Concurrency Utilities for Java EE

Version 1.0

Proposed Final Draft

Document Date: February 22, 2013

Please send comments to users@concurrency-ee-spec.java.net

Specification: JSR-236 Concurrency Utilities for Java EE ("Specification")
Version: 1.0
Status: Proposed Final Draft
Release: February 22, 2013

acquisitions).

REPORT
You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide Oracle with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Oracle a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

GENERAL TERMS
Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

Contents

List of Figures

List of Tables

1. Introduction

1.1 Overview

Java™ Platform, Enterprise Edition (Java EE) server containers such as the enterprise bean or web component container do not recommend 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 2.11 of the Java EE 7 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:

- 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)
- Java EE Connector Architecture
- Java Platform Standard Edition
- Java 2 Platform, Enterprise Edition, Management Specification (JSR-77)
- Java Naming and Directory Interface ™

- Java Transaction API
- Java Transaction Service
- JDBC™ API
- Java Message Service (JMS)
- Enterprise JavaBeans ™

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. The expert group includes the following members: Adam Bien, Marius Bogoevici (RedHat), Cyril Bouteille, Andrew Evers, Anthony Lai (Oracle), Doug Lea, David Lloyd (RedHat), Naresh Revanuru (Oracle), Fred Rowe (IBM Corporation), and Marina Vatkina (Oracle).

We would also like to thank former expert group members for their contribution to this specification, including Jarek Gawor (Apache Software Foundation), Chris D. Johnson (IBM Corporation), Billy Newport (IBM Corporation), Stephan Zachwiega (BEA Systems), Cameron Purdy (Tangosol), Gene Gleyzer (Tangosol), and Pierre VignJras.

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 Platform, Standard Edition (Java SE) in the `java.util.concurrent` package.

The Java SE 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 Java SE 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.concurrent` 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.

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 a thread's original context.

The types of contexts to be propagated include JNDI naming context, classloader, and security information. Containers must support propagation of these context types. In addition, containers can choose to support propagation of other types of context.

The relationships between the various Java EE architectural elements, containers and concurrency constructs are shown in Figure 2-1.

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). Four new concurrency services (represented by four dark-gray rectangles) allow application components and Java EE Standard Services to run asynchronous tasks without violating container contracts.

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

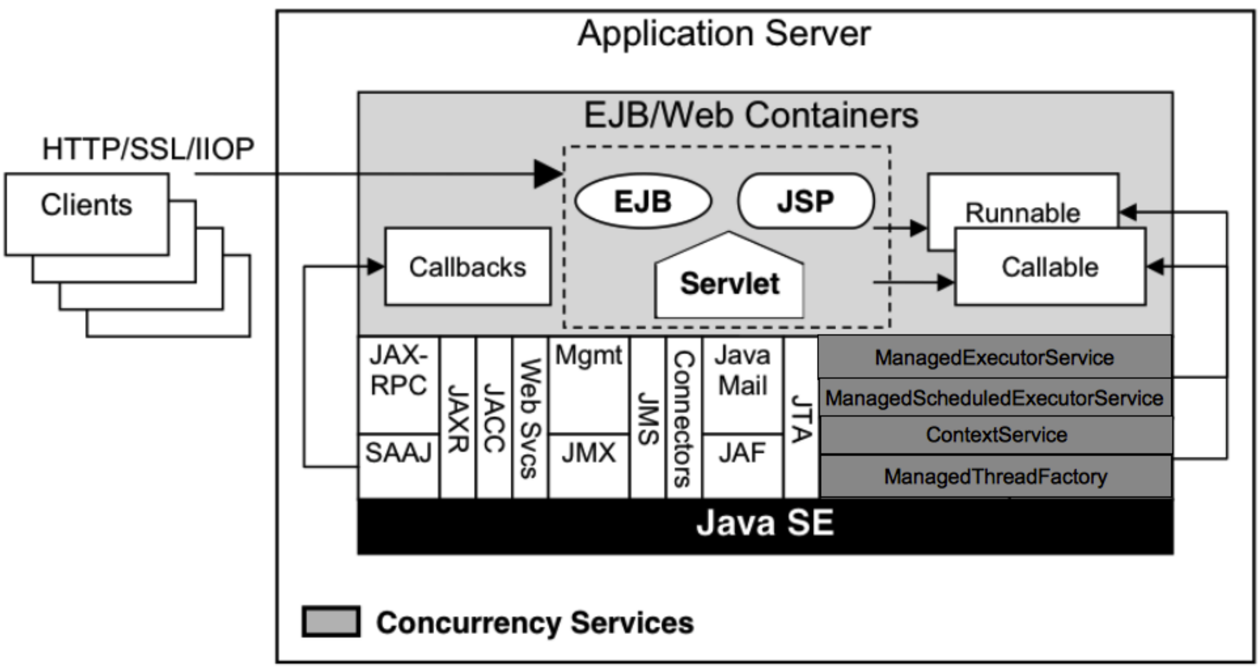

Figure 2-1 Concurrency Utilities for Java EE Architecture Diagram

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()`

2.3.1.1 Optional Contextual Invocation Points

The following callback methods run with unspecified context by default, but may be configured as contextual invocation points if desired:

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

It is not required that container context be propagated to the threads that invoke these methods. This is to avoid the overhead of setting up the container context when it may not be needed in these callback methods. These methods 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).

