

Workflow Engine

Cloud Examples

## Contents

Overview.....	3
Copper Engine .....	4
Workflow Repository.....	4
Processing Engine .....	4
Processor Threads .....	4
Database Layer .....	5
Batcher .....	5
Database.....	5
Audit Trail .....	5
Audit Trail Event Data.....	6
Persistent Queuing .....	6
Monitoring and Management .....	6
Workflow .....	6
Workflow Execution .....	6
Workflow State.....	7
Workflow Priority .....	7
Workflow Wait .....	7
Savepoints .....	8
Asynchronous Communication .....	8
Aborting Workflow execution .....	8
Workflow Exception Handling.....	8
Engine Deployment .....	9
Payara Server Container .....	9
Postgres Container .....	9
Running multiple Engine Instances .....	9
Overview Multi Instance Support.....	10
Rolling Deployments .....	10
Engine Configuration.....	10
Application Context.....	10
Overview of configurable Engine Resources.....	11
File based Workflow Repository .....	11
Class path based Workflow Repository.....	11
Scotty DB Storage .....	12
Copper Transaction Controller .....	12
Logging Statistics Collector .....	12

Batcher .....	12
Persistent Priority Processor Pool .....	12
Batching Audit Trail .....	12
Scotty Audit Trail Query Engine .....	12
JMX Exporter .....	12
Adapter .....	13
Inbound Adapter .....	13
Outbound Adapter .....	13
PostgreSQL Dialect .....	13
JNDI Data Source Lookup .....	13
Copper GUI .....	14
Overview Panel.....	14
Statistics Overview Panel .....	14
Audit Trail Panel .....	15
Broken Workflows Panel .....	16
Waiting Workflows .....	16
Workflow Repository Panel.....	17
Processor Pools Panel.....	17
License .....	18

## Overview

Copper is a high performance business workflow engine, it is best suited for fully automated workflows.

Based on Java and running workflows from pure Java files, creating and managing workflows is easy. Copper has a straightforward programming model that is easy to understand.

When workflows are defined as code, they become more maintainable, versionable, testable, and collaborative.

With multi instance capability, it is ready to scale in the cloud. The open modular structure allows for individual extensions.

With auto-recovery and restart of failed workflows, Copper makes it easy to manage workflows at scale. A management GUI allows for Operational Insight and Workflow Maintenance.

With fast startup, small footprint and easy instance scaling, it is well suited for cloud environments.

This document serves as a reference for setup and programming of Copper Engine in a cloud native environment. This document is part of the [Copper Cloud Examples](#) on GitHub. It explains the main building blocks of Copper Engine and Copper GUI.

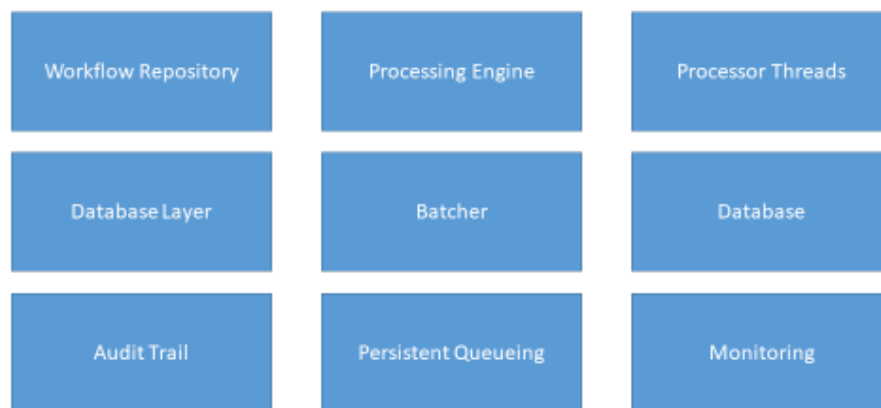
[Copper Cloud Examples](#) covers Docker and Kubernetes examples and shows how to use simple JEE functionality for workflow programming.

Visit [Copper Homepage](#) to get the original documentation and to find out more about the latest Copper Engine and Copper GUI versions.

## Copper Engine

Copper engine consists of multiple modules that are configured by the [Spring](#) framework. New or changed modules can be easily configured. This type of configuration allows for great flexibility. [Copper Cloud Examples](#) uses this flexible approach to embed Copper Engine into an easy to deploy web application context. Copper Engine consists of a manageable amount of modules. Some modules have different variants for specific applications.

