Work Manager for Application Servers

International Business Machines Corp. and BEA Systems, Inc.
Version 1.0
November 2003

Authors

John Beatty, BEA Systems, Inc.
Chris D Johnson, IBM Corporation
Revanuru Naresh, BEA Systems, Inc.
Billy Newport, IBM Corporation
Andy Piper, BEA Systems, Inc.

Copyright Notice

License

Status of this Document

Introduction

The Work Manager for Application Servers specification provides a work service for use within managed environments on the Java™ platform, such as Servlets and EJBs. The work service provides a high-level programming model that enables applications to logically execute multiple work items concurrently under the control of the container. In essence, the work manager provides a container-managed alternative to using the `java.lang.Thread`, which is inappropriate for use within applications hosted in managed environments.

The Work Manager for Application Servers specification enables a number of common use cases:

- A Servlet or JSP needs to aggregate data from various sources and render an HTML page after all the data has been retrieved. In this case, the Work Manager API could be used to retrieve the data in parallel and allow execution to continue once all the data is ready.
- An EJB needs a result from any one of several network services in order to complete its task. The EJB can use the Work Manager API to initiate concurrent requests to the network services and continue execution once one of the services has completed.

When inside managed environments, this Work Manager API is a much better alternative to `java.lang.Thread`, as `Thread` should never be used by application-level code within managed environments as the container needs full visibility and control over all executing threads. Also, this Work Manager API is a better alternative than the J2EE Connector Architecture 1.5 [1] Work Service, as the JCA Work Service is tightly coupled with the JCA framework and thus does not provide a sufficiently independent API for use outside JCA. In particular, the JCA `javax.resource.spi.work.WorkManager` interface exposes methods taking `javax.resource.spi.work.ExecutionContext`, which is not generally the context mechanism that should be used by J2EE applications.

This Work Service API thus provides a clean, simple, and independent API that is appropriate for use within any J2EE container.

This specification is organized as follows:

- **Architecture** describes the design of the Work Manager API
- **Deployment** discusses how Work Managers are configured by deployment descriptors
- **Examples** provides a series of examples showing common usages of the Work Manager API
- The Java API is provided as Javadocs in a separate file

Architecture

The Work Manager for Application Servers specification is comprised of six primary interfaces: `WorkManager`, `Work`, `WorkItem`, `RemoteWorkItem`, `WorkListener`, and `WorkEvent`. The `WorkManager` interface provides a set of `schedule`() methods whereby `Work` can be scheduled for execution. The `WorkManager` then returns a `WorkItem`, which can be used to get the status of the in-flight work. The `WorkManager` executes the scheduled work using an implementation-specific strategy. Most implementations will use thread pools. Configuration of `WorkManager` thread pools or other resources is vendor-dependent.

A managed environment can support an arbitrary number of independent `WorkManager` instances. The primary method for obtaining a `WorkManager` instance is through a JNDI lookup to the local Java environment (i.e., `java:comp/env/wm/[work manager name]`). Thus, Work Managers are configured at deployment time through deployment descriptors as `resource-refs` (see **Deployment** below). Each JNDI `lookup()` of a specific `WorkManager` (e.g. `wm/MyWorkManager`) returns a shared instance of that `WorkManager`. `WorkManager` is a thread-safe.

This specification places no requirements on persistence of in-flight `Work`: if the managed environment is shut down or fails, the work will be irrevocably lost unless the particular implementation in use supports a higher quality of service.

Remote Execution of Work

The Work Manager API supports, but by no means mandates, implementation strategies whereby `Work` can be executed in a JVM that is remote with respect to the JVM on which the `WorkManager` is executing. Implementations may choose to farm out `Work` to remote JVMs when the underyling platform is a parallel architecture and supports high-speed communication between JVMs, for example.

If a `Work` instance that is scheduled on a `WorkManager` implements `java.io.Serializable`, this indicates to the `WorkManager` that remote execution (in a separate JVM) of that `Work` is possible. In this case, the `WorkManager` returns a `RemoteWorkItem`, and thus the client can reliably downcast from `WorkItem` to `RemoteWorkItem`. Note that many implementations of `WorkManager` will execute the `Work` locally even if the `Work` instance implements `java.io.Serializable`.

If the client's `Work` instance implements `java.io.Serializable`, the client must not rely on the `Work` instance submitted to the `WorkManager` to be fresh. Rather, the client should use the `getResult()` method on the `RemoteWorkItem`. This returns the `Work` instance after it has been deserialized from remote execution. Note that in some implementations, the `Work` instance submitted to the `WorkManager` may be fresh, but this is not guaranteed behavior.

Work Listener

A `WorkListener` can be specified when work is being scheduled. The `WorkManager` will call back on `WorkListener` for various work events (e.g. accepted, rejected, started, completed).

`WorkListener` instances are always executed in the same JVM as the thread that scheduled the `Work` with the `WorkManager`.

Waiting for Completion of Work

`WorkManager` also provides simple APIs for common join tasks. `WorkManager` provides two semantics:

- `waitForAll()`: blocks until all specified `WorkItem`s complete, or until the specified timeout. Returns `true` if all items completed within the specified timeout value, and `false` otherwise.
- `waitForAny()`: blocks until any of the specified `WorkItem`s complete until the specified timeout and returns the Collection of completed `WorkItem`s. If no `WorkItem`s completed within the specified timeout, `null` is returned.

Two special timeout values are defined:

- `WorkManager.INDEFINITE`: Waits idefinitely for all/any of the work to complete.
- `WorkManager.IMMEDIATE`: Indicates a peek operation. i.e., the WorkManager returns immediately.