***Note** - Contextual Objects and Tasks referred here is not the same as the Context object as defined in the Contexts and Dependency Injection for the Java EE platform specification (CDI). See section 2.3.2.1 on using CDI beans as tasks.*

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.

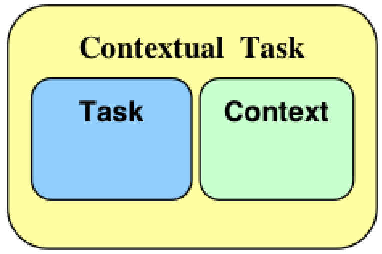

Figure 2-2 Contextual Task

2.3.2.1 Tasks and Contexts and Dependency Injection (CDI)

CDI beans can be used as tasks. Such tasks could make use of injection if they are themselves components or are created dynamically using various CDI APIs. However, application developers should be aware of the following when using CDI beans as tasks:

- Tasks that are submitted to a managed instance of `ExecutorService` may still be running after the lifecycle of the submitting component. Therefore, CDI beans with a scope of `@RequestScoped`, `@SessionScoped`, or `@ConversationScoped` are not recommended to use as tasks as it cannot be guaranteed that the tasks will complete before the CDI context is destroyed.
- CDI beans with a scope of `@ApplicationScoped` or `@Dependent` can be used as tasks. However, it is still possible that the task could be running beyond the lifecycle of the submitting component, such as when the component is destroyed.
- The transitive closure of CDI beans that are injected into tasks should follow the above guidelines regarding their scopes.

2.4 Usage with Java EE Connector Architecture

The Java EE Connector Architecture (Connectors) allows creating resource adapters that can plug into any compatible Java EE application server. The Connectors 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.enterprise.concurrent.ManagedExecutorService` or `java.util.concurrent.ExecutorService` with a `javax.enterprise.concurrent.ManagedThreadFactory`. It is the intent that Connectors `javax.resource.work.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 Managed Objects described in the following sections by looking them up in the JNDI global namespace, through the JNDI context of the accessing application (see section 10.3.2 of the Connectors specification).

2.5 Security

This specification largely defers most security decisions to the container and Java EE Product Provider as defined in the Java EE 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. Managed Objects

This section introduces four programming interfaces for Java EE Product Providers to implement (see EE.2.11 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 managed 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.enterprise.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) (EE2.11.2) 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 (EE.5) or through injection of resource environment references (EE.5.8.1.1).

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.enterprise.concurrent.ManagedExecutorService`. (See EE.5.8.1.3 for information on how resource environment references are declared in the deployment descriptor.)
- Look up the managed object in the application component's environment using JNDI (EE.5.2), or through resource injection by the use of the Resource annotation (EE.5.8.1.1).

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 bound in the `java:comp/env/concurrent` subcontext.

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 classes can optionally implement the `javax.enterprise.concurrent.ManagedTask` interface to provide identity information, execution properties such as whether the task can be run in a remote process, and to register a `javax.enterprise.concurrent.ManagedTaskListener` instance to receive lifecycle events notifications.

Task instances are submitted to a `ManagedExecutorService` instance using any of the defined `submit()`, `execute()`, `invokeAll()`, or `invokeAny()` methods. Task instances will run as an extension of the Java EE container instance that submitted the task and may interact with Java EE resources as defined in other sections of this specification.

It is important for Application Component Providers to identify and document the required behaviors and service-level agreements for each required `ManagedExecutorService`. 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.

3.1.1.1.2 Resource Environment Reference - Reporter Task

The following resource environment reference is added to the web.xml file for the web component. The description reflects the desired configuration attributes (see 3.1.4.1). Alternatively, the `Resource` annotation can be used in the Servlet code.

Note *– Using the description for documenting the configuration attributes of the managed 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:
           Context Info: Local Namespace
    </description>
   <resource-env-ref-name>
        concurrent/LongRunningTasksExecutor
   </resource-env-ref-name>
   <resource-env-ref-type>
        javax.enterprise.concurrent.ManagedExecutorService
   </resource-env-ref-type>
</resource-env-ref>
```

3.1.1.1.3 Task Definition – Reporter Task

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();
       }
   }

   Datasource ds = …;

   void runTransactionReport() {

       try (Connection con = ds.getConnection(); ...) {

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

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

3.1.1.1.4 Task Submission – Reporter Task

The task is started by an 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

   @Resource(name="concurrent/LongRunningTasksExecutor")
   ManagedExecutorService mes;

   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.
       ...
   }
}
```

3.1.1.1.5 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.

