Point85

Overall Equipment Effectiveness (OEE) Applications User Guide

Version 3.12.0

Kent Randall

February 4, 2025

Table of Contents

OVERVIEW

The Point85 Overall Equipment Effectiveness (OEE) applications enable:

- collection of equipment data from multiple sources to support OEE calculations or general purpose data acquisition

- resolution of a collected data value into an availability reason or produced material quantity to provide input to the performance, availability and quality components of OEE
- calculation of the OEE key performance indicator (KPI) for the equipment using an optional work schedule for defining the scheduled production time
- monitoring of equipment availability, performance and quality events

Sources of equipment availability, performance and quality event data include:

- *Manual*: web browser, Android app or iOS app data entry
- *OPC DA*: classic OLE for Process Control (OPC) Data Acquisition (DA)
- *OPC UA*: OLE for Process Control Unified Architecture (UA)
- *HTTP*: invocation of a web service via an HTTP request
- *RMQ Messaging*: an equipment event message received via a RabbitMQ message broker
- *JMS Messaging*: an equipment event message received via a JMS message broker
- *MQTT Messaging*: an equipment event message received via an MQTT message server
- *Kafka Messaging*: an equipment event message received via a Kafka message server
- *Web Socket Messaging*: an equipment event message received via a web socket server
- *Email*: an equipment event message received via an email server
- *Database Interface Table*: a pre-defined table for inserting OEE events
- *File Share*: a server hosting OEE event files
- *Modbus*: a Modbus master communicating with its slaves.
- *Cron Job*: a cron job scheduled to execute at specified points in time
- GE Proficy Historian: a historian collecting tag data

The Point85 applications supporting OEE are:

- *Designer*: a GUI application for defining the plant equipment, data sources, event resolution scripts, manufacturing work schedule, availability reasons, produced materials and units of measure for data collectors. The designer also includes a dashboard and trending capabilities.
- *Collector*: a headless Windows service or Unix deamon to collect the equipment event data and store it in a relational database
- *Monitor*: a GUI application with a dashboard to view the current equipment OEE and status
- *Operator* Desktop: a GUI application for manual entry of equipment events
- *Operator* Web: a web-application for manual entry of equipment events

- *Operator* Mobile Android App: an Android application for manual entry of equipment events
- *Operator* Mobile isOS App: an iOS application for manual entry of equipment events
- Operator macOS: a macOS application for manual entry of equipment events
- Operator Windows: a Windows application for manual entry of equipment events
- Operator Linux: a Linux application for manual entry of equipment events
- Operator Chrome: a Chrome/Edge browser application for manual entry of equipment events

In addition, two GUI test applications assist in the development of an OEE solution:

- *Tester*: HTTP requester and a message publisher
- *Test Collector*: UI front end for a data collector

Please send comments and suggestions to point85.llc@gmail.com. For more information about OEE, please visit the Point85 website at https://point85.github.io/oee/.

OEE Calculations

OEE is the product of equipment availability, performance and quality each expressed as a percentage. The time-loss model is used to accumulate time in loss categories (or "no loss" if the equipment is running normally). See *[Kennedy]* for details. A data source provides an input value to a data collector's resolver JavaScript function that maps that input value to an output value (reason or production count).

For availability and performance, the output value is a reason that is assigned to one of the following loss categories:

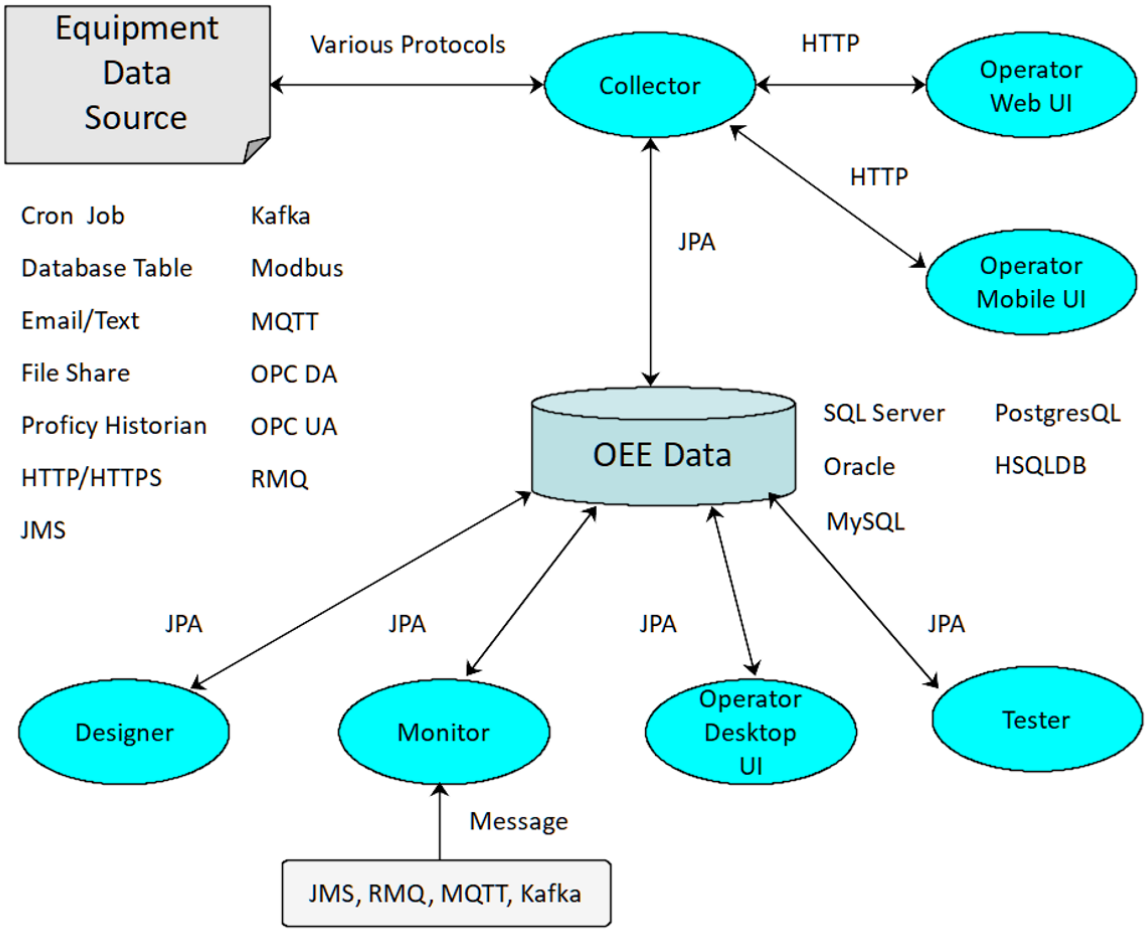
- *Value Adding:* the "no loss" or "running OK" category.
- *Not Scheduled:* this is non-working time. Non-working periods (e.g. holidays) typically are planned in the work schedule that is assigned to equipment.
- *Unscheduled*: working time when the equipment is not scheduled for normal production (e.g. an R&D or laboratory test run).
- *Planned Downtime*: working time when the equipment is not scheduled for normal production but the activity is intended to support production (e.g. planned preventive maintenance).
- *Unplanned Downtime*: working time when the equipment is not available due to an unexpected fault (e.g. motor failure or jam).
- *Setup*: working time when the equipment is being changed over in order to run new material.
- *Stoppages*: minor or short periods of time when the equipment is not producing as expected (such as a blocked or starved condition).

- *Reduced Speed*: the equipment is producing, but not at its design speed or ideal run rate.

For quality or yield, the data source provides a production count in the good, reject/rework or startup & yield categories in the defined units of measure for the material being produced.

ARCHITECTURE

The diagram below is an overview of the system architecture.



The OEE applications can be grouped into design-time and run-time. The design-time Designer application is used to define the plant equipment, data sources, event resolution scripts, manufacturing work schedule, availability reasons, produced materials and units of measure for data collectors. The designer also includes a dashboard and trending capabilities. It uses the Java Persistence API (JPA).

A run-time data collector service receives an input value from an equipment data source of various types and protocols (e.g. OPC UA), and executes a JavaScript resolver on this input value to calculate an output value. The output value is a reason (mapped to an OEE loss category) for an availability event, a new production count (good, reject/rework or startup) for performance and quality events or a

material/job change event (a job is also known as an order, lot or batch). For the case of a custom event, the output value is ignored.

The event data is stored in a relational database where it is available for OEE calculations. Microsoft SQL Server, Oracle 12c/18c/19c Express Edition, MySQL, PostgresQL and HSQLDB are currently supported using the Java Persistence API (JPA).

A manual web-based data collector and desktop collector record the OEE events based on information entered by an operator using HTTP requests. This data is also stored in the relational database.

If the system is configured for RMQ, JMS, Kafka or MQTT messaging, the event data is also sent to a message server to which a run-time Monitor application can subscribe for near real time updates. A monitor displays a dashboard for viewing equipment OEE events. It also displays collector notifications and status information as messages are received. The monitor can also poll the database via JPA for new events to display.

A desktop Operator application stores equipment events in the database via JPA.

A Tester application is used for debugging an OEE application.

The Android mobile Operator app is downloaded from Google's Play Store. After installation, the name or IP address and port of the embedded HTTP server for a Collector application must be specified under the Settings tab. Once a connection is established with the HTTP server, OEE events can be entered.

Similarly, the Apple iOS and macOS apps are downloaded from Apple's App Store.

The Windows, Linux and Chrome/Edge apps are downloaded from the latest OEE mobile release, e.g. Release OEE Mobile · point85/OEE-Mobile.

TECHNOLOGIES

Technologies used for the OEE applications are:

- Java 11+ programing language and JavaFX 17+ for the GUIs
- JavaScript for resolving an input value into an output value based on an OEE event. The OpenJDK Nashorn script engine is used.
- Vaadin and Tomcat for the web application
- j-Interop and OpenSCADA utgard for the OPC DA client
- Eclipse Milo for the OPC UA client
- Hibernate with a Hikari connection pool and JPA interface for persisting data to a relational database
- Jetty for the embedded HTTP server
- RabbitMQ and Erlang for the messaging middle-ware
- ActiveMQ client for JMS

- Eclipse Paho client for MQTT
- Google Gson for message and HTTP request serialization into a JSON string and deserialization
- Java Service Wrapper (JSW) for Windows process and Unix collector daemon
- j2mod for Modbus
- Quartz Scheduler for cron jobs
- Apache Kafka client for publisher and subscriber messaging
- Too Tall Nate Java-WebSocket client and server
- Java mail extension
- Java web sockets library for the web socket source
- Google's Flutter framework with the Dart 3.0+ language for creating the operator applications.

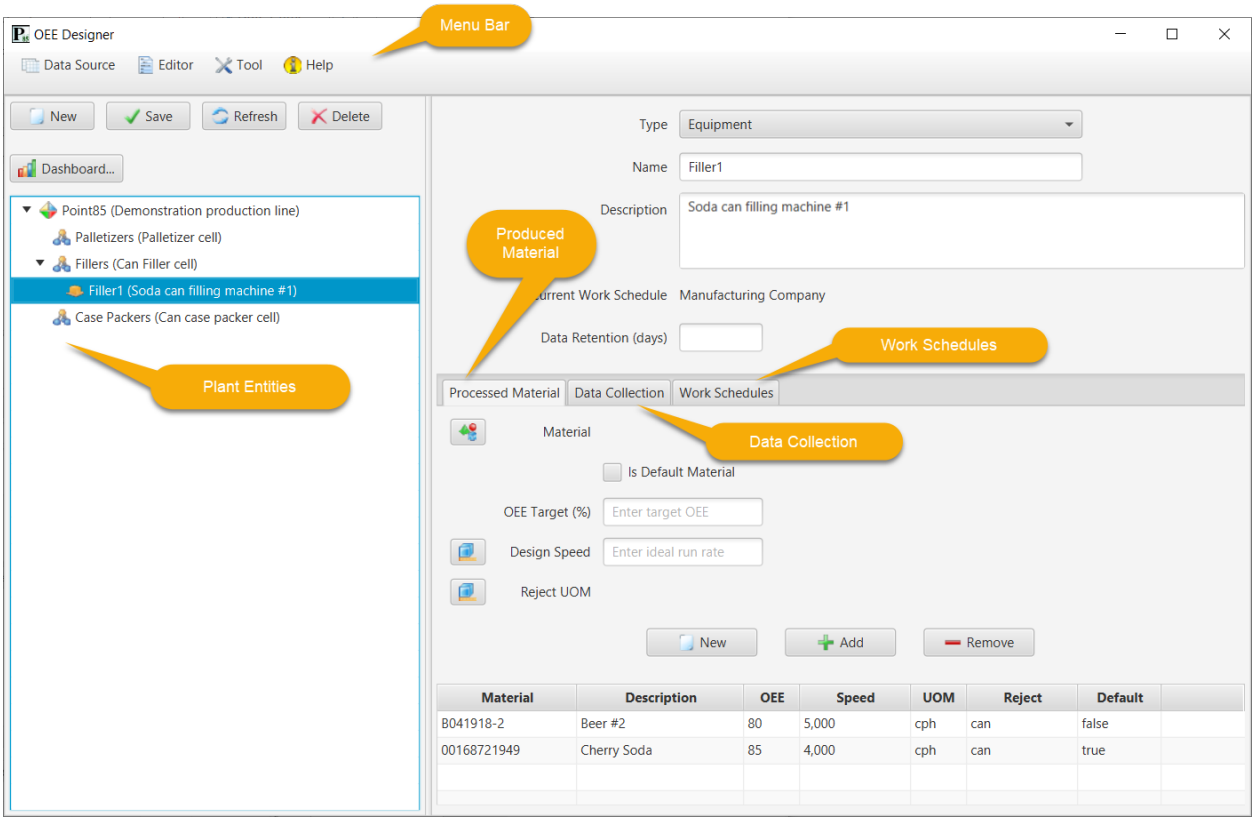
Please see the *Contributing Technologies* section below for additional details.

Designer Application

The Designer application is used to define all aspects of an OEE solution. It is launched from a shell script. A splash screen is first displayed during the time the database connection is being established and plant entity objects are being created.

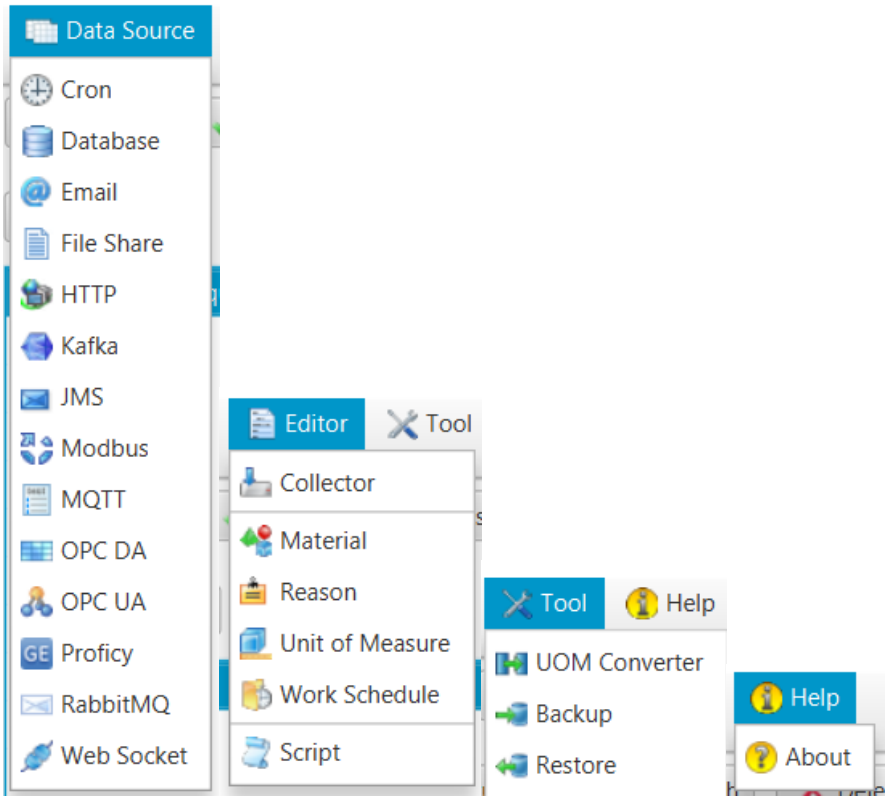
Plant Entity Editor

Upon launch of the Designer, the plant entity editor is displayed. It will be similar to:

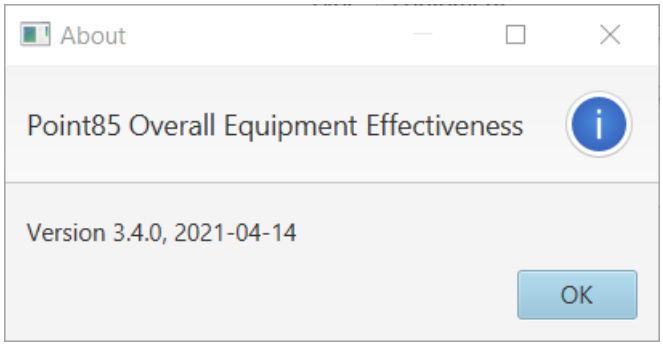


Menubar

Menubar menus that launch other editors or applications are discussed below. Data source editors are launched from the *Data Source* menu, other editors from the *Editor* menu, the UOM conversion tool from the Tool menu and the about dialog from the *Help* menu:



The About button shows an informational dialog:



Entity Hierarchy

To begin, the plant entity physical model on the left side of the editor would typically be defined first. This is the ISA-95 organizational hierarchy of Enterprise -> Site -> Area -> Production Line -> Work Cell -> Equipment and is closely related to the process industry's ISA 88 unit/machine model. Only equipment can have OEE calculations, but higher level entities can have an associated work schedule and data retention period that apply to equipment contained below them. It is not necessary to define all levels, only equipment objects are required for OEE calculations, and any object can be a top-level object.

The physical model editor buttons are:

- *New*: clear the editor to begin defining a new entity

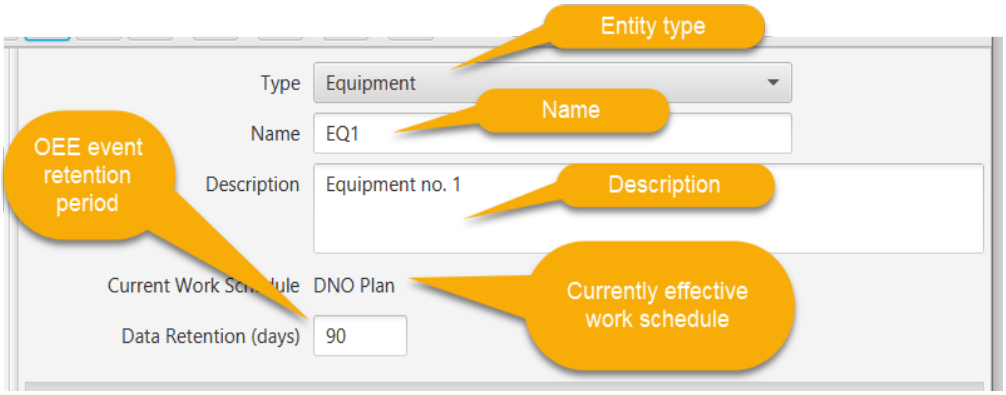
- *Save*: save the selected entity to the database. The parent entity (if any) must be selected first. If a parent is selected, the child must be created of the proper type (e.g. Equipment if the parent is a Work Cell).
- *Refresh*: refresh the selected entity from the database to synchronize the editor with the entity's saved state
- *Delete*: delete the selected entity and any children from the database
- *Backup*: export the selected entity (or all entities if none is selected) to the specified .p85x file

The Dashboard button displays the OEE dashboard for the selected equipment entity. The dashboard is discussed below.

The physical model has a context menu accessed by right-clicking in the left-hand pane. The menu items are:

- *Save All Entities*: save all entities in the hierarchy to the database
- *Refresh All Entities*: restore all entities in the hierarchy from their state in the database
- *Clear Selected Entity*: de-select the entity so that no entity is selected

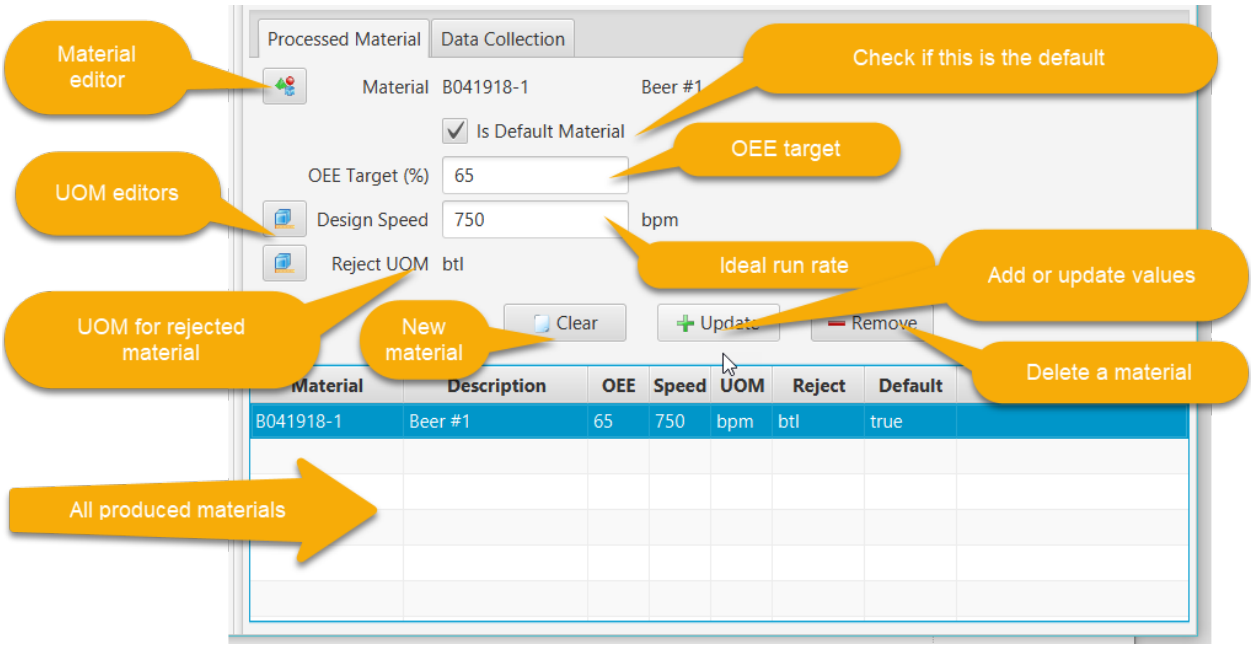
The entity's attributes are displayed and edited in the upper right-hand corner of the editor:



The work schedule and data retention days apply to that entity and to all children below it. For example, a work schedule could be defined for each Site or Area within a plant and have it apply to the contained equipment. A retention of 90 days means that any event records older than 90 days for that equipment will be deleted from the database when a new availability event is recorded if at least one hour as passed since the last (or first) purge. The default retention period is 360 days.

Equipment Processed Materials

Equipment can produce many different materials. This one-to-many relationship is defined in the Processed Material tab:



The editor actions are:

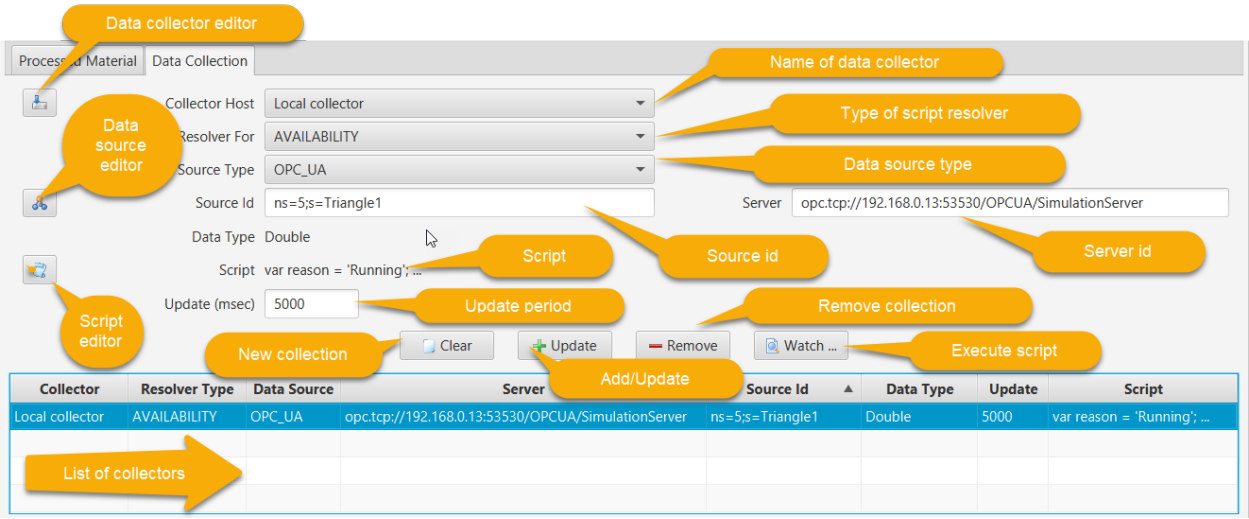
- *New*: clear the editor controls in order to define a new processed material. The Add/Update button text will change to "Add".
- *Add*: After defining the properties of a new material, clicking this button will add it to the list of materials
- *Update*: after selecting an existing processed material, the current values will be moved into the editing controls. Make the necessary edits, then click this button to apply the changes to the processed material.
- *Remove*: after selecting an existing processed material, click this button to remove it from the list.

Note that after making changes to the list of processed materials, the plant entity must be saved to the database by clicking the Save button. An unsaved entity will be marked as such.

The material editor is accessed by clicking on the material button in this tab. The produced material is then selected in the editor and the dialog closed. The unit of measure editor for the design speed and rejects is accessed in a similar fashion.

Equipment Data Collection Events

Equipment can have many different sources of availability and production events. This one-to-many relationship is defined in the Data Collection tab:



The editor actions are:

- *New*: clear the editor controls in order to define a new resolver. The Add/Update button text will change to "Add".
- *Add*: After defining the properties of the resolver, clicking this button will add it to the list
- *Update*: after selecting an existing resolver, the current values will be moved into the editing controls. Make the necessary edits, then click this button to apply the changes to the resolver.
- *Remove*: after selecting an existing resolver, click this button to remove it from the list.
- *Watch*: after selecting a resolver, click this button to launch a dialog to observe execution of the resolver script when a new input value arrives or one is manually entered. The output resolution is not saved to the database. The dialogs are discussed below under the individual data source editors.

Note that after making changes to the list of event resolvers, the plant entity can be saved to the database by clicking the Save button.

A data collector host is a Windows process or Unix daemon that interfaces to one or more data sources to collect data for OEE calculations. A previously defined data collector for this event can be selected in the dropdown, or the editor can be launched by clicking the editor button. This editor is discussed below.

The event type is selected in the next dropdown:

- AVAILABILITY - availability
- PROD_GOOD - good production
- PROD_REJECT - reject/rework production
- PROD_STARTUP - setup & yield production
- MATL_CHANGE - material change

- JOB_CHANGE - job/order change
- CUSTOM - application defined according to the JavaScript being executed

Note that a reason can be associated with a production quantity. If manually inputting the data, enter a production quantity followed by a comma followed by the reason name, e.g. “10, 101” for a quantity of 10 produced with reason named “100”. If set by JavaScript, the value is set into the event resolver (see below).

The data source type (OPC DA, OPC UA, HTTP, RMQ, JMS, MQTT, Kafka, Web Socket, Modbus, file, email, database table or GE Proficy Historian) is selected in the next dropdown.

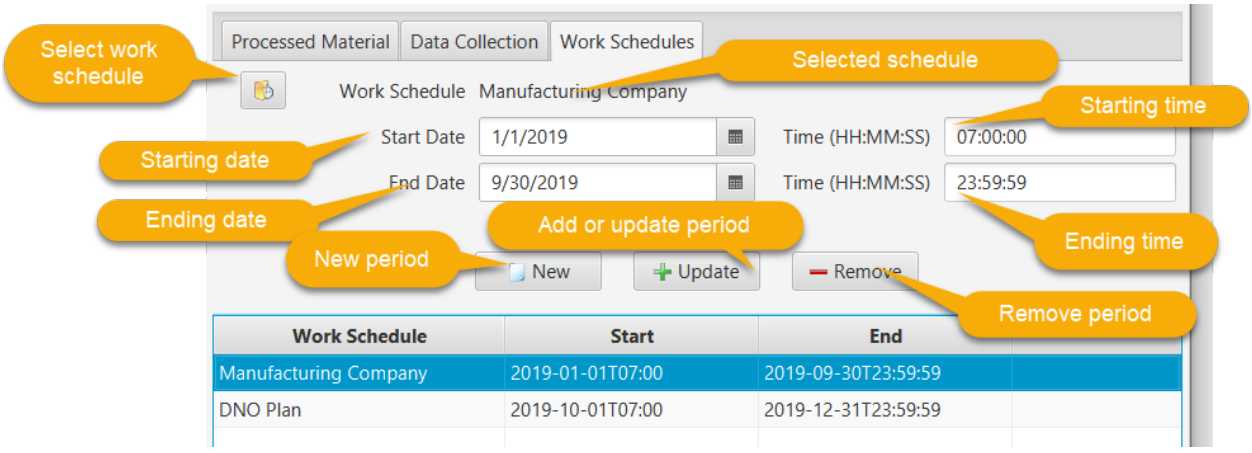
The data source editor is launched by clicking on the button next to the source id field. For an OPC DA source, a tag is selected in the browser. For OPC UA a node is selected. For HTTP, the HTTP data collector’s embedded server is selected. For an RMQ messaging source, the RabbitMQ broker is selected. Similarly, for a JMS source, the JMS broker is selected and for an MQTT source, the MQTT server. For a Kafka source, the Kafka server is selected. Similarly for an email source, the email server is selected. For Proficy, a Proficy Historian is selected. For a web socket source a web socket server is seleted.

For the HTTP, RMQ, JMS, MQTT, Kafka, web socket, file, email and database table sources, a default source id will be automatically created, but can be changed as long as it is unique system-wide. For OPC DA the source id is a tag identifier and for OPC UA a node identifier. For Modbus the source id is an endpoint and cannot be changed. For a Proficy Historian, the source id is the tag name.

Entity Work Schedules

One or more work schedules can be assigned to a plant entity. Each work schedule has beginning and ending effectivity times. The schedules must not overlap so that at any given point in time, there is only one current work schedule.

In this example, the “Manufacturing Company” schedule is effective from January 1 through September 30th. The “DNO Plan” is effective for the rest of the year.

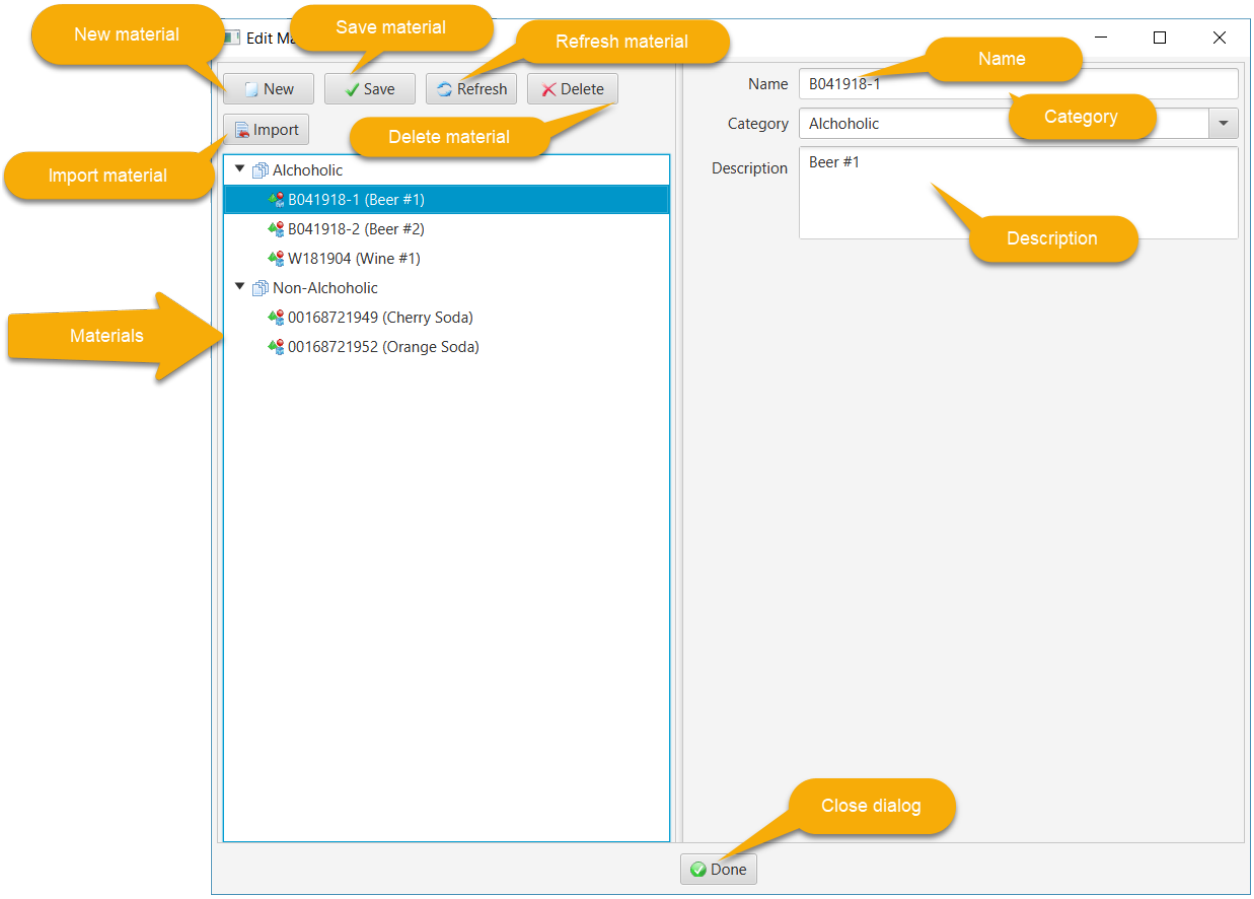


The editor actions are:

- *New*: clear the editor controls in order to define a new effective work schedule. The Add/Update button text will change to “Add”.
- *Add*: After defining the properties of a new effective work schedule, clicking this button will add it to the list of schedules
- *Update*: after selecting an existing effective work schedule the current values will be moved into the editing controls. Make the necessary edits, then click this button to apply the changes to the schedule.
- *Remove*: after selecting an existing effective work schedule, click this button to remove it from the list.

Material Editor

Materials produced by equipment are defined in this editor. For example:



The material editor buttons are:

- *New*: clear the editor to begin defining a new material
- *Save*: save the selected material to the database.

- *Refresh*: refresh the selected material from the database to synchronize the editor with the material's saved state
- *Delete*: delete the selected material from the database
- *Import*: Import materials from a comma-separated value (CSV) file
- *Backup*: export the selected material (or all materials if none is selected) to the specified .p85x file

The material editor has a context menu accessed by right-clicking in the left-hand pane. The menu items are:

- *Save All Material*: save all materials to the database
- *Refresh All Material*: restore all materials from their state in the database

The material category provides a convenient way to organize different material.

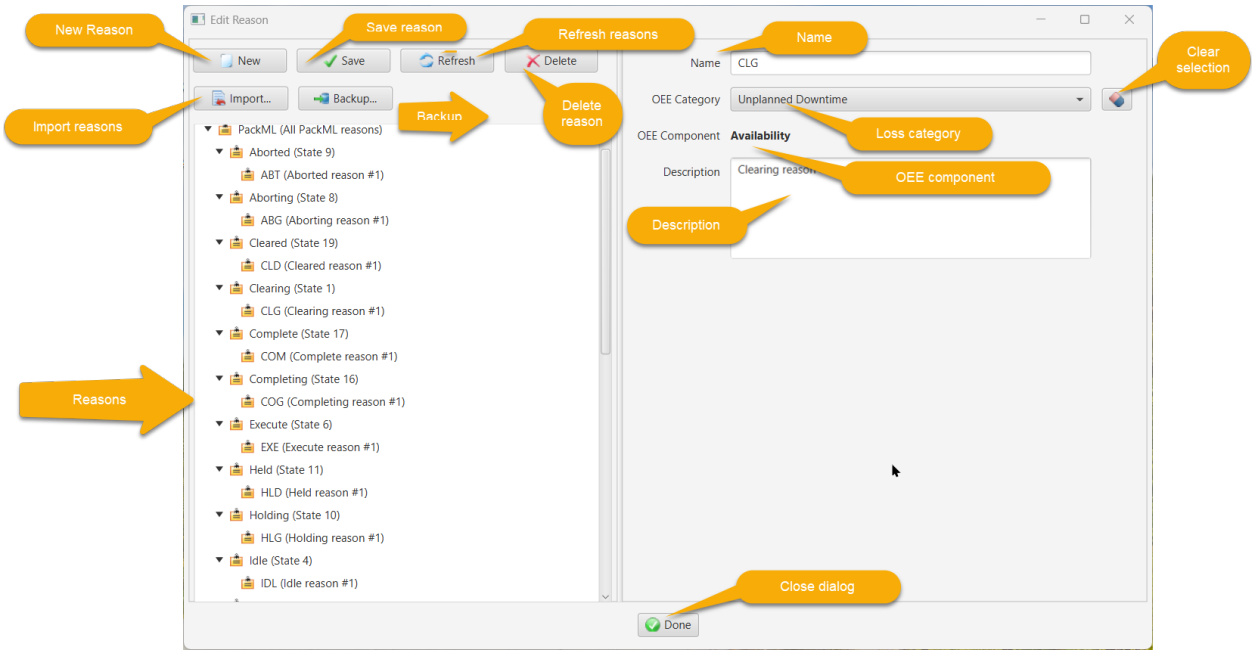
Clicking the Import button launches a dialog to choose a text file with a comma-separated lines for each material. The format is "name, description, category". The "materials.csv" file is included in the project as a example. For example:

```
B041918-1, Beer #1, Alcoholic
B041918-2, Beer #2, Alcoholic
W181904, Wine #1, Alcoholic
00168721952, Orange Soda, Non-Alcoholic
00168721949, Cherry Soda, Non-Alcoholic
```

REASON EDITOR

Description

Availability and performance reasons are defined in this editor. For example, this screen shot shows the PackML reason hierarchy:



The reason editor buttons are:

- *New*: clear the editor to begin defining a new reason
- *Save*: save the selected reason to the database.
- *Refresh*: refresh the selected reason from the database to synchronize the editor with the reason's saved state
- *Delete*: delete the selected reason from the database
- *Import*: Import reasons from a file
- *Backup*: save the selected reason (or all reasons if none is selected) to a .p85x backup file for later restoration

When saving a reason, if a parent reason is selected after creating the new reason, you will be asked to confirm that the new reason is a child of this parent. This hierarchy provides a way to organize the reasons. If a reason is used to determine an availability or performance event, it must have a loss category. Any reason with a loss category can be used in an OEE availability event.

The reason editor has a context menu accessed by right-clicking in the left-hand pane. The menu items are:

- *Save All Reasons*: save all reasons to the database
- *Refresh All Reasons*: restore all reasons from their state in the database
- *Clear Selected Reason*: unselect the selected reason so that no reason is selected

Clicking the Import button launches a dialog to choose a text file with comma-separated lines for each reason. The "reasons.csv" file is included in the project as a example. The format is "name, description, loss category, parent reason". For example:

```
Running, Normal production state, NO_LOSS
Setups, Setup reasons, ,
Setup1, Setup reason #1, SETUP, Setups
Setup2, Setup reason #2, SETUP, Setups
Planned Downtimes, Planned downtime reasons, ,
Planned1, Downtime reason #1, PLANNED_DOWNTIME, Planned Downtimes
Planned2, Downtime reason #2, PLANNED_DOWNTIME, Planned Downtimes
Unplanned Downtimes, Unplanned downtime reasons, ,
Unplanned1, Unplanned downtime reason #1, UNPLANNED_DOWNTIME, Unplanned Downtimes
Unplanned2, Unplanned downtime  reason #2, UNPLANNED_DOWNTIME, Unplanned Downtimes
Short Stops, Minor stoppage reasons, ,
Minor1, Minor stoppage reason #1, MINOR_STOPPAGES, Short Stops
Minor2, Minor stoppage reason #2, MINOR_STOPPAGES, Short Stops
```

The loss category name must match the TimeLoss.java enum name:

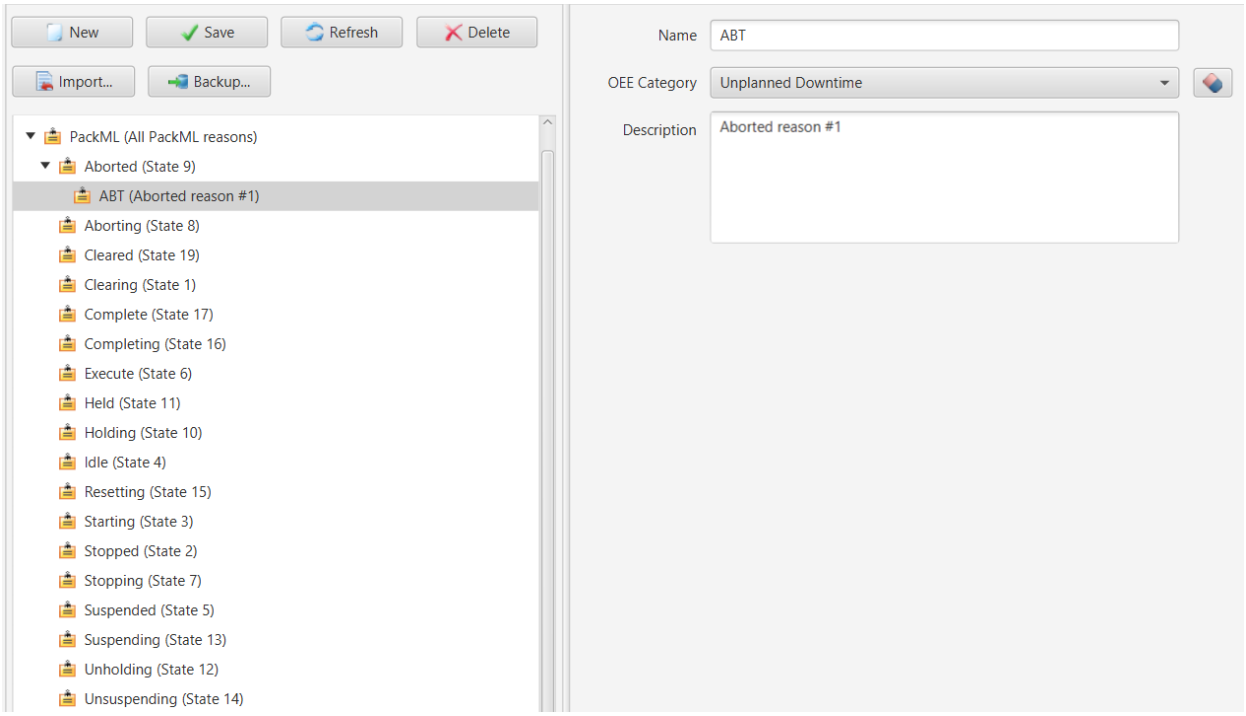
- NO_LOSS
- NOT_SCHEDULED
- UNSCHEDULED
- PLANNED_DOWNTIME
- SETUP
- UNPLANNED_DOWNTIME
- MINOR_STOPPAGES
- REDUCED_SPEED
- REJECT_REWORK
- STARTUP_YIELD

When selecting a loss category, the OEE component (performance, availability, quality, non-working or value adding) is displayed beneath it.

PackML

The OPC UA companion specification for the PackML state machine defines states of the equipment. When a state transition is caused, a reason needs to be provided to the resolver availability script. By convention, the availability reason belongs to a "state" reason which in turn has the "PackML Reasons" parent. The availability reason's time loss category determines which OEE loss the time in this state should be assigned to.

In the /config/database/demo folder is the PackMLReasons.p85x file. This file can be restored to provide a starting point to define the mapping between OEE time losses and the PackML states. For example:



When the equipment state transitions to Aborted, with the “ABT” reason, the time in this state will be assigned to the Unplanned Downtime loss category.

Unit of Measure (UOM)

Description

Good, reject/rework and setup/yield production quantities must have a unit of measure. The equipment’s design speed (a.k.a. ideal run rate) must be a quotient UOM, i.e. rate. The reject/rework units must also be specified. These units of measure are used in OEE calculations when the input and output UOMs differ. For example, a case packer might accept “can” as the input and reject UOM, but output a “case” of 12 cans.

A measurement system is a collection of units of measure where each pair has a linear relationship, i.e. y = ax + b where 'x' is the abscissa unit to be converted, 'y' (the ordinate) is the converted unit, 'a' is the scaling factor and 'b' is the offset. In the absence of a defined conversion, a unit will always have a conversion to itself where a = 1 and b = 0. A bridge unit conversion is defined to convert between the fundamental SI and International customary units of mass (i.e. kilogram to pound mass), length (i.e. metre to foot) and temperature (i.e. Kelvin to Rankine). These three bridge conversions permit unit of measure conversions between the two systems. A custom unit can define any bridge conversion such as a bottle to US fluid ounces or litres if needed.

A simple unit, for example a metre, is defined as a scalar UOM. A special scalar unit of measure is unity or dimensionless "1".

A unit of measure that is the product of two other units is defined as a product UOM. An example is a Joule which is a Newton·metre.

A unit of measure that is the quotient of two other units is defined as a quotient UOM. An example is a velocity, e.g. metre/second.

A unit of measure that has an exponent on a base unit is defined as a power UOM. An example is area in metre^2. Note that an exponent of 0 is unity, and an exponent of 1 is the base (root) unit itself. An exponent of 2 is a product unit where the multiplier and multiplicand are the base unit. A power of -1 is a quotient unit of measure where the dividend is 1 and the divisor is the base unit.

Units are classified by type, e.g. length, mass, time and temperature. Only units of the same type can be converted to one another. System pre-defined units of measure are also enumerated, e.g. kilogram, Newton, metre, etc. Custom units (e.g. a 1 litre bottle) do not have a pre-defined type or enumeration and are referred to by a unique base symbol.

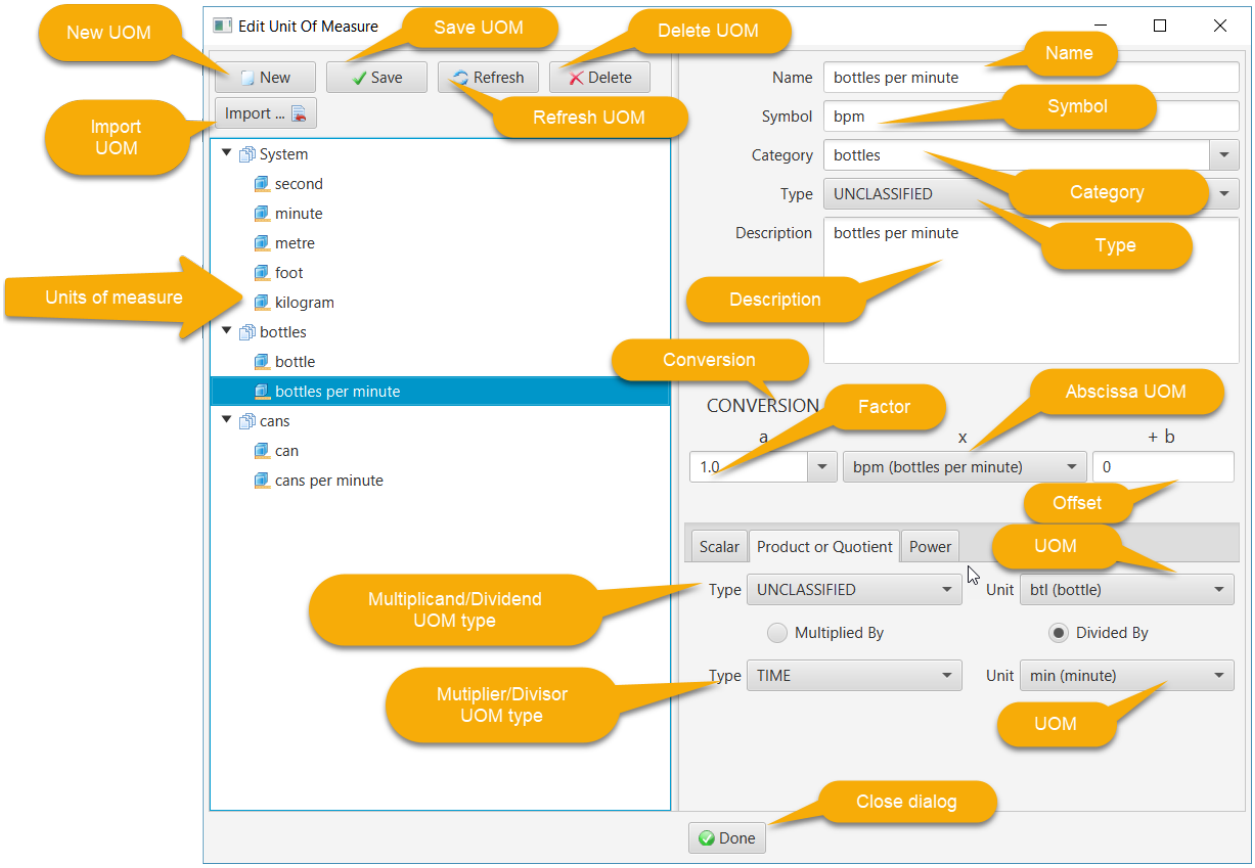
All units have a base symbol that is the most reduced form of the unit. For example, a Newton is kilogram·metre/second^2. The base symbol is used in the measurement system to register each unit and to discern the result of arithmetic operations on quantities. For example, dividing a quantity of Newton·metres by a quantity of metres results in a quantity of Newtons.

A quantity is an decimal amount together with a unit of measure. When arithmetic operations are performed on quantities, the original units can be transformed. For example, multiplying a length quantity in metres by a force quantity in Newtons results in a quantity of energy in Joules (or Newton-metres).

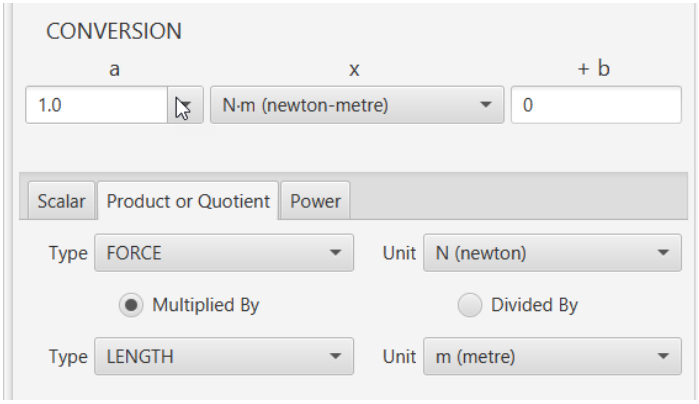
The unit of measure code is available as a standalone Java project at https://github.com/point85/Caliper and as a C# project at https://github.com/point85/CaliperSharp. More information about unit of measure capabilities along with examples can be found at the Caliper web sites.

Editor

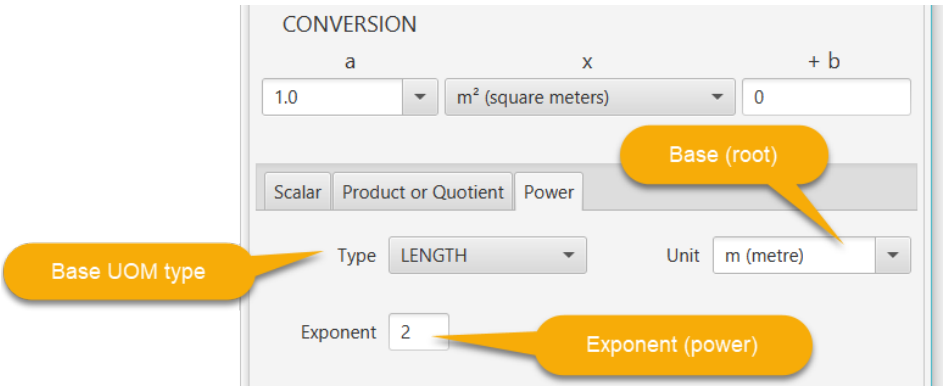
Units of measure are defined in the editor shown below. In this example, a quotient (rate) UOM “bottles per minute” has been created. The dividend is a scalar unit of “bottle” that was previously created. The divisor is a system time unit of “minute”.



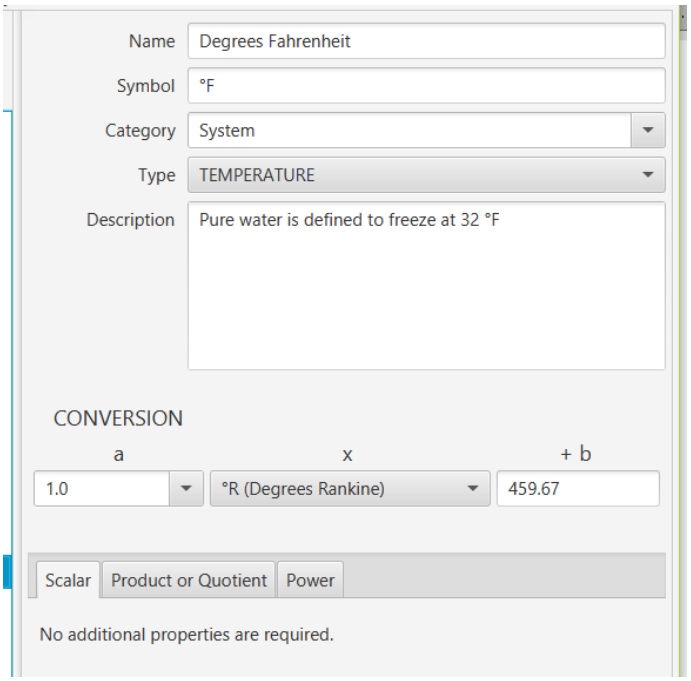
The UOM of measure editor is a general purpose editor and thus supports creation of quotient, product and power units as well as scalar ones (e.g. bottle, can, minute). For example, a Newton-metre is a system-defined product UOM created by importing it (as described below):



As a second example, square metres is a power UOM, again created by importing it:



Temperature in Fahrenheit is an example where the unit has a defined offset from Rankine units:



The UOM editor buttons are:

- *New*: clear the editor to begin defining a new UOM
- *Save*: save the selected UOM to the database.
- *Refresh*: refresh the selected UOM from the database to synchronize the editor with the UOM's saved state
- *Delete*: delete the selected UOM from the database
- *Import*: Import a system UOM from pre-defined choices
- *Backup*: save the selected UOM (or all UOMs if none is selected) to a .p85x backup file for later restoration

When creating a UOM, a category needs to be specified. The category provides a way to group related units. The “System” category is reserved for the pre-defined UOMs such as metre.

