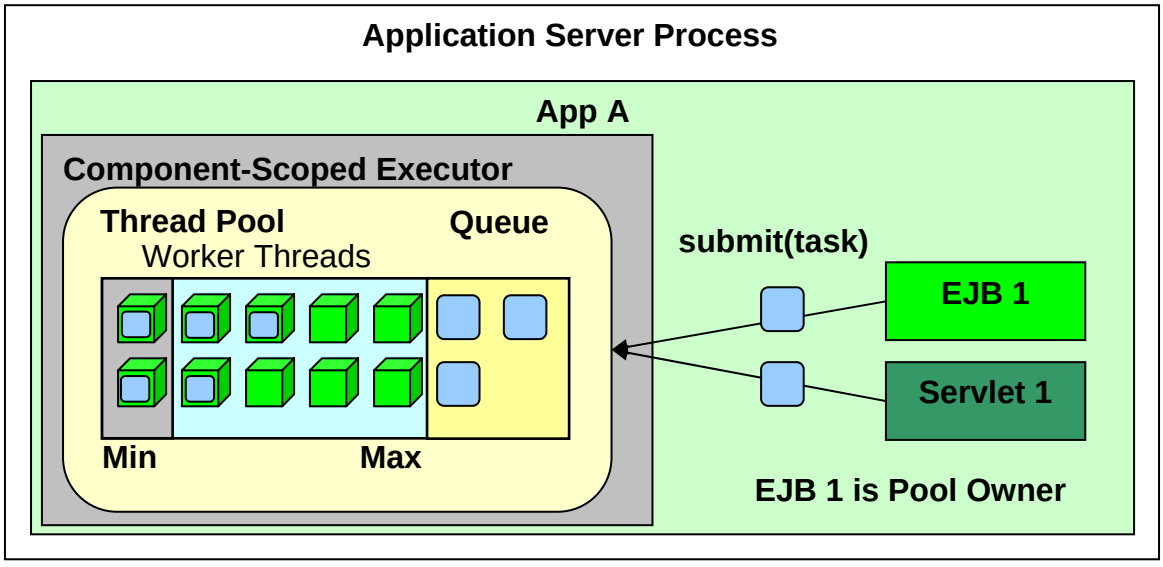

Figure 4 - Component-Managed Thread Pool Executor Component Relationship

3.1.6.2.1 Java EE Product Provider Requirements

This subsection describes additional requirements for component-managed `ManagedExecutorService` providers.

1. Each look-up will result in a unique `ManagedExecutorService` instance.

2. All tasks, when executed by the `ManagedExecutorService` will run with the context of the application component that looked-up the `ManagedExecutorService`.
3. The lifecycle of a `ManagedExecutorService` is managed by the component that created it. All lifecycle operations are permitted if the calling thread has the appropriate security manager permissions as described in the `java.util.concurrent.ExecutorService` Javadoc.
4. All tasks submitted to an executor should not run if the task's component is not started.
5. If a component is stopped, the `ManagedExecutorService` should terminate.

3.1.6.2.2 Component-Managed Executor Lifecycle

Component-managed `ManagedExecutorService` instances are owned and managed by the component that looked-up the service. The component can shutdown the `ManagedExecutorService` using the `shutdown()` and `shutdownNow()` methods and can monitor its lifecycle using the `awaitTermination()`, `isShutdown()` and `isTerminated()` methods. When a component-managed `ManagedExecutorService` is shutdown by the application component, the executor behaves as described in sections 3.1.6.2.2.1 and 3.1.6.2.2.2.

Component-managed `ManagedExecutorService` instances may also be terminated or suspended by the application server when applications or components are stopped or the application server itself is shutting down.

3.1.6.2.2.1 Component-managed executor shutdown requirements

All `ManagedExecutorService` instances have the following requirements when shutdown using the `shutdown` API method:

1. All attempts to submit new tasks are rejected.
2. Previously submitted tasks may run.
3. Running task threads may be interrupted.
4. Registered `ManagedTaskListeners` may be invoked..

3.1.6.2.2.2 Component-managed executor shutdownNow requirements

All `ManagedExecutorService` instances have the following requirements when shutdown using the `shutdownNow` API method:

1. All attempts to submit new tasks are rejected.
2. All submitted tasks are cancelled if not running.
3. All running task threads are interrupted.
4. All registered `ManagedTaskListeners` are invoked.

3.1.7 Quality of Service

`ManagedExecutorService` implementations must support the at-most-once quality of service. The at-most-once quality of service guarantees that a task will run at most, one

time. This quality of service is the most efficient method to run tasks. Tasks submitted to an executor with this quality of service are transient in nature, are not persisted and do not survive process restarts.

Other qualities of service are allowed, but are not addressed in this specification.

3.1.8 Transaction Management

`ManagedExecutorService` implementations must support user-managed global transactions with similar semantics to EJB bean-managed transaction demarcation (see the Enterprise JavaBeans specification). User-managed transactions allow components to manually control global transaction demarcation boundaries. Task implementations may optionally begin, commit and roll-back a transaction. See J2EE.4 for details on transaction management in Java EE.

3.1.8.1 Java EE Product Provider Requirements

This subsection describes the transaction management requirements of a `ManagedExecutorService` implementation.

1. The `javax.transaction.UserTransaction` interface must be made available in the local JNDI namespace as environment entry: `java:comp/UserTransaction` (J2EE.5.7 and J2EE.4.2.1.1)
2. All resource managers must enlist with a `UserTransaction` instance when a transaction is active using the `begin()` method.
3. The executor is responsible for coordinating commits and rollbacks when the transaction ends using `commit()` and `rollback()` methods.

3.1.8.2 Application Component Provider's Requirements

This subsection describes the transaction management requirements of each task provider's implementation.

1. A task instance that starts a transaction must complete the transaction before starting a new transaction.
2. The task provider uses the `javax.transaction.UserTransaction` interface to demarcate transactions.
3. Transactions are demarcated using the `begin()`, `commit()` and `rollback()` methods of the `UserTransaction` interface.
4. While an instance is in an active transaction, resource-specific transaction demarcation APIs must not be used (e.g. if `a javax.sql.Connection` is enlisted in the transaction instance, the `Connection.commit()` and `Connection.rollback()` methods must not be used).
5. The task instance must complete the transaction before the task method ends.

3.1.8.2.1 UserTransaction Usage Example

The following example illustrates how a task can interact with two XA-capable resources in a single transaction:

```
public class TranTask implements Runnable {
```

```
   public void run() {
      InitialContext ctx = new InitialContext();
      UserTransaction ut = (UserTransaction)
         ctx.lookup("java:comp/UserTransaction");

      // Start a transaction
      ut.begin();

      // Invoke an EJB
      ...
      // Update a database using an XA capable JDBC DataSource
      ...

      // Commit the transaction
      ut.commit();
   }
}
```

3.2 ManagedScheduledExecutorService

The `javax.util.concurrent.ManagedScheduledExecutorService` is an interface that extends the `java.util.concurrent.ScheduledExecutorService` and `javax.util.concurrent.ManagedExecutor` interfaces. Java EE Product Providers provide implementations of this interface to allow applications to run tasks at specified and periodic times.

The `ManagedScheduledExecutorService` offers the same managed semantics as the `ManagedExecutorService` and includes the delay and periodic task running capabilities that the `ScheduledExecutorService` interface provides with the addition of `Triggers`.

3.2.1 Application Component Provider's Responsibilities

Application Component Providers (application developers) (J2EE2.10.2) use a `ManagedScheduledExecutorService` instance and associated interfaces to develop application components that utilize the concurrency functions that these interfaces provide. Instances for these objects are retrieved using the Java Naming and Directory Interface (JNDI) Naming Context (J2EE.5).

The Application Component Provider may use resource environment references to obtain references to a `ManagedScheduledExecutorService` instance as follows:

- Assign an entry in the application component's environment to the reference using the reference type of: `javax.util.concurrent.ManagedScheduledExecutorService`. (See J2EE.5.5.1.2 for information on how resource environment references are declared in the deployment descriptor.)
- This specification recommends, but does not require, that all resource environment references be organized in the appropriate subcontext of the component's environment for the resource type. For Example, all `ManagedScheduledExecutorService` references should be declared in the `java:comp/env/concurrent` subcontext.
- Look up the administered object in the application component's environment using JNDI (J2EE.5).

Note *– Future versions of Java EE may have alternatives to JNDI such as annotations and resource injection. These alternatives may be added once this specification is incorporated into an umbrella JSR.*

- Components create task classes by implementing the `java.lang.Runnable` or `java.util.concurrent.Callable` interfaces. These task classes are typically stored with the Java EE application component.
- Task instances are submitted to a `ManagedScheduledExecutorService` instance using any of the defined `submit()`, `execute()`, `scheduleAtFixedRate()` or `scheduleWithFixedDelay()` methods. Task instances will run as an extension of the Java EE container instance that submitted the task as and may interact with Java EE resources as defined in other sections of this specification.