3.1.1.1.6 Resource Environment Reference – Builder Tasks

The following resource environment reference is added to the web.xml file for the web component. The description reflects the desired configuration attributes (see 3.1.4.1). Alternatively, the `Resource` annotation can be used in the Servlet code:

***Note** – Using the description for documenting the configuration attributes of the managed 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:
           Context Info: Local Namespace, Security
   </description>
   <resource-env-ref-name>
        concurrent/BuilderExecutor
   </resource-env-ref-name>
   <resource-env-ref-type>
        javax.enterprise.concurrent.ManagedExecutorService
   </resource-env-ref-type>
</resource-env>
```

3.1.1.1.7 Task Definition – Builder Tasks

The task itself simply uses some mechanism such as JDBC queries to retrieve the data from the persistent store. The task implements the `javax.enterprise.concurrent.ManagedTask` interface and supplies an identifiable name through the `IDENTITY_NAME` property to allow system administrators to diagnose problems.

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

   public AccountTask(String reqID, String accountID) {
      this.reqID=reqID;
      this.accountID=accountID;
      execProps = new HashMap<>();
      execProps.put(ManagedTask.IDENTITY_NAME, getIdentityName());
   }

   public AccountInfo call() {
      // Retrieve account info for the account from some persistent store
        AccountInfo info = ...;
        return info;
    }

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

   public Map<String, String> getExecutionProperties() {
      return execProps;
   }

}

public class InsuranceTask implements Callable<InsuranceInfo>, ManagedTask {
  // The ID of the request to report on demand.
  String reqID;
  String accountID;
  Map<String, String> execProps;

  public InsuranceTask (String reqID, String accountID) {
       this.reqID=reqID;
       this.accountID=accountID;
       execProps = new HashMap<>();
       execProps.put(ManagedTask.IDENTITY_NAME, getIdentityName());
   }

  public InsuranceInfo call() {
       // Retrieve the insurance info for the account from some persistent store
       InsurnaceInfo info = ...;
       return info;
     }

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

    public Map<String, String> getExecutionProperties() {
       return execProps;
    }

}
```

