JSR 236: Concurrency Utilities for Java EE

Anthony Lai

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Agenda

- Introduction
- Overview
- Technical Details

Introduction

Brief History

- BEA-IBM CommonJ API for Java EE
- 2003 - JSR 236-237 provides context aware Thread Pools and Timers to Java EE applications
- 2006 – Extending existing Java SE concurrency foundations, replacing CommonJ
- 2008 - Combined into JSR 236. JSR 237 withdrawn.
- April 2012 - Restarted under JCP 2.8
- Nov 2012 – EDR under JCP 2.9
- Jan 2013 – Public Review started

Introduction

Expert Group

Corporate Members
IBM
Oracle
RedHat

Individual Members
Adam Bien
Cyril Bouteille
Andrew Evers
Doug Lea

Introduction
JSR Transparency

- Mailing lists:
  - users@concurrency-ee-spec.java.net
  - jsr236-experts@concurrency-ee-spec.java.net
  - Both lists are archived and are publicly readable
  - Users list available to public to subscribe. Includes all discussions in experts list
- Issue Tracker:
  - http://java.net/jira/browse/CONCURRENCY_EE_SPEC
  - Everyone can track existing issues and file new issues

Introduction
JSR Transparency

- Spec work
  - Open source project
  - concurrency-ee-spec java.net
  - Latest browseable javadoc
  - Updated spec document drafts
- Reference Implementation
  - Open source project
  - cu-javaee java.net project is used for reference implementation work

Introduction
Schedule

- Align with Java EE 7 schedule
  - Early Draft: Nov 15 – Dec 15, 2012
  - Public Review: Jan 3 – Feb 4, 2013
  - Proposed Final Draft: Mar 2013
  - Final Release: Apr 2013

Overview

Limitation of concurrency in Java EE

- Java SE threads and timers are not well integrated with Java EE containers
  - Threads not controlled by Java EE containers
  - Thread context like class loader, security, naming are not propagated
  - Lack of manageability and transaction isolation semantics
- Asynchronous support is available in Java EE, such as in servlet and EJB, but
  - Advanced features can be provided by java.util.concurrent APIs such as invokeAll, invokeAny, or custom thread pool with managed threads provided by Java EE product provider
  - Needs mechanism to provide managed threads for async servlets

Overview

Concurrency uses in Java EE

- Decouple user execution from slow moving background processing
- Improvements in processor architecture promote parallelism
- One big task into smaller concurrent tasks
- Asynchronous notification use case
- Timer use cases like periodic cleanup, cache maintenance

Overview

Special Java EE requirements

- Managed threads
  - Coordination between application server lifecycle and asynchronous task lifecycle
    - Server shutdown
    - Application deployment/undeployment
  - Administration and monitoring
- Intelligent workload classification and routing
  - Batch vs interactive
  - Local vs distributed
- Application Integrity
  - Different context for different applications
  - Applications to coexist

Overview

Goals

- Provide consistent programming model
- Leverage existing technology to provide migration from Java SE
- Allow adding concurrency to existing applications
- Provide simple API for simple use cases
- Provide flexible API for advanced use cases

Overview
Extending Java SE

- Extend existing Java SE concurrency utilities by providing managed versions:
  - ManagedThreadFactory
  - ManagedExecutorService
  - ManagedScheduledExecutorService
- Add Java EE extensions
  - ContextService
  - ManagedTaskListener
  - Trigger
  - ManagedTask