A UOM must also have a type. All units with the same type can be converted between each other. For packaging units like “can” or “bottle” the UNCLASSIFIED type should be chosen. Whether or not a conversion between units in this category makes sense is determined by application logic, since there is no physical basis for such units.

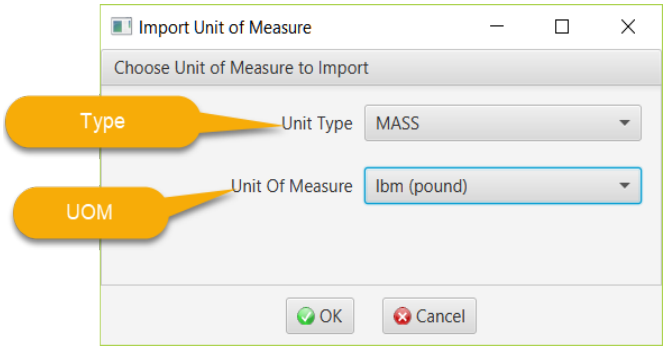
A large number of system-defined types and the corresponding UOMs are included:

- *dimension-less "1"*: UNITY
- *fundamental physical*: LENGTH, MASS, TIME, ELECTRIC_CURRENT, TEMPERATURE, SUBSTANCE_AMOUNT, LUMINOSITY
- *other physical*: AREA, VOLUME, DENSITY, VELOCITY, VOLUMETRIC_FLOW, MASS_FLOW, FREQUENCY, ACCELERATION, FORCE, PRESSURE, ENERGY, POWER, ELECTRIC_CHARGE, ELECTROMOTIVE_FORCE, ELECTRIC_RESISTANCE, ELECTRIC_CAPACITANCE, ELECTRIC_PERMITTIVITY, ELECTRIC_FIELD_STRENGTH, MAGNETIC_FLUX, MAGNETIC_FLUX_DENSITY, ELECTRIC_INDUCTANCE, ELECTRIC_CONDUCTANCE, LUMINOUS_FLUX, ILLUMINANCE, RADIATION_DOSE_ABSORBED, RADIATION_DOSE_EFFECTIVE, RADIATION_DOSE_RATE, RADIOACTIVITY, CATALYTIC_ACTIVITY, DYNAMIC_VISCOSITY, KINEMATIC_VISCOSITY, RECIPROCAL_LENGTH, PLANE_ANGLE, SOLID_ANGLE, INTENSITY, TIME_SQUARED, MOLAR_CONCENTRATION, IRRADIANCE
- *computer science*
- *currency*

The UOM editor has a context menu accessed by right-clicking in the left-hand pane. The menu items are:

- *Save All Units of Measure*: save all UOMs to the database
- *Refresh All Units of Measure*: restore all UOMs from their state in the database

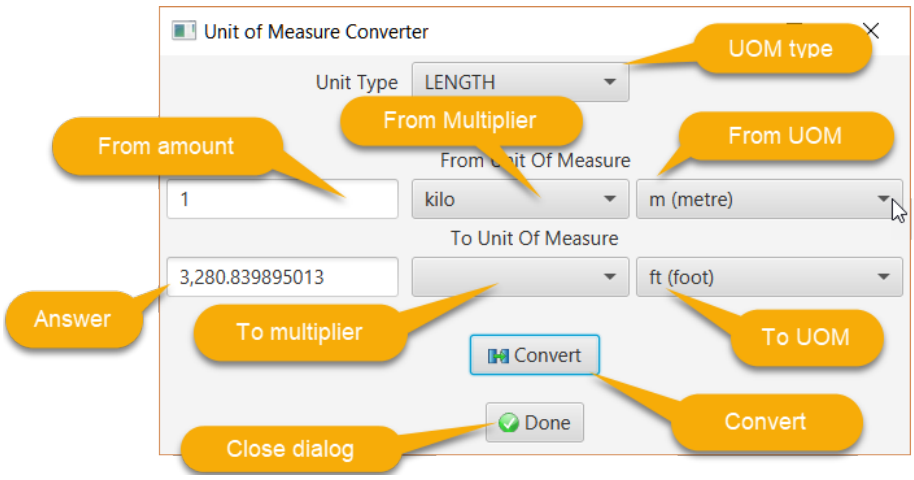
Clicking the Import button launches a dialog to choose the system-defined type and then the UOM of interest. For example to import the customary UOM of pound-mass:



Click “OK” to import the unit or “Cancel” to cancel out. Note that importing a UOM will import that UOM and all referenced units.

Converter

This utility is launched from the toolbar. It is used to convert from one UOM to another UOM of the same type (e.g. length to length). For example, 1 kilometre (1000 metre) is 3,280.8 feet:



First, choose the unit type (e.g. LENGTH), the enter the “from” amount (1), “from” factor if desired (kilo) and “from” unit (metre). Then, select the “to” factor (if any) and “to” unit (foot). Click the Convert button to display the answer of 3,280.8 feet.

Work Schedule

Description

The time equipment is scheduled for production is defined by a work schedule. The work schedule is attached to the equipment itself or to any node in the hierarchy above it. The search starts at the equipment with the availability event and moves up the hierarchy until a schedule is found. Therefore a work schedule could be defined for area or site and apply to all equipment below it.

A work schedule consists of one or more teams who rotate through a sequence of shift and off-shift periods of time. Breaks during shifts can be defined as well as non-working periods of time (e.g. holidays and scheduled maintenance periods) that are applicable to the entire work schedule. A break is defined as a working period of time for operators during a shift, for example lunch. For OEE calculations, a time loss can be associated with a break if the equipment is not scheduled for production during this period. Overtime and non-working periods (holidays) can also be defined in a similar way.

A work schedule has a name and description. Zero or more non-working periods can be defined. A non-working period has a defined starting date and time of day and duration. For example, the New Year's Day holiday starting at midnight for 24 hours, or three consecutive days for preventive maintenance of manufacturing equipment starting at the end of the night shift. Similarly, overtime periods can be defined such as for working a seasonal schedule.

A shift is defined with a name, description, starting time of day and duration. An off-shift period is associated with a shift. Shifts can be overlapped (typically when a hand-off of duties is important). A rotation is a sequence of shifts and off-shift days. An instance of a shift has a starting date and time of day and has an associated shift definition.

A shift can have zero or more break periods. A break period has a name, description, starting time of day, duration and optional OEE time loss category. If no loss category is assigned, the break is assumed to be scheduled productive time.

A team/crew is defined with a name and description. It has a rotation with a starting date. The starting date shift will have an instance with that date and a starting time of day as defined by the shift. The same rotation can be shared between more than one team, but with different starting times.

A rotation is a sequence of working periods (segments). Each segment starts with a shift and specifies the number of days on-shift and off-shift. A work schedule can have more than one rotation.

A non-working period is a duration of time where no production teams are working. For example, a holiday or a period of time when a plant is shutdown for preventative maintenance. A non-working period starts at a defined day and time of day and continues for the specified duration of time. An overtime period is a duration of time where a production team is working outside of scheduled shifts.

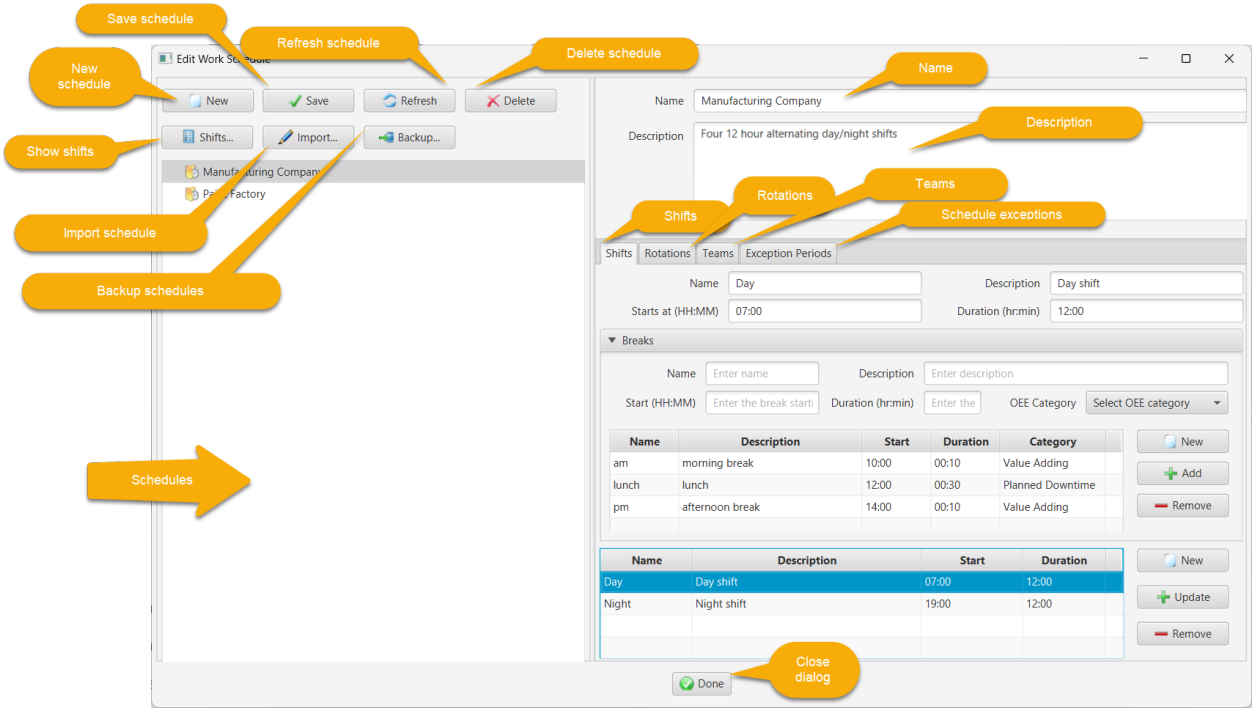
A shift instance is the duration of time from a specified date and time of day and continues for the duration of the associated shift. A team works this shift instance.

After a work schedule is defined, the working time for all shifts can be computed for a defined time interval. This duration of time is the maximum available productive time and is the input to the calculation of Overall Equipment Effectiveness (OEE). Time accumulated in the various loss categories subtracts from this total time to finally arrive at the value adding time. The shift when an OEE event occurs will be also recorded in the database.

The work schedule code is available as a standalone Java project at https://github.com/point85/Shift and as a C# project at https://github.com/point85/ShiftSharp. More information about work schedule capabilities along with examples can be found at the Shift web sites.

Editor

For the work schedule editor shown below, the Manufacturing Company schedule has been selected:



The work schedule editor buttons are:

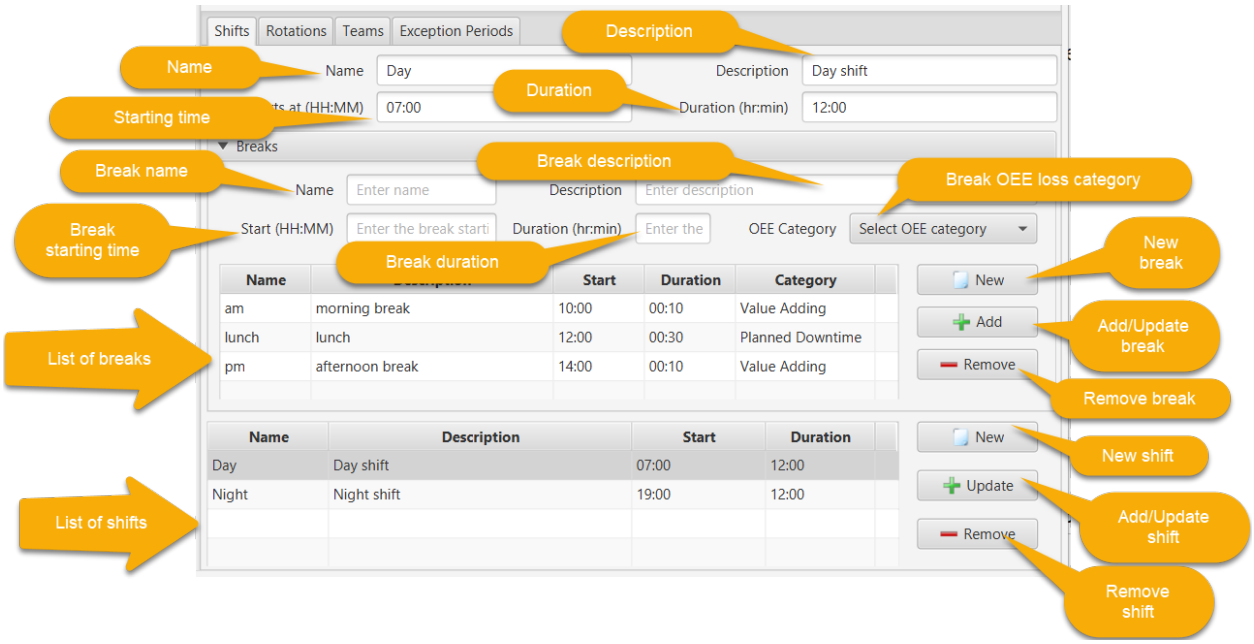
- *New*: clear the editor to begin defining a new schedule
- *Save*: save the selected schedule to the database.
- *Refresh*: refresh the selected schedule from the database to synchronize the editor with the schedule's saved state
- *Delete*: delete the selected schedule from the database
- *Shifts*: display a dialog to show the shift instances for a specfied period of time
- *Import*: import a schedule from pre-defined templates
- *Backup*: save the selected work schedule (or all work schedules if non is selected) to a .p85x file

The work schedule editor has a context menu accessed by right-clicking in the left-hand pane. The menu items are:

- *Save All Schedules*: save all schedules to the database
- *Refresh All Schedules*: restore all schedules from their state in the database
- *Clear Selected Schedule:* unselect the selected work schedule

Shifts

The "Shifts" tab is used to define shifts and breaks. The break editor appears after expanding its pane and then choosing a shift:



The editor actions for a shift are:

- *New*: clear the editor controls in order to define a new shift. The Add/Update button text will change to "Add".
- *Add*: After defining the properties of a new shift, clicking this button will add it to the list of shifts
- *Update*: after selecting an existing shift, the current values will be moved into the editing controls. Make the necessary edits, then click this button to apply the changes to the shift list.
- *Remove*: after selecting an existing shift, click this button to remove it from the list.

The starting time of day is entered in 24-hour format (hours:minutes from 00:00 to 23:59). The shift duration is entered as hours:minutes between 00:00 and 24:00. Note that after making changes to the list of shifts, the work schedule must be saved to the database by clicking the Save button.

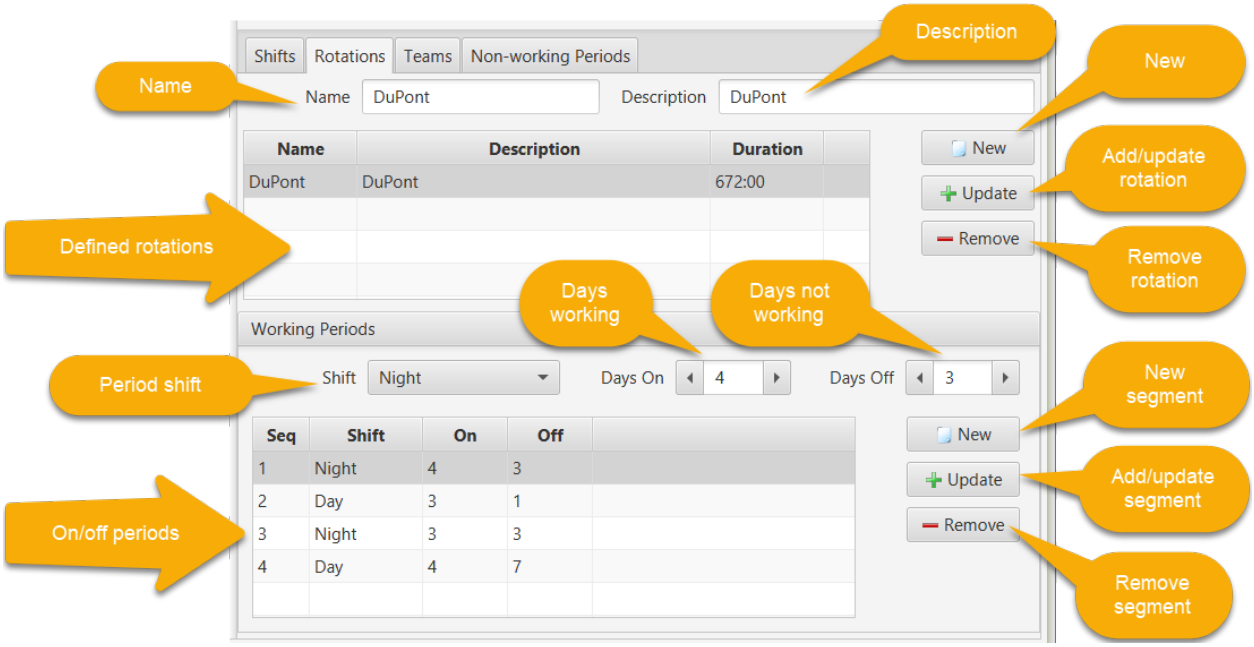
The editor actions for a break are:

- *New*: clear the editor controls in order to define a new break. The Add/Update button text will change to "Add".
- *Add*: After defining the properties of a new break, clicking this button will add it to the list of breaks
- *Update*: after selecting an existing break, the current values will be moved into the editing controls. Make the necessary edits, then click this button to apply the changes to the break list.
- *Remove*: after selecting an existing break, click this button to remove it from the list.

The starting time of day is entered in 24-hour format (hours:minutes from 00:00 to 23:59). The break duration is entered as hours:minutes between 00:00 and 24:00. Note that after making changes to the list of breaks, the work schedule must be saved to the database by clicking the Save button.

Rotations

The "Rotations" tab is used to define rotations:



In this example, the DuPont 12-hour rotating shift schedule uses 4 teams (crews) and 2 twelve-hour shifts to provide 24/7 coverage. It consists of a 4-week cycle where each team works 4 consecutive night shifts, followed by 3 days off duty, works 3 consecutive day shifts, followed by 1 day off duty, works 3 consecutive night shifts, followed by 3 days off duty, work 4 consecutive day shift, then have 7 consecutive days off duty. Personnel works an average 42 hours per week.

This DuPont example has one long rotation of 672 hours (28 days or 4 weeks) consisting of 4 segments in this sequence:

1. Night shift, 4 days on followed by 3 days off
2. Day shift, 3 days on followed by 1 day off
3. Night shift, 3 days on followed by 3 days off
4. Day shift, 4 days on followed by 7 days off

The editor actions for rotations are:

- *New*: clear the editor controls in order to define a new rotation. The Add/Update button text will change to "Add".
- *Add*: After defining the properties of a new rotation, clicking this button will add it to the list of rotations

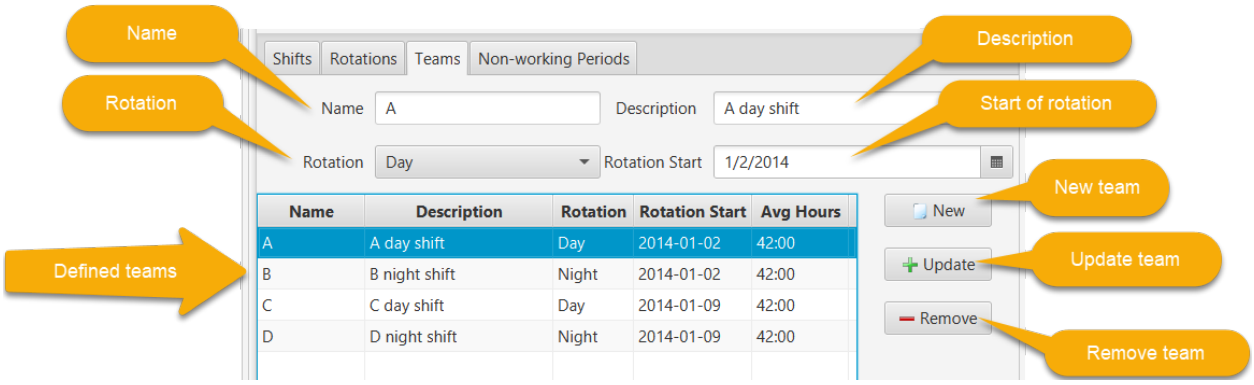
- *Update*: after selecting an existing rotation, the current values will be moved into the editing controls. Make the necessary edits, then click this button to apply the changes to the list.
- *Remove*: after selecting an existing rotation, click this button to remove it from the list.

The editor actions for rotation segments are:

- *New*: clear the editor controls in order to define a new rotation segment. The Add/Update button text will change to "Add".
- *Add*: After defining the properties of a new rotation segment, clicking this button will add it to the list of rotation segments
- *Update*: after selecting an existing rotation segment, the current values will be moved into the editing controls. Make the necessary edits, then click this button to apply the changes to the list.
- *Remove*: after selecting an existing rotation segment, click this button to remove it from the list.

Teams

The "Teams" tab is used to define teams (crews):

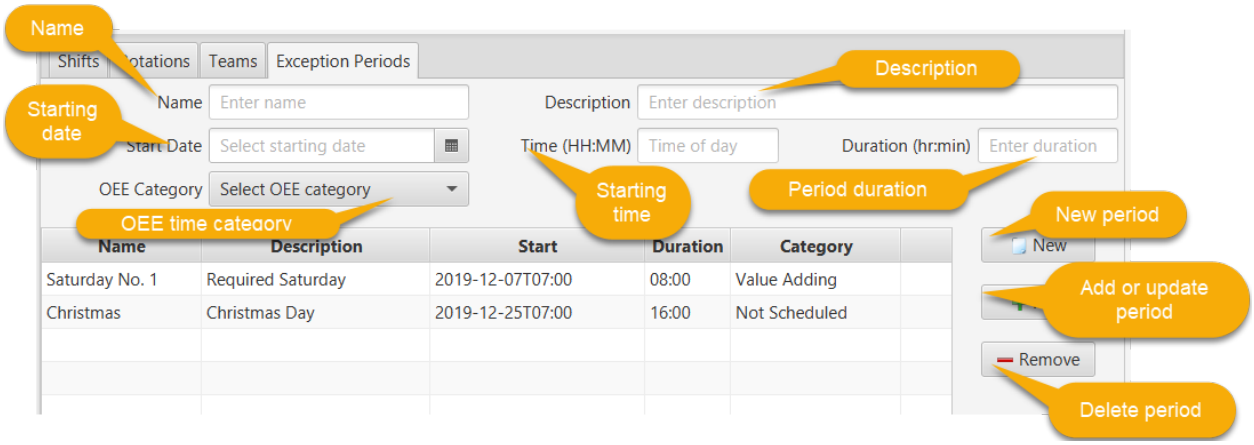


The editor actions for teams are:

- *New*: clear the editor controls in order to define a new team. The Add/Update button text will change to "Add".
- *Add*: After defining the properties of a new team, clicking this button will add it to the list of teams. The average working hours will be displayed in the last column, e.g. 42 hours and 0 minutes in this example.
- *Update*: after selecting an existing team, the current values will be moved into the editing controls. Make the necessary edits, then click this button to apply the changes to the list.
- *Remove*: after selecting an existing team, click this button to remove it from the list.

Exception Periods

The "Exception Periods" tab is used to define intervals of time on an exception basis where no production will take place, for example holidays and planned maintenance outages. It is also used to define time periods where production will take place outside of regular shifts, for example working a few Saturdays.

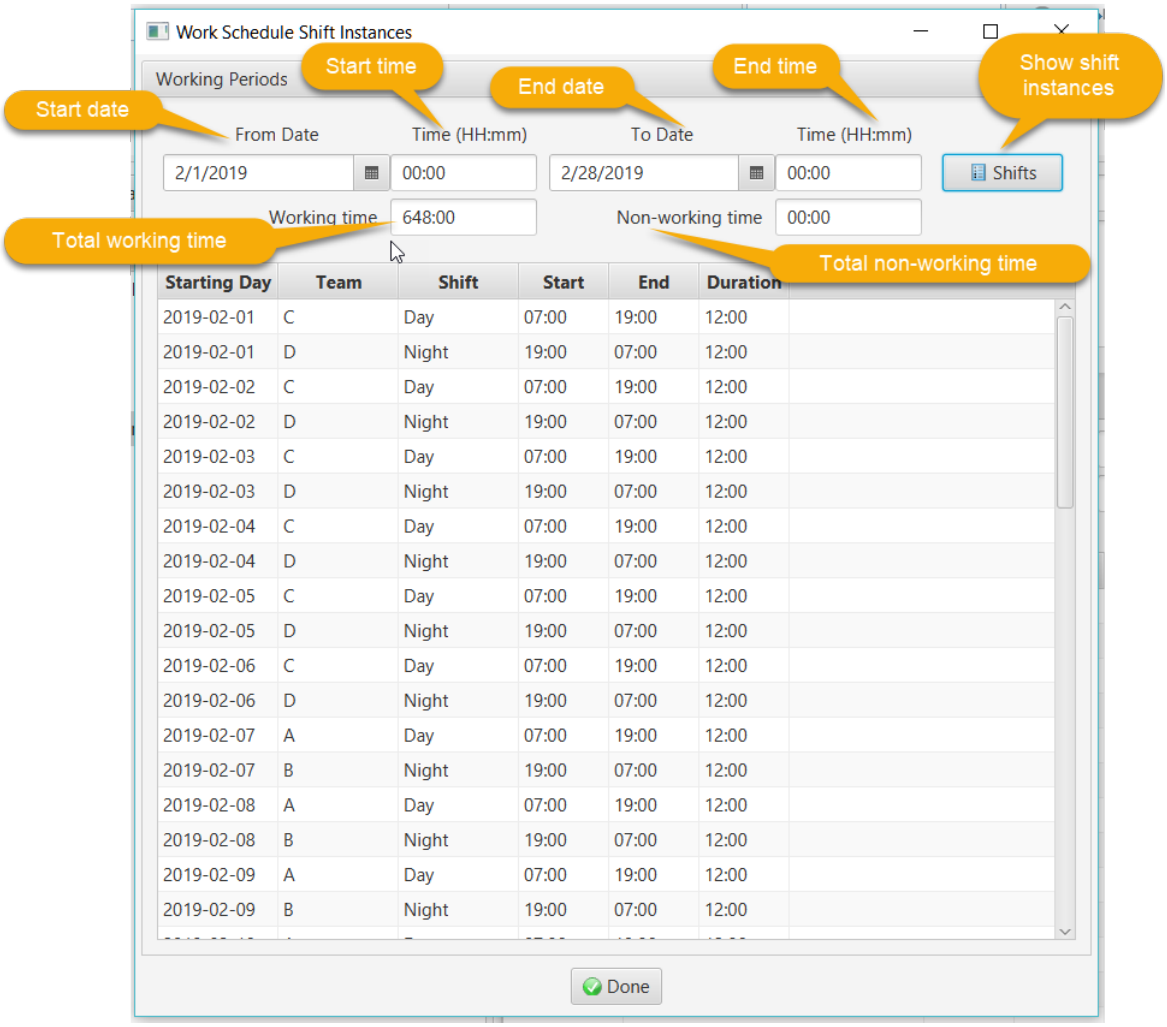


The editor actions for teams are:

- *New*: clear the editor controls in order to define a new exception period. The Add/Update button text will change to "Add".
- *Add*: After defining the properties of a new exception period, clicking this button will add it to the list of periods. The loss category is one of two choices (1) Not Scheduled (e.g. holiday), or (2) Unscheduled (special event). An overtime period is "Value Adding".
- *Update*: after selecting an existing exception period, the current values will be moved into the editing controls. Make the necessary edits, then click this button to apply the changes to the list.
- *Remove*: after selecting an existing exception period, click this button to remove it from the list.

Show Shift Instances

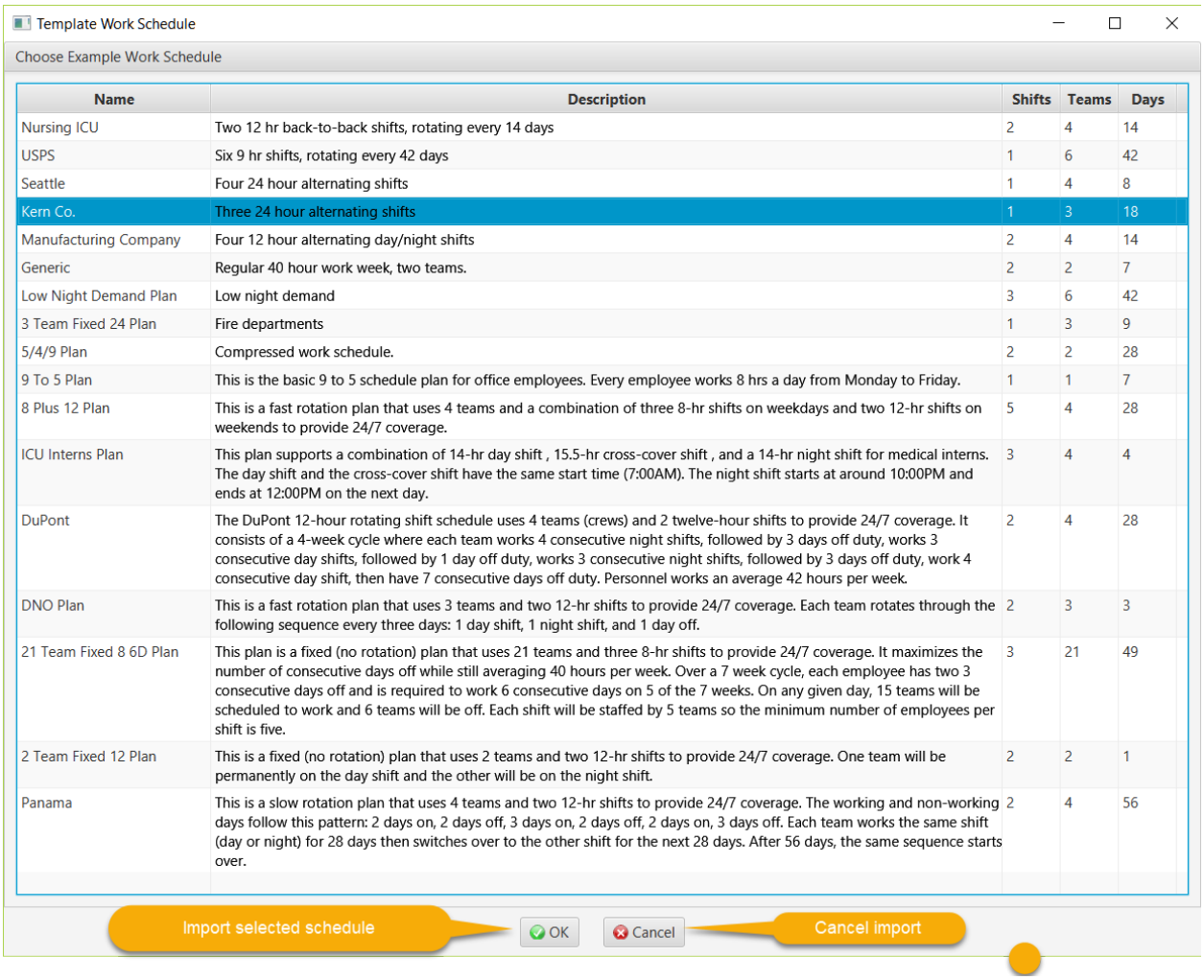
Click the Shifts button to launch a dialog to view shift instances for the selected work schedule. For example, the dialog below shows shift instances for the month of February for the Manufacturing Company work schedule:



Note that if a value-adding exception period is defined, the working time total will be more than the scheduled shift time.

Import Schedule

Click the Import button to launch a dialog to choose a pre-defined work schedule (similar to a desired one) and then save it to the database for further editing:



SCRIPTING

Description

A JavaScript function must defined for each equipment resolver. This script is executed by the OpenJDK Nashorn script engine. It accepts an input value from a data source event and returns a value matching the type of the resolver (e.g. availability, production count, material or job change). For the case of a CUSTOM event, the output value (if any) is used only for display in the trend chart. The script editor is used to write and test the body of this function. The script can call any OEE method accessible through the OeeContext object or any java code in external jar files.

An availability script outputs the name of a Reason (which is associated with an OEE loss category). A good, reject or startup production script outputs a count in the unit of measure defined for that equipment. A good production amount has the dividend UOM of the design speed, whereas a reject or startup production amount has the defined reject UOM. A material change script outputs the name of a defined material. A job change script outputs the name of a job/order. In order to perform OEE calculations, the material must be defined as the first event before the time period of interest.

The script has three input arguments:

1. OeeContext *context*: an instance of the OeeContext class containing information about the script execution environment (see below for details)
2. Object *value*: the input value. The data in this input depends on the event source and data type. For OPC DA and UA the value can be a scalar or array.
3. EventResolver *resolver*: the event resolver. The resolver's last value can be used in rollover calculations for production counts. The equipment for the executing script is also available as a resolver attribute.

The OeeContext class has these primary public "getter" methods (note that domain javadocs are available in the "docs" folder in the domain_docs.zip file):

- *getLogger()*: Get an instance of an org.slf4j.Logger. The logging output(s) is configured in the log4j2.xml file.
- *getMaterial(Equipment equipment)*: Get the material currently being processed on this equipment. The equipment object is obtained from resolver.getEquipment().
- *getJob(Equipment equipment)*: Get the job currently being run on this equipment. The equipment object is obtained from resolver.getEquipment().
- *getOpcDaClients()*: Get the collection of DaOpcClient objects. A client object can then be used for reading and writing tags, e.g. readSynch()/writeSynch().
- *getOpcUaClients()*: Get the collection of UaOpcClient objects. A client object can then be used for reading and writing nodes, e.g. readSynch()/writeSynch().
- *getMessagingClient()*s: Get the collection of MessagingClient objects. A client object can then be used to send a message to the associated server, for example "Point85" exchange on a RabbitMQ broker, e.g. publish().
- *getJMSClient()*s: Get the collection of JMSClient objects. A client object can then be used to send a message to the JMS broker.
- *getMQTTClient()*s: Get the collection of MQTTOeeClient objects. A client object can then be used to send a message to the MQTT server.
- *getKafkaClient()*s: Get the collection of KafkaOeeClient objects. A client object can then be used to send a message to the Kafka broker.
- *getWebSocketServer()*s: Get the collection of WebSocketOeeServer objects.
- *getEmailClient()*s: Get the collection of EmailClient objects. A client object can then be used to send a message to the email server.
- *getHttpServer()*s: Get the collection of HTTP servers.
- *getDatabaseEventClient()*s: Get the collection of DatabaseEventClients.

- *getFileEventClient()*s: Get the collection of FileEventClients.
- getModbusMasters(): Get the collection of ModbusMasters.
- *getCronEventClient()*s: Get the collection of CronEventClients.
- *getProficyClient()*s: Get the collection of ProficyClients.
- get/setProperty(): A hashmap for caching user-defined values during script execution

The JavaScript below in the Custom Scripting section shows examples of calling these methods.

The default script for availability, material and job is to simply return the input value (passthrough):

```
return value;
```

However, the default script for a production count provides for a rollover of the counting sensor:

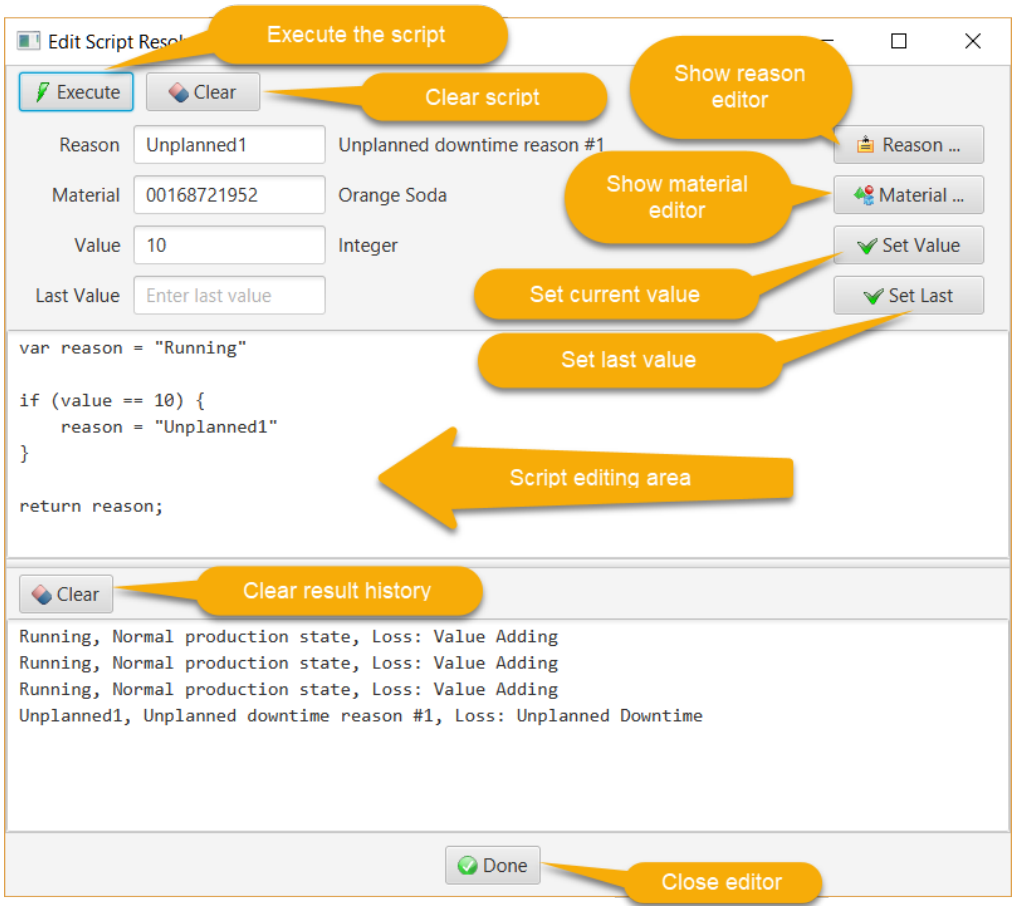
```
var ROLLOVER = 0;
var lastValue = resolver.getLastValue();
var delta = value - lastValue;
if (value < lastValue) {
    delta += ROLLOVER;
}
return delta;
```

The variable “ROLLOVER” must be defined for the sensor. If the last value of the count is greater than the current value, then the counter must have rolled over at the ROLLOVER value to zero in this example.

Editor

The script editor dialog is launched either by (1) selecting an equipment resolver in the list in the “Data Collection” tab of the equipment entity and then clicking the editor button or (2) clicking a button on the toolbar. If the editor is launched from the toolbar, functionality will be limited since the resolver input argument to the script will be null.

The editor looks like:



The editor actions are:

- *Execute*: execute the script. The returned object's toString() method will be called and the result displayed in the execution history in the bottom pane.
- *Clear*: clear the script editor or history
- *Reason*...: Display the reason editor to create or update a reason and choose an existing reason. The name is displayed in the text field where it is available for cutting and pasting into the script.
- *Material*...: Display the material editor to create or update a material and choose an existing material. The name is displayed in the text field where it is available for cutting and pasting into the script.
- *Set Value*: Set the value in the text field as the input value to the script before it is executed.
- *Set Last*: Set the value in the text field as the previous input value to the script before it is executed.

Depending upon the type of resolver, the script area initially will be populated by a default script. Availability, material change and job change scripts pass the input value through to the output:

```
return value;
```

whereas a production count script provides a variable called "ROLLOVER" as a place-holder to take into account the case where a counter can output a lower value than a previous value:

```
var ROLLOVER = 0;
var lastValue = resolver.getLastValue();
var delta = value - lastValue;
if (value < lastValue) {
    delta += ROLLOVER;
}
return delta;
```

External Java Code

To call custom Java code from script, the jar(s) are placed in the lib/ext folder for the JavaFX desktop applications since it is on the class path. The jar(s) are placed in the WEB-INF/lib folder for the Vaadin operator application. For an example, see Example #11 below.

Custom Scripting

A resolver script can be used for general purpose (non-OEE) data collection by choosing the "CUSTOM" type. In this case, the input value to the script is used in the script logic as required by a custom application. The output value (if any) is displayed in the trend chart.

Example 1 - Logging

This example show how to log to the Point85.log file.

```
// logger
var logger = context.getLogger()
// equipment
var eq = resolver.getEquipment()

// log info
logger.info("Material: " + context.getMaterial(eq))
logger.info("Job: " + context.getJob(eq))
logger.info("Source id: " + resolver.getSourceId())
logger.info("Last value: " + resolver.getLastValue())
logger.info("Last timestamp: " + resolver.getTimestamp())
logger.info(context.toString())
```

Example 2 - Database

For this example, suppose that a custom database table (TEST_CUSTOM) has been created with three columns (string, integer and float values). When the input string (value) is received, a record is inserted into this table by making use of the PersistenceService singleton's executeUpdate() method:

```
var PersistenceService = Java.type('org.point85.domain.persistence.PersistenceService');
```

```
var sql = "insert into dbo.TEST_CUSTOM (A_STRING, AN_INT, A_FLOAT) values ('" + value + "', 1,
0.0)";

var result = PersistenceService.instance().executeUpdate(sql);
```

The executeQuery() method of PersistenceService returns a JSON list of all records in this table:

```
var PersistenceService = Java.type('org.point85.domain.persistence.PersistenceService');

var sql = "select * from dbo.TEST_CUSTOM";

var json = PersistenceService.instance().executeQuery(sql);

print(json);
```

A modfied version of this query returns all rows in the TEAM table as a List of Object[]:

```
var PersistenceService = Java.type('org.point85.domain.persistence.PersistenceService');
var em = PersistenceService.instance().getEntityManager();
var sql = "select * from dbo.TEAM";

// list of Object[] for each table row
var rowList = em.createNativeQuery(sql).getResultList();
em.close();

// get the team name for each team
for (var i = 0; i < rowList.size(); i++) {
  var columns = rowList.get(i);
  var name = columns[1];
}
```

Example 3 - Publish Message

```
// send RMQ message for a configured messaging data source

var CollectorNotificationMessage =
Java.type("org.point85.domain.messaging.CollectorNotificationMessage")

var routingKey = Java.type("org.point85.domain.rmq.RoutingKey").NOTIFICATION_MESSAGE

var severity = Java.type("org.point85.domain.messaging.NotificationSeverity").INFO

var msg = new CollectorNotificationMessage("localhost", "192.168.0.8")

msg.setText("This is a notification")

msg.setSeverity(severity)

context.getMessagingClient().publish(msg, routingKey, 30)
```

Example 4 - Send Alarm Notification

```
// send RMQ message to the collector's notification server.  This alarm will be displayed in the
monitor's "Collector Notifications" tab

var RmqClient = Java.type("org.point85.domain.rmq.RmqClient")

var client = new RmqClient()

var collector = resolver.getCollector()

var host = collector.getBrokerHost()

var port = collector.getBrokerPort()

var user = collector.getBrokerUserName()

var pwd = collector.getBrokerUserPassword()

var HIGH = 20000
```

```
var LOW = 10000
if (value > HIGH) {
    var severity = Java.type("org.point85.domain.messaging.NotificationSeverity").ERROR
    var text = "Alarm high level of " + HIGH + " exceeded.  Value is " + value
    client.connect(host, port, user, pwd)
    client.sendNotification(text, severity)
    client.disconnect()
}  else if (value < LOW) {
    var severity = Java.type("org.point85.domain.messaging.NotificationSeverity").WARNING
    var text = "Alarm low level of " + LOW + " exceeded.  Value is " + value
    client.connect(host, port, user, pwd)
    client.sendNotification(text, severity)
    client.disconnect()
}
```

Example 5 - Read/Write Values with OPC DA

```
// OPC DA read integer value
var logger = context.getLogger()
var variant = context.getOpcDaClient().readSynch("Random.Int4")
logger.info("Value: " +  variant.getValueAsNumber())

// OPC DA write integer value
var OpcDaVariant = Java.type("org.point85.domain.opc.da.OpcDaVariant")
var variant = new OpcDaVariant(100)
context.getOpcDaClient().writeSynch("Data Type Examples.16 Bit Device.K Registers.Short1",
variant)
```

Example 6 - Read/Write Values with OPC UA

```
// OPC UA read current server time
var logger = context.getLogger()
var NodeId = Java.type("org.eclipse.milo.opcua.stack.core.types.builtin.NodeId")
var nodeId = new NodeId(0, 2258)
var dataValue = context.getOpcUaClient().readSynch(nodeId)
logger.info(dataValue.getValue().getValue())

// OPC UA write integer value
var logger = context.getLogger()
var NodeId = Java.type("org.eclipse.milo.opcua.stack.core.types.builtin.NodeId")
```

```
var Variant = Java.type("org.eclipse.milo.opcua.stack.core.types.builtin.Variant")

var nodeId = new NodeId(3, "Int32DataItem")

var value = new Variant(100)

code = context.getOpcUaClient().writeSynch(nodeId, value)

if (code.isBad()) {

	logger.error("Write failed, code = " + code.getValue())

}
```

Example 7 - Database Event Table Query

```
// database query

var logger = context.getLogger()

var service = Java.type("org.point85.domain.persistence.PersistenceService").instance()

var sql = "Select top 10 EVENT_TYPE, JOB from OEE_EVENT order by START_TIME desc"

var rows = service.getEntityManager().createNativeQuery(sql).getResultList()

for (i = 0; i < rows.size(); i++) {

	var row = rows.get(i)

	logger.info("Type: " + row[0] + ", Job: " + row[1])

}
```

Example 8 - Set a Reject and Rework Reason

```
// set a reject and rework reason named "002" for this production event

resolver.setReason("002");

return value;
```

Example 9 - Read/Write Modbus Data Values

```
// 1.  In the resolver script, the value is a List<ModbusVariant> containing an Integer

return value.get(0).getNumber();


// 2.  Write a boolean to a coil at address 0

var master = context.getModbusMaster();

master.writeCoil(255, 0, false);


// 3.  Write a signed 16-bit short integer value to a holding register

var ModbusVariant = Java.type("org.point85.domain.modbus.ModbusVariant");

var ModbusDataType = Java.type("org.point85.domain.modbus.ModbusDataType");

var ModbusEndpoint = Java.type("org.point85.domain.modbus.ModbusEndpoint");

var Short = Java.type("java.lang.Short");
```

```
var variant = new ModbusVariant(ModbusDataType.INT16, new Short(1234));

var endpoint = new ModbusEndpoint();

endpoint.setUnitId(255);

endpoint.setRegisterAddress(0);

endpoint.setReverseEndianess(true);

context.getModbusMaster().writeHoldingRegister(endpoint, variant);

// 4.  Read a single value (in a list) for a reason (availability) code.  Only save an event
record if the value has changed.

var newValue = value.get(0).toString();

var returnValue = newValue;

if (resolver.getLastValue() != null) {

    lastValue = resolver.getLastValue().get(0).toString();

    if (newValue.equals(lastValue)) {

        returnValue = null;

    }

}

return returnValue;

// 5.  Read a single value (in a list) for a production amount. Only save an event record if the
amount is positive.

var variant = value.get(0);

var returnValue = null;

if (variant.isPositiveNumber()) {

    returnValue = variant.getNumber();

}

return returnValue;
```

Example 10 - Send Email or Text Alert

In this example, if the destination account “user@domain” is an email account, an email will be sent. If the account is in the form of 10_digit_telephone_number@domain, a text message will be sent instead to the telephone number. For example, for an SMS message and US Verizon provider, the format is 6501234567@vtext.com and for an MMS message it is 6501234567@vzwpix.com

```
var logger = context.getLogger();

// get the one and only email client

var emailClient = context.getEmailClient();

// send alert if rejects are above the high limit

if (parseInt(value) > 100) {

    emailClient.sendMail("user@domain", "High Limit Exceeded", "Reject production of " + value +
" exceeds high limit of 100.");

    logger.info("Sent email to " + emailClient.getSource().getSendHost());
```

```
}
return value;
```

Example 11 - Save Water Consumption Data

In this example, a company is piloting a program to monitor wash water consumption with a goal of reducing usage by 20%. The first phase is to store at 00:01:00 the previous day's hourly consumption for later analysis and trending after improvements are made to the process.

For this purpose, the washer's Flume water meter has an OAuth 2.0 protected API that is called by a Cron job triggered to fire at 00:01:00 after the meter has transmitted the latest data to its server.