3.1.1.1.8 Task Invocation – Builder Tasks

Tasks are created on demand by a request to the servlet from an HTTP client.

```
public class AppServlet extends HTTPServlet implements Servlet {
    // Retrieve our executor instance.
    @Resource(name="concurrent/BuilderExecutor")
    ManagedExecutorService mes;

   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));
       builderTasks.add(new InsuranceTask(reqID, accountID));

       // 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 (EE.2.11.3) is responsible for assembling the application components into a complete Java EE application and providing assembly instructions that describe the dependencies to the managed objects.

3.1.3 Deployer's Responsibilities

The Deployer (EE.2.11.4) 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 managed objects with the properly defined attributes. See EE.5.8.2 for details.

3.1.4 Java EE Product Provider's Responsibilities

The Java EE Product Provider's responsibilities are as defined in EE.5.8.3.

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

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

3.1.4.1 ManagedExecutorService Configuration Attributes

Each `ManagedExecutorService` may support one or more runtime behaviors as specified by configuration attributes. The Java EE Product Provider will determine both the appropriate attributes and the means of configuring those attributes for their product.

3.1.4.2 Configuration Examples

This section and subsections illustrate some examples of how a Java EE Product Provider could configure a `ManagedExecutorService` and the possible options that such a 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. If both Context and ThreadFactory attributes are specified, the Context attribute of the ThreadFactory configuration should be ignored.
- **ThreadFactory**: A reference to a `ManagedThreadFactory` instance (see section 3.4). The `ManagedThreadFactory` instance can create threads with different attributes (such as priority).
- **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.
    - **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.

3.1.4.2.1 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 of 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
Hung Task Threshold	60000 ms
Pool Info:	Core Size: 5 Max Size: 25 Keep Alive: 5000 ms Work Queue: 15 Capacity:
Reject Policy	☑ Abort ☐ Retry and Abort

Table 1: Typical Thread Pool Configuration Example

3.1.4.2.2 Thread Pool for Long-Running Tasks

This executor describes a configuration in which the executor is used to run a few long-running tasks in the local process. In this example the task can run up to 24 hours before it is considered hung.

ManagedExecutorService	
Name:	Long-Running Tasks Thread Pool
JNDI Name:	concurrent/execsvc/LongRunning
Context:	concurrent/ctx/AllContexts
Thread Factory:	concurrent/tf/longRunningThreadsFactory
Hung Task Threshold	24 hours
Pool Info:	Core Size: 0 Max Size: 5 Keep Alive: 1000 ms Work Queue: 5 Capacity:
Reject Policy	☑ Abort

	☐ Retry and Abort

Table 2: Long-Running Tasks Thread Pool Configuration Example

3.1.4.2.3 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.

ManagedExecutorService	
Name:	Shared OLTP Thread Pool
JNDI Name:	concurrent/execsvc/OLTPShared
Context:	concurrent/ctx/OLTPContexts
Thread Factory:	concurrent/tf/oltp
Hung Task Threshold	20000 ms
Pool Info:	Core Size: 100 Max Size: 250 Keep Alive: 10000 ms Work Queue: 100 Capacity:
Reject Policy	☑ Abort ☐ Retry and Abort

Table 3: OLTP Thread Pool Configuration Example

3.1.4.3 Default ManagedExecutorService

The Java EE Product Provider must provide a preconfigured, default `ManagedExecutorService` for use by application components under the JNDI name `java:comp/DefaultManagedExecutorService`. The types of contexts to be propagated by this default `ManagedExecutorService` must include naming context, classloader, and security information.

3.1.5 System Administrator's Responsibilities

The System Administrator (EE.2.11.5) 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)

3.1.6 Lifecycle

The lifecycle of `ManagedExecutorService` instances are centrally managed by the application server and cannot be changed by an application.

A `ManagedExecutorService` instance 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 3-1, a single `ManagedExecutorService` instance 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 task would be run in the context of the submitting component (as noted by different colored boxes in the figure).

