
Tempus Documentation

Release 1.3

Hashmap, Inc

May 14, 2019

Contents

1	About	1
2	Features	1
3	Contents	2
3.1	Quick Start	2
3.2	Features	28
3.3	Installation	63
3.4	API	65
3.5	Security	78
3.6	Administration UI	83
3.7	Tempus Rule Engine Reference	146
3.8	Data Model Reference	159
3.9	Development	168
3.10	Gateway	188
3.11	Metadata	190
3.12	Indices and tables	194

1 About

Tempus Cloud is an open IIoT/IoT framework (Field/Edge to Cloud/DC) and rapid analytics application creator that provides a spectrum of outcome-based experiences on an unmodified data stream and does not force customers into a proprietary approach but focuses on a use-case driven approach for top-line/bottom-line benefits.

2 Features

- **Business Unit Management:** Tempus cloud allows for authorization by business unit to allow for segmentation of users, while still allowing all the data to be viewed in context if required.

- **Asset Management:** Tempus Cloud has a flexible asset management model that is based on a graph representation. This allows for different users targeting different types of analysis to arrange the data as they wish. For example, if User A would like to look at their data geographically, they might choose to create relations on a Country/Region/City basis, while User B has the freedom to organize assets by workflows and facilities.
- **Device Management:** Devices can be managed and controlled via Tempus Cloud. All of the communication is facilitated via standard MQTT. This can be secured via X.509 certificates or simple access token for authentication and authorization, and TLS for encryption of the data in motion.

Additionally, there is a comprehensive attribute management system in place, that allows for attributes to be provided by the client, the server, or shared between the two.
- **Visualization:** Tempus includes a flexible Visualization and dashboarding framework. These visualizations can be created and modified on the fly allowing users to see what they need to see, when they need to see it.
- **Rules Engine:** There is a flexible rules engine that allows for the invocation of plugins to allow for multiple outcomes based on a combination of rule sets. These outcomes can be (but not limited to):

Alarming Data Routing Emailing / Notifications RPC communication to the Edge Modification of the data flow from the edge

Rapid Computation Deployment: Tempus Cloud also has the capability to orchestrate and manage Spark computations on the data. This allows for a deeper analysis of the streaming data, in which the results can be visualized in real time, or even sent back to the edge.

3 Contents

3.1 Quick Start

These guides will help you get started as quickly as possible.

Linux Installation

This guide describes how to install Tempus Cloud on a Linux based server machine. Instructions below are provided for Ubuntu 16.04 and CentOS 7. These instructions can be easily adapted to other similar operating systems.

Hardware requirements

To run Tempus Cloud and third-party components on a single machine you will need at least 1Gb of RAM.

Third-party components installation

Java

Tempus Cloud service is running on Java 8. Although you are able to start the service using OpenJDK, the solution is actively tested on Oracle JDK.

Follow this instructions to install Oracle JDK 8:

- [Ubuntu 16.04](#)
- [CentOS 7](#)

Please don't forget to configure your operating system to use Oracle JDK 8 by default. Corresponding instructions are in the same articles listed above.

[Optional] External database installation

Tempus Cloud is able to use a SQL or Cassandra database. By default, Tempus Cloud uses embedded HSQLDB instance which is convenient for evaluation or development purposes. If this is your first experience with Tempus Cloud we recommend to skip this step and use the embedded database. Alternatively, you can configure your platform to use either scalable Cassandra DB cluster or various SQL databases. If you prefer to use an SQL database, we recommend PostgreSQL.

SQL Database: PostgreSQL

NOTE: This is an **optional** step. It is required only for production usage. You can use embedded HSQLDB for platform evaluation or development Instructions listed below will help you to install PostgreSQL.

CentOS

```
Copy resources/postgresql-ubuntu-installation.sh to clipboard
# Update your system
sudo apt-get update
# Install packages
sudo apt-get install postgresql postgresql-contrib
# Initialize PostgreSQL DB
sudo service postgresql start
# Optional: Configure PostgreSQL to start on boot
sudo systemctl enable postgresq
```

Ubuntu

```
sudo apt-get update
sudo apt-get install postgresql postgresql-contrib
sudo service postgresql start
```

Once PostgreSQL is installed you may want to create a new user or set the password for the the main user. See the following guides for more details:

- [Using postgresql roles and databases](#)
- [Changing the postgres user password](#)

When it's done, connect to the database and create Tempus Cloud DB:

```
psql -U postgres -d postgres -h 127.0.0.1 -W
CREATE DATABASE Tempus;
\q
```

NoSQL Database: Cassandra

NOTE: This is an **optional** step. It is required only for production usage. You can use embedded HSQLDB for platform evaluation or development instructions listed below will help you to install Cassandra.

CentOS

```

sudo touch /etc/yum.repos.d/datastax.repo
echo '[datastax]' | sudo tee --append /etc/yum.repos.d/datastax.repo > /dev/null
echo 'name = DataStax Repo for Apache Cassandra' | sudo tee --append /etc/yum.repos.d/
↳datastax.repo > /dev/null
echo 'baseurl = http://rpm.datastax.com/community' | sudo tee --append /etc/yum.repos.
↳d/datastax.repo > /dev/null
echo 'enabled = 1' | sudo tee --append /etc/yum.repos.d/datastax.repo > /dev/null
echo 'gpgcheck = 0' | sudo tee --append /etc/yum.repos.d/datastax.repo > /dev/null

# Cassandra installation
sudo yum install dsc30
# Tools installation
sudo yum install cassandra30-tools
# Start Cassandra
sudo service cassandra start
# Configure the database to start automatically when OS starts.
sudo chkconfig cassandra on

```

Ubuntu

```

# Add cassandra repository
echo 'deb http://www.apache.org/dist/cassandra/debian 311x main' | sudo tee --append /
↳etc/apt/sources.list.d/cassandra.list > /dev/null
curl https://www.apache.org/dist/cassandra/KEYS | sudo apt-key add -
sudo apt-get update
## Cassandra installation
sudo apt-get install cassandra
## Tools installation
sudo apt-get install cassandra-tools

```

Tempus Cloud Service Installation

CentOS

```

sudo rpm -Uvh tempus-1.3.1.rpm

```

Ubuntu

```

sudo dpkg -i tempus-1.3.1.deb

```

[Optional] Configure Tempus Cloud to use an external database

NOTE: This is an **optional** step. It is required only for production usage. You can use embedded HSQLDB for platform evaluation or development

Edit Tempus Cloud configuration file

```

sudo nano /etc/Tempus/conf/Tempus.yml

```

Comment '# HSQLDB DAO Configuration' block.

```

# HSQLDB DAO Configuration
#spring:
#  data:

```

(continues on next page)

(continued from previous page)

```
# jpa:
#   repositories:
#     enabled: "true"
# jpa:
#   hibernate:
#     ddl-auto: "validate"
#   database-platform: "org.hibernate.dialect.HSQLDialect"
# datasource:
#   driverClassName: "${SPRING_DRIVER_CLASS_NAME:org.hsqldb.jdbc.JDBCdriver}"
#   url: "${SPRING_DATASOURCE_URL:jdbc:hsqldb:file:${SQL_DATA_FOLDER:/tmp}/TempusDb;
→sql.enforce_size=false}"
#   username: "${SPRING_DATASOURCE_USERNAME:sa}"
#   password: "${SPRING_DATASOURCE_PASSWORD:}"
```

For *PostgreSQL*:

Uncomment ‘# PostgreSQL DAO Configuration’ block. Be sure to update the postgres databases username and password in the bottom two lines of the block (here, as shown, they are both “postgres”).

```
# PostgreSQL DAO Configuration
spring:
data:
  jpa:
    repositories:
      enabled: "true"
jpa:
  hibernate:
    ddl-auto: "validate"
    database-platform: "org.hibernate.dialect.PostgreSQLDialect"
datasource:
  driverClassName: "${SPRING_DRIVER_CLASS_NAME:org.postgresql.Driver}"
  url: "${SPRING_DATASOURCE_URL:jdbc:postgresql://localhost:5432/Tempus}"
  username: "${SPRING_DATASOURCE_USERNAME:postgres}"
  password: "${SPRING_DATASOURCE_PASSWORD:postgres}"
```

For *Cassandra DB*: Locate and set database type configuration parameter to ‘cassandra’.

```
database:
  type: "${DATABASE_TYPE:cassandra}" # cassandra OR sql
```

Memory Update for Slow Machines (1GB of RAM)

We recommend to use embedded HSQLDB or PostgreSQL DB in this setup. We don’t recommend to use Cassandra on machines with less than 4GB of RAM.

For Tempus Cloud service:

```
# Update Tempus memory usage and restrict it to 256MB in /etc/Tempus/conf/Tempus.conf
export JAVA_OPTS="${JAVA_OPTS} -Xms256M -Xmx256M"
```

Run installation script

Once Tempus Cloud service is installed, you can execute the following script:

```
# --loadDemo option will load demo data: users, devices, assets, rules, widgets.
sudo /usr/share/Tempus/bin/install/install.sh --loadDemo
```

Start Tempus Cloud service

Execute the following command to start Tempus Cloud:

```
sudo service tempus start
```

Once started, you will be able to open Web UI using the following link:

```
http://localhost:8080/
```

NOTE: Please allow up to 90 seconds for the Web UI to start

Troubleshooting

Tempus Cloud logs are stored in the following directory:

```
/var/log/tempus
```

You can issue the following command in order to check if there are any errors on the backend side:

```
cat /var/log/tempus/tempus.log | grep ERROR
```

Create a Tenant

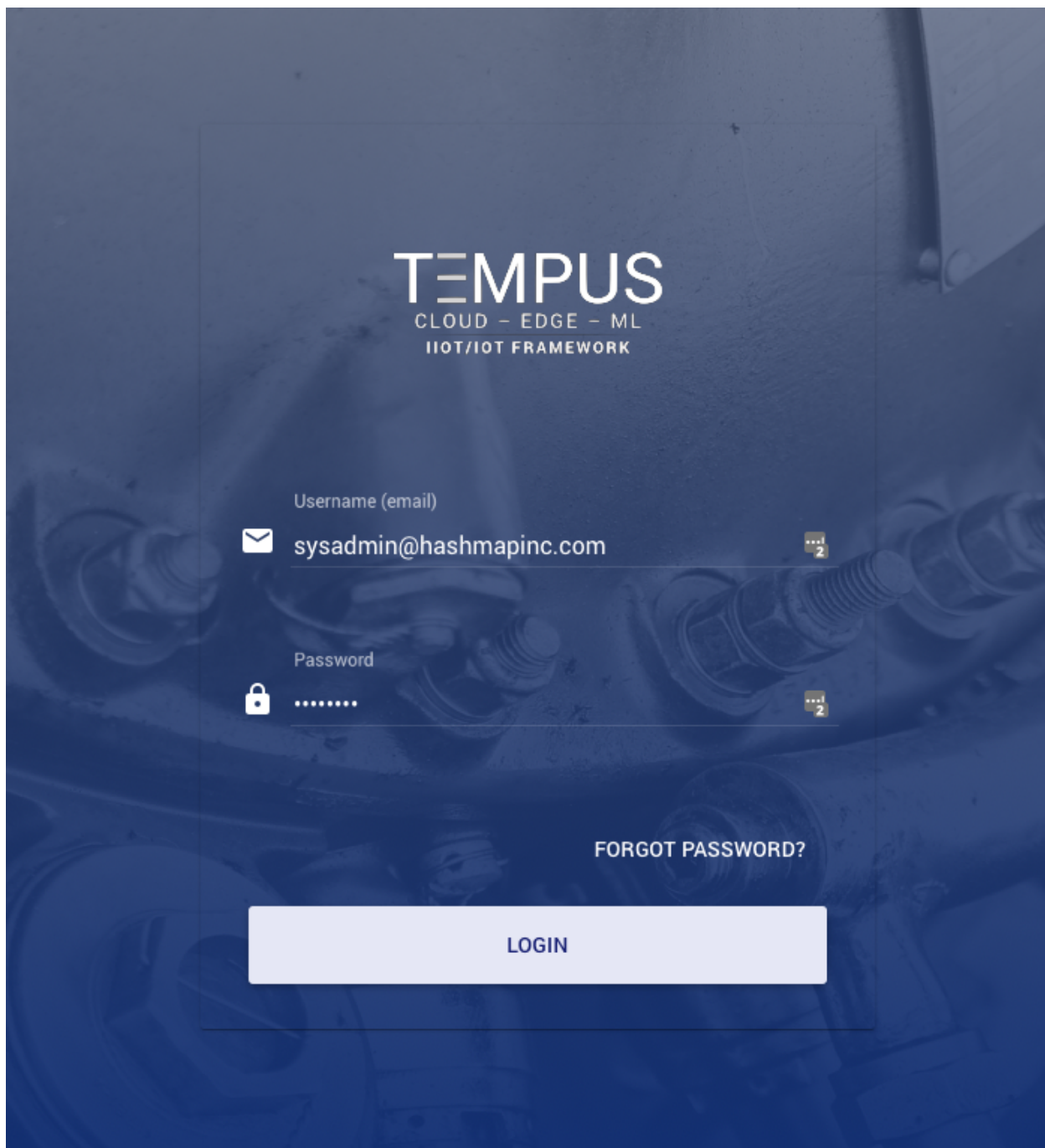
This will guide you through how to create a tenant in Tempus Cloud.

Setup and Requirements

If you don't have access to a running Tempus Cloud instance please follow this guide:

Create a Tenant

1. Navigate to the login page for Tempus Cloud (by default http://host_ip:8080, where host ip is the address running Tempus)

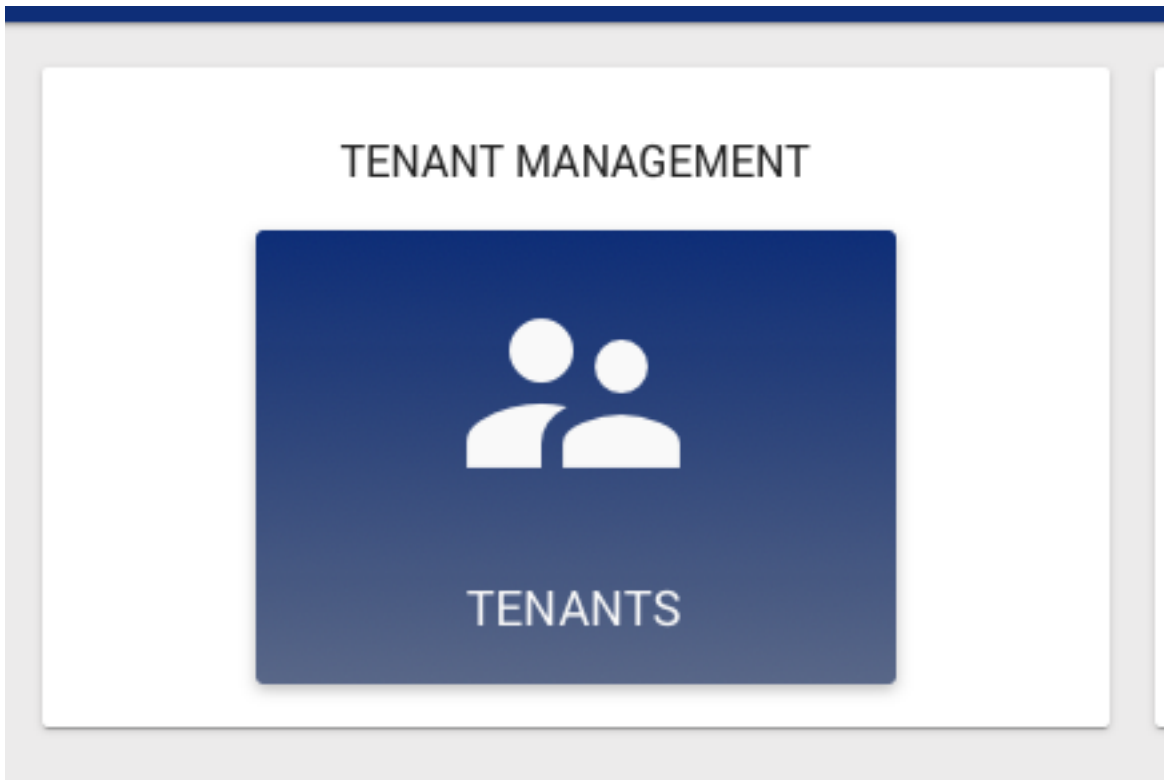


2. **Login with the following credentials:**

- User: sysadmin@hashmapinc.com
- Password: sysadmin

Please Note: This password **MUST** be changed prior to production

3. Click on **Tenants** under **TENANT MANAGEMENT**



4. Click on the “+” icon at the bottom right of the screen to add a tenant



5. Fill in the required information, and then click Add

Add Tenant?×

Title *

Demo

Description

Demo Tenant

Country

United States

City

Houston

State

TX

Postal code

77494

Address

1212 Hashmap Road

Address 2

Phone

713-555-5555

Email

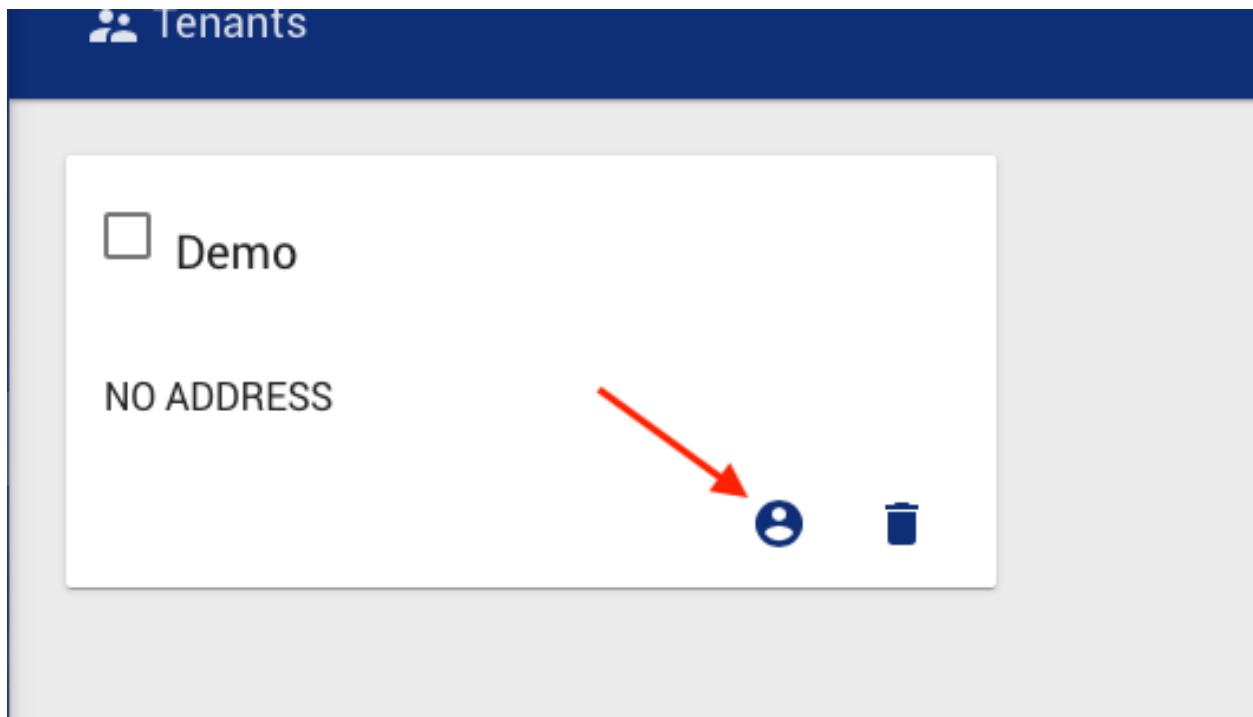
demo@hashmapinc.com

ADD

CANCEL

Create a Tenant Administrator

1. After performing the steps to create a tenant, navigate to the Tenants page and click on the user button of the tenant card that you want to create the administrator for.