A Java FlumeClient class has a public fetchConsumption() method to be called by JavaScript in a custom Cron resolver to read the 24 hourly values to be stored in the database. The Java code below is for illustrative purposes only and does not contain supporting class source code. The code is exported into a jar file in the lib/ext folder so that it is on the classpath.

```
public class FlumeClient {

	private static final String BASE_URL = "https://api.flumewater.com";

	private static final String TOKEN_PATH = "/oauth/token";

	private String clientId;

	private String clientSecret;

	private String userName;

	private String userPassword;

	private final Integer userId = 123456;

	private final String deviceId = "ABC";

	private final HttpOAuthClient client = new HttpOAuthClient();

	private final Gson gson = new Gson();


	public FlumeClient(String clientId, String clientSecret, String userName, String
userPassword) {

		this.clientId = clientId;

		this.clientSecret = clientSecret;

		this.userName = userName;

		this.userPassword = userPassword;

	}

	private OAuthAccessToken getAccessToken() throws Exception {

		OAuthAccessToken oAuthToken = client.getOAuthAccessToken();

		if (!oAuthToken.isValid()) {

			// new token

			FlumeAccessTokenRequest tokenRequest = new
FlumeAccessTokenRequest(clientId, clientSecret);

			tokenRequest.setUsername(userName);
```

```
                        tokenRequest.setPassword(userPassword);
                        String payload = gson.toJson(tokenRequest);
                        String responseStr = client.sendRequest(BASE_URL + TOKEN_PATH, "POST",
payload, null);
                        FlumeTokenResponse ftr = gson.fromJson(responseStr,
FlumeTokenResponse.class);
                        if (!ftr.isSuccess()) {
                                throw new Exception(ftr.buildErrorMessage());
                        }
                        FlumeToken ft = ftr.getData()[0];
                        oAuthToken = new OAuthAccessToken(ft.getAccessToken());
                        oAuthToken.setRefreshToken(ft.getRefreshToken());
                        oAuthToken.setExpiresIn(ft.getExpiresIn());
                        client.setOauthAccessToken(oAuthToken);
                } else if (oAuthToken.willExpire()) {
                        // refresh token
                        FlumeRefreshTokenRequest rtr = new FlumeRefreshTokenRequest(clientId,
clientSecret);
                        rtr.setRefreshToken(oAuthToken.getRefreshToken());
                        String payload = gson.toJson(rtr);
                        String responseStr = client.sendRequest(BASE_URL + TOKEN_PATH, "POST",
payload, null);
                        FlumeTokenResponse ftr = gson.fromJson(responseStr,
FlumeTokenResponse.class);
                        if (!ftr.isSuccess()) {
                                throw new Exception(ftr.buildErrorMessage());
                        }
                        FlumeToken ft = ftr.getData()[0];
                        oAuthToken.setAccessToken(ft.getAccessToken());
                        oAuthToken.setRefreshToken(ft.getRefreshToken());
                        oAuthToken.setExpiresIn(ft.getExpiresIn());
                }
                return oAuthToken;
        }
        public List<FlumeValue> fetchConsumption(String from, String to, String bucket) throws
Exception {
                // query for consumption
                String queryPath = "/users/" + userId + "/devices/" + deviceId + "/query";
                FlumeQuery query = new FlumeQuery(FlumeDataList.REQ_ID, bucket);
                query.setSinceDatetime(from);
                query.setUntilDatetime(to);
```

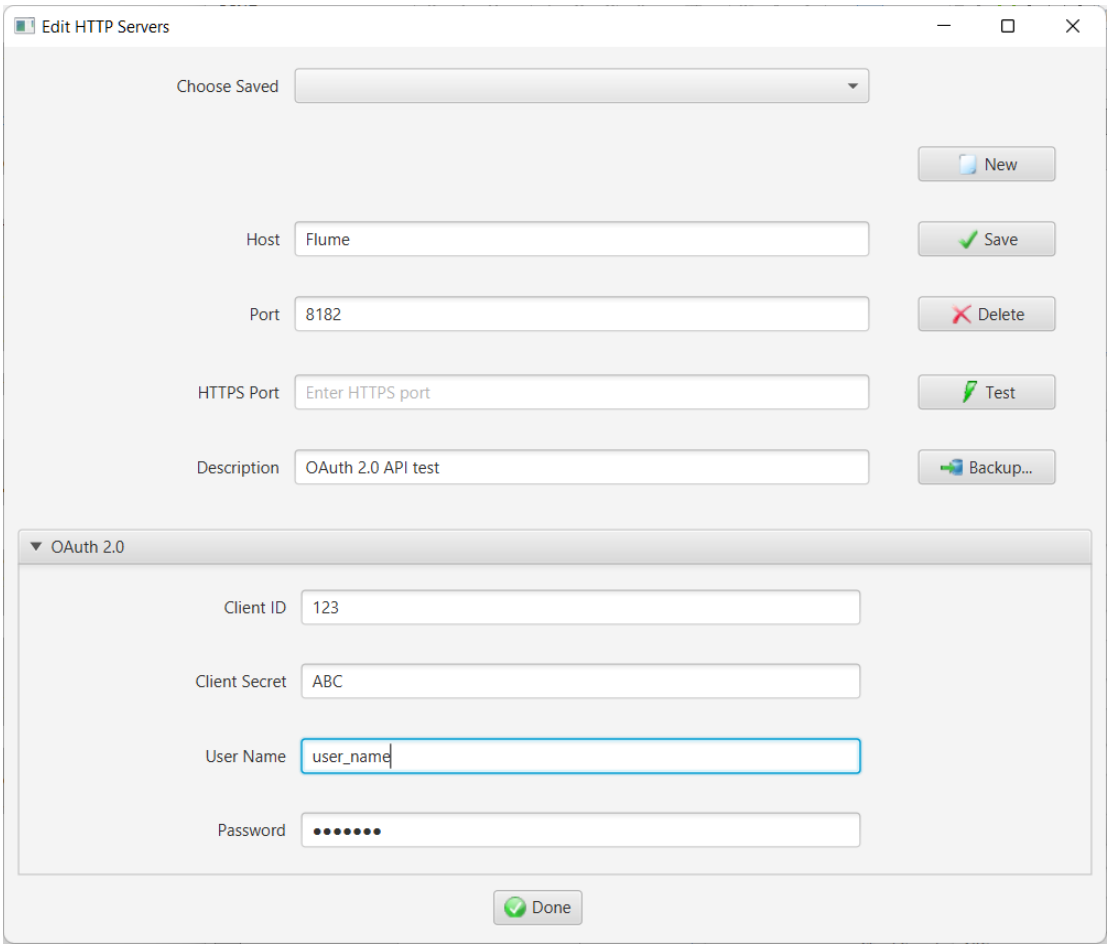
```
                query.setUnits("GALLONS");
                FlumeQueries queries = new FlumeQueries(query);
                String payload = gson.toJson(queries, FlumeQueries.class);
                OAuthAccessToken oAuthToken = getAccessToken();
                String responseStr = client.sendRequest(BASE_URL + queryPath, "POST", payload,
oAuthToken);
                FlumeQueryResponse fqr = gson.fromJson(responseStr, FlumeQueryResponse.class);
                if (!fqr.isSuccess()) {
                        throw new Exception(fqr.buildErrorMessage());
                }
                FlumeDataList dataList = fqr.getData().get(0);
                return dataList.getFlumeDataList();
        }
}
```

The JavaScript for the washer equipment is:

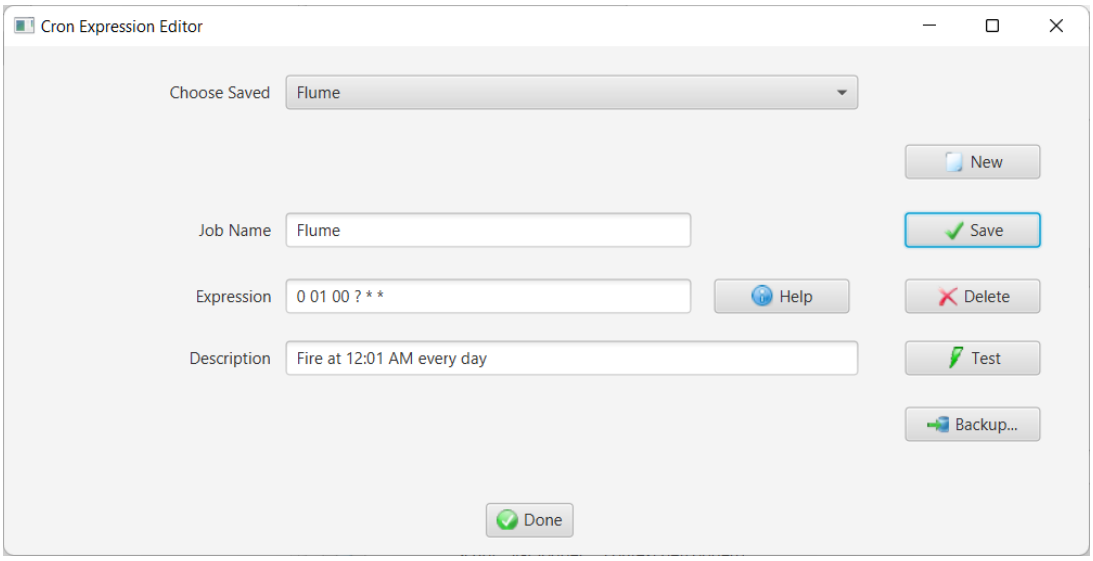
```
var logger = context.getLogger()
// date range
var DateTimeFormatter = Java.type('java.time.format.DateTimeFormatter')
var LocalDate= Java.type('java.time.LocalDate')
var now = LocalDate.now()
var to = now.format(DateTimeFormatter.ofPattern("YYYY-MM-dd"))  + " 00:00:00"
var before = now.plusDays(-1)
var from = before.format(DateTimeFormatter.ofPattern("YYYY-MM-dd"))  + " 00:00:00"
// persistence service
var PersistenceService = Java.type('org.point85.domain.persistence.PersistenceService')
var flumeSource = PersistenceService.instance().fetchDataSourceByName("Flume:8182")
var clientId = flumeSource.getClientId()
var clientSecret = flumeSource.getClientSecret()
var userName = flumeSource.getUserName()
var userPassword = flumeSource.getUserPassword()
var client = context.getProperty("FlumeClient")
if (client === null) {
    // create the Flume client
    client = new org.point85.flume.FlumeClient(clientId, clientSecret, userName, userPassword);
    context.setProperty("FlumeClient", client)
}
// read the hourly consumption
```

```
var values = client.fetchConsumption(from, to, "HR");

logger.info("Fetched hourly consumption " + from + ", to: " + to)

for (var i = 0; i < values.size(); ++i) {

    var item = values.get(i);

    logger.info(item.getDatetime() + ", " + item.getValue());

    var datetime = "'" + item.getDatetime() + "'";

    // insert into database

    var sql = "INSERT INTO PUBLIC.PUBLIC.WATER (\"date\", \"value\") VALUES(" + datetime + "," +
item.getValue() + ")";

    var count = PersistenceService.instance().executeUpdate(sql);

}

return value;
```

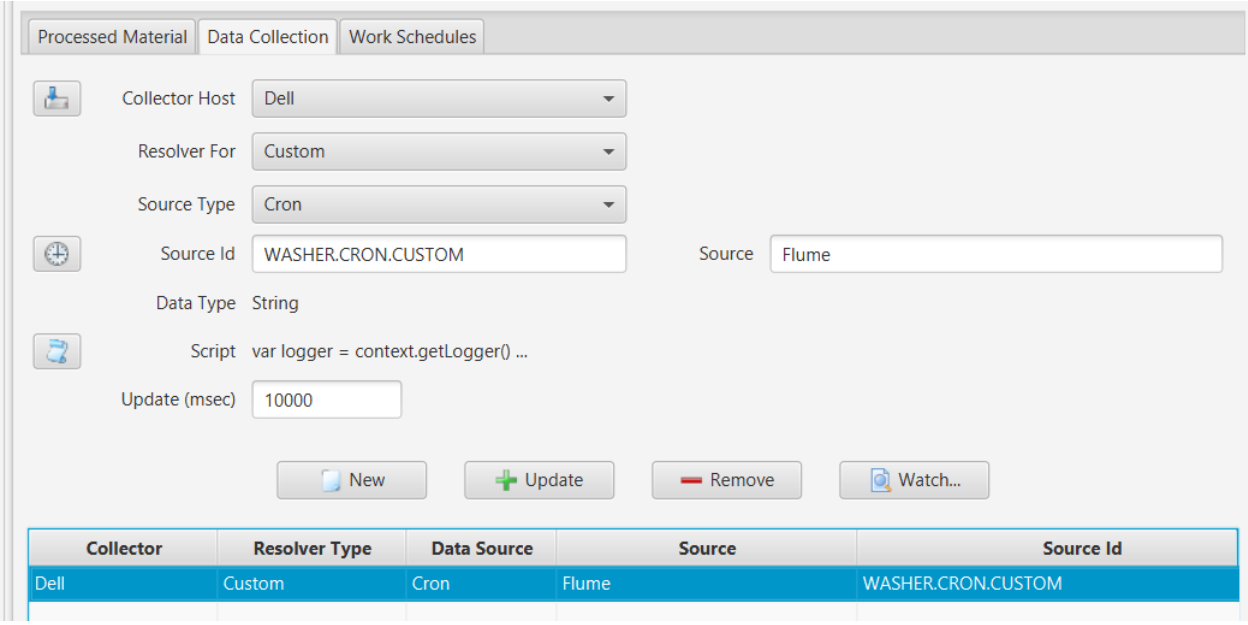
The OAuth 2.0 client id, client secret, user name and password are defined in an HTTP data source named “Flume:8182”:



The Cron expression is:



And the equipment data collection is:



Example 12 - Compute Incremental Production

For this example, a proximity sensor totals up good production cans as they move past it. An OPC UA server periodically sends the total good production. The OEE resolver script needs to compute the difference in production counts between these events. For this purpose, we initially query the database to get the stating count, then cache the last value in memory.

The script is (neglicting a possible rollover) is:

```
var PersistenceService = Java.type('org.point85.domain.persistence.PersistenceService');
```

```
var OeeEventType = Java.type('org.point85.domain.script.OeeEventType');

var Float = Java.type('java.lang.Float');

// get the last cached total good production

var lastTotal = context.getProperty("LAST_TOTAL");

if (lastTotal == null) {

	// fetch the last good production OEE event for this equipment

	var lastEvent = PersistenceService.instance().fetchLastEvent(resolver.getEquipment(),
OeeEventType.PROD_GOOD);

	// get the input (total good count) value

	lastTotal = Float.valueOf(lastEvent.getInputValue());

}

// new total

var currentTotal = Float.valueOf(value)

var delta = currentTotal - lastTotal;

context.setProperty("LAST_TOTAL", currentTotal);

return delta
```

Example 13 - Collect Environment Data with HTTP

For this example, a WiFi Arduino UNO R4 microcontroller is monitoring the temperature and humidity in a soaking area with the DHT20 sensor. It responds to an HTTP GET request by returning a string with the measured temperature and humidity. The Arduino URL is: http://192.168.1.17:80/TH.

The Arduino code is:

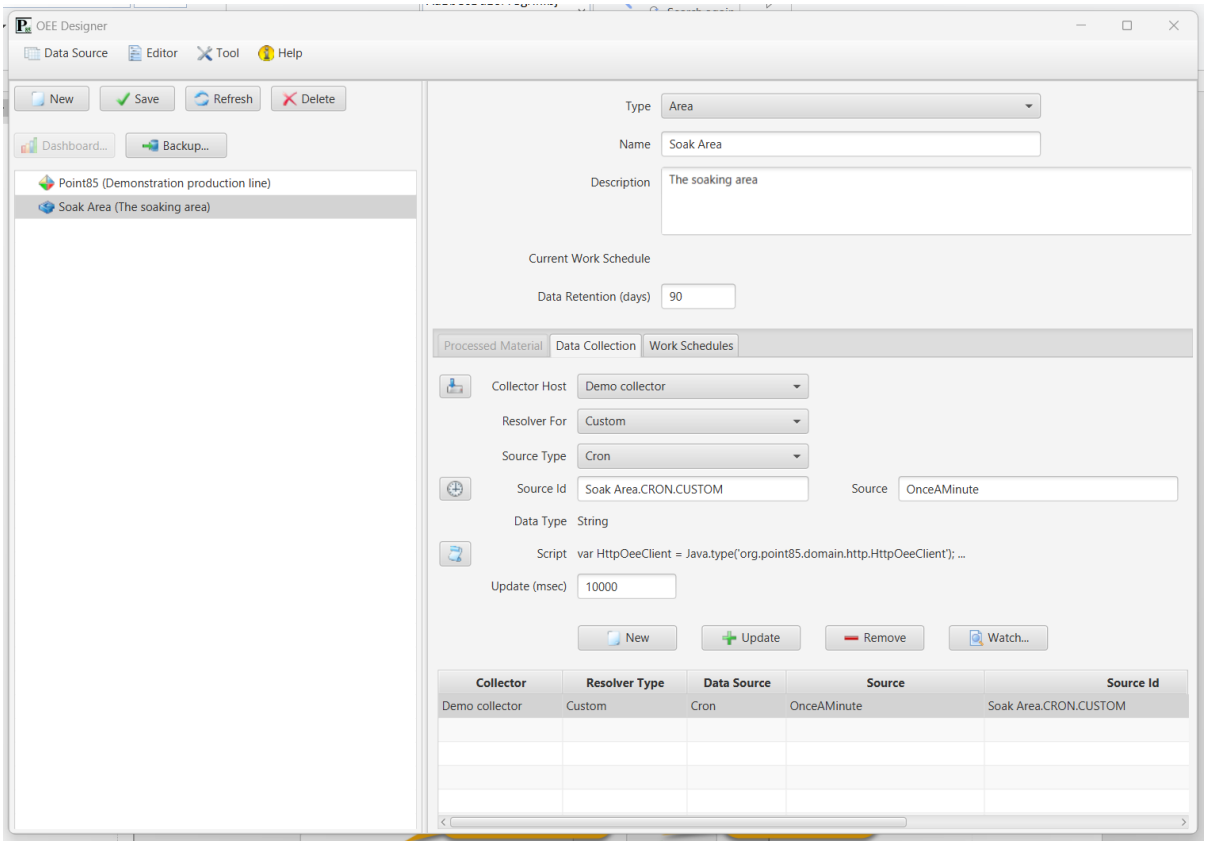
```
#include <WiFiS3.h>

#include "DHT20.h"

const char ssid[] = "Meru2G";

const char pass[] = "Denal!1985";

const char TH_RESOURCE[] = "TH";

// sensor class

DHT20 DHT;

// request

String request = "";

// string value of response to command

String response = "";

// server status

int status = WL_IDLE_STATUS;

// construct web server on port 80

WiFiServer server(80);
```

```
// read the DHT sensor
void readDHT() {
  int dht_status = DHT.read();
  float temp = DHT.getTemperature();
  float humid = DHT.getHumidity();
  switch (dht_status) {
    case DHT20_OK:
      response = "TH:" + String(temp) + ":" + String(humid);
      break;
    case DHT20_ERROR_CHECKSUM:
      response = "Checksum error";
      break;
    case DHT20_ERROR_CONNECT:
      response = "Connect error";
      break;
    case DHT20_MISSING_BYTES:
      response = "Missing bytes";
      break;
    case DHT20_ERROR_BYTES_ALL_ZERO:
      response = "All bytes read zero";
      break;
    case DHT20_ERROR_READ_TIMEOUT:
      response = "Read time out";
      break;
    case DHT20_ERROR_LASTREAD:
      response = "Error read too fast";
      break;
    default:
      response = "Unknown error";
      break;
  }
}
void setup() {
  // attempt to connect to WiFi network:
  while (status != WL_CONNECTED) {
    // Connect to WPA/WPA2 network.
```

```
    status = WiFi.begin(ssid, pass);

    // wait for connection:
    delay(10000);
  }
  server.begin();
  Wire.begin();
  DHT.begin();
}

void loop() {
  // listen for incoming clients
  WiFiClient client = server.available();
  if (client) {
    // read the HTTP request header line by line
    while (client.connected()) {
      if (client.available()) {
        String HTTP_header = client.readStringUntil('\n');
        int idx = HTTP_header.indexOf(TH_RESOURCE);
        if (idx != -1) {
          request = TH_RESOURCE;
        }
        if (HTTP_header.equals("\r"))
          break;
      }
    }
    if (request.equals(TH_RESOURCE)) {
      // read sensors and construct a string response
      readDHT();
    }
    // send the HTTP response
    client.println("HTTP/1.1 200 OK");
    client.println("Content-Type: text/plain");
    client.println("Connection: close");
    client.println();
    if (!request.equals("TH")) {
```

```
      response = "Invalid request";
    }
    client.println(response);
    client.flush();
    // give the web browser time to receive the data
    delay(100);
    // close the connection:
    client.stop();
    response = "";
    request = "";
  }
}
```

The Soak Area data collection is configured with a custom Cron data source firing once a minute on the minute (expression is “0 * * * * ?”):



The JavaScript makes an HTTP GET request to the Arduino’s URL:

```
var HttpOeeClient = Java.type('org.point85.domain.http.HttpOeeClient');
var HttpSource = Java.type('org.point85.domain.http.HttpSource');
var Float = Java.type('java.lang.Float');
```

```
// create the HTTP source for the GET
var source = new HttpSource("192.168.1.17", 80);
// create the HTTP client
var client = new HttpOeeClient(source);
// make the request
var response = client.sendGetRequest("TH", null);
// parse the returned temperature and humidity
tokens = response.split(":");
temperature = Float.valueOf(tokens[1]);
humidity = Float.valueOf(tokens[2]);
print(temperature);
print(humidity);
```

Example 14 - Collect Environment Data with MQTT

For this example, an Arduino UNO R4 microcontroller is monitoring the temperature and humidity in a soaking area with the DHT20 sensor. The device listens for a “TH” message from an MQTT broker request topic, then publishes the temperature and humidity response to a response topic on the broker. The readDHT() function is the same as in the HTTP example above.

The relevant Arduino code is:

```
#include <ArduinoMqttClient.h>
const char mqttDevice[] = "Arduino";
const char mqttUser[] = "";
const char mqttPass[] = "";
const char broker[] = "test.mosquitto.org";
int port   = 1883;
const char topic[]  = "Point85_REQ";
const char hostTopic[] = "Point85_RSP";
// create the MQTT client
WiFiClient wifiClient;
MqttClient mqttClient(wifiClient);
...
void setup() {
 mqttClient.setId(mqttDevice);
 mqttClient.setUsernamePassword(mqttUser, mqttPass);
 if (!mqttClient.connect(broker, port)) {
```

```
  while (1);
 }
 // set the message received callback
 mqttClient.onMessage(onMqttMessage);
 // subscribe to request topic
 mqttClient.subscribe(topic);
}
void loop() {
 mqttClient.poll();
}
void onMqttMessage(int messageSize) {
  // Read the message contents into a String
  String message = mqttClient.readString();
  // Check if the contents of the message match for reading temperature and humidity
  if(message.equals("TH")) {
    // read the sensor
    readDHT();
    // send response message
    mqttClient.beginMessage(hostTopic);
    mqttClient.print(response);
    mqttClient.endMessage();
  }
}
```

The JavaScript code from the HTTP example is changed to construct an MqttOeeClient and send the request to the Arduino, thus receiving the temperature and humidity string.

```
var MqttOeeClient = Java.type("org.point85.domain.mqtt.MqttOeeClient");
var QualityOfService = Java.type("org.point85.domain.mqtt.QualityOfService");
var client = new (MqttOeeClient);
client.connect("test.mosquitto.org", 1883);
client.subscribeToTopic("Point85_RSP");
// send the request message for temperature and humidity.  Wait for the response message
var response = client.publishAndWait("Point85_REQ", "TH", QualityOfService.EXACTLY_ONCE, 10);
client.disconnect();
print(response);
```

Example 15 - PackML OPC UA Production Count

In this example, the accumulated good production count is read from the PackMLCountDataType structure after subscribing to a ProdProcessedCount object node. The PackMLUtils.getDeltaAccumCount() method is call and its value returned by the script:

```
var PackMLUtils = Java.type('org.point85.domain.opc.ua.packml.PackMLUtils')

return PackMLUtils.getDeltaAccumCount(value, resolver)
```

This method first checks to see if a previous record has been saved in the OEE event table in order to establish a starting count for future events. Then, the current count is obtained by calling getAccCount() from the PackMLCountDataType object, and the difference returned as the increase in good production from the last event.

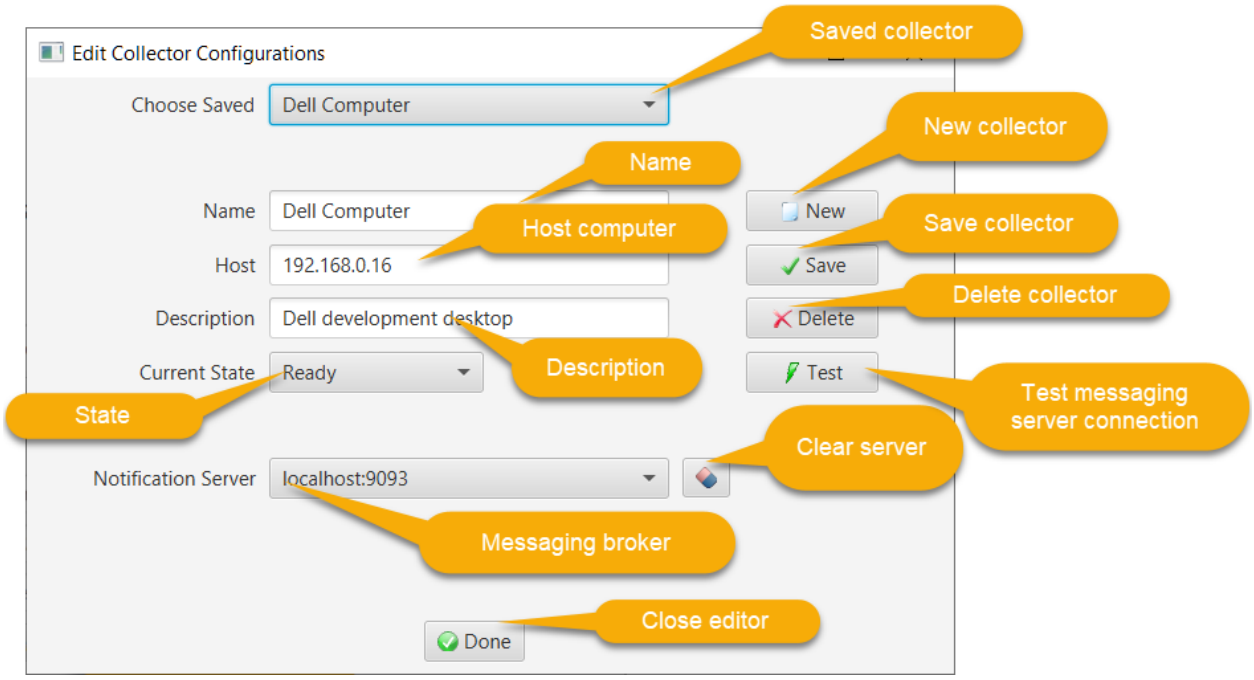
Example 16 - PackML OPC UA Alarm

In this example, a PackMLAlarmDataType node is subscribed to. When a value is received, the message id is extracted and sent as an email or text message by the script:

```
var logger = context.getLogger();

// get the one and only email client

var emailClient = context.getEmailClient()

// send the PackML alarm message

emailClient.sendMail("user@domain", "PackML Alarm", value.getMessage())

logger.info("Sent email to " + emailClient.getSource().getSendHost())

return value.getMessage()
```

Data Collector Editor

The collector for receiving data from data sources and resolving them to OEE events is defined in this editor. For example:



The state is one of DEV (under development and cannot be used), READY (released for use) or RUNNING (in use). All ready or running collectors will be started to collect data.

If collector status and notification messaging to the Monitor application is desired, then the previously defined messaging server is selected in the combobox. The setting may be removed by clicking the button next to the combobox.

The data collector editor buttons are:

- *New*: clear the editor to begin defining a new collector
- *Save*: save the selected collector to the database.
- *Delete*: delete the selected collector from the database
- *Backup*: save the selected collector (or all collectors if non is selected) to a .p85x file
- *Test*: test the messaging server connection

Data Source Editors

The data source editors define where a non-manual input value to a resolver script can come from. The supported sources are:

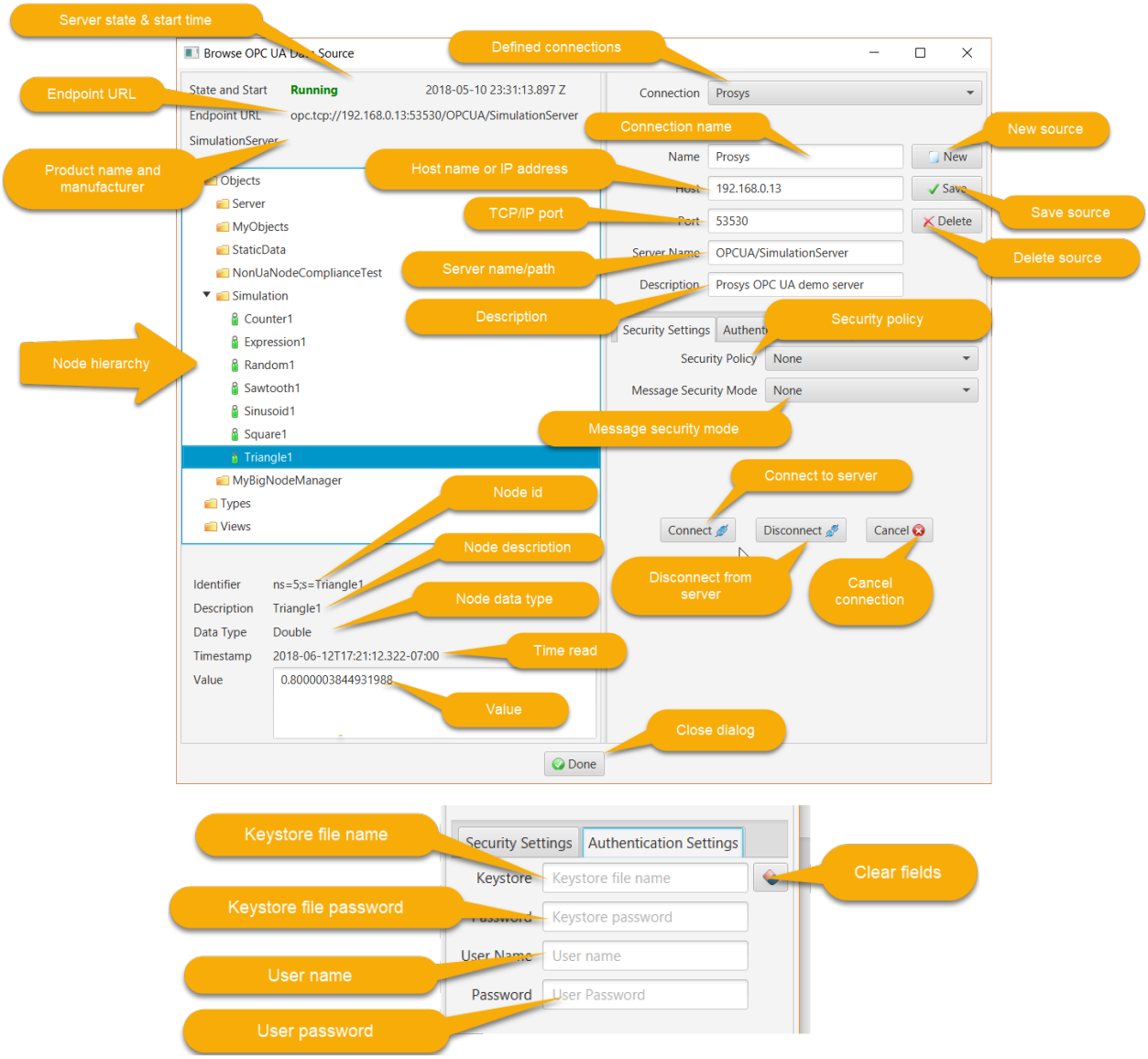
- OPC DA: classic OLE for Process Control (OPC) Data Acquisition
- OPC UA: OLE for Process Control Unified Architecture (UA)
- HTTP: invocation of an HTTP POST request with the data in the request body

- RabbitMQ: an event message received via a RabbitMQ message broker with the data as the message payload
- JMS: an event message received via an JMS message broker with the data as the message payload
- MQTT: an event message received via an MQTT message server with the data as the message payload
- Kafka: an event message received via a Kafka message broker with the data as the message payload
- Web Socket: an event message received via a web socket server with the data as the message payload
- Database: an event record(s) inserted into the DB_EVENT table. The JavaScript resolver's input value is a list of DatabaseEvent objects.
- File: a text file written into the ready folder. The JavaScript resolver's input value is content of this file.
- Modbus: A standard for reading and writing to a controller such as a PLC. The JavaScript resolver's input value is a list of data values read from a slave register(s).
- Cron Job: A job that is executed at points in time defined by a cron expression. When the job is executed it invokes the associated java script resolver.
- Email: an event message received via an email server with the data as the email Mime content
- Proficy: a GE Proficy Historian

OPC UA Data Source

Browser

The OPC UA data source browser dialog is launched from the toolbar or from the Data Collection tab after an equipment object with an OPC UA source has been selected in the physical model. It is used to browse to the node providing the input value. This dialog looks like:



In this example, the browser is anonymously connected to the Prosys demo server running on a host at 192.168.0.13 IP address on port 53530 with a server name/path of OPCUA/SimulationServer.

If a secure connection is desired for a server, under the Security Settings tab, the Security Policy can be chosen from None, Basic128Rsa15, Basic256, Basic256Sha256, Aes128_Sha256_RsaOaep or Aes256_Sha256_RsaPss. The Message Security Mode can be chosen from None, Sign or Sign & Encrypt. Under the Authentication Settings tab, the Java keystore file name can be specified in the Keystore text field and its password in the Password text field. The keystore file must be placed in the config/security folder. The user name and user password text fields can be used to specify the user name and password for user authentication. The button to the right of the keystore file name clears out these security settings.

The actions for an OPC UA data source are:

- *New*: clear the editing controls to define a new data source

- *Save*: save the data source to the database
- *Delete*: delete the data source from the database
- *Backup*: save the selected data source (or all sources if non is selected) to a .p85x file

The actions for establishing a connection are:

- *Connect*: connect to the data source
- *Disconnect*: disconnect from the data source
- *Cancel*: cancel an unsuccessful connection attempt

After a connection is established, the server namespace can be browsed. Selection of a node will display the current value and information about it below the tree view. In this example, the node namespace is 5 with a string id of "Triangle1".

Clicking the Done button will assign the selected node as the script resolver's input source.

Trending

Suppose that an OPC UA event resolver executes an availability script for a Unified Automation OPC UA demo server. An availability script must output a reason. For this node (triangle trend for a double value), the publishing interval is every 5 seconds. The script outputs a reason based on the input value:

```
var reason = 'Running';
if (value < 0.0)
{
    reason = 'Unplanned1';
}
return reason;
```

By clicking the Watch button for this resolver, the execution of the script can be observed:

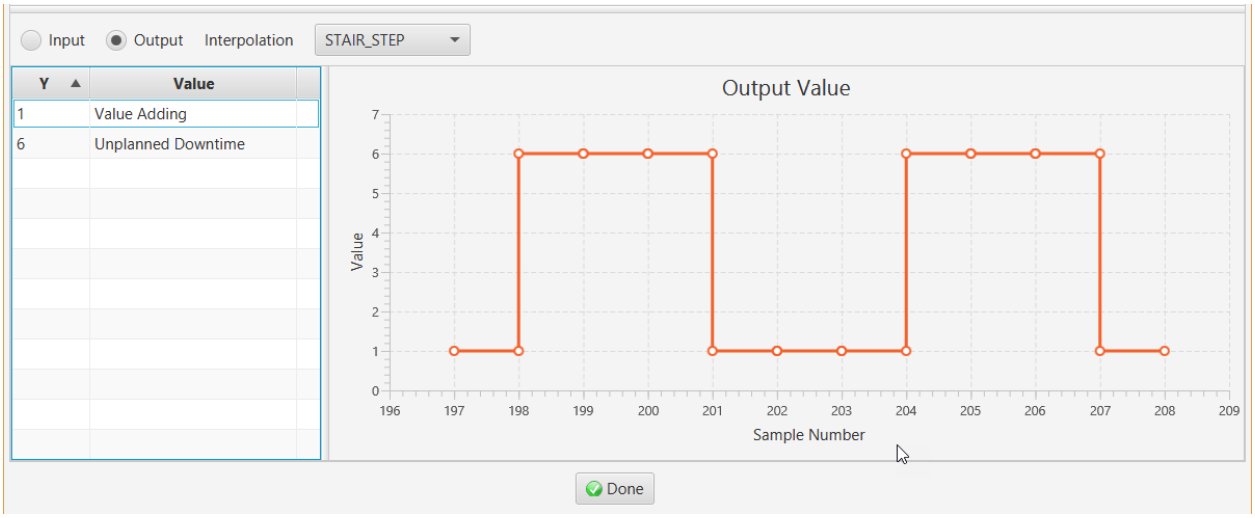
The actions are:

- *Connect*: connect to the OPC UA data source. After a successful connection, the server id and node ids are displayed along with the server status (Running in this case).
- *Disconnect*: disconnect from the data source
- *Cancel*: abort an unsuccessful connection attempt
- *Stop/Start*: The trending can be paused by clicking this button. The text will change to Start. The trend can be restarted by clicking it again.
- *Reset*: Restart trending after changing either the update period or number of points to display.

The table shows the item id, input value, timestamp and output value. If the output is an availability reason, the time loss category is displayed (Unplanned Downtime in this case).

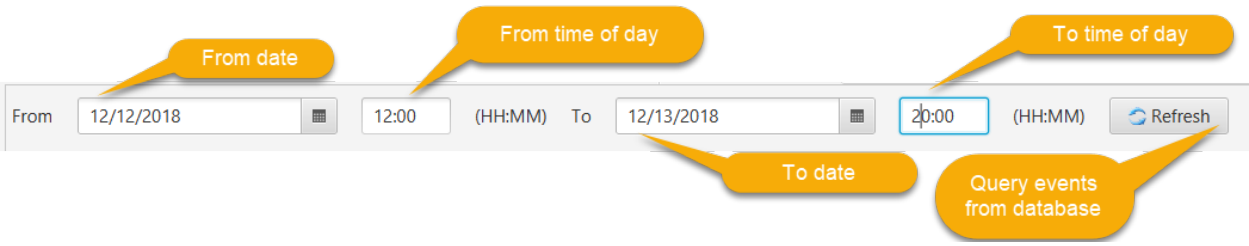
The current update period and number of data points to display on the X axis is displayed. These values can be changed when the trend is stopped and take effect after it is reset and restarted.

If Output is selected for the trend, the chart looks like:



Since the output is a reason, the values are discrete and thus a stair step interpolation is desired (the screen capture above displays a linear interpolation). The table on the left shows an integral value for the Y axis and the corresponding discrete value. Linear interpolation applies to continuous values such as integer or floating point data.

Previous event records can be fetched from the database and displayed in the table and in the trend chart.

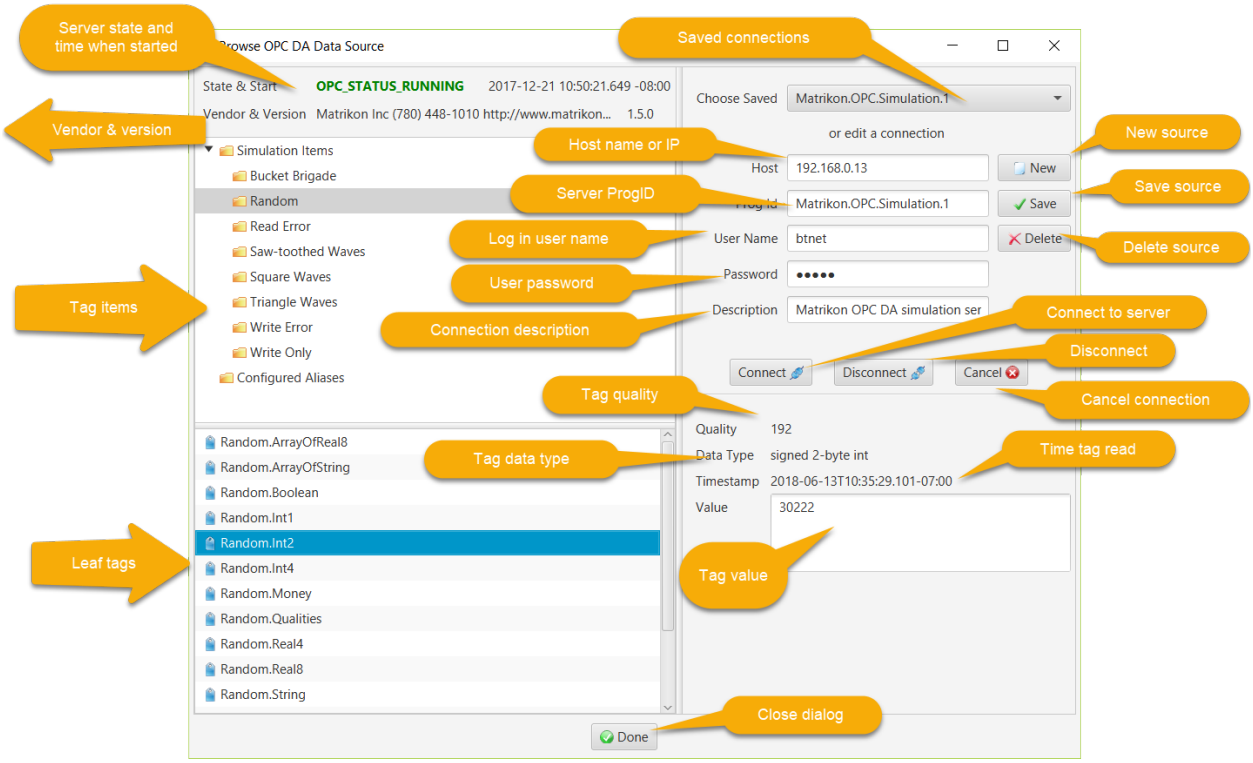


The beginning date and time of day can be set as well as the ending date and time of day. Both date/times are optional. Clicking the Refresh button will fetch records matching the date/time range from the database and display them in the event table and in the trend chart below.

OPC DA Data Source

Browser

The OPC DA data source browser dialog is launched from the toolbar or from the Data Collection tab after an equipment object with an OPC DA data source has been selected in the physical model. It is used to browse to the tag providing the input value. This dialog looks like:



In this example, the browser is connected to the Matrikon OPC simulation server on host 192.168.0.13 with a ProgID of Matrikon.OPC.Simulation.1 and the “btnet” user and password (note that the user name can include a Windows domain name).

The actions for an OPC DA data source browser are:

- *New*: clear the editing controls to define a new data source
- *Save*: save the data source to the database
- *Delete*: delete the data source from the database
- *Backup*: save the selected data source (or all sources if non is selected) to a .p85x file

The actions for establishing a connection are:

- *Connect*: connect to the data source
- *Disconnect*: disconnect from the data source
- *Cancel*: cancel an unsuccessful connection attempt

After a connection is established, the server tags can be browsed. Selection of a parent item of a leaf tag will display the children below the tree view. Selecting a leaf tag will display the tag’s current value and information about it to the right of the tree view.

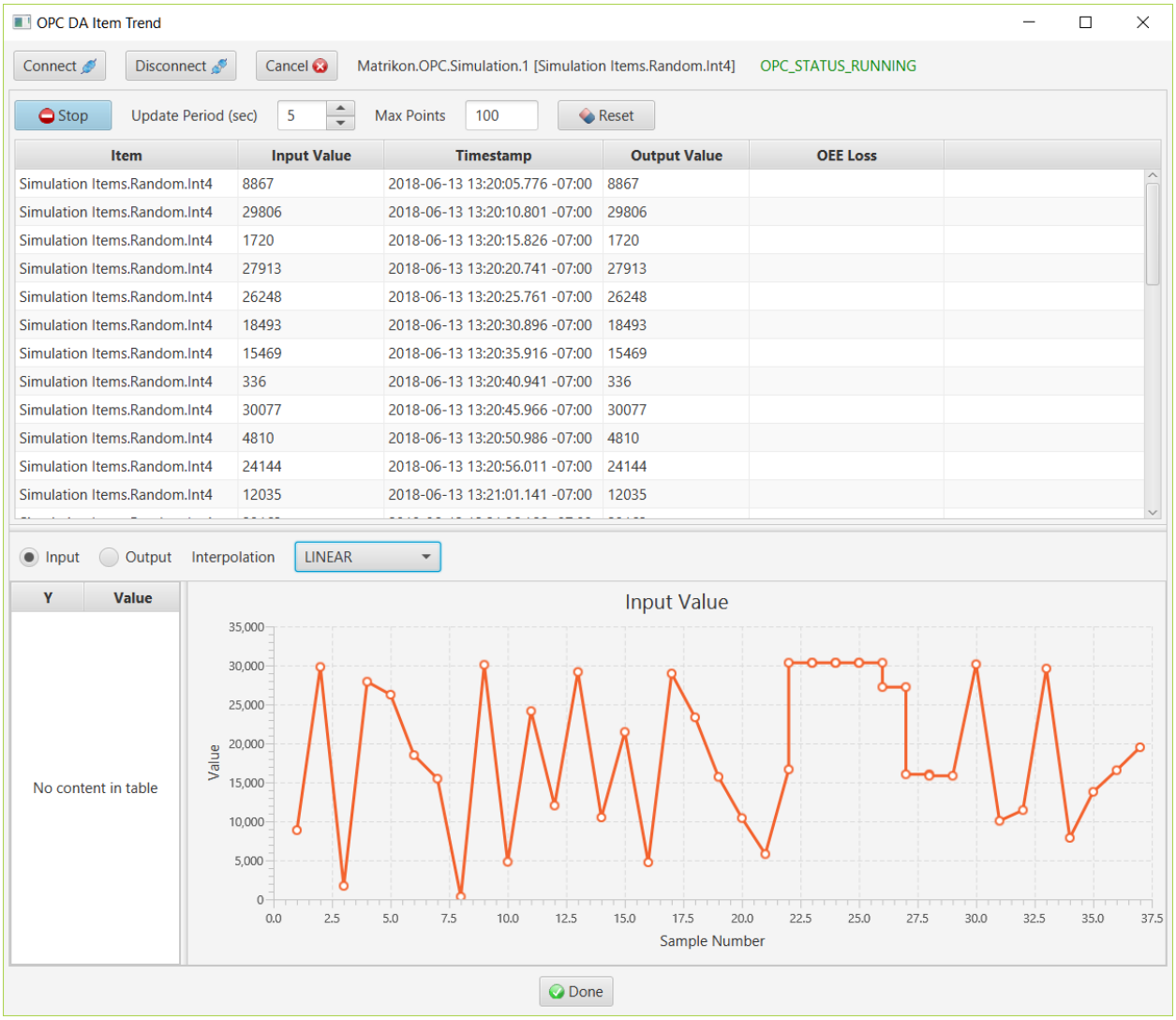
Clicking the Done button will assign the selected tag as the OPC DA script resolver’s input source.

Trending

By clicking the Watch button for an OPC DA resolver, the execution of the script can be observed. A trend chart for an OPC DA source for a 4-byte integer good production count with a pass-through resolver script of:

```
return value;
```

looks like:

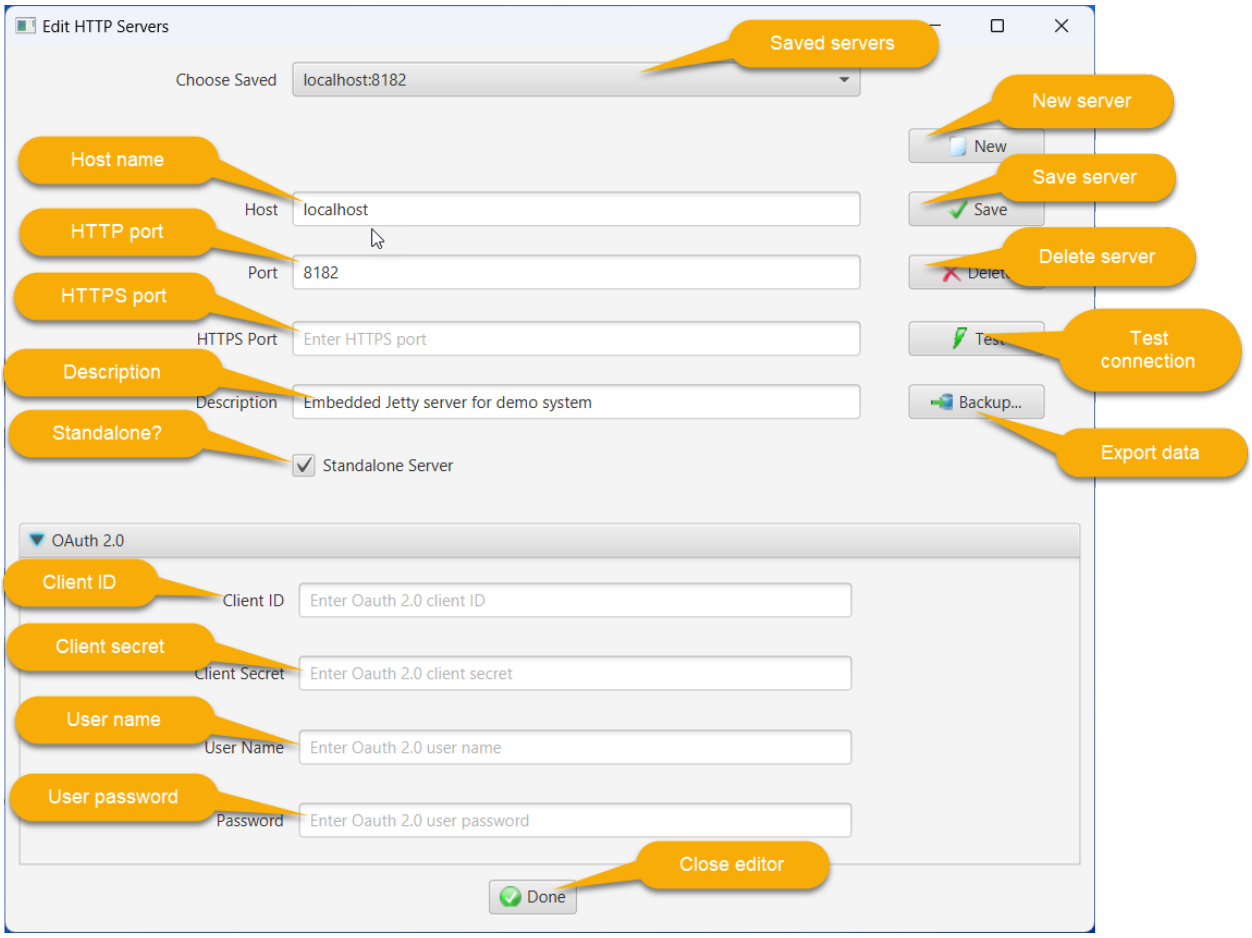


HTTP Data Source

Definition

The HTTP data source definition dialog is launched from the toolbar or from the Data Collection tab after an equipment object with an HTTP source has been selected in the physical model. It is used to define the host and port for the Jetty embedded HTTP server in the data collector.

This dialog is similar to:



The actions for an HTTP data source are:

- *New*: clear the editing controls to define a new data source
- *Save*: save the data source to the database
- *Delete*: delete the data source from the database
- *Test*: connect to the data source. A dialog will be presented indicating success or failure.
- *Backup*: save the selected data source (or all sources if non is selected) to a .p85x file

Clicking the Done button will assign the HTTP server as the script resolver's input source.

When the data collector is started on the specified host (localhost in this example), it will listen to the specified port (8180) ready to receive POST requests.

The "Standalone" checkbox indicates that this server is not associated with a resolver for a plant entity. If checked, then the Collector will start a Jetty servlet to handle requests from an external application.

If an HTTPS port is specified, HTTPS requests (GET or POST) can also be sent. In this case, the SSL certificate sent by a client must first be added to the keystore file called “point85-keystore.jks” located in the /config/security folder.

OAuth 2.0 configuration data can be associated with an HTTP source. The additonal fields are:

- Client ID: client identifier
- Client Secret: client secret
- User Name and Password: basic authentication credentials

For an example of OAuth 2.0 use, please see scripting Example #11.

Post Content

The POST request has the EquipmentEventRequestDto JSON serialized DTO (Data Transfer Object) as a body (sourceId, value and timestamp fields). This request is for an availability, production of material, setup or job change event.

There are three fields:

- sourceId (required): the source identifier as defined in the data collection script resolver
- value (required): the data value
- timestamp (optional): event time in ISO 8601 format

For example:

{"sourceId":"EQ1.HTTP.AVAILABILITY","value":"Running","timestamp":"2019-03-19T11:28:13.143-07:00"}

The HTTP server responds with the corresponding EquipmentEventResponseDto JSON body (status and errorText fields).

The body for no error is similar to:

```
{"status":"OK", "errorText":"OK"}
```

Java Client Example

An HTTP Java client can post an equipment event request. For example to loop-back test in the HTTP trend dialog executes this code:

```
@FXML
private void onLoopbackTest() {
        HttpURLConnection conn = null;
        try {
                // get the HTTP data source
                EventResolver eventResolver = trendChartController.getEventResolver();
                HttpSource dataSource = (HttpSource) eventResolver.getDataSource();

                // build the URL for an equipment event
                URL url = new URL(
                        "http://" + dataSource.getHost() + ":" + dataSource.getPort() + '/' +
OeeHttpServer.EVENT_EP);

                // create a connection for a JSON POST request
                conn = (HttpURLConnection) url.openConnection();
                conn.setDoOutput(true);
```

```
                conn.setRequestMethod("POST");
                conn.setRequestProperty("Content-Type", "application/json");

                // the value to send (must match the configured resolver)
                String value = tfLoopbackValue.getText();

                // timestamp when sent
                String timestamp = DomainUtils.offsetDateTimeToString(OffsetDateTime.now());

                // create the data transfer event object
                EquipmentEventRequestDto dto = new EquipmentEventRequestDto(eventResolver.getSourceId(),
value, timestamp);

                // serialize the body
                Gson gson = new Gson();
                String payload = gson.toJson(dto);

                // make the request
                OutputStream os = conn.getOutputStream();
                os.write(payload.getBytes());
                os.flush();

                if (logger.isInfoEnabled()) {
                        logger.info("Posted equipment event request to URL " + url + " with value " + value);
                }

                // check the response code
                int codeGroup = conn.getResponseCode() / 100;

                if (codeGroup != 2) {
                        String msg = "Post failed, error code : " + conn.getResponseCode() + "\nEquipment
event response ...";
                        BufferedReader br = new BufferedReader(new
InputStreamReader((conn.getInputStream())));
                        String output;

                        while ((output = br.readLine()) != null) {
                                msg += "\n" + output;
                        }
                        throw new Exception(msg);
                }
        } catch (Exception e) {
                AppUtils.showErrorDialog(e);
        } finally {
                conn.disconnect();
        }
}
```

Database Trigger Example

For another example, a database table insertion trigger can be used to asynchronously post equipment event messages to an HTTP collector. For example, SQL Server supports creating a stored procedure in C#. This procedure can then be executed in a trigger. The C# codes makes the HTTP request and receives the response. For simplicity, the values inserted into an EQUIPMENT_EVENT table row will be input to the stored procedure and then posted to the HTTP collector at the specified URL.

The EQUIPMENT_EVENT data table is created as:

```
        CREATE TABLE [dbo].[EQUIPMENT_EVENT] (

            [Id]          INT                NOT NULL,

            [SOURCE_ID]   NVARCHAR (64)      NOT NULL,

            [VALUE]       NVARCHAR (32)      NOT NULL,

            [EVENT_TIME] DATETIMEOFFSET (3) NOT NULL

        );
```

Suppose that a pass-through availability script resolver with source id = "e1.avail" and data value = "r1" has been created. "r1" is a reason with a loss category. When the following row is inserted into the EQUIPMENT_EVENT table, we want to call the stored procedure to make the HTTP request:

```
insert into EQUIPMENT_EVENT (Id, SOURCE_ID, VALUE, EVENT_TIME) values (1, 'e1.avail', 'r1', SYSDATETIMEOFFSET())
```

The insertion database trigger for the table is created as:

```
CREATE TRIGGER [ON_EVENT]
	ON [dbo].[EQUIPMENT_EVENT]
	FOR INSERT
	AS
	BEGIN
		SET NOCOUNT ON
		-- event endpoint
		declare @url nvarchar(128)
		set @url = 'http://machine_ip:8184/event'
		declare @response nvarchar(1024)
		declare @sourceId nvarchar(64)
		declare @value nvarchar(64)
		declare @timestamp datetimeoffset(3)
		declare @event_time nvarchar(64)
		select @sourceId = i.SOURCE_ID, @value = i.VALUE, @timestamp = i.EVENT_TIME from inserted i
		select @event_time = convert(nvarchar(64), @timestamp, 126)
		exec PostEquipmentEvent @url, @sourceId, @value, @event_time, @response output
	END
```

Here the HTTP data collector is running at "machine_ip" address on port 8184. The data to be sent to the collector is obtained from the "inserted" row and then passed into the PostEquipmentEvent stored procedure. The collector's JSON response is returned in the @response output parameter.

The PostEquipmentEvent stored procedure is written in C# as:

```
public partial class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void PostEquipmentEvent(string url, string sourceId, string value, string
timestamp, out string result)
    {
            // POST equipment event
            // json content
```

```
                string content = "{\"sourceId\":\"" + sourceId + "\",\"value\":\"" + value +
"\",\"timestamp\":\"" + timestamp + "\"}";

                // create Http request

                HttpWebRequest request = (HttpWebRequest)WebRequest.Create(url);

                byte[] data = Encoding.ASCII.GetBytes(content);

                request.Method = "POST";

                request.ContentType = "application/json";

                request.ContentLength = data.Length;

                // make the request

                Stream postStream = request.GetRequestStream();

                postStream.Write(data, 0, data.Length);

                // wait for the response

                HttpWebResponse response = (HttpWebResponse)request.GetResponse();

                result = new StreamReader(response.GetResponseStream()).ReadToEnd();

                response.Close();

                postStream.Close();

        }

}
```

Android Example

The HTTP URL can be called from an Android application. In this case, the user interface is built using the native IDE (Android Studio or Visual Studio Code and Dart/Flutter for Android). An HTTP client API is then called to make a request and receive a response.