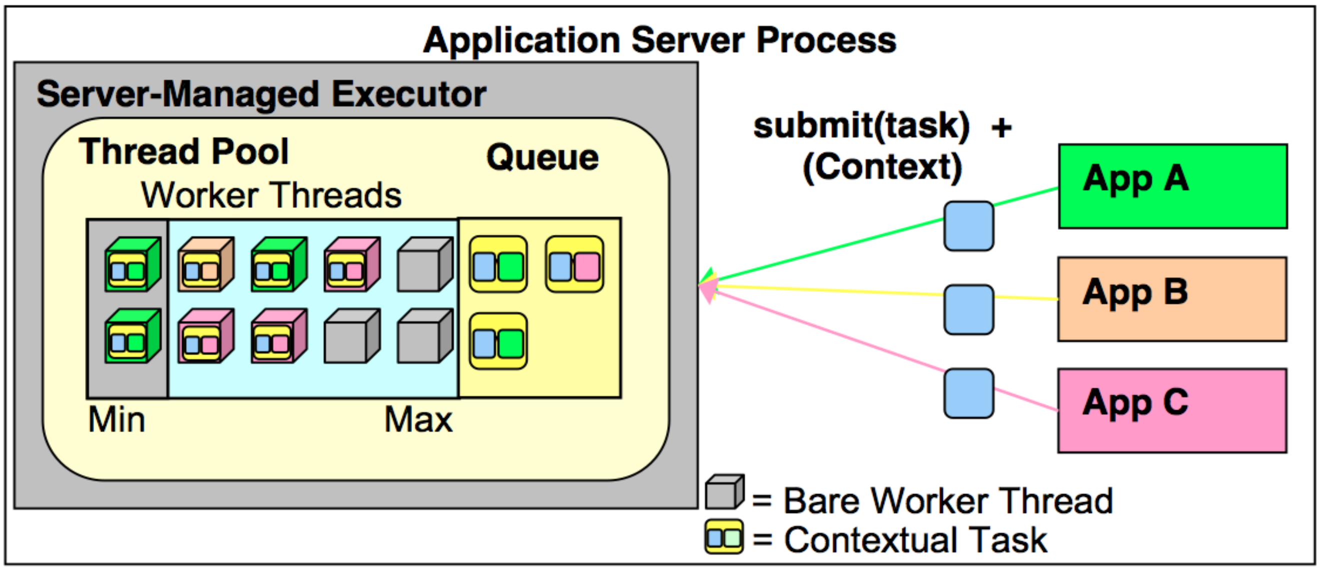

Figure 3-1 Managed Thread Pool Executor Component Relationship

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

3.1.6.1 Java EE Product Provider Requirements

This subsection describes additional requirements for `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. This includes the following methods that are defined in the `java.util.concurrent.ExecutorService` interface: `awaitTermination()`, `isShutdown()`, `isTerminated()`, `shutdown()`, and `shutdownNow()`.
3. No task submitted to an executor can run if task's component is not started.

When a `ManagedExecutorService` instance is being shutdown by the Java EE Product Provider:

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 transaction demarcation using the `javax.transaction.UserTransaction` interface, which is described in the Java Transaction API 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 EE.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` (EE.5.10 and EE.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.
4. A task instance that starts a transaction using the `javax.transaction.UserTransaction` instance but fails to complete the transaction is an error. The Java EE Product Provider is required to detect this error and abort the transaction.
5. A task must have the same ability to use transactions as the component submitting the tasks. For example, tasks are allowed to call transactional enterprise beans, and managed beans that use the `@Transactional` interceptor as defined in the Java Transaction API specification.

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 {

   UserTransaction ut = …;

   public void run() {

        // 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.enterprise.concurrent.ManagedScheduledExecutorService is an interface that extends the java.util.concurrent.ScheduledExecutorService and javax.enterprise.concurrent.ManagedExecutorService 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 `Trigger` and `ManagedTaskListener`.

3.2.1 Application Component Provider's Responsibilities

Application Component Providers (application developers) (EE2.11.2) 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 (EE.5.2) or through injection of resource environment references (EE.5.8.1.1).

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.enterprise.concurrent.ManagedScheduledExecutorService`. (See EE.5.8.1.2 for information on how resource environment references are declared in the deployment descriptor.)
- Look up the managed object in the application component's environment using JNDI (EE.5.2), or through resource injection by the use of the `Resource` annotation (EE.5.8.1.1).

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.

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()`, `invokeAll()`, `invokeAny()`, `schedule()`, `scheduleAtFixedRate()` or `scheduleWithFixedDelay()` methods. Task instances will run as an extension of the Java EE container instance that submitted the task and may interact with Java EE resources as defined in other sections of this specification.

Task classes can optionally implement the `javax.enterprise.concurrent.ManagedTask` interface to provide identity information, execution properties such as whether the task can be run in a remote process, and to register a `javax.enterprise.concurrent.ManagedTaskListener` instance to receive lifecycle events notifications.

It is important for Application Component Providers to identify and document the required behaviors and service-level agreements for each required `ManagedScheduledExecutorService`. 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 lifecycle 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`.

3.2.1.1.2 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 3.2.4.1). Alternatively, the `Resource` annotation can be used in the Servlet code.

***Note** – Using the description for documenting the configuration attributes of the managed 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:
      Context Info: Local Namespace
   </description>
   <resource-env-ref-name>
      concurrent/ScheduledLoggerExecutor
   </resource-env-ref-name>
   <resource-env-ref-type>
      javax.enterprise.concurrent.ManagedScheduledExecutorService
   </resource-env-ref-type>
</resource-env-ref>
```