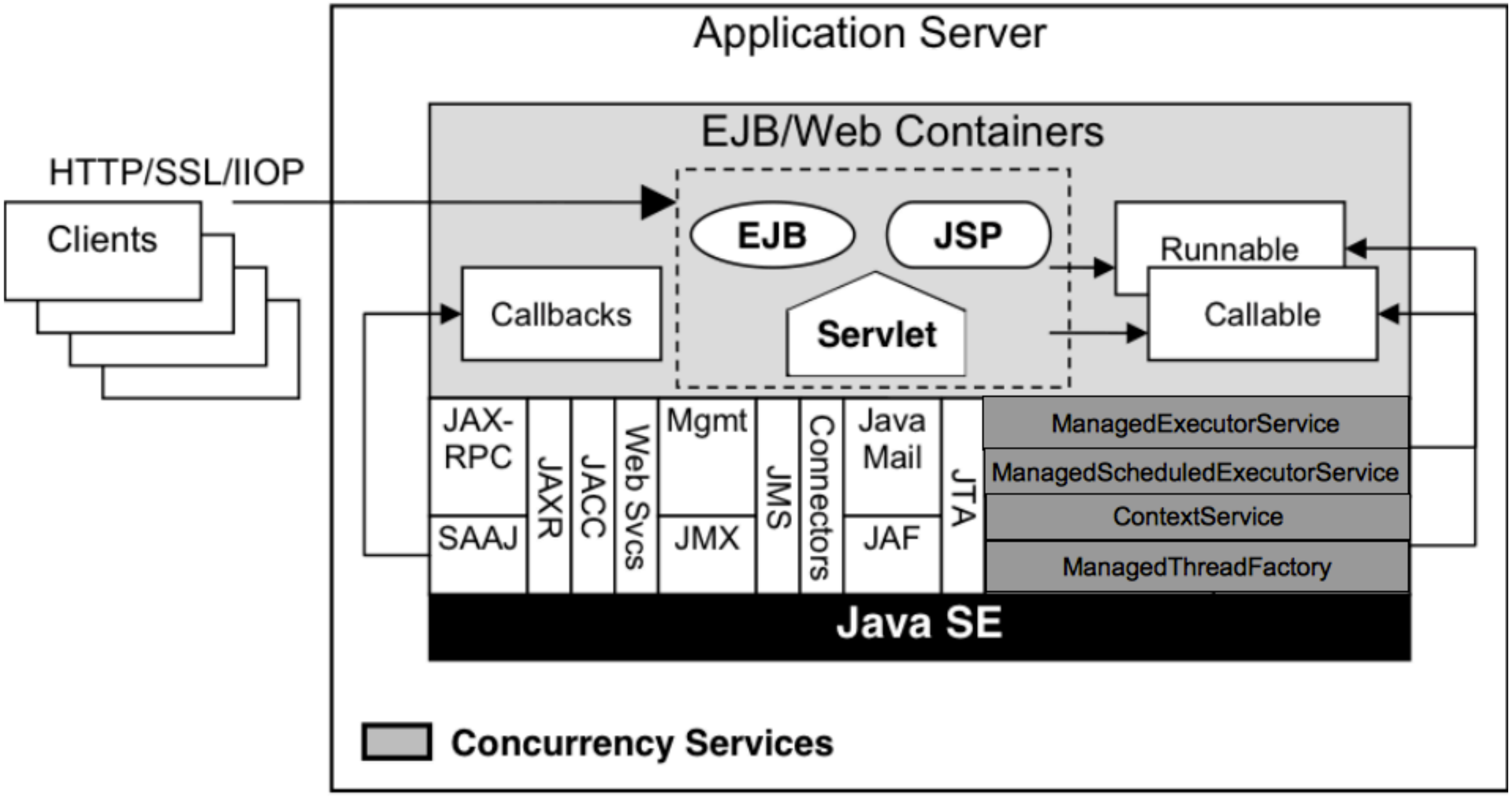
Overview

Extending Java SE

- Provide manageability using JMX Mbeans
  - ManagedThread
  - ManagedThreadFactory
  - ManagedExecutorService
  - Mbeans support are optional

Overview

Java EE Architecture Diagram with Concurrency



ManagedThreadFactory

Overview

- Standard interface and method for creating threads
  - `Thread newThread(Runnable r)`
- Centrally defined on an application server
- Java EE product providers provide the thread
- Referenced by applications through JNDI lookup or @Resource annotation
- Default pre-configured ManagedThreadFactory
- Extension of Java SE ThreadFactory
  - Adds container context and manageability
  - UserTransaction support (does not enlist in parent component's transaction)

ManagedThreadFactory
Usage Scenarios

- Long Running Tasks
  - Work Consumers/Producers
  - Batch jobs
  - Embedded servers
- Custom Thread Pools
  - Use Java SE thread pools
  - Any service that can use ThreadFactory

ManagedThreadFactory
Code Sample - Daemon

```
// Within your servlet or EJB method...
// Lookup the ManagedThreadFactory
InitialContext ctx = new InitialContext();
ManagedThreadFactory tf = (ManagedThreadFactory)
    ctx.lookup("java:comp/env/concurrent/myTF");

// Create and start the thread.
Thread daemonThread = tf.newThread(myDaemonRunnable);
daemonThread.start();
// The runnable behaves as if it were running in the
// servlet or EJB container.
// The thread's lifecycle is tied to the application
// and is interrupted when application stops.
```