2. Click on the “+” symbol at the bottom right to create an administrator
3. Enter in the Tenant admin info as required

Add User

Email *

First Name

Last Name

Description

Activation method

Display activation link

ADD

CANCEL

The user can be activated in one of 2 ways:

- **Display Activation Link:** Useful for when the email server has not been configured, or it is not a real user
- **Send Activation Mail:** Will send an email to the user with the supplied email address to complete the activation process

We will choose the Display Activation Link option.

4. Follow the activation link and create a password, and then click Create Password

Create Password

Password

.....

Password again

.....

CREATE PASSWORD **CANCEL**

5. You have now created the Tenant admin and should be logged in as the Tenant Administrator after activation

Getting Started with Visualization

The goal of this guide is for you to collect and visualize some IoT device data using Tempus Cloud. This guide will help you with:

- Provisioning a device
- Manage device credentials
- Push data from a device to the Tempus cloud instance using MQTT
- Create a dashboard to visualize the data

Setup and Requirements

If you don't have access to a running Tempus Cloud instance please follow this guide:

Make sure you have created a tenant by following this guide:

Login as the tenant administrator

The first step is to login into administration Web UI. If you are using local Tempus cloud installation you can login to administration Web UI using the account you created in the quickstart guide for creating a tenant.

TEMPUS

CLOUD – EDGE – ML
IIOT/IOT FRAMEWORK

Username (email)



demo@hashmapinc.com



Password



....

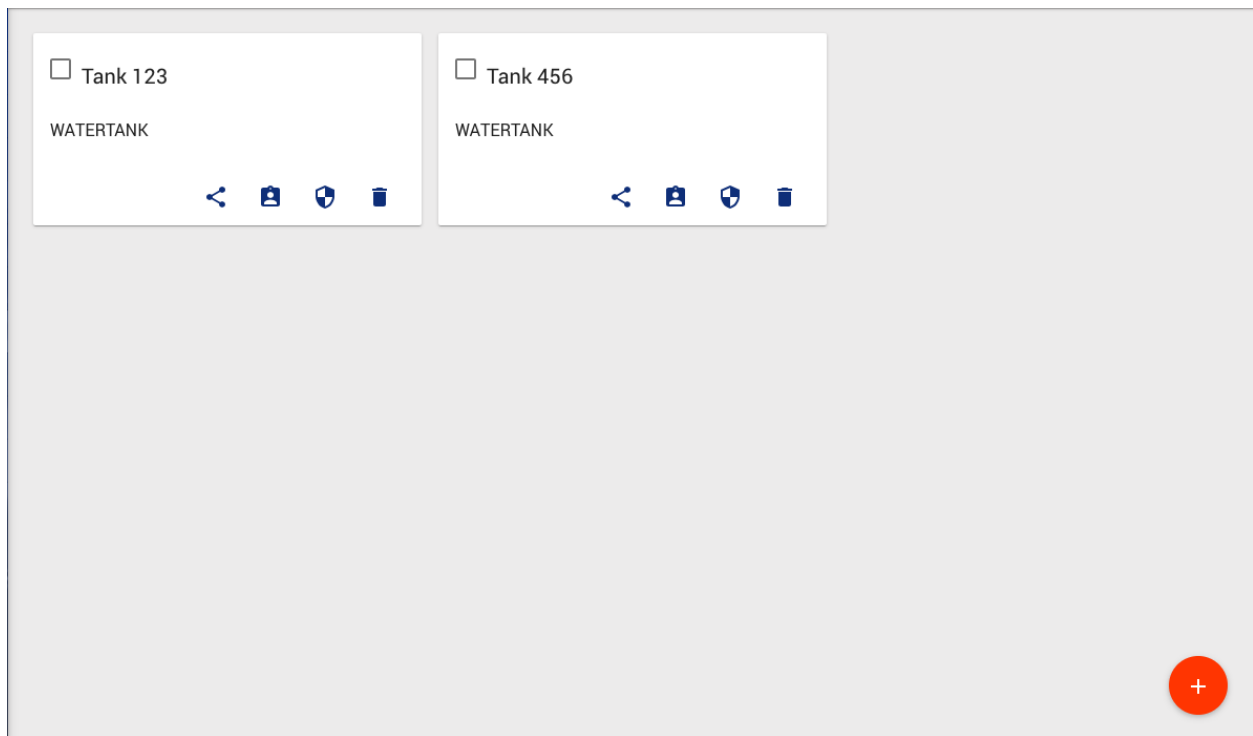


[FORGOT PASSWORD?](#)

LOGIN

Provision your Device

1. Open the Devices panel and click on the “+” button at the bottom-right corner of the page.



2. Populate and save device name (for example, “SN-001”). It will be referred to later as `$DEVICE_NAME`. Device names must be unique. Populating device name based on a unique serial number or other device identifier is generally a good idea. Click “Add” button will add corresponding device card to the panel.

Add Device?×

Name *

Demo123

Device type *

DemoDevice

☐ Is gateway

Description

ADD

CANCEL

Manage device credentials

1. Click on the device card created in the previous step. This action will open “device details” panel. Click on the “manage credentials” button on the top of the panel. This action will open a popup window with device credentials.

Devices

Tenant administrator

DEMO123

Device details

Tank 123

WATERTANK

DETAILS ATTRIBUTES LATEST TELEMETRY ALARMS EVENTS RELATIONS

MAKE DEVICE PUBLIC ASSIGN TO BUSINESS UNIT MANAGE CREDENTIALS DELETE DEVICE

COPY DEVICE ID COPY ACCESS TOKEN

Name *
Demo123

Device type *
DemoDevice

☐ Is gateway

Description

2. Device credentials window will show auto-generated device access token that you can change. Please save this device token. It will be referred to later as **\$ACCESS_TOKEN**

Device Credentials

Credentials type

Access token

Access token *

9xJpRG53XGxKY1PWZqTb

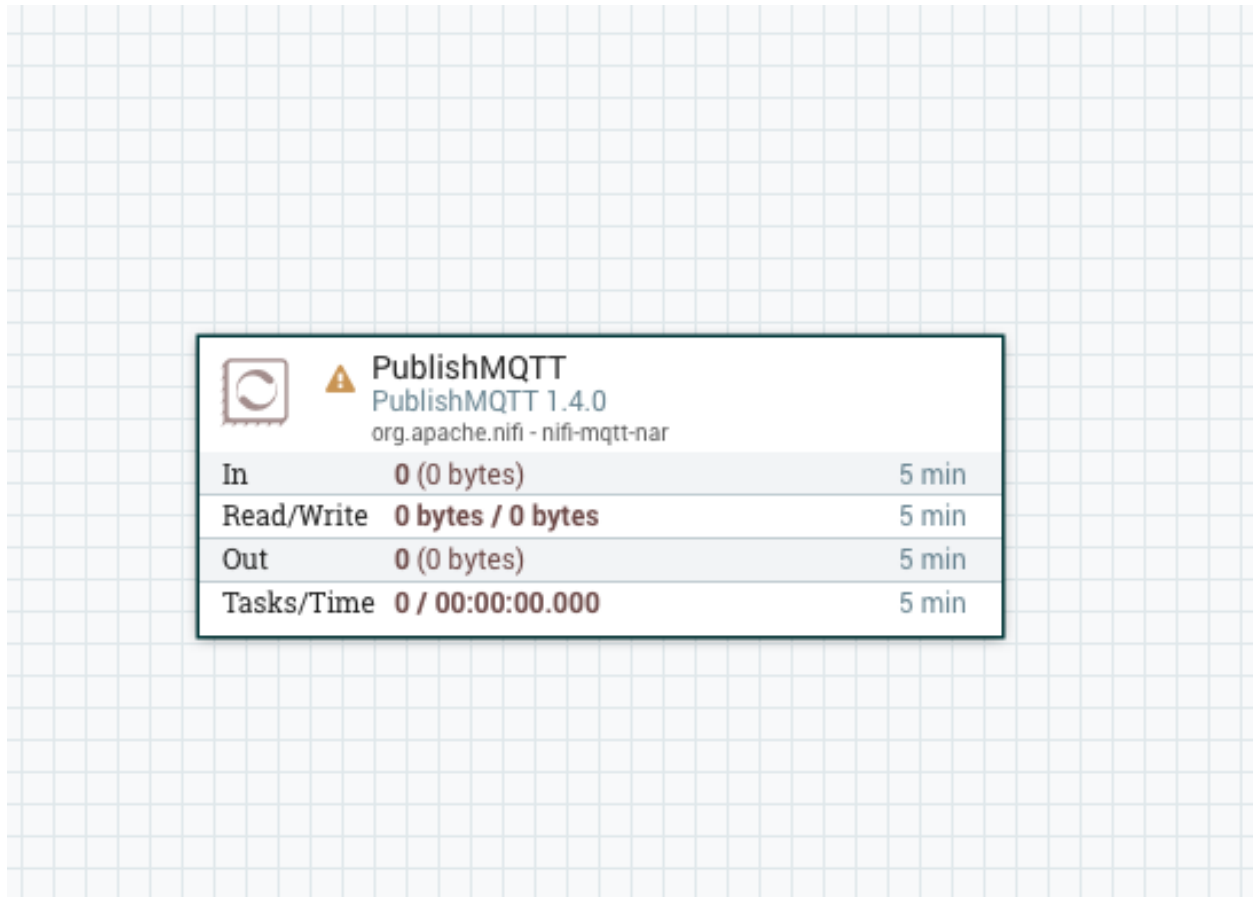
20 / 20

SAVE CANCEL

Pushing Data From the Device - Nifi

Attributes

1. Navigate to a running instance of Apache Nifi
2. Add a Publish MQTT processor to the Canvas



3. Right-click on the PublishMQTT processor and click configure, then click on the Properties tab
4. **Fill in the information as follows**
 - For the Broker URI put in `tcp://host_ip:1883` (1883 is the default mqtt port)
 - For the Client ID enter **nifi**
 - For the User Name enter the **\$ACCESS_TOKEN** that was created earlier
 - For the password enter a space character
 - For the Topic use **v1/devices/me/attributes**
 - For the QoS enter 0
 - For Retain Message enter **false**
5. Click on settings and tick the boxes to auto-terminate the failure and success relationships, as this is the last processor in the flow, and click **Apply**
6. Add a GenerateFlowFile processor to the Canvas

7. Right-click on the GenerateFlowFile processor and click configure, then click on the Properties tab
8. **Fill in the information as follows**
 - For Custom Text Enter: {"firmware_version":"1.0.1", "serial_number":"SN-001"}
9. **Click on the Scheduling tab**
 - Enter 1 sec for the **Run Schedule**
10. Click **Apply**
11. Start both Processors
12. Navigate to the Tempus Devices panel
13. Click on the device card that you published data to
14. Click on **attributes**
15. You should see the 2 attributes appear in the pane as below:

DEMO123

Device details

?

×

DETAILS

ATTRIBUTES

LATEST TELEMETRY

ALARMS

EVENTS

F

>

Entity attributes scope

Client attributes

Client attributes

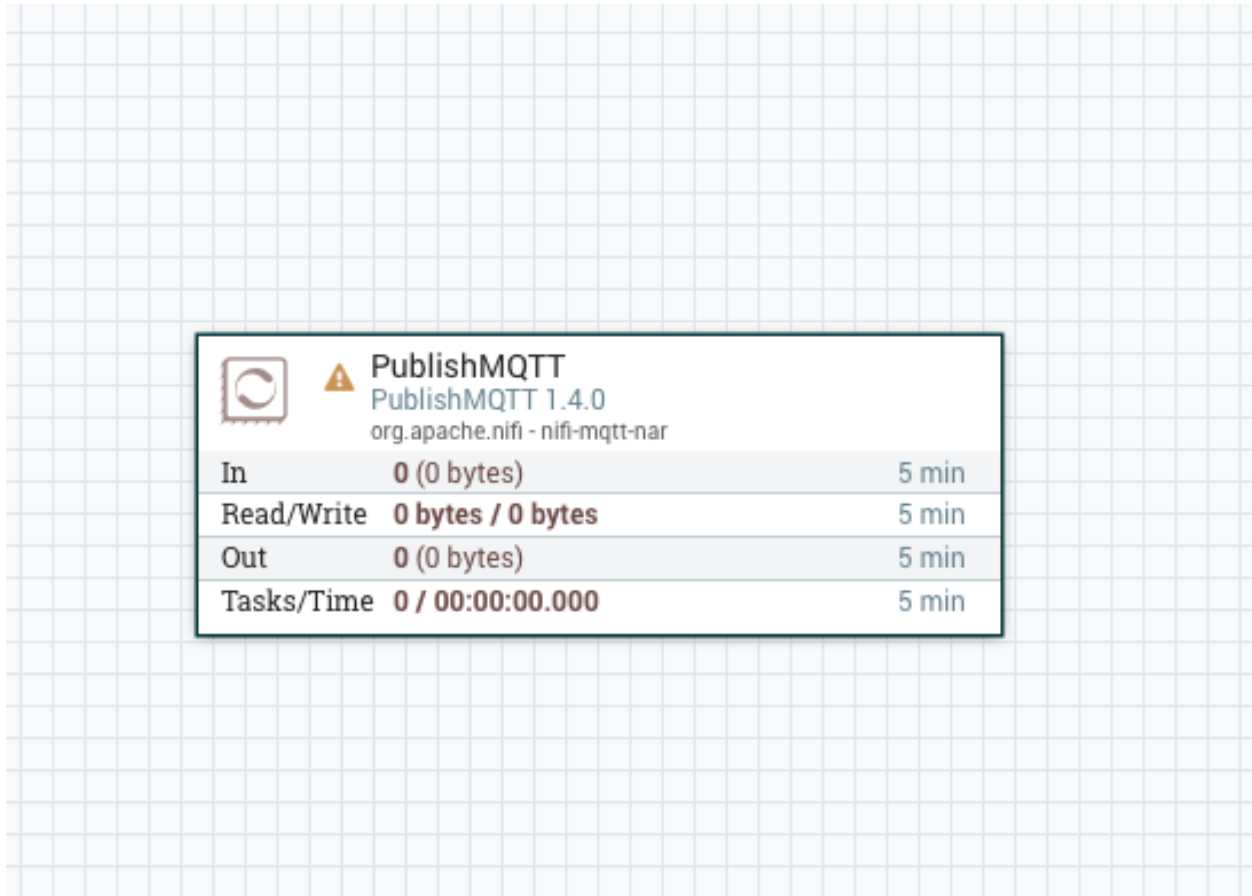
🔍

<input type="checkbox"/>	Last update time	Key ↑	Value
<input type="checkbox"/>	2018-03-14 09:57:08	deviceType	WaterTank
<input type="checkbox"/>	2018-03-14 09:57:08	tankId	123

Page: 1 ▾ Rows per page: 5 ▾ 1 - 2 of 2 < >

Telemetry

1. Navigate to a running instance of Apache Nifi
2. Add a Publish MQTT processor to the Canvas



3. Right-click on the PublishMQTT processor and click configure, then click on the Properties tab
4. **Fill in the information as follows**
 - For the Broker URI put in `tcp://host_ip:1883` (1883 is the default mqtt port)
 - For the Client ID enter **nifi**
 - For the User Name enter the **\$ACCESS_TOKEN** that was created earlier
 - For the password enter a space character
 - For the Topic use **v1/devices/me/telemetry**
 - For the QoS enter 0
 - For Retain Message enter **false**
5. Click on settings and tick the boxes to auto-terminate the failure and success relationships, as this is the last processor in the flow, and click **Apply**
6. Add a GenerateFlowFile processor to the Canvas
7. Right-click on the GenerateFlowFile processor and click configure, then click on the Properties tab
8. **Fill in the information as follows**

- For Custom Text Enter: {"temperature":21, "humidity":55.0, "active": false}

9. Click on the **Scheduling tab**

- Enter 1 sec for the **Run Schedule**

10. Click **Apply**

11. Start both Processors

12. Navigate to the Tempus Devices panel

13. Click on the device card that you published data to

14. Click on **Latest Telemetry**

15. You should see the 2 attributes appear in the pane as below:

The screenshot shows the 'Latest Telemetry' tab for device DEMO123. The table displays the following data:

Last update time	Key	Value
2018-03-19 09:58:13	active	false
2018-03-19 09:58:13	humidity	55.0
2018-03-19 09:58:13	temperature	21

