Common Persistible Processing Execution Runtime

Best Practices

Version 1.0.1

Why you should use COPPER?

Use COPPER if you want to benefit from the COPPER notation and the COPPER features, from which the outstanding one is COPPER's asynchronous processing of workflows, based on a special Java notation.

Whenever you need to realise service calls to external systems or to a database asynchronously, COPPER offers the sequential notation, which allows for asynchronous waiting, without binding or blocking the current thread.

Example:

```
@Override
public void main() throws InterruptException {
      System.out.println("started");
      final String cid = mockAdapter.foo("foo");
      wait(WaitMode.ALL, 5000, cid); // asynchronous wait
      Response<?> r = getAndRemoveResponse(cid);
      System.out.println("finished - response="+r);
}
```

COPPER releases the current *wait* thread immediately released and can execute other workflow instances. Only when the asynchronous response is received, a free thread is used for the further processing of the workflow instance. Thus, the number of workflow instances that you can execute with asynchronous communication parallely is higher than the number of possible operating system threads.

This is highly advantageous if you have to deal with long response times for asynchronous requests, because most of the workflow threads would be busy with waiting. Another plus is the faster and more efficient processing due to the fact that the operating system does not need to perform so many thread context changes.

In addition to this enhanced scalability, COPPER as well offers more precise mechanisms for the priorisation of workflow instances. The priority of each workflow instance is defined as a numerical value, which you can dynamically modify at runtime. It defines whether a workflow instance should be treated in preference to another one. You can of course define your own priorisation rules.

Transient Workflow vs. Persistent Workflow

As the name says, transient workflows are not stored in a permanent medium such as a database, but only reside in the system main memory.

Typically, you would use a transient workflow for processing reading requests or in cases where the workflow instances must not survive the end of lifetime of the related Java process.

Persistent workflows are stored in a permanent medium, usually in a database so that you can restore them at anytime. They are typically used in the following situations:

- **Long-running Tasks**

  Workflow instances exist over a longer time period such as days, weeks, or even months. In such cases, the workflow instances must survive starting and stopping the application.

- **Crash Recovery**

  In case of a crash recovery, the affected workflow instances must be restored. To enable this, so-called checkpoints are written to the database.

- **High Availability/ Load Distribution**

  COPPER runs in a distributed environment. Multiple copper engines, running on different nodes, are then coupled to a cluster. This offers high availability, load distribution and automatic fail over in case that one or more nodes should crash. Please note that this feature requires a high available database system.

Which Data you should add to your Workflow

Workflow data are the fields resp. the members of the workflow Java class, i.e. the *Data Object*s in the workflow (see *de.scoopgmbh.copper.Workflow.getData()*) as well as all data that during the execution of a workflow method lie on the stack, i.e. local variables and method parameters.
When you use transient workflows, you can use any data type within a workflow without any restrictions..
For persistent workflows, however, COPPER implies some technical and organisational restrictions.
The following data are stored with the workflow instance when generating a checkpoint:

- all fields in the workflow that are not declared as *transient*,
- *Data Object* that belongs to the workflow,
- all data that are on the stack while executing the COPPER *wait*. These are the local variables as well as the method parameters.

Based on this list, we can derive the following best practices:

- Implement only required data into the workflow, becaue all data must be stored within the frame of a checkpoint. Less data generate a smaller footprint, which results in a higher performance.
- Avoid local variables in workflow methods that directly or indirectly use COPPER wait , i.e. methods that write a checkpoint.
  - Move code blocks by means of *refactoring* into own methods.
  - Set variables or parameters that are not needed anymore to null.
- Reduce dependencies to a minimum
  - Do not use any externally defined interface data structures. Map external data structures to internal data structures.
  - Try to use less data types in order to reduce dependencies in order to make any later migration of workflow instances as easy as possible.
- Declare references to service beans within a workflow as *transient*. The beans should be set by COPPER via *AutoWire*.
- COPPER uses by default the *Java Object Serialization*, i.e. all data object classes must implement the *Java Interface Serializable*.