ManagedThreadFactory

Code Sample – Custom Thread Pool

```
// Within your servlet or EJB method...
// Lookup the ManagedThreadFactory
@Resource
ManagedThreadFactory tf;

void businessMethod() {
// Use a custom Java SE ThreadPoolExecutor
CustomThreadPoolExecutor pool =
new CustomThreadPoolExecutor(coreSize, maxSize, tf);

// When the executor allocates a new thread, the
// thread will use the current container context.
```

ManagedThreadFactory

Thread Management with JMX

- Monitor when threads are allocated using the ManagedThreadFactory MBean
- Monitor thread activity and health
  - What task is running on the thread?
  - How long has the task been running?
  - Correlate to the Java SE thread name and id
- Cancel a thread (cooperative)
  - Hung threshold notifications help identify problems
  - Proper interruption detection is essential in the task implementation.

ManagedThreadFactory
Task Identity

- Runnable and Callable that are run on a managed thread may optionally implement the ManagedTask interface.
- Allows runtime introspection of thread's current state.
- Exposed on the ManagedThread MBean
- Short name available as an attribute
- Locale-specific description available as an attribute for the current locale or an operation for alternative locales.

ManagedThreadFactory

Code Sample – Supplying identity to task

```
class MyConsumerTask implements Runnable, ManagedTask {
  private Map<String, String> props;
  public void run() {
    // Update the identity name periodically
    props.put(IDENTITY_NAME,
"MonitorApp:MyConsumerTask:Phase1";
    ...
    props.put(IDENITY_NAME,
"MonitorApp:MyConsumerTask:Phase2";
  }

  public Map<String, String> getExecutionProperties()  {
    // Called by ManagedThread.taskIdentityName
    return props;
  }
  public String getIdentityDescription(Locale l) {
    // Called by ManagedThread.taskIdentityDescription
    // Get description from NLS bundle
  }
}
```

ManagedExecutorService
Overview

- Typical way of running tasks asynchronously from a Java EE container method
- Centrally defined on an application server
- Java EE product providers provide the implementation
- Referenced by applications through JNDI lookup or @Resource annotation
- Default pre-configured ManagedExecutorService
- Typically used for centralized thread pooling
- Implementations may offer extended capabilities

ManagedExecutorService

Overview continued

- Extension of Java SE ExecutorService
  - No extension APIs. Same execute, submit, invokeAll, invokeAny APIs as in Java SE parent classes
  - Adds container context, manageability, task lifecycle tracking and application component lifecycle constraints
  - UserTransaction support (does not enlist in parent component transaction)
  - Distributed (remote) capability - optional

ManagedExecutorService

Overview continued

- Server-managed
  - Multiple applications share a single executor
  - Application developer defines the requirements of the executor: What container contexts to propagate (e.g. namespace)
  - Lifecycle managed by server. Lifecycle APIs such as shutdown not allowed
- Deployer configures the appropriate executor and maps the resource environment reference to the executor

ManagedExecutorService

Management

- Hung tasks can be monitored and cancelled using JMX.
  - Threads are created from a ManagedThreadFactory
  - Each thread therefore is associated with a ManagedThread MBean
  - Tasks can provide identifying info
- Task lifecycle can be monitored using ManagedTaskListeners
  - Monitoring extensions (logging)
  - Work-flow control and management

ManagedExecutorService

ManageTaskListener

- Listeners are Java objects that are registered with the task when submitted to the executor.
- The listener method can be configured to run in the same container context as the task.
  - taskSubmitted – The task was submitted to the executor
  - taskAborted – The task was unable to start or was cancelled.
  - taskStarting – The task is about to start
  - taskDone – The task has completed (successfully, exception, cancelled, aborted, or rejected)