Note *– The use of resource environment references is best practice. Using the JNDI name of the `ManagedExecutorService` directly is allowed.*

It is important for Application Component Providers to identify and document the required behaviors and service-level agreements for each required `ManagedScheduledExecutorService`. See section 3.2.3.1 for a listing of required configuration attributes. The following example illustrates how the component can describe and utilize a `ManagedScheduledExecutorService`.

3.2.1.1 Usage Example

In this example, an application component wants to use a timer to periodically write in-memory events to a database log.

The attributes of the `ManagedScheduledExecutorService` reference is documented using the `<description>` tag within the deployment descriptor of the application component and later mapped by the Deployer.

3.2.1.1.1 Logger Timer Task

The Logger Timer Task is a short-running, periodic task that has the same life-cycle as the servlet. It periodically wakes up and dumps a queue's contents to a database log. Its lifecycle is controlled using a `javax.servlet.ServletContextListener`.

3.2.1.1.1.1 Resource Environment Reference

The following resource environment reference is added to the web.xml file for the web component. The description reflects the desired configuration attributes (see 3.1.3.1):

Note *– Using the description for documenting the configuration attributes of the administered object is optional. The format used here is only an example. Future revisions of Java EE specifications may formalize usages such as this.*

```
<resource-env-ref>
   <description>
      This executor is used for the application’s logger task.
      This executor has the following requirements:
         Run Location:    Local
         Comp/Server:     Server-Administered
         Context Info:    Local Namespace
   </description>
```

```
   <resource-env-ref-name>
      concurrent/ScheduledLoggerExecutor
   </resource-env-ref-name>
   <resource-env-ref-type>
      javax.util.concurrent.ManagedScheduledExecutorService
   </resource-env-ref-type>
</resource-env-ref>
```

3.2.1.1.1.2 Task Definition

The task itself simply uses a resource-reference to a JDBC data source, and uses a connect/use/close pattern when invoking the Datasource.

```
public class LoggerTimer implements Runnable {
   DataSource ds;

   public LoggerTimer() {
      // Lookup the DataSource
      InitialContext ctx = new InitialContext();
      ds = (DataSource) ctx.lookup("java:comp/env/jdbc/Logger");
   }

   public void run() {
      logEvents(getData(), ds);
   }
}

void logEvents(Collection data, DataSource ds) {

   // Iterate through the data and log each row.
   for (…) {
      Connection con = ds.getConnection();

      // Write the data using our connection.
      …

      // Commit and close.
      con.commit();
      con.close();
   }
}
```

3.2.1.1.1.3 Task Submission

The task is started and stopped by a `javax.servlet.ServletContextListener`.

```
public class CtxListener implements ServletContextListener {
   Future loggerHandle = null;
   public void contextInitialized(ServletContextEvent scEvent) {

      // Lookup the executor and submit our task.
      InitialContext ctx = new InitialContext();
      ManagedScheduledExecutorService mes =
         (ManagedScheduledExecutorService)
         ctx.lookup("java:comp/env/concurrent/ScheduledLoggerExecutor");

      LoggerTimer logger = new LoggerTimer();
      loggerHandle = mes.scheduleAtFixedRate(
         logger, 5, TimeUnit.SECONDS);

   }

   public void contextDestroyed(ServletContextEvent scEvent) {
```

```
        // Cancel and interrupt our logger task
        if(loggerHandle!=null) {
            loggerHandle.cancel(true);
        }
    }
}
```

3.2.2 Application Assembler's Responsibilities

The Application Assembler (J2EE.2.10.3) is responsible for assembling the application components into an Enterprise Archive (.ear) and providing assembly instructions that describe the dependencies to the administered objects.

3.2.3 Deployer's Responsibilities

The Deployer (J2EE.2.10.4) is responsible for deploying the application components into a specific operational environment. In the terms of this specification, the Deployer installs the application components and maps the dependencies defined by the Application Component Provider and Application Assembler to administered objects with the properly defined attributes. See J2EE.5.5.2 for details.

3.2.3.1 ManagedScheduledExecutorService Configuration Attributes

Each `ManagedScheduledExecutorService` may support one or more runtime behaviors as specified by the attributes in this section. `ManagedScheduledExecutorService` configuration attributes that are a requirement of this specification are as indicated. If a configuration attribute is not required, it is considered optional. Implementations that declare support for of the optional behaviors must pass the Technology Compatibility Kit (TCK) for that configuration attribute.

3.2.3.1.1 Run Location

Identifies where the tasks will run.

Local	The task is run in the same process that submitted the task. (Required)
Distributable	The task may be run in any process including the process that submitted the task. See section 3.5.
Distributable with Affinity	The task may be run in any one process including the process that submitted the task. All tasks will run on the same process. See section. See section 3.5.

3.2.3.1.2 Other Container Contexts

Java EE Product Providers may include other contexts that may be propagated to a task or `javax.util.concurrent.ManagedTaskListener` thread (e.g. Locale). `ManagedScheduledExecutorService` implementations may add any additional contexts and

provide the means for configuration those contexts in any way so long as these contexts do not violate the required aspects of this specification.

3.2.4 Java EE Product Provider's Responsibilities

The Java EE Product Provider's responsibilities are as defined in J2EE.5.5.3 and must provide implementation that provide the behaviors defined in section 3.2.3.1.

3.2.4.1 Configuration Examples

This section and subsections illustrate some examples how a Java EE Product Provider could configure a `ManagedScheduledExecutorService` and the possible options that such service could provide.

Providers may choose a more simplistic approach, or may choose to add more functionality, such as a higher quality-of-service or persistence.

Each of the examples has the following attributes:

- **Name**: An arbitrary name of the service for the deployer to use as a reference.
- **JNDI name**: The arbitrary, but required, name to identify the service instance. The deployer uses this value to map the service to the component's resource environment reference.
- **Context**: A reference to a `ContextService` instance (see section 3.3). The context service can be used to define the context to propagate to the threads when running tasks. Having multiple `ContextService` instances, each with a different policy may be desirable for some implementations.
- **ThreadFactory**: A reference to a Managed ThreadFactory instance (see section 3.4). The managed ThreadFactory instance can create threads with different attributes (such as priority).
- **Management**: Section 3.1.6 describes server-managed and component-managed executors.
- **Thread Use**: If the application intends to run short vs. long-running tasks they can specify to use pooled or daemon threads.
- **Hung Task Threshold**: The amount of time in milliseconds that a task can execute before it is considered hung.
- **Pool Info**: If the executor is a thread pool, then the various thread pool attributes can be defined (this is based on the attributes for the Java `java.util.concurrent.ThreadPoolExecutor` class):
    - **Core Size**: The number of threads to keep in the pool, even if they are idle.
    - **Maximum Size**: The maximum number of threads to allow in the pool (could be unbounded).
    - **Keep Alive**: The time to allow threads to remain idle when the number of threads is greater than the core size.

- **Reject Policy**: The policy to use when a task is to be rejected by the executor. In this example, two policies are available:
  - **Abort**: Throw an exception when rejected.
  - **Retry and Abort**: Automatically resubmit to another instance and abort if it fails.
- **Run Location**: Identifies this executor as a distributable or local type (See section 3.2.3.1.1).

3.2.4.1.1 Typical Timer

This example describes a typical configuration for a `ManagedScheduledExecutorService` that uses a bounded thread pool. Only 10 timers can run simultaneously and are considered hung if they have run more than 5 seconds. An executor such as this can be shared between applications and is designed to run very short-duration tasks. For example, marking a transaction to rollback after a timeout.

ManagedScheduledExecutorService	
Name:	Typical Timer
JNDI Name:	concurrent/execsvc/Timer
Context:	concurrent/cf/AllContexts
Thread Factory:	concurrent/tf/normal
Management:	(•) Server-Managed ( ) Component-Managed
Thread Use:	(•) Pooled ( ) Daemon
Hung Task Threshold:	5000 ms
Pool Info:	Core Size: 2 Max Size 10 Keep Alive: 3000 ms
Reject Policy:	(•) Abort ( ) Retry and Abort
Run Location:	(•) Local ( ) Distributable ( ) Distrib. with Affinity

Table 4 - Typical Timer Configuration Example

3.2.4.1.2 Component Timer

This example describes a high-performance component-managed timer. This type of timer could be used for timing-out sessions or invalidating cached data for each client. If it is important to have the tasks run at the designed time as accurately as possible, eliminating the overhead of creating the threads and propagating some container context can help.