Which Code is part of the Workflow?

The workflow implements the technical process semantic (high level). The workflow code, i.e. the methods, should contain only the process control of the logical workflow steps.

Example:

- Call system *A* and wait for response from *A*.
- Write an AuditTrail entry.

  - Cancel if *A* responds with an error.
  - For all elements in response from *A*:
    - Transmit the element to *B* and wait for response from *B*.
    - Write an AuditTrail entry.
  - …

The processing itself, e.g. the mapping from an internal to an external interface should lie outside the workflow.

You should keep the dependencies to another source code at a minimum. Avoid dependencies to external interfaces.

COPPER supports the usage of loggers such as *Simple Logging Fascade*, *Java* or *Log4J*.

To minimise the dependencies within a COPPER project and to structure the source code properly, we recommend sticking to the project structure as described in the following paragraph.

How to structure your COPPER Project

Create separate sub-projects for the following:

- Workflows
  - Contains the COPPER workflows and has only one dependency to the *Internal Interfaces* sub-project. Thus, you avoid dependencies to the implementation of internal interfaces or to external interfaces or data structures.
- Internal Interfaces
  - Contains the internally used interfaces and data structures
- External Interfaces
  - Contains all externally deployed interfaces and data structures, e.g. *JAX-WS* generated from *WSDL* and *JAXB Binding*.
- Implementation of Interfaces
  - Contains the implementation of the internally used interfaces. It might contain adapters that encapsulate service calls to external interfaces.

How to Version your Workflow

After deploying a workflow in a production environment, modifications on the workflows such as bug fixes or new features might be necessary. In case of a persistent workflows, it raises the question if workflow instances that already exist in a production environment will be running after deploying a newer workflow version.

In COPPER, a persistent workflow instance references the related workflow via its Java class name. If you modify a workflow and redeploy it, then all already existing and still active workflow instances and all workflow instances that are generated at that point in time, will use this new workflow.

This feature might be helpful when a modification should impact existing workflow instances. In such case, make sure that only downward-compatible modifications are performed on the workflow. The *COPPER-workflow-compatibility-rules.pdf* document contains a list of compatible modifications.

Please be careful when performing such a modification. COPPER does not check during deployment whether existing workflow instances are still compatible. In case of incompatibilities, runtime errors might occur when you reactivate the workflow instance. In case of an error, COPPER will set the workflow instance to an error state and will stop execution until a manual retry. This allows for the correction of the erroneous instance and for a later retry.

It is easier if a modification should not impact an existing workflow instance. In such case, you can just create a copy of the workflow resp. of its Java class and can define a new workflow/ class name. We recommend using a serial version number in the class name, e.g. *ooWorkflow_001*, *FooWorkflow_002*, etc...

Make sure to keep the old workflow in the deployment as long as related workflow instances exist in the environment. You should delete the workflow only when there are no instances of that workflow in any environment.

To avoid that you have to modify the related source code in such places where workflow instances are generated and started after the introduction of a new workflow version, COPPER provides a so-called *WorkflowDescriptionAnnotation* that you can add to the workflow. It contains an alias and a version number for major and minor versions as well as for the patch level.

Thus, you can generate a workflow instance for an alias or for a specific version or to generate the newest version, e.g.

engine.run(new WorkflowInstanceDescr<Data>("wfAliasName", new Data(...), null, null, null, new WorkflowVersion(5,1,0));

Finally, below is a summary of all best practices for versioning your workflow:

- After deployment in a production environment, you should not modify a workflow anymore.
- If you, however, have to modify/ patch a productive workflow, apply the compatibility rules described in the *COPPER-workflow-compatibility-rules.pdf* document
- The class name or the workflow package usually contains a version number, e.g. *FooWorkflow_001*
- New workflow versions are generated as a copy of a previous version with an incremented version number.
- You can add a common alias and version information to a workflow by means of the *WorkflowDescription* annotation. With the alias, you can start a workflow instance for a specific version or for the lates version of a major or minor release (see *public void run(WorkflowInstanceDescr<?> wfInstanceDescr)*)