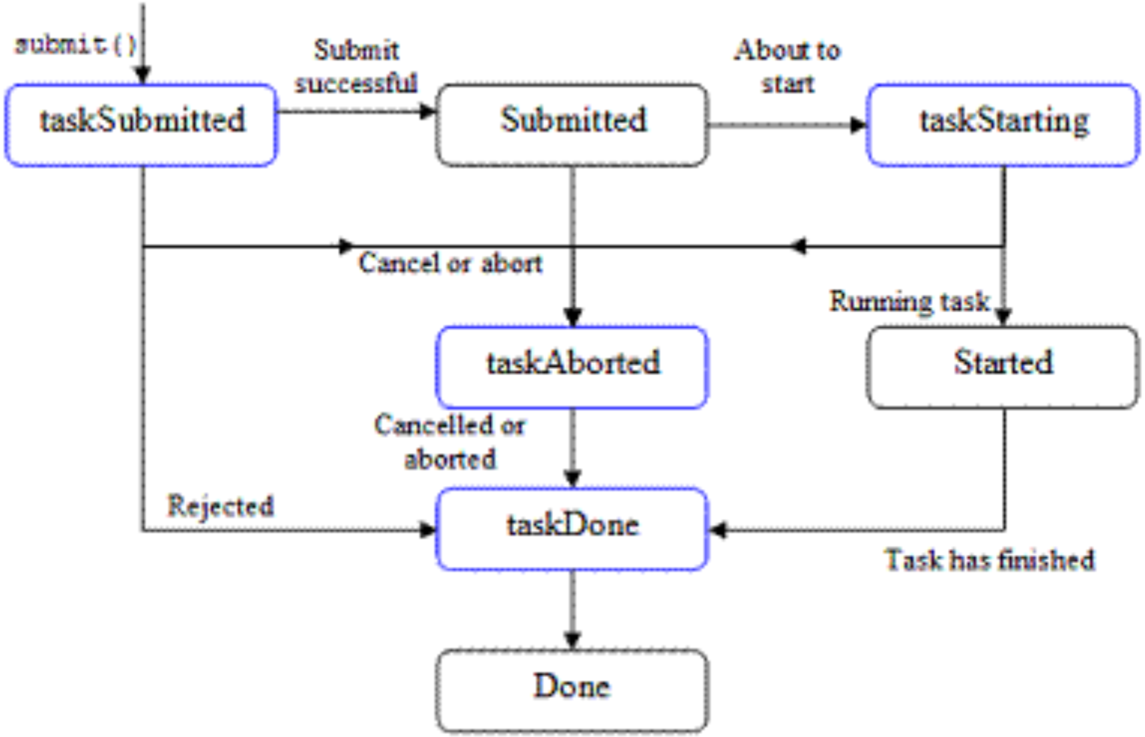
ManagedExecutorService

Code Sample – Registering ManagedTaskListener

```
// Runnable implements ManagedTask
Public class TaskWithListener implements Runnable,
ManagedTask {
  ...
  public ManagedTaskListener getManagedTaskListener {
    return aManagedTaskListener;
  }
}
// Or use ManagedExecutors utility method to associate
// a ManagedTaskListener to a task
Runnable aTask;
ManagedTaskListener myTaskListner;
Runnable taskWithListener =
  ManagedExecutors.managedTask(aTask, myTaskListener);
// submit taskWithListener to a ManagedExecutorService
```

ManagedExecutorService

ManageTaskListener - Lifecycle



ManagedExecutorService

Code Sample – Typical Parallelism

```
// Within your [async] servlet or [async] EJB method
@Resource(name=“concurrent/myExecutor”)
ManagedExecutorService mes;
void businessMethod() {
  Callable<Integer> c = new Callable<>() {
    Integer call() {
    // Interact with a database... Return answer.
    // The namespace is available here!
  }
}
// Submit the task and do something else. The task
// will run asynchronously on another thread.
Future result = mes.submit(c);
...
// Get the result when ready...
int theValue = result.get();
...
```

ManagedExecutorService
Distributable

- Same rules as a ManagedExecutorService
- Allows distributing the task to a peer on another server instance (JVM).
  - Task must implement serializable
- Optional feature – Java EE Providers do not have to supply a distributable ManagedExecutorService.
- Two distributable types are available:
  - With and without affinity
- Tasks could provide distributable hint to Java EE product providers through executionProperties in ManagedTask.