For example, a Swift function for a POST request is:

```
    // send an HTTP POST request with body data

    private func sendPostRequest(_ url: String, body: String) -> NSError? {

        if let error = validateRequest() {

            return error

        }

        let nsUrl = URL(string : url)!

        var request = URLRequest(url: nsUrl)

        request.httpMethod = "POST"

        let bodyData : Data = body.data(using: String.Encoding.utf8)!

        request.httpBody = bodyData

        dataTask = dataSession.dataTask(with: request, completionHandler: {

            data, response, error in
```

```
            // flag that task is done
            self.dataTask = nil
            // call back handler
            self.handler!.handleResponse(nsUrl, data: data, error: error)
        })
        dataTask?.resume()
        return nil
    }
```

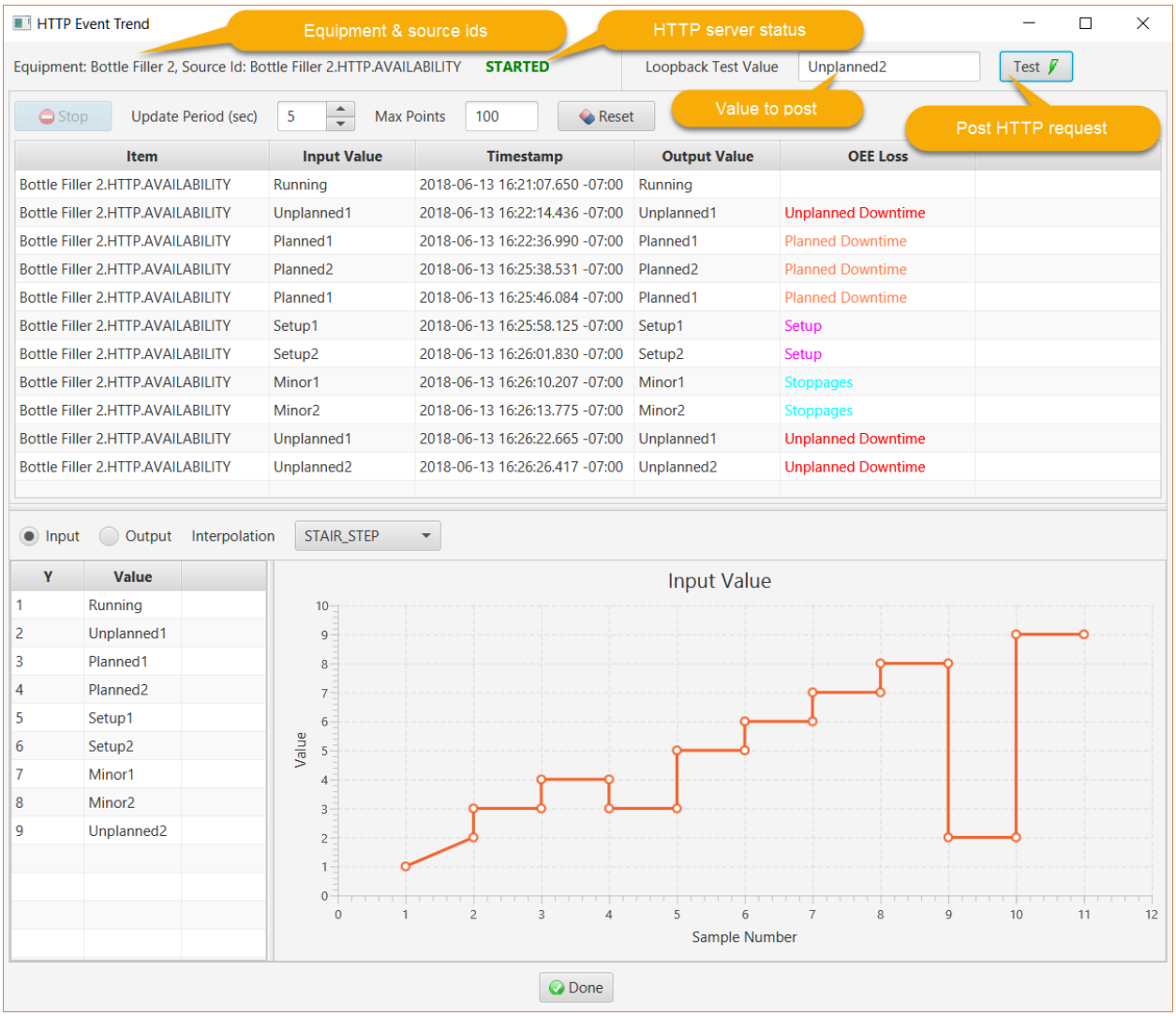
The caller of this function provides the URL with the endpoint (e.g. “event”) and the JSON serialized body.

Trending

By clicking the Watch button for an HTTP resolver, the execution of the script can be observed. An HTTP source for equipment availability with a pass-through resolver script of:

```
return value;
```

looks like:



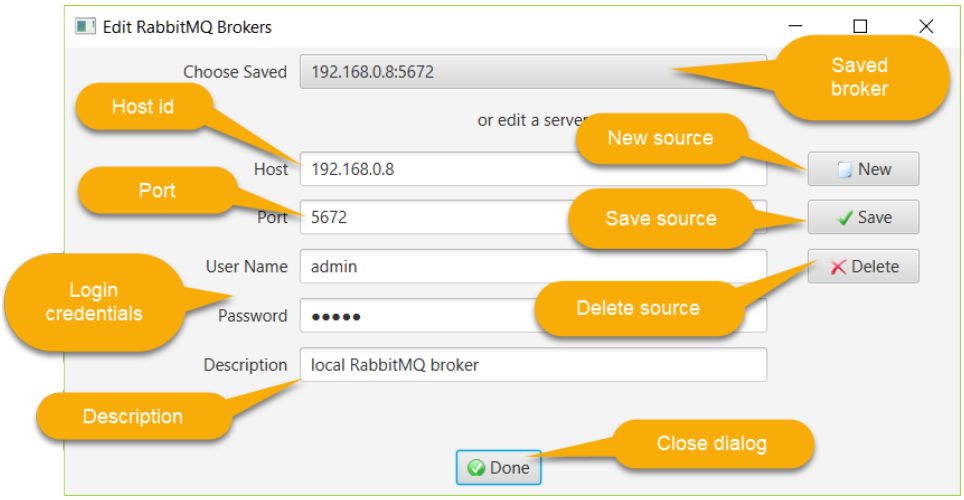
In this example, reasons have been entered into the loop-back test field and the Test button clicked to send a POST request to the HTTP server embedded in the controller for this dialog.

RMQ Data Source

Definition

The RMQ messaging data source definition dialog is launched from the toolbar or from the Data Collection tab for a RMQ messaging resolver after an equipment object has been selected in the physical model. It is used to define the RabbitMQ broker host, port and login credentials. By default the RMQ broker uses the AMQP protocol.

This dialog is similar to:



For this example, the RabbitMQ broker is running on host 192.168.0.8 on the default port of 5672. The client will login as the "admin" user.

The actions for a messaging data source are:

- *New*: clear the editing controls to define a new data source
- *Save*: save the data source to the database
- *Delete*: delete the data source from the database
- *Backup*: save the selected data source (or all sources if non is selected) to a .p85x file
- *Test*: connect to the data source. A dialog will be presented indicating success or failure

Clicking the Done button will assign the broker as the script resolver's input source.

Message Content

The body of the RMQ message is a JSON-serialized EquipmentEventMessage with four fields:

- sourceId (required): the source identifier as defined in the data collection script resolver
- value (required): the data value
- messageType (required): must be "EQUIPMENT_EVENT"
- timestamp (optional): event time in ISO 8601 format

For example:

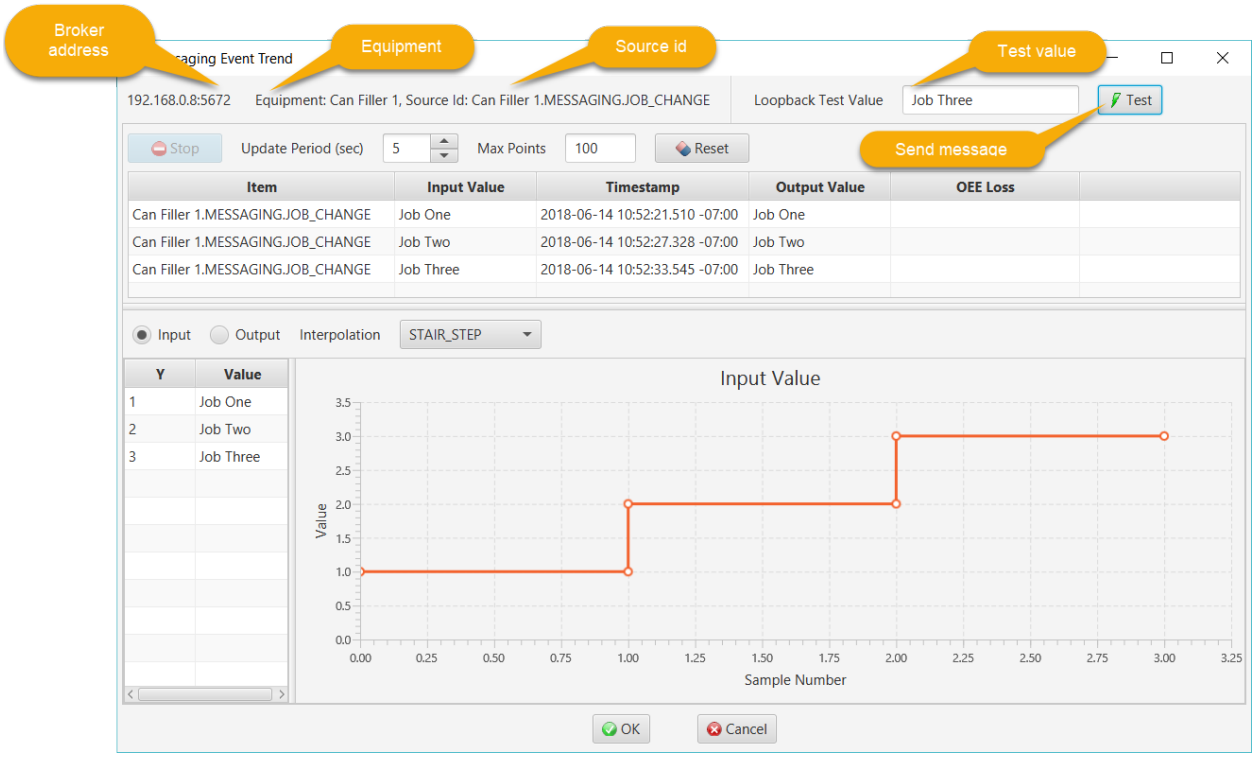
{"sourceId":"EQ1.MESSAGING.PROD_STARTUP","value":"1","messageType":"EQUIPMENT_EVENT","timestamp":"2019-03-19T11:24:22.923-07:00"}

Trending

By clicking the Watch button for an messaging resolver, the execution of the script can be observed. A trend for a messaging source for a job change with a pass-through resolver script of:

```
return value;
```

looks like:



In this example, job identifiers have been entered into the loop-back test field and the Test button clicked to send a JSON serialized EquipmentEventMessage to the specified RabbitMQ broker. The messaging trend controller is listening for these messages from the Point85 exchange and routed to its queue.

The Java code is:

```
@FXML
private void onLoopbackTest() {
        try {
                if (pubSub == null) {
                        throw new Exception("The trend is not connected to an RMQ broker.");
                }
                EventResolver eventResolver = trendChartController.getEventResolver();
                String sourceId = eventResolver.getSourceId();
                String value = tfLoopbackValue.getText();
                EquipmentEventMessage msg = new EquipmentEventMessage();
                msg.setSourceId(sourceId);
                msg.setValue(value);
                pubSub.publish(msg, RoutingKey.EQUIPMENT_SOURCE_EVENT, 30);
        } catch (Exception e) {
```

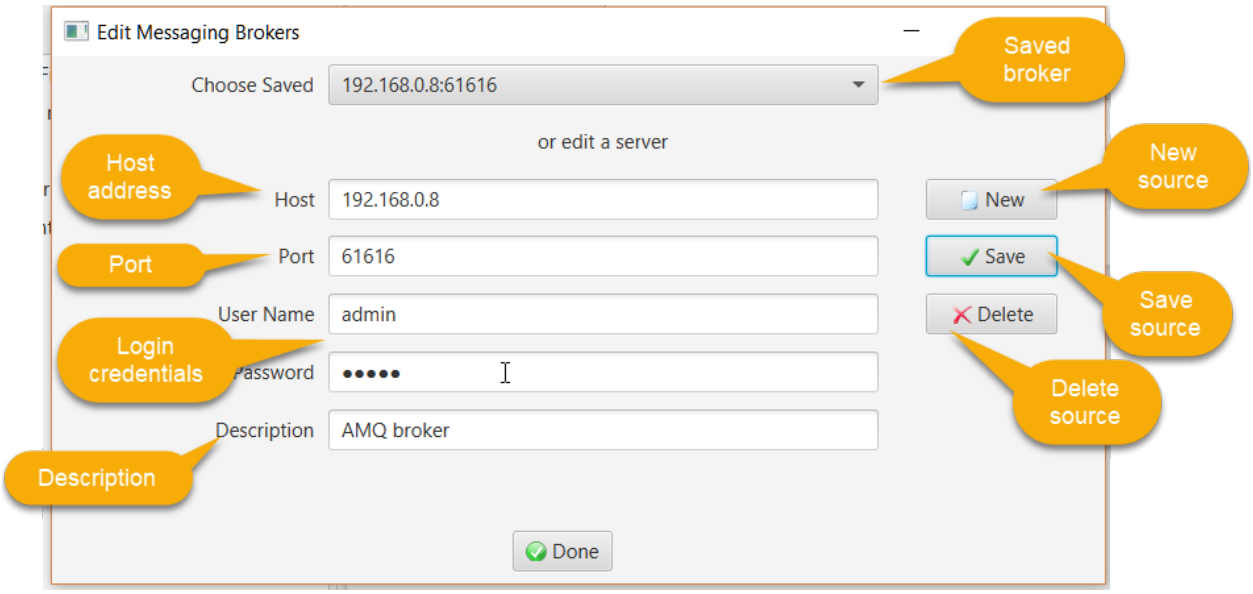
```
                    AppUtils.showErrorDialog(e);
                }
            }
```

JMS Data Source

Definition

The JMS data source definition dialog is launched from the toolbar or from the Data Collection tab for a JMS resolver after an equipment object has been selected in the physical model. It is used to define the JMS broker host, port and login credentials. By default the JMS client uses the AMQP protocol.

This dialog is similar to:



For this example, the ActiveMQ broker is running on host 192.168.0.8 on the default port of 61616. The client will login as the “admin” user.

The actions for a JMS data source are:

- *New*: clear the editing controls to define a new data source
- *Save*: save the data source to the database
- *Delete*: delete the data source from the database
- *Backup*: save the selected data source (or all sources if non is selected) to a .p85x file
- *Test*: connect to the data source. A dialog will be presented indicating success or failure

Clicking the Done button will assign the JMS broker as the script resolver’s input source.

Message Content

The body of the JMS message is a JSON-serialized EquipmentEventMessage with four fields:

- sourceId (required): the source identifier as defined in the data collection script resolver
- value (required): the data value
- messageType (required): must be “EQUIPMENT_EVENT”
- timestamp (optional): event time in ISO 8601 format

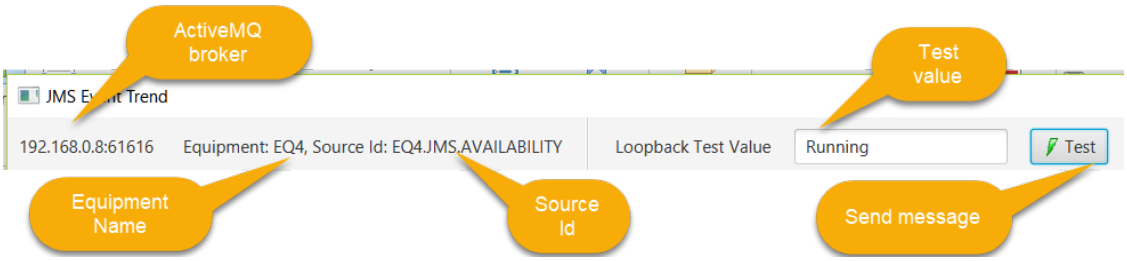
For example:

{"sourceId":"EQ1.JMS.PROD_GOOD","value":"1","messageType":"EQUIPMENT_EVENT","timestamp":"2019-03-19T11:13:45.977-07:00"}

Trending

By clicking the Watch button for an JMS resolver, the execution of the script can be observed.

A trend for a JMS source is similar to the RMQ messaging trend chart. In this case however, the top portion of the dialog looks like:

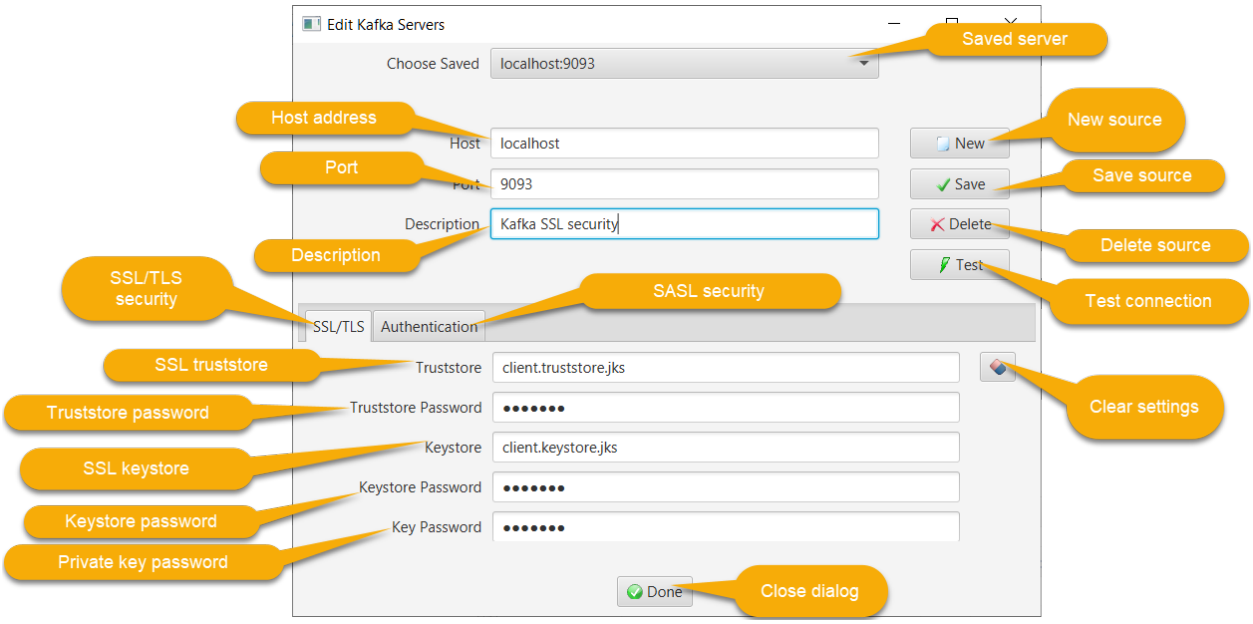


Kafka Data Source

Definition

The Kafka data source definition dialog is launched from the toolbar or from the Data Collection tab for a Kafka resolver after an equipment object has been selected in the physical model. It is used to define the Kafka server host, port and security credentials.

This dialog is similar to:



For this example, the Kafka server is running on the localhost on the SSL default port of 9093 and is using two-way SSL/TLS security. The SSL truststore file is named "client.truststore.jks" and must be located in the \config\security folder. The SSL keystore file is named "client.keystore.jks" and must be located in the \config\security folder. Both SSL stores have a password. In addition, with mutual SSL, the private key password must be specified. The keystores are of type PKCS12.

The second tab allows configuration of the user name and password for SASL authentication:

All security settings are optional.

The actions for a Kafka data source are:

- *New*: clear the editing controls to define a new data source
- *Save*: save the data source to the database
- *Delete*: delete the data source from the database
- *Backup*: save the selected data source (or all sources if non is selected) to a .p85x file
- *Test*: connect to the data source. A dialog will be presented indicating success or failure

Clicking the Done button will assign the Kafka broker as the script resolver's input source if the dialog is launched from the plant entity's data collection tab.

Message Content

The Kafka message is a JSON-serialized EquipmentEventMessage with four fields:

- sourceId (required): the source identifier as defined in the data collection script resolver
- value (required): the data value
- messageType (required): must be “EQUIPMENT_EVENT”
- timestamp (optional): event time in ISO 8601 format

For example:

{"sourceId":"EQ1.KAFKA.PROD_GOOD","value":"1","messageType":"EQUIPMENT_EVENT","timestamp":"2019-11-19T11:13:45.977-07:00"}

The Kafka message key is a string identifying the type of the message. Event messages are consumed from the Point85_Event topic with keys “COMMAND” or “EQUIP_EVENT”. Notification messages are produced to the Point85_Notification topic with keys “NOTIFICATION”, “STATUS” or “RESOLVED_EVENT”.

Trending

By clicking the Watch button for an Kafka resolver, the execution of the script can be observed.

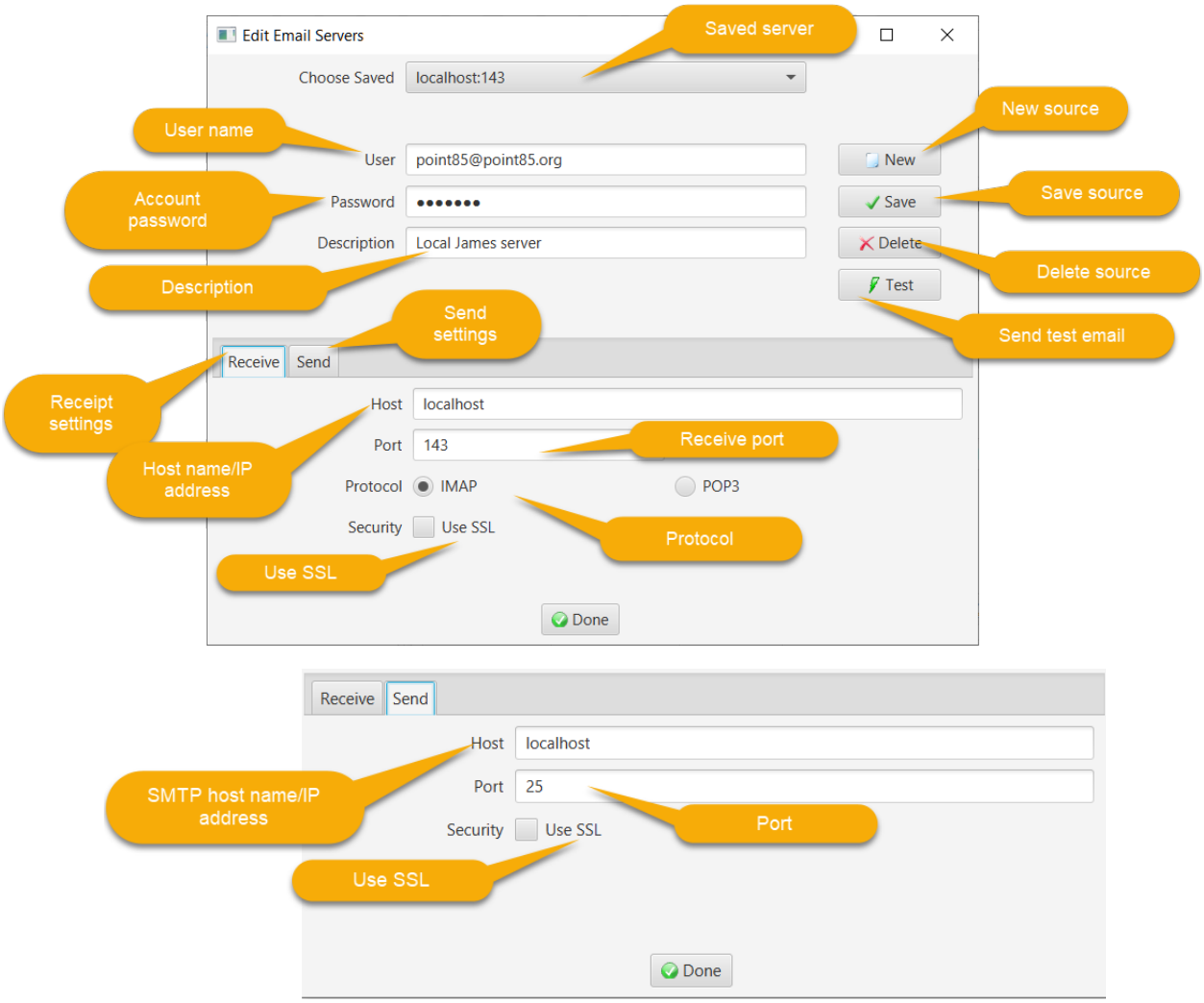
A trend for a Kafka source is similar to the RMQ messaging trend chart. In this case however, the top portion of the dialog looks like:

Email Data Source

Definition

The email data source definition dialog is launched from the toolbar or from the Data Collection tab for an email event resolver after an equipment object has been selected in the physical model. It is used to define the email server account, host, port and security credentials.

This dialog is similar to:



For this example, the Apache James server is running on the localhost on the IMAP default port of 143 and SMTP port of 25. SSL is not used for either receiving or sending mail. The account credentials are entered following by an optional description.

The Test button will send an email to the server to verify that the send settings are correct. Receipt of this email can be verified by using an appropriate email client for the account or by using the Point85 resolver trend chart capability as described below.

The actions for an email data source are:

- *New*: clear the editing controls to define a new data source
- *Save*: save the data source to the database
- *Delete*: delete the data source from the database
- *Backup*: save the selected data source (or all sources if non is selected) to a .p85x file
- *Test*: connect to the data source. A dialog will be presented indicating success or failure

Clicking the Done button will assign the email server as the script resolver's input source if the dialog is launched from the plant entity's data collection tab.

Message Content

The email content is a JSON-serialized EquipmentEventMessage with four fields:

- sourceId (required): the source identifier as defined in the data collection script resolver
- value (required): the data value as a string
- messageType (required): must be "EQUIPMENT_EVENT"
- timestamp (optional): event time in ISO 8601 format

For example:

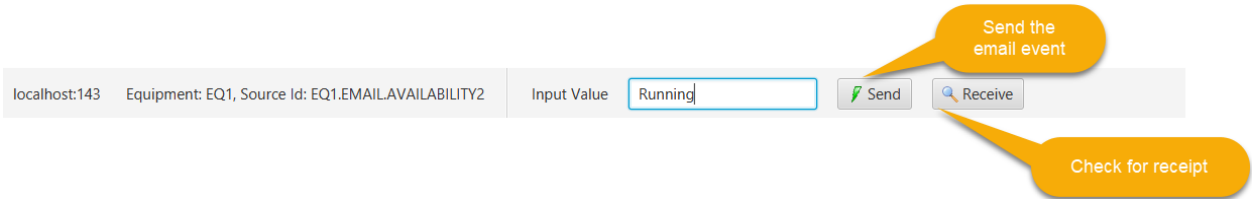
{"sourceId":"EQ1.EMAIL.PROD_GOOD","value":"100","messageType":"EQUIPMENT_EVENT","timestamp":"2020-11-19T11:13:45.977-07:00"}

Email messages are received according to the data source's IMAP or POP3 settings.

Trending

By clicking the Watch button for an email resolver, the execution of the script can be observed.

A trend for an email source is similar to the RMQ or JMS messaging trend chart. In this case however, the top portion of the dialog looks like:



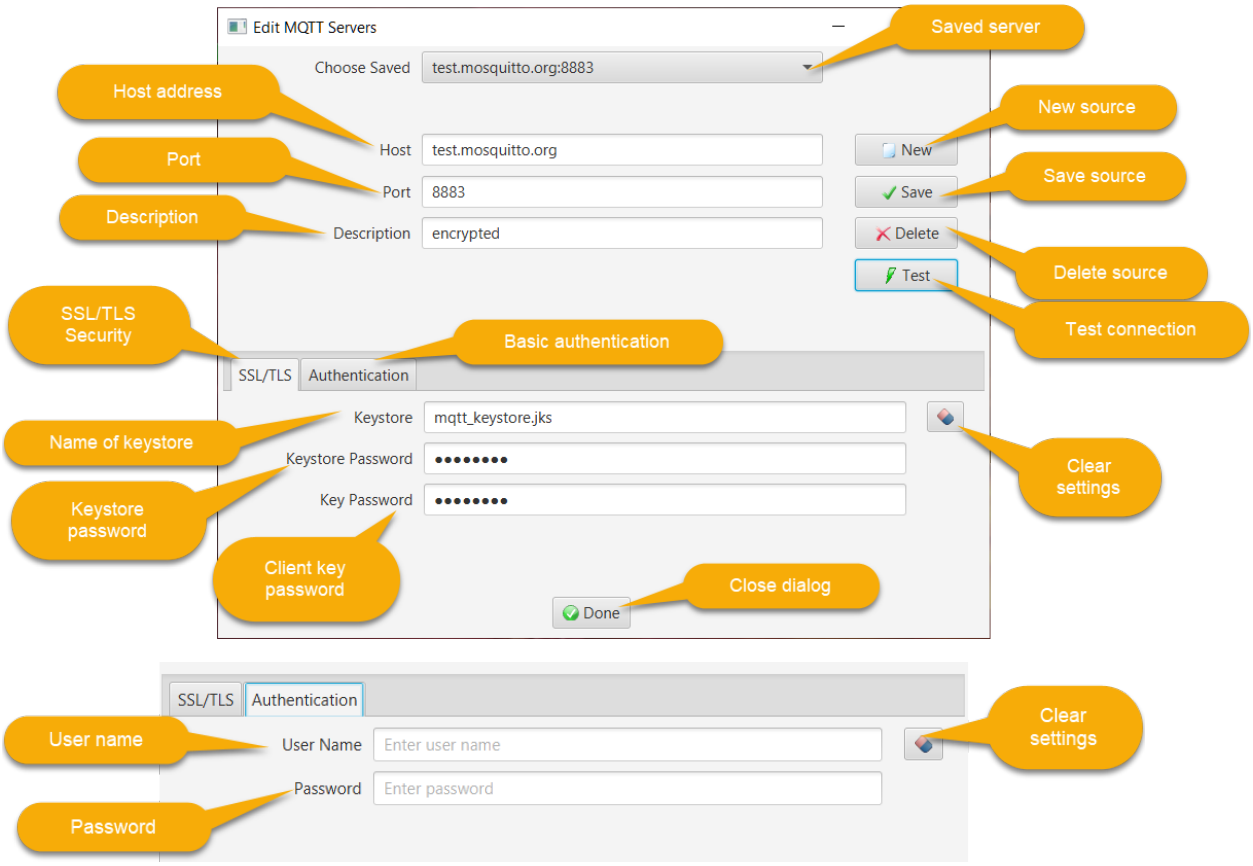
The Send button sends an availability email to the server. The same email is received by clicking on the Receive button sometime after the server has received the email.

MQTT Data Source

Definition

The MQTT data source definition dialog is launched from the toolbar or from the Data Collection tab for an MQTT resolver after an equipment object has been selected in the physical model. It is used to define the MQTT server host, port and security credentials.

This dialog is similar to:



In this example, the MQTT server is running on host test.eclipse.org on the SSL/TLS port of 8883. The keystore (and truststore) is named "mqtt_keystore.jks" and is located in Point85's config/security folder. The keystore and client key passwords are required. Since SSL/TLS is being used, the user name and password basic authentication is not defined.

The actions for an MQTT data source are:

- *New*: clear the editing controls to define a new data source
- *Save*: save the data source to the database
- *Delete*: delete the data source from the database
- *Backup*: save the selected data source (or all sources if non is selected) to a .p85x file
- *Test*: connect to the data source. A dialog will be presented indicating success or failure

Clicking the Done button will assign the MQTT server as the script resolver's input source.

Message Content

The device must publish a message to the /Point85 topic. A collector subscribes to this topic.

The body of the MQTT message is a JSON-serialized EquipmentEventMessage with four fields:

- sourceId (required): the source identifier as defined in the data collection script resolver

- value (required): the data value
- messageType (required): must be “EQUIPMENT_EVENT”
- timestamp (optional): event time in ISO 8601 format

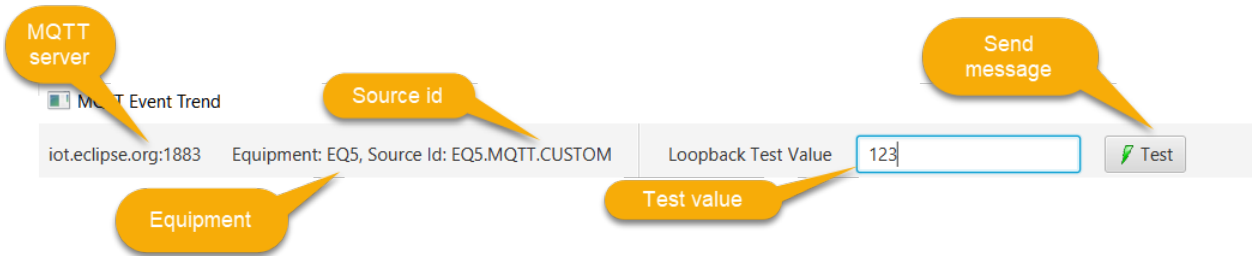
For example:

{"sourceId":"EQ1.MQTT.PROD_REJECT","value":"1","messageType":"EQUIPMENT_EVENT","timestamp":"2019-03-19T11:02:32.240-07:00"}

Trending

By clicking the Watch button for an MQTT equipment resolver, the execution of the script can be observed.

A trend for an MQTT source is similar to the RMQ and JMS messaging trend charta. In this case however, the top portion of the dialog looks like:

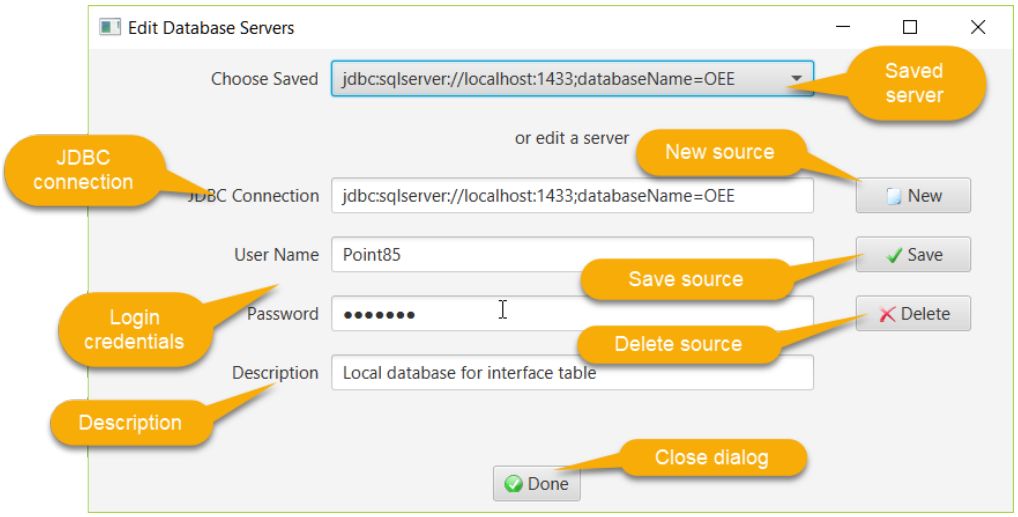


Database Table Data Source

Definition

The database interface table data source definition dialog is launched from the toolbar or from the Data Collection tab for a database table resolver after an equipment object has been selected in the physical model. It is used to define the database server JDBC connection string and login credentials.

This dialog is similar to:



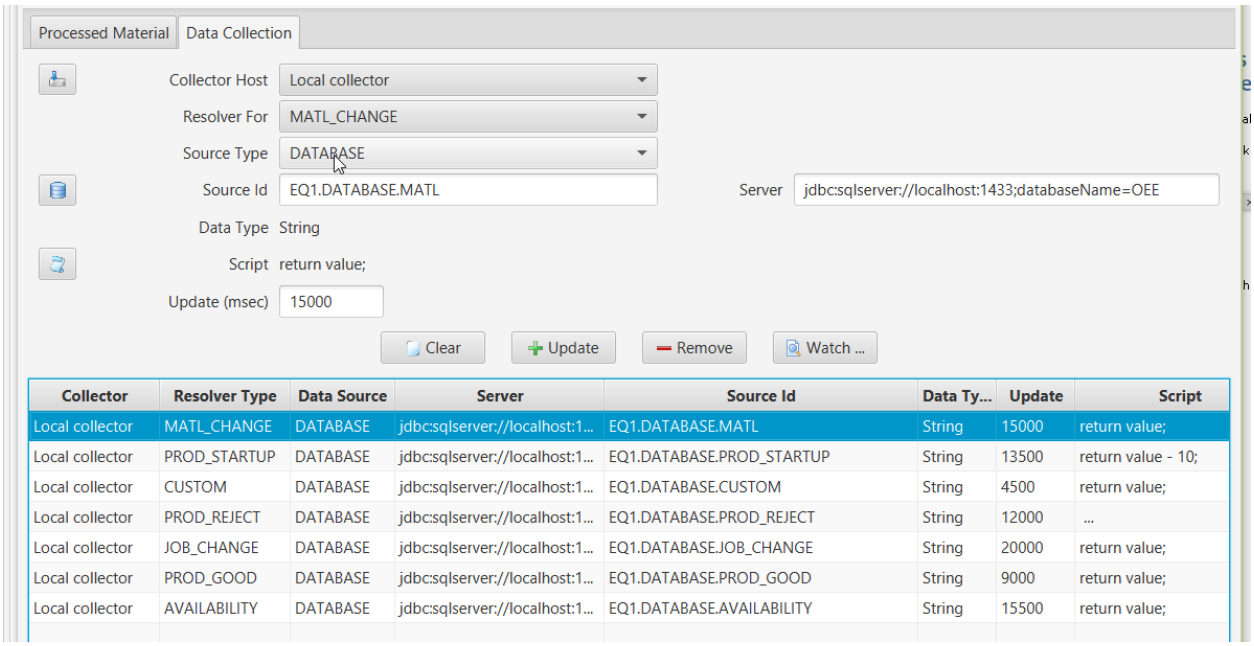
For this example, the SQL Server is running on localhost on the default port of 1433. The client will login as the "Point85" user to the OEE database where the DB_EVENT table has been created.

The actions for a database data source are:

- *New*: clear the editing controls to define a new data source
- *Save*: save the data source to the database
- *Delete*: delete the data source from the database
- *Test*: connect to the data source. A dialog will be presented indicating success or failure

Clicking the Done button will assign the DB_EVENT table as the script resolver's input source.

This table will be queried every N msec where N is the update period defined in the equipment resolver(s). For example, there are 7 resolvers defined for this equipment with a source type of "DATABASE":



Collector	Resolver Type	Data Source	Server	Source Id	Data Ty...	Update	Script
Local collector	MATL_CHANGE	DATABASE	jdbc:sqlserver://localhost:1...	EQ1.DATABASE.MATL	String	15000	return value;
Local collector	PROD_STARTUP	DATABASE	jdbc:sqlserver://localhost:1...	EQ1.DATABASE.PROD_STARTUP	String	13500	return value - 10;
Local collector	CUSTOM	DATABASE	jdbc:sqlserver://localhost:1...	EQ1.DATABASE.CUSTOM	String	4500	return value;
Local collector	PROD_REJECT	DATABASE	jdbc:sqlserver://localhost:1...	EQ1.DATABASE.PROD_REJECT	String	12000	...
Local collector	JOB_CHANGE	DATABASE	jdbc:sqlserver://localhost:1...	EQ1.DATABASE.JOB_CHANGE	String	20000	return value;
Local collector	PROD_GOOD	DATABASE	jdbc:sqlserver://localhost:1...	EQ1.DATABASE.PROD_GOOD	String	9000	return value;
Local collector	AVAILABILITY	DATABASE	jdbc:sqlserver://localhost:1...	EQ1.DATABASE.AVAILABILITY	String	15500	return value;

A material change event will be looked for every 15 seconds and a good production record every 9 seconds.

DB_EVENT Table Schema

The create_event_table.sql script in the \database\mssql folder will create this table for SQL Server. The create_event_table.sql script in the \database\oracle folder will create this table for Oracle. For MySQL the script is in the \database\mysql folder, for PostgresQL in the \database\postgresql folder and for HSQLDB in the \database\hsql folder.

For example, for SQL Server the schema is:

```
CREATE TABLE [dbo].[DB_EVENT](
        [EVENT_KEY] [bigint] IDENTITY(1,1) NOT NULL,
        [SOURCE_ID] [nvarchar](128) NOT NULL,
        [IN_VALUE] [nvarchar](64) NOT NULL,
        [EVENT_TIME] [datetime2](3) NULL,
        [EVENT_TIME_OFFSET] int NULL,
        [STATUS] [nvarchar](16) NOT NULL,
        [ERROR] [nvarchar](512) NULL
) ON [PRIMARY]
```

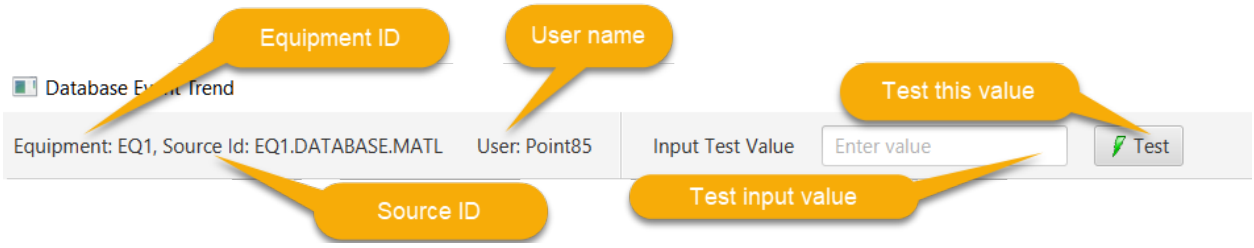
The column values are as follows:

- *EVENT_KEY*: identity for each record inserted into the table.
- *SOURCE_ID*: must match that defined in the associated equipment resolver (e.g. EQ1.DATABASE.MATL for a material change as above).

- *IN_VALUE*: a string value that can be converted to the data type required by the resolver. Availability, material, job and custom inputs are passed directly into the JavaScript resolver as a string. Good, reject and startup amounts are converted to double values.
- *EVENT_TIME*: the local date and time of day when the event occurred (e.g. 2018-11-28 21:30:07.670).
- EVENT_TIME_OFFSET: the time zone offset in seconds from UTC when the event occurred (e.g. -28800)
- STATUS: one of READY, PROCESSING, PASS or FAIL. The record must be inserted with a status of READY. The status is set to PROCESSING before the script is executed. If the JavaScript resolver successfully processes the record, the status will be updated to PASS. If there is an error during processing, the status will be updated to FAIL and the ERROR column will contain text explaining the error. The status can be set back from FAIL to READY in order to retry the event. It is the responsibility of the orginator to delete PASSed or FAILed records when they are no longer of any interest.
- ERROR: this column will contain text explaining the error if the record was not successfully processed.

Trending

By clicking the Watch button for an database table resolver, the execution of the script can be observed. A trend for a database source is similar to the other trend charts. In this case however, the top portion of the dialog looks like:



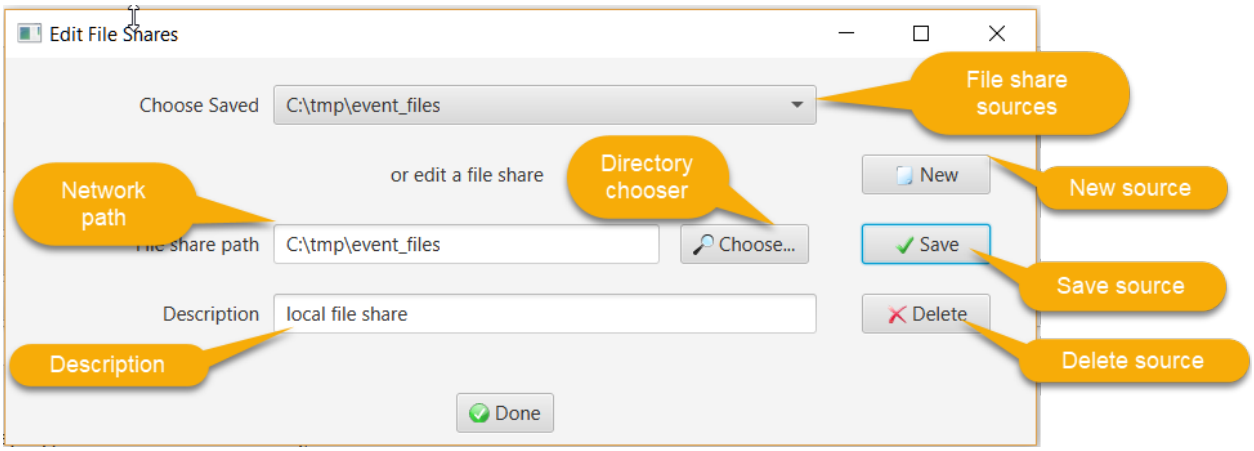
Entering a value, then clicking the Test button will write a record into the DB_EVENT table with the current system time. It will be read on the next polling cycle and processed. A successful execution of the script resolver will display the point in the trend chart. A failure will result in an error dialog being displayed and a status of FAIL in the table.

File Share Data Source

Definition

The file share data source definition dialog is launched from the toolbar or from the Data Collection tab for a file share resolver after an equipment object has been selected in the physical model. It is used to define the network path to the file share server where the event files are stored.

This dialog looks like:



For this example, the root folder is called “events” on the C: drive. Subfolders must be named the same as the source identifier in the equipment resolver. Under the source id folder are folders containing the event files and are named “ready”, “processing”, “pass” and “fail” as explained below. Read/write permissions must be granted to these folders for the Designer application or standalone collector.

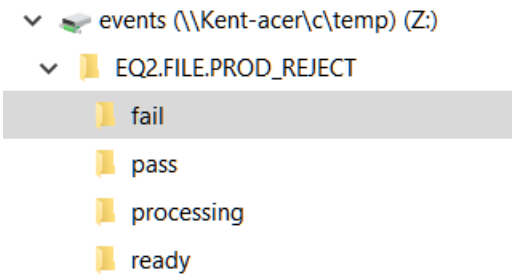
The actions for a file share source are:

- *New*: clear the editing controls to define a new data source
- *Save*: save the data source to the database
- *Delete*: delete the data source from the database

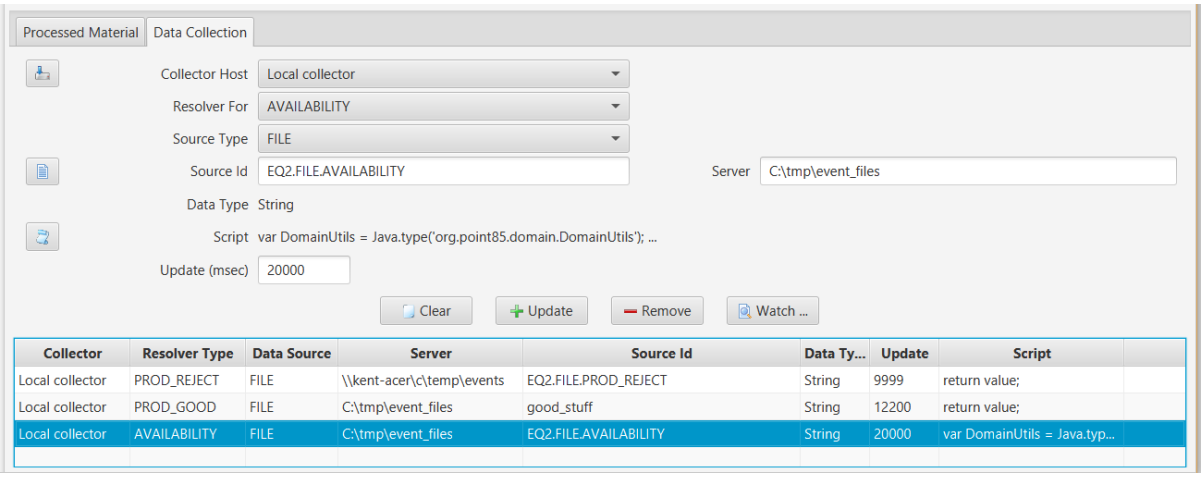
Clicking the Done button will assign the a file event source named the same as the share network path.

Directory Structure

The folder structure for the file share source defined above looks and equipment with a source id of “EQ2.FILE.PROD_REJECT” is:



The “ready” sub-folder will be queried every N msec where N is the update period defined in the equipment resolver(s). For example, there are three resolvers defined for this equipment with a source type of “FILE”:



On the remote server, reject production event files will be looked for every 9999 milli-seconds. On a local file share, good production files will be polled for every 12.2 seconds and availability files every 20 seconds.

When a new file is found in the "ready" folder, the file is read and its contents input as a string to the equipment resolver's JavaScript function as the "value" variable. The script must parse this string and return a value consistent with the resolver's type (e.g. for availability, this is the name of a reason and for good, reject or startup a production count).

The JavaScript can optionally set the event timestamp on the equipment resolver. For example, this script obtains the timestamp as an ISO 8601 string and then sets into the resolver:

```
resolver.setIsoTimestamp("2019-02-24T10:10:10.000-08:00");
```

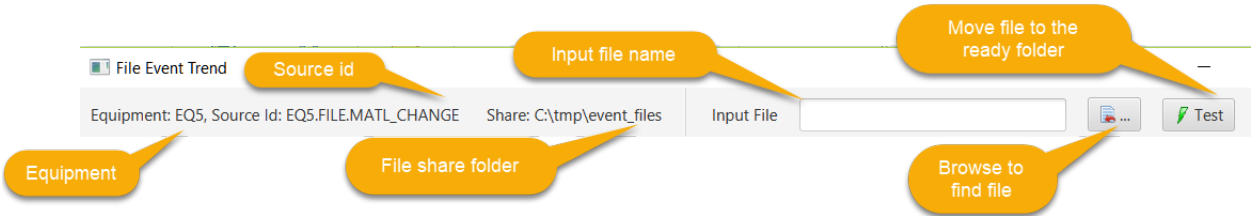
Files in the "ready" folder are processed oldest first according to the file's last modified time. If the resolver's timestamp is set, then the time set by the script will be used instead of the last modified time.

Before the equipment resolver's JavaScript is invoked, the file is moved to the "processing" folder. If the script execution completes successfully, the file is moved to the "pass" folder. If script execution fails, the file is moved to the "fail" folder. In addition a text file with the same name as the event file, but with an ".error" extension will contain information about the exception.

The application writing the event files is responsible for deleting files in the "pass" and "fail" folders when they are no longer of any interest.

Trending

By clicking the Watch button for an file resolver, the execution of the script can be observed. A trend for a file source is similar to the other trend charts. In this case however, the top portion of the dialog looks like:



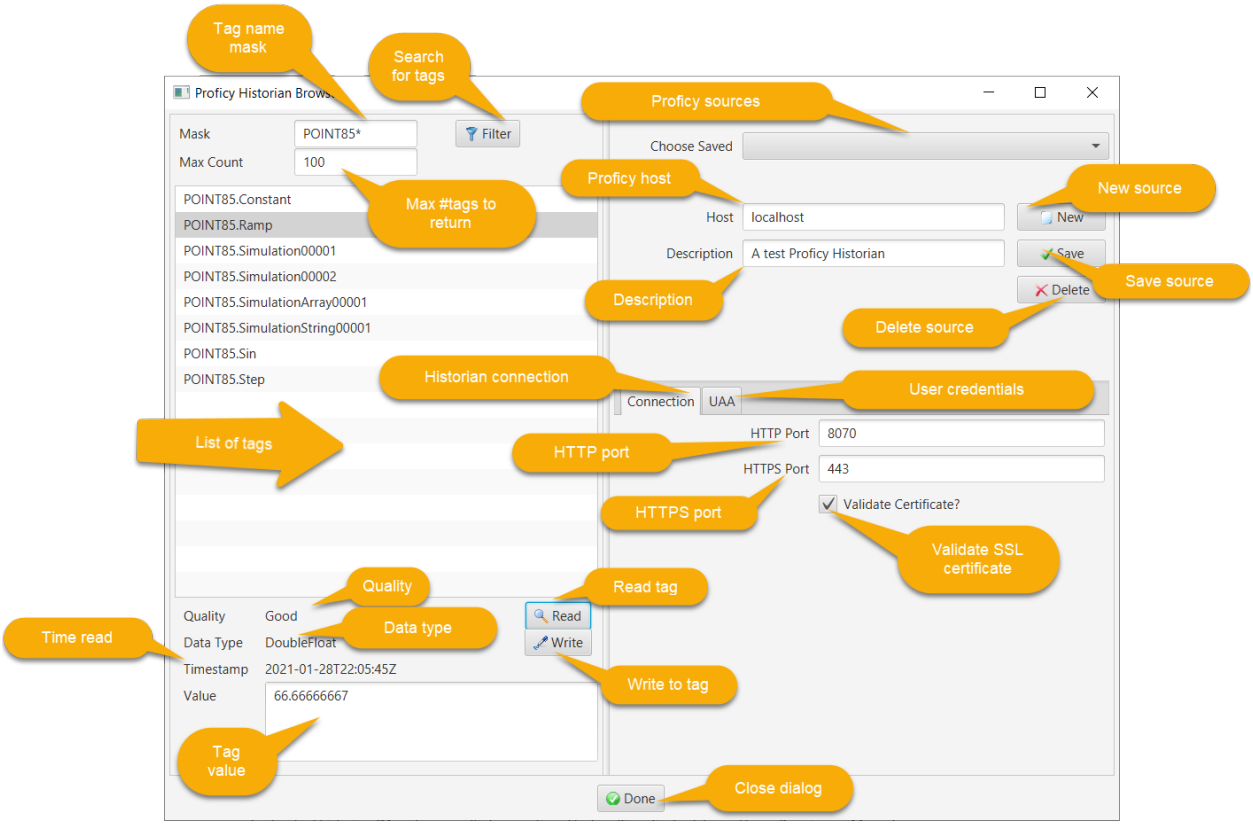
Browsing to the event file, then clicking the Test button will write it to the "ready" folder under a folder named the same as the source identifer (EQ5.FILE.MATL_CHANGE in this example). The file will be read on the next polling cycle and processed. A successful execution of the script resolver will display the point in the trend chart. A failure will result in an error dialog being displayed and files in the "fail" folder.

Proficy Data Source

Definition

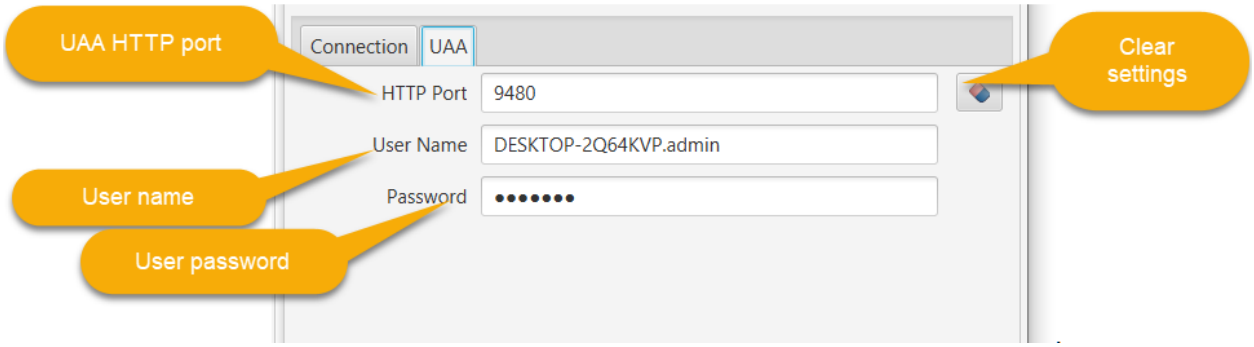
The Proficy data source definition dialog is launched from the toolbar or from the Data Collection tab for a Proficy resolver after an equipment object has been selected in the physical model. It is used to browse to a tag to be monitored for data changes. The tag is polled periodically at the defined frequency, or more frequently if multiple tags are being monitored.

This dialog looks similar to:



For this example, an HTTPS connection is used to a Proficy Historian running on the "localhost" host on port 443. If the HTTPS port is not specified, then the HTTP will be used. If the "Validate Certificate?" checkbox is checked for HTTPS, then the client's SSL certificate will be sent to the Historian and validated.

