

Jactor2 Revisited by Example

by Bill la Forge, 2014

Jactor2 is a robust and high-performance alternative to threads and locks. Jactor2 Revisited focuses on a subset of the API that is easy to learn but reasonably comprehensive.

The HelloWorld Example

```
package org.agilewiki.jactor2.core.revisited;

import org.agilewiki.jactor2.core.blades.IsolationBladeBase;
import org.agilewiki.jactor2.core.impl.Plant;
import org.agilewiki.jactor2.core.requests.AsyncResponseProcessor;
import org.agilewiki.jactor2.core.requests.impl.AsyncRequestImpl;

public class HelloWorld extends IsolationBladeBase {

    public static void main(final String[] args) throws Exception {
        new Plant();
        new HelloWorld();
        System.out.println("initialized");
    }

    public HelloWorld() throws Exception {
        new AIO("run") {
            @Override
            protected void processAsyncOperation(final AsyncRequestImpl _asyncRequestImpl,
                final AsyncResponseProcessor<Void> _asyncResponseProcessor)
                throws Exception {
                System.out.println("Hello world!");
                Plant.close();
                System.out.println("finished");
            }
        }.signal();
    }
}
```

Output:

```
initialized
Hello world!
finished
```

The *HelloWorld* class is a blade. It has a reactor that is created when the default constructor of *IsolationBladeBase* is called.

```
public static void main(final String[] args) throws Exception {
    new Plant();
    new HelloWorld();
    System.out.println("initialized");
}
```

The *main* method does three things:

1. An instance of *Plant* is created. This provides the operating environment and configuration for the reactors.
2. An instance of *HelloWorld* is created. And
3. The line *initialized* is printed, as this completes the program initialization.

```
public HelloWorld() throws Exception {
    new AIO("run") {
        @Override
        protected void processAsyncOperation(final AsyncRequestImpl _asyncRequestImpl,
            final AsyncResponseProcessor<Void> _asyncResponseProcessor)
            throws Exception {
            System.out.println("Hello world!");
            Plant.close();
            System.out.println("finished");
        }
    }
}
```

```

    }.signal();
}

```

The constructor creates a *run* signal which is passed to the *HelloWorld* blade via its reactor. On receipt of this signal, the blade prints the line *Hello world!*, closes the operating environment and then prints the line *finished*.

Notes:

1. The *AIO.signal* method can be called from any thread and within any context. In this case the method was called from the main thread.
2. *AIO* is a nested class, defined in one of the super classes of *HelloWorld*. This is how the *signal* method accesses the reactor of *HelloWorld*.

The Worker Blade

```

package org.agilewiki.jactor2.core.revisited;

import org.agilewiki.jactor2.core.blades.IsolationBladeBase;
import org.agilewiki.jactor2.core.requests.AsyncResponseProcessor;
import org.agilewiki.jactor2.core.requests.impl.AsyncRequestImpl;

public class Worker extends IsolationBladeBase {
    public final String id;
    private int count;

    public Worker(final int _id) throws Exception {
        id = "Worker" + _id;
    }

    public int getCount() {
        return count;
    }

    public AO<Void> run(final long _iterations, final int _timeoutMillis) {
        return new AO<Void>("run" + id) {

            @Override
            protected void processAsyncOperation(final AsyncRequestImpl _asyncRequestImpl,
                final AsyncResponseProcessor<Void> _asyncResponseProcessor)
                throws Exception {
                _asyncRequestImpl.setMessageTimeoutMillis(_timeoutMillis);
                System.out.println(id + ": started " + count++);
                for (long i = 0L; i < _iterations; i++);
                System.out.println(id + ": finished " + count);
                _asyncResponseProcessor.processAsyncResponse(null);
            }
        };
    }
}

```

The *Worker* blade is useful for simulating a CPU load and we will use it in a number of examples. It has one operation, *run*, which returns an Asynchronous Operation, *AO*, that can be used to pass the *run* request to *Worker*.

Like *AIO*, *AO* is defined as a nested class in a super class of *Worker*, which again is how it can access the reactor of *Worker*. But unlike *AIO*, *AO* can not be used to send a signal. (This is the only difference—*AO* is the super class of *AIO*.)

Note that *count*, which is the number of times a run request has been received, is *private* and is only updated when processing a request, *run*. This means that there will be no race conditions for *count*, as requests are processed strictly one at a time.

We have not yet covered the *AsyncRequestImpl.setMessageTimeoutMillis* method, but we will do that after covering some examples that use *Worker*.