### Copper Building Blocks



#### Workflow Repository

Workflows are defined in workflow files. Workflow files itself are Java source files that are compiled at build or runtime. Each workflow has a name and version that identifies the workflow definition. If a workflow is compiled at runtime, the workflow source file is archived in a special folder configured in the application context.

This setup allows for dynamic workflow updates. For cloud native applications, this is often not the preferred scenario. To get a consistent versioning across all cloud artefacts often an application build version is used. So if a cloud application part is updated, new application build is generated. Copper supports this concept by loading workflow class files from a bundled Java workflow package.

#### Processing Engine

The Processing Engine is at the heart of workflow processing. Workflows are prepared for execution and incoming events are assigned to waiting workflows. The Processing Engine manages the lifecycle of the engine and the workflows.

#### Processor Threads

Processor Threads are used to execute workflows. Workflows ready for scheduling are read from an internal in-memory queue and are executed by a Processor Thread. The pool of Processor Threads is configurable and can be adapted to the underlying CPU resources.

### Database Layer

Persistent workflows are read and stored in a database. The database layer abstracts persistence from a concrete database implementation. With this abstraction in place, copper can work with a variety of databases.

### Batcher

Copper is designed for high performance. To speed up SQL execution a special batcher component coordinates SQL inserts and updates by creating a SQL batch from a bunch of single SQL calls.

### Database

Copper supports a variety of relational and non-relational databases. The following list gives an overview of all currently supported databases.

Database	Type
<b>Derby</b>	Relational
<b>H2</b>	Relational (In Memory)
<b>MySQL</b>	Relational
<b>Oracle</b>	Relational
<b>PostgreSQL</b>	Relational
<b>Cassandra</b>	Non-Relational

### Audit Trail

Workflow Instances use Audit Trail to log important events to an internal database table. Display and filtering of Audit Trail events is available in Copper GUI. Audit Trail uses Audit Trail Events for logging. Audit Trail Event has reference fields for identifying request/response pairs, transactions and context.

### Audit Trail Event Data

Field	Description
<b>LOGLEVEL</b>	the level on that the audit trail event is recorded, might be used for filtering
<b>OCCURRENCE</b>	timestamp of the audit trail event
<b>CONVERSATIONID</b>	workflow name with actual version
<b>CONTEXT</b>	workflow action (location in execution path)
<b>INSTANCEID</b>	workflow instance id
<b>CORRELATIONID</b>	id for a request response pair
<b>TRANSACTIONID</b>	Same ID for several conversations, that belong to the same transaction
<b>MESSAGE</b>	a message describing the audit trail event
<b>MESSAGETYPE</b>	Type of the message e.g. XML. Can be used for automatic message conversion

### Persistent Queuing

After creation, Workflow Instances are put into a copper internal persistent queue. This allows for a nearly unlimited amount of workflow instances at a time. Every queue belongs to a Processor Pool Id. A Workflow Instance is associated with a queue by the Processor Pool Id.

### Monitoring and Management

Copper Engine offers a JMX management Interface that allows for queue management, workflow monitoring and workflow restarting. In combination with a time-series database, long time metrics recording of all critical engine parameters is possible.

Copper Cloud Examples has two time-series database examples, [copper-influxdb](#) and [copper-prometheus](#).

Copper GUI is the default Monitoring and Management GUI.

### Workflow

Workflows are at the heart of every workflow engine. Copper makes it easy to create and manage workflows at scale.

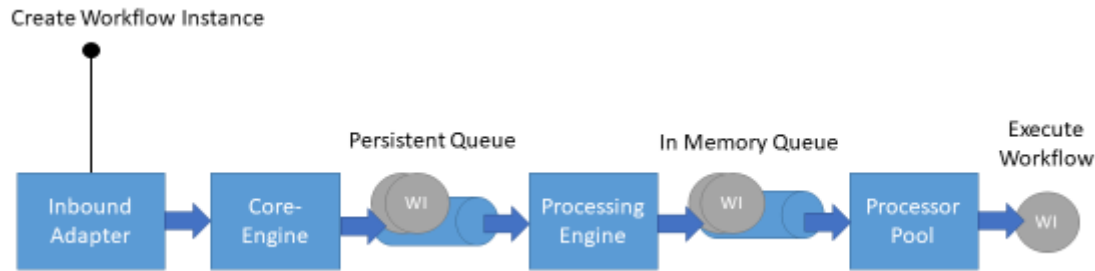
Copper-Engine supports transient and persistent workflows. Only persistent workflows are described in this document. Transient workflows are held in memory and do not need a database, on the whole they are only used for a few use cases.