In order to obtain a valid Oauth bearer token for REST GET and POST calls, the user must be authenticated with Proficy UAA (User Authentication and Authorization) settings:



HTTP port, user name and password are all required.

The actions for a Proficy Historian source are:

- *New*: clear the editing controls to define a new data source
- *Save*: save the data source to the database
- *Delete*: delete the data source from the database
- *Backup*: save the selected data source (or all sources if non is selected) to a .p85x file

Clicking the Done button will assign this data source to a resolver if launched from the data collection tab.

The Proficy connection can be tested by reading data from or writing data to a tag. First, define the tag name mask for searching for tags and the number to return. A blank name mask will return all tags up to the number specified. In the screen shot above, there are 8 tag names starting with "POINT85".

Selecting a tag will display the data quality (bad, good, uncertain or not available), data type, time collected in ISO 8601 format ("Z" is UTC time) and current value.

The actions are:

- Read: read the tag and display its value in the "Value" text area
- Write: write the value in the "Value" text area to the tag. Arrays are not supported.

Event Resolver

The equipment event resolver is similar to the other data sources. Like OPC DA and OPC UA, the source id is the tag name and cannot be changed. For example, this screen shot shows three Proficy tag sources for production counts:

All three will be read every 5 seconds. The “value” argument in the resolution script is a list of ModbusVariants. See the “Scripting” section 9 for example scripts.

In the resolver script, note that a handshake should be implemented between the Modbus master and data collector to reset the data after a successful read for example by writing “true” to a coil (see the “Scripting” section 9). Since production values are always positive, the variant value returned from a read operation should be checked for being a positive number, e.g. value.get(0).isPositiveNumber(). If the value is not valid, the script can return null. For availability reason code, material id and job id, the last value can be checked to see if the new polled value has changed by calling resolver.getLastValue(). If it has not changed, the script should return null.

Trending

By clicking the Watch button for a Proficy resolver, the execution of the script can be observed. A trend for a Proficy tag source is similar to the other trend charts. In this case however, the top portion of the dialog looks like:

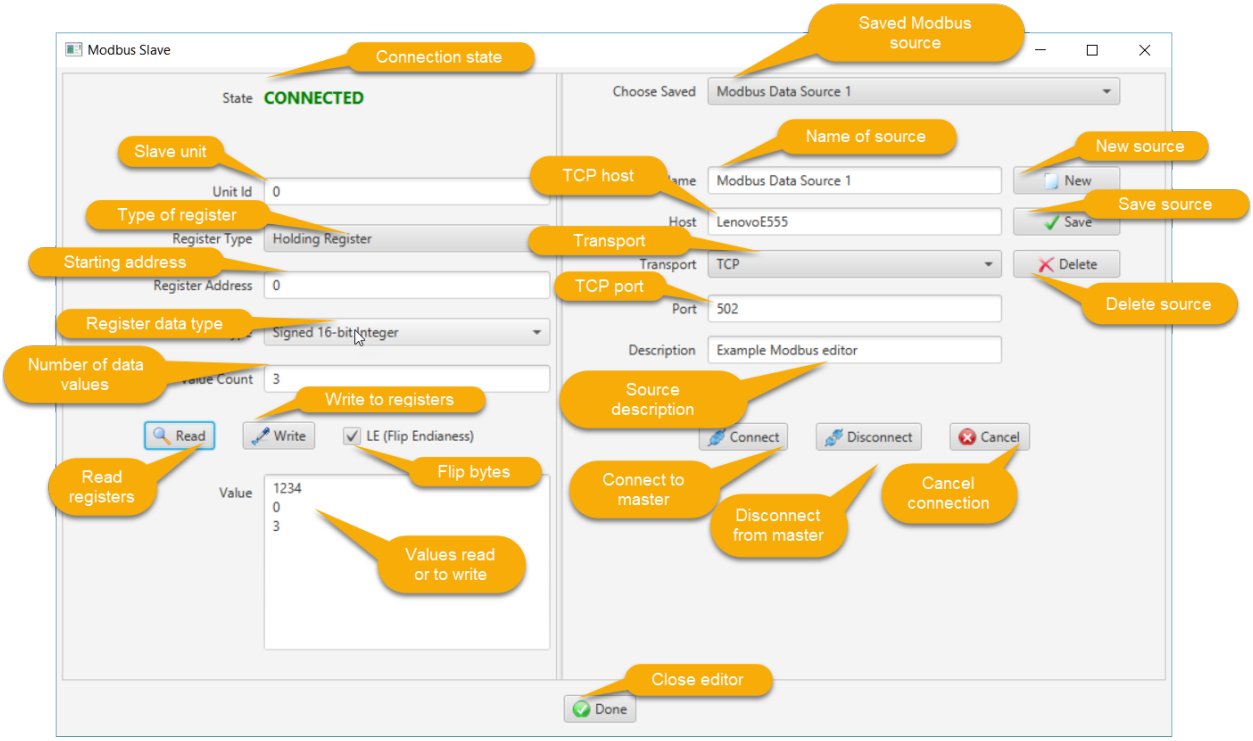
The trend chart will be updated when good new values are found on a polling cycle.

Modbus Slave Data Source

Definition

The Modbus data source definition dialog is launched from the toolbar or from the Data Collection tab for a Modbus resolver after an equipment object has been selected in the physical model. It is used to define a slave register’s address and data type where the value is read from. The master polls the slave periodically at the defined frequency.

This dialog looks like:



For this example, a TCP connection is used to a Modbus master running on the "LenovoE555" host on port 502. The transport choices are:

- TCP: TCP connection
- UDP: datagram
- Serial: serial connection with 9600 baud, 8 data bits, 1 stop bit and no parity

The actions for a Modbus source are:

- *New*: clear the editing controls to define a new data source
- *Save*: save the data source to the database
- *Delete*: delete the data source from the database
- *Backup*: save the selected data source (or all sources if non is selected) to a .p85x file
- *Connect*: connect to the data source. The connection status is shown in the left-hand panel.
- *Disconnect*: disconnect from the data source
- *Cancel*: cancel an in-progress connection

Clicking the Done button will assign this data source to a resolver if launched from the data collection tab.

The Modbus connection can be tested by reading data from or writing data to a modbus slave serviced by this master. The information is:

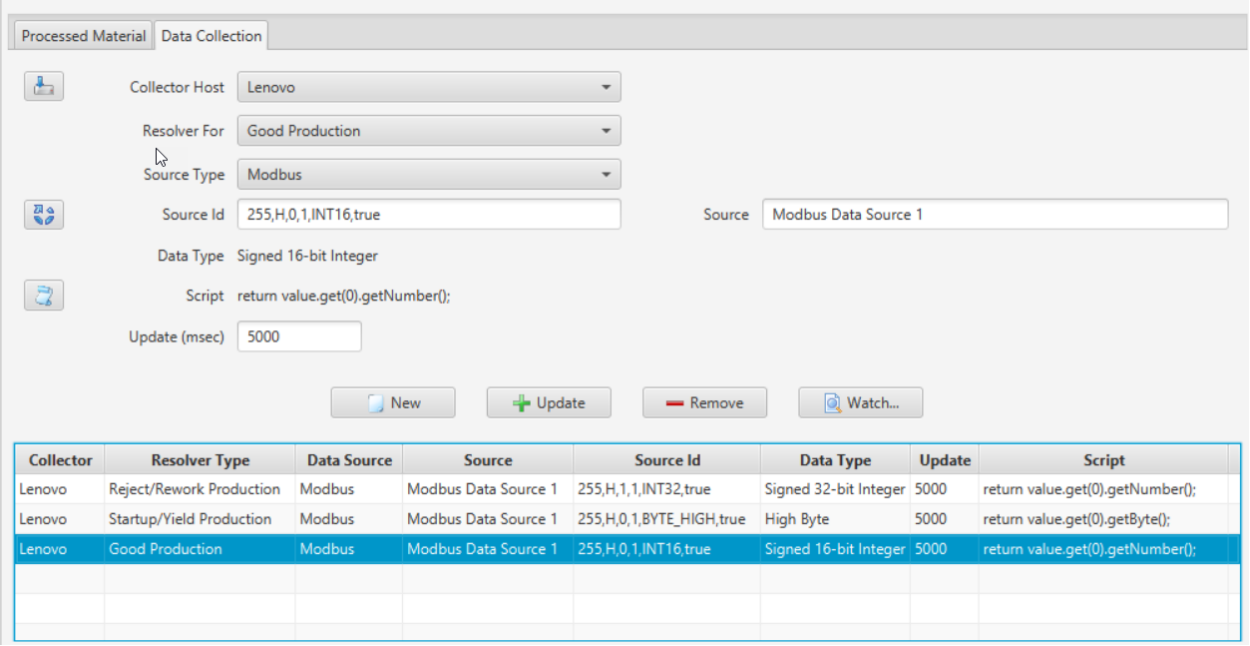
- Unit Id: the identifier of the slave (0 -255, default is 0)
- Register Type: the type of register to read or write data:
    - Coil - R/W boolean value
    - Discrete - RO discrete value
    - Holding Register - R/W data register
    - Input Register - RO data register
- Register Address: starting register for read/write operations. If a data value exceeds one word (16 bits), sequential registers will be used
- Data Type: the type of data for read/write operations
    - Discrete (1 bit)
    - Low Byte (1 byte)
    - High Byte (1 byte)
    - Signed 16-bit integer (2 bytes)
    - Unsigned 16-bit integer (4 bytes)
    - Signed 32-bit integer (4 bytes)
    - Unsigned 32-bit integer (8 bytes)
    - Signed 64-bit integer (8 bytes)
    - Single precision float (4 bytes)
    - Double precision float (8 bytes)
    - String (UTF8, 1 byte per character)
- Value Count: number of values to read. For a string, this is the character (byte) count.
- LE (Flip Endianness) : reverse the bytes in the data value for the case where the Modbus slave and collector machines have different byte orders, i.e. little endianess (LE) vs. big endianess (BE).

The actions are:

- Read: read the data values from the starting register and display them as decimals or a string in the “Value” text area
- Write: write the list of data values in the “Value” text area (one per line) to the starting register. Only one string is supported.

Event Resolver

The equipment event resolver is similar to the other data sources. Like OPC DA and OPC UA, the source id is a generated value and cannot be changed. The source id is a comma-separated list of slave endpoint information. For example, this screen shot shows three Modbus sources for production counts:

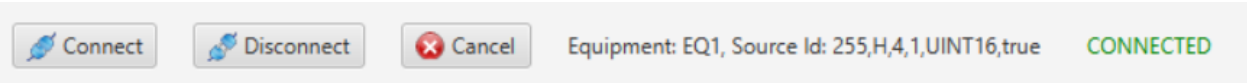


All three will be read every 5 seconds. The "value" argument in the resolution script is a list of ModbusVariants. See the "Scripting" section 9 for example scripts.

In the resolver script, note that a handshake should be implemented between the Modbus master and data collector to reset the data after a successful read for example by writing "true" to a coil (see the "Scripting" section 9). Since production values are always positive, the variant value returned from a read operation should be checked for being a positive number, e.g. value.get(0).isPositiveNumber(). If the value is not valid, the script can return null. For availability reason code, material id and job id, the last value can be checked to see if the new polled value has changed by calling resolver.getLastValue(). If it has not changed, the script should return null.

Trending

By clicking the Watch button for a Modbus resolver, the execution of the script can be observed. A trend for a Modbus slave source is similar to the other trend charts. In this case however, the top portion of the dialog looks like:

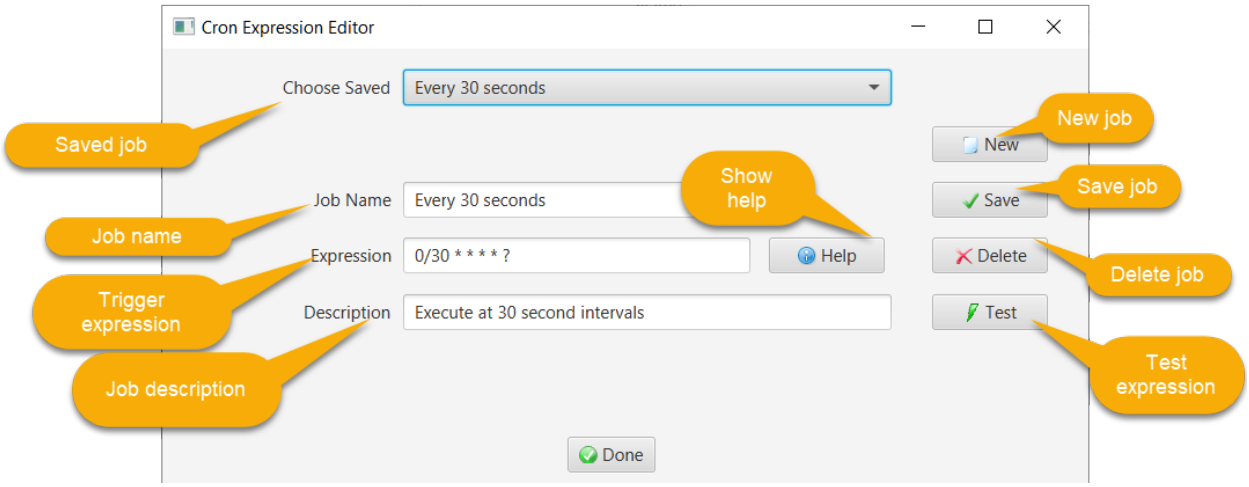


Cron Job Data Source

Definition

The cron job data source definition dialog is launched from the toolbar or from the Data Collection tab for a cron job resolver after an equipment object has been selected in the physical model. It is used to define the cron expression that is triggered by the Quartz job scheduler.

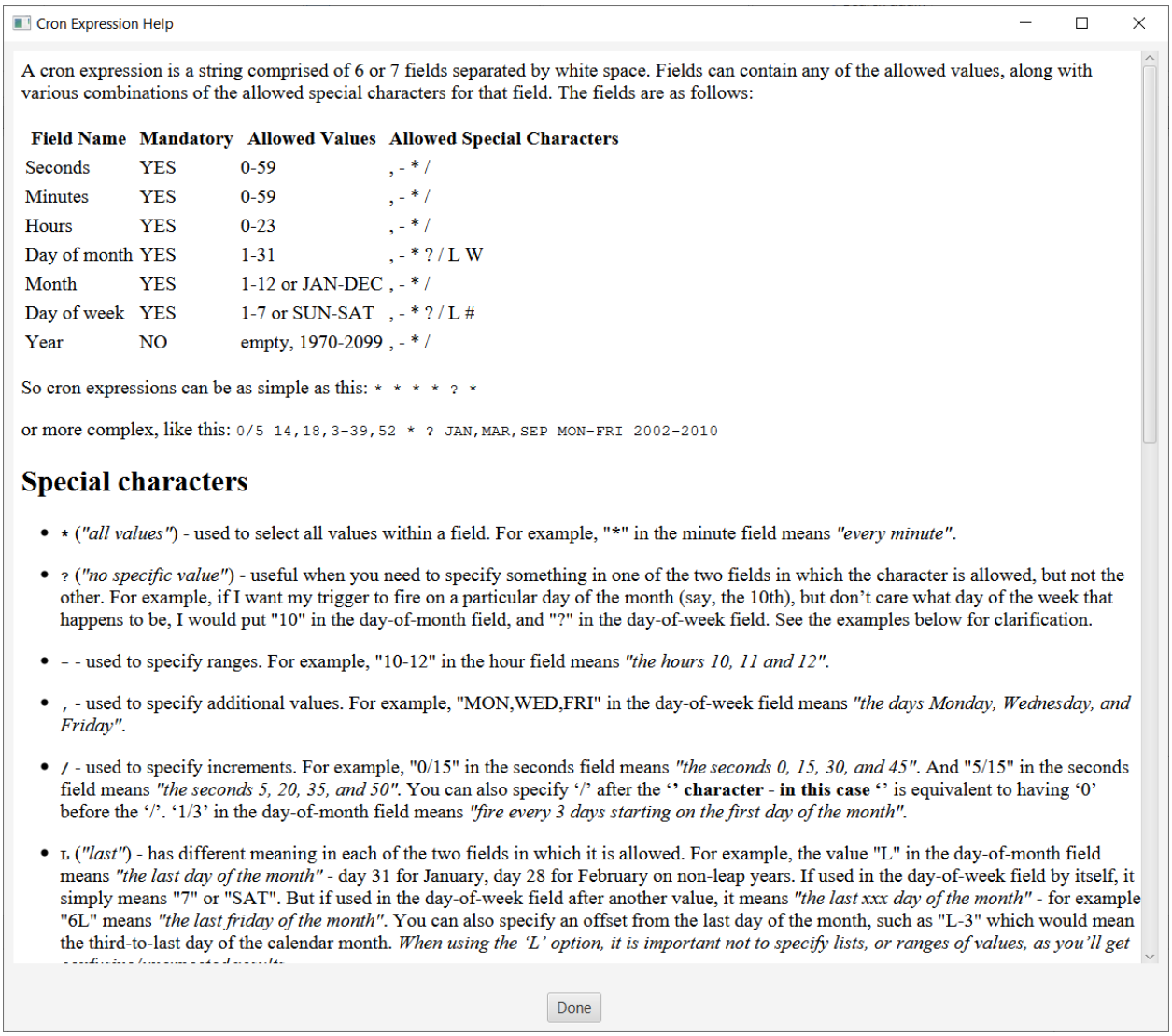
This dialog looks like:



The attributes of a cron event source are:

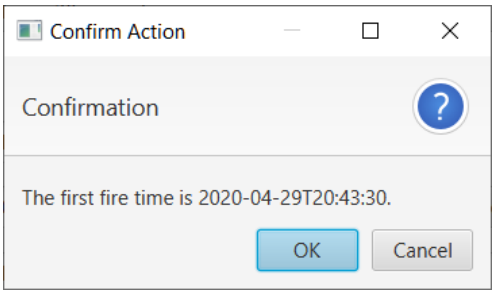
- Job Name: name of the job
- Expression: cron expression
- Description: description of the job

Help on composing the cron expression is available by clicking the Help button. A dialog is displayed that explains the cron expression fields. For example:



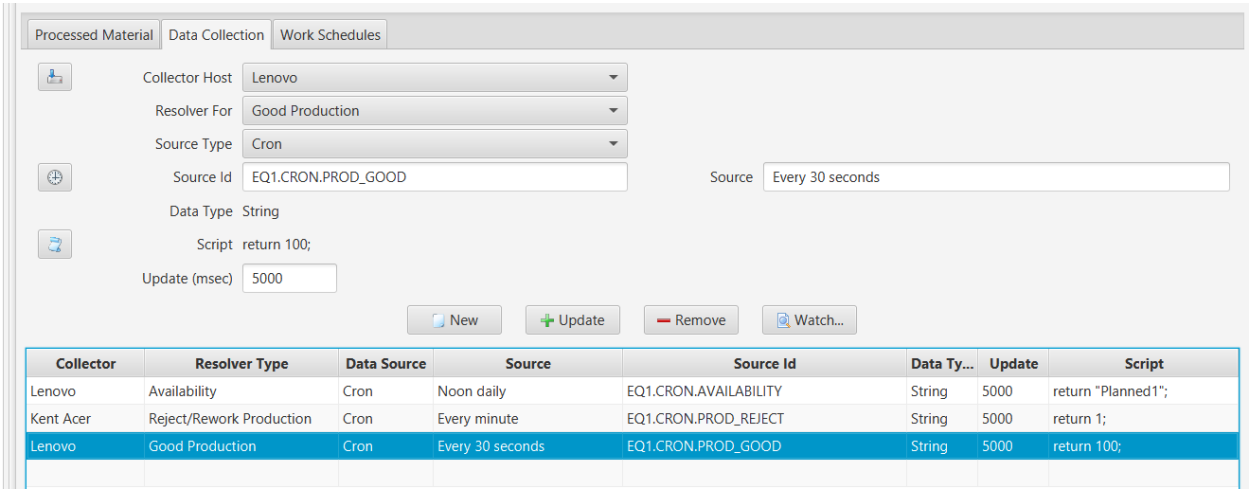
The actions for a cron event source are:

- *New*: clear the editing controls to define a new data source
- *Save*: save the data source to the database
- *Delete*: delete the data source from the database
- *Backup*: save the selected data source (or all sources if non is selected) to a .p85x file
- Test: execute the cron expression and display the next fire time, for example:



Event Resolver

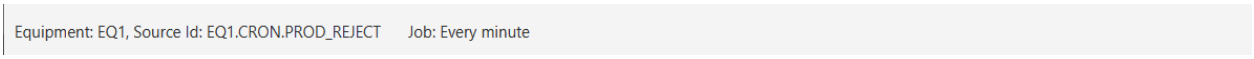
The equipment event resolver is similar to the other data sources:



In this example, the source type is selected as “Cron” for good production count that is collected every 30 seconds (note that the update period is not applicable). The test script just returns a count of 100.

Trending

By clicking the Watch button for a cron job resolver, the execution of the script can be observed. A trend for a cron event source is similar to the other trend charts. In this case however, the top portion of the dialog looks like:

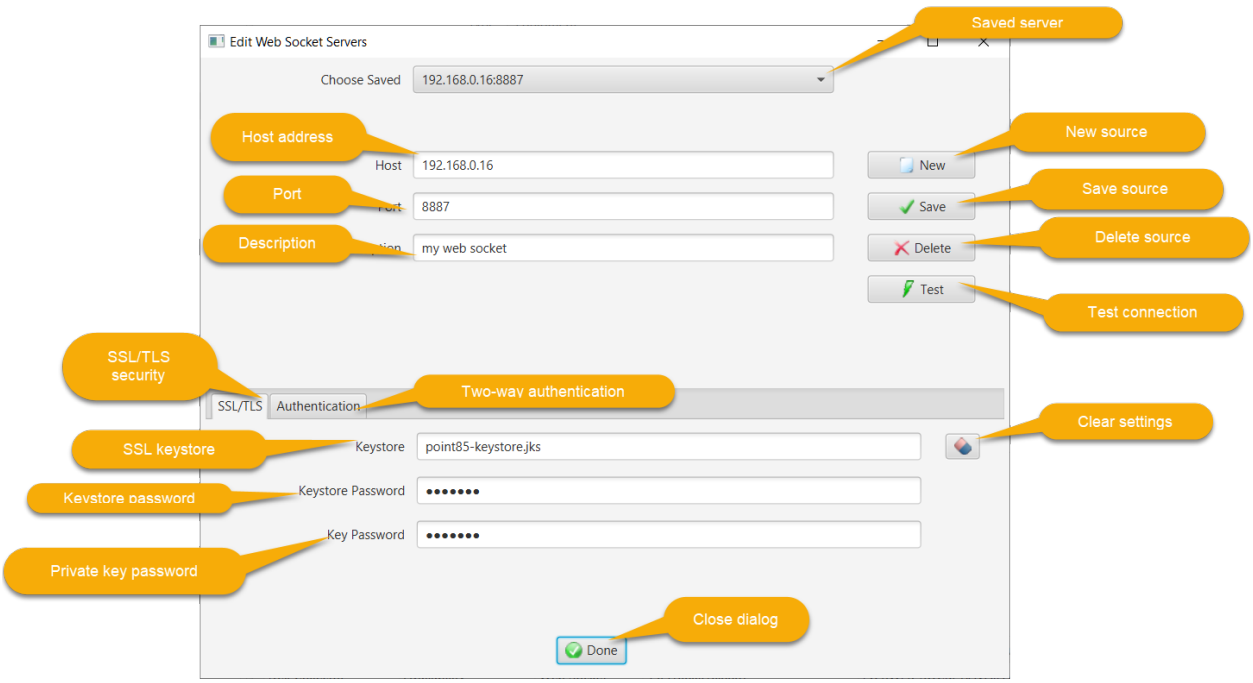


Web Socket Data Source

Definition

The web socket data source definition dialog is launched from the toolbar or from the Data Collection tab for a web socket resolver after an equipment object has been selected in the physical model. It is used to define the web socket server host, port and security credentials.

This dialog is similar to:



For this example, the web socket server is running on the host at address 192.168.0.16 on the SSL port of 8887 and is using SSL/TLS security. The SSL keystore file is named "point85-keystore.jks" and must be located in the \config\security folder. The SSL store has a password.

The second tab allows enabling of two-way SSL authentication:

If checked, then the client must present a valid SSL certificate to the server.

All security settings are optional.

The actions for a web socket data source are:

- *New*: clear the editing controls to define a new data source
- *Save*: save the data source to the database
- *Delete*: delete the data source from the database
- *Backup*: save the selected data source (or all sources if non is selected) to a .p85x file
- *Test*: connect to the data source. A dialog will be presented indicating success or failure.

Clicking the Done button will assign the web socket server as the script resolver's input source if the dialog is launched from the plant entity's data collection tab.

Message Content

The web socket message is a JSON-serialized EquipmentEventMessage with four fields:

- sourceId (required): the source identifier as defined in the data collection script resolver
- value (required): the data value
- messageType (required): must be “EQUIPMENT_EVENT”
- timestamp (optional): event time in ISO 8601 format

For example:

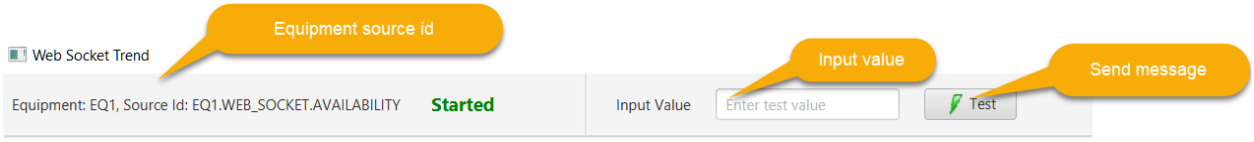
{"sourceId":"EQ1.WEB_SOCKET.PROD_GOOD","value":"1","messageType":"EQUIPMENT_EVENT","timestamp":"2021-11-19T11:13:45.977-07:00"}

The web socket message key is a string identifying the type of the message. Event messages are sent by a web socket client application.

Trending

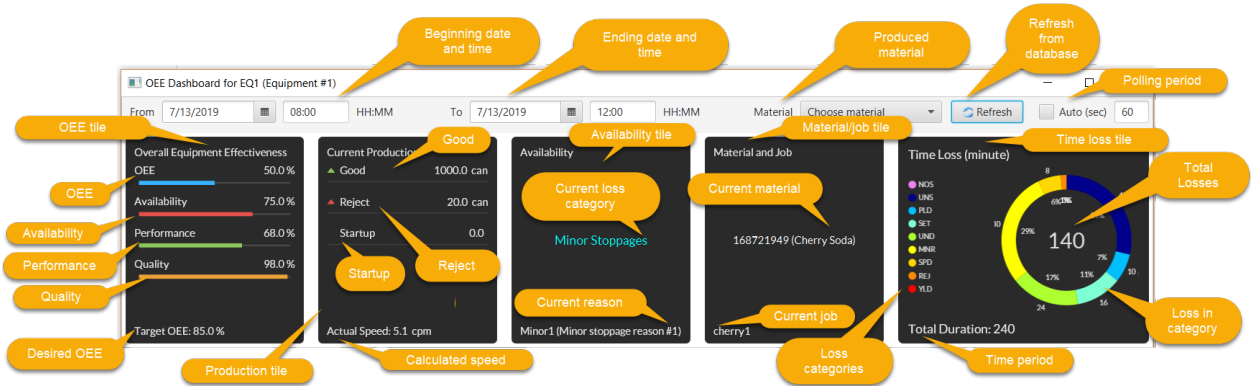
By clicking the Watch button for an web socket resolver, the execution of the script can be observed.

A trend for a web socket source is similar to the RMQ messaging trend chart. In this case however, the top portion of the dialog looks like:



DASHBOARD

After equipment is selected, the dashboard can be displayed. This form consists of five tiles at the top that display OEE, availability, production counts, material and job and time losses:



The bottom portion has tabs for event history, time losses by category and various Pareto charts.

A time range is first selected from a starting date and time of day to and ending date and time of day. In the above example, the time period is 4 hours (240 minutes) during a single day. A specific material that

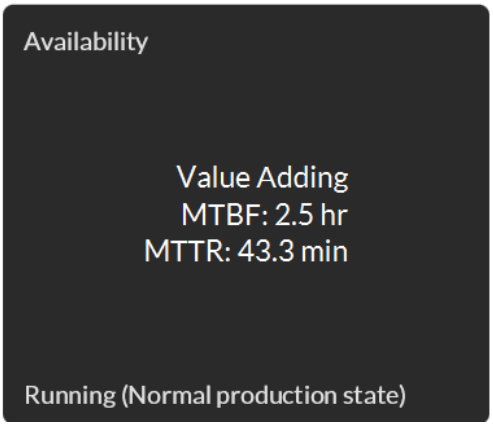
has equipment setups can be selected in the combobox, or all materials with setups during the specified time period. Clicking the refresh button will query the event records from the database and compute the OEE with its three components. In this case, the OEE is 50% where availability = 75%, performance = 68% and quality = 98%. The paint can fillter machine has a desired OEE of 85%. If the "Auto" checkbox is checked, the form will refresh with a period in seconds as entered in the text box to the right of the checkbox. The default is every 60 seconds.

The current production tile displays the cumulative good (1000 cans), reject (20 cans) and startup counts (0 cans) from the 8 event records.

The availability tile displays the last availability event (a Minor Stoppages loss category) with a reason of "Minor 1" and description. Note that in this example, the availability events were entered in summary form (where the event duration includes multiple events for the same reason) over this 4 hour period. The events can also be entered as they occurred in chronological sequence.

If availability events are entered as they occur (vs. in summary form), the Mean Time Between Failures (MTBF) and Mean Time to Repair (MTTR) will be shown in the Availability tile below the loss category. MTBF (in hours) is calculated as the average time between successive unplanned downtime events. MTTR (in minutes) is calculated as the average time between the first availability event after the failure that is not unplanned.

For example this equipment is running normally, but has experienced unplanned downtime



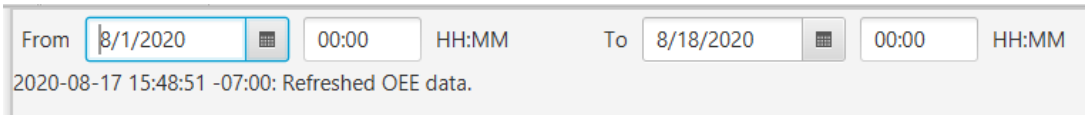
The material and job tile displays the current material being run ("Cherry Soda") and job name ("cherry1").

The time loss tile displays the time loss in each of the 9 categories:

- Not scheduled
- Unscheduled
- Planned downtime
- Setup

- Unplanned downtime
- Minor stoppages
- Speed
- Rejects and rework
- Start-up and yield

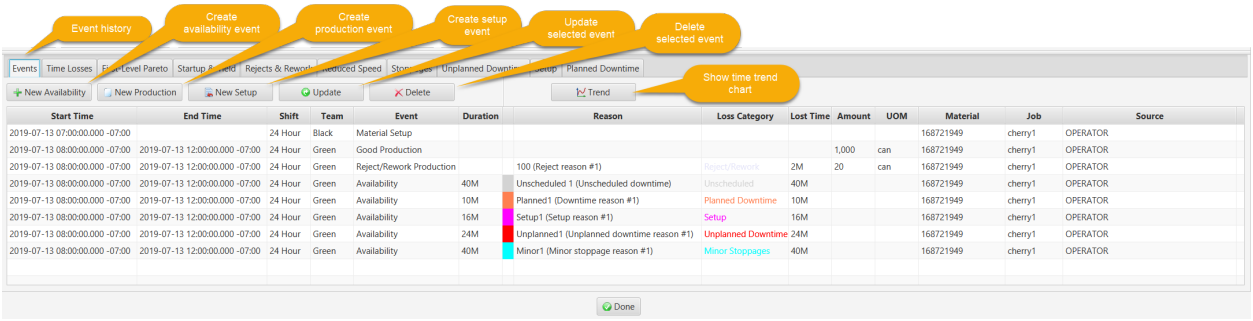
When auto refresh is turned on, informational messages (as well as any errors) will be displayed below the top bar. For example:



Below the tiles the following tabs have more detailed information.

Events

This tab displays the event history in a table:



Note that a job change without a corresponding material change is not shown since it does not affect the OEE calculation. The starting and ending date and times of day are displayed. Current events do not have an ending time (e.g. the setup that changed to the current material and job).

If a work schedule is defined for this equipment, the shift and team working that shift will be shown. The duration of an event is the availability time period or the total time period for production events. Therefore, for availability events, the lost time is equal to the duration. But, for reject or startup production, the counts are converted into the equivalent time period as per the OEE time loss model.

The availability reason is shown and its color-coded loss category. Production events have the amount and unit of measure. The last table columns show the material and job when the event ocurred.

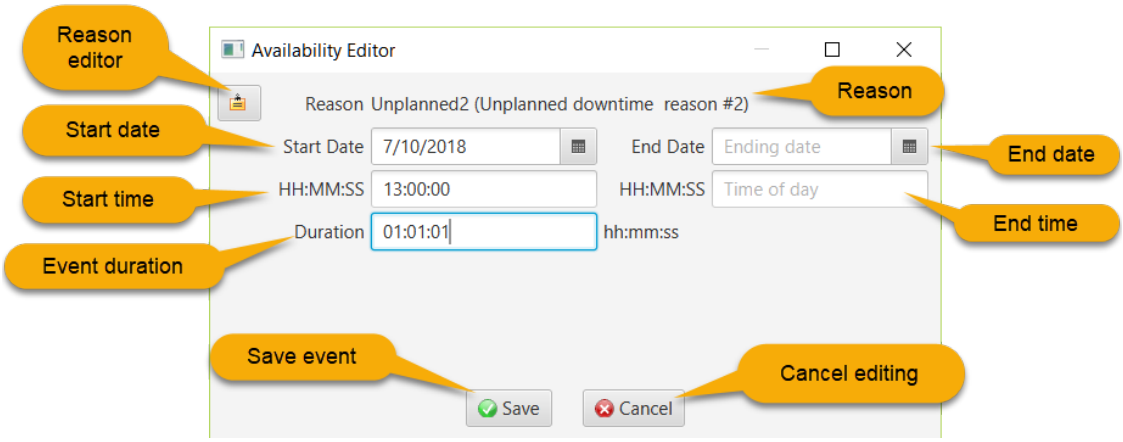
The following actions are available:

- *New Availability*: create an availability record. The dialog discussed below will be displayed to enter the event information.

- *New Production*: create a production record. The dialog discussed below will be displayed to enter the event information.
- *New Setup*: create a set up record. The dialog discussed below will be displayed to enter the event information.
- *Update*: edit an existing availability, production or setup record. The dialog discussed below will be displayed to enter the event information.
- *Delete*: delete an existing set up record.
- *Trend*: display a line chart of production and availability events for this time period.

Availability Editor

To create an availability event or to edit an existing event, the following dialog is used:



If an existing event is being edited, the fields will be populated with current data.

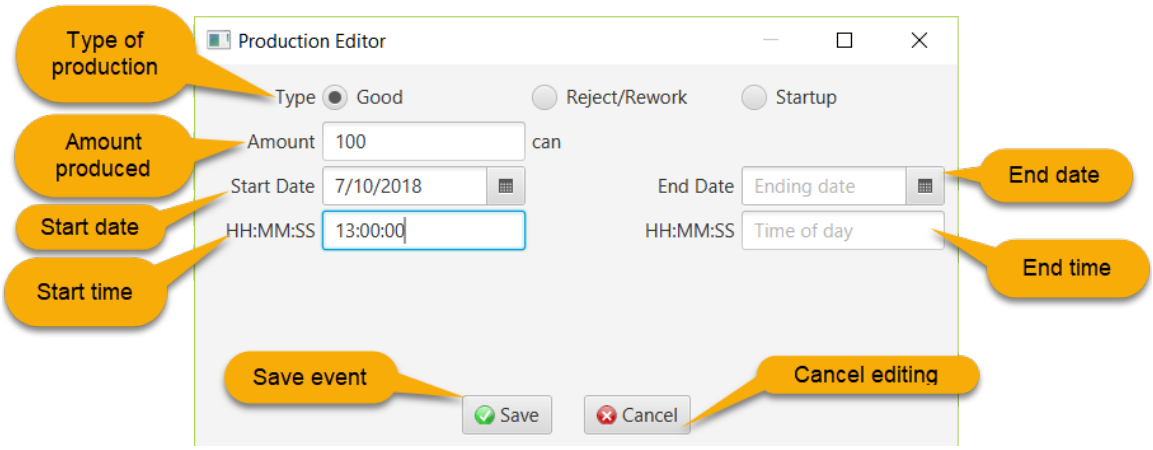
A reason must be chosen by clicking on the reason editor button and selecting an existing reason or creating a new reason. A starting date and time of day is required as is the duration of the event.

An ending time is not required if the events are being entered in chronological order as they happened. On the other hand, if the event is being summarized over a period of time, the ending date and time of day is required.

Clicking the Save button inserts a record into the database. Cancel will exit the editor without making any changes.

Production Editor

To create a production event or to edit an existing event, the following dialog is used:



If an existing event is being edited, the fields will be populated with current data.

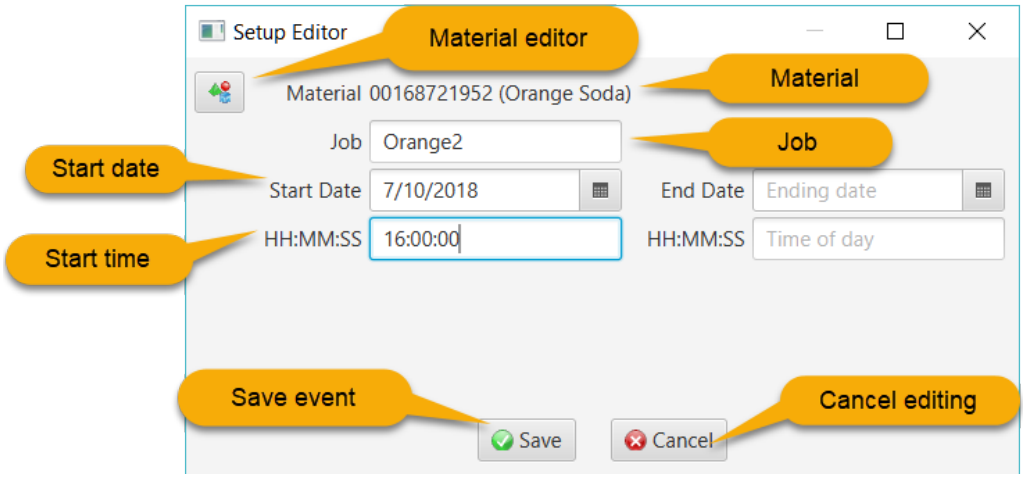
A type of production must be selected in the radio buttons and the amount produced. The unit of measure is obtained from the UOM configured for the equipment.

An ending time is not required if the events are being entered in chronological order as they happened. On the other hand, if the event is being summarized over a period of time, the ending date and time of day is required.

Clicking the Save button inserts a record into the database. Cancel will exit the editor without making any changes.

Setup Editor

To create a setup event or to edit an existing event, the following dialog is used:



If an existing event is being edited, the fields will be populated with current data.

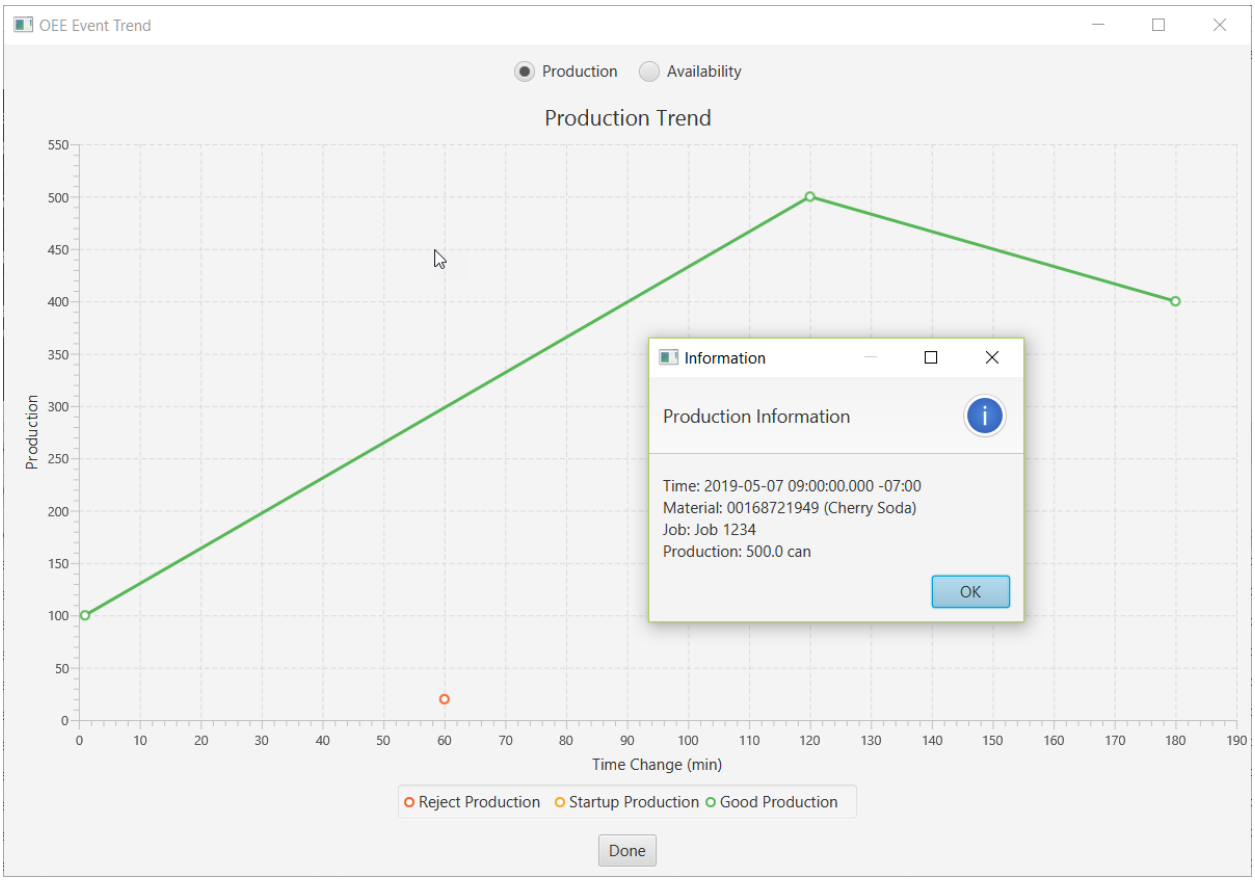
The material must be chosen by clicking on the material editor button and selecting an existing material or creating a new material. The name of a job/order is optional. A starting date and time of day is required.

An ending time is not required if the events are being entered in chronological order as they happened. On the other hand, if the event is being summarized over a period of time, the ending date and time of day is required. Note that there must be at least one open setup event since OEE data is dependent upon knowing what material is being produced.

Clicking the Save button inserts a record into the database. Cancel will exit the editor without making any changes.

Trend

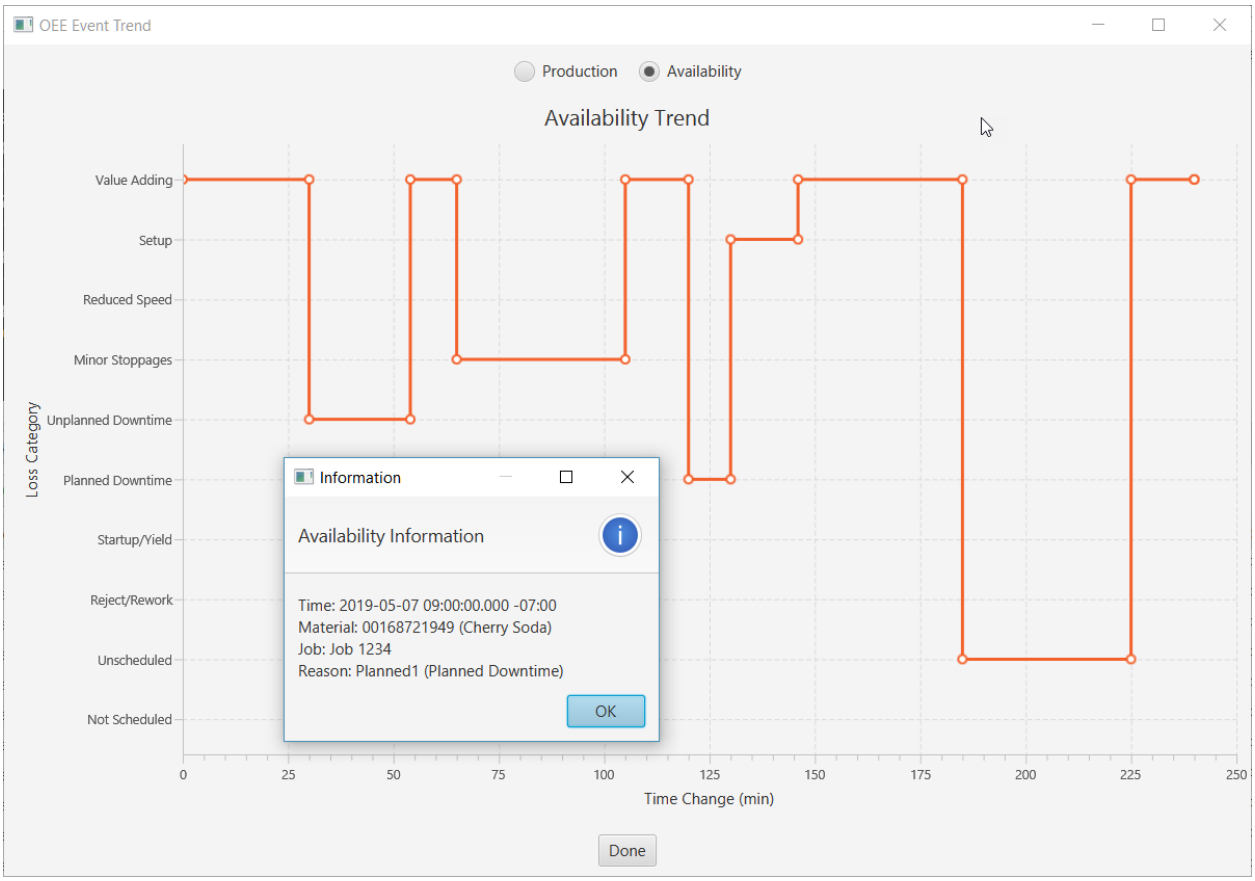
Clicking the Trend button displays a dialog with two radio buttons for showing a line chart of either (1) good, reject/rework and startup/yield production events, or (2) availability events. For example, a production chart looks like:



The x-axis is the time in minutes from the start of the time period of interest until the last production event (3 hours later in this example). The y-axis is the produced amount in the unit of measure defined as the numerator of the design speed for this equipment ("can" in this example). If the reject quantity's UOM differs from the design speed UOM, it will be converted.

Clicking on a point displays a dialog with more information about the event: exact time, material being produced, the job identifier and the quantity.

For example, an availability chart looks like:

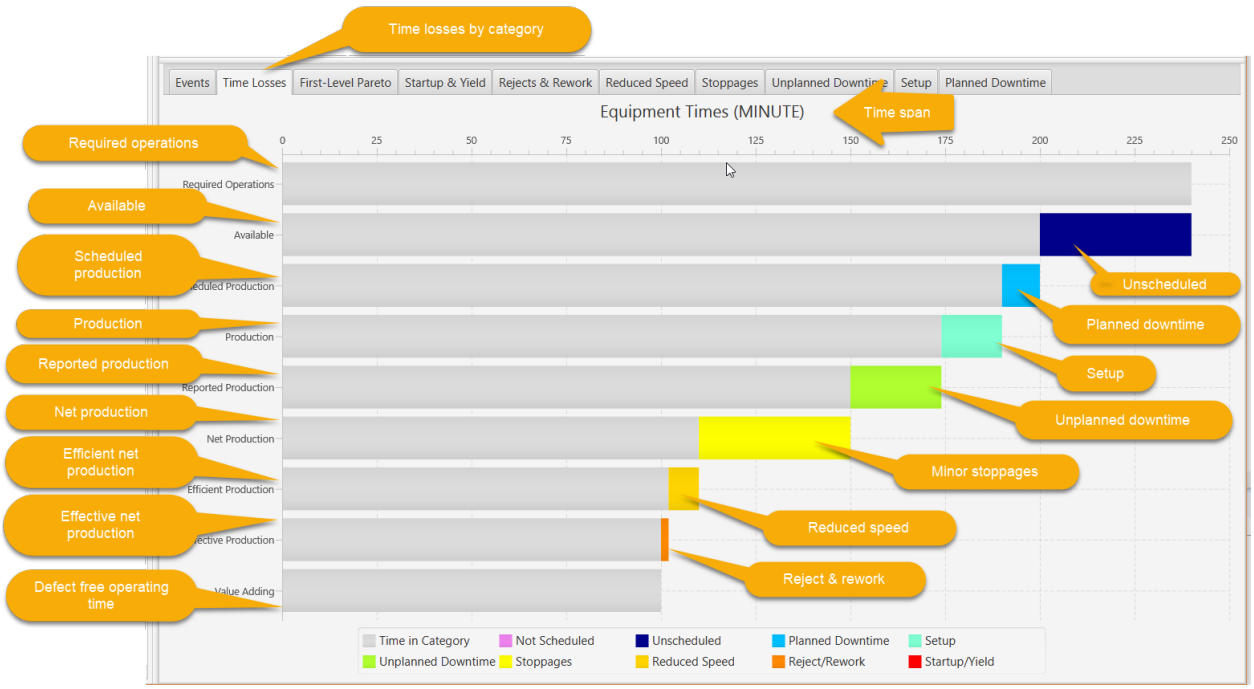


The x-axis is the time in minutes from the start of the time period of interest until the end of the period (4 hours in this example). The y-axis is the loss category, or "Value Adding" if there is no loss.

Clicking on a point displays a dialog with more information about the event: exact time, material being produced, the job identifier and the reason for the event.

Time Losses

The time losses tab displays a bar chart of of the losses encountered before arriving at the net defect free or value adding time (see [Kennedy]).



The bars represent:

- *Required Operations*: the time period of interest (4 hours or 240 minutes in this example) minus the Not Scheduled time (also 4 hours since there is no Not Scheduled time)
- *Available*: the required production time that subtracts the Unscheduled time (40 minutes)
- *Scheduled Production*: available time less Planned Downtime (10 minutes)
- *Production*: scheduled production time less the Setup time (16 minutes)
- *Reported Production*: production time less the Unplanned Downtime time (24 minutes)
- *Net Production*: reported production time less the Minor Stoppages time (40 minutes)
- *Efficient Net Production*: net production time less the Reduced Speed time (8 minutes)
- *Effective Net Production*: effecient net production time less the Reject/Rework time (2 minutes)
- *Value Adding, Defect Free or Ideal Speed*: effective net production time less the Startup & Yield time (0 minutes)

Hovering the mouse over a bar chart segment displays the time in the category.

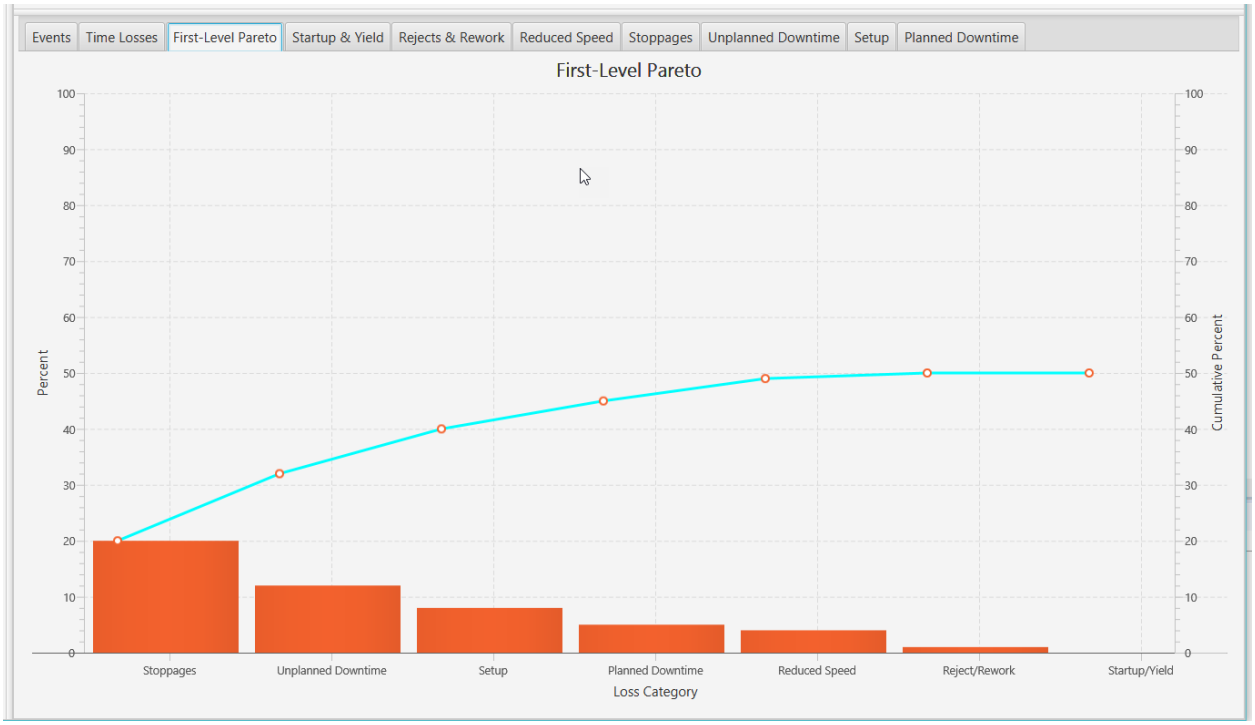
First-Level Pareto Chart

The first-level pareto chart displays the percentage of Available Time consumed by each of the 7 OEE losses:

1. Planned Downtime (10 minutes)
2. Setup (16 minutes)

3. Unplanned Downtime (24 minutes)
4. Minor Stoppages (40 minutes)
5. Reduced Speed (8 minutes)
6. Rejects and Rework (2 minutes)
7. Startup and Yield (0 minutes)

The total time lost in these 7 categories is thus 100 minutes.