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

   void logEvents(Collection data, DataSource ds) {

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

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

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

3.2.1.1.4 Task Submission

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

```
public class CtxListener implements ServletContextListener {
   Future loggerHandle = null;

   @Resource(name="concurrent/ScheduledLoggerExecutor")
   ManagedScheduledExecutorService mes;

   public void contextInitialized(ServletContextEvent scEvent) {



      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 (EE.2.11.3) is responsible for assembling the application components into a complete Java EE Application and providing assembly instructions that describe the dependencies to the managed objects.

3.2.3 Deployer's Responsibilities

The Deployer (EE.2.11.4) 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 managed objects with the properly defined attributes. See EE.5.8.2 for details.

3.2.4 Java EE Product Provider's Responsibilities

The Java EE Product Provider's responsibilities are as defined in EE.5.8.3.

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

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

3.2.4.1 ManagedScheduledExecutorService Configuration Attributes

Each `ManagedScheduledExecutorService` may support one or more runtime behaviors as specified by configuration attributes. The Java EE Product Provider will determine both the appropriate attributes and the means of configuring those attributes for their product.

3.2.4.2 Configuration Examples

This section and subsections illustrate some examples of how a Java EE Product Provider could configure a `ManagedScheduledExecutorService` and the possible options that such a 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. If both Context and ThreadFactory attributes are specified, the Context attribute of the ThreadFactory configuration should be ignored.
- **ThreadFactory**: A reference to a `ManagedThreadFactory` instance (see section 3.4). The managed ThreadFactory instance can create threads with different attributes (such as priority).
- **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.

3.2.4.2.1 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/ctx/AllContexts
Thread Factory:	concurrent/tf/normal
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

Table 4: Typical Timer Configuration Example

3.2.4.3 Default ManagedScheduledExecutorService

The Java EE Product Provider must provide a preconfigured, default `ManagedScheduledExecutorService` for use by application components under the JNDI name `java:comp/DefaultManagedScheduledExecutorService`. The types of contexts to be propagated by this default `ManagedScheduledExecutorService` must include naming context, class loader, and security information.

3.2.5 System Administrator’s Responsibilities

The System Administrator (EE.2.110.5) 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).

3.2.6 Lifecycle

The lifecycle of `ManagedScheduledExecutorService` instances are centrally managed by the application server and cannot be changed by an application.

A `ManagedScheduledExecutorService` instance can 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 3-1 Managed Thread Pool Executor Component Relationship.

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

3.2.6.1 Java EE Product Provider Requirements

This subsection describes requirements for `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. This includes the following methods that are defined in the `java.util.concurrent.ExecutorService` interface: `awaitTermination(), isShutdown(), isTerminated(), shutdown(),` and `shutdownNow()`.
3. All tasks submitted to an executor must not run if task's component is not started.

When a `ManagedScheduledExecutorService` instance is being shutdown by the Java EE Product Provider:

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 transaction demarcation using the `javax.transaction.UserTransaction` interface, which is described in the Java Transaction API 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 EE.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.
4. A task instance that starts a transaction using the `javax.transaction.UserTransaction` instance but fails to complete the transaction is an error. The Java EE Product Provider is required to detect this error and abort the transaction.
5. A task must have the same ability to use transactions as the component submitting the tasks. For example, tasks are allowed to call transactional enterprise beans, and managed beans that use the `@Transactional` interceptor as defined in the Java Transaction API specification.

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 `java.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.enterprise.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.).

3.3.1 Application Component Provider's Responsibilities

Application Component Providers (application developers) (EE2.11.2) 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 (EE.5) or through injection of resource environment references (EE.5.8.1.1).

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.enterprise.concurrent.ContextService`. (See EE.5.8.1.2 for information on how resource environment references are declared in the deployment descriptor.)
- Look up the managed object in the application component's environment using JNDI (EE.5.2), or through resource injection by the use of the `Resource` annotation (EE.5.8.1.1).

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.

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

It is important for Application Component Providers to identify and document the required behaviors and service-level agreements for each required `ContextService`. 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 `ExecutorService` 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 Connectors resource adapter. In this example, we use a singleton session bean.

Since the `ExecutorService` is a singleton session bean, it can be accessed by several EJB or Servlet instances. The `ExecutorService` uses threads created from a `ManagedThreadFactory` (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.

3.3.1.1.2 ExecutorService Singleton

Create a singleton session bean `ExecutorAccessor` with a getter for the `ExecutorService`. The `ExecutorAccessor` should be included with the EJB module or other jar that is in the scope of the application component.