ManagedScheduledExecutorService	
Name:	Component Timer
JNDI Name:	concurrent/execsvc/CompTimer

Context:	concurrent/cf/NoContext
Thread Factory:	concurrent/tf/normal
Management:	○ Server-Managed ◉ Component-Managed
Thread Use:	◉ Pooled ○ Daemon
Hung Task Threshold:	5000 ms
Pool Info:	Core Size: 20 Max Size 20 Keep Alive: n/a
Reject Policy:	◉ Abort ○ Retry and Abort
Run Location:	◉ Local ○ Distributable ○ Distrib. with Affinity

Table 5 - Component Timer Configuration Example

3.2.5 System Administrator's Responsibilities

The System Administrator (J2EE.2.10.5) is responsible for monitoring and overseeing the runtime environment. In the scope of this specification, these duties may include:

- Monitoring for hung tasks.
- Monitoring resource usage (for example, threads and memory).

See section 4, "Managed Objects" for details on how to monitor thread usage.

3.2.6 Server and Component-Managed ManagedExecutorServices

Each `ManagedScheduledExecutorService` is either server-managed or component-managed. Server-managed `ManagedScheduledExecutorService` indicate that the instance and lifecycle are centrally managed by the application server. Therefore, the lifecycle server-managed `ManagedScheduledExecutorService` cannot be changed by an application.

Component-managed `ManagedScheduledExecutorService` are local to the component that looks it up. Each look-up of a component-managed `ManagedScheduledExecutorService` will result in a new instance. The lifecycle of the `ManagedScheduledExecutorService` can be manually controlled by the component and its lifecycle is bound to the component's lifecycle and behaves much like a Java SE `ExecutorService`.

3.2.6.1 Server-Managed ManagedScheduledExecutorService

A server-managed `ManagedScheduledExecutorService` is intended to be used by multiple components and applications. When the executor runs a task, the context of the thread is changed to match the component instance that submitted the task. The context is then restored when the task is complete. See Figure 3 - Server-Managed Thread Pool Executor Component Relationship.

3.2.6.1.1 Java EE Product Provider Requirements

This subsection describes additional requirements for server-managed `ManagedScheduledExecutorService` providers.

1. All tasks, when executed from the `ManagedScheduledExecutorService`, will run with the context of the application component that submitted the task.
2. The lifecycle of a `ManagedScheduledExecutorService` is managed by an application server. All lifecycle operations on the `ManagedScheduledExecutorService` interface will throw a `java.lang.IllegalStateException` exception (see 3.2.6.2.2).
3. All tasks submitted to an executor should not run if task's component is not started.

3.2.6.2 Component-Managed Executor

A component-managed `ManagedScheduledExecutorService` is intended to be used by a single application component instance or session. All threads that run under this executor instance will run with the context of the application component that created (looked-up) the `ManagedScheduledExecutorService`. If the threads are pooled, the overhead of reusing a thread disappears since the context does not have to change the thread's context. See Figure 4 - Component-Managed Thread Pool Executor Component Relationship.

3.2.6.2.1 Java EE Product Provider Requirements

This subsection describes additional requirements for component-managed `ManagedScheduledExecutorService` providers.

1. Each look-up will result in a unique `ManagedScheduledExecutorService` instance.
2. All tasks, when executed by the `ManagedScheduledExecutorService` will run with the context of the application component that looked-up the `ManagedScheduledExecutorService`.
3. The lifecycle of a `ManagedScheduledExecutorService` is managed by the component that created it. All lifecycle operations are permitted if the calling thread has the appropriate security manager permissions as described in the `java.util.concurrent.ExecutorService` Javadoc.
4. All tasks submitted to an executor should not run if the task's component is not started.
5. If a component is stopped, the `ManagedScheduledExecutorService` should terminate.

3.2.6.2.2 Component-Managed Executor Lifecycle

Component-managed `ManagedScheduledExecutorService` instances are owned and managed by the component that looked-up the service. The component can shutdown the `ManagedScheduledExecutorService` using the `shutdown()` and `shutdownNow()` methods and can monitor its lifecycle using the `awaitTermination()`, `isShutdown()` and `isTerminated()` methods. When a component-managed

`ManagedScheduledExecutorService` is shutdown by the application component, the executor behaves as described in sections 3.2.6.2.2.1 and 3.2.6.2.2.2.

Component-managed `ManagedScheduledExecutorService` instances may also be terminated or suspended by the application server when applications or components are stopped or the application server itself is shutting down.

3.2.6.2.2.1 Component-managed executor shutdown requirements

All `ManagedScheduledExecutorService` instances have the following requirements when shutdown using the `shutdown` API method:

1. All attempts to submit new tasks are rejected.
2. Previously submitted tasks may run.
3. Running task threads may be interrupted.
4. Registered `ManagedTaskListeners` may be invoked..

3.2.6.2.2.2 Component-managed executor shutdownNow requirements

All `ManagedScheduledExecutorService` instances have the following requirements when shutdown using the `shutdownNow` API method:

1. All attempts to submit new tasks are rejected.
2. All submitted tasks are cancelled if not running.
3. All running task threads are interrupted.
4. All registered `ManagedTaskListeners` are invoked.

3.2.7 Quality of Service

`ManagedScheduledExecutorService` implementations must support the at-most-once quality of service. The at-most-once quality of service guarantees that a task will run at most, one time. This quality of service is the most efficient method to run tasks. Tasks submitted to an executor with this quality of service are transient in nature, are not persisted and do not survive process restarts.

Other qualities of service are allowed, but are not addressed in this specification.

3.2.8 Transaction Management

`ManagedScheduledExecutorService` implementations must support user-managed global transactions with similar semantics to EJB bean-managed transaction demarcation (see the Enterprise JavaBeans specification). User-managed transactions allow components to manually control global transaction demarcation boundaries. Task implementations may optionally begin, commit and roll-back a transaction. See J2EE.4 for details on transaction management in Java EE.

3.2.8.1 Java EE Product Provider Requirements

This subsection describes the transaction management requirements of a `ManagedScheduledExecutorService` implementation.

1. The `javax.transaction.UserTransaction` interface must be made available in the local JNDI namespace as environment entry: `java:comp/UserTransaction` (J2EE.5.7 and J2EE.4.2.1.1)
2. All resource managers must enlist with a `UserTransaction` instance when a transaction is active using the `begin()` method.
3. The executor is responsible for coordinating commits and rollbacks when the transaction ends using `commit()` and `rollback()` methods.

3.2.8.2 Application Component Provider's Requirements

This subsection describes the transaction management requirements of each task provider's implementation.

1. A task instance that starts a transaction must complete the transaction before starting a new transaction.
2. The task provider uses the `javax.transaction.UserTransaction` interface to demarcate transactions.
3. Transactions are demarcated using the `begin()`, `commit()` and `rollback()` methods of the `UserTransaction` interface.
4. While an instance is in an active transaction, resource-specific transaction demarcation APIs must not be used (e.g. if `a javax.sql.Connection` is enlisted in the transaction instance, the `Connection.commit()` and `Connection.rollback()` methods must not be used).
5. The task instance must complete the transaction before the task method ends.

See section 3.1.8.2.1 for an example on how to use a UserTransaction within a task.

3.3 ContextService

The `javax.util.concurrent.ContextService` allows applications to create contextual objects without using a managed executor. The `ContextService` uses the dynamic proxy capabilities found in the `java.lang.reflect` package to associate the application component container context with an object instance. The object becomes a contextual object (see section 2.3.2) and whenever a method on the contextual object is invoked, the method executes with the thread context of the associated application component instance.

Contextual objects allow application components to develop a wide variety of applications and services that are not normally possible in the Java EE platform, such as workflow systems. When used in conjunction with a `ManagedThreadFactory`, customized Java SE platform `ExecutorService` implementations can be used.

The `ContextService` also allows non-Java EE service callbacks (such as JMS `MessageListeners` and JMX™ `NotificationListeners`) to run in the context of the listener registrant instead of the implementation provider's undefined thread context. See section 4.4.3 for an example).

3.3.1 Application Component Provider's Responsibilities

Application Component Providers (application developers) (J2EE2.10.2) use a `ContextService` instance to create contextual object proxies. Instances for these objects are retrieved using the Java Naming and Directory Interface (JNDI) Naming Context (J2EE.5).

The Application Component Provider may use resource environment references to obtain references to a `ContextService` instance as follows:

- Assign an entry in the application component's environment to the reference using the reference type of: `javax.util.concurrent.ContextService`. (See J2EE.5.5.1.2 for information on how resource environment references are declared in the deployment descriptor.)
- This specification recommends, but does not require, that all resource environment references be organized in the appropriate subcontext of the component's environment for the resource type. For Example, all `ContextService` references should be declared in the `java:comp/env/concurrent` subcontext.
- Look up the administered object in the application component's environment using JNDI (J2EE.5).