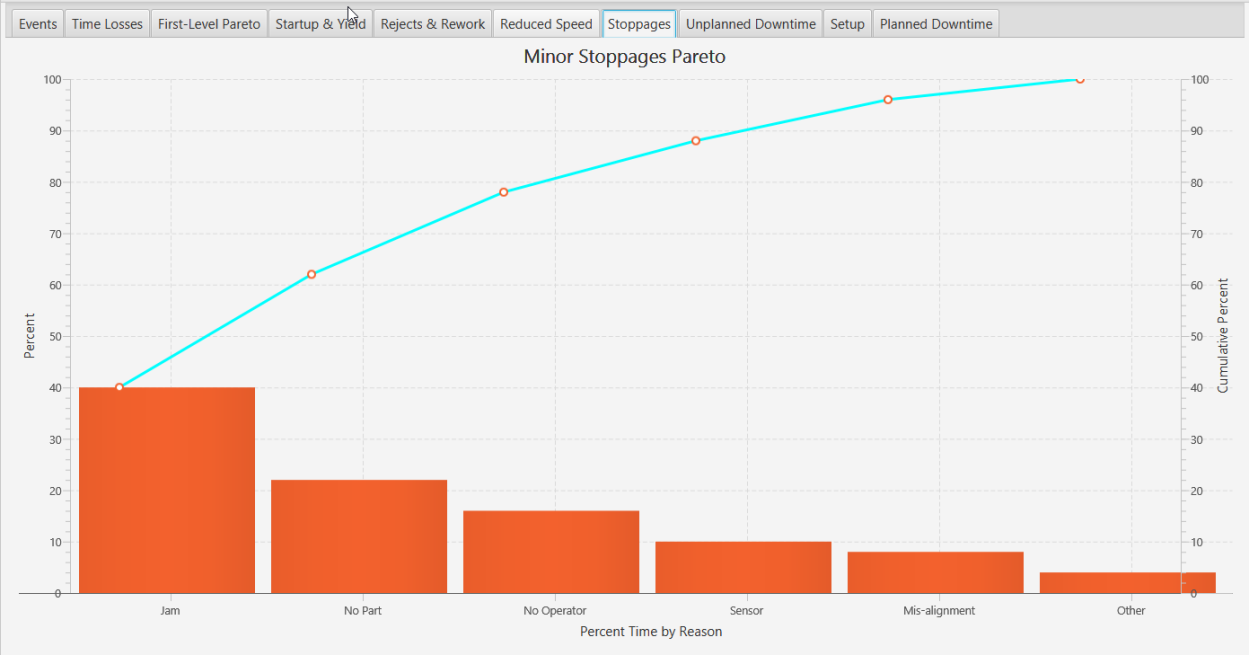


The y-axis shows the 7 loss categories. For the example above, minor stoppages consumed 20% (40 minutes/ 200 minutes), whereas startup & yield has no losses.

This Pareto chart also shows the cumulative percentage increasing from 20% (40/200) to 50% (100/200).

Second-Level Pareto Charts

A Pareto chart for each of the 7 loss categories is available by clicking on the color-coded bar chart segment in the time loss chart, or by selecting the tab of interest. For example, the minor stoppages Pareto looks like this:

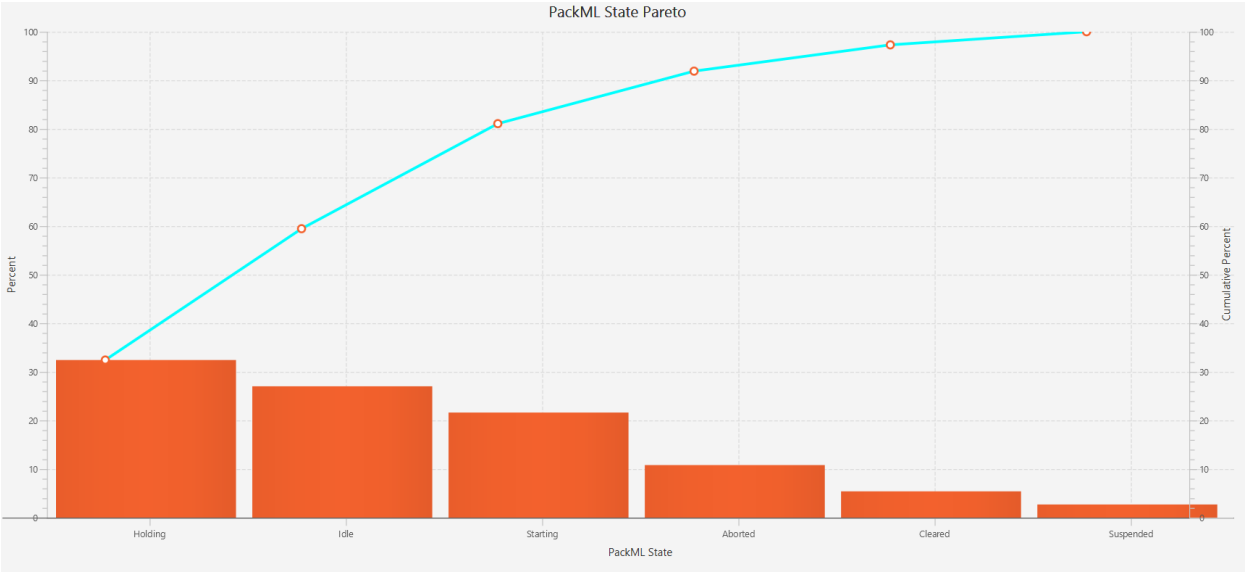


The y-axis shows the top 10 reasons for each minor stoppage (6 in this example). For example, 40% of the minor stoppage losses were due to jams and 4% due to other reasons.

This Pareto chart also shows the cumulative percentage increasing from 40% (16 minutes/40 minutes) to 100% (since there were only 6 reasons for all of the minor stoppages losses).

PackML State Pareto

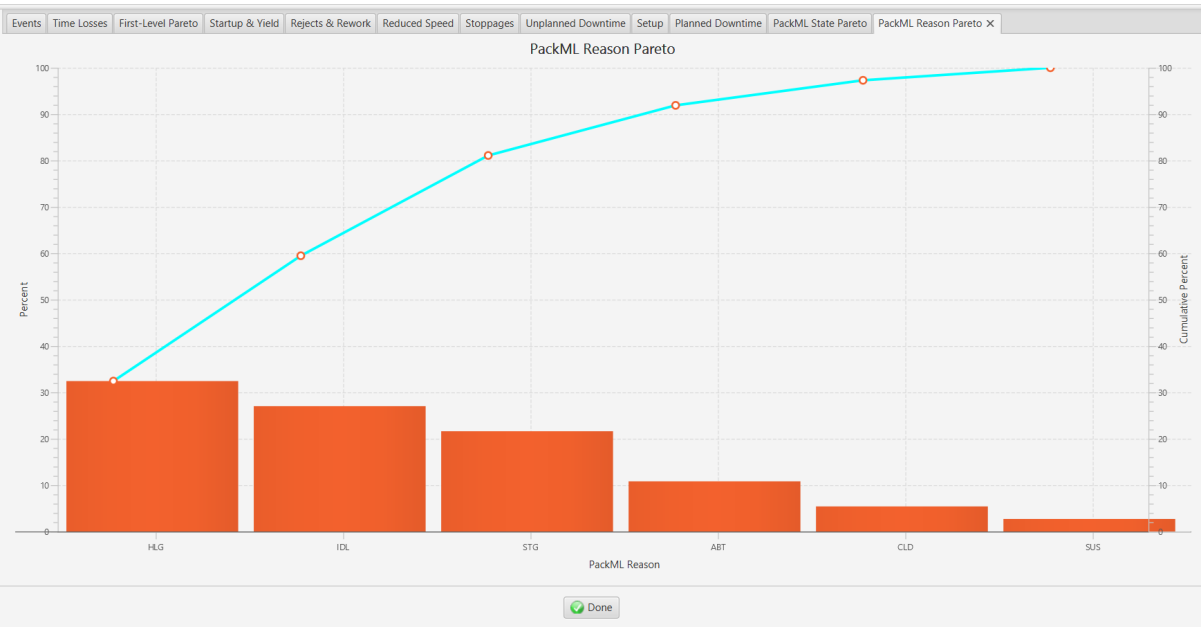
This Pareto chart shows the PackML availability and performance states that are not value-adding, i.e. they represent lost running time. For example:



In this chart we see that the Holding state consumed 33% of the lost performance and availability time. Note that the actual time can be seen by hovering the cursor over the corresponding point in the cumulative line chart.

PackML Reason Pareto

This Pareto chart shows the PackML availability and performance reasons that are not value-adding, i.e. they represent lost running time. For example:



In this chart we see that the "HLG" reason consumed 33% of the lost performance and availability time and the "ABT" reason 12%. Note that the actual time can be seen by hovering the cursor over the corresponding point in the cumulative line chart.

Backup and Restore

The editors each have a Backup button to save the selected object or all objects of the associated class to a disk file with an extension of ".p85x" by default. The selected object can be cleared by right-clicking on the context menu and selecting "Clear selected object".

The backup file contains JSON serialized objects, and is compressed in GZIP format. This file can then be restored to another database.

After clicking the backup (or restore) button, the user chooses a disk file. By using backup and restore capabilities, design-time objects can be moved from a development database to a production database.

The Tool menu also has backup and restore buttons. In this case however, all design-time objects are backed up or all objects in the selected .p85x restore file. When restoring objects, if an object of that name already exists in the target database, that object will be replaced.

COLLECTOR APPLICATION

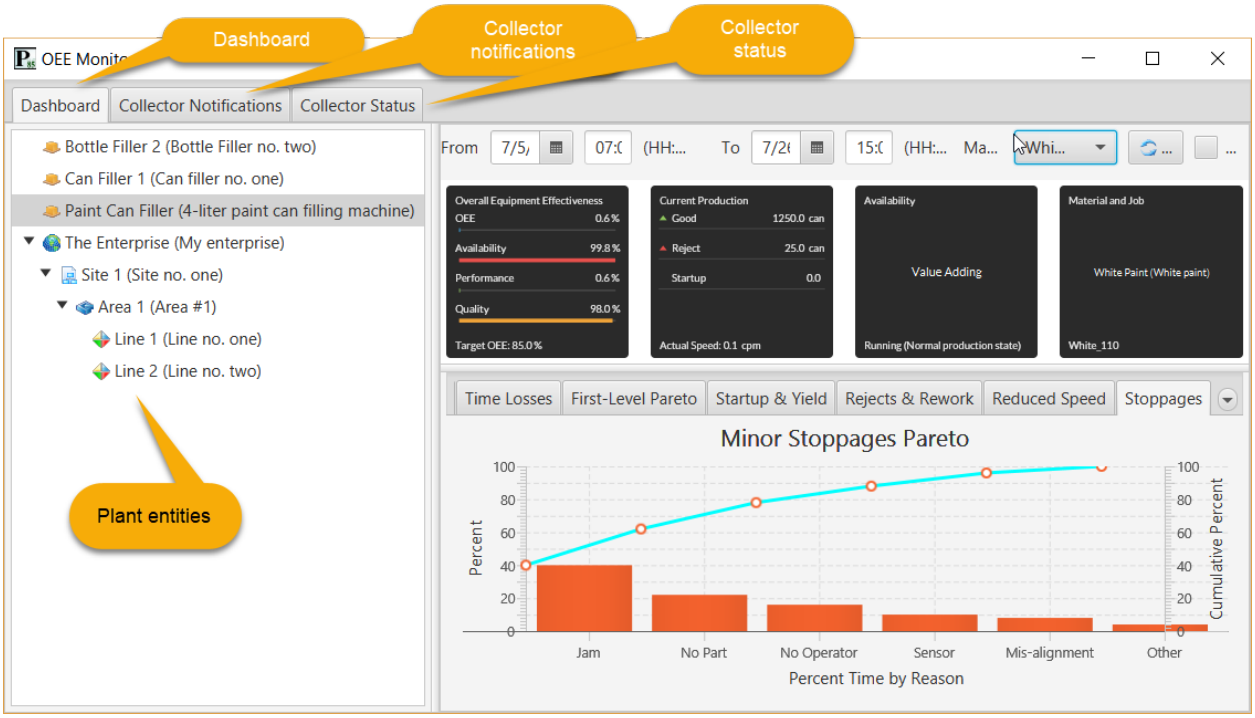
The collector application runs as a Windows service or Unix daemon on the configured host computer. A collector application executes equipment event resolver scripts upon receipt of an input value and stores the availability, production, material or job change event data in the database. This data is used for OEE calculations.

The Tanuki Java Service wrapper provides the infrastructure for the Windows service or Unix daemon. The JVM is 64-bits for all supported O/S's.

MONITOR APPLICATION

The monitor application has three main functions, to observe:

- Equipment performance via metrics available in the dashboard.
- Notifications from the data collectors for abnormal conditions
- Data collector status.



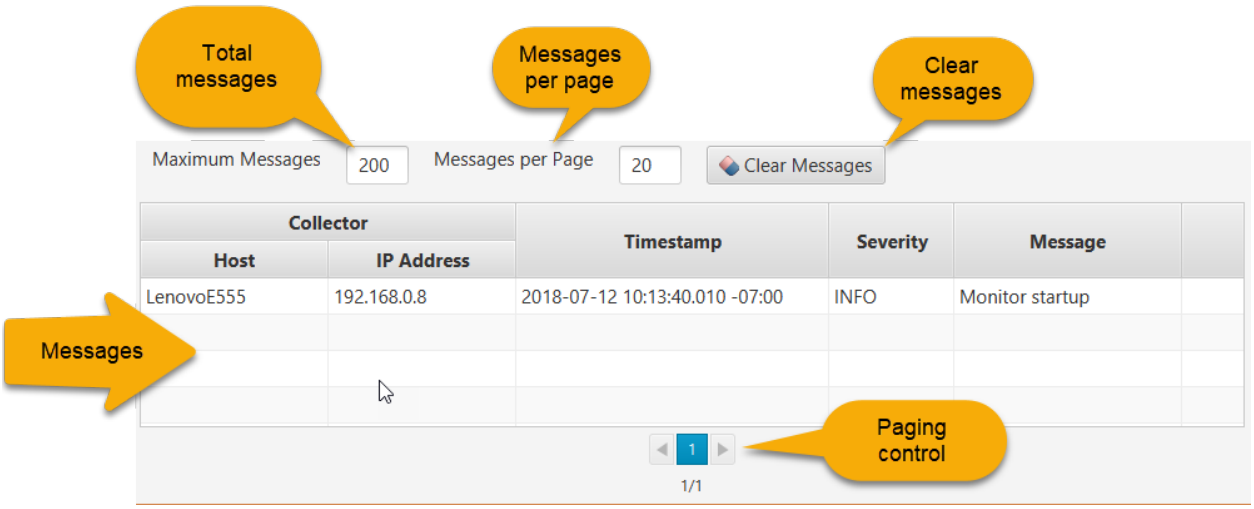
DASHBOARD

The dashboard is displayed by selecting the Dashboard tab. The plant entity hierarchy is displayed in the tree view on the left. After selecting a piece of equipment, the dashboard is shown on the right. For details on the dashboard, see the Designer dashboard section above.

The dashboard will update asynchronously as equipment event messages are received from the messaging broker (if so configured). Otherwise, the auto-refresh feature can be enabled to periodically query the database.

Collector Notifications

Data collector notifications are displayed by selecting the Collector Notifications tab:



If the collector is configured for messaging, when a notification event occurs, a message is sent to the RabbitMQ broker and delivered to all notification subscribers. In the example above, the Monitor sends an informational startup message to itself.

The total number of messages to retain can be specified (e.g. 200). If more than this number of messages is received, the oldest messages will be discarded. The number of messages to display in the table can be specified (e.g. 20). If the number of messages exceeds this page count, the paging control can be used to move forward or backward through them.

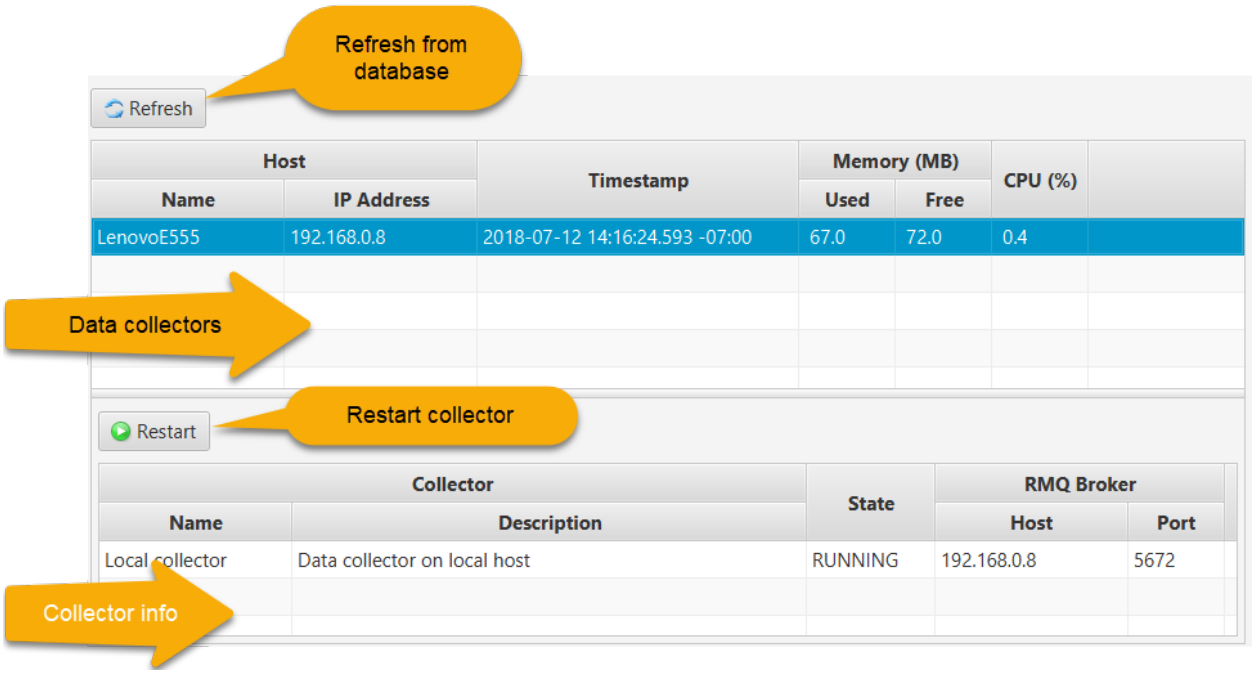
All messages can be removed by clicking on the Clear Messages button.

Information shown for each message is:

- *Host*: the name of the computer on which the collector is running
- *IP Address*: IP address of collector
- *Timestamp*: date and time of day with time zone offset when the message was sent
- *Severity*: severity of message (ERROR, WARNING, INFO)
- *Message*: the text of the message

COLLECTOR STATUS

Data collector status is displayed by selecting the Collector Status tab:



The top table shows the following information about each of the data collectors:

- *Name*: collector host computer name
- *IP Address*: collector host computer IP address
- *Timestamp*: date and tinme of day with zone offset when the status message was sent
- *Used Memory*: heap memory allocated by the collector's JVM
- *Free Memory*: free JVM heap memory
- *CPU*: current CPU load

After selecting a host computer, the following information is shown:

- *Name*: collector name
- *Description*: collector description
- State: collector state - one of DEV, READY or RUNNING
- *Message Broker Host*: the host computer for the collector's message broker
- *Message Broker Port*: the TCP/IP port of the collector's message broker
- *Message Broker Type*: the type (RabbitMQ, JMS, MQTT, KAFKA or EMAIL) of the collector's message broker

OPERATOR DESKTOP APPLICATION

OVERVIEW

The desktop operator application allows a user to manually enter availability, performance, production, material change and job events. The events can be recorded in chronological order as they happened ("By Event") or in summary form ("By Time Period") over a period of time by duration of event. Value adding (running) time is assumed in summary form for availability.

For example, during a time period from 07:00 to 11:00, the chronological events might be:

1. 07:30 - Unplanned downtime with reason #1
2. 08:00 - Running again
3. 09:00 - Good production of 100 units
4. 09:15 - Reject production of 10 units
5. 09:45 - Planned downtime for preventive maintenance with reason #2
6. 10:00 - Running again
7. 10:30 - Good production of 50 units

These 7 events would be entered in chronological order.

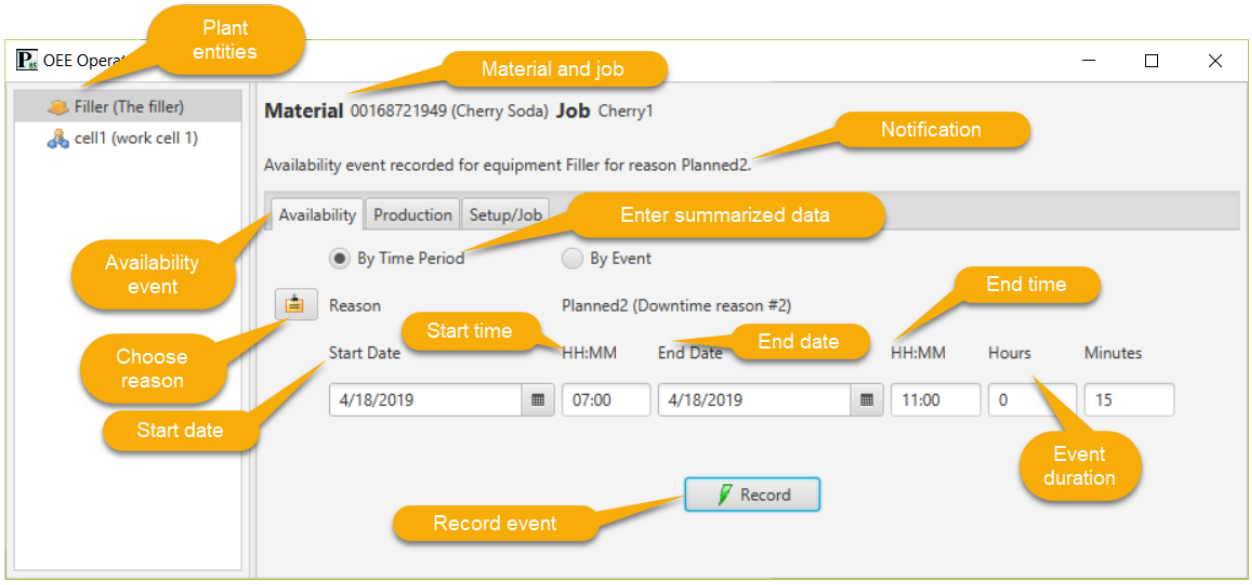
Rather than enter individual events in chronological order, the data for the 07:00 to 11:00 time period could be entered in summarized form as:

1. Good production of 1000 units
2. Reject production of 20 units with a reason
3. Planned downtime of 10 minutes with a reason
4. Unplanned downtime of 24 minutes with a reason
5. Unscheduled downtime of 40 minutes with a reason
6. Setup time of 16 minutes with a reason
7. Minor stoppages of 40 minutes with a reason

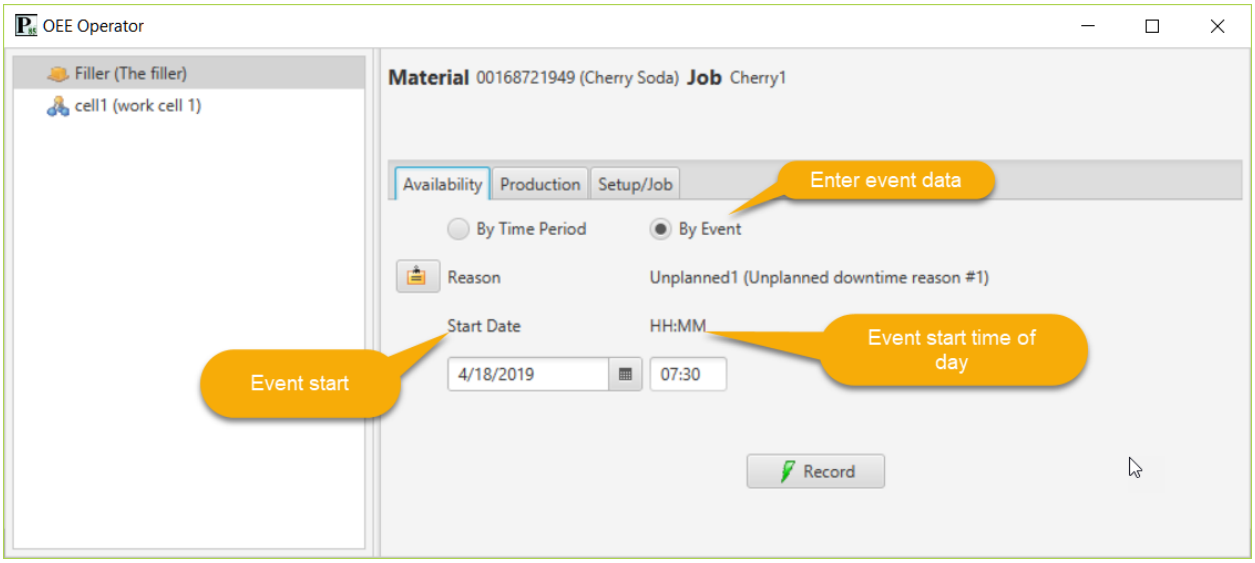
If the equipment producing this material has an ideal speed of 10 units/minute, the OEE for this 4 hour period wil be 50% with an availability of 75%, performance of 68% and quality of 98%.

USER INTERFACE

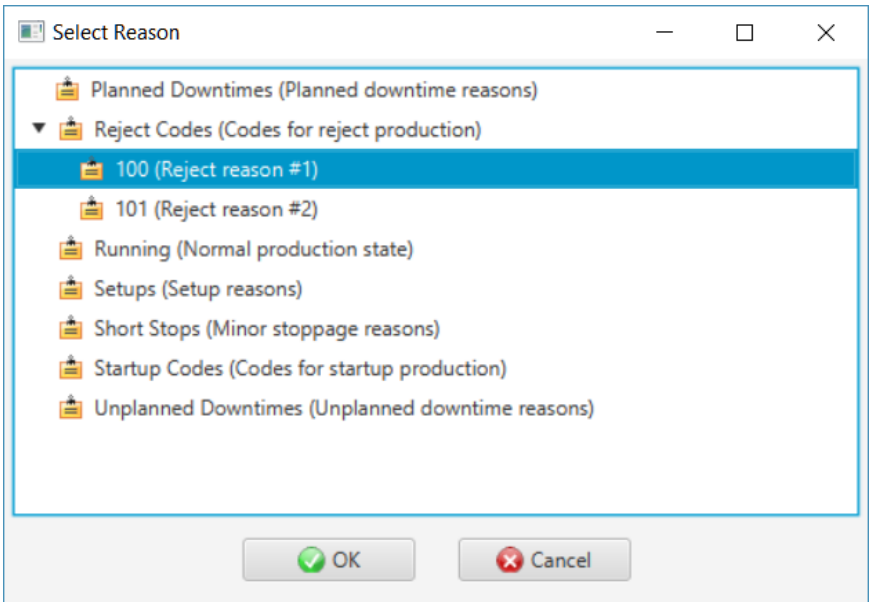
The user interface for availability/performance tab for a summarized event looks like:



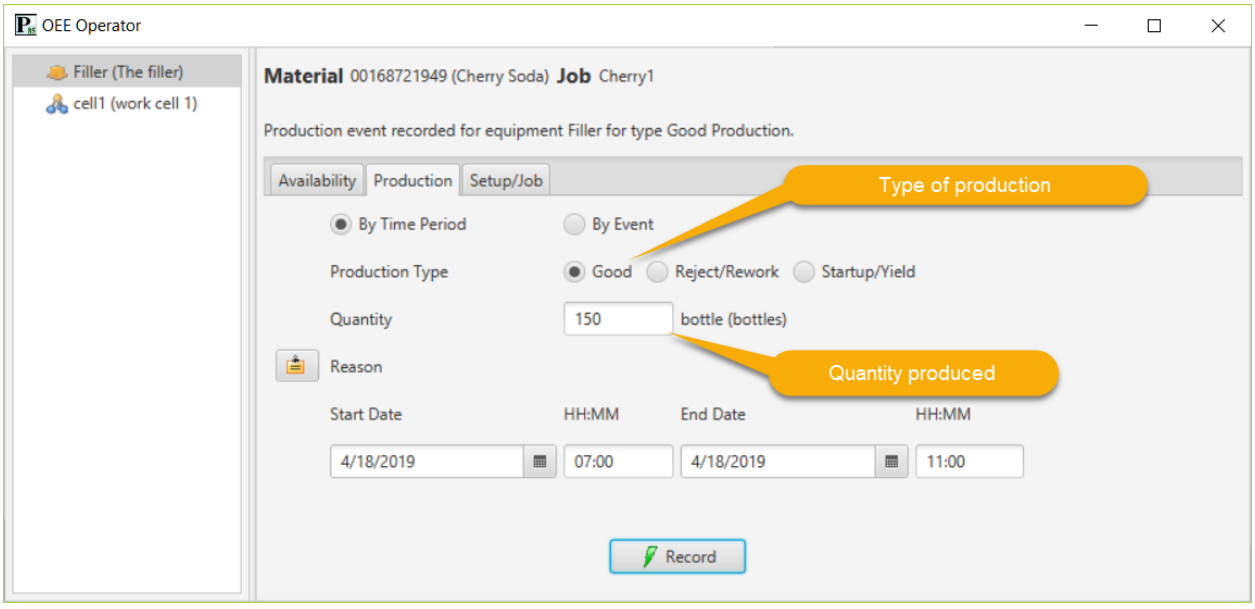
Note that if “By Event” is selected, the UI will change to allow entry of only the date and time of day when the availability event occurred:



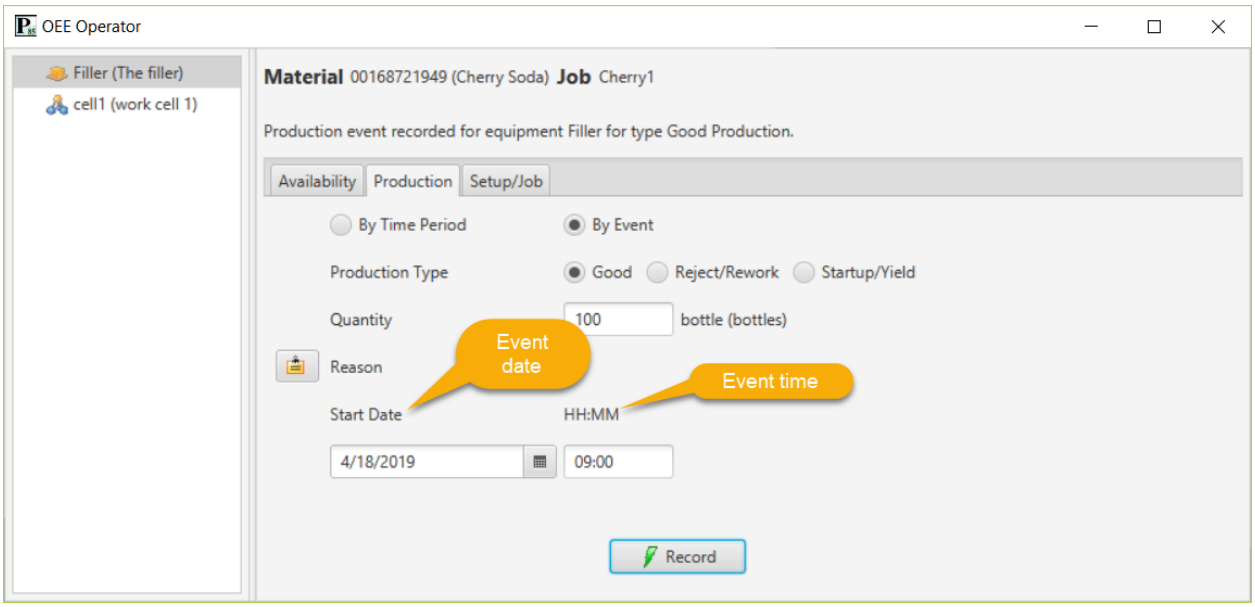
An availability reason can be choose by clicking the “Reason” button and selecting from the reason hierarchy:



The production tab for a summarized production event is shown below. For reject and rework as well as startup and yield events, a reason can be entered as well:

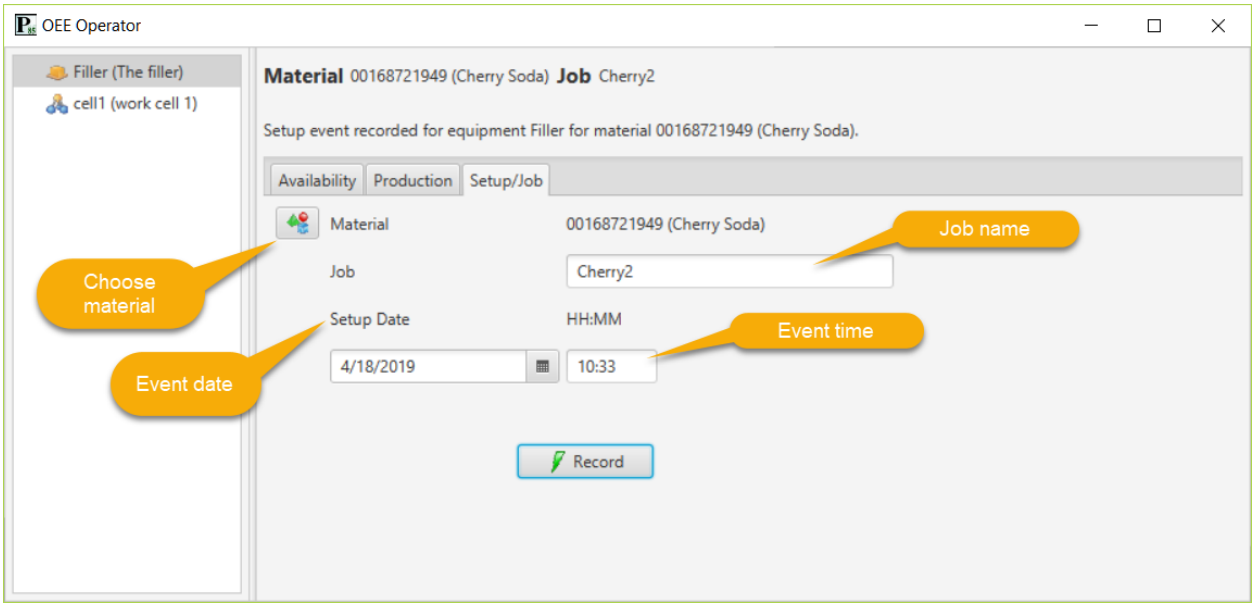


Note that if “By Event” is selected, the UI will change to allow entry of only the date and time of day when the production event occurred:

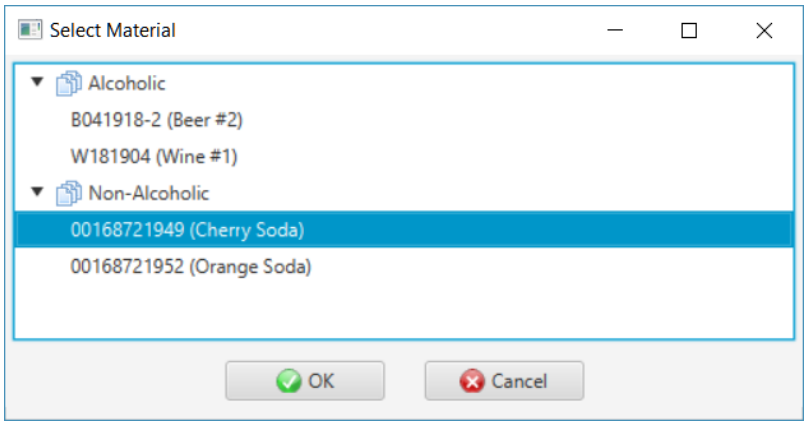


An production reason can be choose by clicking the “Reason” button and selecting from the reason hierarchy.

The material and job change tab looks like:



The produced material is chosen from the pre-defined materials by clicking on the “Material” button:



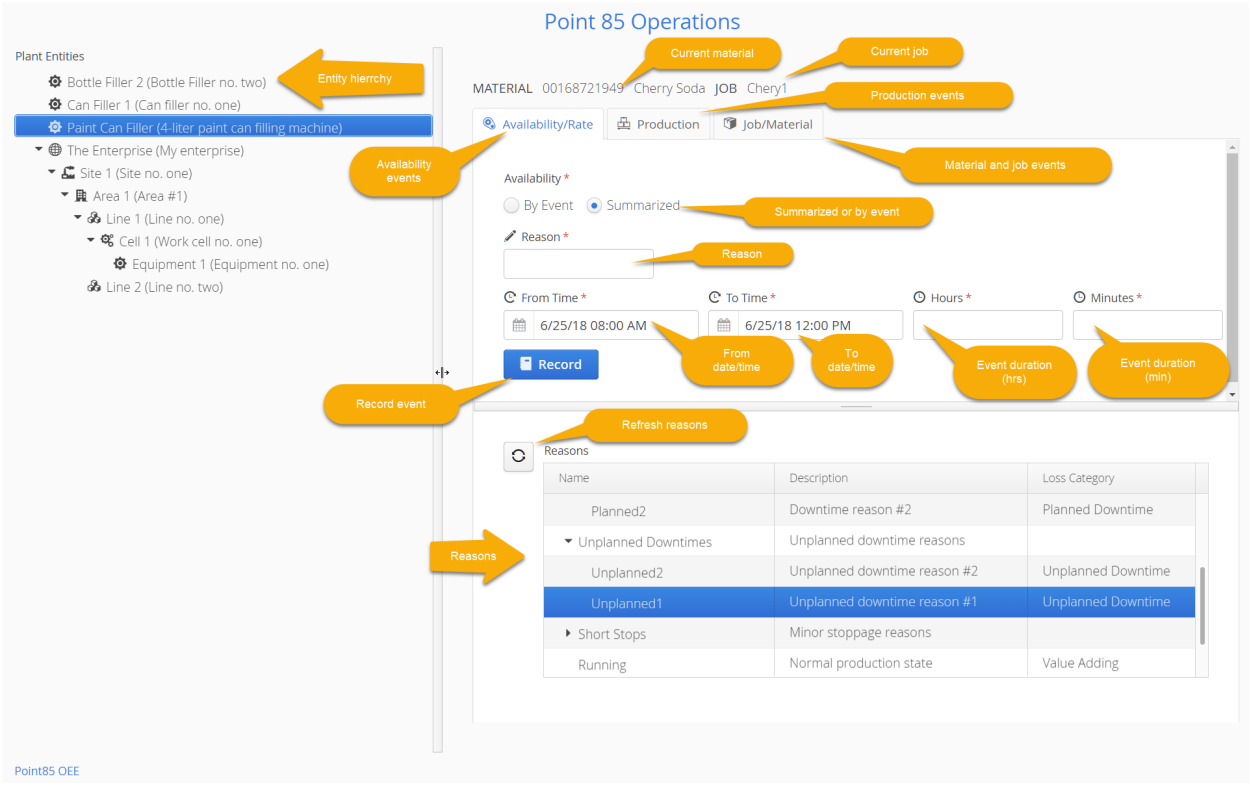
Operator Web Application

Overview

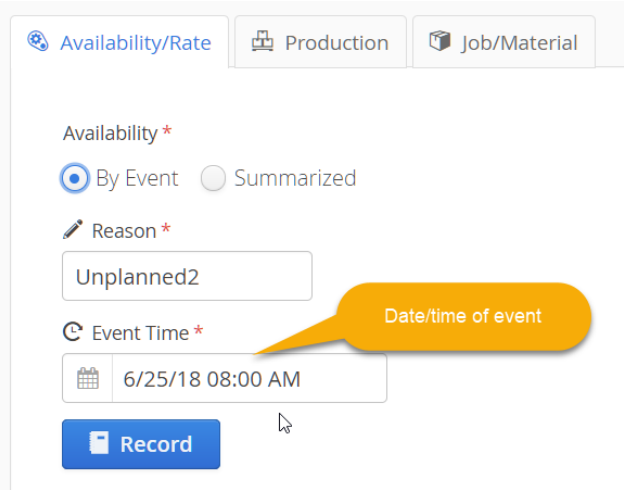
The operator web application is browser-based and allows a user to enter availability, performance, production, material change and job events. The events can be recorded in chronological order as they happened ("By Event") or in summary form ("Summarized") over a period of time by duration of event. Value adding time is assumed in summary form for availability.

User Interface

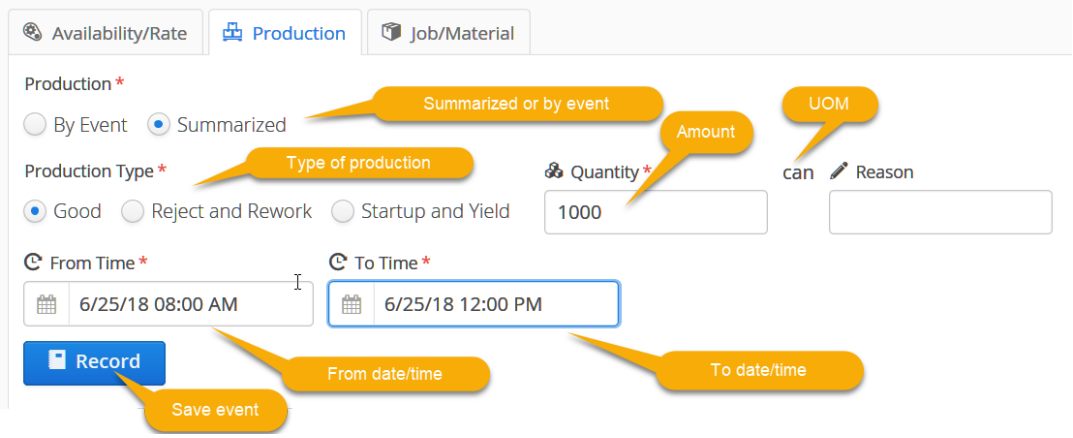
The user interface for availability/rate tab for a summarized event looks like:



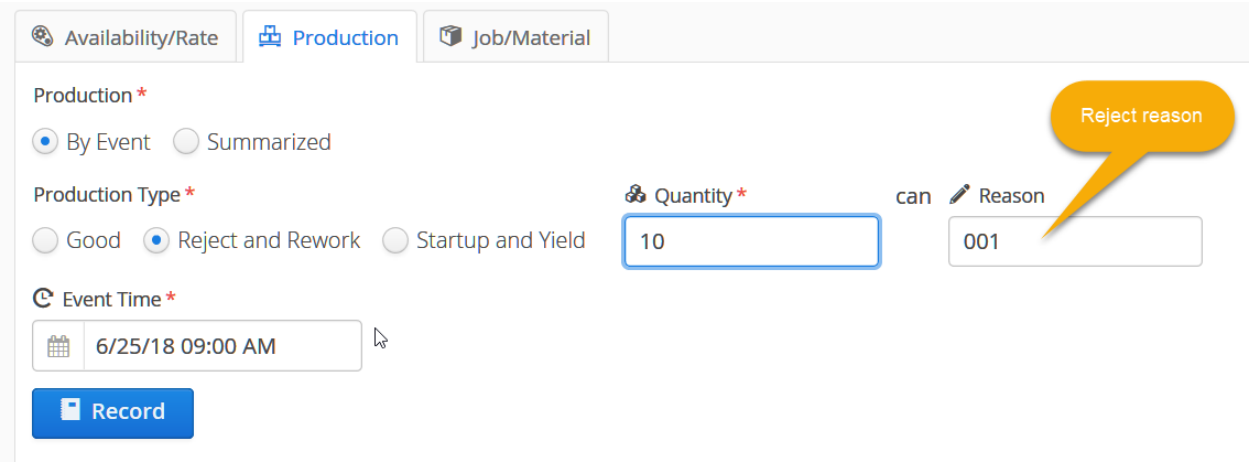
Note that if “By Event” is selected, the UI will change to allow entry of only the date and time of day when the availability event occurred:



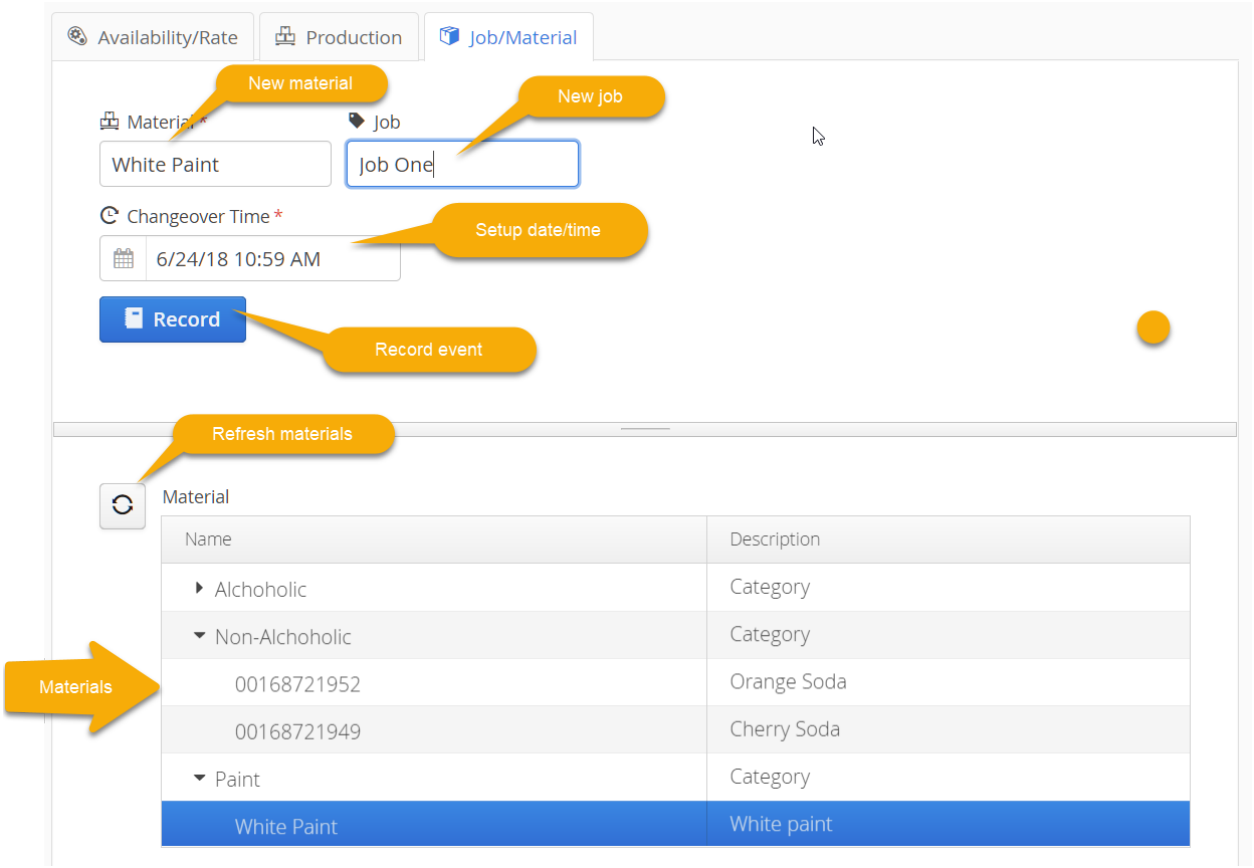
The production tab for a summarized event looks like:



Note that if "By Event" is selected, the UI will change to allow entry of only the date and time of day when the production event occurred. For reject and rework as well as startup and yield event, a reason can be entered as well:



The material and job change tab looks like:



Operator Mobile Application

Overview

The Android and iOS mobile applications allow a user to enter availability, performance, production amounts, material change and job events. The events can be recorded in chronological order as they happened ("By Event") or in summary form ("By Time Period") over a period of time by duration of event. Value adding time is assumed in summary form for availability. The applications are written in Flutter/Dart and use the HTTP REST service APIs.

The Android app is downloaded from the Google Play Store. Search for "Overall Equipment Effectiveness", "OEE" or "OEE Mobile". The iOS app is downloaded from the Apple App Store. Search for "Overall Equipment Effectiveness", "OEE" or "Point85 OEE".

User Interface

Settings

The host name or IP address and port of the HTTP server embedded in a Collector must be entered before any APIs can be called. On the plant entity home page, tap on the Settings button, then enter the host name and port as below:

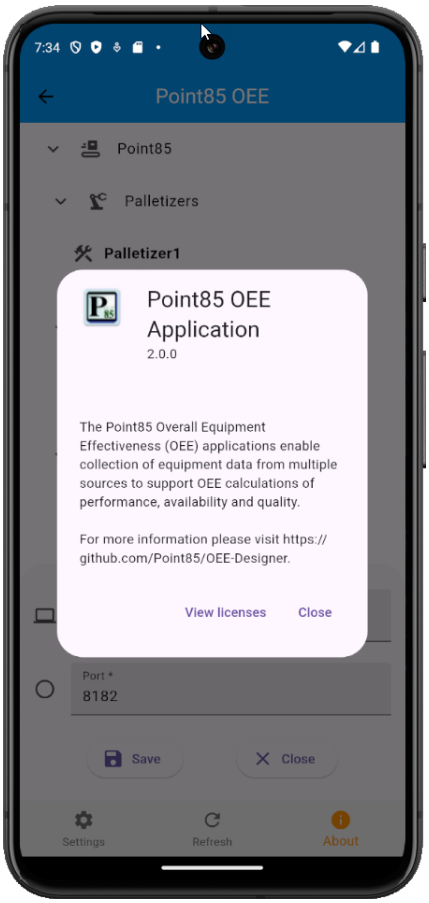
Tap the Save button to save these settings. If necessary, tap the Refresh button to query the database (note this will be necessary after initializing or changing the HTTP settings). Once a valid server is specified and communications established, the main screen will be populated with the top-level plant entities.

Refresh

Tap the Refresh button to re-build the hierarchy of equipment. This will be necessary if changes have been made in the Designer configuration.

About

Tapping the About button will show information about the application:



Entity Page

On the home entity page, tapping the down arrow expands that node in the nested list as shown in the home page screen capture above.

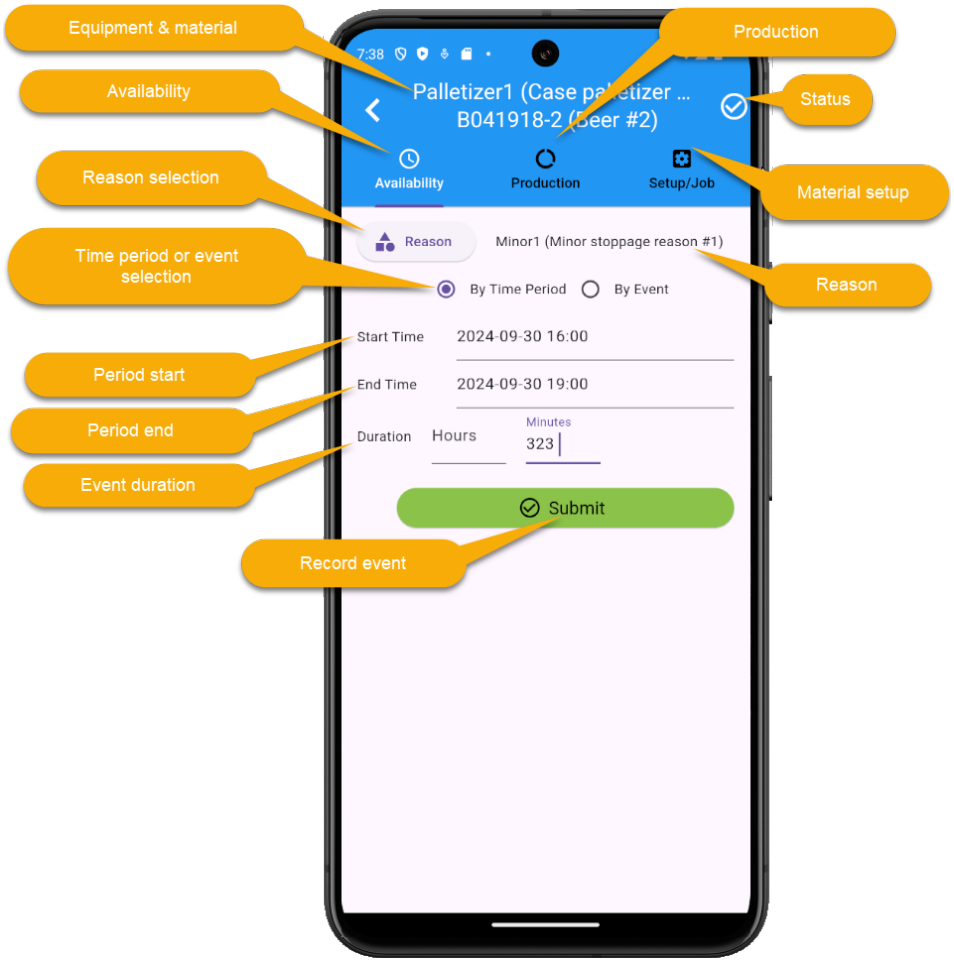
In this example, Point85 production line is at the top-level and Palletizer1 (belonging to work cell Palletizers) is an equipment entity that can have OEE data recorded for it.

Equipment Event Page

Tapping on an equipment node displays the data entry page for availability, production and setup events.

Availability

Tapping on the Availability tab displays the availability data entry page. An availability event can be entered in summary form for a period of time, or on an event by event basis. For example, this screen capture shows the Palletizer1 equipment:



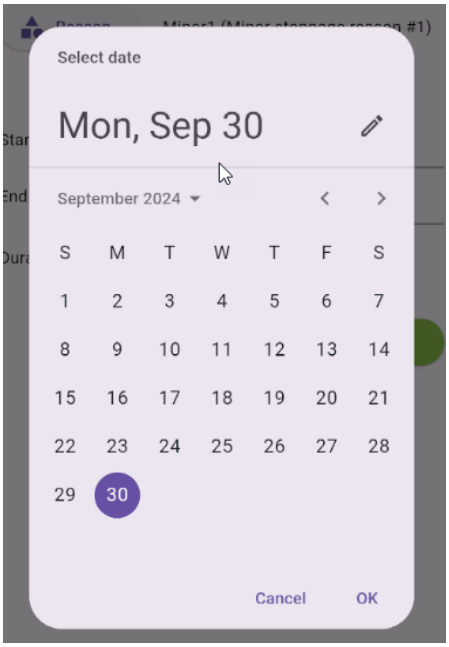
The app bar shows the equipment name and description, currently producing material name and description and an icon for the current availability status:

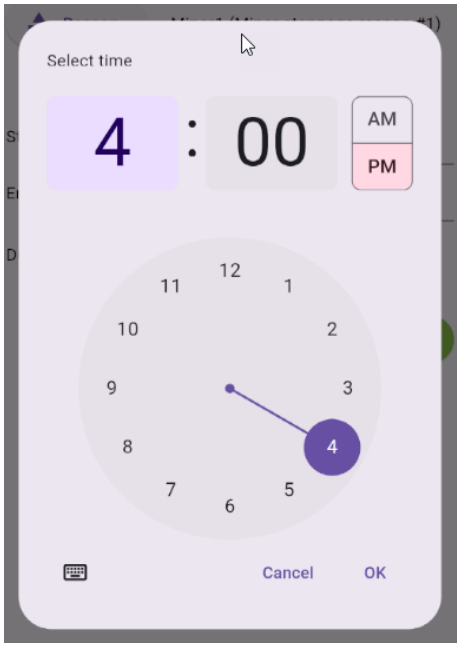
- No loss:
- Minor stoppage:
- Not scheduled:
- Planned downtime:
- Reduced speed:
- Reject/rework:
- Setup:
- Startup & yield:

- Unplanned downtime: 
- Unscheduled: 

Tapping on the Reason button displays a page for selecting an availability reason as discussed below. After selecting a reason, tap on the “By Time Period” or “By Event” radio button.