### Workflow Execution

Workflow definition is kept in a Java file that defines the workflow logic. A Workflow Instance is the runtime incarnation of a workflow Java file plus workflow data, internal state and variables.

Workflows extend the Workflow class, which implements methods for queue and engine interaction. A Workflow Instance is typically created by the Inbound Adapter. On creation, workflow data and the workflow file name are handed over to the engines *run* method. Workflow Instances are then queued in a persistent queue until they are dequeued by the Processing Engine and put into an in-memory queue for execution.

The Processor Pool picks up Workflow Instances from the in-memory queue and hands it over to a Processor Thread if a Thread becomes available in the Processor Pool.



## Workflow State

Workflow Instances have an associated state in relation to the engine. The states can be transitional or final.

Workflow Instance State	Description
<b>RAW</b>	Workflow was initialized, but has not started execution
<b>ENQUEUED</b>	Workflow is in queue and awaits execution
<b>DEQUEUED</b>	Workflow is pulled from database and put into in-memory queue
<b>RUNNING</b>	Workflow is currently running
<b>WAITING</b>	Workflow is in WAIT state. Awake conditions are not fully met
<b>FINISHED</b>	Workflow finished execution normally
<b>ERROR</b>	Workflow stopped execution because of a runtime exception
<b>INVALID</b>	Workflow is invalid e.g. through deserialization error

## Workflow Priority

Workflow Instances have a priority property that governs their priority within the persistent queue. On workflow creation, instances are assigned the lowest priority by default. Priority can be changed within the workflow at any time. Possible priority values are five (Lowest) up to one (Highest). Ordering within the persistent queue depends on enqueue time and priority.

## Workflow Wait

The Workflow *wait* method allows a workflow to be paused. Pausing a workflow can have multiple reasons like waiting for an event.

A common use case for wait is asynchronous communication with external systems. A request is send via an outbound adapter and the workflow then waits until a response is received. Copper offers varying wait functionalities for different use cases.

### Wait Examples:

Method	Description
<code>wait(WaitMode.ALL,1000," 4a90699d-3e48-4534-913b-13bedaf03353")</code>	Wait for a maximum of 1 second for a response with correlation id 4a90699d-3e48-4534-913b-13bedaf03353
<code>waitForAll(" 4a90699d-3e48-4534-913b-13bedaf03353", " 2c90699d-3e48-4534-913b-13bedaf03373")</code>	Wait for all responses
<code>wait(WaitMode.First, 1000, (" 4a90699d-3e48-4534-913b-13bedaf03353", " 2c90699d-3e48-4534-913b-13bedaf03373"))</code>	Wait for first incoming response or a maximum of 1 second
<code>wait(WaitMode.ALL,1, TimeUnit.HOURS, getEngine().createUUID());</code>	Pauses execution for 1 Hour (getEngine().createUUID() is just for filling up the mandatory correlation id parameter)

During a workflow wait, the Workflow Instance is removed from memory and is persisted into the database.

The Workflow Instance is reloaded into memory if all wait criteria are met.



## Savepoints

Savepoints are a great way to relaunch Workflow Instances from a defined point of execution. If a Workflow is restarted after an error or if it is reloaded from a WAIT state, the Workflow Instance will continue its execution from exactly the same point it was persisted.

While Savepoints are set automatically on enqueue or wait, they can also be set manually from within the Workflow by calling the *savepoint* method. Behind the scenes, the *savepoint* method will just resubmit the Workflow Instance to its associated persistent queue.

## Asynchronous Communication

When using the asynchronous communication pattern, the Workflow Instance has to wait for a response with the same Correlation Id that was send in the request. The time between request and response can vary from some milliseconds up to a few month.

If all criteria for a wait statement are met the waiting Workflow Instance is reloaded into memory for execution. After a wait, the asynchronous response must be read with the *getAndRemove* method.

The method takes a Correlation Id and returns the response.

If no response with the corresponding id was received, the method will just return null. Workflows should always be prepared for a null value (e.g. on a timeout).

## Aborting Workflow execution

In case of an unrecoverable workflow error, a Runtime Exception must be thrown to set the Workflow Instance into ERROR state. The instance will remain in this state until it is restarted e.g. from Copper GUI.