  ***Note** – Future versions of Java EE may have alternatives to JNDI such as annotations and resource injection. These alternatives may be added once this specification is incorporated into an umbrella JSR.*

- Contextual object proxies instances are created with a `ContextService` instance using the `createContextObject()` method. Contextual object proxies will run as an extension of the application component instance that created the proxy as and may interact with Java EE container resources as defined in other sections of this specification.

***Note** – The use of resource environment references is best practice. Using the JNDI name of the `ContextService` directly is allowed.*

It is important for Application Component Providers to identify and document the required behaviors and service-level agreements for each required `ContextService`. See section 3.3.3.1 for a listing of required configuration attributes. The following example illustrates how the component can describe and utilize a `ContextService`.

3.3.1.1 Usage Example

This section provides an example that shows how a custom `ExecutionService` can be utilized within an application component.

3.3.1.1.1 Custom ExecutorService

This example demonstrates how a singleton Java SE ExecutorService implementation (such as the `java.util.concurrent.ThreadPoolExecutor`) can be used from an EJB . In

this example, the reference `ThreadPoolExecutor` implementation is used instead of the implementation supplied with the Java EE Product Provider.

A custom `ExecutorService` can be created like any Java object. For applications to use an object, it can be accessed using a singleton or using a JCA 1.5 resource adapter. In this example, we use a singleton.

Since the `ExecutorService` is a singleton (within the application component's ClassLoader), it can be accessed by several EJB or Servlet instances. The ExecutorService uses threads created from a managed `ThreadFactory` (see section 3.4) provided by the Java EE Product Provider. The `ContextService` is used to guarantee that the task, when it runs on one of the worker threads in the pool, will have the correct component context available to it.

3.3.1.1.1.1 ExecutorService Singleton

Create a singleton for the `ExecutorService` instance in an ExecutorAccessor class. Since there is no portable module or application lifecycle capability in Java EE, one must rely on creating singletons. The `ExecutorAccessor` should be included with the EJB module or other jar that is in the scope of the application component.

```
public class ExecutorAccessor {
    private static ExecutorService threadPoolExecutor = null;
    private static Throwable initException = null;
    static {
        try {
            InitialContext ctx = new InitialContext();
            ThreadFactory threadFactory = (ThreadFactory)
                ctx.lookup("java:comp/env/concurrent/ThreadFactory");
            threadPoolExecutor = new ThreadPoolExecutor(
                 5, 10, 5, TimeUnit.SECONDS,
                 new ArrayBlockingQueue<Runnable>(10), threadFactory);
        } catch (Throwable e) {
            initException = e;
        }
    }

    public static ExecutorService getThreadPool() {
        return threadPoolExecutor;
    }

    public static Throwable getInitException() {
        return initException;
    }
}
```

3.3.1.1.1.2 CreditReport Task

The CreditReport task retrieves a credit report from a given credit agency for a given tax identification number. Multiple tasks are invoked in parallel by an EJB business method.

3.3.1.1.1.2.1 Resource Environment References

This example refers to a ContextService and managed ThreadFactory.

Note *– Using the description for documenting the configuration attributes of the administered object is optional. The format used here is only an example. Future revisions of Java EE specifications may formalize usages such as this.*

```
<resource-env-ref>
   <description>
      This ThreadFactory is used for the singleton ThreadPoolExecutor.
      This ThreadFactory has the following requirements:
         Priority:    Normal
         Context Info:    NA
   </description>
   <resource-env-ref-name>
      concurrent/ThreadFactory
   </resource-env-ref-name>
   <resource-env-ref-type>
      javax.util.concurrent.ManagedThreadFactory
   </resource-env-ref-type>
</resource-env-ref>

<resource-env-ref>
   <description>
      This ContextService is used in conjunction with the custom
      ThreadPoolExecutor that the credit report component is using.
      This ContextService has the following requirements:
         Context Info:    Local namespace, security
   </description>
   <resource-env-ref-name>
      concurrent/AllContexts
   </resource-env-ref-name>
   <resource-env-ref-type>
      javax.util.concurrent.ContextService
   </resource-env-ref-type>
</resource-env-ref>
```

3.3.1.1.1.2.2 Task Definition

This task logs the request in a database, which requires the local namespace in order to locate the correct Datasource. It also utilizes the Java Authentication and Authorization API (JAAS) to retrieve the user's identity from the current thread in order to audit access to the credit report.

```
public class CreditScoreTask implements Callable<Long>{

   private long taxID;
   private int agency;

   public CreditScoreTask(long taxID, int agency) {
      this.taxID = taxID;
      this.agency = agency;
   }

   public Long call() {
      // Log the request in a database using the identity of the user.
      // Use the local namespace to locate the datasource
      Subject currentSubject =
         Subject.getSubject(AccessController.getContext());
      logCreditAccess(currentSubject, taxID, agency);

      // Use Web Services to retrieve the credit score from the
```

```
      // specified agency.
      return getCreditScore(taxID, agency);
   }

   ...
}
```

3.3.1.1.1.2.3 Task Invocation

The LoanCheckerBean is a stateless session EJB that has one method that is used to retrieve the credit scores for one tax ID from three different agencies. It uses three threads to accomplish this, including the EJB thread.

While the EJB thread is retrieving one credit score, two other threads are retrieving the other two scores.

```
class LoanCheckerBean {

   public long[] getCreditScores(long taxID) {
      // Retrieve our singleton threadpool, but wrap it in
      // a ExecutorCompletionService
      ExecutorCompletionService<Long> threadPool =
         new ExecutorCompletionService<Long>(
            ExecutorAccessor.getThreadPool());

      // Get an instance to a ContextService
      InitialContext ctx = new InitialContext();
      ContextService ctxSvc = (ContextService)
         ctx.lookup("java:comp/env/concurrent/AllContexts");

      Class[] callableClass = {Callable.class};

      // Use this thread to retrieve one credit score, and
      // use two other threads to process the other two scores.
      // Since we are using a custom executor (component-managed) and
      // because our tasks depend upon the context in which this
      // method is running, we use a contextual task.
      CreditScoreTask agency1 = new CreditScoreTask(taxID, 1);
      CreditScoreTask agency2 = (CreditScoreTask)
         ctxSvc.createContextObject(
            new CreditScoreTask(taxID, 2), callableClass);
      CreditScoreTask agency3 = (CreditScoreTask)
         ctxSvc.createContextObject(
            new CreditScoreTask(taxID, 3), callableClass);

      threadPool.submit(agency2);
      threadPool.submit(agency3);

      long[] scores = {0,0,0};

      try {
         // Retrieve one credit score on this thread.
         scores[0] = agency1.call();

         // Retrieve the other two credit scores
         scores[1] = threadPool.take().get();
         scores[2] = threadPool.take().get();

      } catch (InterruptedException e) {
         // The app may be shutting down.
      } catch (ExecutionException e) {
         // There was an error retrieving one of the asynch scores.
      }
```

```
        return scores;
    }
}
```

3.3.2 Application Assembler's Responsibilities

The Application Assembler (J2EE.2.10.3) is responsible for assembling the application components into an Enterprise Archive (.ear) and providing assembly instructions that describe the dependencies to the administered objects.

3.3.3 Deployer's Responsibilities

The Deployer (J2EE.2.10.4) is responsible for deploying the application components into a specific operational environment. In the terms of this specification, the Deployer installs the application components and maps the dependencies defined by the Application Component Provider and Application Assembler to administered objects with the properly defined attributes. See J2EE.5.5.2 for details.

3.3.3.1 ContextService Configuration Attributes

Each `ContextService` may support one or more runtime behaviors as specified by the attributes in this section. `ContextService` configuration attributes that are a requirement of this specification are as indicated. If a configuration attribute is not required, it is considered optional. Implementations that declare support for of the optional behaviors must pass the Technology Compatibility Kit (TCK) for that configuration attribute.

3.3.3.1.1 Other Container Contexts

All objects created by a `ContextFactory` instance are required to propagate Java EE container context information (see section 2.3) to the methods invoked on the proxied object.

Java EE Product Providers may add any additional container contexts to the managed `ThreadFactory` and provide the means for configuration those contexts in any way so long as these contexts do not violate the required aspects of this specification.

3.3.4 Java EE Product Provider's Responsibilities

The Java EE Product Provider's responsibilities are as defined in J2EE.5.5.3 and must provide implementation that provide the behaviors defined in section 3.3.3.1 and the following:

- All invocation handlers for the contextual proxy implementation must implement `java.io.Serializable`.
- All invocations to any of the proxied interface methods will fail with a `java.lang.IllegalStateException` exception if the application component is not started or deployed.