If entering a single event, the event time is entered by tapping in the time field. A date selector is presented. Select the date then tap on the OK button. A selector for hours and minutes of the day is presented next. For example:





After entering the reason and time information, tap the Submit button. If the data is successfully recorded, a banner at the bottom of the screen will display for a short time. An error dialog will be presented if there is an error.

If summarized data by period of time is to be entered, tap on the “By Time Period” radio button. In addition to entering the period start time, also enter the period end time. After the time period is defined, enter the duration of the events during this period in hours and minutes. In this example, over the 3 hour period, 32 minutes of short stop time was accumulated.

Production Amounts

Tapping on the Production tab displays the production data entry page. Similar to availability events, production amounts can be entered summarized over a time period or by event when the amounts were produced.

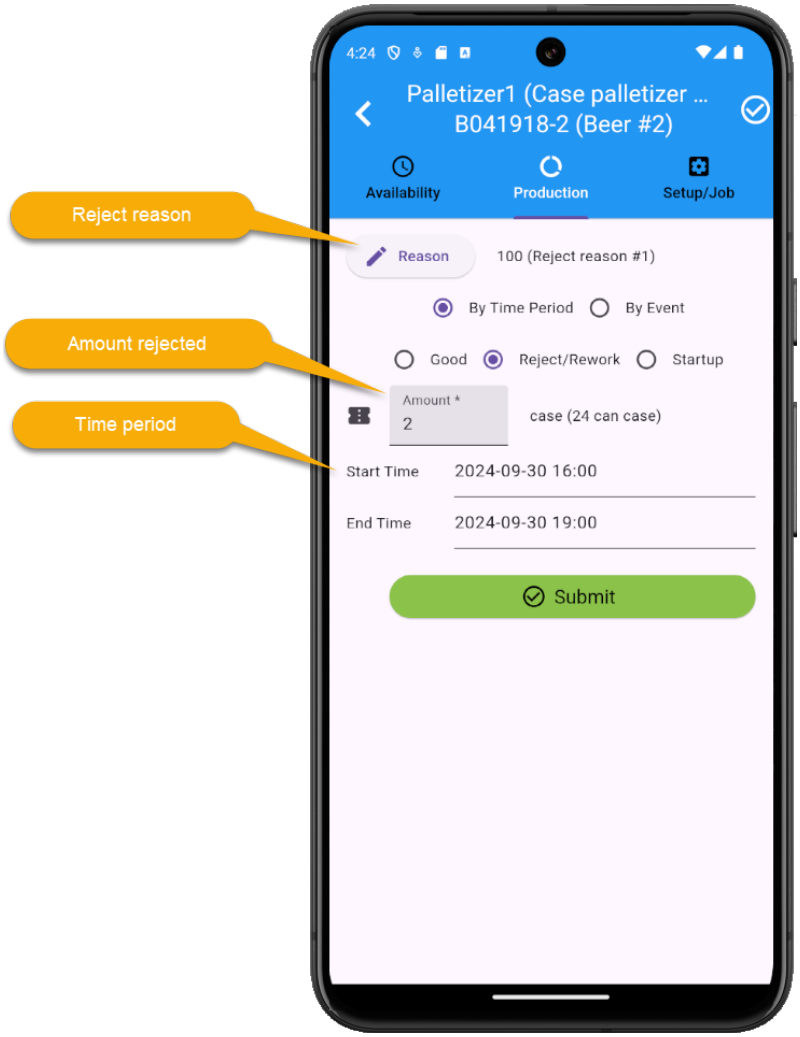
First select “By Time Period” or “By Event”. A reason (such as for scrap/reject) is optional. Next, select the type of event:

- Good: good production

- Reject/Rework: production that failed to meet good standards
- Startup: production lost to starting up the equipment

Finally enter the amount of production. The unit of measure and description will be displayed next to the amount. Tapping the Submit button will record the production in the database.

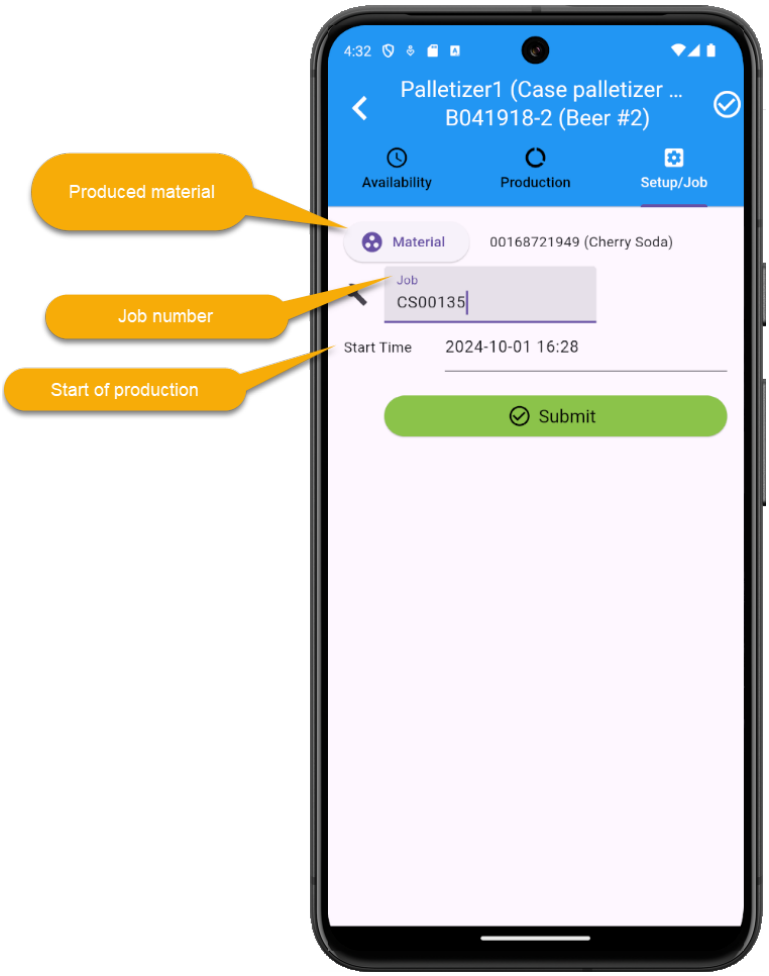
For example, in the screen shot below, 2 cases were rejected over an 3 hour time period due to reason “100”.



Equipment Setup

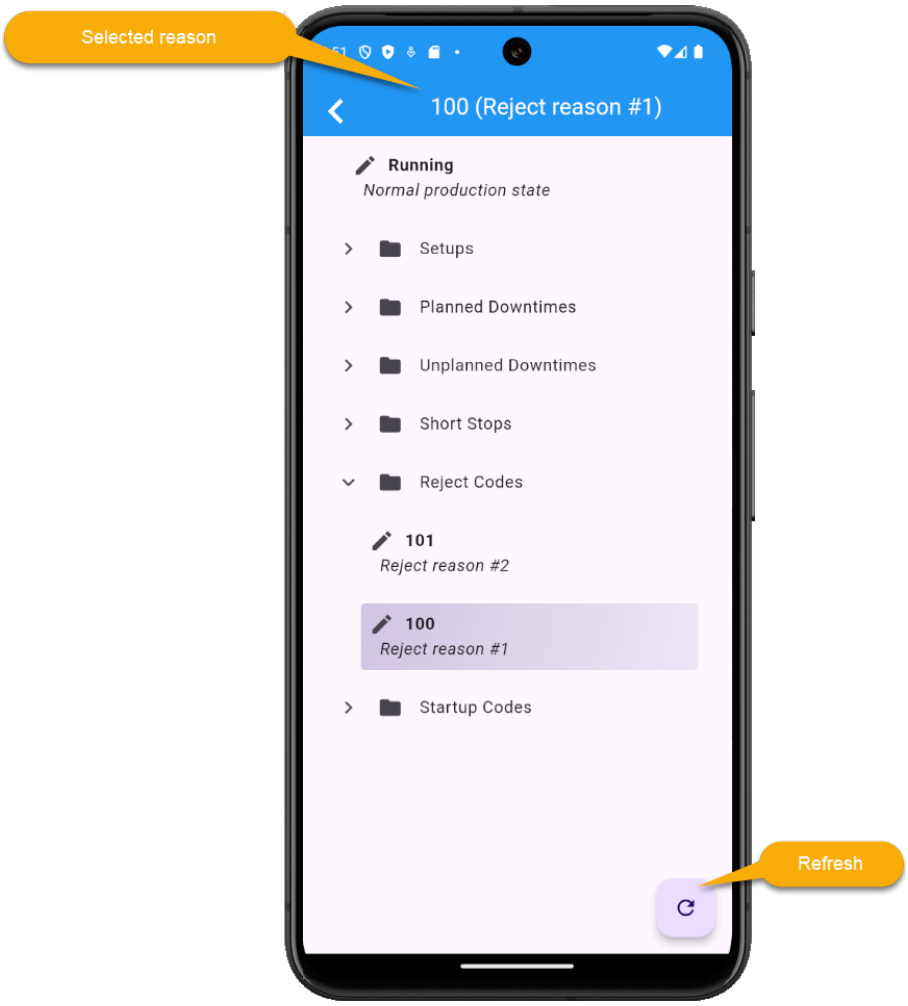
Tapping on the Setup/Job tab displays the material setup and job data entry page. Material and job changes are recorded by event only. If the material is being changed, tap on the “Material” button and select the new material as described below. Enter an optional job number. If only the job is being changed, it is not necessary to choose the same material again. Tap the “Submit” button to record the setup event. Note that a setup must be performed before entering any availability or production events.

For example, in the screen capture below, the equipment has been changed over to produce cherry soda for job #CS00135:



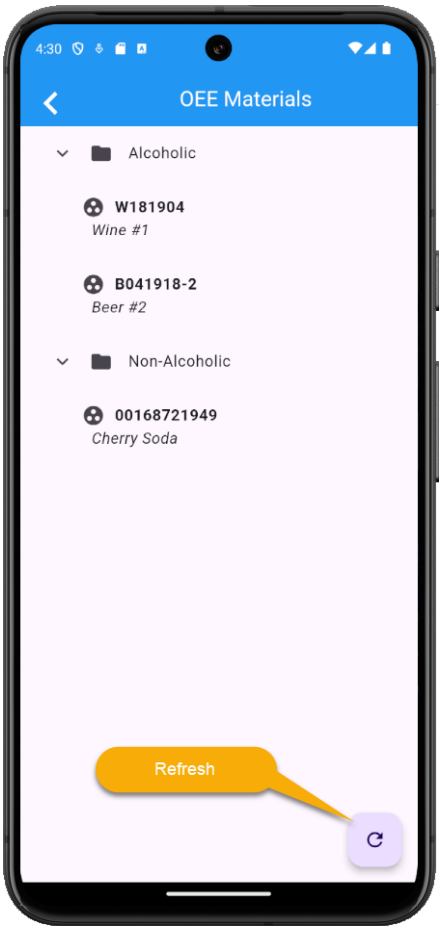
Reason Page

Reasons are organized hierarchically. Tapping on a reason in the list expands to show the children reasons. A loss category is associated with an availability reason. For example:



Material Page

Produced materials are organized within a category as defined in the Designer. Tapping on the down arrow expands that category to show the materials. For example:

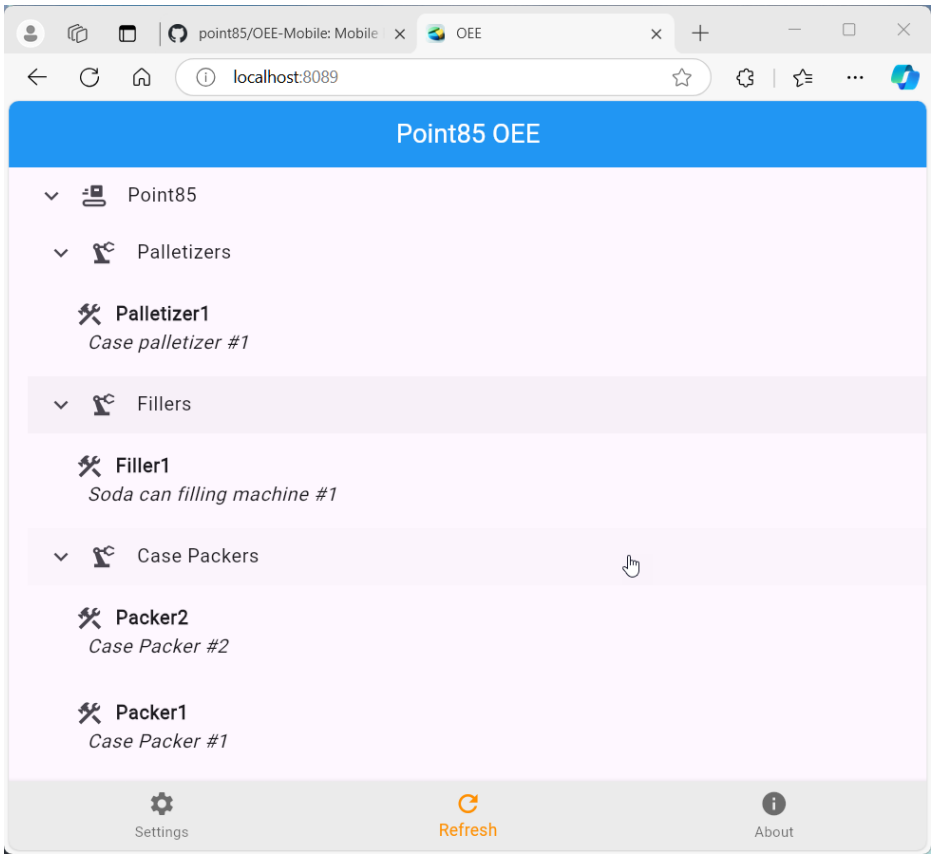


Operator Applications

Web

As an alternative to the web application described above, a web app built for the Edge or Chrome browers can be installed from the OEE Mobile project release on Github (see https://github.com/point85/OEE-Mobile).

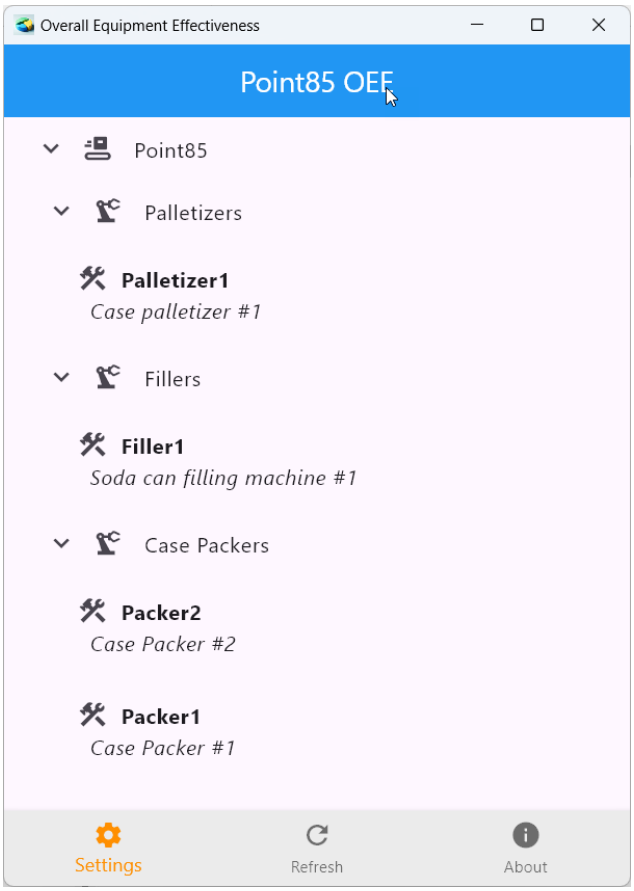
The user interface is identical to the Android and iOS mobile apps discussed above. For example, the home page looks similar to:



WINDOWS

As an alternative to the desktop application described above, a Windows app can be installed from the OEE Mobile project release on Github (see https://github.com/point85/OEE-Mobile).

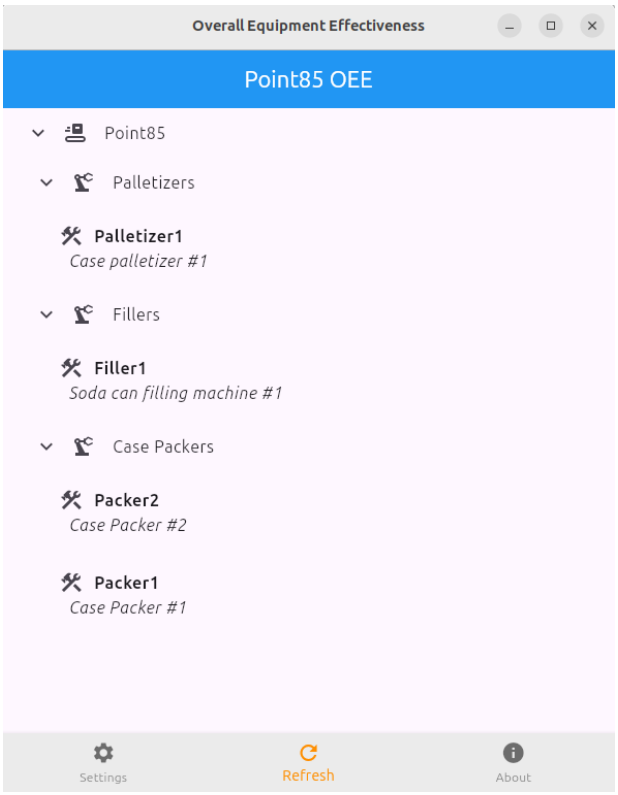
The user interface is identical to the apps discussed above. For example, the home page looks similar to:



LINUX

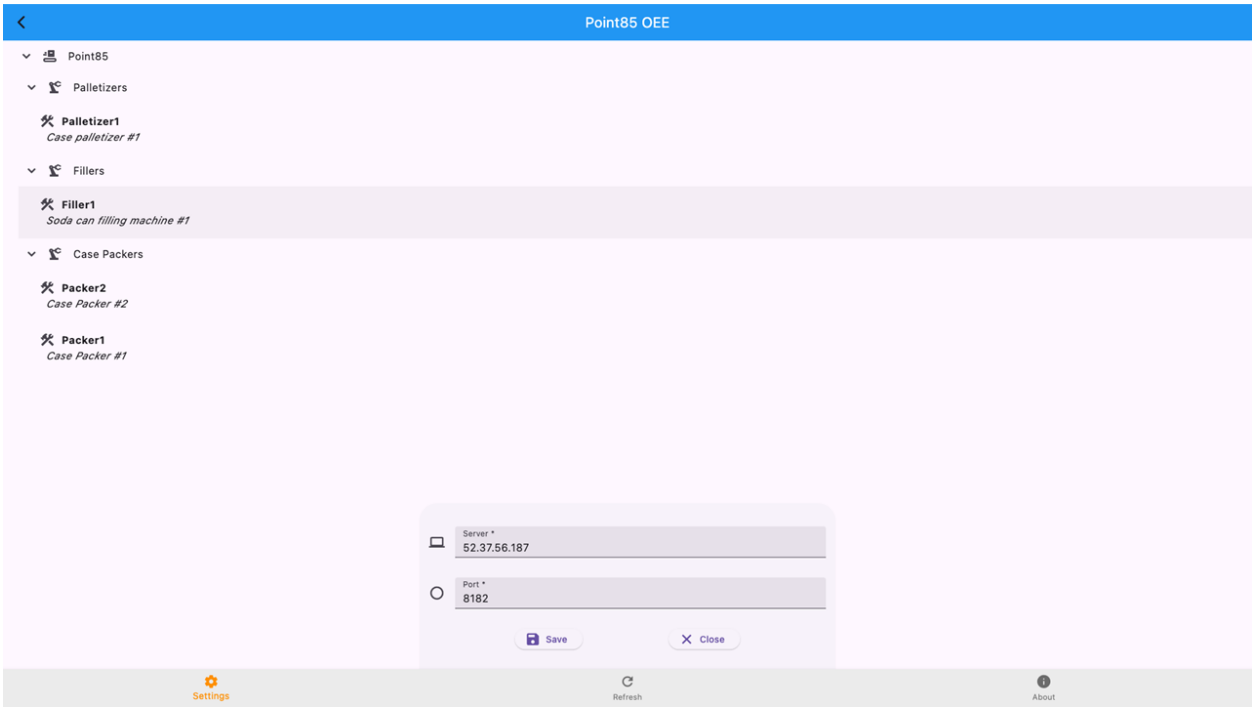
As an alternative to the desktop application described above, a Linux app can be installed from the OEE Mobile project release on Github (see https://github.com/point85/OEE-Mobile).

The user interface is identical to the apps discussed above. For example, the home page looks similar to:



MACOS

As an alternative to the desktop application described above, a macOS app can be installed from the Apple App Store. For example, the home page looks similar to:



REST API

Equipment availability, production and setup events can be POSTed to the collector's HTTP server. The plant entity hierarchy, reasons, materials, equipment status are also accessible by a GET request, updated by a PUT request and deleted by a DELETE request.

The base REST URL is "http://<server address>:<port>/rest". The table sbelow list the REST requests and responses. Body data are JSON strings with a timestamp in ISO 8601 format if applicable. For creation and update requests, the JSON is in the body. All other requests read and delete, the object is identified in the request URL. Required values are indicated by an asterisk. An object is renamed on a PUT by including the new name in the request body.

A successful response is has an HTTP response code of 200. An error response (indicated by a HTTP error code of 400 or 500) has the error text in the JSON. For example:

{"status":"OK","errorText":"<error text>"}

Note that the "OK" status means that the request was processed by the server, but the associated business logic did not complete successfully.

MATERIAL API

The endpoint is "/materials". For one material (read, update or delete), the endpoint is "/materials/<material name>". Material attributes serialized from the MaterialDto class are:

- *name: String
- description: String
- category: String

Description	Verb	Example Body	Request Example	Response Example
Create	POST	{"category": "Alcoholic", "name":"B082001-1","description":"Beer #5"}	http://localhost:8183/rest/materials	{"category":"Alcoholic","name":"B082001-1","description":"Beer #5"}
Read all	GET		http://localhost:8183/rest/materials	[{"category":"Alcoholic","name":"B041918-1","description":"Beer #1"}, ...]
Read one	GET		http://localhost:8183/rest/materials/B041918-1	{"category":"Alcoholic","name":"B041918-1","description":"Beer #1"}
Update	PUT	{"category": "Alcoholic", "description":"Newest beer"}	http://localhost:8183/rest/materials/B082001-1	{"category":"Alcoholic","name":"B082001-1","description":"Newest beer"}
Delete	DELETE		http://localhost:8183/rest/materials/B082001-1	{"category":"Alcoholic","name":"B082001-1","description":"Newest beer"}

REASON API

The endpoint is "/reasons". For one reason (read, update or delete), the endpoint is "/reasons/<reason name>". Reason attributes serialized from the ReasonDto class are:

- *name: String
- description: String
- lossCategory: String, OEE loss category (NOT_SCHEDULED, UNSCHEDULED, REJECT_REWORK, STARTUP_YIELD, PLANNED_DOWNTIME, UNPLANNED_DOWNTIME, MINOR_STOPPAGES, REDUCED_SPEED, SETUP, NO_LOSS)
- parent: String, name of parent reason
- childrenNames: String list of child reason names for GET

Description	Verb	Example Body	Request Example	Response Example

Create	POST	{"parent":"R1","lossCategory":"MINOR_STOPPAGES","name":"R1.2","description":"R1.2"}	http://localhost:8183/rest/reasons	{"parent":"R1","lossCategory":"MINOR_STOPPAGES","childrenNames":[],"name":"R1.2","description":"R1.2"}
Read all	GET		http://localhost:8183/rest/reasons	[{"lossCategory":"PLANNED_DOWNTIME","childrenNames":["R1.1"],"name":"R1","description":"R1"},{"parent":"R1","lossCategory":"MINOR_STOPPAGES","childrenNames":["R1.1.1"],"name":"R1.1","description":"R1.1"},...]
Read one	GET		http://localhost:8183/rest/reasons/R1.1	{"parent":"R1","lossCategory":"MINOR_STOPPAGES","childrenNames":["R1.1.1"],"name":"R1.1","description":"R1.1"}
Update	PUT	{"name":"R1.2","description":"OEE reason R1.2"}	http://localhost:8183/rest/reasons/R1.1	{"parent":"R1","childrenNames":[],"name":"R1.2","description":"OEE reason R1.2"}
Delete	DELETE		http://localhost:8183/rest/reasons/R1.2	{"parent":"R1","children":[],"name":"R1.2","description":"OEE reason R1.2"}

ENTITY API

The endpoint is "/entities". For one entity (read, update or delete), the endpoint is "/entities/<entity name>". Enterprise, Site, Area, ProductionLine, WorkCell and Equipment common attributes are:

- *name: String
- description: String
- *level: String (ENTERPRISE, SITE, AREA, PRODUCTION_LINE, WORK_CELL or EQUIPMENT)
- parent: String, name of parent entity
- children: String, list of child entity names
- retentionDuration: int, number of seconds to retain event data
- entitySchedules: list of WorkScheduleDtos

    - workSchedule: String, name
    - startDateTime: String, ISO8601 schedule start time
    - endDateTime: String, ISO8601 schedule end time
- eventResolvers: list of EventResolverDtos
    - dataSource: String, name of data source
    - sourceId: String, data source identifier
    - functionScript: String, event resolver script
    - dataType: String, data type of input value
    - collector: String, name of data collector
    - type: String, one of AVAILABILITY, PROD_GOOD, PROD_REJECT, PROD_STARTUP, MATL_CHANGE, JOB_CHANGE or CUSTOM

Equipment can have an additional attribute of materials produced:

- equipmentMaterials: list of EquipmentMaterialDtos:
    - oeeTarget: double, target OEE
    - runRateAmount: double, design speed amount
    - runRateUOM: String, design speed unit of measure
    - rejectUOM: String, reject unit of measure
    - material: String, name of material produced
    - isDefault: Boolean, if true, then this is the default produced material

Description	Verb	Example Body	Request Example	Response Example
Create	POST	{"level":"EQUIPMENT","name":"EQ1","description":"Equipment #1"}	http://localhost:8183/rest/entities	{"children":[],"level":"EQUIPMENT","entitySchedules":[],"eventResolvers":[],"name":"EQ1","description":"Equipment #1"}
Read all	GET		http://localhost:8183/rest/entities	[{"children":["Case Packer Cells","Palletizers","Fillers"],"level":"PRODUCTION_LINE","entitySchedules":[{"startDateTime":"2020-01-01T00:00:00.000","endDateTime":"2030-01-01T00:00:00.000","workSched

				ule":"Manufacturing Company"}],"eventResolvers":[],"retentionDuration":2592000,"name":"Point85 Production Line","description":"Demonstration production line"}]
Read one	GET		http://localhost:8183/rest/entities/Palletizer1	{"parent":"Palletizers","children":[],"level":"EQUIPMENT","entitySchedules":[],"eventResolvers":[{"dataSource":"localhost:8183","sourceId":"Palletizer1.HTTP.PROD_REJECT","functionScript":"function f5f504255(context, value, resolver) {return value;}","dataType":"String","collector":"Demo collector","type":"PROD_REJECT"}],"name":"Palletizer1","description":"Case palletizer #1"}
Update	PUT	{"description":"Newest case palletizer"}	http://localhost:8183/rest/entities/Palletizer1	<see GET response>
Delete	DELETE		http://localhost:8183/rest/entities/Palletizer1	<see GET response>

EQUIPMENT API

In addition to its plant entity methods, the equipment's current status can be queried or an event recorded for it.

Status

The status endpoint is "/status/<equipment name>" for a GET request. Equipment attributes serialized from the EquipmentStatusResponseDto class are:

- material: name, category and description from MaterialDto class
- reason: name, desription, lossCategory and children names from ReasonDto class
- job: String
- runRateUOM: String, unit of measure of design speed
- rejectUOM: String, unit of measure of rejects

Request Example:

http://localhost:8183/rest/status/Palletizer1

Response Example:

{"material":{"category":"Non-Alcoholic","name":"00168721952","description":"Orange Soda"},"reason":{"parent":"R2.1.1","lossCategory":"PLANNED_DOWNTIME","children":[],"childrenNames":[],"name":"R2.1.1.1","description":"R2.1.1.1"},"runRateUOM":"pallet (96 cases per pallet)","rejectUOM":"pallet (96 cases per pallet)"}

Event

An equipment event is one of availability, production or setup/job change. The event endpoint is "/event/<equipment name>" for a POST request. The event is identified either by source identifier or by event type.

Equipment attributes serialized from EquipmentEventRequestDto class are:

- *sourceId: String, the data source identifier as configured for this equipment and resolver type
- *eventType: String, the event identifier for an anonymous HTTP request. One of AVAILABILITY, PROD_GOOD, PROD_REJECT, PROD_STARTUP, MATL_CHANGE or JOB_CHANGE.
- *value: String, depends on the resolver type. For availability, it is the reason. For good, reject or startup production, it is the amount produced. For a material change, it is the material name. For a job change, it is the job identifier.
- *timestamp: String, the local date and time of day when the event occurred, e.g. "2024-09-02T09:46:53.417-07:00", or the beginning of an event interval
- endtimeStamp: String, the local date and time of day at the end of the event interval
- duration: the number of seconds that the equipment was in the availability state for the specified time interval
- reason: String, an optional reason for this event, e.g. a reason for reject production.
- immediate: boolean, if false, then execute the request asycnronously in a thread pool, otherwise execute it immediately. Default is false.

The response to an event posting is the JSON serialized EquipmentEventResponseDto:

- status: String, e.g. OK
- errorText: String, error description if an error
- requestDto: JSON serialized EquipmentEventRequestDto

Request Example #1 for good production between two times:

http://localhost:8183/rest/event/Palletizer1

with body:

{"value":"2500.0","timestamp":"2024-09-17T18:08:00.000","endTimestamp":"2024-09-17T19:08:00.000","duration":"0","eventType":"PROD_GOOD","reason":null,"immediate":true}

and esponse:

{"status":"OK","requestDto":{"eventType":"PROD_GOOD","equipmentName":"Palletizer1","value":"2500.0","timestamp":"2024-09-17T18:08:00.000","endTimestamp":"2024-09-17T19:08:00.000","duration":"0","immediate":true}}

Request Example #2 for 36 minutes of unplanned downtime between two times with body:
{"value":"Unplanned1","timestamp":"2024-09-17T17:15:00.000","endTimestamp":"2024-09-17T18:15:00.000","duration":"2160","eventType":"AVAILABILITY","reason":null,"immediate":true}

and response:
{"status":"OK","requestDto":{"eventType":"AVAILABILITY","equipmentName":"Palletizer1","value":"Unplanned1","timestamp":"2024-09-17T17:15:00.000","endTimestamp":"2024-09-17T18:15:00.000","duration":"2160","immediate":true}}

WEB SERVICE

Equipment availability, production and setup events can be POSTed to the collector's HTTP server. The plant entity hierarchy, reasons, materials, data sources, data source ids, equipment status, OEE Events and calculations are also accessible by a GET request.

The table below lists the web service requests and responses. The Java DTO (Data Transfer Object) is serialized into a JSON string with a timestamp in ISO 8601 format. For Postman examples, see below.

Endpoint	Verb	Parameters	Request Class	Response Class
/event	POST	JSON body with equipment name, sourceId, value[1], time stamp, end time stamp, reason[2], duration, job name and immediate flag	EquipmentEventRequestDto for availability, production of material and setup or job change event	EquipmentEventResponseDto (status and errorText)
/events	GET	equipment=<name> material=<material> [3] type=<event type> [4] from=<date/time>		OeeEventsResponseDto (list of OeeEventDto with event data)

[1] value is availability reason, production count, material id or job id.
[2] reason is an optional startup/reject production count reason

		to=<date/time>[5]		
/reason	GET			ReasonResponseDto (list of ReasonDto with name, description, loss category, parent and children)
/material	GET			MaterialResponseDto (list of MaterialDto with name, description and category)
/entity	GET			PlantEntityResponseDto (list of PlantEntityDto with name, description, level, parent and children)
/data_source	GET	sourceType[6]		DataSourceResponseDto (list of DataSourceDto with name, description, host and port)
/source_id	GET	equipment= <name> sourceType= as above		SourceIdResponseDto (list of source ids)
/status	GET	equipment= <name>		EquipmentStatusResponseDto (MaterialDto, job name, ReasonDto, run rate UOM and reject UOM)
/oee	GET	equipment=<name> material=<material> from=<date/time>[7] to=<date/time>[8]		OeeResponseDto with OEE (%), performance (%), availability (%) and quality (%). The theoretical, required operations, available, scheduled production, reported production, net production, efficient net production, effective net production and value adding times (in seconds) are included. The seven loss times (not scheduled, unscheduled, planned downtime, setup, uplanned downtime, minor stoppages, reduced speed, rejects and yield) in seconds are included. The first level Pareto data is a list of ParetoItemDto (time loss category and value in

[3] optional, default is all materials

[4] type is AVAIL (availability), GOOD (good production), REJECT (reject production), STARTUP (startup production), MATL (material setup), JOB (job change) or CUSTOM

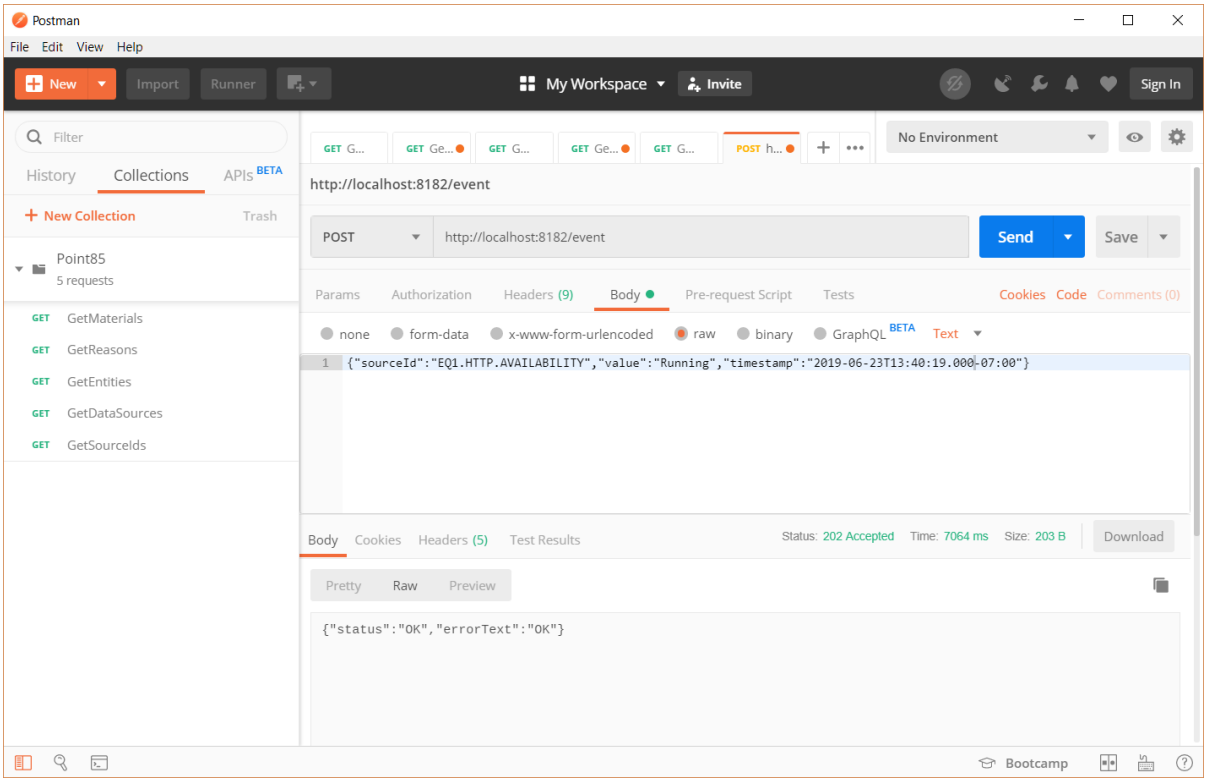
[5] timestamps are optional. Default is for all time. Format is ISO 8601 with millisecond resolyution and a time zone offset, e.g. 2021-09-16T08:00:00.000-07:00

[6] sourceType is OPC_DA, OPC_UA, HTTP, MESSAGING, JMS, MQTT, KAFKA, EMAIL, DATABASE, FILE, PROFICY, MODBUS, CRON or WEB_SOCKET

				seconds). The seven second level Pareto data are lists of ParetoItemDto's. The good, reject and startup quantities are include along with their UOMs.

Event

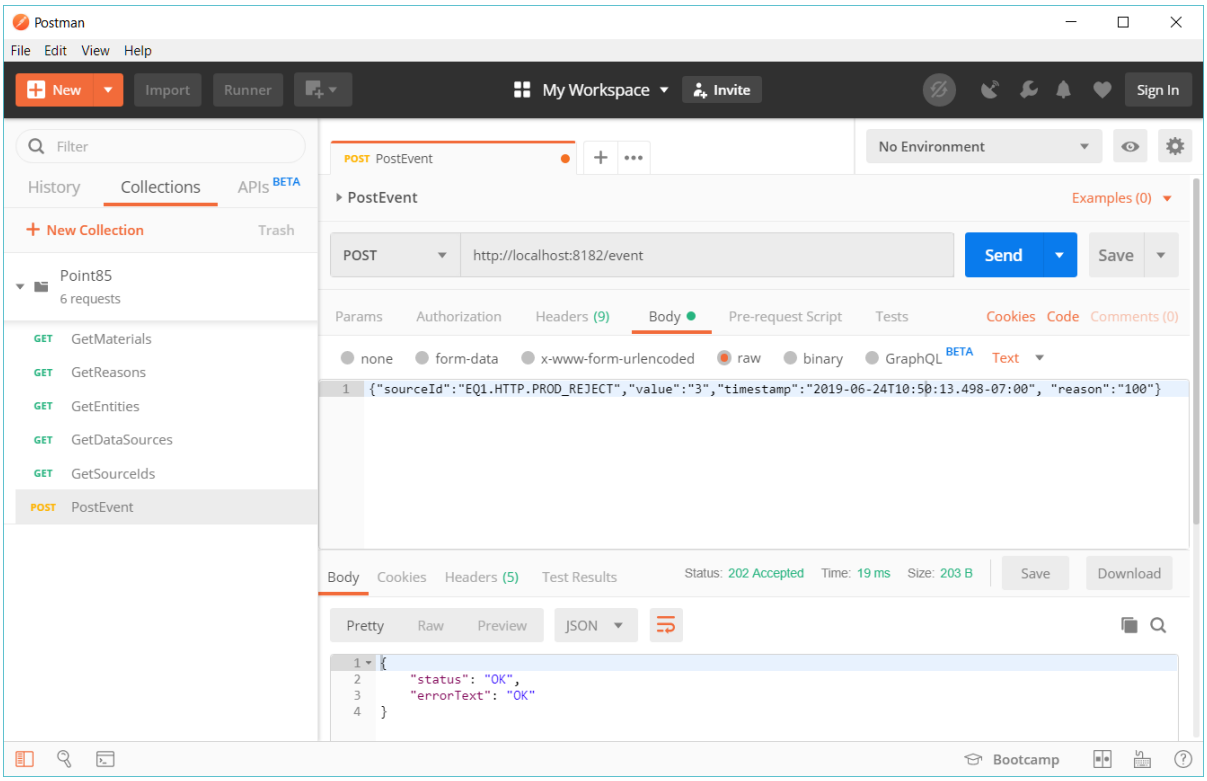
The Postman application can be used to post to the /event endpoint. For example, to post a new availability event:



or for a reject production count:

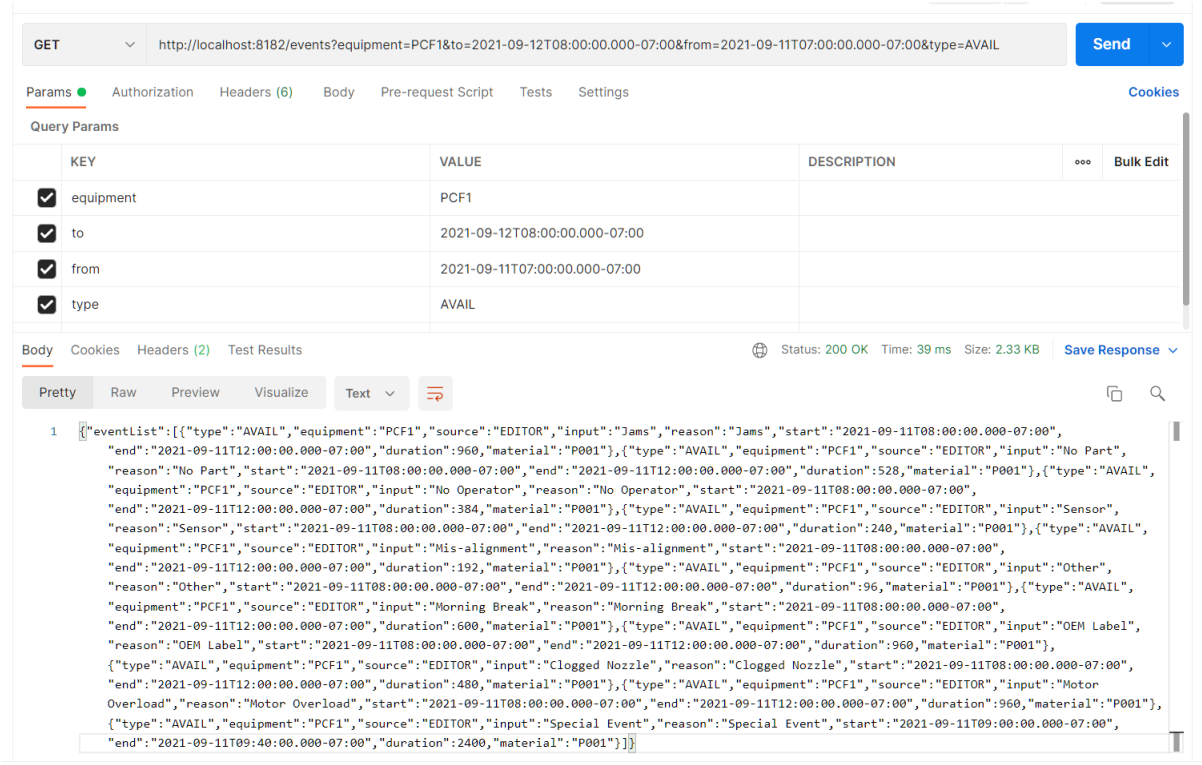
[7] default is today at midnight

[8] default is tomorrow at midnight

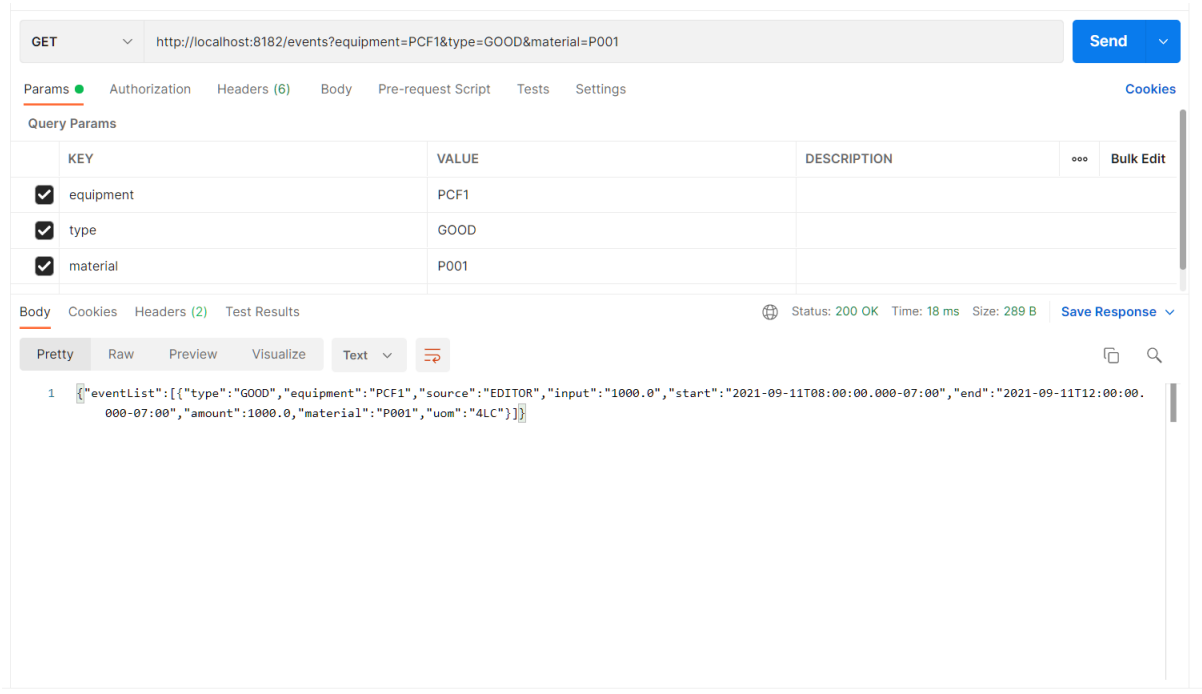


Events

The Postman application can be used to GET the /events endpoint. For example, for equipment “PCF1” producing any material in the time range 2021-09-11 07:00 through 2021-09-12 08:00 for availability events:

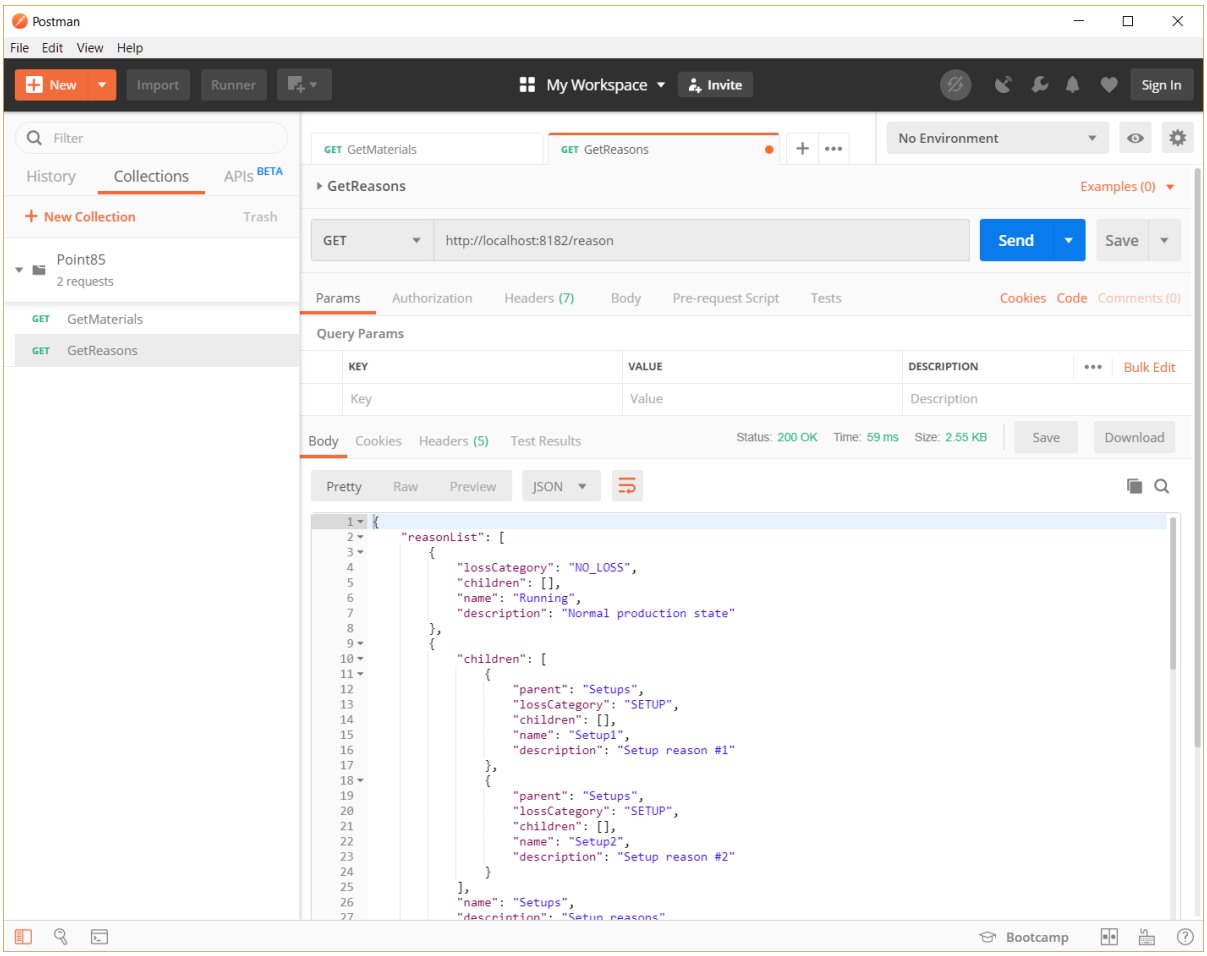


or for this equipment's good production of material P001 at any time:



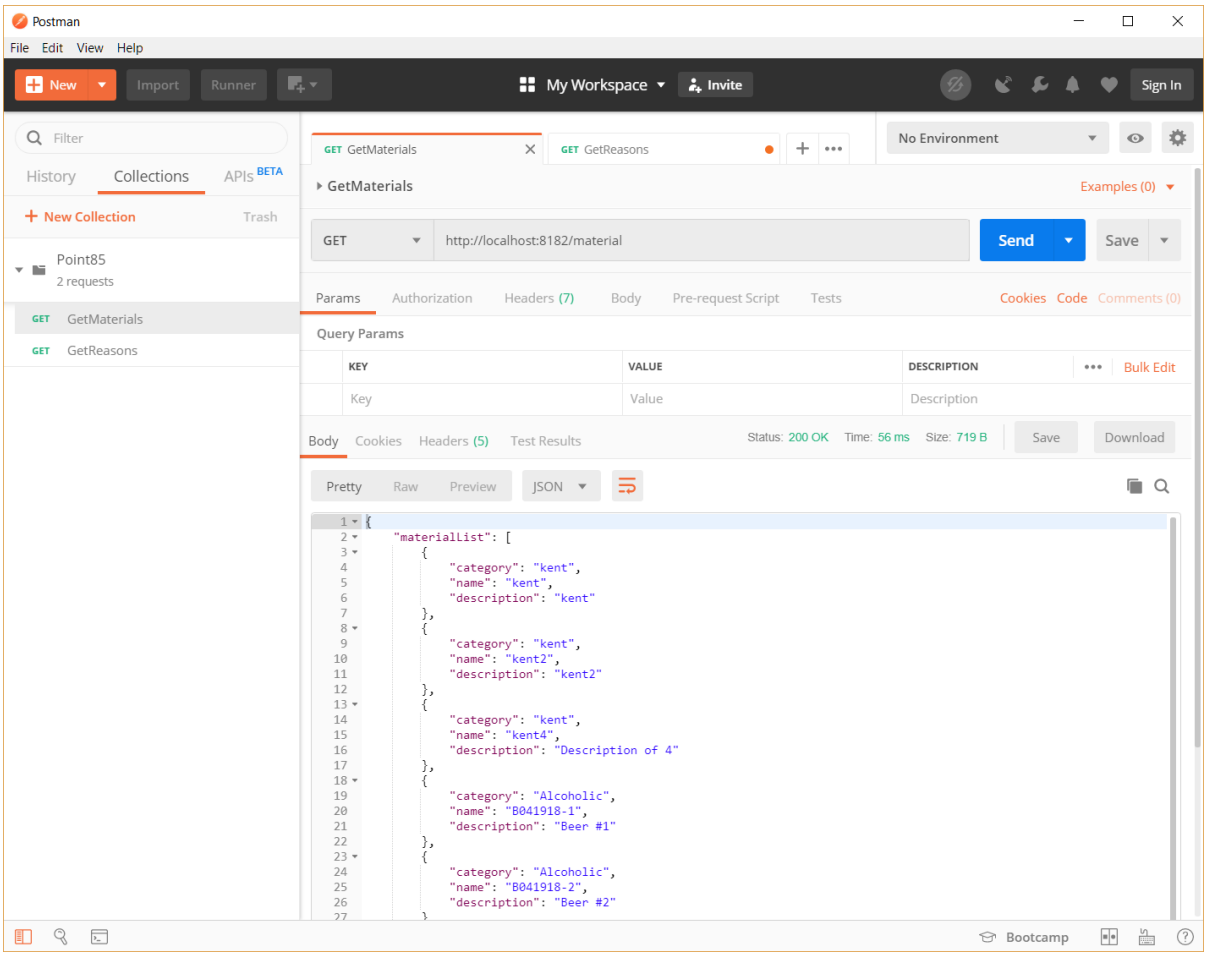
Reason

The Postman application can be used to explore the /reason endpoint. For example:



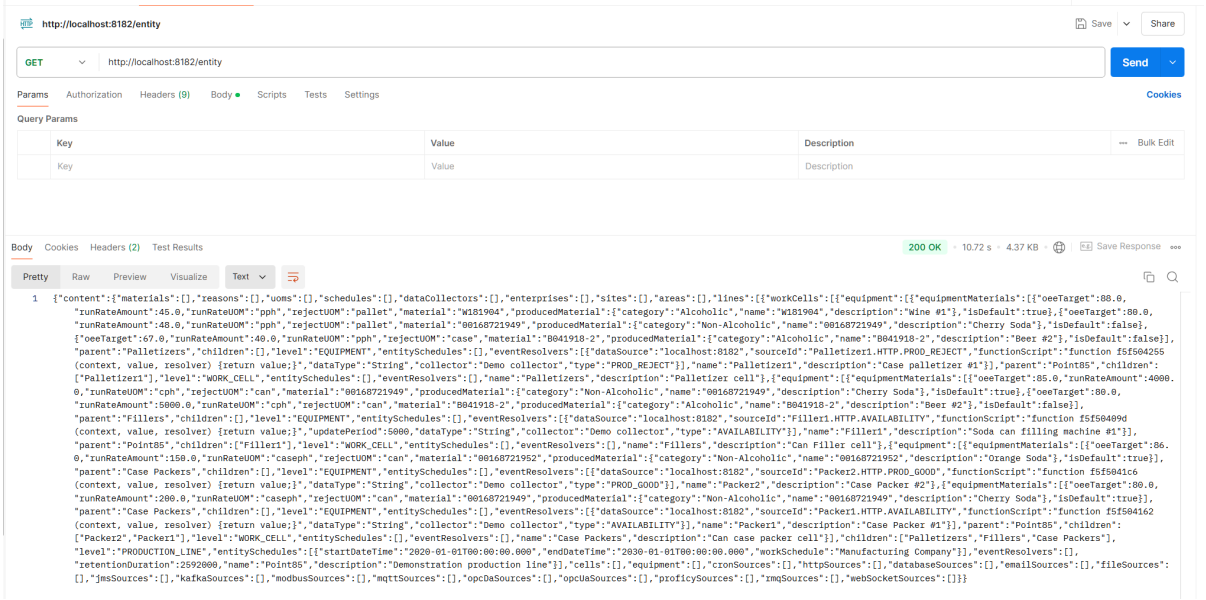
Material

The Postman application can be used to explore the /material endpoint. For example:



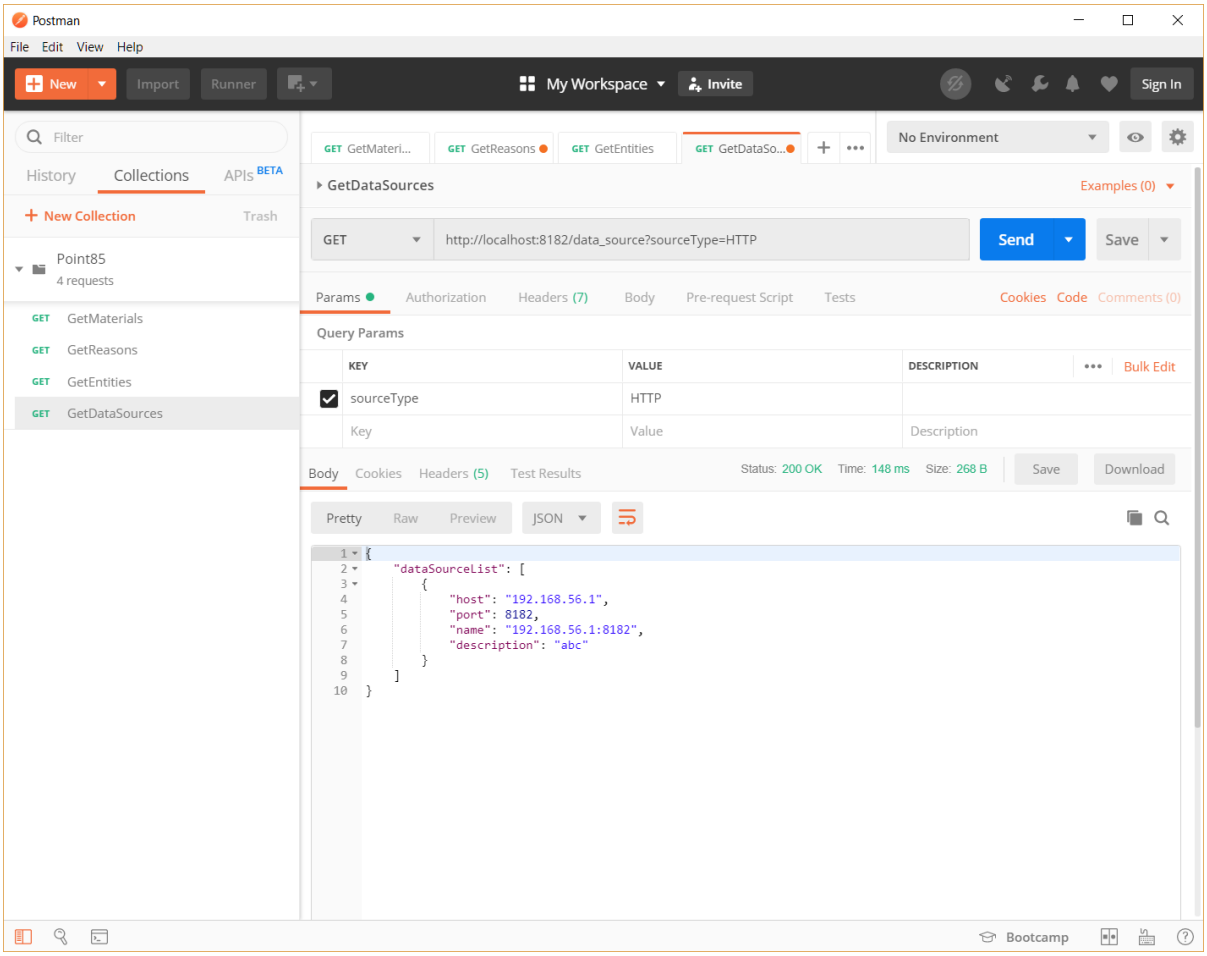
Entity

The Postman application can be used to explore the /entity endpoint. For example:



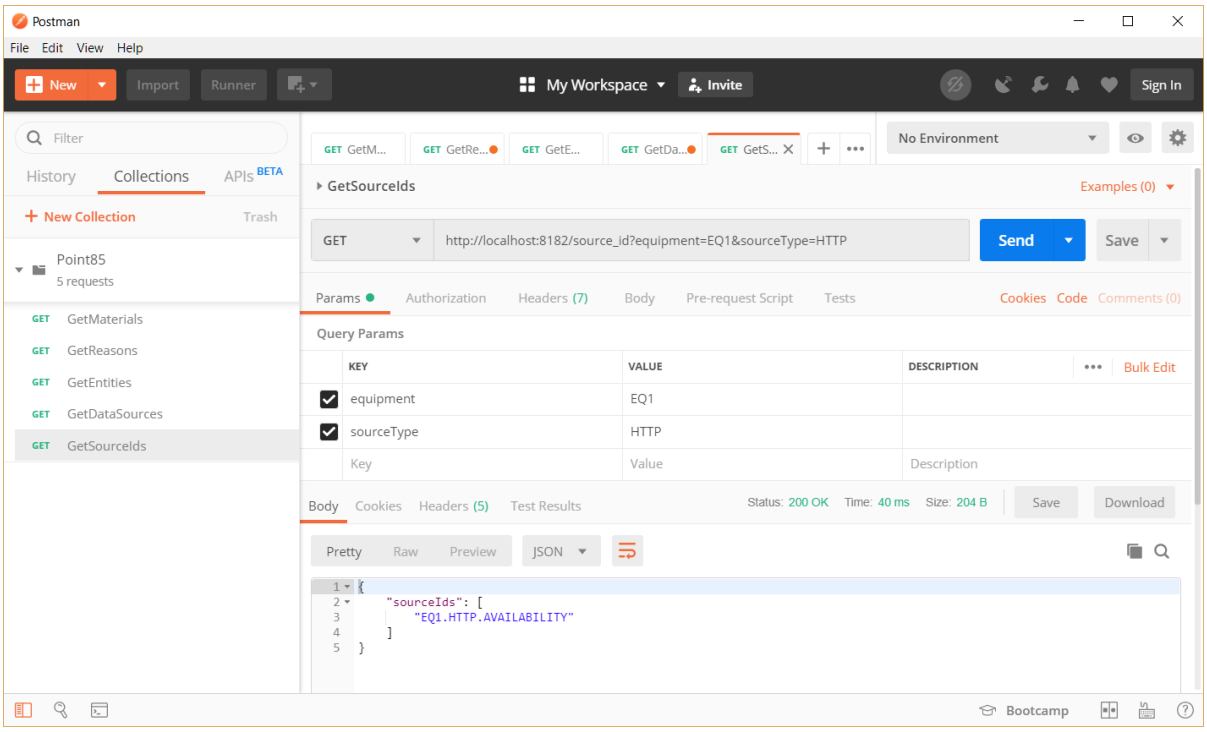
Data Source

The Postman application can be used to explore the /data_source endpoint for a particular type of data source. For example, for HTTP source:



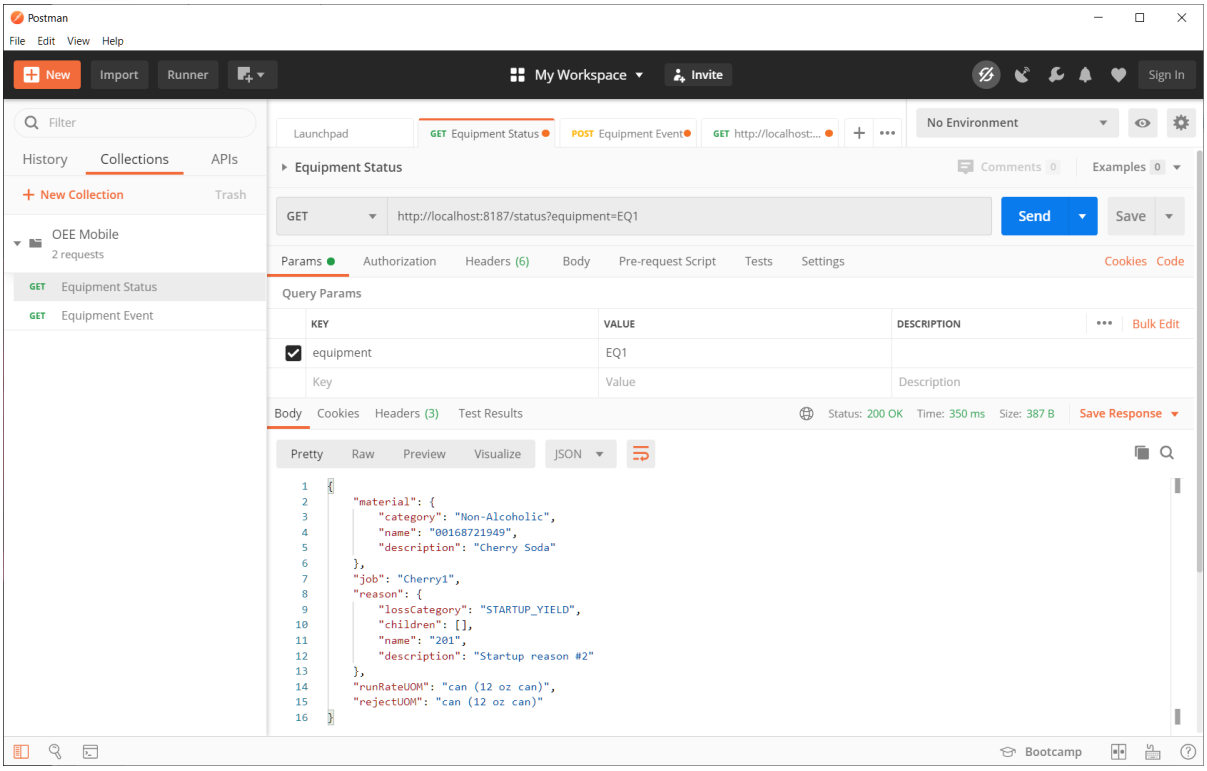
Source Id

The Postman application can be used to explore the /source_id endpoint for a particular equipement and type of data source. For example, for equipment “EQ1” and an HTTP source:



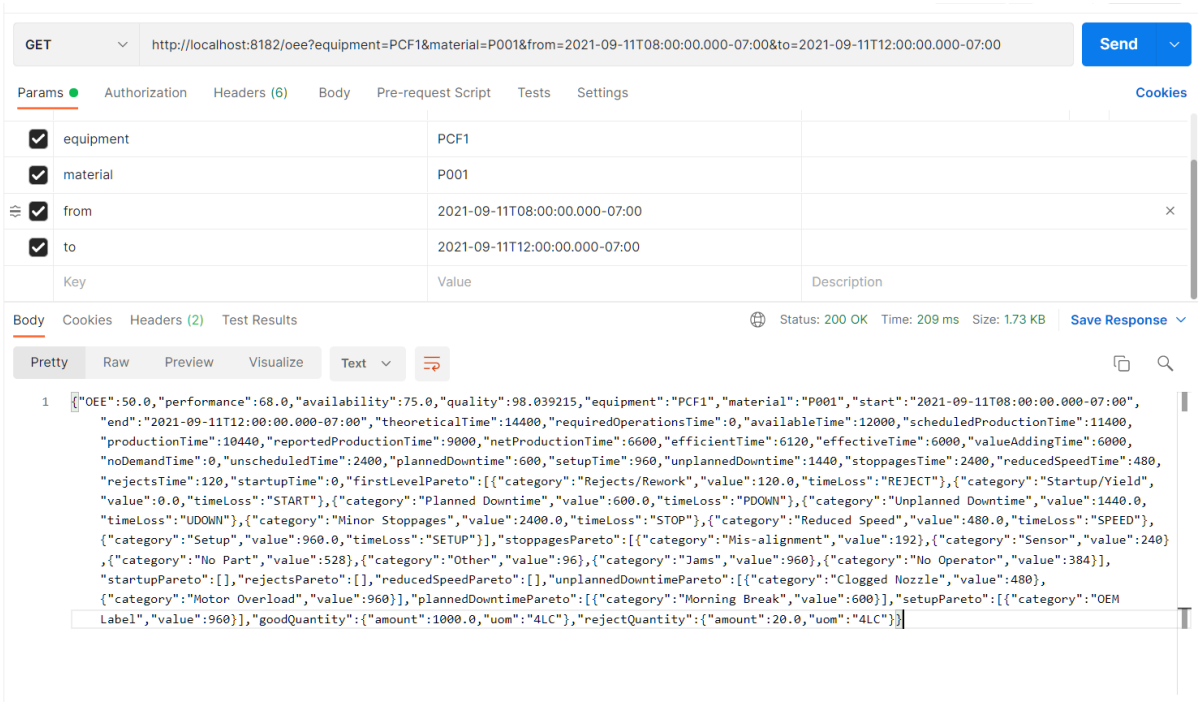
Equipment Status

The Postman application can be used to obtain the status of equipment including current production material, last availability reason and production units of measure. For example, for equipment “EQ1”:



OEE Calculations

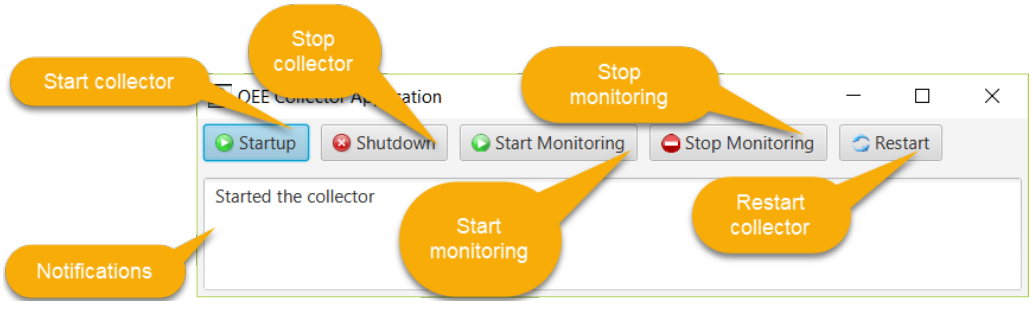
The Postman application can be used to obtain the OEE calculations, for example for equipment PCF1 producting material P001 in the time range from 2021-09-11 at 08:00 to 2021-09-11 at 12:00 with a -7 hour zone offset:



Testing Applications

Collector User Interface

This application provides a user interface for a data collector and is used for testing purposes. The application is launched from a host computer that has configured data sources. After the initial splash screen, the main form is displayed:



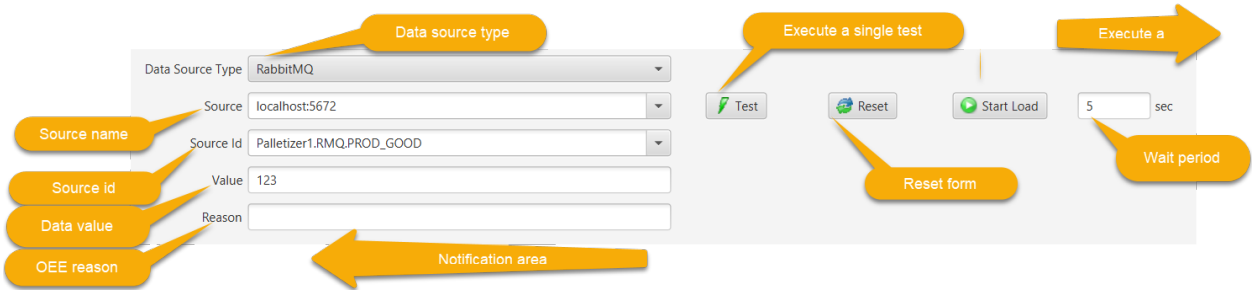
The actions are:

- *Startup*: Start the collector. It will connect to all OPC DA and OPC UA servers and be prepared to listen to HTTP requests as well as receive messages. It is in the monitoring state.
- *Shutdown*: Stop the collector. It is in the shutdown state.
- *Start Monitoring*: Start monitoring data input after having been stopped.
- *Stop Monitoring*: Stop monitoring all data inputs.
- *Restart*: Stop monitoring then restart monitoring.

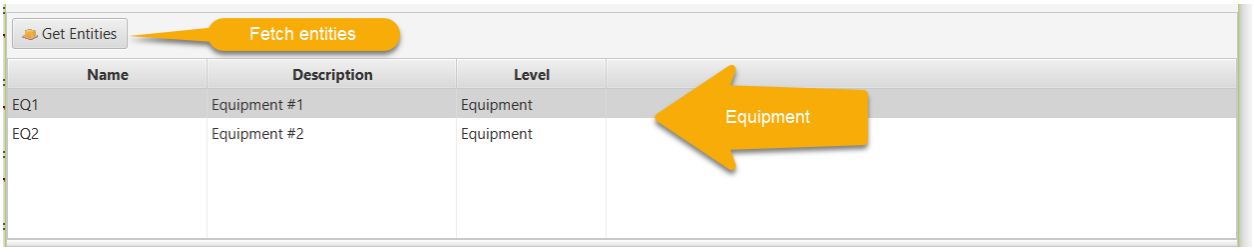
TESTER

This application provides a user interface to send HTTP or web socket requests and RabbitMQ, JMS, Kafka, email, MQTT and web socket messages. It can also write to OPC UA nodes and OPC DA and Proficy tags as well as to a database interface table, file, MODBUS field or schedule a cron job or email. It is used for testing purposes.

After the initial splash screen, the main form is displayed with a dropdown for selecting the data source type. The data sources are populated and then selected from the dropdown (e.g. RabbitMQ). The interface will be similar to:

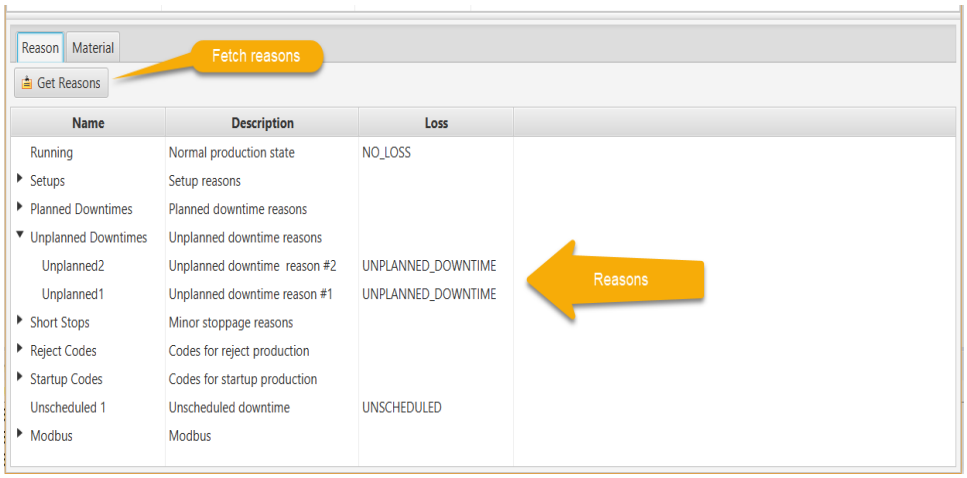


Clicking "Get Entities" will populate the entity tree table (in this example with equipment):



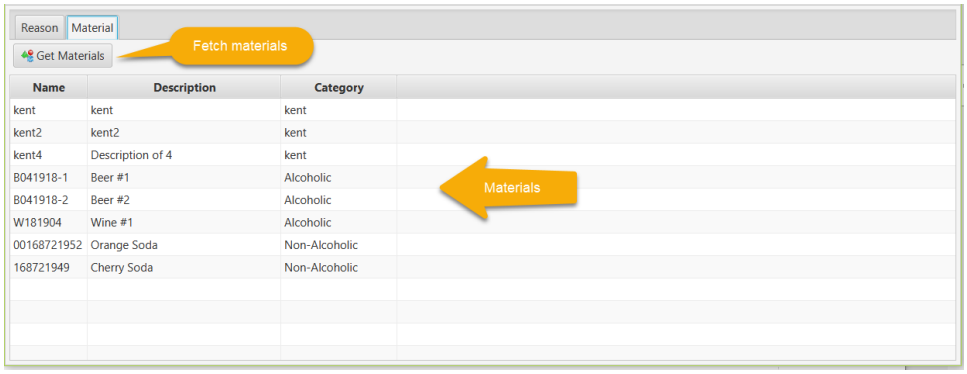
Selecting the equipment populates the source id dropdown for this source. Once a source identifier is selected, a value (and optional production reason) can be entered. Clicking the Test button writes to the selected source id as explained below.

For availability, the value is a reason. Reasons are displayed by clicking on the "Get Reasons" button:



Selecting a reason places it into the Value and Reason text fields.

Material changes require selecting the material. Clicking on the "Get Materials" button populates the table with materials:



Selecting a material places it into the Value text field.

Buttons

The Test button executes a one time test to send a message or write to a data source. It is used to verify the integrity of the connection. Data will be saved to the database with a running collector.

The *Reset* button re-queries the database for data sources and clears the source id and data entry fields.

The Start/Stop Load button initiates events with a period defined by the number of seconds entered into the edit control. It is used for long-term load testing of data sources such as HTTP that do not create events from a server (like OPC UA).

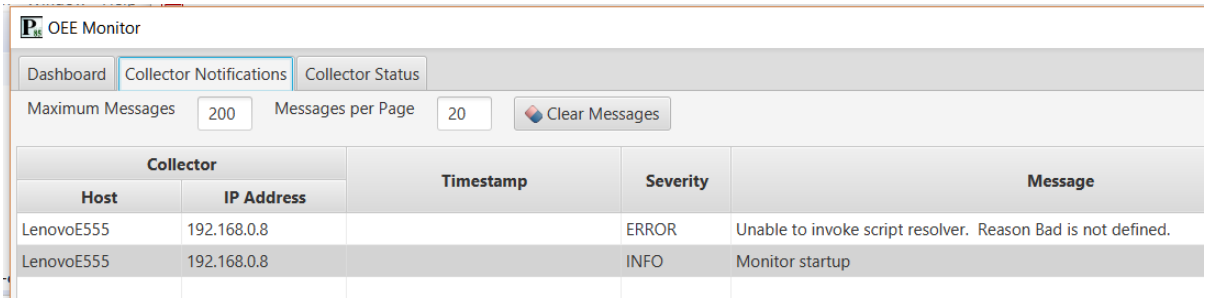
HTTP/Web Socket

The configured servers will be listed in the combobox. Choose one to send requests to it. The test button makes an HTTP POST equipment event request (or Web Socket request) to the selected server with the event data.

Messaging

Select either the RMQ, JMS, KAFKA, EMAIL or MQTT radio buttons. The configured servers will be listed in the combobox. Choose a broker and source id to send messages to it. Clicking the Test button will send an equipment event message to the selected broker with the event data.

Note that the Point85 log should be checked for a successful action, or a Monitor application launched during testing. If an exception occurs, it will appear in the list under "Collector Notifications". For example, reason "Bad' does not exist:



OPC DA

Select the OPC DA radio button. The configured OPC DA servers will be listed in the dropdown. Choose a server and source id (tag) to write to it. Clicking the Test button will write to this tag.

OPC UA

Select the OPC UA radio button. The configured OPC UA servers will be listed in the dropdown. Choose a server and source id (node) to write to it. Clicking the Test button will write to this node.

Database

Select the Database radio button. The configured database server JDBC connections will be listed in the source dropdown. Choose a server and source id to insert a row into the interface table. Clicking the Test button will write the row.

File

Select the File radio button. The file shares will be listed in the source dropdown. Choose a source and source id (folder). There is no text value to enter. Drop a file into the "ready" folder. The file will be parsed on the next polling cycle.

Modbus

Select the Modbus radio button. The Modbus masters will be listed in the source dropdown. Choose a source and source id (endpoint). Enter the value to be written to this register or coil. Clicking the Test button will write to this endpoint.

Cron Job

Select the Cron radio button. The job names will be listed in the source dropdown. Choose a job. Clicking the Test button will schedule the job for execution. When its trigger fires, an informational record will be written to the Point85 log.

Proficy Historian

Select the PROFICY radio button. The configured Proficy Historians will be listed in the dropdown. Choose a historian and source id (tag) to write to it. Clicking the Test button will write to this tag.

LOCALIZATION

All applications with user-visible text use resource bundles for localization. The locale is the default locale of the desktop or web server machine. Logging records are not localized. Each application has two default resource bundles, one for text named *<app name>Lang.properties* and one for errors/exceptions named *<app name>Error.properties* with US English text.

For the domain classes, in the \i18n directory in the oee-domain-<version>.jar file are the following files:

- DomainLang.properties and DomainError.properties

In the \org\point85\i18n directory in the oee-apps-<version>.jar file are the following files:

- DesignerLang.properties and DesignerError.properties
- MonitorLang.properties and MonitorError.properties
- CollectorLang.properties and CollectorError.properties
- OperatorLang.properties and OperatorError.properties
- TesterLang.properties and TesterError.properties

In the \WEB-INF\classes\i18n\ directory in the OEE-Operator-<version>.war file are the following files:

- WebOperatorLang.properties and WebOperatorError.properties

The jar and war files can be opened in a zip file manager (such as 7-Zip) and the default files edited or translated files added to the archive.

DATABASE

The Java Persistence 2.2 API (JPA) as implemented by the Hibernate ORM framework together with the Hikari connection pool is used to persist OEE information to the database.

Hibernate and JPA abstract-away database specific aspects of inserting, updating, reading and deleting records in the tables. The API is designed to work with any relational database supported by Hibernate.

DESIGN SCHEMA

The following sections document the design-time database schema. Note that foreign key relationships are not defined in the schema. Rather, referential integrity is enforced in java code in the PersistenceService class.

BREAK_PERIOD Table

This table contains the data for the break periods in a shift.

Column Name	Description
BREAK_KEY	Primary key
SHIFT_KEY	Primary key of shift
NAME	Break period name
DESCRIPTION	Break period description
START_TIME	Break period starting time of day
DURATION	Duration of break period
LOSS	OEE loss category for break period

COLLECTOR Table

This table contains the data describing each data collector.

Column Name	Description
COLLECT_KEY	Primary key
VERSION	Optimistic locking version
NAME	Collector name
DESCRIPTION	Collector description
STATE	Current state of collector
HOST	Computer host name or IP address of the collector
BROKER_TYPE	Messaging server type (RMQ, JMS or MQTT) - no longer used
BROKER_HOST	Messaging server host computer name or IP address - no longer used
BROKER_PORT	Messaging server TCP/IP port - no longer used
BROKER_USER	Messaging server user name - no longer used
BROKER_PWD	Messaging server user password - no longer used
SOURCE_KEY	Primary key of the notification messaging server (RMQ, JMS, KAFKA, EMAIL or MQTT)

DATA_SOURCE Table

This table contains the data describing each source of data for OEE events.

Column Name	Description
SOURCE_KEY	Primary key
VERSION	Optimistic locking version
NAME	Data source name
DESCRIPTION	Data source description
TYPE	Source type (e.g. OPC_DA, OPC_UA, HTTP or MESSAGING)
HOST	Computer host name or IP address of the source
USER_NAME	Name of authenticated user
PASSWORD	Password of authenticated user
PORT	Source TCP/IP port
END_PATH	OPC UA server path name; Email send security policy
SEC_POLICY	OPC UA server security policy; Email receive security policy; Kafka truststore password; web socket two-way authentication
MSG_MODE	OPC UA message mode; Email protocol; Kafka and MQTT client key password; web socket key password; HTTP OAuth client id
KEYSTORE	OPC UA and web socket java keystore name; Email send host; HTTP OAuth client secret
KEYSTORE_PWD	OPC UA and web socket java keystore password; Email send port

ENTITY_SCHEDULE Table

This table contains the data relating multiple effective work schedules for a plant entity.

Column Name	Description
ES_KEY	Primary key
WS_KEY	Primary key of work schedule
ENT_KEY	Primary key of plant entity
START_DATE_TIME	Effectivity starting date and time of day
END_DATE_TIME	Effectivity ending date and time of day

EQUIPMENT_MATERIAL Table

This table contains the data describing material produced by an equipment entity.

Column Name	Description
EM_KEY	Primary key

MAT_KEY	Primary key of produced material
EQ_KEY	Primary key of equipment
OEE_TARGET	Desired OEE
RUN_AMOUNT	Design speed or ideal run rate amount
RUN_UOM_KEY	Primary key of the design speed or ideal run rate unit of measure
REJECT_UOM_KEY	Primary key of the rejected/reworked material's unit of measure
IS_DEFAULT	True if this material is the default material produced by the equipment

EVENT_RESOLVER Table

This table contains the data describing the JavaScript event resolvers.

Column Name	Description
ER_KEY	Primary key
ENT_KEY	Primary key of equipment
SOURCE_KEY	Primary key of the data source
COLLECT_KEY	Primary key of data collector
SOURCE_ID	Data source identifier
SCRIPT	JavaScript function body
PERIOD	OPC DA update period, OPC UA publishing interval
ER_TYPE	Event resolver type (availability, production, material or job)
DATA_TYPE	The source's native data type

PLANT_ENTITY Table

This table contains the data for the plant entity hierarchy.

Column Name	Description
ENT_KEY	Primary key
VERSION	Optimistic locking version
NAME	Entity name
DESCRIPTION	Entity description
PARENT_KEY	Primary key of parent entity
HIER_LEVEL	Level in S95 hierarchy
RETENTION	Event data retention period

MATERIAL Table

This table contains the data for the materials produced by equipment.

Column Name	Description
MAT_KEY	Primary key
VERSION	Optimistic locking version
NAME	Material name
DESCRIPTION	Material description
CATEGORY	Material category

NON_WORKING_PERIOD Table

This table contains the data for the non-working and overtime periods in a work schedule.

Column Name	Description
PERIOD_KEY	Primary key
WS_KEY	Primary key of work schedule
NAME	Non-working period name
DESCRIPTION	Non-working period description
START_DATE_TIME	Period starting date and time of day
DURATION	Duration of non-working period
LOSS	Loss category for non-working period

REASON Table

This table contains the data for the OEE loss reasons.

Column Name	Description
REASON_KEY	Primary key
VERSION	Optimistic locking version
NAME	Reason name
DESCRIPTION	Reason description
PARENT_KEY	Parent reason
LOSS	Loss category

ROTATION Table

This table contains the data for a work schedule rotation in a shift.

Column Name	Description

ROTATION_KEY	Primary key
NAME	Rotation name
DESCRIPTION	Rotation description
WS_KEY	Primary key of work schedule

ROTATION_SEGMENT Table
This table contains the data for a portion of a work schedule rotation.

Column Name	Description
SEGMENT_KEY	Primary key
ROTATION_KEY	Primary key of rotation
SHIFT_KEY	Primary key of shift
SEQUENCE	The order of the sequence with a rotation
DAYS_ON	Number of days working the shift
DAYS_OFF	Number of days not working the shift

SHIFT Table
This table contains the data for a shift within a work schedule.

Column Name	Description
SHIFT_KEY	Primary key
WS_KEY	Primary key of work schedule
NAME	Shift name
DESCRIPTION	Shift description
START_TIME	Time of day when the shift begins
DURATION	Duration of shift

TEAM Table
This table contains the data for a team working a shift.

Column Name	Description
TEAM_KEY	Primary key
WS_KEY	Primary key of work schedule
ROTATION_KEY	Primary key of shift rotation
NAME	Team name
DESCRIPTION	Team description

ROTATION_START	Date when the rotation starts for this team

UOM Table

This table contains the data for a unit of measure.

Column Name	Description
UOM_KEY	Primary key
VERSION	Optimistic locking version
NAME	Unit of measure name
DESCRIPTION	Unit of measure description
SYMBOL	Unit of measure symbol
CATEGORY	'System' or user-defined category
UNIT_TYPE	Type or dimension, e.g. 'LENGTH'
UNIT	Identifier for system-defined UOM
CONV_FACTOR	Linear conversion factor
CONV_UOM_KEY	Abscissa UOM primary key
CONV_OFFSET	Linear conversion offset
BRIDGE_FACTOR	Linear conversion factor to a UOM in a different system, e.g foot to metre
BRIDGE_UOM_KEY	Target UOM primary key
BRIDGE_OFFSET	Target UOM offset
UOM1_KEY	Scalar, dividend, multiplicand or base UOM primary key
EXP1	First UOM exponent
UOM2_KEY	Divisor or multiplier UOM primary key
EXP2	Second UOM exponent

WORK_SCHEDULE Table

This table contains the data for a work schedule.

Column Name	Description
WS_KEY	Primary key
VERSION	Optimistic locking version
NAME	Schedule name
DESCRIPTION	Schedule description

Execution Schema

The following sections document the runtime database schema.

OEE_EVENT Table

This table contains the data for the OEE availability, production, setup and job change events. Rows are deleted according to the retention period for a plant entity.

Column Name	Description
EVENT_KEY	Primary key
ENT_KEY	Primary key of equipment plant entity
MATL_KEY	Primary key of produced material
REASON_KEY	Primary key of loss reason
SHIFT_KEY	Primary key of shift
TEAM_KEY	Primary key of team
AMOUNT	Amount of production
UOM_KEY	Primary key of unit of measure for production
JOB	Job/order
START_TIME	Local date and time of day for start of event
START_TIME_OFFSET	Time zone offset for start time
END_TIME	Local date and time of day for end of event
END_TIME_OFFSET	Time zone offset for end time
DURATION	Duration of event
EVENT_KEY	Type of event (availability, production, material or job change)
SOURCE_ID	Event resolver source identifier
REASON_KEY	Primary key of the reason for the production
COLLECTOR	Name of data collector

Schema Migration

The current schema version is noted in a comment in the create_tables.sql script on the first line, for example "-- SQL Server script file, schema version 5". If you are running an older version 4 schema, you will need to migrate the database from version 4 to version 5 by executing the migrate_v4_v5.sql script.

INSTALLATION

JAVAFX APPLICATIONS

The JavaFX applications are packaged in the oee-<version>.zip file in the OEE-Designer's dist folder. Expand the zip archive into the following folder structure:

- root: oee-apps-*<version>*.jar (JavaFX Designer, Monitor, Collector and Tester apps), oee-collector-*<version>*.jar (data collector in-process app), run-collector-app.bat (example Windows shell script for executing the data collector test UI), run-designer-app.bat (example Windows shell script for executing the designer application), run-monitor-app.bat (example Windows shell script for executing the monitor app), run-tester-app.bat (example Windows shell script for executing the tester application). The corresponding Unix bash scripts have the same file name with the ".sh" extension. The program arguments are:
    - [0]: application id (e.g. "DESIGNER")
    - [1]: JDBC connection string
    - [2]: user name
    - [3]: user password
    - [4]: optional name of a collector if more than JVM is running on the same host machine. Only applies to a collector application.
- config > logging: log4j2.xml configuration file
- database
    - import: example CSV import files (reasons.csv and materials.csv)
    - mssql: create_tables.sql and create_event_table.sql - SQL scripts to create the Microsoft SQL Server database tables
    - oracle: create_tables.sql and create_event_table.sql - SQL scripts to create the Oracle database tables
    - mysql: create_tables.sql and create_event_table.sql - SQL scripts to create the MySQL database tables
    - postgresql: create_tables.sql and create_event_table.sql - SQL scripts to create the PostgresQL database tables
    - hsql: create_tables.sql, create_event_table.sql, create_indexes.sql and create_event_table_indexes.sql - SQL scripts to create the HSQLDB database tables and indexes. Note that if the default local OEE database is being used, these scripts have already been executed. run_hsql_server.bat - Windows shell script to launch the HSQLDB server or ".sh" for Unix. The database files are in the "data" folder.

- lib: contains oee-domain-<*version*>.jar domain classes plus dependent jars
- lib/ext: folder for user-defined external jars for calling by JavaScript code
- logs: empty folder to contain the Log4j2 and Java Service Wrapper logging files
- wrapper
    - Win
        - bin: 64-bit Tanuki Java Service Wrapper community edition (wrapper.exe), install-oee-collector.bat (Windows shell script to install the data collector as a Windows service), uninstall-oee-collector.bat (Windows shell script to uninstall the data collector Windows service), oee-collector.bat (Windows shell script to execute the wrapper as a console app)
        - conf: wrapper.conf (Java Service Wrapper configuration file)
        - lib: wrapper.dll and wrapper.jar for Java Service Wrapper
    - MacOSX
        - bin: 64-bit Tanuki Java Service Wrapper community edition (wrapper), oee-collector (OS X shell script to execute the wrapper as a console app)
        - conf: wrapper.conf (Java Service Wrapper configuration file)
        - lib: libwrapper.jnilib and wrapper.jar for Java Service Wrapper
    - Linux
        - bin: 64-bit Tanuki Java Service Wrapper community edition as built by Simon Krenger (wrapper), oee-collector.sh (Linux bash shell script to execute the wrapper as a console app or deamon)
        - conf: wrapper.conf (Java Service Wrapper configuration file)
    - lib: libwrapper.so and wrapper.jar for Java Service Wrapper

The Java Service Wrapper wrapper.conf file requires that the following parameters be defined:

- wrapper.java.command: path to a Windows 64-bit Java 11 JRE compatible with the 64-bit Java Service Wrapper (or Unix 64-bit JRE compatible with a 64-bit Java Service Wrapper), e.g. for Windows:
    - set.JAVA_HOME=C:/jdk/jdk-11.0.13/jre
    - wrapper.java.command=%JAVA_HOME%/bin/java
- program arguments for the JDBC connection string and autenticated user. For example for Microsoft SQL Server running on localhost at port 1433 and connecting to the OEE database with SQL Server authenticated user “Point85” and password “Point85”:
    - wrapper.app.parameter.2=jdbc:sqlserver://localhost:1433;databaseName=OEE

- wrapper.app.parameter.3=Point85
- wrapper.app.parameter.4=Point85
- wrapper.app.parameter.5=<*optional name of collector*>

The 5th parameter is optional and specifies the name of a collector if more than one JVM is running on the same host machine. If not specified, then all collectors for the host machine will be run.

For Oracle 12c, the JDBC connection string for a ORCL SID would be similar to jdbc:oracle:thin:@localhost:1521:orcl SYSTEM admin (and jdbc:oracle:thin:@//localhost:1521/XE SYSTEM admin for the XE service). For MySQL, the JDBC connection string would be similar to jdbc:mysql://localhost:3306/oee Point85 Point85. For PostgresQL, the JDBC connection string would be similar to jdbc:postgresql://localhost/oee Point85 Point85 and for HSQLDB to jdbc:hsqldb:hsql://localhost/OEE Point85 Point85.

The steps to run the JavaFX applications are:

- JavaFX can be downloaded from Gluon at https://gluonhq.com/products/javafx/. The JAVAFX_HOME environment variable must be set.
- Create a database and then initialize it by executing the create_tables.sql script for SQL Server, Oracle, MySQL, PostgresQL or HSQLDB. If using an interface table as a data source, execute the create_event_table.sql script.
- Edit the run *.bat (or *.sh) scripts to set the JDBC connection and user credentials.
- Edit the config/logging/log4j2.xml file to set the location of the Point85.log file and logging levels.
- Execute the run-designer-app.bat (or .sh) script and define the plant model.
- Optionally, download and install the RabbitMQ broker from https://www.rabbitmq.com. The monitor application now can be used. In lieu of RabbitMQ, ActiveMQ or MQTT server can be used.
- Optionally execute the collector and tester applications.

DATA COLLECTOR

For your operating system (wrapper/MacOSX, wrapper/Linux or wrapper/Win), the in-process data collector can be deployed as follows:

- Edit the conf/wrapper.conf file to set JAVA_HOME to a 64-bit JRE or JDK and the database JDBC connection, user name and password properties (wrapper.app.parameter.2, 3 and 4). Parameter 5 is the optional name of a data collector if more than one collector is to run on the same machine.
- Execute the shell script to install the collector as a Windows service (Win/bin/install-oee-collector.bat and uninstall-oee-collector.bat), Unix daemon (MacOSX/bin/oee-collector.sh, Linux/bin/oee-collector.sh) or Windows console program (Win/bin/oee-collector.bat).

OPERATOR WEB APPLICATION

Download the operator web application's war file (OEE-Operator-<version>.war) from the latest Git release link for the Operations project (https://github.com/point85/OEE-Operations/releases) . The web.xml file in the war needs to be edited for the database connection information. To do this use a zip file manager application such as 7-Zip to edit WEB-INF/web.xml's jdbcConn, userName and password and collectorName parameters. For example:

```
<init-param>

  <param-name>jdbcConn</param-name>

  <param-value>jdbc:sqlserver://localhost:1433;databaseName=OEE</param-value>

</init-param>

<init-param>

  <param-name>userName</param-name>

  <param-value>Point85</param-value>

</init-param>

<init-param>

  <param-name>password</param-name>

  <param-value>Point85</param-value>

</init-param>

<init-param>
  <param-name>collectorName</param-name>
  <!-- ALL, NONE or a specific collector name -->
  <param-value>ALL</param-value>
</init-param>
```

The collectorName parameter can have the following values:

- ALL: all collectors defined for this machine will be run
- NONE: no collectors will be run even if some are defined for this machine
- <specific collector>: the named collector for this machine will be run

If using Apache Tomcat, run the Tomcat Web Application Manager. In the section of the web page titled "WAR file to deploy," browse to the war file and click the Deploy button. Under the Applications section, the path will be "/OEE-Operator-<version>". Note that if using the default maximum war file size of 50 MB, you will need to increase it to at least 60 MB in web.xml (<multipart-config> <max-file-size>).

The Point85 operator application URL is http://<host>:<port>/<war_file_name>/. If Tomcat is installed locally on the default port of 8080, the URL will be http://localhost:8080/OEE-Operator-<version>/.

PROJECT STRUCTURE

There are four Maven projects:

- OEE-Domain: domain logic with no user interface code
- OEE-Designer: JavaFX user interface code for the Designer, Monitor, Operator, Collector and Tester desktop applications
- OEE-Collector: the Windows service or Unix daemon in-process data collector with the Java Service Wrapper
- OEE-Operations: the Vaadin operator web application.

OEE-Domain

The OEE-Domain GitHub Maven project contains the OEE domain logic - there is no user interface code. The folders and files are:

- root: pom.xml, README.md, LICENSE, install-opc-da-jars.bat (Windows shell script for local Maven repository of OPC DA jars), install-opc-domain-jar.bat (Windows shell script for building the local Maven repository of the domain jar)
- lib: non-Maven Central jars stored in the “maven_repo” local repository.
    - opcda: j-Interop and openSCADA jars
    - oracle: Oracle database JDBC driver jar
- src
    - main > java: java code
    - main > resources: Unit.properties (resource bundle for units of measure), UomMessage.properties (resource bundle for UOM error exceptions), WorkScheduleMessage.properties (resource bundle for work schedule exceptions)

OEE-Designer

The OEE-Designer GitHub Maven project contains the OEE JavaFX user interface code for the Designer, Monitor, Collector and Tester applications. It depends on the OEE-Domain project. The folders and files are:

- root: pom.xml, README.md, LICENSE, build-jfx-app.bat (Windows shell script for building the JavaFX application jar), build-distro.bat (Windows shell script for building the oee.zip file distribution), run-designer-app.bat (example Windows shell script for executing the Designer application for a SQL Server database), run-monitor-app.bat (example Windows shell script for executing the Monitor application for a SQL Server database), run-collector-app.bat (example Windows shell script for executing the Collector application for a SQL Server database), run-tester-app.bat (example Windows shell script for executing the HTTP & messaging application for a SQL Server database), install-oracle-jdbc.bat (example Windows shell script for installing the Oracle JDBC driver jar in a local Maven repository). The Javafx applications can also be executed from a Unix bash shell from the corresponding ”.sh” scripts.

- config
    - logging: log4j2.xml configuration file
    - security: contains the point85-keystore.jks file and OPC UA keystores. The keystore and manager passwords are “Point85”.
- database
    - import: example CSV import files (reasons.csv and materials.csv)
    - demo: example database backup OeeDemoDB.p85x. This demo database can be used as a starting point for exploring Point85 OEE.
    - mssql: SQL scripts to create the Microsoft SQL Server database tables
    - oracle: SQL scripts to create the Oracle database tables
    - mysql: SQL scripts to create the MySQL database tables
    - postgresql: SQL scripts to create the PostgresQL database tables
    - hsql: SQL scripts to create the HSQLDB database tables and run a local server
- docs: this document (Point85 OEE User Guide.tmdx) and the getting started guide
- fxbuild: build.xml (ant build file)
    - archive: Point85 OEE.zip (distribution file)
    - dist: build folder
- src
    - main > java: java code
    - main > resources: .css stylesheets, .png images, .otf fonts
- ssl: example Windows shell scripts for creating an X509 certificate and a java keystore

OEE-Collector

The OEE-Collector GitHub Maven project contains the source code for the Windows service or Unix daemon in-process data collector with the Java Service Wrapper. It depends on the OEE-Domain project.

- root: pom.xml, README.md, LICENSE, build-collector.bat (Windows shell script for building the collector jar)
- config
    - logging: log4j2.xml configuration file
    - security: point85.keystore (java OPC UA keystore file)

        - help: CronExpression.html help file for the cron data source editor. This file can be translated if desired.
- src
    - main > java: java code
- wrapper
    - Win
        - bin: 64-bit Tanuki Java Service Wrapper community edition (wrapper.exe), install-oee-collector.bat (Windows shell script to install the data collector as a Windows service), uninstall-oee-collector.bat (Windows shell script to uninstall the data collector Windows service), oee-collector.bat (Windows shell script to execute the wrapper as a console app)
        - conf: wrapper.conf (Java Service Wrapper configuration file)
        - lib: wrapper.dll and wrapper.jar for Java Service Wrapper
    - MacOSX
        - bin: 64-bit Tanuki Java Service Wrapper community edition (wrapper), oee-collector (OS X shell script to execute the wrapper as a console app or daemon)
        - conf: wrapper.conf (Java Service Wrapper configuration file)
        - lib: libwrapper.jnilib and wrapper.jar for Java Service Wrapper
    - Linux
        - bin: 64-bit Tanuki Java Service Wrapper community edition as built by Simon Krenger (wrapper), oee-collector.sh (Linux bash shell script to execute the wrapper as a console app or deamon)
        - conf: wrapper.conf (Java Service Wrapper configuration file)
    - lib: libwrapper.so and wrapper.jar for Java Service Wrapper

OEE-Operations

The OEE-Operations GitHub Maven project contains the source code for the Vaadin operator web application.

- root: pom.xml, README.md, LICENSE, build-operator.bat (Windows shell script for building the operator war)
- dist: war distribution file
- src
    - main > java: java code

- main > webapp > WEB-INF: web.xml deployment descriptor file. The jdbcConn, userName and password must be defined.

BUILDING

Builiding Point85 OEE projects require that the ant and Maven tools be installed first. In the Maven setting.xml file, change to the local repository "maven_repo":

```
<localRepository>C:/maven_repo</localRepository>
```

The build steps are:

1. Execute install-opc-da-jars.bat in the OEE-Domain project to install the OPC DA jars into a local Maven repository
2. Execute install-oee-domain-jar.bat in the OEE-Domain project to build the domain jar and install it into a local Maven repository
3. Execute install-orcle-jdbc.bat in the OEE-Designer project to install the Oracle JDBC driver jar into a local Maven repository
4. Execute build-jfx-app.bat in the OEE-Designer project to build the OEE JavaFX application
5. Execute build-collector.bat in the OEE-Collector project to build the data collector
6. Execute build-operator.bat in the OEE-Operations project to build the operator Vaadin application. The war file is in the "dist" folder and can be deployed to a web server.
7. Execute build-distro.bat in the OEE-Designer project to build the oee-<version>.zip distribution file in the "dist" folder. The zip archive contains the JavaFX applications and the in-process Java Service Wrapper collector.

Note that these steps are combined in the build-all.bat script.

The Ant build script creates a lib/ext folder for external jar files that need to be on the classpath in order to be visible to JavaScript.

CONTRIBUTING TECHNOLOGY

The author wishes to acknowledge the following software upon which the OEE applications are built.

https://github.com/eclipse/milo	Milo is an open-source implementation of OPC UA. It includes a high-performance stack (channels, serialization, data structures, security) as well as client and server SDKs built on top of the stack.
j-interop.org	j-Interop is a Java Open Source library (under LGPL) that implements the DCOM wire protocol (MSRPC) to enable development of Pure, Bi-Directional, Non-Native Java applications which can interoperate with any COM component.

openscada.org	openSCADA is an open source Supervisory Control And Data Acquisition System. It is platform independent and based on a modern system design that provides security and flexibility at the same time.
https://www.eclipse.org/jetty/	Eclipse Jetty provides a Web server and javax.servlet container, plus support for HTTP/2, WebSocket, OSGi, JMX, JNDI, JAAS and many other integrations.
http://hibernate.org	Hibernate ORM enables developers to more easily write applications whose data outlives the application process. As an Object/Relational Mapping (ORM) framework, Hibernate is concerned with data persistence as it applies to relational databases (via JDBC).
https://brettwooldridge.github.io/HikariCP/	HikariCP is a “zero-overhead” production-quality connection pool.
https://www.rabbitmq.com/	RabbitMQ is an open source message broker software (sometimes called message-oriented middleware) that originally implemented the Advanced Message Queuing Protocol (AMQP) and has since been extended with a plug-in architecture to support Streaming Text Oriented Messaging Protocol (STOMP), Message Queuing Telemetry Transport (MQTT), and other protocols.
www.erlang.org	Erlang is a general-purpose, concurrent, functional programming language, as well as a garbage-collected runtime system.
http://activemq.apache.org	Apache ActiveMQ ™ is the most popular and powerful open source messaging and Integration Patterns server.
https://www.eclipse.org/paho	The Eclipse Paho project provides open-source client implementations of MQTT and MQTT-SN messaging protocols aimed at new, existing, and emerging applications for the Internet of Things (IoT).
https://mvnrepository.com/artifact/com.google.code.gson/gson	Gson JSON serializer and deserializer
https://vaadin.com/	Vaadin is an open-source platform for web application development.
https://wrapper.tanukisoftware.com	The Java Service Wrapper (JSW) enables a Java Application to be run as a Windows Service or UNIX Daemon. It also monitors the health of your Application and JVM. The 64-bit Windows version is by Simon Krenger (https://www.krenger.ch/blog/java-service-wrapper-3-5-38-for-windows-x64/)
https://github.com/HanSolo/tilesfx	A JavaFX library containing tiles that can be used for dashboards.
https://www.bouncycastle.org	Bouncy Castle is a collection of APIs used in cryptography.

https://logging.apache.org/log4j/2.x/	Apache Log4j 2 is an upgrade to Log4j that provides significant improvements over its predecessor, Log4j 1.x, and provides many of the improvements available in Logback while fixing some inherent problems in Logback's architecture.
https://mvnrepository.com/artifact/org.slf4j	Log4j logging facade
https://github.com/steveohara/j2mod	j2Mod is an enhanced Modbus library implemented in the Java programming language.
https://www.quartz-scheduler.org/	Quartz is a richly featured, open source job scheduling library that can be integrated within virtually any Java application - from the smallest stand-alone application to the largest e-commerce system. Quartz can be used to create simple or complex schedules for executing tens, hundreds, or even tens-of-thousands of jobs; jobs whose tasks are defined as standard Java components that may execute virtually anything you may program them to do.
https://flutter.dev	Flutter is an open-source UI software development kit created by Google. It is used to develop applications for Android, iOS, Linux, Mac, Windows, Google Fuchsia and the web from a single codebase. The mobile applications are programmed in the Dart 3.0+ language.
http://kafka.apache.org/	Apache Kafka is an open-source stream-processing software platform developed by the Apache Software Foundation, written in Scala and Java. The project aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds.
javax.mail	Maven repository for API interface and javax.mail implementation.
GE Proficy Historian REST API	Historian 8.0 - REST APIs Reference Manual.pdf
Nashorn JavaScript engine	Nashorn engine is an open source implementation of the ECMAScript Edition 5.1 Language Specification. It also implements many new features introduced in ECMAScript 6.
https://github.com/TooTallNate/Java-WebSocket	This repository contains a barebones WebSocket server and client implementation written in 100% Java. The underlying classes are implemented java.nio, which allows for a non-blocking event-driven model (similar to the WebSocket API for web browsers).
https://pub.dev/packages/datetime_picker_formfield_new	A TextFormField that emits DateTimes and helps show Material, Cupertino, and other style picker dialogs.

https://pub.dev/packages/arborio	An elegant, flexible Treeview with Animation. Display hierarchical data in Flutter.
https://pub.dev/packages/flutter_riverpod	A reactive caching and data-binding framework.

REFERENCES

[Kennedy] *Understanding, Measuring, and Improving Overall Equipment Effectiveness.* How to Use OEE to Drive Significant Process Improvement, Ross Kenneth Kennedy, CRC Press, 2018.

[Stamatis] *The OEE Primer*, Understanding Overall Equipment Effectiveness, Reliability, and Maintainability, D.H. Stamatis, CRC Press, 2010.

[Koch] *OEE Industry Standard 2003*, Defining OEE for Optimal Loss Visualization, www.OEEFoundation.org, Arno Koch, 2003.

[Kraus] *OEE for Operators*, Overall Equipment Effectiveness, The Productivity Development Team, Productivity Press, 1999.