JSR 352 Expert Group

Working Session

23 March 2012

Bean Instantation

- DI allowed, but not required
- “Bean” names in Job XML (no class names)
- Consistent annotations for developers to identify “batch beans”
- Batch deployment descriptor (maps bean name to class name)
- Replaceble “bean instantiation” plugin for batch container
  - Default plugin reads batch deployment descriptor
  - Spec implementor can supply plugin that exploits DI

Parameters and XML

- Proposal
  - Provide way to pass parameters to batch applications
    - Support @Property annotation in code
    - Support property element in Job XML
      - Specifiable on: Job, listener, step, reader, writer
    - Batch container assigns property from XML to annotated field
  - Support substitution in Job XML
    - E.g. ${fileName}
  - Support optional vs required substitutions (optional has default value)
  - Support passing substitution values during “job launch”
    - E.g. java –jar mybatch.jar fileName=“/tmp/tmp.txt”
  - Expose properties in Job/Step context
- Example:
  - In code:

    @Property(“FileName”) String fname;

  - In Job XML:

```
<step id=“DoPostings”>
    <property name=“FileName” value=“/tmp/…”/>
</step>
```

- Question: should support typed properties or String type only is sufficient?

Exit Codes and Step Conditions

- Follow Spring Batch Batch/Exit Status approach
  - Batch Status: COMPLETED, STARTING, STARTED, STOPPING, STOPPED, FAILED, ABANDONED or UNKNOWN
  - Exist Status = Batch Status by default; can be overridden by user
- Follow Spring Batch model step conditions
  - “next on”, “fail on”, “stop on”, “end on”, “decision”

Batch Metrics

- Per job:
  - Number of restarts
  - Total elapsed time
- For “chunking” steps:
  - Number of records filtered
  - Number of records Skips
  - Number of retries
  - Total elapsed time
  - Records per second

Java EE Integration

- initialization hook to initialize the batch container
- config plugin to get batch container configuration from a product-specific source
- transaction plugin for checkpoint control
- dual mode plugin for job repository – JDBC or file (file is default)
- JNDI reference to access the batch container
  - e.g. java:comp/env/BatchContainer
- CDI “bean instantiation plugin”

Next Steps

Discussion

Reference Slides

Bean Instantiation: Ingredients …

- Consistent “bean name” approach in Job XML

```
<job id="PostingsJob">
        <step id="Step1">
                <chunk reader="..." processor="DoPostings" writer="..."/>
        </step>
</job>
```

- Consistent annotation(s) to identify “batch beans”

```
package com…;

@ItemProcessor
public class DoPostings {
    @ProcessItem PostingOut process(PostingIn item) {...}
}
```

- “Batch deployment descriptor”

  META-INF/batch.xml: <item-processor id="**DoPostings**" class="com....DoPostingsImpl"/>

- Batch Container “bean instantiation” plugin
  - public Object createBean(String beanName) { … }
  - Default implementation uses “batch.xml”

Using DI Containers

- Spec implementor can add DI at their discretion by supplying alternate “bean instantiation” plugin.
- Note different ways “beans” are identified:
  - CDI - @Named
  - Spring, Blueprint - @Service (or Spring XML)
- For batch container that supports CDI:
  - Spec implementer supplies batch container plugin specific to desired DI container
  - Developer uses annotations specific to target DI container – e.g.

```
@Named(“DoPostings”)
@ItemProcessor
public class DoPostingsImpl {
    @ProcessItem PostingOut process(PostingIn item) {...}
}
```

Using DI Containers

- CDI “bean instantation” plugin:

```
public static Object createBean(String beanName) throws Exception {

    ContainerLifecycle lifecycle =
        WebBeansContext.currentInstance().getService(ContainerLifecycle.class);
    lifecycle.startApplication(null);

    BeanManager beanManager = lifecycle.getBeanManager();
    Bean<?> bean = beanManager.getBeans(beanName).iterator().next();

    return lifecycle.getBeanManager().getReference(
            bean, Object.class, beanManager.createCreationalContext(bean));
}
```