```
@Singleton
public class ExecutorAccessor {

    private ExecutorService threadPoolExecutor = null;


    @Resource(name="concurrent/ThreadFactory")
    ManagedThreadFactory threadFactory;

    public ExecutorService getThreadPool() {
      if (threadPoolExecutor == null) {
            threadPoolExecutor = new ThreadPoolExecutor(
               5, 10, 5, TimeUnit.SECONDS,
               new ArrayBlockingQueue<Runnable>(10), threadFactory);
      }
      return threadPoolExecutor;
    }
}
```

3.3.1.1.3 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.

3.3.1.1.4 Resource Environment References

This example refers to a `ContextService` and a `ManagedThreadFactory`.

Note *– Using the description for documenting the configuration attributes of the managed 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.
          Priority: Normal
          Context Info: NA
   </description>
   <resource-env-ref-name>
      concurrent/ThreadFactory
   </resource-env-ref-name>
   <resource-env-ref-type>
```

```
      javax.enterprise.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.enterprise.concurrent.ContextService
   </resource-env-ref-type>
</resource-env-ref>
```

3.3.1.1.5 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);
   }

   ...
}
```

3.3.1.1.6 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 {
```

```
    @Resource(name="concurrent/AllContexts")
    ContextService ctxSvc;

    @EJB private ExecutorAccessor executorAccessor;

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

       // 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 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);
       Callable<Long> agency2 = ctxSvc.createContextualProxy(
             new CreditScoreTask(taxID, 2), Callable.class);
       Callable<Long> agency3 = ctxSvc.createContextualProxy (
             new CreditScoreTask(taxID, 3), Callable.class);

       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 (EE.2.11.3) is responsible for assembling the application components into a complete Java EE Application and providing assembly instructions that describe the dependencies to the managed objects.

3.3.3 Deployer’s Responsibilities

The Deployer (EE.2.11.4) 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 managed objects with the properly defined attributes. See EE.5.8.2 for details.

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

3.3.4 Java EE Product Provider's Responsibilities

The Java EE Product Provider's responsibilities are as defined in EE.5.8.3 and must provide an implementation of any behaviors defined in 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.

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

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

3.3.4.1 ContextService Configuration Attributes

Each `ContextService` may support one or more runtime behaviors as specified by configuration attributes. The Java EE Product Provider will determine both the appropriate attributes and the means of configuring those attributes for their product.

3.3.4.2 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 a 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 3.1.4.2 3.2.4.2 and 3.4.4.2 refer to one of the `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.
- **Context info**: The context information to be propagated.
    - **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.

3.3.4.2.1 All Contexts

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

Table 5: All Contexts Configuration Example

3.3.4.2.2 OLTP Contexts

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

Table 6: OLTP Contexts Configuration Example

3.3.4.2.3 No Contexts

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

Table 7: No Contexts Configuration Example

3.3.4.3 Default ContextService

The Java EE Product Provider must provide a preconfigured, default `ContextService` for use by application components under the JNDI name `java:comp/DefaultContextService`. The types of contexts to be propagated by this default `ContextService` must include naming context, class loader, and security information.

3.3.5 Transaction Management

Contextual dynamic proxies support user-managed global transaction demarcation using the `javax.transaction.UserTransaction` interface, which is described in the Java Transaction API 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 rollback a transaction. See EE.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.enterprise.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` (EE.5.10 and EE.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.
4. A task instance that starts a transaction using the `javax.transaction.UserTransaction` instance but fails to complete the transaction is an error. The Java EE Product Provider is required to detect this error and abort the transaction.
5. A task must have the same ability to use transactions as the component submitting the tasks. For example, tasks are allowed to call transactional enterprise beans, and managed beans that use the `@Transactional` interceptor as defined in the Java Transaction API specification.

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 of using a UserTransaction within a task.

3.4 ManagedThreadFactory

The `javax.enterprise.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) (EE2.11.2) use a `javax.enterprise.concurrent.ManagedThreadFactory` instance to create manageable threads. `ManagedThreadFactory` instances are retrieved using the Java Naming and Directory Interface (JNDI) Naming Context (EE.5) or through injection of resource environment references (EE.5.8.1.1).

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.enterprise.concurrent.ManagedThreadFactory`. (See EE.5.8.1.2 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 managed object in the application component's environment using JNDI (EE.5), or through resource injection by the use of the `Resource` annotation (EE.5.8.1.1).
- 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 (see section 2.3). 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 a `ManagedThreadFactory` instance is stopped, all subsequent calls to `newThread()` must throw a `java.lang.IllegalStateException`

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 lifecycle 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.`

3.4.1.1.2 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 section 3.4.4.2). Alternatively, the `Resource` annotation can be used in the Servlet code.

***Note** – Using the description for documenting the configuration attributes of the managed 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.enterprise.concurrent.ManagedThreadFactory
   </resource-env-ref-type>
 </resource-env-ref>
```