Deployment

Applications signal their need for a work manager through including a resource-ref in the appropriate deployment descriptor (e.g., web.xml, ejb-jar.xml, ra.xml, etc.). The absolute name for the JNDI namespace for `WorkManager` objects is `java:comp/env/wm`, and thus the relative name for use within the resource-ref is simply `wm`.

The following provides an example resource-ref fragment configuring a WorkManager named MyWorkManager:

```
<resource-ref>
  <res-ref-name>wm/MyWorkManager</res-ref-name>
  <res-type>commonj.work.WorkManager</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

Examples

The following example shows a `WorkManager` being looked up in JNDI and used to schedule work:

```
import commonj.work.*;
...
RetrieveDataWork work1 =
       new RetrieveDataWork(new URI("http://www.example.com/1"));
RetrieveDataWork work2 =
       new RetrieveDataWork(new URI("http://www.example.com/2"));
InitialContext ctx = new InitialContext();
WorkManager mgr = (WorkManager)
       ctx.lookup("java:comp/env/wm/MyWorkManager");
WorkItem wi1 = mgr.schedule(work1);
WorkItem wi2 = mgr.schedule(work2);
```

This example uses a `RetrieveDataWork` class, which is a fictitious worker classes that retrieves data from a resource specified by a URI:

```
public class RetrieveDataWork implements Work {
       private URI uri;
       private String data;

       public RetrieveDataWork(URI uri) {
              this.uri = uri;
       }

       public void release() {
              // release my resources
       }

       public boolean isDaemon() {
              return false;
       }

       public void run() {
              // do the actual work here
              data = "Hello, World";
       }

       public String getData() {
              return data;
       }

       public String toString() {
              return "RetrieveDataWork(" + uri + ")";
       }
}
```

The following example shows an example deployment descriptor for a Servlet that configures the `WorkManager` used above.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app …>
  <display-name>A Simple Application</display-name>
  <servlet>
    <servlet-name>OrderTracking</servlet-name>
    <servlet-class>com.mycorp.OrderTracking</servlet-class>
  </servlet>
  <resource-ref>
    <res-ref-name>wm/MyWorkManager</res-ref-name>
    <res-type>commonj.work.WorkManager</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
  </resource-ref>
```

```
</web-app>
```

The following example, building on the prior example, shows how the application can block waiting for these work items to complete:

```
// block until all items are done
Collection coll = new ArrayList();
coll.add(wi1);
coll.add(wi2);
mgr.waitForAll(coll, WorkManager.INDEFINITE);
```

Once the application knows that work is completed, the data can be retrieved from the `Work` object:

```
System.out.println("work1 data: " + work1.getData());
System.out.println("work2 data: " + work2.getData());
```

The next example is a slight variation on the example above: the application blocks waiting for *any* of the items to complete. `waitForAny()` returns the `WorkItem`(s) that completed, at which point we can extract the result and continue:

```
String result = null;
Collection coll = new ArrayList();
coll.add(work1);
coll.add(work2);
Collection finished = mgr.waitForAny(coll, WorkManager.INDEFINITE);
if(finished != null) {
	Iterator i = finished.iterator();
	if(i.hasNext()) {
		WorkItem wi = (WorkItem) i.next();
		if(wi.equals(wi1)) {
			result = work1.getData();
		} else if(wi.equals(wi2)){
			result = work2.getData();
		}
	}
}
```

If the concrete class that implemented the `Work` interface also implements `Serializable`, then the following code above can be simplied because `RemoteWorkItem` supports the `getResult()` method, which returns the `Work` instance, which typically holds the result state. This alleviates the application code from correlating `WorkItem` instances back to the original `Work` instances.

```
// block until any of the items are done
String result = null;
Collection coll = new ArrayList();
coll.add(work1);
coll.add(work2);
Collection finished = mgr.waitForAny(coll, WorkManager.INDEFINITE);
Iterator i = finished.iterator();
if(i.hasNext()) {
	RemoteWorkItem wi = (RemoteWorkItem) i.next();
	RetrieveDataWork work = (RetrieveDataWork) wi.getResult();
	result = work.getData();
}
```

The application can also check the status of the `WorkItem` instances at any time:

```
if(wi1.getStatus() == WorkEvent.WORK_COMPLETED) {
	System.out.println("wi1 completed");
}
```

When scheduling work with a `WorkManager`, a `WorkListener` can be used. To use a `WorkListener`, a concrete class first needs to be defined that implements the `WorkListener` interface:

```
import commonj.work.WorkEvent;
import commonj.work.WorkListener;

public class ExampleListener implements WorkListener {

	public void workAccepted(WorkEvent we) {
		System.out.println("Work Accepted: " + we.getWork());
	}

	public void workRejected(WorkEvent we) {
		System.out.println("Work Rejected: " + we.getWork());
	}

	public void workStarted(WorkEvent we) {
		System.out.println("Work Started: " + we.getWork());
	}

	public void workCompleted(WorkEvent we) {
		System.out.println("Work Completed: " + we.getWork());
	}
}
```

Once the listener class is defined, it can be used in conjunction with the `WorkManager`:

```
RetrieveDataWork work1 =
	new RetrieveDataWork(new URI("http://www.example.com/1"));
RetrieveDataWork work2 =
	new RetrieveDataWork(new URI("http://www.example.com/2"));
InitialContext ctx = new InitialContext();
WorkManager mgr = (WorkManager)
	ctx.lookup("java:comp/env/wm/MyWorkManager");
WorkListener listener = new ExampleListener();
WorkItem wi1 =
	mgr.schedule(work1, listener);
WorkItem wi2 =
	mgr.schedule(work2,listener);
```

References

[1] JSR 112, J2EE Connector Architecture 1.5. http://www.jcp.org/en/jsr/detail?id=112

Trademarks

IBM is a registered trademark of International Business Machines Corporation.

BEA is a registered trademark of BEA Systems, Inc.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.