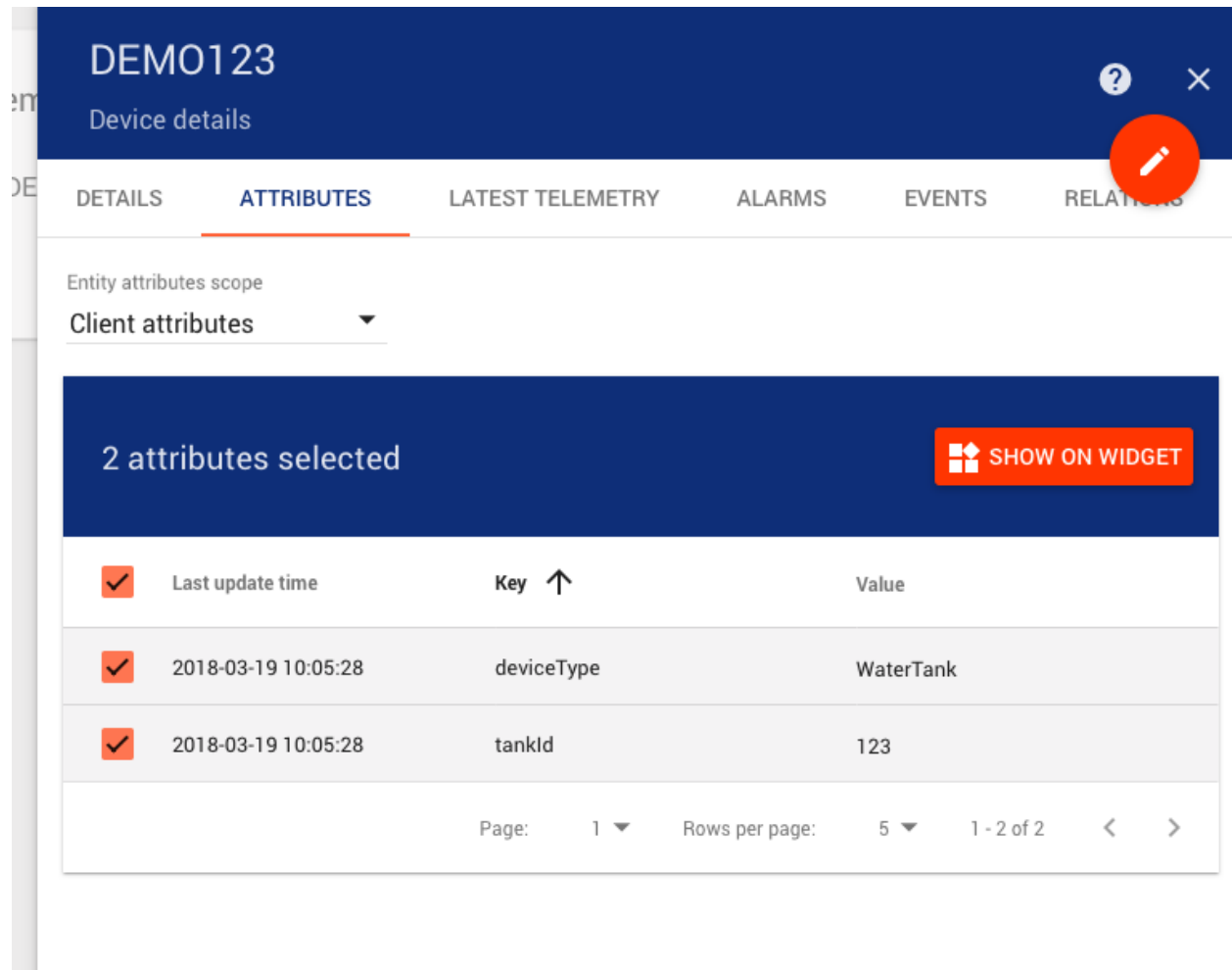
Page: 1 Rows per page: 5 1 - 3 of 3

Observe device data on the Web UI

Create A New Dashboard to Visualize the Data

Attributes

The easiest way to create new dashboard is to select device attributes and show them on widget.



The screenshot shows the 'DEMO123' device details page. The 'ATTRIBUTES' tab is selected, showing a table of attributes. A red circle highlights the 'SHOW ON WIDGET' button. The table has columns for 'Last update time', 'Key', and 'Value'. Two attributes are selected, indicated by red checkmarks in the first column.

	Last update time	Key	Value
<input checked="" type="checkbox"/>	2018-03-19 10:05:28	deviceType	WaterTank
<input checked="" type="checkbox"/>	2018-03-19 10:05:28	tankId	123

Page: 1 Rows per page: 5 1 - 2 of 2

Once you click on “Show on widget” button, you will see a “widget preview” panel where you can

- Select widget bundle
- Select preferred widget
- Add widget to new or existing dashboard

Select the Widget Bundle **Cards** and click the > until you get to the **Attributes Card** as shown below

DEMO123

Device details

?

✕

<

DETAILS

ATTRIBUTES

LATEST TELEMETRY

ALARMS

EVENTS

F

Entity attributes scope

Client attributes

Current bundle

Cards

System

ADD TO DASHBOARD

✕

Attributes card

✕

DEMO123

deviceType	WaterTank
tankId	123

<

Click **Add to Dashboard**

Select **Create New Dashboard**

Add widget to dashboard



☐ Select existing dashboard

Select dashboard

☒ Create new dashboard

New dashboard title *

Demo Dashboard

☐ Open dashboard

ADD

CANCEL

Type in the title **Demo Dashboard** and click **ADD**

Next we will add some of the telemetry data as well.

Click on the **Latest Telemetry** tab

Select the Telemetry values that you would like to monitor on the dashboard and click **Show on Widget**

DEMO123

Device details

?

×

<

DETAILS

ATTRIBUTES

LATEST TELEMETRY

ALARMS

EVENTS

F

>

3 telemetry units selected

SHOW ON WIDGET

<input checked="" type="checkbox"/>	Last update time	Key <div>↑</div>	Value
<input checked="" type="checkbox"/>	2018-03-19 13:43:37	active	false
<input checked="" type="checkbox"/>	2018-03-19 13:43:37	humidity	55.0
<input checked="" type="checkbox"/>	2018-03-19 13:43:37	temperature	21

Page:

1

▼

Rows per page:

5

▼

1 - 3 of 3

<

>

Under Cards choose **Time Series Table**

DEMO123

Device details

?

×

DETAILS

ATTRIBUTES

LATEST TELEMTRY

ALARMS

EVENTS

F

Current bundle

Cards

System

ADD TO DASHBOARD

×

Timeseries table

Timestamp

↓

active

humidity

temperature

2018-03-19 13:45:31

false

55

21

2018-03-19 13:45:30

false

55

21

2018-03-19 13:45:29

false

55

21

2018-03-19 13:45:28

false

55

21

2018-03-19 13:45:27

false

55

21

Page:

1

Rows per page:

5

1 - 5 of 61

<

>

Click **Add to Dashboard**

This time select **Select Existing Dashboard**

Click in the box that says **Select Dashboard** and choose the **Demo Dashboard** that we created above.

Add widget to dashboard



☒ Select existing dashboard

Demo Dashboard



☐ Create new dashboard

New dashboard title



Open dashboard

ADD

CANCEL

Tick the **Open Dashboard** checkbox as well.

Click **ADD**

The 2 Widgets that were created in the steps above should now be shown.

Attributes card

Demo123

deviceType	WaterTank
tankId	123

Timeseries table

Timestamp ↓	active	humidity
2018-03-19 13:48:47	false	55
2018-03-19 13:48:46	false	55
2018-03-19 13:48:45	false	55
2018-03-19 13:48:44	false	55
2018-03-19	false	55

3.2 Features

There are several Features to the Tempus Cloud Framework that are delivered out of the box.

Entities and Relations

Entities Overview

Tempus Cloud provides the user interface and REST APIs to provision and manage multiple entity types and their relations in your IoT application. Supported entities are:

- **Tenants** - you can treat tenant as a separate business-entity: individual or organization who owns or produce devices and assets; Tenant may have multiple tenant administrator users and millions of customers;
- **Customers** - customer is also a separate business-entity: individual or organization who purchase or uses tenant devices and/or assets; Customer may have multiple users and millions of devices and/or assets;
- **Users** - users are able to browse dashboards and manage entities;
- **Devices** - basic IoT entities that may produce telemetry data and handle RPC commands. For example sensors, actuators, switches;
- **Assets** - abstract IoT entities that may be related to other devices and assets. For example factory, field, vehicle;
- **Alarms** - events that identify issues with your assets, devices or other entities;
- **Dashboards** - visualization of your IoT data and ability to control particular devices through user interface;
- **Rules** - processing units for incoming messages, entity lifecycle events, etc;
- **Plugins** - extensions to the platform that process IoT data and help to integrate with other server applications

Each entity supports:

- **Attributes** - static and semi-static key-value pairs associated with entities. For example serial number, model, firmware version;
- **Telemetry data** - time-series data points available for storage, querying and visualization. For example temperature, humidity, battery level;
- **Relations** - directed connections to other entities. For example contains, manages, owns, produces.

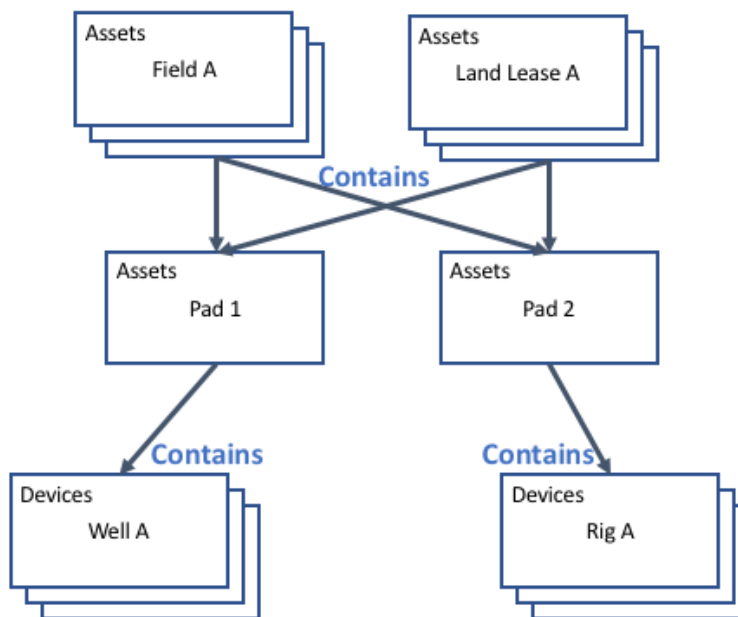
Additionally, devices and assets also have a type. This allows distinguishing them and process data from them in a different way. This guide provides the overview of the features listed above, some useful links to get more details and real-life examples of their usage.

Real-life application

The easiest way to understand the concepts of Tempus is to implement your first Tempus application. Let's assume we want to build an application that collects data from soil moisture and temperature sensors, visualize this data on the dashboard, detect issues, raise alarms and control the irrigation. Let's also assume we want to support multiple fields with hundreds of sensors. Fields may be also grouped to the Geo regions. We believe there should be following logical steps to build such an application:

Provision entities and relations

We are going to setup following hierarchy of assets and devices:



Please review the following screen cast to learn how to provision land lease, field, and pad assets and their relations using Tempus Web UI

Please review the following screen cast to learn how to provision devices and their relations with assets using Tempus Web UI

You can automate this actions using Tempus REST API. You can provision new assets by using POST requests to the following URL

```
http(s)://host:port/api/asset
```

For example:

create-asset.sh

```
curl -v -X POST -d @create-asset.json http://localhost:8080/api/asset \
--header "Content-Type:application/json" \
--header "X-Authorization: $JWT_TOKEN"
```

create-asset.json

```
{"name": "Field C", "type": "field"}
```

Note: in order to execute this request, you will need to substitute **\$JWT_TOKEN** with a valid JWT token. This token should belong to a user with **TENANT_ADMIN** role. You can use following guide to get the token.

Also, you can provision new relation using POST request to the following URL

```
http(s)://host:port/api/asset
```

create-relation.sh

```
curl -v -X POST -d @create-asset.json http://localhost:8080/api/relation \
--header "Content-Type:application/json" \
--header "X-Authorization: $JWT_TOKEN"
```

create-relation.json

```
{ "from": { "id": "$FROM_ASSET_ID", "entityType": "ASSET" }, "type": "Contains", "to": {  
  ↪ "entityType": "ASSET", "id": "$TO_ASSET_ID" } }
```

Note: Don't forget to replace \$FROM_ASSET_ID and \$TO_ASSET_ID with valid asset ids. **Note:** One can relate any entities. For example, assets to devices or assets to users. You can receive them as a result of previous REST API call or use Web UI.

Attribute Management

Attribute types

Attributes are separated into three main groups:

- **server-side** - attributes are reported and managed by the server-side application. Not visible to the device application. Some secret data that may be used by tempus cloud rules, but should not be available to the device. Any Tempus Cloud entity supports server-side attributes: Device, Asset, Customer, Tenant, Rules, etc.

image

- **client-side** - see device specific attributes
- **shared** - see device specific attributes

Device specific Attribute types

All attributes may be used in Rule Engine components: filters, processors, and actions. This guide provides the overview of the features listed above and some useful links to get more details.

Device specific attributes are separated into two main groups:

- **client-side** - attributes are reported and managed by the device application. For example current software/firmware version, hardware specification, etc.

image

- **shared** - attributes are reported and managed by the server-side application. Visible to the device application. For example customer subscription plan, target software/firmware version.

image

Device attributes API

Tempus Cloud provides following API to device applications:

- upload client-side attributes to the server
- request client-side and shared attributes from the server.
- subscribe to updates of shared attributes.

Attributes API is specific for each supported network protocol. You can review API and examples in corresponding reference page:

MQTT Device API Reference

Getting Started

MQTT basics

MQTT is a lightweight publish-subscribe messaging protocol which probably makes it the most suitable for various IoT devices. You can find more information about MQTT [here](#). Tempus Cloud server nodes act as an MQTT Broker that supports QoS levels 0 (at most once) and 1 (at least once) and a set of predefined topics.

Client libraries setup

You can find a large number of MQTT client libraries on the web. Examples in this article will be based on Mosquitto and MQTT.js. In order to setup one of those tools, you can use instructions in our [Hello World](#) guide.

MQTT Connect

We will use access token device credentials in this article and they will be referred to later as \$ACCESS_TOKEN. The application needs to send MQTT CONNECT message with username that contains \$ACCESS_TOKEN. Possible return codes and their reasons during connect sequence:

- 0x00 Connected - Successfully connected to Tempus Cloud MQTT server.
- 0x04 Connection Refused, bad user name or password - Username is empty.
- 0x05 Connection Refused, not authorized - Username contains invalid \$ACCESS_TOKEN.

Key-Value Format

By default, Tempus Cloud supports key-value content in JSON. Key is always a string, while value can be either string, boolean, double or long. Using custom binary format or some serialization framework is also possible. See protocol customization for more details. For example:

```
{ "stringKey": "value1", "booleanKey": true, "doubleKey": 42.0, "longKey": 73 }
```

Telemetry upload API

```
v1/devices/me/telemetry
```

The simplest supported data formats are:

```
{ "key1": "value1", "key2": "value2" }
```

or

```
[ { "key1": "value1" }, { "key2": "value2" } ]
```

Please note that in this case, the server-side timestamp will be assigned to uploaded data!

In case your device is able to get the client-side timestamp, you can use following format:

```
{ "ts":1451649600512, "values":{"key1":"value1", "key2":"value2"}}
```

In the example above, we assume that “1451649600512” is a unix timestamp with milliseconds precision. For example, the value ‘1451649600512’ corresponds to ‘Fri, 01 Jan 2016 12:00:00.512 GMT’

Mosquitto

```
# Publish data as an object without timestamp (server-side timestamp will be used)
mosquitto_pub -d -h "127.0.0.1" -t "v1/devices/me/telemetry" -u "$ACCESS_TOKEN" -f
↪ "telemetry-data-as-object.json"
# Publish data as an array of objects without timestamp (server-side timestamp will
↪ be used)
mosquitto_pub -d -h "127.0.0.1" -t "v1/devices/me/telemetry" -u "$ACCESS_TOKEN" -f
↪ "telemetry-data-as-array.json"
# Publish data as an object with timestamp (server-side timestamp will be used)
mosquitto_pub -d -h "127.0.0.1" -t "v1/devices/me/telemetry" -u "$ACCESS_TOKEN" -f
↪ "telemetry-data-with-ts.json"
```

MQTT.js

```
# Publish data as an object without timestamp (server-side timestamp will be used)
cat telemetry-data-as-object.json | mqtt pub -v -h "127.0.0.1" -t "v1/devices/me/
↪ telemetry" -u '$ACCESS_TOKEN' -s
# Publish data as an array of objects without timestamp (server-side timestamp will
↪ be used)
cat telemetry-data-as-array.json | mqtt pub -v -h "127.0.0.1" -t "v1/devices/me/
↪ telemetry" -u '$ACCESS_TOKEN' -s
# Publish data as an object with timestamp (server-side timestamp will be used)
cat telemetry-data-with-ts.json | mqtt pub -v -h "127.0.0.1" -t "v1/devices/me/
↪ telemetry" -u '$ACCESS_TOKEN' -s
```

telemetry-data-as-object.json

```
{ "key1": "value1", "key2": true, "key3": 3.0, "key4": 4 }
```

telemetry-data-as-array.json

```
[ { "key1": "value1" }, { "key2": true } ]
```

telemetry-data-with-ts.json

```
{ "ts":1451649600512, "values":{"key1":"value1", "key2":"value2"}}
```

Telemetry upload API

In order to support depth data, the administrator must first configure Tempus to handle depth data. This is done by changing the configuration in Tempus.yml and under the heading **UI Related configuration** set depthSeries to true:

```
#UI Related Configuration
configurations:
  ui:
    depthSeries: "true"
```

The depth topic is as follows:

```
v1/devices/me/depth/telemetry
```

The supported data format is:

```
{"ds":5844.23,"values":{"viscosity":0.1, "humidity":22.0}}
```

Notice the use of DS. DS stands for depth stamp. This can either be in meters or feet, but must be consistent throughout the publication of data to the device. Mixing of units will cause data integrity issues.