3.4.1.1.3 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 {
```

```
   DatsSource ds = ...;

   public void run() {



      // 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 (...) {
         try (Connection con = ds.getConnection();... {

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

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

3.4.1.1.4 Task Submission

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

```
public class CtxListener implments ServletContextListener {
   Thread loggerThread = null;

   @Resource(name="concurrent/LoggerThreadFactory")
   ManagedThreadFactory threadFactory;

   public void contextInitialized(ServletContextEvent scEvent) {

       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 (EE.2.11.3) is responsible for assembling the application components into a complete Java EE Application and providing assembly instructions that describe the dependencies to the managed objects.

3.4.3 Deployer's Responsibilities

The Deployer (EE.2.11.4) 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 managed objects with the properly defined attributes. See EE.5.8.2 for details.

3.4.4 Java EE Product Provider's Responsibilities

The Java EE Product Provider's responsibilities are as defined in EE.5.8.3 and must support the following:

- Threads returned by the `newThread()` method must implement the `ManageableThread` interface.
- When a `ManagedThreadFactory` instance is stopped, such as when the component that created it is stopped or when the application server is shutting down, all threads that it has created using the `newThread()` method are interrupted. Calls to the `isShutdown()` method in the `ManageableThread` interface on these threads must return true.

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

- Threads that are created by a `ManagedThreadFactory` instance but are started after the `ManagedThreadFactory` has shut down is required to start with an interrupted status. Calls to the `isShutdown()` method in the `ManageableThread` interface on these threads must return true.

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

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

3.4.4.1 ManagedThreadFactory Configuration Attributes

Each managed `ManagedThreadFactory` may support one or more runtime behaviors as specified by configuration attributes. The Java EE Product Provider will determine both the appropriate attributes and the means of configuring those attributes for their product.

3.4.4.2 Configuration Examples

This section and subsections illustrate some examples of how a Java EE Product Provider could configure a `ManagedThreadFactory` and the possible options that such a 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 3.1.4.2 and 3.2.4.2 refer to one of the `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.

3.4.4.2.1 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 8: Normal ManagedThreadFactory Configuration Example

3.4.4.2.2 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 9: OLTP ManagedThreadFactory Configuration Example

3.4.4.2.3 Threads for Long-Running Tasks

This configuration example describes a `ManagedThreadFactory` that creates lower-priority threads that can be used for background, long-running tasks.

ManagedThreadFactory	
Name:	Long Running Tasks Threads
JNDI Name:	Concurrent/tf/longRunningThreads Factory
Context:	Concurrent/cf/AllContexts
Priority:	4

Table 10: Long-Running Tasks ManagedThreadFactory Configuration Example

3.4.4.3 Default ManagedThreadFactory

The Java EE Product Provider must provide a preconfigured, default `ManagedThreadFactory` for use by application components under the JNDI name `java:comp/DefaultManagedThreadFactory`. The types of contexts to be propagated by this default `ManagedThreadFactory` must include naming context, class loader, and security information.

3.4.5 System Administrator's Responsibilities

The System Administrator (EE.2.11.5) 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).

3.4.6 Transaction Management

`ManagedThreadFactory` implementations must support user-managed global transaction demarcation using the `javax.transaction.UserTransaction` interface, which is described in the Java Transaction API specification s 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 EE.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.

1. The `javax.transaction.UserTransaction` interface must be made available in the local JNDI namespace as environment entry: `java:comp/UserTransaction` (EE.5.10 and EE.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.
4. A task instance that starts a transaction using the `javax.transaction.UserTransaction` instance but fails to complete the transaction is an error. The Java EE Product Provider is required to detect this error and abort the transaction.
5. A task must have the same ability to use transactions as the component submitting the tasks. For example, tasks are allowed to call transactional enterprise beans, and managed beans that use the `@Transactional` interceptor as defined in the Java Transaction API specification.

3.4.6.2 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 of using a UserTransaction within a task.

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 Changes Since Public Review Draft

- Added section 2.3.2.1 “Tasks and Contexts and Dependency Injection (CDI)”.
- Renamed “Administered Objects” to “Managed Objects”.
- Added requirements to sections 3.1.8.1, 3.2.8.1, 3.3.5.1, and 3.4.6.1 Transaction Management for Java EE Product Provider to rollback incomplete UserTransaction and to provide same transaction ability as submitting application components.
- Removed Contextual-Callback configuration attribute
- Removed section 3.5 – Distributable ManagedExecutorService
- Removed “Run Location” configuration attribute
- Removed section 4 – Managed Objects
- Clarified callback methods are invoked with unspecified context in section 2.3.1.1.
- Updated example in section 3.1.1.1 to remove EJB 2.x home reference
- Renamed BatchExecutor in examples to avoid confusion with Batch JSR.
- Minor fixes to example codes

A.2 Public Review Draft (December 27, 2012)

- Removed “Preserve the fundamental capabilities and integrity of the Java EE platform” from list of goals as this is redundant
- Clarified types of context to be propagated in section 2.3.
- Updated section 2.3.1.1 to use the new CONTEXTUAL_CALLBACK_HINT execution property for requesting contextual callback, instead of using Contextual-Callback configuration attribute which is not a required configuration attribute.
- Clarified how Connectors gain access to JNDI context in section 2.4.
- Changes related to replacement of Identifiable interface with ManagedTask interface. Added paragraphs on new ManagedTask interface in sections 3.1.1 and 3.2.1, updated example in section 3.1.1.1 to use ManagedTask interface, and updated means of getting values for taskIdentityName and taskIdentityDescription in sections 4.4.1 and 4.4.2.
- Remove sentences on configuration attributes requirement and requirement that support for optional behavior must pass the TCK from sections 3.1.3.1, 3.2.3.1, 3.3.3.1 and 3.4.3.1. Configuration attributes listed are not required and optional behaviors are not tested by TCK.
- Added requirement to Java EE Product Provider to provide default ManagedExecutorService, ManagedScheduledExecutorService, ContextService, and ManagedThreadFactory.
- Added requirement to section 3.5.1 that only tasks with value of execution property DISTRIBUTABLE_HINT of true can be run remotely.

- Renamed createContextObject() in ContextService to createContextualProxy().

A.3 Early Draft (November 2, 2012)

- Fixed usages of dependency injections in code examples throughout the document.
- Clarified "optional" in optional contextual invocation points in section 2.3.1.1.
- Added section for open issues.
- Reduced levels of sub-sections in various Usage Example sections.

A.4 Early Draft Candidate (August 6, 2012)

- Updated section 2.3.1 to not require callback methods as contextual invocation points, and added new configuration parameter Contextual-Callback in sections 3.1.4.1 and 3.2.4.1.
- In sections 3.1.4.1 and 3.2.4.1, clarified behavior when both Context and ThreadFactory configuration attributes are specified.
- Removed component-managed ManagedExecutorService and ManagedScheduledExecutorService.
- Moved applicable shutdown behavior of component-managed ManagedExecutorService and ManagedScheduledExecutorService to sections 3.1.6.1 and 3.2.6.1.
- Updated example in section 3.3.1.2.3 to reflect API change in ContextService.
- Added more details to ManagedThreadFactory shutdown mechanism in section 3.4.4.
- Updated section 3.5 to make Distributable ManagedExecutorService optional.
- Updated section 3.5.1 to require Trigger to be Serializable.
- Removed requirements in section 3.5.1 that defines behavior when master or slave executor becomes unavailable.
- Managed Object in section 4 is made optional.
- Fixed more typos.
- Updated more code examples to use resource injection instead of JNDI lookup.

A.5 Early Draft Candidate (June, 2012)

- Updated Copyrights and Licenses
- Updated package name from `javax.util.concurrent` to `javax.enterprise.concurrent`
- Fixed typos
- Updated references to Java EE specifications
- Added references to resource injection in sections 3.1.1, 3.2.1, 3.3.1, and 3.4.1 and updated code examples
- Added Thread Use configuration attribute to `ManagedExecutorService` configuration example in section 3.1.4.1

A.6 Version 0.1: Early Draft Preview (Apr 4, 2006)