Exceptions for notifications failures are wrapped into a *CopperRuntimeException*. This will set the Workflow Instance to ERROR state.

## Workflow Exception Handling

The *main* method of the workflow might throw an *Interrupt* Throwable at any time.

Interrupt is part of the internal workflow handling and should never be thrown or caught by workflow code. Therefore, generic catch statements should not be used in workflow code.

Throwing a RuntimeException puts the workflow into ERROR state. With Copper GUI, the Exception Stack trace for a Workflow Instance in ERROR state can be displayed.

## Engine Deployment

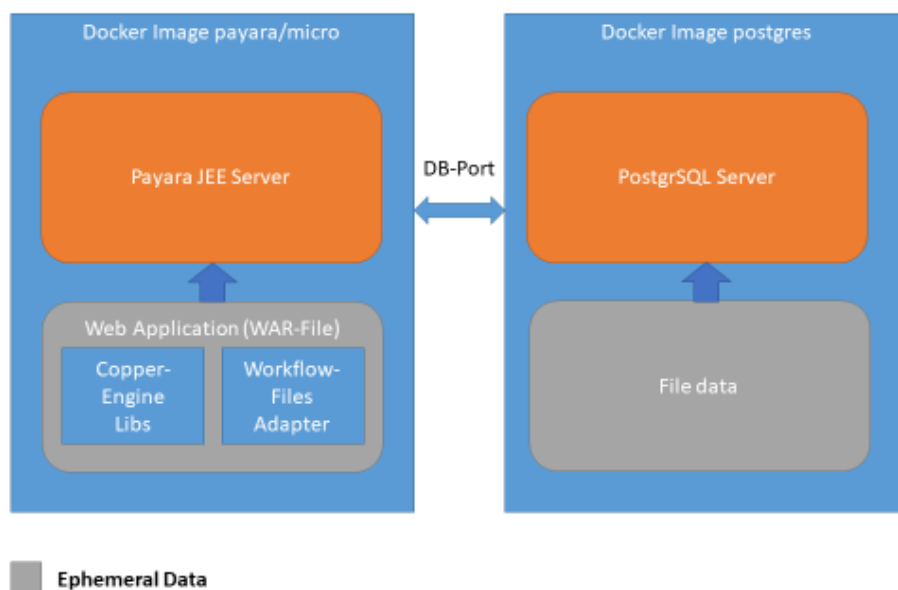
The [copper-starter](#) example provides Docker containers for Copper Engine and PostgreSQL database. A Kubernetes deployment that works with Minikube is available in the [copper-kubernetes](#) example.

### Payara Server Container

[Payara JEE Server](#) is used as a rich JEE environment for Copper-Engine.

The JEE server provides support for server/client side REST, Pooling, Queuing and many other useful functionalities.

The engine itself is packaged into a web app that includes all necessary libraries, workflow definitions and other application specific code. In the [copper-starter](#) example, Docker Compose is used for runtime management.



On startup, Payara Server looks for web apps in the configured deployment directory and starts the default web app. After startup, all REST services defined in the Inbound Adapter are available via Payara Server.

The server logs all available REST services at the end of the web app deployment. In this example the [Payara/micro edition](#) is used, which is optimized for Cloud deployment, fast startup and small footprint.

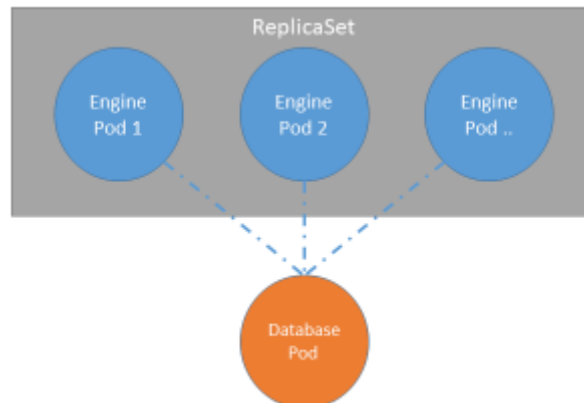
### Postgres Container

Startup of the postgres container is also straightforward. On startup, the default container volume is used to store the internal and public tables. If the Docker container is destroyed all database data is removed along. By adding a persistent volume to the Docker Compose file the data is independent from the container lifecycle.