Attributes API

Tempus Cloud attributes API allows devices to

- Upload client-side device attributes to the server.
- Request client-side and shared device attributes from the server.
- Subscribe to shared device attributes from the server.

Publish attribute update to the server

In order to publish client-side device attributes to Tempus Cloud server node, send PUBLISH message to the following topic:

```
v1/devices/me/attributes
```

Mosquitto

```
# Publish client-side attributes update
mosquitto_pub -d -h "127.0.0.1" -t "v1/devices/me/attributes" -u "$ACCESS_TOKEN" -f
↪"new-attributes-values.json"
```

MQTT.js

```
# Publish client-side attributes update
cat new-attributes-values.json | mqtt pub -d -h "127.0.0.1" -t "v1/devices/me/
↪attributes" -u '$ACCESS_TOKEN' -s
```

new-attributes-values.json

```
{"attribute1":"value1", "attribute2":true, "attribute3":42.0, "attribute4":73}
```

Request attribute values from the server

In order to request client-side or shared device attributes to Tempus Cloud server node, send PUBLISH message to the following topic:

```
v1/devices/me/attributes/request/$request_id
```

where **\$request_id** is your integer request identifier. Before sending PUBLISH message with the request, client need to subscribe to:

```
v1/devices/me/attributes/response/+
```

The following example is written in javascript and is based on mqtt.js. Pure command-line examples are not available because subscribe and publish need to happen in the same mqtt session.

MQTT.js

```
export TOKEN=$ACCESS_TOKEN
node mqtt-js-attributes-request.js
```

mqtt-js-attributes-request.js

```
var mqtt = require('mqtt')
var client = mqtt.connect('mqtt://127.0.0.1',{
  username: process.env.TOKEN
})

client.on('connect', function () {
  console.log('connected')
  client.subscribe('v1/devices/me/attributes/response+')
  client.publish('v1/devices/me/attributes/request/1', '{"clientKeys":"attribute1,
↪attribute2", "sharedKeys":"shared1,shared2"}')
})

client.on('message', function (topic, message) {
  console.log('response.topic: ' + topic)
  console.log('response.body: ' + message.toString())
  client.end()
})
```

Result

```
{"key1": "value1"}
```

Note: The intersection of client-side and shared device attribute keys is a bad practice! However, it is still possible to have same keys for client, shared or even server-side attributes.

Subscribe to Attribute Updates from the Server

In order to subscribe to shared device attribute changes, send SUBSCRIBE message to the following topic:

```
v1/devices/me/attributes
```

Once shared attribute will be changed by one of the server-side components (REST API or custom plugins) the client will receive the following update:

```
{"key1": "value1"}
```

Mosquitto

```
# Subscribes to attribute updates
mosquitto_sub -d -h "127.0.0.1" -t "v1/devices/me/attributes" -u "$ACCESS_TOKEN"
```

MQTT.js

```
# Subscribes to attribute updates
mqtt sub -v "127.0.0.1" -t "v1/devices/me/attributes" -u '$ACCESS_TOKEN'
```

RPC API

Server-side RPC

In order to subscribe to RPC commands from the server, send SUBSCRIBE message to the following topic:

```
v1/devices/me/rpc/request/+
```

Once subscribed, the client will receive individual commands as a PUBLISH message to the corresponding topic:

```
v1/devices/me/rpc/request/$request_id
```

where \$request_id is an integer request identifier. The client should publish the response to the following topic:

```
v1/devices/me/rpc/response/$request_id
```

The following example is written in javascript and is based on mqtt.js. Pure command-line examples are not available because subscribe and publish need to happen in the same mqtt session.

MQTT.js

```
export TOKEN=$ACCESS_TOKEN  
node mqtt-js-rpc-from-server.js
```

mqtt-js-rpc-from-server.js

```
var mqtt = require('mqtt');  
var client = mqtt.connect('mqtt://127.0.0.1',{  
  username: process.env.TOKEN  
});  
  
client.on('connect', function () {  
  console.log('connected');  
  client.subscribe('v1/devices/me/rpc/request/+')  
});  
  
client.on('message', function (topic, message) {  
  console.log('request.topic: ' + topic);  
  console.log('request.body: ' + message.toString());  
  var requestId = topic.slice('v1/devices/me/rpc/request/'.length);  
  //client acts as an echo service  
  client.publish('v1/devices/me/rpc/response/' + requestId, message);  
});
```

Client-side RPC

In order to send RPC commands to server, send PUBLISH message to the following topic:

```
v1/devices/me/rpc/request/$request_id
```

where \$request_id is an integer request identifier. The response from server will be published to the following topic:

```
v1/devices/me/rpc/response/$request_id
```

The following example is written in javascript and is based on mqtt.js. Pure command-line examples are not available because subscribe and publish need to happen in the same mqtt session.

MQTT.js

```
export TOKEN=$ACCESS_TOKEN
node mqtt-js-rpc-from-client.js
```

mqtt-js-rpc-from-client.js

```
var mqtt = require('mqtt');
var client = mqtt.connect('mqtt://127.0.0.1', {
  username: process.env.TOKEN
});

client.on('connect', function () {
  console.log('connected');
  client.subscribe('v1/devices/me/rpc/response/+');
  var requestId = 1;
  var request = {
    "method": "getTime",
    "params": {}
  };
  client.publish('v1/devices/me/rpc/request/' + requestId, JSON.stringify(request));
});

client.on('message', function (topic, message) {
  console.log('response.topic: ' + topic);
  console.log('response.body: ' + message.toString());
});
```

Protocol Customization

MQTT transport can be fully customized for specific use-case by changing the corresponding module.

CoAP Device API Reference

Getting started

CoAP basics

CoAP is a light-weight IoT protocol for constrained devices. You can find more information about CoAP [here](#). CoAP protocol is UDP based, but similar to HTTP it uses request-response model. CoAP observe option allows to subscribe to resources and receive notifications on resource change.

Tempus Cloud nodes act as CoAP Servers that support both regular and observe requests.

Client libraries setup

You can find CoAP client libraries for different programming languages on the web. Examples in this article will be based on CoAP cli. In order to setup this tool, you can use instructions in our [Hello World guide](#).

CoAP Authentication and error codes

We will use access token device credentials in this article and they will be referred to later as \$ACCESS_TOKEN. The application needs to include \$ACCESS_TOKEN as a path parameter into each CoAP request. Possible error codes and their reasons:

- 4.00 Bad Request - Invalid URL, request parameters or body.
- 4.01 Unauthorized - Invalid \$ACCESS_TOKEN.
- 4.04 Not Found - Resource not found.

Key-value format

By default, Tempus Cloud supports key-value content in JSON. Key is always a string, while value can be either string, boolean, double or long. Using custom binary format or some serialization framework is also possible. See protocol customization for more details. For example:

```
{ "stringKey": "value1", "booleanKey": true, "doubleKey": 42.0, "longKey": 73 }
```

Telemetry upload API

In order to publish telemetry data to Tempus Cloud server node, send POST request to the following URL:

```
coap://host:port/api/v1/$ACCESS_TOKEN/telemetry
```

The simplest supported data formats are:

```
{ "key1": "value1", "key2": "value2" }
```

or

```
[ { "key1": "value1" }, { "key2": "value2" } ]
```

Please note that in this case, the server-side timestamp will be assigned to uploaded data!

In case your device is able to get the client-side timestamp, you can use following format:

```
{ "ts": 1451649600512, "values": { "key1": "value1", "key2": "value2" } }
```

In the example above, we assume that “1451649600512” is a unix timestamp with milliseconds precision. For example, the value ‘1451649600512’ corresponds to ‘Fri, 01 Jan 2016 12:00:00.512 GMT’

Example

coap-telemetry.sh

```
# Publish data as an object without timestamp (server-side timestamp will be used)
cat telemetry-data-as-object.json | coap post coap://localhost/api/v1/$ACCESS_TOKEN/
↪telemetry
# Publish data as an array of objects without timestamp (server-side timestamp will
↪be used)
cat telemetry-data-as-array.json | coap post coap://localhost/api/v1/$ACCESS_TOKEN/
↪telemetry
# Publish data as an object with timestamp (server-side timestamp will be used)
cat telemetry-data-with-ts.json | coap post coap://localhost/api/v1/$ACCESS_TOKEN/
↪telemetry
```

(continues on next page)

(continued from previous page)

telemetry-data-as-object.json

```
{ "key1": "value1", "key2": true, "key3": 3.0, "key4": 4 }
```

telemetry-data-as-array.json

```
[ { "key1": "value1" }, { "key2": true } ]
```

telemetry-data-with-ts.json

```
{ "ts": 1451649600512, "values": { "key1": "value1", "key2": "value2" } }
```

Attributes API

Tempus Cloud attributes API allows devices to

- Upload client-side device attributes to the server.
- Request client-side and shared device attributes from the server.
- Subscribe to shared device attributes from the server.

Publish attribute update to the server

In order to publish client-side device attributes to Tempus Cloud server node, send POST request to the following URL:

```
coap://host:port/api/v1/$ACCESS_TOKEN/attributes
```

coap-telemetry.sh

```
# Publish client-side attributes update
cat new-attributes-values.json | coap post coap://localhost/api/v1/$ACCESS_TOKEN/
↪ attributes
```

new-attributes-values.json

```
{ "attribute1": "value1", "attribute2": true, "attribute3": 42.0, "attribute4": 73 }
```

Request attribute values from the server

In order to request client-side or shared device attributes to Tempus Cloud server node, send GET request to the following URL:

```
coap://host:port/api/v1/$ACCESS_TOKEN/attributes?clientKeys=attribute1,attribute2&
↪ sharedKeys=shared1,shared2
```

Example

```
# Send CoAP attributes request
coap get coap://localhost/api/v1/$ACCESS_TOKEN/attributes?clientKeys=attribute1,
↪ attribute2&sharedKeys=shared1,shared2
```

(continues on next page)

Result

```
{ "key1": "value1" }
```

Please note: the intersection of client-side and shared device attribute keys is a bad practice! However, it is still possible to have same keys for client, shared or even server-side attributes.

Subscribe to attribute updates from the server

In order to subscribe to shared device attribute changes, send GET request with Observe option to the following URL:

```
coap://host:port/api/v1/$ACCESS_TOKEN/attributes
```

Once shared attribute will be changed by one of the server-side components (REST API or custom plugins) the client will receive the following update:

Example

```
# Subscribe to attribute updates
coap get -o coap://localhost/api/v1/$ACCESS_TOKEN/attributes
```

Result

```
{ "key1": "value1" }
```

RPC API

Server-side RPC

In order to subscribe to RPC commands from the server, send GET request with observe flag to the following URL:

```
coap://host:port/api/v1/$ACCESS_TOKEN/rpc
```

Once subscribed, a client may receive rpc requests. An example of RPC request body is shown below:

```
{
  "id": "1",
  "method": "setGpio",
  "params": {
    "pin": "23",
    "value": 1
  }
}
```

where

- **id** - request id, integer request identifier
- **method** - RPC method name, string
- **params** - RPC method params, custom json object

and can reply to them using POST request to the following URL:

```
coap://host:port/api/v1/$ACCESS_TOKEN/rpc/{$id}
```

where **\$id** is an integer request identifier.

Example Subscribe

```
# Subscribe to RPC requests
coap get -o coap://localhost/api/v1/$ACCESS_TOKEN/rpc
```

Example Reply

```
# Publish response to RPC request
cat rpc-response.json | coap post coap://localhost/api/v1/$ACCESS_TOKEN/rpc/1
```

Reply Body

```
{ "result": "ok" }
```

Client-side RPC

In order to send RPC commands to the server, send POST request to the following URL:

```
coap://host:port/api/v1/$ACCESS_TOKEN/rpc
```

Both request and response body should be valid JSON documents. The content of the documents is specific to the plugin that will handle your request.

Example Request

```
# Post client-side rpc request
cat rpc-client-request.json | coap post coap://localhost/api/v1/$ACCESS_TOKEN/rpc
```

Request Body

```
{ "method": "getTime", "params": {} }
```

Response Body

```
{ "time": "2016 11 21 12:54:44.287" }
```

Protocol customization

CoAP transport can be fully customized for specific use-case by changing the corresponding module.

HTTP Device API Reference

Getting Started

HTTP basics

HTTP is a general-purpose network protocol that can be used in IoT applications. You can find more information about HTTP [here](#). HTTP protocol is TCP based and uses request-response model.

Tempus Cloud server nodes act as an HTTP Server that supports both HTTP and HTTPS protocols.

Client libraries setup

You can find HTTP client libraries for different programming languages on the web. Examples in this article will be based on curl. In order to setup this tool, you can use instructions in our Hello World guide.

HTTP Authentication and error codes

We will use access token device credentials in this article and they will be referred to later as \$ACCESS_TOKEN. The application needs to include \$ACCESS_TOKEN as a path parameter in each HTTP request. Possible error codes and their reasons:

- 400 Bad Request - Invalid URL, request parameters or body.
- 401 Unauthorized - Invalid \$ACCESS_TOKEN.
- 404 Not Found - Resource not found.

Key-value format

By default, Tempus Cloud supports key-value content in JSON. Key is always a string, while value can be either string, boolean, double or long. Using custom binary format or some serialization framework is also possible. See protocol customization for more details. For example:

```
{ "stringKey": "value1", "booleanKey": true, "doubleKey": 42.0, "longKey": 73 }
```

Telemetry upload API

In order to publish telemetry data to Tempus Cloud server node, send POST request to the following URL:

```
http(s)://host:port/api/v1/$ACCESS_TOKEN/telemetry
```

The simplest supported data formats are:

```
{ "key1": "value1", "key2": "value2" }
```

or

```
[ { "key1": "value1" }, { "key2": "value2" } ]
```

Please note that in this case, the server-side timestamp will be assigned to uploaded data!

In case your device is able to get the client-side timestamp, you can use following format:

```
{ "ts": 1451649600512, "values": { "key1": "value1", "key2": "value2" } }
```

In the example above, we assume that “1451649600512” is a unix timestamp with milliseconds precision. For example, the value ‘1451649600512’ corresponds to ‘Fri, 01 Jan 2016 12:00:00.512 GMT’

http-telemetry.sh

```
# Publish data as an object without timestamp (server-side timestamp will be used)
curl -v -X POST -d @telemetry-data-as-object.json http://localhost:8080/api/v1/
↪$ACCESS_TOKEN/telemetry --header "Content-Type:application/json"
# Publish data as an array of objects without timestamp (server-side timestamp will
↪be used)
curl -v -X POST -d @telemetry-data-as-array.json http://localhost:8080/api/v1/$ACCESS_
↪TOKEN/telemetry --header "Content-Type:application/json"
# Publish data as an object with timestamp (server-side timestamp will be used)
curl -v -X POST -d @telemetry-data-with-ts.json http://localhost:8080/api/v1/$ACCESS_
↪TOKEN/telemetry --header "Content-Type:application/json"
```

telemetry-data-as-object.json

```
{ "key1": "value1", "key2": true, "key3": 3.0, "key4": 4 }
```

telemetry-data-as-array.json

```
[ { "key1": "value1" }, { "key2": true } ]
```

telemetry-data-with-ts.json

```
{ "ts": 1451649600512, "values": { "key1": "value1", "key2": "value2" } }
```

Attributes API

Tempus Cloud attributes API allows devices to

- Upload client-side device attributes to the server.
- Request client-side and shared device attributes from the server.
- Subscribe to shared device attributes from the server.

Publish attribute update to the server

In order to publish client-side device attributes to Tempus Cloud server node, send POST request to the following URL:

```
http(s)://host:port/api/v1/$ACCESS_TOKEN/attributes
```

Example

```
# Publish client-side attributes update
curl -v -X POST -d @new-attributes-values.json http://localhost:8080/api/v1/$ACCESS_
↪TOKEN/attributes --header "Content-Type:application/json"
```

new-attributes-values.json

```
{ "attribute1": "value1", "attribute2": true, "attribute3": 42.0, "attribute4": 73 }
```

Request attribute values from the server

In order to request client-side or shared device attributes to Tempus Cloud server node, send GET request to the following URL:

```
http(s)://host:port/api/v1/$ACCESS_TOKEN/attributes?clientKeys=attribute1,attribute2&
↳sharedKeys=shared1,shared2
```

Example

```
# Send HTTP attributes request
curl -v -X GET http://localhost:8080/api/v1/$ACCESS_TOKEN/attributes?
↳clientKeys=attribute1,attribute2&sharedKeys=shared1,shared2
```

Result

```
{"key1": "value1"}
```

Please note: the intersection of client-side and shared device attribute keys is a bad practice! However, it is still possible to have same keys for client, shared or even server-side attributes.

Subscribe to attribute updates from the server

In order to subscribe to shared device attribute changes, send GET request with optional “timeout” request parameter to the following URL:

```
http(s)://host:port/api/v1/$ACCESS_TOKEN/attributes/updates
```

Once shared attribute will be changed by one of the server-side components (REST API or custom plugins) the client will receive the following update:

Example

```
# Send subscribe attributes request with 20 seconds timeout
curl -v -X GET http://localhost:8080/api/v1/$ACCESS_TOKEN/attributes/updates?
↳timeout=20000
```

Result

```
{"key1": "value1"}
```

RPC API

Server-side RPC

In order to subscribe to RPC commands from the server, send GET request with optional “timeout” request parameter to the following URL:

```
http(s)://host:port/api/v1/$ACCESS_TOKEN/rpc
```

Once subscribed, a client may receive rpc request or a timeout message if there are no requests to a particular device. An example of RPC request body is shown below:

```
{
  "id": "1",
  "method": "setGpio",
  "params": {
    "pin": "23",
```

(continues on next page)

(continued from previous page)

```
    "value": 1
  }
}
```

where

- **id** - request id, integer request identifier
- **method** - RPC method name, string
- **params** - RPC method params, custom json object

and can reply to them using POST request to the following URL:

```
http://host:port/api/v1/$ACCESS_TOKEN/rpc/{$id}
```

where **\$id** is an integer request identifier.