3.3.4.1 Configuration Examples

This section and subsections illustrate some examples how a Java EE Product Provider could configure a `ContextService` and the possible options that such service could provide.

The `ContextService` can be used directly by application components by using resource environment references or providers may choose to use the context information supplied as default context propagation policies for a `ManagedExecutorService`, `ManagedScheduledExecutorService` or `ManagedThreadFactory`. The configuration examples covered in sections, 3.1.4.1, 3.2.4.1 and 3.4.4.1 all refer to one of the following ContextService configuration examples that follow.

Each of the examples has the following attributes:

- **Name**: An arbitrary name of the service for the deployer to use as a reference.
- **JNDI name**: The arbitrary, but required, name to identify the service instance. The deployer uses this value to map the service to the component's resource environment reference.
    - **Security**: If enabled, propagate the container security principal.
    - **Locale**: If enabled, the locale from the container thread is propagated.
    - **Custom**: If enabled, custom, thread-local data is propagated.

3.3.4.1.1 All Contexts

ContextService	
Name:	All Contexts
JNDI Name:	concurrent/cs/AllContexts
Context Info:	☑ Security ☑ Locale ☑ Custom

Table 6 - All Contexts Configuration Example

3.3.4.1.2 OLTP Contexts

ContextService	
Name:	OLTP Contexts
JNDI Name:	concurrent/cs/OLTPContexts
Context Info:	☑ Security ☐ Locale ☑ Custom

Table 7 - OLTP Contexts Configuration Example

3.3.4.1.3 No Contexts

ContextService	
Name:	No Contexts
JNDI Name:	concurrent/cs/NoContexts
Context Info:	☐ Security ☐ Locale ☐ Custom

Table 8 - No Contexts Configuration Example

3.3.5 Transaction Management

Contextual dynamic proxies support user-managed global transactions with similar semantics to EJB bean-managed transaction demarcation (see the Enterprise JavaBeans specification). Proxy methods suspend any transactional context on the thread and allow components to manually control global transaction demarcation boundaries. Context objects may optionally begin, commit and roll-back a transaction. See J2EE.4 for details on transaction management in Java EE.

Transaction management can be disabled on the proxy instance using a context property (see the Javadoc for the `javax.util.concurrent.ContextService` interface for details and examples). When disabled, the transaction (if any) currently in progress on the thread (for example, the transaction that the container is managing) will not be suspended and any resources used by the task will be enlisted.

3.3.5.1 Java EE Product Provider Requirements

This subsection describes the transaction management requirements of a `ContextService` implementation when transaction management is enabled (this is the default behavior).

1. The `javax.transaction.UserTransaction` interface must be made available in the local JNDI namespace as environment entry: `java:comp/UserTransaction` (J2EE.5.7 and J2EE.4.2.1.1)
2. All resource managers must enlist with a `UserTransaction` instance when a transaction is active using the `begin()` method.
3. The executor is responsible for coordinating commits and rollbacks when the transaction ends using `commit()` and `rollback()` methods.

3.3.5.2 Application Component Provider's Requirements

This subsection describes the transaction management requirements of each task provider's implementation when transaction management is enabled (this is the default behavior).

1. A task instance that starts a transaction must complete the transaction before starting a new transaction.

2. The task provider uses the `javax.transaction.UserTransaction` interface to demarcate transactions.
3. Transactions are demarcated using the `begin()`, `commit()` and `rollback()` methods of the `UserTransaction` interface.
4. While an instance is in an active transaction, resource-specific transaction demarcation APIs must not be used (e.g. if `a javax.sql.Connection` is enlisted in the transaction instance, the `Connection.commit()` and `Connection.rollback()` methods must not be used).
5. The task instance must complete the transaction before the task method ends.

See section 3.1.8.2.1 for an example on how to use a UserTransaction within a task.

3.4 ManagedThreadFactory

The `javax.util.concurrent.ManagedThreadFactory` allows applications to create thread instances from a Java EE Product Provider without creating new `java.lang.Thread` instances directly. This object allows Application Component Providers to use custom executors such as the `java.util.concurrent.ThreadPoolExecutor` when advanced, specialized execution patterns are required.

Java EE Product Providers can provide custom `Thread` implementations to add management capabilities and container contextual information to the thread.

3.4.1 Application Component Provider's Responsibilities

Application Component Providers (application developers) (J2EE2.10.2) use a `javax.util.concurrent.ManagedThreadFactory` instance to create manageable threads. `ManagedThreadFactory` instances are retrieved using the Java Naming and Directory Interface (JNDI) Naming Context (J2EE.5).

The Application Component Provider may use resource environment references to obtain references to a `ManagedThreadFactory` instance as follows:

- Assign an entry in the application component's environment to the reference using the reference type of: `javax.util.concurrent.ManagedThreadFactory`. (See J2EE.5.5.1.2 for information on how resource environment references are declared in the deployment descriptor.)
- This specification recommends, but does not require, that all resource environment references be organized in the appropriate subcontext of the component's environment for the resource type. For Example, all `ManagedThreadFactory` references should be declared in the `java:comp/env/concurrent` subcontext.
- Look up the administered object in the application component's environment using JNDI (J2EE.5).

Note *– Future versions of Java EE may have alternatives to JNDI such as annotations and resource injection. These alternatives may be added once this specification is incorporated into an umbrella JSR.*

- New threads are created using the `newThread(Runnable r)` method on the `java.util.concurrent.ThreadFactory` interface.
- The application component thread has permission to interrupt the thread. All other modifications to the thread are subject to the security manager, if present.
- All `Threads` are contextual. When the thread is started using the `Thread.start()` method, the `Runnable` that is executed will run with the context of the application component instance that created the `ManagedThreadFactory` instance.

Note *– The `ManagedThreadFactory` instance may be invoked from several threads in the application component, each with a different container context (for example, user identity). By always applying the context of the `ManagedThreadFactory` creator, each thread has a consistent context. If a different context is required for each thread, use the `ContextService` to create a contextual object (see section 3.3).*

- If the component that created the `ManagedThreadFactory` instance is stopped, all subsequent calls to `newThread()` must throw a `java.lang.IllegalStateException`.

Note *– The use of resource environment references is best practice. Using the JNDI name of the `ManagedThreadFactory` directly is allowed.*

3.4.1.1 Usage Example

In this example, an application component uses a background daemon task to dump in-memory events to a database log, similar to the timer usage example in section 3.2.1.1.1.

The attributes of the `ManagedThreadFactory` reference is documented using the `<description>` tag within the deployment descriptor of the application component and later mapped by the Deployer.

3.4.1.1.1 Logger Task

The Logger Task is a long-running task that has the same life-cycle as the servlet. It continually monitors a queue and waits for events to a database log. Its lifecycle is controlled using a `javax.servlet.ServletContextListener`.

3.4.1.1.1.1 Resource Environment Reference

The following resource environment reference is added to the web.xml file for the web component. The description reflects the desired configuration attributes (see 3.1.3.1):

Note *– Using the description for documenting the configuration attributes of the administered object is optional. The format used here is only an example. Future revisions of Java EE specifications may formalize usages such as this.*

```
<resource-env-ref>
   <description>
```

```
      This ManagedThreadFactory is used to create a thread for for the
      application's logger task.
      This ManagedThreadFactory has the following requirements:
         Context Info:    Local Namespace
   </description>
   <resource-env-ref-name>
      concurrent/LoggerThreadFactory
   </resource-env-ref-name>
   <resource-env-ref-type>
      javax.util.concurrent.ManagedThreadFactory
   </resource-env-ref-type>
</resource-env-ref>
```

3.4.1.1.1.2 Task Definition

The task itself simply uses a resource-reference to a JDBC data source, and uses a connect/use/close pattern when invoking the Datasource.

```
public class LoggerTask implements Runnable {
   public void run() {

      // Lookup the DataSource (the local naming
      // environment is inherited from the parent
      // component (servlet).
      InitialContext ctx = new InitialContext();
      DataSource ds = (DataSource)
         ctx.lookup("java:comp/env/jdbc/Logger");

      // Wait for data and log it.
      while(!Thread.interrupted()) {
         logEvents(getData(), ds);
      }
   }
}

void logEvents(Collection data, DataSource ds) {

   // Iterate through the data and log each row.
   for (…) {
      Connection con = ds.getConnection();

      // Write the data using our connection.
      …

      // Commit and close.
      con.commit();
      con.close();
   }
}
```

3.4.1.1.1.3 Task Submission

The task is started and stopped by a `javax.servlet.ServletContextListener`.

```
public class CtxListener implmments ServletContextListener {
   Thread loggerThread = null;
   public void contextInitialized(ServletContextEvent scEvent) {

      // Lookup the executor and submit our task.
      InitialContext ctx = new InitialContext();
      ManagedThreadFactory threadFactory= (ManagedThreadFactory)
         ctx.lookup("java:comp/env/concurrent/LoggerThreadFactory");
```

```
        LoggerTask logger = new LoggerTask();
        Thread loggerThread = threadFactory.newThread(logger);
        loggerThread.start();
    }

    public void contextDestroyed(ServletContextEvent scEvent) {

        // Interrupt our logger task since it is no longer available.
        // Note:  The server will do this for us as well.
        if(loggerThread!=null) {
            synchronized(loggerThread) {
                loggerThread.interrupt();
            }
        }
    }
}
```

3.4.2 Application Assembler's Responsibilities

The Application Assembler (J2EE.2.10.3) is responsible for assembling the application components into an Enterprise Archive (.ear) and providing assembly instructions that describe the dependencies to the administered objects.