ManagedScheduledExecutorService
Overview

- Typical way of running periodic tasks asynchronously from a Java EE container method
- Typically used for transient timers
- Inherits semantics of ManagedExecutorService:
  - Centrally defined on an application server
  - Java EE product providers provide the implementation
  - Referenced by applications through JNDI lookup or @Resource annotation
  - Default pre-configured ManagedScheduledExecutorService
  - Implementations may offer extended capabilities

ManagedScheduledExecutorService

Overview continued

- Extension of ScheduledExecutorService
  - Adds container context, manageability, task lifecycle tracking and application component lifecycle constraints
  - UserTransaction support (does not enlist in parent component transaction)
  - Trigger mechanism
- Server-managed
  - Multiple applications share a single executor
  - Application developer defines the requirements of the executor: What container contexts to propagate (e.g. namespace)
  - Lifecycle managed by server. Lifecycle APIs such as shutdown not allowed

ManagedScheduledExecutorService

Usage Scenarios

- Periodic cache invalidations
- Request timeouts
- Polling
- Custom Scheduler
  - Would need implementation extension to support persistence.
  - Use Triggers for custom calendaring:
    - N-time fixed-rate with time-sensitive skip.
    - Run time based on previous task calculation result.
    - Condition-based trigger
    - Centralized business calendar

ManagedScheduledExecutorService

API

- execute, submit, invokeAny, invokeAll, schedule, scheduleAtFixedRate, scheduleWithFixedDelay from Java SE parent classes
- Extension APIs for custom trigger schedule support
  - `ScheduledFuture<?> schedule(Runnable command, Trigger trigger)`
  - `<V> ScheduledFuture<V> schedule(Callable<V> callable, Trigger trigger)`

ManagedScheduledExecutorService

Trigger

```
interface Trigger {
  // Return true if you want to skip the
  // currently-scheduled execution.
  boolean skipRun(LastExecution
lastExecutionInfo, Date scheduledRunTime);

  // Retrieves the time in which to run the task
  // next. Invoked during submit time and after
  // each task has completed.
  Date getNextRunTime(LastExecution
lastExecutionInfo, Date taskScheduledTime);
}
```

ContextService

Overview

- Mechanism for applications to capture container context and run within that context later, even on another server or after server restart
  - Security, naming, classloader
  - Additional types of context could be supported by Java EE product providers
  - ManagedExecutorService likely to use this service internally to propagate container context.
- Centrally defined on an application server
- Referenced by applications through JNDI lookup or @Resource annotation
- Default pre-configured ContextService

ContextService

Overview continued

- Java EE product providers provide the implementation
  - Implementations may offer extended capabilities
- Current thread context is captured and stored within a context proxy for your object
- Customizable through executionProperties
  - Can enable transaction pass-through
- Used in advanced scenarios
- Use with non-ManagedThreadFactory-created threads (threads created with new Thread())

ContextService

Use Cases

- Workflow
  - Store and propagate user identity
- Java SE or third-party thread reuse
  - Allows thread to behave as-if it were on a container thread.

ContextService

API

- For creating new contextual object proxy for the input object instance
  - `Object createContextualProxy(Object instance, Class<?>... Interfaces)`
  - `Object createContextualProxy(Object instance, Map<String,String> executionProperties, Class<?>... Interfaces)`
  - `<T> T createContextualProxy(T instance, Class<T> intf)`
  - `<T> T createContextualProxy(T instance, Map<String,String> executionProperties, Class<T> intf)`
- For returning the execution properties on the given contextual object proxy instance
  - `Map<String,String> getExecutionProperties(Object contextualProxy)`

ContextService

Code Example – Creating Contextual Object Proxy

```
// Within your servlet or EJB method...
@Resource
ContextService ctxSvc;
void businessMethod() {
  Runnable runnableTask = new Runnable() {
    void run() {
      // Interact with a database... use component's
      // security
    }
  }
  // Wrap with the current context
  Runnable runnableTaskWithCtx = (Runnable)
    ctxSvc.createContextualProxy (runnableTask,
Runnable.class}
  // Store the runnable with context somewhere and
  // run later..
  store.putIt(runnableTaskWithCtx);
```

ContextService

Code Example – Using Contextual Object Proxy

```
// Retreive the Runnable with Context
Runnable runnableTaskWithContext = store.getIt();

// Runnable will run on this thread, but with the
// context of the servlet/EJB that created it.
runnableTaskWithContext.run();

// If the Runnable implemented Serializable and it
// was serialized/deserialized... the context would
// still come with it.
```

Resources

- JSR 236 page
  - http://jcp.org/en/jsr/detail?id=236
- java.net project for spec work
  - http://concurrency-ee-spec.java.net
- JSR 236 javadoc
  - http://concurrency-ee-spec.java.net/javadoc/
- java.net project for RI work
  - http://java.net/projects/cu-javaee