### Running multiple Engine Instances

For most usage scenarios, it is sufficient to run only one Engine instance, but if needed instance scaling is supported.

The [copper-kubernetes](#) example shows Copper Engine scaling with Kubernetes. In the Kubernetes configuration file, the Copper Engine image is configured for a ReplicaSet.



Copper Engine instances can be scaled by simply changing the *replicas* count in the Kubernetes configuration file. Multi Instance mode is not supported by all databases, because it is based on explicit database level locking.

#### Overview Multi Instance Support

Database	Multi Instance supported
Derby	False
H2	False
MySQL	True
Oracle	True
PostgreSQL	True
Cassandra	False

Multi Instance support has to be activated in the [Spring](#) configuration. The property *MultiEngineMode must be set* to true in the database dialect bean. If enabled Copper will use explicit locking on database level to manage engine access.

#### Rolling Deployments

The multi instance feature also enables rolling deployment scenarios. With no breaking changes, all workflow versions can be executed by the same new deployment version while the old versions are phased out during deployment.

For breaking changes, a new Processor Pool Id for the new version is configured. New workflows are then executed by the new instance versions and old workflow versions are still executed by the old deployment versions. Once all old workflows are finished, the old deployment instances can be decommissioned.

#### Engine Configuration

Core Engine modules are configured by a [Spring](#) context. [Cloud Examples](#) uses a [Spring](#) application context file for engine configuration. The Inbound Adapter is handled by the JEE Server and uses auto wiring for its [Spring](#) beans.

#### Application Context

Copper uses Spring configuration to configure all relevant runtime modules. [Cloud Examples](#) uses an applicationContext.xml file.

## Overview of configurable Engine Resources

Class	Description
<b>FileBasedWorkflowRepository</b>	Directory based workflow repo
<b>ClasspathWorkflowRepository</b>	Workflow repo based on a special Java package
<b>ScottyDBStorage</b>	DB storage controller
<b>CopperTransactionController</b>	database transaction controller
<b>LoggingStatisticsCollector</b>	Collector for runtime statistics
<b>BatcherImpl</b>	SQL Batch component
<b>PersistentPriorityProcessorPool</b>	Workflow processor with worker threads
<b>PersistentScottyEngine</b>	Processing engine for persistent workflows
<b>BatchingAuditTrail</b>	DB based audit trail class
<b>ScottyAuditTrailQueryEngine</b>	JMX Class for audit trail queries with a filter
<b>JmxExporter</b>	Exports Copper JMX Classes
<b>Custom Adapter</b>	Custom inbound and outbound adapters
<b>PostgreSQLDialect</b>	Postgres specific database dialect
<b>JNDI Datasource Lookup</b>	Standard JEE data source lookup

### File based Workflow Repository

The *FileBasedWorkflowRepository* class loads workflow source files from a dedicated workflow directory.

During startup or if a file in the repository changes, the new or changed workflow source file is compiled. Because compilation happens at runtime a full JDK or a compile package like ecj (Eclipse Compiler) is needed. The compiled workflow class files are placed in to a build dir.

### Class path based Workflow Repository

The *ClasspathWorkflowRepository* class reads workflow class files from a special Java package. Other than the file-based repository, no runtime compiler is needed. In this configuration, only a JRE is needed for Server and web app.

The workflow is compiled during application build. Therefore, it shares the same build version with all other application source code, which is a common scenario in modern CI/CD pipelines.

### Scotty DB Storage

The *ScottyDBStorage* class manages persistence for workflow instances and asynchronous responses. It is configured with a transaction controller, a database dialect class and batcher class for SQL batch processing. Inserting and dequeuing persistent workflows are the main tasks for the class.

### Copper Transaction Controller

The *CopperTransactionController* augments a Transaction Controller with a transaction retry mechanism. If a transaction fails the copper transaction controller retries the transaction for a configurable max. retry count. A data source is needed for configuration.

### Logging Statistics Collector

The *LoggingStatisticsCollector* class collects runtime statistics and logs average processing times to the logging system. The collection interval is configurable.

### Batcher

*BatcherImpl* collects simple SQL insert, update and delete statements and executes them in one batch. This boosts overall database throughput. Batcher is configured with a data source and optionally with a statistics collector.

### Persistent Priority Processor Pool

The *PersistentPriorityProcessPool* takes a Transaction Controller and an Id that identifies its persistent queue. The maximum id length is 32 characters.