3.4.3 Deployer's Responsibilities

The Deployer (J2EE.2.10.4) is responsible for deploying the application components into a specific operational environment. In the terms of this specification, the Deployer installs the application components and maps the dependencies defined by the Application Component Provider and Application Assembler to administered objects with the properly defined attributes. See J2EE.5.5.2 for details.

3.4.3.1 ManagedThreadFactory Configuration Attributes

Each managed `ManagedThreadFactory` may support one or more runtime behaviors as specified by the attributes in this section. `ManagedThreadFactory` configuration attributes that are a requirement of this specification are as indicated. If a configuration attribute is not required, it is considered optional. Implementations that declare support for of the optional behaviors must pass the Technology Compatibility Kit (TCK) for that configuration attribute.

3.4.3.1.1 Other Container Contexts

All threads created by a `ManagedThreadFactory` instance are required to propagate container context information (see section 2.3) to the thread's `Runnable`.

Java EE Product Providers may add any additional container contexts to the managed `ManagedThreadFactory` and provide the means for configuration those contexts in any way so long as these contexts do not violate the required aspects of this specification.

3.4.4 Java EE Product Provider's Responsibilities

The Java EE Product Provider's responsibilities are as defined in J2EE.5.5.3 and must provide implementation that provide the behaviors defined in section 3.4.3.1 with the following additions:

- If the component that created the `ManagedThreadFactory` instance is stopped, all threads that it has created using the `newThread()` method are interrupted.

***Note** – The intent is to prevent access to components that are no longer available.*

3.4.4.1 Configuration Examples

This section and subsections illustrate some examples how a Java EE Product Provider could configure a `ManagedThreadFactory` and the possible options that such service could provide.

A `ManagedThreadFactory` can be used directly by application components by using resource environment references or providers may choose to use the context information supplied as default context propagation policies for `ManagedExecutorService`, or `ManagedScheduledExecutorService` instances. The configuration examples covered in sections, 3.1.4.1 and 3.2.4.1 all refer to one of the following `ManagedThreadFactory` configuration examples that follow.

Each of the examples has the following attributes:

- **Name**: An arbitrary name of the service for the deployer to use as a reference.
- **JNDI name**: The arbitrary, but required, name to identify the service instance. The deployer uses this value to map the service to the component's resource environment reference.
- **Context**: A reference to a `ContextService` instance (see section 3.3). The context service can be used to define the context to propagate to the threads when running tasks. Having multiple `ContextService` instances, each with a different policy may be desirable for some implementations.
- **Priority**: The priority to assign to the thread (the higher the number, the higher the priority). See the `java.lang.Thread` Javadoc for details on how this value can be used.

3.4.4.1.1 Normal Threads

This configuration example illustrates a typical `ManagedThreadFactory` that creates normal priority threads with all available context information.

ManagedThreadFactory	
Name:	Normal Threads
JNDI Name:	concurrent/tf/normal
Context:	concurrent/cf/AllContexts
Priority:	5 (Normal)

Table 9 - Normal ManagedThreadFactory Configuration Example

3.4.4.1.2 OLTP Threads

This configuration example describes a `ManagedThreadFactory` that creates threads with a higher than normal priority that can be used for OLTP-type requests.

ManagedThreadFactory	
Name:	OLTP Threads
JNDI Name:	concurrent/tf/OLTP
Context:	concurrent/cf/AllContexts
Priority:	6

Table 10 - OLTP ManagedThreadFactory Configuration Example

3.4.4.1.3 Batch Threads

This configuration example describes a `ManagedThreadFactory` that creates lower-priority threads that can be used for background tasks, such as batch jobs.

ManagedThreadFactory	
Name:	Batch Threads
JNDI Name:	concurrent/tf/batch
Context:	concurrent/cf/AllContexts
Priority:	4

Table 11 - Batch ManagedThreadFactory Configuration Example

3.4.5 System Administrator's Responsibilities

The System Administrator (J2EE.2.10.5) is responsible for monitoring and overseeing the runtime environment. In the scope of this specification, these duties may include:

- Monitoring for hung tasks.
- Monitoring resource usage (for example, threads and memory).

See section 4, "Managed Objects" for details on how to monitor thread usage.

3.4.6 Transaction Management

`ManagedThreadFactory` implementations must support user-managed global transactions with similar semantics to EJB bean-managed transaction demarcation (see the Enterprise JavaBeans specification). User-managed transactions allow components to manually control global transaction demarcation boundaries. Task implementations may optionally begin, commit and roll-back a transaction. See J2EE.4 for details on transaction management in Java EE.

3.4.6.1 Java EE Product Provider Requirements

This subsection describes the transaction management requirements of a `ManagedThreadFactory` implementation.

4. The `javax.transaction.UserTransaction` interface must be made available in the local JNDI namespace as environment entry: `java:comp/UserTransaction` (J2EE.5.7 and J2EE.4.2.1.1)

5. All resource managers must enlist with a `UserTransaction` instance when a transaction is active using the `begin()` method.
6. The executor is responsible for coordinating commits and rollbacks when the transaction ends using `commit()` and `rollback()` methods.

3.4.6.2 Application Component Provider's Requirements

This subsection describes the transaction management requirements of each task provider's implementation.

6. A task instance that starts a transaction must complete the transaction before starting a new transaction.
7. The task provider uses the `javax.transaction.UserTransaction` interface to demarcate transactions.
8. Transactions are demarcated using the `begin()`, `commit()` and `rollback()` methods of the `UserTransaction` interface.
9. While an instance is in an active transaction, resource-specific transaction demarcation APIs must not be used (e.g. if `a javax.sql.Connection` is enlisted in the transaction instance, the `Connection.commit()` and `Connection.rollback()` methods must not be used).
10. The task instance must complete the transaction before the task method ends.

See section 3.1.8.2.1 for an example on how to use a UserTransaction within a task.

3.5 Distributable ManagedExecutorService

Distributable `ManagedExecutorService` implementations allow distribution of tasks to peer or child application servers. Users of the distributed `ManagedExecutorService` can submit serializable tasks and the `ManagedExecutorService` can route these tasks to other servers to complete the task and return a serializable result.

There are two types of distributable `ManagedExecutorServices` (see section 3.1.3.1.1): Distributable and Distributable with Affinity. A Distributable executor allows tasks to run on any process. Distributable executors allow a great amount of flexibility to the implementer to route tasks based on classification. A Distributable with Affinity executor allows classifying a task one time and routes all tasks to the same process (may be the local process or a single remote process).

3.5.1 Distributable Requirements

Each `ManagedExecutorService` implementation that has a run location of "Distributable" must support the following:

1. Support the requirements of the `ManagedExecutorService` described in section 3.1.
2. Tasks submitted to the executor may optionally route the request to any application server process including the source application server (the server that submits the task). The source executor is known as the master and the downstream executors are known as the slaves. An executor can act in both roles.

3. All tasks submitted to an executor must implement the `java.io.Serializable` interface. If the task does not implement this interface, the executor will throw a `java.util.concurrent.RejectedExecutionException` with a cause of `java.io.NotSerializableException`.
4. All `Callable` task result types must extend the `java.io.Serializable` interface. If the result type does not extend this interface, the result of the Future may be a `java.io.NotSerializableException` exception.
5. Tasks submitted to the master executor are owned by the executor instance. If the master executor becomes unavailable, the submitted tasks are cancelled.
6. All tasks are not considered to be idempotent. If a slave executor becomes unavailable, all Futures for the tasks submitted to that executor that have not yet started will be cancelled.
7. If a slave executor becomes unavailable, and the task has started, the result of the task's Future will throw a `javax.util.concurrent.ExecutorNotAvailableException` exception.