Example Subscribe

```
# Send rpc request with 20 seconds timeout
curl -v -X GET http://localhost:8080/api/v1/$ACCESS_TOKEN/rpc?timeout=20000
```

Example Reply

```
# Publish response to RPC request
curl -v -X POST -d @rpc-response.json http://localhost:8080/api/v1/$ACCESS_TOKEN/rpc/
↪1 --header "Content-Type:application/json"
```

Reply Body

```
{ "result": "ok" }
```

Client-side RPC

In order to send RPC commands to the server, send POST request to the following URL:

```
http://host:port/api/v1/$ACCESS_TOKEN/rpc
```

Both request and response body should be valid JSON documents. The content of the documents is specific to the plugin that will handle your request.

Example Request

```
# Post client-side rpc request
curl -X POST -d @rpc-client-request.json http://localhost:8080/api/v1/$ACCESS_TOKEN/
↪rpc --header "Content-Type:application/json"
```

Request Body

```
{ "method": "getTime", "params": {} }
```

Response Body

```
{ "time": "2016 11 21 12:54:44.287" }
```

Protocol customization

HTTP transport can be fully customized for specific use-case by changing the corresponding module.

Device Telemetry Data Download

Time Series Data

Device Time Series data can be downloaded in CSV format between two timestamp values. The API can be invoked as below except that values in brackets needs to be replaced with actual values(device id, start timestamp, end timestamp).Authorization header values also need to be provided in headers.

```
http://host:port/api/download/deviceSeriesData?deviceId=<Device_Id>&type=ts&startValue  
↪<Start_Timestamp_Long_Value>&endValue<End_Timestamp_Long_Value>
```

Depth Series Data

Device Depth Series data can be downloaded in CSV format between two depth values. The API can be invoked as below except that values in brackets needs to be replaced with actual values(device id, start depth, end depth).Authorization header values also need to be provided in headers.

```
http://host:port/api/download/deviceSeriesData?deviceId=<Device_Id>&type=ds&startValue  
↪<Start_Depth_Double_Value>&endValue<End_Depth_Double_Value>
```

Attributes Data

Device Attributes data can be downloaded in CSV format. The API can be invoked as below except that value in bracket needs to be replaced with actual value(device id).Authorization header values also need to be provided in headers.

```
http://host:port/api/download/deviceAttributesData?deviceId=<Device_Id>
```

Telemetry plugin

Tempus Cloud consists of core services and pluggable modules called plugins. Telemetry plugin is responsible for persisting attributes data to internal data storage; provides server-side API to query and subscribe for attribute updates. Since Telemetry plugin functionality is critical for data visualization purposes in dashboards, it is configured on the system level by a system administrator. Advanced users or platform developers can customize telemetry plugin functionality.

Internal data storage

Tempus Cloud uses either Cassandra NoSQL database or SQL database to store all data.

Although you can query the database directly, Tempus Cloud provides a set of RESTful and Websocket API that simplify this process and apply certain security policies:

Tenant Administrator user is able to manage attributes for all entities that belong to the corresponding tenant. Customer user is able to manage attributes only for entities that are assigned to the corresponding customer.

Data Query API

Telemetry plugin provides following API to fetch device attributes:

Attribute keys API

You can fetch list of all attribute keys for particular entity type and entity id using GET request to the following URL

```
http(s)://host:port/api/plugins/telemetry/{entityType}/{entityId}/keys/attributes
```

get-attributes-keys.sh

```
curl -v -X GET http://localhost:8080/api/plugins/telemetry/DEVICE/ac8e6020-ae99-11e6-  
↪b9bd-2b15845ada4e/keys/attributes \  
--header "Content-Type:application/json" \  
--header "X-Authorization: $JWT_TOKEN"
```

get-attributes-keys-result.json

```
[{"model"},"softwareVersion"]
```

Supported entity types are: TENANT, CUSTOMER, USER, RULE, PLUGIN, DASHBOARD, ASSET, DEVICE, ALARM

Attribute values API

You can fetch list of latest values for particular entity type and entity id using GET request to the following URL

```
http(s)://host:port/api/plugins/telemetry/{entityType}/{entityId}/values/attributes?  
↪keys=key1,key2,key3
```

get-attributes-values.sh

```
curl -v -X GET http://localhost:8080/api/plugins/telemetry/DEVICE/ac8e6020-ae99-11e6-  
↪b9bd-2b15845ada4e/values/attributes?keys=model,softwareVersion \  
--header "Content-Type:application/json" \  
--header "X-Authorization: $JWT_TOKEN"
```

get-attributes-values-result.json

```
[  
  {  
    "lastUpdateTs": 1479735871836,  
    "key": "model",  
    "value": "Model 42"  
  },  
  {  
    "lastUpdateTs": 1479735871836,  
    "key": "softwareVersion",  
    "value": "1.0.0"  
  }  
]
```

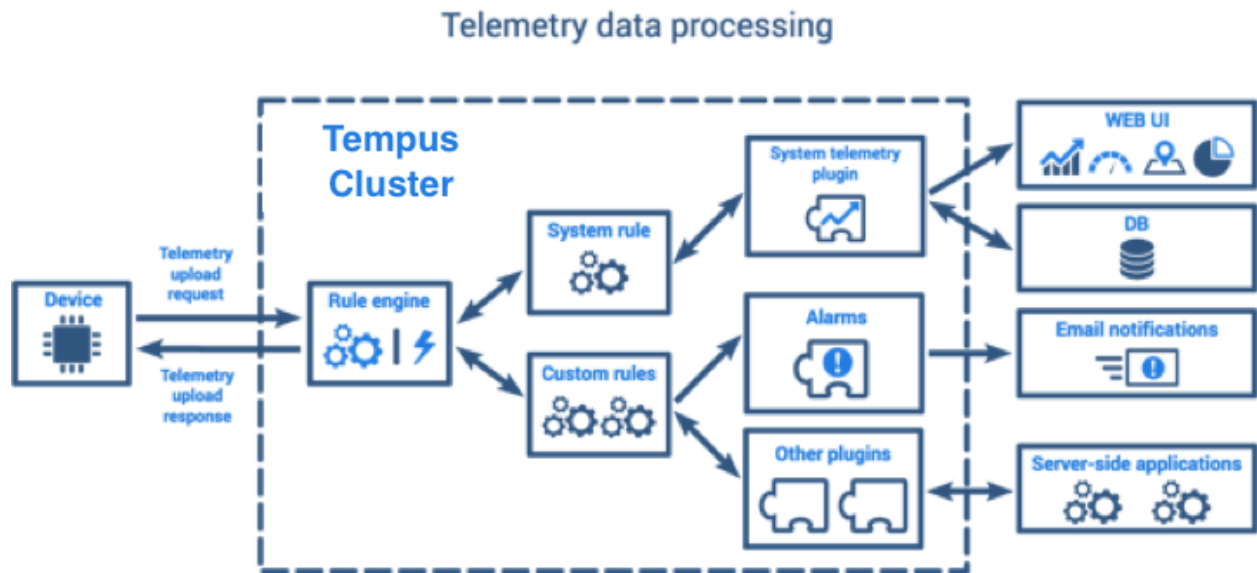
Supported entity types are: TENANT, CUSTOMER, USER, RULE, PLUGIN, DASHBOARD, ASSET, DEVICE, ALARM

Working with Telemetry Data

Tempus Cloud provides a rich set of features related to telemetry data:

- **collect** data from devices using MQTT, CoAP or HTTP protocols.
- **store** timeseries data in Cassandra (efficient, scalable and fault-tolerant NoSQL database).
- **query** latest timeseries data values or all data within the specified time interval.
- **subscribe** to data updates using websockets (for visualization or real-time analytics).
- **visualize** timeseries data using configurable and highly customizable widgets and dashboards.
- **filter** and analyze data using flexible Rule Engine (/docs/user-guide/rule-engine/).
- **generate** alarms based on collected data.
- **forward** data to external systems using plugins (e.g. Kafka or RabbitMQ plugins).

This guide provides an overview of the features listed above and some useful links to get more details.



Device telemetry upload API

Tempus Cloud provides an API to upload timeseries key-value data. Flexibility and simplicity of key-value format allow easy and seamless integration with almost any IoT device on the market. Telemetry upload API is specific for each supported network protocol. You can review API and examples in corresponding reference page:

Telemetry Plugin

Tempus Cloud consists of core services and pluggable modules called plugins. Telemetry plugin is responsible for persisting timeseries data to internal data storage; provides server-side API to query and subscribe for data updates. Since Telemetry plugin functionality is critical for data visualization purposes in dashboards, it is configured on the system level by a system administrator. Advanced users or platform developers can customize telemetry plugin functionality.

Internal data storage

Tempus Cloud uses either Cassandra NoSQL database or SQL database to store all data. A device that is sending data to the server will receive confirmation about data delivery as soon as data is stored in DB. Modern MQTT clients allow temporary local storage of undelivered data. Thus, even if one of the Tempus Cloud nodes goes down, the device will not lose the data and will be able to push it to other servers. Server side applications are also able to publish telemetry valued for different entities and entity types. Although you can query the database directly, Tempus Cloud provides set of RESTful and Websocket API that simplify this process and apply certain security policies:

- Tenant Administrator user is able to fetch data for all entities that belong to the corresponding tenant.
- Customer user is able to fetch data only for entities that are assigned to the corresponding customer.

Data Query API

Telemetry plugin provides following API to fetch entity data:

Timeseries data keys API

You can fetch list of all data keys for particular entity type and entity id using GET request to the following URL

```
http(s)://host:port/api/plugins/telemetry/{entityType}/{entityId}/keys/timeseries
```

get-telemetry-keys.sh

```
curl -v -X GET http://localhost:8080/api/plugins/telemetry/DEVICE/ac8e6020-ae99-11e6-  
↪b9bd-2b15845ada4e/keys/timeseries \  
--header "Content-Type:application/json" \  
--header "X-Authorization: $JWT_TOKEN"
```

get-telemetry-keys-result.json

```
[ "gas", "temperature" ]
```

Supported entity types are: TENANT, CUSTOMER, USER, RULE, PLUGIN, DASHBOARD, ASSET, DEVICE, ALARM

Timeseries data values API

You can fetch list of latest values for particular entity type and entity id using GET request to the following URL

```
http(s)://host:port/api/plugins/telemetry/{entityType}/{entityId}/values/timeseries?  
↪keys=key1,key2,key3
```

get-telemetry-keys.sh

```
curl -v -X GET http://localhost:8080/api/plugins/telemetry/DEVICE/ac8e6020-ae99-11e6-  
↪b9bd-2b15845ada4e/values/timeseries?keys=gas,temperature \  
--header "Content-Type:application/json" \  
--header "X-Authorization: $JWT_TOKEN"
```

get-telemetry-keys-result.json

```
{
  "gas": [
    {
      "ts": 1479735870786,
      "value": "1"
    }
  ],
  "temperature": [
    {
      "ts": 1479735870786,
      "value": "3"
    }
  ]
}
```

Supported entity types are: TENANT, CUSTOMER, USER, RULE, PLUGIN, DASHBOARD, ASSET, DEVICE, ALARM

You can also fetch list of historical values for particular entity type and entity id using GET request to the following URL

```
http(s)://host:port/api/plugins/telemetry/{entityType}/{entityId}/values/timeseries?
↳keys=key1,key2,key3&startTs=1479735870785&endTs=1479735871858&interval=60000&
↳limit=100&agg=AVG
```

The supported parameters are described below:

- **keys** - comma separated list of telemetry keys to fetch.
- **startTs** - unix timestamp that identifies start of the interval in milliseconds.
- **endTs** - unix timestamp that identifies end of the interval in milliseconds.
- **interval** - the aggregation interval, in milliseconds.
- **agg** - the aggregation function. One of MIN, MAX, AVG, SUM, COUNT, NONE.
- **limit** - the max amount of data points to return or intervals to process.

Tempus Cloud will use startTs, endTs and interval to identify aggregation partitions or sub-queries and execute asynchronous queries to DB that leverage built-in aggregation functions.

get-telemetry-values.sh

```
curl -v -X GET "http://localhost:8080/api/plugins/telemetry/DEVICE/ac8e6020-ae99-11e6-
↳b9bd-2b15845ada4e/values/timeseries?keys=gas,temperature&startTs=1479735870785&
↳endTs=1479735871858&interval=60000&limit=100&agg=AVG" \
--header "Content-Type:application/json" \
--header "X-Authorization: $JWT_TOKEN"
```

get-telemetry-values-result.json

```
{
  "gas": [
    {
      "ts": 1479735870786,
      "value": "1"
    },
    {
      "ts": 1479735871857,
```

(continues on next page)

(continued from previous page)

```
        "value": "2"
      }
    ],
    "temperature": [
      {
        "ts": 1479735870786,
        "value": "3"
      },
      {
        "ts": 1479735871857,
        "value": "4"
      }
    ]
  ]
}
```

Supported entity types are: TENANT, CUSTOMER, USER, RULE, PLUGIN, DASHBOARD, ASSET, DEVICE, ALARM

Websocket API

Websockets are actively used by Thingsobard Web UI. Websocket API duplicates REST API functionality and provides the ability to subscribe to device data changes. You can open a websocket connection to a telemetry plugin using the following URL

```
ws(s)://host:port/api/ws/plugins/telemetry?token=$JWT_TOKEN
```

Once opened, you can send subscription commands and receive subscription updates:

- **cmdId** - unique command id (within corresponding websocket connection)
- **entityType** - unique entity type. Supported entity types are: TENANT, CUSTOMER, USER, RULE, PLUGIN, DASHBOARD, ASSET, DEVICE, ALARM
- **entityId** - unique entity identifier
- **keys** - comma separated list of data keys
- **timeWindow** - fetch interval for timeseries subscriptions, in milliseconds. Data will be fetch within following interval [now()-timeWindow, now()]
- **startTs** - start time of fetch interval for historical data query, in milliseconds.
- **endTs** - end time of fetch interval for historical data query, in milliseconds.

Example

Change values of the following variables :

- token - to the JWT token which you can get using the following link.
- entityId - to your device id.

In case of live-demo server :

- replace host:port with demo-Tempus Cloud.io and choose secure connection - wss://

In case of local installation :

- replace host:port with 127.0.0.1:8080 and choose ws://

```

<!DOCTYPE HTML>
<html>
<head>

  <script type="text/javascript">
    function WebSocketAPIExample() {
      var token = "YOUR_JWT_TOKEN";
      var entityId = "YOUR_DEVICE_ID";
      var websocket = new WebSocket("ws(s)://host:port/api/ws/plugins/telemetry?
↪token=" + token);

      if (entityId === "YOUR_DEVICE_ID") {
        alert("Invalid device id!");
        websocket.close();
      }

      if (token === "YOUR_JWT_TOKEN") {
        alert("Invalid JWT token!");
        websocket.close();
      }

      websocket.onopen = function () {
        var object = {
          tsSubCmds: [
            {
              entityType: "DEVICE",
              entityId: entityId,
              scope: "LATEST_TELEMETRY",
              cmdId: 10
            }
          ],
          historyCmds: [],
          attrSubCmds: []
        };
        var data = JSON.stringify(object);
        websocket.send(data);
        alert("Message is sent: " + data);
      };

      websocket.onmessage = function (event) {
        var received_msg = event.data;
        alert("Message is received: " + received_msg);
      };

      websocket.onclose = function (event) {
        alert("Connection is closed!");
      };
    }
  </script>

</head>
<body>

<div id="sse">
  <a href="javascript:WebSocketAPIExample()">Run WebSocket</a>
</div>

```

(continues on next page)


```
</body>
</html>
```

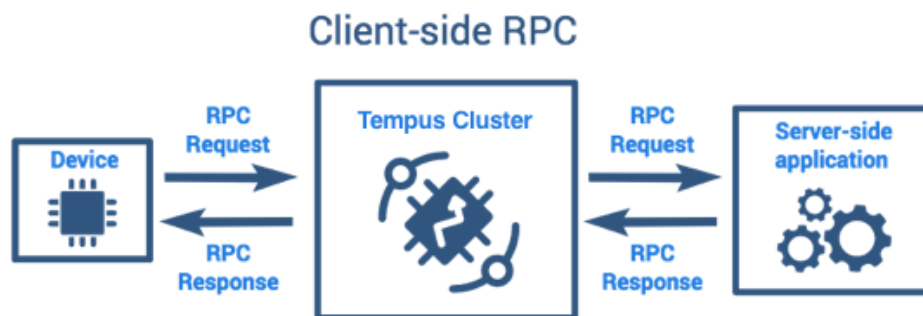
Using RPC Capabilities

Tempus Cloud allows you to send remote procedure calls (RPC) from server side applications to devices and vice versa. Basically, this feature allows you to send commands to devices and receive results of commands execution. Similar, you can execute request from the device, apply some calculations or other server-side logic on the back-end and push the response back to the device. This guide covers Tempus Cloud RPC capabilities. After reading this guide, you will get familiar with following topics:

- RPC call types
- Basic RPC use-cases
- RPC client-side and server-side APIs
- RPC widgets

RPC Call Types

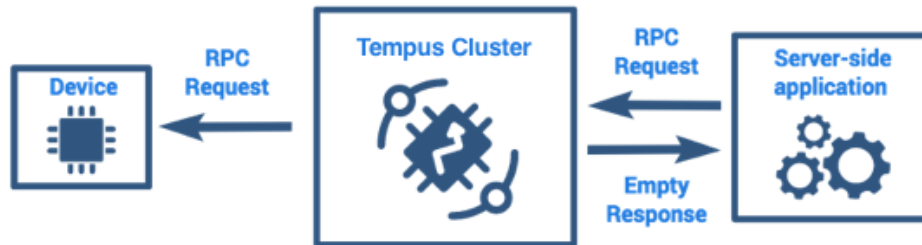
Thinsboard RPC feature can be divided into two types based on originator: device-originated and server-originated RPC calls. In order to use more familiar names, we will name device-originated RPC calls as a client-side RPC calls and server-originated RPC calls as server-side RPC calls.