### Batching Audit Trail

The *BatchingAuditTrail* writes audit trail messages into the database. It is configured with the Database Storage class *ScottyDBStorage* and an optional message data compressor. With a message compressor, the message text is stored in a packed format in the database. In compressed state the message is not in a readable in the database table. Compressed messages are still readable by Copper-GUI.

### Scotty Audit Trail Query Engine

The *ScottyAuditTrailQueryEngine* class is used to query the audit trail table. The class is exposed via Copper JMX. Copper-GUI uses the functionality to display audit trail entries. If configured with a message compressor class it can work on compressed data.

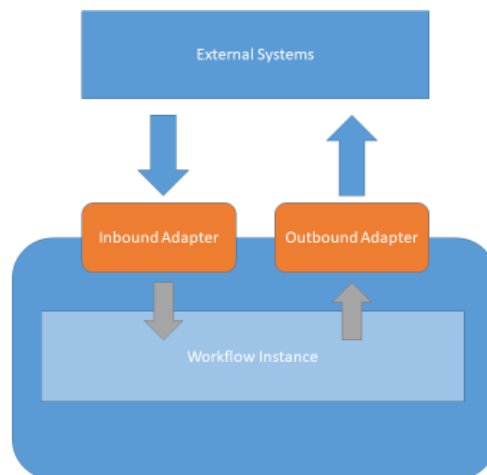
### JMX Exporter

The JMX Exporter exposes all standard Copper JMX beans from the Spring application context.

## Adapter

An Adapter implements one or more business interfaces. In a cloud native environment, typical interfaces are REST and queue protocols. Adapters are managing all inbound and outbound communication for a workflow.

An adapter is configured by a Spring configuration. The [copper-full](#) example comes with an inbound and outbound adapter.



It is good design to have dedicated adapters for each external system or business process.

### Inbound Adapter

In the [copper-full](#) example, the inbound adapter is a REST endpoint that receives workflow data and creates new workflow instances.

The *creditCardCheckCompletion* operation receives responses for the asynchronous credit card check.

### Outbound Adapter

Outbound messages and events are sent by an outbound adapter. Communication with external systems can be synchronous or asynchronous.

In case of asynchronous communication a **Correlation Id** must be sent to match the response for a specific Workflow Instance.

### PostgreSQL Dialect

Databases are often using their own SQL dialect. Copper supports dialects by using a database specific dialect class. In [copper-full](#) example, the `PostgreSQLDialectFix` class is used that includes a fix for a postgres related audit trail bug.

### JNDI Data Source Lookup

[Cloud Examples](#) uses the JEE server payara/micro for workflow engine hosting. Packaged as a web application the engine uses standard JEE pooling and JNDI lookup functionalities.

JNDI Data Source lookup is configured in the `web.xml` file and referenced by Copper Engine via Spring application context file.

## Copper GUI

Copper GUI is bundled in a separate Docker image and is independent from the core engine. The GUI enables engine monitoring and management. Management of multiple engines is supported. The list of GUI panels gives a detailed overview of the GUI functionality.

### Overview Panel

The Overview Panel allows selection of different detail views.

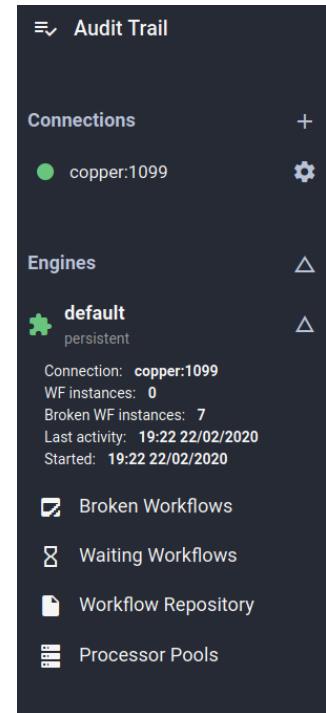
The panel shows all connected engines with their queue name and current running/in error workflows.

A dedicated Connection is configured for each monitored Copper Engine.

The Connection parameters contain host name, JMX port and user/password for port access. Active connections are marked with a green dot in front of the *host:port* entry.