3.5.2 Distributable with Affinity Requirements

Each `ManagedExecutorService` implementation that has a run location of "Distributable with Affinity" must support the following:

1. Support the requirements of the `ManagedExecutorService` with a run location of "Distributable" described in section 3.5.1.
2. All tasks submitted to the executor must route the request to the same application server process. The source executor is known as the master and the downstream executors are known as the slaves. An executor can act in both roles if the request is routed to the local process.
3. If the slave executor becomes inactive, subsequent submissions to the master will throw a `java.util.concurrent.RejectedExecutionException` with a cause of `javax.util.concurrent.ExecutorUnavailableException`.

4 Managed Objects

This specification extends the J2EE Management Model specification (JSR-77) by defining the format, semantics of the managed objects required by all compliant implementations of this specification.

Server-managed `ManagedExecutorService` and `ManagedScheduledExecutorService` instances must provide management capabilities as defined in this specification. The intent is to allow operators and administrators to monitor and identify problems.

All managed objects extend the J2EEManagedObject model, described in section 7.3.1 of the J2EE Management Model specification. This chapter contains the models and metamodels that specify the format, semantics and relationship of the managed objects required by all compliant implementations of this specification.

Since each managed object extends the J2EEManagedObject model, the objectName, stateManageable, statisticsProvider and eventProvider attributes must be supported. The StateManageable, StatisticsProvider and EventProvider may be optionally implemented and these attributes therefore can return false.

If the eventProvider attribute returns true, the Notification model as described in the following subsections must be implemented.

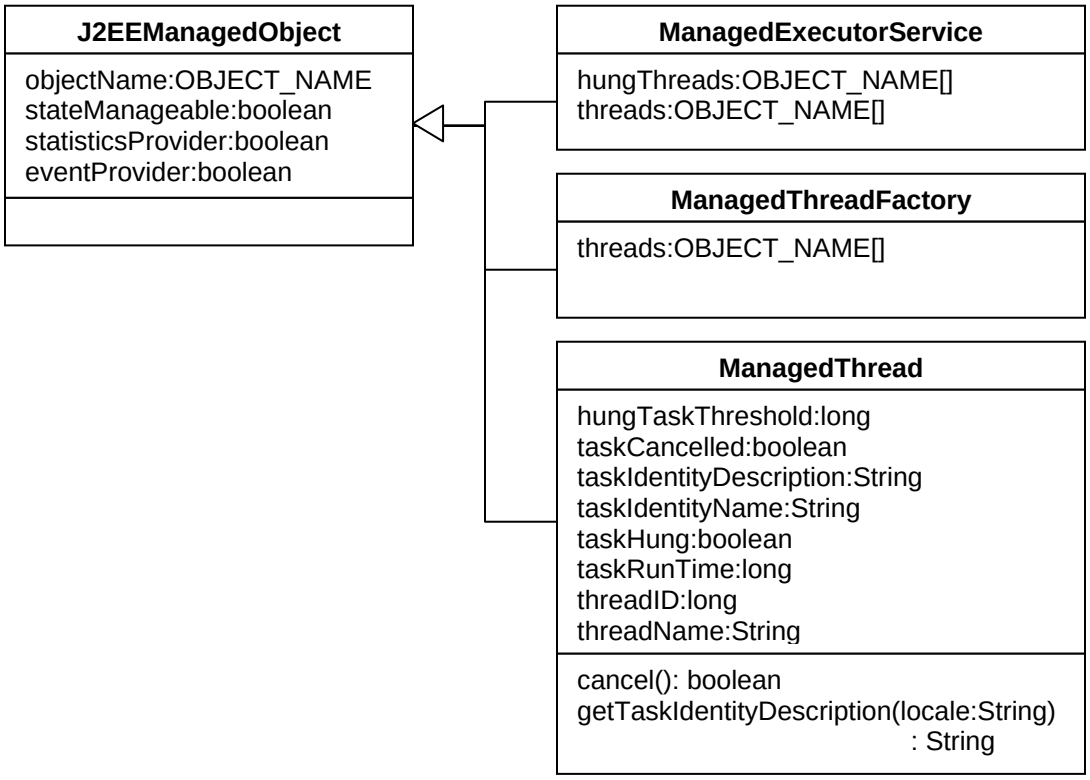

Figure 5 - J2EEManagedObject detail with inheritance overview

Attributes which identify managed objects are specified to be of the OBJECT_NAME type. The type identifier OBJECT_NAME refers to a formatted string those syntax is

defined by the specification of the objectName attribute of the J2EEManagedObject model (see section JSR77.3.1.1.1 on page 21). The OBJECT_NAME string type provides a unique identifier for a managed object within a J2EE management domain. Every managed object has a name attribute whose value complies with the OBJECT_NAME syntax.

4.1 Object Name Key Properties

Each J2EEManagedObject is required to have a set of mandatory keys as part of the object name. This specification defines the values for some of the keys.

- j2eeType – The values for the j2eeType have been extended in this specification to include new types as defined in Table 12 and are further described in their own respective sections.
- name – Specifies the name of the J2EEManagedObject as described in section 7.3.1.1.1.3 of the J2EE Management Model specification.
- parent-j2eeType – The parent J2EE types are defined as follows in Table 12:

Managed Object j2eeType	Required parent-j2eeType keys
ManagedExecutorService	J2EEServer
ManagedThreadFactory	J2EEServer
ManagedThread	J2EEServer

Table 12 - Managed Object j2eeTypes and required <parent-j2eeType> keys

4.2 ManagedExecutorService extends J2EEManagedObject

The ManagedExecutorService managed object type represents one instance of a `javax.util.concurrent.ManagedExecutorService` (and subclasses such as `javax.util.concurrent.ManagedScheduledExecutorService`) Administered Object (see section 3.1 and 3.2). There must be one managed object that implements the ManagedExecutorService model for each executor instance.

4.2.1 Attribute Detail

4.2.1.1 hungTaskThreads

```
OBJECT_NAME[] hungTaskThreads
```

- supplier cardinality 0..*
- access: read-only

Returns all ManagedThread managed objects that have the taskHung attribute set to true for the ManagedExecutorService.

4.2.1.2 threads

```
OBJECT_NAME[] threads
```

- supplier cardinality: 0..*
- access: read-only

Returns all ManagedThread managed objects managed by the ManagedExecutorService.

4.2.2 Notifications

This section specifies the Notification model (as described in chapter 3 of the J2EE Management Model specification) that the ManagedExecutorService managed object must implement if it supports the EventProvider model.

4.2.2.1 Notification Types

The following event types must be supported (also see Figure 6 - ManagedThread Managed Object Notification State Diagram):

- `task.state.hung` - A task on the thread is now hung.
- `task.state.cancelled` - A thread has entered the cancelled state.
- `task.state.released` - A previously hung task is no longer hung.

4.2.2.2 Notification userData

Notifications as defined in the J2EE Management Model specification may optionally include an Object in the `userData` attribute of the Notification. Each notification as defined in section 4.2.2.1 must include a `userData` object of type `java.util.Properties`. The properties object must contain the following keys and values:

- `managedthread` – The object name of the ManagedThread managed object that the event pertains to.

4.3 ManagedThreadFactory extends J2EEManagedObject

The `ManagedThreadFactory` managed object type represents one instance of a managed `javax.util.concurrent.ManagedThreadFactory` Administered Object (see section 3.4). There must be one managed object that implements the ManagedThreadFactory model for each `ManagedThreadFactory` instance.

4.3.1 Attribute Detail

4.3.1.1 threads

```
OBJECT_NAME[] threads
```

- supplier cardinality: 0..*
- access: read-only

Returns all ManagedThread managed objects created by the ManagedThreadFactory instance.

4.3.2 Notifications

This section specifies the Notification model (as described in chapter 3 of the J2EE Management Model specification) that the ManagedThreadFactory managed object must implement if it supports the EventProvider model.

4.3.2.1 Notification Types

The following event types must be supported:

- `threadfactory.newthread` – A new thread has been created.

4.3.2.2 Notification userData

Notifications as defined in the J2EE Management Model specification may optionally include an Object in the `userData` attribute of the Notification. Each notification as defined in section 4.2.2.1 must include a `userData` object of type `java.util.Properties`. The properties object must contain the following keys and values:

- `managedthread` – The object name of the ManagedThread managed object that the event pertains to.