Server-side RPC calls can be divided into one-way and two-way:

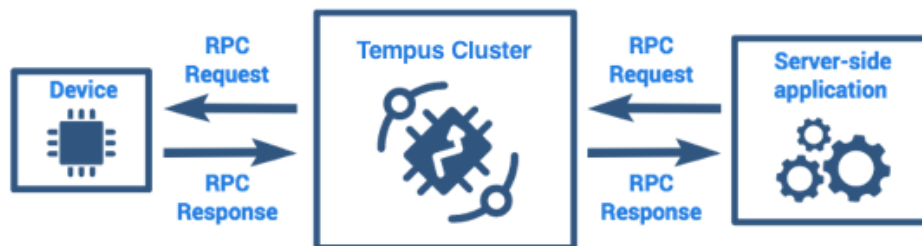
- One-way RPC request is sent to the device without delivery confirmation and obviously, does not provide any response from the device. RPC call may fail only if there is no active connection with the target device within a configurable timeout period.

One-way server-side RPC



- Two-way RPC request is sent to the device and expects to receive a response from the device within the certain timeout. The Server-side request is blocked until the target device replies to the request.

Two-way server-side RPC



Device RPC API

Tempus Cloud provides convenient API to send and receive RPC commands from applications running on the device. This API is specific for each supported network protocol. You can review API and examples in corresponding reference

page:

Server-side RPC API

Tempus Cloud provides System RPC Plugin that allows you to send RPC calls from server-side applications to the device. In order to send RPC request you need execute HTTP POST request to the following URL:

```
http(s)://host:port/api/plugins/rpc/{callType}/{deviceId}
```

where

- **callType** is either **oneway** or **twoway**
- **deviceId** is your target device id

The request body should be a valid json object with two elements:

- **method** - method name, json string
- **params** - method parameters, json object

For example:

set-gpio-request.sh

```
curl -v -X POST -d @set-gpio-request.json http://localhost:8080/api/plugins/rpc/  
↪twoway/${DEVICE_ID} \  
--header "Content-Type:application/json" \  
--header "X-Authorization: $JWT_TOKEN"
```

set-gpio-request.json

```
{  
  "method": "setGpio",  
  "params": {  
    "pin": "23",  
    "value": 1  
  }  
}
```

Please note that in order to execute this request, you will need to substitute **\$JWT_TOKEN** with a valid JWT token. This token should belong to either

- user with **TENANT_ADMIN** role
- user with **CUSTOMER_USER** role that owns the device identified by **\$DEVICE_ID**

You can use following topic to get the token:

Swagger

Tempus Cloud REST API can be explored using Swagger UI.

Once you will install the Tempus Cloud server you can open the Swagger Tempus REST API UI using the following URL:

```
http://YOUR_HOST:PORT/swagger-ui.html
```

Generate JWT Token

In order to get a JWT token, you need to execute the following request:

In case of local installation:

- replace \$TEMPUS_URL with 127.0.0.1:8080

In case of remote installation:

- replace \$TEMPUS_URL with <host>:<port> (replacing host and port with the respective host and port of the machine running Tempus).

get-token.sh

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/
↪json' -d '{"username":"demo@hashmapinc.com", "password":"tenant"}' 'http://TEMPUS_
↪URL/api/auth/login'
```

response.json

```
{"token":"$YOUR_JWT_TOKEN", "refreshToken":"$YOUR_JWT_REFRESH_TOKEN"}
```

Set Api Key Authorization Value

Now, in the Tempus REST API URL click ‘Authorize’ button located in the page header, then in the ‘Api key authorization’ ‘value’ text field enter “Bearer \$YOUR_JWT_TOKEN”

Rules Engine

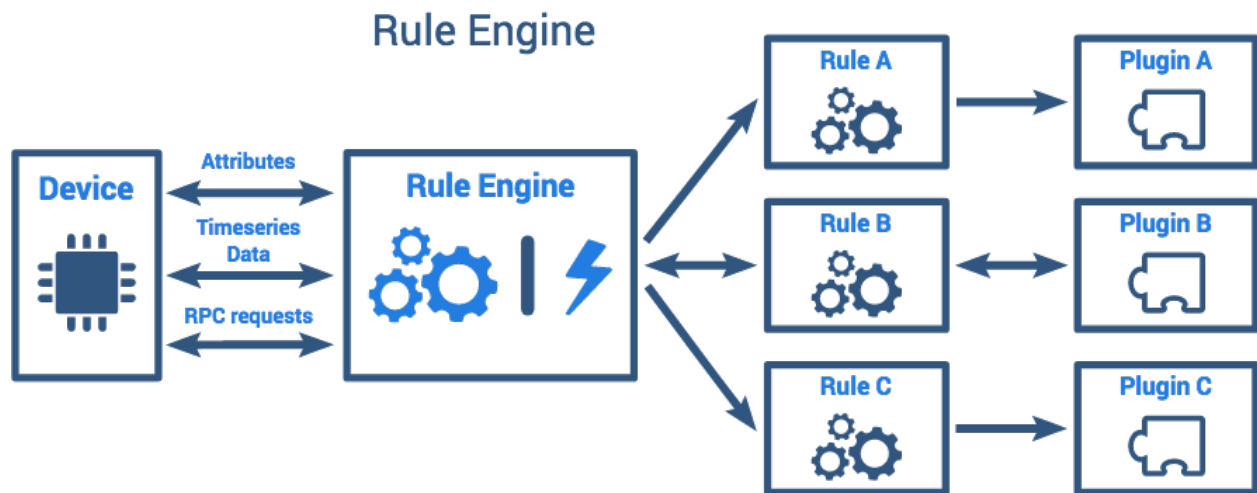
Getting Started

The Rules engine allows you to process messages from devices with a configurable set of rules. With Rule Engine you can:

- send an email when device attribute changes.
- create an alarm when telemetry value exceeds a certain threshold.
- forward telemetry data to Kafka, RabbitMQ or external RESTful server.
- and much more using other rules and plugins.

Please Note we recommend to get familiar with basic Tempus Cloud features below, before you proceed:

Rule engine operates with two main components: Rules and Plugins. The job of Rule Engine is to sequentially apply configured rules to incoming messages. Although rules are applied sequentially, rule execution is asynchronous and is based on actors. This allows processing of messages from the device in the same order in which they are received without sacrificing performance. The results of processing may or may not be delivered back to device application. This depends on rule configuration. For example, you may push incoming telemetry data to internal Cassandra DB, and simultaneously send it to Kafka. You are able to configure to ignore Kafka errors or report an error back to the device. In this case, the device may either retry the operation, re-send the data to a different server or simply ignore the error.



Rule vs Plugin

We will review following example to explain the difference between Rule and Plugin. Let's assume you want to send an email to an engineer when the engine temperature is too high. In this case, Rule is responsible for analyzing telemetry data and building email (body, to, cc, etc). However, the plugin is responsible for actual communication with the email server and sending emails. So, multiple rules may use the same email plugin that is configured once.

Scopes

Rules and Plugins may be operating on System and Tenant levels.

- **System** level rules and plugins are managed by System Administrator. They process messages from all devices.
- **Tenant** level rules and plugins are managed by corresponding Tenant Administrator. They process messages from devices that belong to particular tenant. Tenant level rules are able to target system level plugins. For example, you can configure plugin that sends notifications on the system level. You will need to specify your email account in the plugin configuration. However, all tenant level rules may use this plugin to send email notifications.

Lifecycle

Tempus Cloud Rules and Plugins components have same lifecycle events:

- **Created** - component is provisioned, but is not processing any messages yet.
- **Activated** - component is able to receive and process new device messages.
- **Suspended** - component is not able to receive new device messages.
- **Deleted** - component is stopped and deleted from the database.

Rules

Tempus Cloud Rule consists of three main components: Filters, Processor and Action. Depending on implementation, each component may require certain configuration before it can be used. In order to configure Rule, you need to specify at least one filter and one action. Rule Processors are optional. Let's review role of each component.

Filters

Rule Filter is responsible for filtering incoming messages. You can treat it as a boolean function that has device attributes and message as parameters:

```
boolean filter(DeviceAttributes attributes, FromDeviceMessage message)
```

Tempus Cloud provides the following Rule filters out of the box:

- Message Type Filter - allows to filter incoming messages by type.
- Device Attributes Filter - allows to filter incoming messages based on current device attributes. You can define filter using javascript. See filter documentation for more details.
- Device Telemetry Filter - allows to filter incoming “Post Telemetry” message based on it’s values. You can define filter using javascript. See filter documentation for more details.
- Method Name Filter - allows to filter incoming “RPC Request” message based on it’s method name.

Single Rule may contain multiple filters. Usually, one can filter based on message type and device attributes first and then apply additional filtering based on content of the message.

Processors

Rule Processor is responsible for processing incoming message and adding metadata to it. You can treat it as a function that has device attributes and message as parameters and returns certain metadata:

```
MessageMetadata process(DeviceAttributes attributes, FromDeviceMessage message)
```

Tempus Cloud provides the following Rule processors out of the box:

- Alarm Deduplication Processor generates and persists unique alarms. Populates certain metadata tag to identify new alarms. See filter documentation for more details.

Actions

Plugin Action is responsible for converting incoming message and metadata to new custom message that is forwarded to certain plugin. Action may be oneway or twoway. In case of oneway action, rule is not expecting any reply from plugin. In case of twoway action, rule is expecting a reply from plugin within a certain timeout. If there is no reply within a configured timeout, Rule will report an error to the device. For example, storing data into the internal storage is twoway action, however, pushing data into external system may be oneway (optional). Although particular Action is a part of the corresponding Rule, you can also treat it as part of the corresponding Plugin. So, you can treat an action as the following interface between Rule and Plugin:

```
RuleToPluginMessage<T> convert(DeviceAttributes attributes, FromDeviceMessage message,  
    ↳ MessageMetadata metadata)  
  
ToDeviceMsg convert(PluginToRuleMsg<T> message)  
  
boolean isOneWay()
```

Tempus Cloud provides the following Actions out of the box:

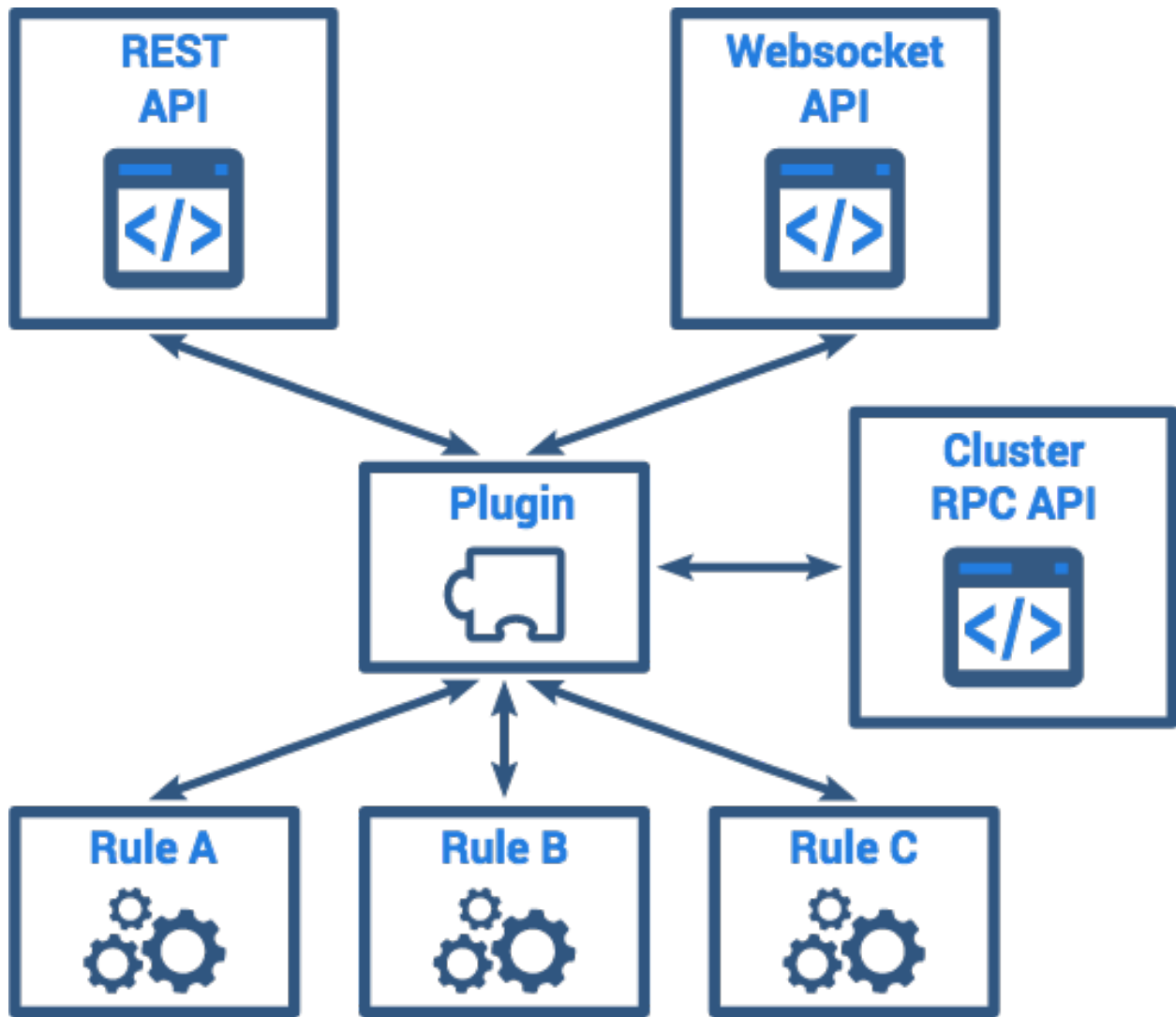
- Telemetry Plugin Action - system action that push data to system Telemetry Plugin.

- Send Mail Action - action that allows to send emails. You can specify templates for email body and recipients. During templates evaluation, you may substitute device attributes and data from incoming messages. See action documentation for more details.
- Kafka Plugin Action - action that allows to push messages to kafka topics. You can specify template for message body. During templates evaluation you may substitute device attributes and data from incoming messages. See action documentation for more details.
- RabbitMQ Plugin Action - similar to Kafka action but pushes data to RabbitMQ.
- REST API Call Plugin Action - similar to Kafka action but executes a REST API call.
- RPC Plugin Action - forwards device RPC call to corresponding plugin.

Plugins

Tempus Cloud Plugins allow you to configure and customize system behavior. Although plugins are able to process messages from Rules, they also provide APIs to integrate with your server-side applications. With Plugin you can: *

- * process messages from devices
- * process REST API calls from server side applications
- * communicate with server-side applications using websockets.
- * communicate between instances of the same plugin in the Tempus Cloud cluster using asynchronous RPC calls.
- * persist and query events, telemetry data, and device attributes.



As we already discussed, Rules may communicate with Plugins using Actions. Let's review other Plugin APIs.

REST API

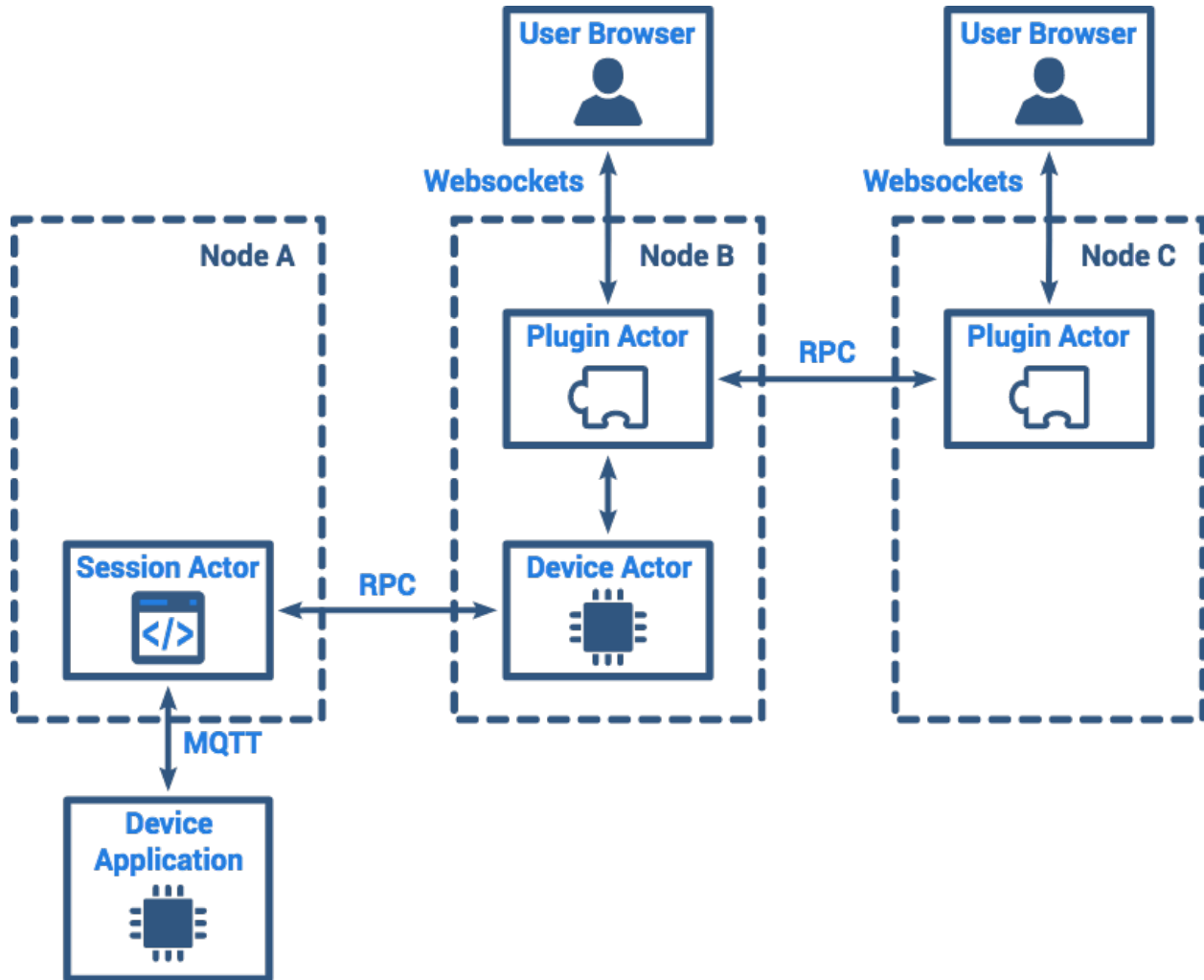
Plugins are able to process REST API calls from authorized users - customers and system or tenant administrators. This may be useful for server-side integrations. For example, System RPC Plugin allows executing RPC calls to devices using REST API.

Websocket API

Plugins are able to handle websocket messages from authorized users - customers and system or tenant administrators. This may be useful for integrations with server-side applications that want to receive real-time updates. For example, System Telemetry Plugin allows subscribing to device attributes and timeseries data changes using websockets.

Clustering API

When plugin is provisioned, Tempus Cloud creates an instance of the plugin actor on each Tempus Cloud server in the cluster. In order to implement complex use-cases, plugin instances may require a way to communicate with each other. For example, let's assume that we have a three node cluster (nodes A, B and C). A device may be connected via MQTT session to node A. Customer users may open a Web UI to observe telemetry data in real time and load balancer will forward their browsers to different nodes (B and C in our case). Telemetry plugin needs to keep track of websocket subscriptions for particular device in order to push update to customer's browser.



Implementations

Tempus Cloud provides the following Plugins out of the box:

- * **Telemetry Plugin** - system plugin that is responsible for processing various requests related to device attributes and telemetry.
- * **RPC Plugin** - allows to execute RPC calls to devices using REST API. RPC call will be delivered to device using supported network protocols.
- * **Device Messaging Plugin** - allows devices that are assigned to the same customer exchange events.
- * **Send Mail Plugin** - allows to send emails. You can specify mail server properties. See plugin documentation for more details.
- * **Kafka Plugin** - allows to push telemetry messages to Apache Kafka. See plugin documentation for more details.
- * **RabbitMQ Plugin** - allows to push telemetry messages to RabbitMQ. See plugin documentation for more details.
- * **REST API Call Plugin** - allows to push telemetry messages to external servers using REST API. See plugin documentation for more details.
- * **Time RPC Plugin** - allows to send RPC requests from device to get current server-side timestamp.

Troubleshooting and statistics

Tempus Cloud keeps track of usage statistics, errors and lifecycle events for each rule and plugin.

Statistics

Tempus Cloud collects and periodically persists usage statistics to internal database. You can review this statistics via Web UI or get it using REST API. Statistics contain an amount of successfully processed messages and amount of errors during processing.

Errors

Whenever exception or error occurs, Tempus Cloud persists it to the internal database. You can review this errors via Web UI or get it using REST API. In case of misconfiguration or critical issue with plugin, errors may happen quite frequently. Persisting all errors may significantly reduce performance, so you can configure how often errors are persisted.

Lifecycle events

Whenever plugin or rule lifecycle event happens, Tempus Cloud persists it to the internal database. You can observe status of the lifecycle, stack trace in case of errors, event time and server address. In case of misconfiguration, you can easily detect the root cause by browsing the error stack trace.

Data Visualization

Getting Started

Tempus Cloud allows you to configure customizable IoT dashboards. Each IoT Dashboard may contain multiple dashboard widgets that visualize data from multiple IoT devices. Once IoT Dashboard is created, you may assign it to one of the customers of you IoT project. IoT Dashboards are light-weight and you may have millions of dashboards. For example, you may automatically create a dashboard for each new customer based on data from registered customer IoT devices. Or you may modify dashboard via script when a new device is assigned to a customer. All these actions may be done manually or automated via REST API. You can find useful links to get started below:

- Getting started guide - will cover basic steps to create a dashboard.
- IoT Dashboards - contains tutorials about basic IoT dashboard operations.
- Samples - contains several examples that include both client-side applications and corresponding data visualization.
- **Widget Library - contains an overview of dashboard widget bundles:**
 - Digital and analog gauges for latest real-time values visualization
 - Highly customizable Bar and Line charts for visualization of historical and sliding-window data points
 - Map widgets for tracking movement and latest positions of IoT devices on Google or OpenStreet maps.
 - GPIO control widgets that allow sending GPIO toggle commands to devices.
 - Card widgets to enhance your dashboards with flexible HTML labels based on static content or latest telemetry values from IoT devices.

3.3 Installation

Tempus Cloud can be installed on several different platforms

Platforms

AWS EC2 Installation

This guide describes how to install [Tempus Cloud](#) on AWS EC2 using community AWS AMIs.

Choose AMI type, instance type and region

AMI is based on the microservices version of [Tempus Cloud](#). The microservices are deployed as docker containers using docker-compose on the Amazon Linux 2 OS. This AMI to simplify the deployment and getting started process. We recommend to use the AMI as a trial environment and move to [Tempus Cloud on EKS](#) once you plan a production deployment. For Tempus Cloud AMI you can choose any instance type with at least 4GB of RAM. The AMI is available only in the N. Virginia zone.

Use the following link to start the installation of AMIs:

- [N. Virginia](#)

Configure Instance

No specific configuration items here. You can choose a t2.medium instance or above.

Add Storage

Minimum 20 Gb of Storage is required. We recommend having at least 50 if you plan to upload some data.

Add Tags

No specific configuration items here. You can leave this tab without changes or apply a configuration that is specific to your use-case.

Configure Security Group

We recommend to create new security group, for example “Tempus”. Configure following inbound rules:

Type	Protocol	Port Range	Source
HTTP	TCP	80	0.0.0.0/0
SSH	TCP	22	0.0.0.0/0
Custom TCP Rule	TCP	1883	0.0.0.0/0

Review and launch your instance

Once the instance is launched, please wait some time for services to boot up and open Administration UI in the browser using public DNS from instance details.

Accessing Tempus Cloud service

Once the instance from AMI is created, please access http://your_public_ip to access the application.

You can use the following credentials to login:

username(email)	Password	Role
sysadmin@hashmapinc.com	sysadmin	System Administrator
demo@hashmapinc.com	tenant	Tenant
bob.jones@hashmapinc.com	driller	User

NOTE: Please allow up to 90 seconds for the Web UI to start

Troubleshooting

Tempus Cloud logs can be viewed using:

```
docker logs -f $(docker ps -q --filter "ancestor=hashmapinc/tempus:dev")
```

For identity and discovery service logs:

```
docker logs -f $(docker ps -q --filter "ancestor=hashmapinc/redtail-api-  
↪discovery:latest")  
docker logs -f $(docker ps -q --filter "ancestor=hashmapinc/redtail-identity-  
↪service:latest")
```

You can issue the following command in order to check if there are any errors on the backend side:

```
docker logs $(docker ps -q --filter "ancestor=hashmapinc/hashmapinc/tempus:dev") |  
↪grep ERROR
```

Issue Logging

You can log your issues at:

- [GitHub](#)

Schema Upgrade

Schema Upgrade For Stories

These commands need to be run manually to alter/update the database for the mentioned stories.

Tempus-364 Serverless compute on Tempus

1. The key point here is now, there can be computations other than spark computation namely kubeless computations.
2. The schema of the computation tables which currently holds the configuration parameters like jar_name, jar_path etc will not be present in kubeless computation.

3. This means kubeless computation have different configuration, hence a need for schema alteration of computations.

For SQL

1. Refer the file `../../../../dao/src/main/resources/sql/upgrade/5.sql`

For NOSQL

2. Refer the file `../../../../dao/src/main/resources/cassandra/upgrade/5.cql`

3.4 API

Depending on the use case there are several different API's that can be leveraged in Tempus. The following is a high level breakdown of the API's available in Tempus.

Data Acquisition

Provisioning Nifi WITSML FLOW

Getting started

This will guide you through how to provision nifi flow to fetch data for selective mnemonics for logs using nifi-witsml-bundle.

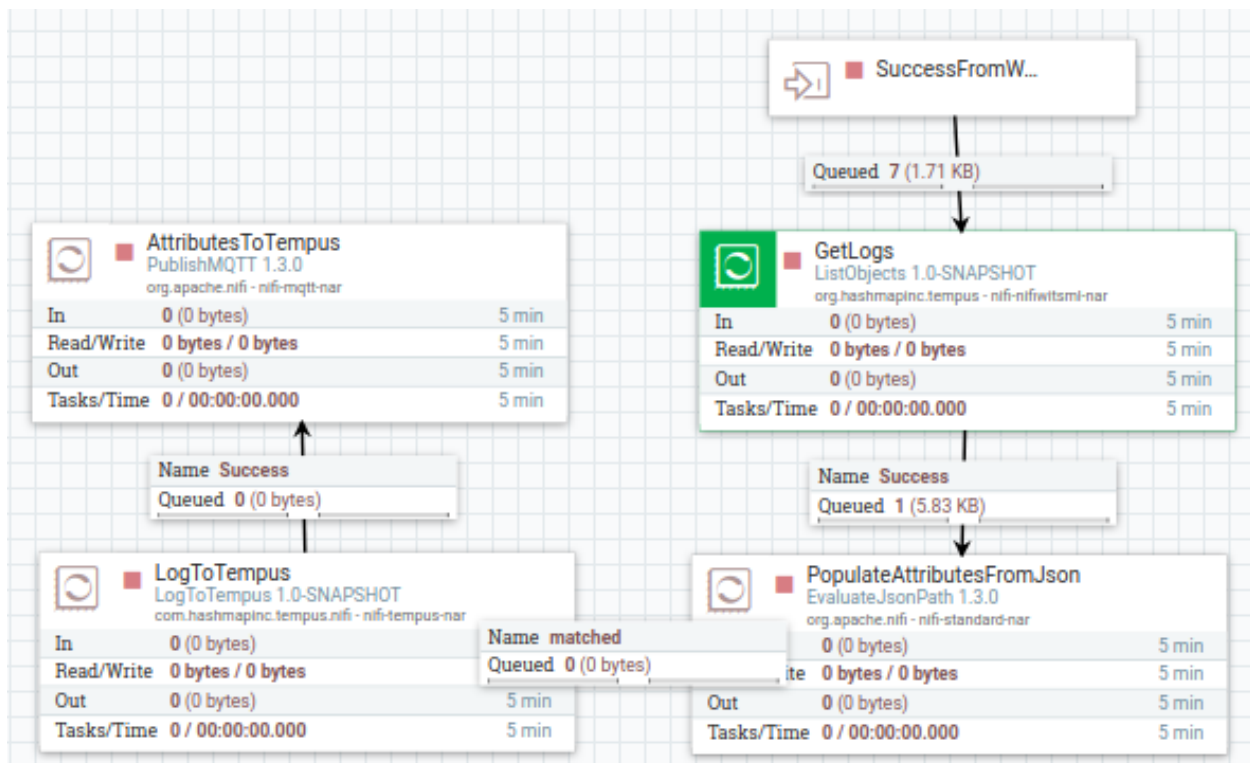
Setup and Requirements

You will need nifi-tempus-bundle and nifi-witsml-bundle for creating provisioning flow. Clone nifi-tempus-bundle and nifi-witsml-bundle and build the respective maven projects. Finally copy the nar files to nifi lib directory.

- nifi-tempus-bundle : <https://github.com/hashmapinc/nifi-tempus-bundle>
- nifi-witsml-bundle : <https://github.com/hashmapinc/nifi-witsml-bundle>

Fetch Log Mnemonic List

1. On your nifi canvas create a flow as shown in image. This flow will let you fetch mnemonics for log object.



2. Configure ListObjects(GetLogs) processor as shown. This processor will fetch log meta-data with mnemonic list from Witsml server.

Configure Processor

SETTINGS	SCHEDULING	PROPERTIES	COMMENTS
Required field +			
Property	Value		
WITSML Service	?	Witsml1311Service_Tempus	→
Parent URI	?	\${uri}	
Object Types	?	log	
Well Status Filter	?	drilling	
Distributed Cache Service	?	LOG-DistrMapCacheCIntService	→
Maintain Query State	?	true	

3. EvaluateJsonPath(PopulateAttributesFromJson) processor will add flowfile-attributes from logs json data. Configuration is being shown in image, don't forget to add the highlighted mnemonicList attribute.

Configure Processor

SETTINGS

SCHEDULING

PROPERTIES

COMMENTS

Required field +

Property	Value	
Destination	? flowfile-attribute	
Return Type	? json	
Path Not Found Behavior	? ignore	
Null Value Representation	? empty string	
direction	? direction	🗑
endDateTimeIndex	? endDateTimeIndex	🗑
endIndex	? endIndex.value	🗑
indexType	? indexType	🗑
messageLog	? name	🗑
mnemonicList	? \$.logCurveInfo[*].mnemonic.value	🗑
name	? name	🗑
nameWell	? nameWell	🗑
nameWellbore	? nameWellbore	🗑
startDateTimeIndex	? startDateTimeIndex	🗑

CANCEL

APPLY

4. To validate the process you will need to list-queue between EvaluateJsonPath and LogToTempus. On attribute tab you can see mnemonicList as below.

FlowFile

DETAILS

ATTRIBUTES

Attribute Values

messageLog

EcoScope Data - Time Log

mime.type

application/json

mnemonicList