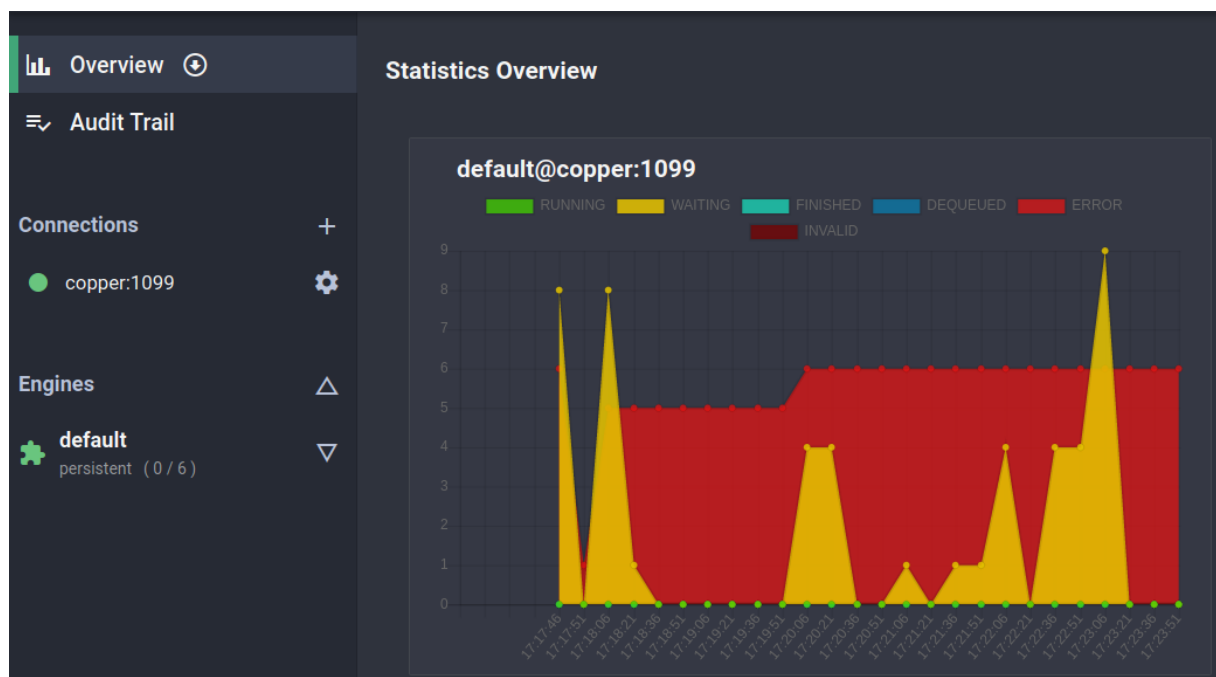
The following views can be selected from the Overview Panel:

- Statistics Overview
- Audit Trails
- Broken Workflows
- Waiting Workflows
- Workflow Repository
- Processor Pools

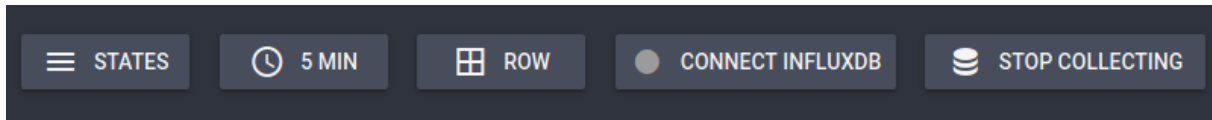


### Statistics Overview Panel

In the Statistics Overview Panel, time series metrics for Workflow Instances are visualized.



The Statistics Overview panel shows instance counts for the different workflow states. A click on the state color bar toggles visibility for the associated state.



The STATES button, switches state metrics on/off.

Fetch Period allows selection of a specific time interval for metrics collection.

ROW Layout switches between different visualization layouts.

CONNECT INFLUXDB opens the connection settings for an optional InfluxDB configuration. With a running InfluxDB metrics can be recorded continuously. Copper GUI can use influxDB data to show metrics for different time ranges.

STOP/START COLLECTING deactivates/activates the collection of metrics by Copper GUI. This button has no effect on InfluxDB recording.

## Audit Trail Panel

All Audit Trail Events logged into the database are available in the Audit Trail Panel.