4.3.3 Usage Example

This example illustrates how a JMX client can locate the ManagedExecutorService MBean and identify and act on any hung threads.

```
public void checkForHungThreads(String domainName,
   MBeanServer mbeanServer, String name) {

   // Using the specified domain name (usually vendor-specific),
   // the MbeanServer instance and the name of the executor,
   // locate the MBean for the ManagedExecutorService
   ObjectName query = new ObjectName(domainName +
      ":*,j2eeType=ManagedExecutorService,name=" + name);
   Set mbeanNames = mbeanServer.queryNames(query, null);

   if(mbeanNames.size()==0) {
      // Unable to find the MBean
      return;
   }
   ObjectName mesMBean = (ObjectName)mbeanNames.iterator().next();

   // See if we have any hung threads.  If we do,
   // do something interesting, like an SNMP action.
   String[] hungThreads = (String[])
      mbeanServer.getAttribute(mesMBean, "hungThreads");

   for(int i=0;i<hungThreads.length;i++) {
      // For each hung thread, do something...
   }
}
```

4.4 ManagedThread extends J2EEManagedObject

The ManagedThread managed object type represents one instance of a thread created by a `ManagedThreadFactory` or a server-managed `ManagedExecutorService` or `ManagedScheduledExecutorService`. These Administered objects typically have one or

more threads available to run tasks. Each of these threads will have a ManagedThread object type associated with it.

4.4.1 Attribute Detail

4.4.1.1 hungTaskThreshold

```
long hungTaskThreshold
```

- access: read/write

The amount of time in milliseconds that a task can execute before it is considered hung.

4.4.1.2 taskCancelled

```
boolean taskCancelled
```

- access: read-only

If true, a task has been cancelled. Each time a thread executes a new task, this attribute will reset to false.

4.4.1.3 taskIdentityDescription

```
String taskIdentityDescription
```

- access: read-only

The identity description of the identifiable task using the locale of the process in which the thread is running. If a task is executing on the thread and the task implements the `javax.util.concurrent.Identifiable` interface, the value of `Identifiable.getIdentityDescription(Locale locale)` is returned. If no task is executing, then the string "null" is returned. If a task is executing but does not implement the `Identifiable` interface, then the result of `toString()` is returned.

Note *– This method should typically invoke the task's object instance directly and not cache the results. Task implementations should not rely on any execution context and should simply return instance data.*

4.4.1.4 taskIdentityName

```
String taskIdentityName
```

- access: read-only

The identity name of the identifiable task. If a task is executing on the thread and the task implements the `javax.util.concurrent.Identifiable` interface, the value of `Identifiable.getIdentityName()` is returned. If no task is executing, then the string "null" is returned. If a task is executing but does not implement the `Identifiable` interface, then the result of `toString()` is returned.

Note *– This method should typically invoke the task's object instance directly and not cache the results. Task implementations should not rely on any execution context and should simply return instance data.*

4.4.1.5 taskHung

`boolean taskHung`

- access: read-only

If true, the currently running task's `runTime` has exceeded the `hungThreadThreshold` value. Each time a thread executes a new task, this attribute will reset to false.

4.4.1.6 taskRunTime

`long taskRunTime`

- access: read-only

The number of milliseconds that the current task has been executing on this thread or 0 if no task is currently executing.

4.4.1.7 threadID

`long threadID`

- access: read-only

The identifier of this thread (see `java.lang.Thread.getId()` in the Java SE specification);

4.4.1.8 threadName

`String threadName`

- access: read-only

The name of this thread (see `java.lang.Thread.getName()` in the Java SE specification);

4.4.2 Operation Detail

4.4.2.1 cancelTask

`boolean cancelTask()`

If the task running on the thread is hung, this method will cancel (and interrupt) the task that is currently running on this thread.

4.4.2.2 getTaskIdentityDescription()

`String getTaskIdentityDescription(String locale)`

The locale-specific identity description using the format defined by `java.util.Locale.toString()` method.. If a task is executing on the thread and the task implements the `javax.util.concurrent.Identifiable` interface, the value of `Identifiable.getIdentityDescription(Locale locale)` is returned. If no task is executing, then the string "null" is returned. If a task is executing but does not implement the `Identifiable` interface, then the result of `toString()` is returned and the locale is ignored.

***Note** – This method should typically invoke the task's object instance directly and not cache the results. Task implementations should not rely on any execution context and should simply return instance data.*

4.4.3 Usage Example

This example illustrates how a JMX client can locate the ManagedExecutorService MBean and add automatically monitor all hung threads. In this case, the JMX client is a servlet and the listener wants to log the notification to a database and trigger an SNMP action.

Since the listener is invoked by the JMX service, the thread in which the NotificationListener is invoked will not have access to the Java EE container. A `ContextService` proxy is used to run the listener methods within the container context.

```
// Create a custom fulter to only worry about task state.
private NotificationFilter filter = new NotificationFilter() {
   public boolean isNotificationEnabled(Notification notification) {
      return notification.getType().startsWith("task.state.");    }
};

public void monitorHungThreads(String domainName,
   MBeanServer mbeanServer, String name) {

   // Using the specified domain name (usually vendor-specific),
   // the MbeanServer instance and the name of the executor,
   // locate the MBean for the ManagedExecutorService
   ObjectName query = new ObjectName(domainName +
      ":*,j2eeType=ManagedExecutorService,name=" + name);
   Set mbeanNames = mbeanServer.queryNames(query, null);

   if(mbeanNames.size()==0) {
      // Unable to find the MBean
      return;
   }
   ObjectName mesMBean = (ObjectName)mbeanNames.iterator().next();

   // Create a NotificationListener instance and wrap it with
   // a ContextService proxy to guarantee that when invoked,
   // the listener methods will have access to the container
   // context information.
   InitialContext ctx = new InitialContext();
   ContextService ctxSvc =(ContextService)
      ctx.lookup("java:comp.env/concurrent/AllContexts");

   NotificationListener listener = new NotificationListener() {
      public void handleNotification(Notification n, Object data) {
         String type = n.getType();

         if("task.state.hung".equals(type)) {
            Properties p = (Properties)n.getUserData();
            String threadObjName = p.getProperty("managedthread");
            // Notify the administrator of the hung task.
            processHungTask(mbeanServer, threadObjName);

         } else if("task.state.released".equals(type)) {
            Properties p = (Properties)n.getUserData();
            String threadObjName = p.getProperty("managedthread");
            // Notify the administrator that our task is no longer
            // hung…
            ...
         }
      }
   };
```

```
   NotificationListner listenerWithCtx =
      ctxSvc.createContextObject(listener, new
         class[]{NotificationListener.class});

   // Add our context-aware NotificationListener and a filter
   mbeanServer.addNotificationListener(listenerWithCtx, filter, null);
}

void processHungTask(MBeanServer mbeanServer, String objName) {
   // Identify the task's properties, and notify the
   // administrator ObjectName hungThread = new ObjectName(objName);
   Long threadID = (Long) mbeanServer.getAttribute(
      hungThread, "threadID");
   String threadName =(String) mbeanServer.getAttribute(
      hungThread, "threadName");
   String taskIdentity = (String)mbeanServer.getAttribute(
      hungThread, "taskIdentityName");

   // Send an SMTP event with the thread and task info.
   // Log the info to a database
}
```

4.5 Notifications

This section specifies the Notification model (as described in chapter 3 of the J2EE Management Model specification) that the ManagedThread managed object must implement if it supports the EventProvider model.

4.5.1.1 Notification Types

The following event types must be supported:

- `task.state.hung` - A task on the thread is now hung.
- `task.state.cancelled` - A thread has entered the cancelled state.
- `task.state.released` - A previously hung task is no longer hung.

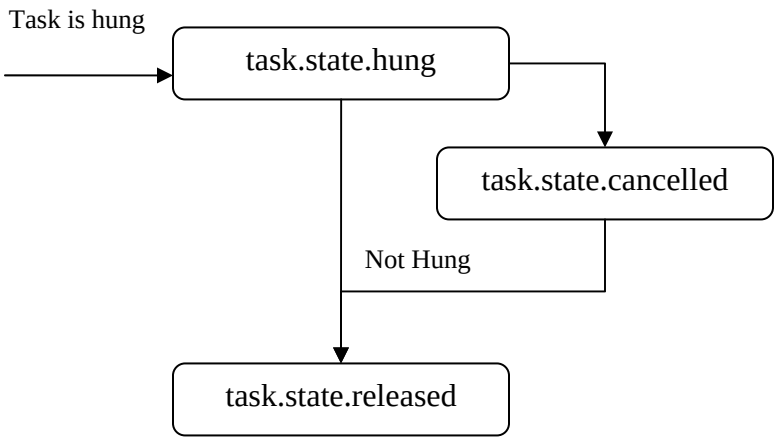

Figure 6 - ManagedThread Managed Object Notification State Diagram

Appendix A Change History

This appendix lists the significant changes that have been made during the development of the Concurrency Utilities for Java EE specification.

A.1 Version 0.1: Early Draft Preview

This is the initial public version of this specification.