ID	Occurrence	Log Level	WF Instance Id Conversation Id	Correlation Id Transaction Id	Message Preview	Actions
1	14:36 11/02/2020	1	4de3713f-f021-4924-8a91-f33bca8c367e Order-id: 2		Credit card number: 7277226662626236 not valid	▼
2	17:18 16/02/2020	1	1daef5d6-1f1a-40e1-8596-55326f217370 Order-id: 2		Credit card number: 7277226662626236 not valid	▼
3	17:18 16/02/2020	1	9b5a6fb5-ef5d-4df7-91ff-81c266c3209e Order-id: 2		Credit card number: 7277226662626236 not valid	▼
4	17:18 16/02/2020	1	44893a91-6afc-4000-b975-194167b81027 Order-id: 2		Credit card number: 7277226662626236 not valid	▼
5	17:18 16/02/2020	1	07a4840e-e75f-4df1-9519-2d8a267c7cb6 Order-id: 2		Credit card number: 7277226662626236 not valid	▼
6	17:20 16/02/2020	1	7ff61aaa-084d-404f-b885-826049ae30f8 Order-id: 2		Credit card number: 7277226662626236 not valid	▼

Filtering can be applied to the list of Audit Trails Events. The Filter button opens a Filter mask with event properties and a time range.

**Audit Trail**  
Number of Audit Trail logs: 6

Filter By:

Log Level: ▼ Occurred - From 📅 To 📅

Workflow ID:  Conversation ID:

Correlation ID:  Transaction ID:

✓ APPLY 🔄 CLEAR



## Broken Workflows Panel

This panel shows an overview of all Workflow Instances that are in ERROR or INVALID state. If a workflow instance throws a Runtime Exception, it will enter ERROR state. A workflow instance in ERROR/INVALID state can be restarted via Copper GUI. An INVALID workflow instance is a workflow that tried to execute invalid code (e.g. trying to serialize a non-serializable object).

**Broken Workflows**

Total number of broken workflows: 7







RESTART ALL DELETE ALL FILTER

Filter By:

Include Classname: Created - From To

Include States: Modified - From To

APPLY CLEAR

#	Data	Processor Pool	State	Priority	Timeout	Error Time Creation Last Modification	Actions
1	OrderCheckWorkflow	P#DEFAULT	ERROR	5		17:18 16/02/2020 17:18 16/02/2020 17:18 16/02/2020	  
2	OrderCheckWorkflow	P#DEFAULT	ERROR	5		17:18 16/02/2020 17:18 16/02/2020 17:18 16/02/2020	  

From the panel a single Workflow Instance or all Instance can be restarted. The restart will happen from the last save point. A list filter is available to filter for workflow class, state (ERROR/INVALID) and a time range. With the DELETE ALL button, all Workflow Instances with ERROR/INVALID state are deleted.

## Waiting Workflows

All waiting Workflow Instances are listed in this panel. The list can be filtered by workflow class name and time range. The Panel allows deletion of a single waiting instance or all instances.

**Waiting Workflows**

Total number of waiting workflows: 5







DELETE ALL FILTER

Filter By:

Include Classname: Created - From To

Modified - From To

APPLY CLEAR

#	Data	Processor Pool	Priority	Creation Last Modification	Actions
1	OrderCheckWorkflow	P#DEFAULT	5	20:39 22/02/2020 20:39 22/02/2020	 
2	OrderCheckWorkflow	P#DEFAULT	5	20:39 22/02/2020 20:39 22/02/2020	 
3	OrderCheckWorkflow	P#DEFAULT	5	20:39 22/02/2020 20:39 22/02/2020	 

## Workflow Repository Panel

This panel shows an overview of all available workflow classes in the repository.

Workflow Repository			
Repository Type: ClasspathWorkflowRepository			
#	Alias	Class	Version
1	OrderCheckWorkflow	org.wkl.copper.full.wf.OrderCheckWorkflow	Version: 1.0.0
Per page 10			

## Processor Pools Panel

The Processor Pools Panel shows the actual queue state and overall activity of the Processor Threads.

The SUSPEND button suspends all Processor Threads.

The RESUME buttons resumes Processor Thread execution.

The SUSPEND DEQUEUE button suspends dequeuing from the in-memory queue.

The RESUME DEQUEUE reactivates a suspended in-memory queue.



## License

Copyright 2020 Winfried Klum

Licensed under the Apache License, Version 2.0 (the "License");  
you may not use this file except in compliance with the License.

You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software  
distributed under the License is distributed on an "AS IS" BASIS,  
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
See the License for the specific language governing permissions and  
limitations under the License.

Most information in this document was assembled from Copper Engine source code of version 5.1.0.  
For the faultlessness of the documentation and for damages, which arise from the use of the  
documentation no liability can be assumed.