```
[ "TIME","ACTC","AJAM_MWD","AZIM_CONT_RT","BDTI","BITRUN","BONB","BPOS","BVEL","CRPM_RT","DBTM","DBTV","DCAV_D
H_ECO_RT","DEPT","DHAP_DH_ECO_RT","DHAT_DH_ECO_RT","DMEA","ECD_ECO_RT","GRMB_DH_ECO_RT","HKLD","HKLD3
0s","INCL_CONT_RT","MBOT","MON_DH_ECO_RT","MWTI","PASS_NAME","PD_RTSTAT","PDAZIMLO","PDINCL","PDSHKRSK","
PDSTEER","PMPT","QC_DRIL_DH_ECO_RT","QC_SPEC_DH_ECO_RT","ROP","ROP30s","ROP5","RPM","RPM30s","SDEP_CONT
_RT","SHKL_DH_ECO_RT","SHKTOT_RT","SPM1","SPM2","SPM3","SPPA","STICK_RT","SWOB","SWOB30s","TFLO","TFLO30s","T
HKD","TQA","TREV","TRPM_RT","TSPM","TVDE","VIBLAT_RT","VIBX_RT"]
```

name

EcoScope Data - Time Log

nameWell

Demonstration Well

OK

5. To convert the logs mnemonic list to Tempus gateway device attribute json format you will use LogToTempus Processor.

6. LogToTempus processor configuration are shown as follows.

Configure Processor

SETTINGS

SCHEDULING

PROPERTIES

COMMENTS

Required field



Property		Value	
Device Name	?	nameWell	
Log Name	?	name	
Log Type	?	attribute	
Log Source Fieldname	?	DEPTH	

7. Finally configure PublishMqtt(AttributesToTempus) processor to publish the generated json to Tempus.

Configure Processor

SETTINGS

SCHEDULING

PROPERTIES

COMMENTS

Required field

+

Property	Value
Broker URI	<div>?</div> tcp://192.168.1.135:1883
Client ID	<div>?</div> nifi
Username	<div>?</div> GatewayToken
Password	<div>?</div>
SSL Context Service	<div>?</div> No value set
Last Will Topic	<div>?</div> No value set
Last Will Message	<div>?</div> No value set
Last Will Retain	<div>?</div> No value set
Last Will QoS Level	<div>?</div> No value set
Session state	<div>?</div> Clean Session
MQTT Specification Version	<div>?</div> AUTO
Connection Timeout (seconds)	<div>?</div> 30
Keep Alive Interval (seconds)	<div>?</div> 60
Topic	<div>?</div> v1/gatewav/attributes

CANCEL

APPLY

8. Move back to your Tempus UI, on that particular Well Device you can see client attributes as log mnemonics as shown in image.

DEMONSTRATION WELL

Device details

?

✕

<

DETAILS

ATTRIBUTES

LATEST TELEMETRY

LATEST DEPTH

ALARMS

OPEN >

Entity attributes scope

Client attributes

▼

Client attributes

🔍

<input type="checkbox"/>	Last update time	Key ↑	Value
<input type="checkbox"/>	2018-04-09 18:42:50	W-437456/B-437515/L-437516-MD@AJAM_MWD	false
<input type="checkbox"/>	2018-04-09 18:42:50	W-437456/B-437515/L-437516-MD@BDTI	false
<input type="checkbox"/>	2018-04-09 18:42:50	W-437456/B-437515/L-437516-MD@BITRUN	false
<input type="checkbox"/>	2018-04-09 18:42:50	W-437456/B-437515/L-437516-MD@BPHI_ECO_RT	false
<input type="checkbox"/>	2018-04-09 18:42:50	W-437456/B-437515/L-437516-MD@CRPM_RT	false

Page:

2

▼

Rows per page:

5

▼

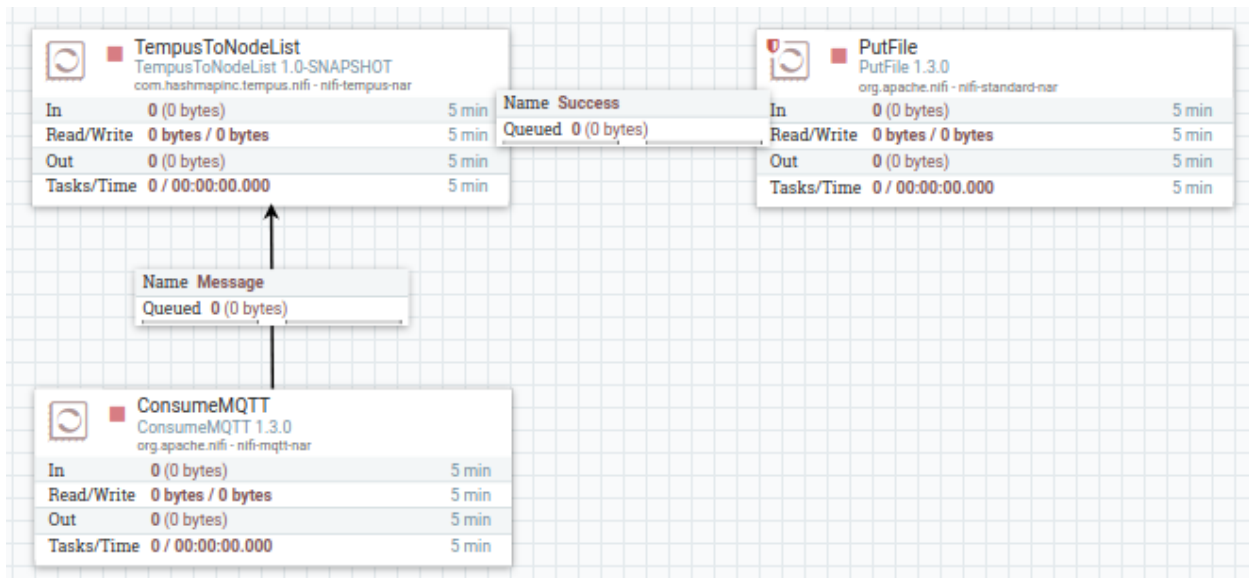
6 - 10 of 316

<

>

Fetch Data for selected Mnemonic using Tempus Attributes

1. Create a nifi flow as follows to subscribe to shared attributes of tempus device and store the list of mnemonics in a file.



2. ConsumeMqtt processor will subscribe to Tempus Device Attribute topic for shared attributes :

Configure Processor

SETTINGS
SCHEDULING
PROPERTIES
COMMENTS

Required field +

Property	Value
Broker URI	tcp://192.168.1.135:1883
Client ID	Well
Username	vQqU50U435NEToXNUdyv
Password	
SSL Context Service	No value set
Last Will Topic	No value set
Last Will Message	No value set
Last Will Retain	No value set
Last Will QoS Level	No value set
Session state	Clean Session
MQTT Specification Version	AUTO
Connection Timeout (seconds)	30
Keep Alive Interval (seconds)	60
Topic Filter	v1/devices/me/attributes

CANCEL
APPLY

3. TempusToNodeList is processor from nifi-tempus-bundle will process the different types shared attributes json messages from tempus.
4. TempusToNodeList takes File path which will be used as storage for extracted node mnemonic list from Json messages. You also need to select the Data type as “WITSML”.

Configure Processor

SETTINGS	SCHEDULING	PROPERTIES	COMMENTS
----------	------------	------------	----------

Required field +

Property	Value
File Path	? /home/pc/Output/witsml.json
Data Type	? WITSML

5. Output of TempusToNodeList will be passed on to PutFile processor to write/replace the updated node list to the storage file :

Configure Processor

SETTINGS	SCHEDULING	PROPERTIES	COMMENTS
----------	------------	------------	----------

Required field +

Property	Value
Directory	? /home/pc/Output
Conflict Resolution Strategy	? replace
Create Missing Directories	? true
Maximum File Count	? No value set
Last Modified Time	? No value set
Permissions	? No value set
Owner	? No value set
Group	? No value set

CANCELAPPLY

NOTE : Change the conflict Resolution Strategy to “replace”.

6. Start the nifi ConsumeMqtt processor and move back to Tempus UI.
7. Select the mnemonics from Tempus UI client attribute window for which you want to fetch data and click on SEND TO SHARED button :

DEMONSTRATION WELL

Device details

?

✕

<

DETAILS

ATTRIBUTES

LATEST TELEMETRY

LATEST DEPTH

ALARMS

OPEN >

Entity attributes scope

Client attributes

3 attributes selected

SEND TO SHARED

SHOW ON WIDGET

<input type="checkbox"/>	Last update time	Key ↑	Value
<input checked="" type="checkbox"/>	2018-04-09 18:42:50	W-437456/B-437515/L-437516-MD@AJAM_MWD	false
<input checked="" type="checkbox"/>	2018-04-09 18:42:50	W-437456/B-437515/L-437516-MD@BDTI	false
<input type="checkbox"/>	2018-04-09 18:42:50	W-437456/B-437515/L-437516-MD@BITRUN	false
<input checked="" type="checkbox"/>	2018-04-09 18:42:50	W-437456/B-437515/L-437516-MD@BPHI_ECO_RT	false
<input type="checkbox"/>	2018-04-09 18:42:50	W-437456/B-437515/L-437516-MD@CRPM_RT	false

Page: 2 Rows per page: 5 6 - 10 of 316 < >

8. Selected mnemonics will be added to the shared attributes window of the device with their value as true :

DEMONSTRATION WELL
Device details

DETAILS
ATTRIBUTES
LATEST TELEMETRY
LATEST DEPTH
ALARMS

Entity attributes scope
Shared attributes

Shared attributes

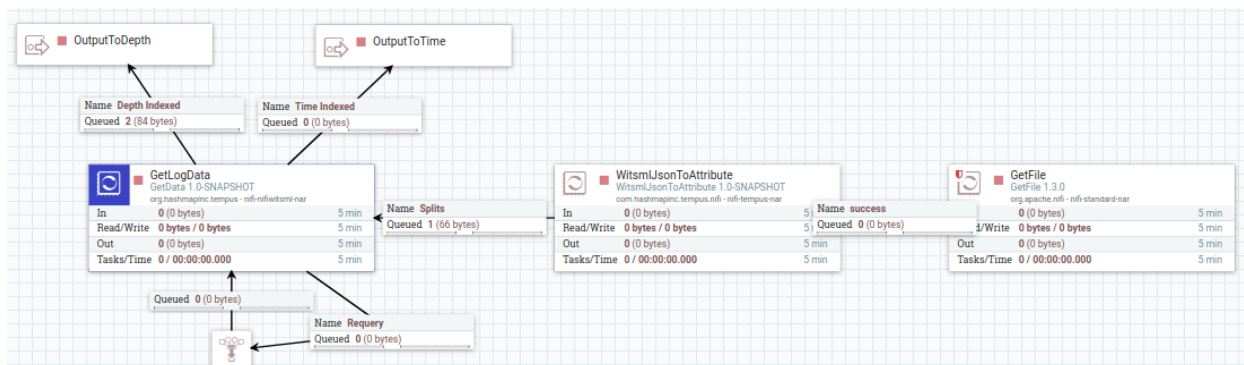
	Last update time	Key ↑	Value
<input type="checkbox"/>	2018-04-10 15:51:32	Demonstration Well.W-437456/B-437515/L-437516-MD@AJAM_MWD	true
<input type="checkbox"/>	2018-04-10 15:51:32	Demonstration Well.W-437456/B-437515/L-437516-MD@BDTI	true
<input type="checkbox"/>	2018-04-10 15:51:32	Demonstration Well.W-437456/B-437515/L-437516-MD@BPHI_ECO_RT	true

Page: 1
Rows per page: 5
1 - 3 of 3

- On the nifi side these attributes will be subscribed by ConsumeMqtt processor and passed on to the TempusToN-odeList processor for processing and storing them in a storage file using PutFile.

Start fetching data for mnemonic list from witsml server

- Create nifi flow to read witsml-storage file and pass the mnemonics list to WitsmlJsonToAttribute processor.



- GetFile** processor will read the storage file in which you have been putting mnemonics list in previous steps.

Configure Processor

SETTINGS	SCHEDULING	PROPERTIES	COMMENTS
----------	------------	------------	----------

Required field +

Property	Value
Input Directory	? /home/pc/Output
File Filter	? [^\\].*
Path Filter	? No value set
Batch Size	? 10
Keep Source File	? true
Recurse Subdirectories	? true
Polling Interval	? 0 sec
Ignore Hidden Files	? true
Minimum File Age	? 0 sec
Maximum File Age	? No value set
Minimum File Size	? 0 B
Maximum File Size	? No value set

CANCEL APPLY

NOTE : make “Keep Source File” as true.

- WitsmlJsonToAttribute will read the flowfile content and add wellId, wellboreId, logId, wellName and mnemonics as flowfile attributes.
- Data from WitsmlJsonToAttribute will be passed to GetData(GetLogData) processor. GetData processor will fetch data for only the selected mnemonics.
- LogData from processor can be passed to LogToTempus processor to process the data and convert it into Tempus GatewayJson format. Finally this data can be published through PublishMqtt processor.

Streaming Data Quality Analysis

Introduction

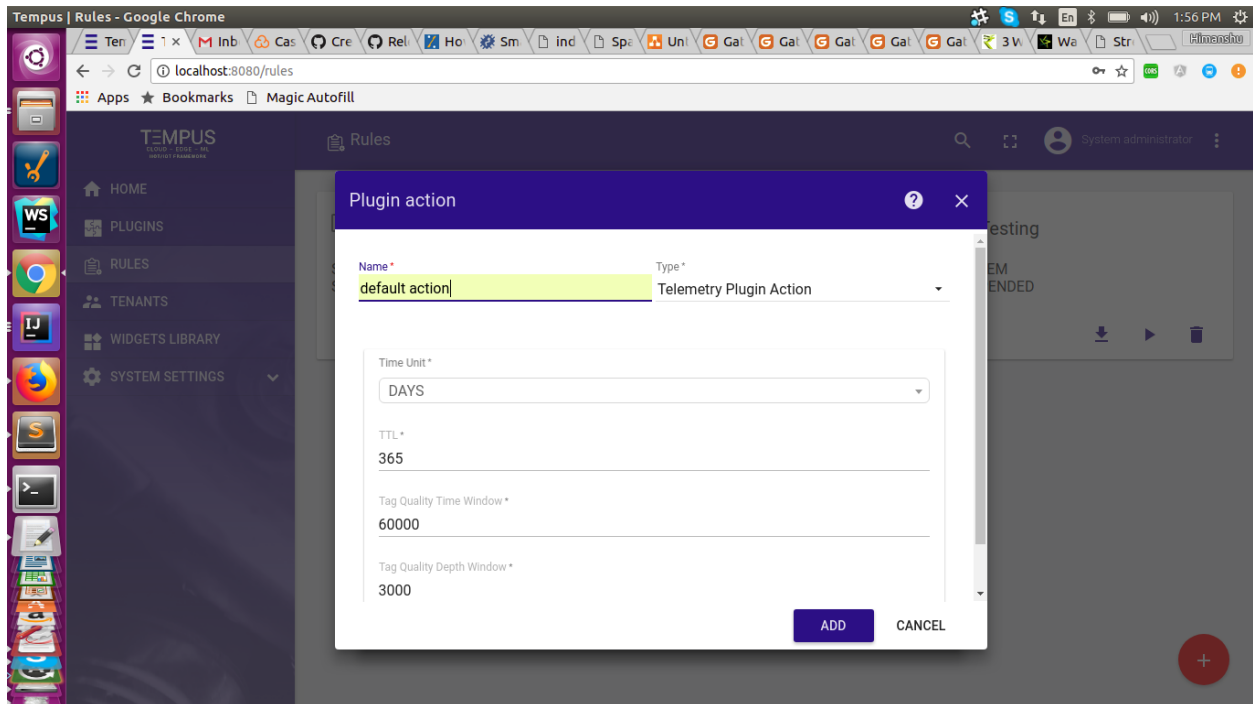
Tempus supports the streaming data quality analysis, which means the meta data and quality information related to tags(i.e. keys) in telemetry data can be processed and stored.

For quality timeseries and depthseires data is processed over a period of time and depth respectively and various aggregation constants avg, mean, median etc are calculated and stored for timestamp and depthstamp for each tag.

Configuration for Timeseries data

- For timeseries data System telemetry plugin and rule is used.
- The action tab in the system telemetry rule consists of a configuration parameter **Tag Quality Time Window**.

This configuration is used to set the Time period over which processing will take place. Refer image:



Configuration for Depthseries data

- For depthseries data System telemetry plugin is used.
- Create a tenant rule for System telemetry plugin for depthseries data forwarding.
- In the action tab of this new rule created set this configuration parameter **Tag Quality Depth Window**.

This configuration is used to set the Depth period over which processing will take place.

Sparkplug B Specification(Spec) support

Getting Started

Sparkplug basics

Sparkplug is a specification for MQTT enabled devices and applications to send and receive messages in a stateful way. While MQTT is stateful by nature it doesn't ensure that all data on a receiving MQTT application is current or valid. Sparkplug provides a mechanism for ensuring that remote device data is current and valid.

Sparkplug B specification specifies how various MQTT devices must connect and disconnect from the MQTT server i.e. Tempus in our case. This includes device lifecycle messages such as the required birth and last will & testament messages that must be sent to ensure the device lifecycle state and data integrity.

Sparkplug specification doc can be found here : <https://s3.amazonaws.com/cirrus-link-com/Sparkplug+Specification+Version+1.0.pdf>

Open source library used

- We use cirrus link java client library for encoding and decoding the sparkplug B format data.
- Link of repo <https://github.com/Cirrus-Link/Sparkplug.git>
- This library internally uses Google protobuf for encoding/ decoding purpose.

Implementation

In tempus the data gets published to devices through MQTT clients, so we primarily focus on the sparkplug message types related to device only i.e DBIRTH, DDATA, DDEATH. The state management for devices will also include usage of these message types.

The implementation is divided in the following stages:

- Persisting sparkplug b specification payload data to tempus devices(sparkplug devices).
- State management for sparkplug devices on tempus side.
- Subscription of sparkplug b spec data devices on tempus by some external app.

Persisting sparkplug b specification payload data to tempus devices

- Tempus uses a gateway device to act as a bridge between external devices connected to different systems and Tempus. Gateway API provides the ability to exchange data between multiple devices and the platform using single MQTT connection.
- Publishing to a device through a gateway device results in creation of that device if it does not exists.
- Similarly a sparkplug B supported message would be published through a gateway device resulting in creation of new devices, if not already created, which are compliant with sparkplug B specification.
- The topic name for publish would be different for each of the sparkplug device compliant with topic name format defined in sparkplug B specification. For example the format structure is:

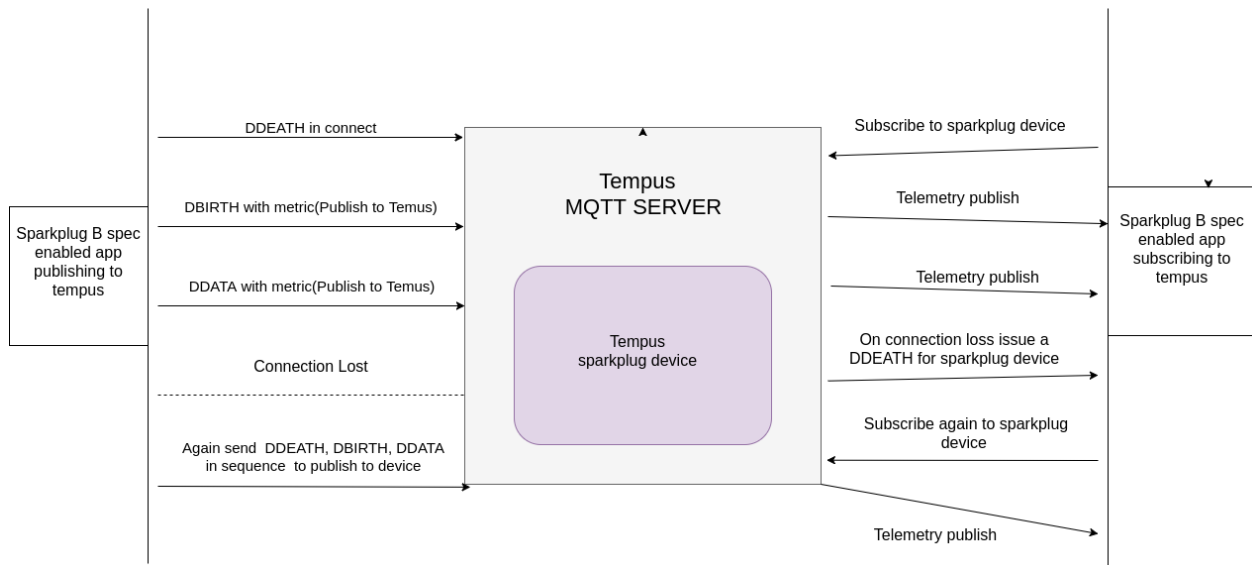
namespace/group_id/msg_type/edge_node_id/device_id

- The **namespace** element of the Topic Namespace is the root element that will define both the structure of the remaining namespace elements as well as the encoding used for the associated payload data.
 - The **group_id** element of the Topic Namespace provides for a logical grouping of MQTT EoN and Devices into the MQTT Server and back out to the consuming MQTT Clients.
 - The **message_type** element of the Topic Namespace provides an indication as to what the MQTT Payload of message will contain(DDEATH, DBIRTH, DDATA).
 - The **edge_node_id** element of the Sparkplug Topic Namespace uniquely identifies the MQTT EoN Node within the infrastructure.
 - The **device_id** element of the Sparkplug Topic Namespace identifies an MQTT Device attached to the MQTT EoN Node. In our case it would be the name of device.
- The device access token would be the gateway device token.

State management for devices adhering to sparkplug b spec on tempus side

- Dbirth, Ddeath, and Ddata messages would be used to manage the state of devices.

- For example if the client publishing loses connection with tempus mqtt server, the subscriber client connected to tempus mqtt server is sent a Ddeath msg.
- If the client comes up again to publish data, it should publish Dbirth again followed by Ddata otherwise the publish message would be discarded.
- Refer to the below diagram:



Subscription of sparkplug b spec data on tempus devices(sparkplug devices) by some external app

- Telemetry data for each of sparkplug device on tempus can be subscribed on a separate topic, whose name is as per the sparkplug specification.
- The subscriber would be able to see the telemetry data if the telemetry data is being published to that sparkplug device at that point of time.

Usage/ Testing the sparkplug B spec support in Tempus

- To test the sparkplug B spec support on Tempus, we need to have something which publishes data to tempus in sparkplug B format.
- For this purpose we can again use the open source cirrus link sparkplug repo.
- Standalone sparkplug example for java i.e. SparkplugExample.java can be slightly modified for publishing as well as subscribing telemetry data to and from Tempus.

API Usage

3.5 Security

Tempus MQTT Server and clients can be secure with multiple options.

Security Options

MQTT over SSL

Tempus provides the ability to run MQTT server over SSL. Both one-way and two-way SSL are supported. To enable SSL, you will need to obtain a valid or generate a self-signed SSL certificate and add it to the keystore. Once added, you will need to specify the keystore information in **tempus.yml** file. See the instructions on how to generate SSL certificate and use it in your Tempus installation below. You can skip certificate generation step if you already have a certificate.

Self-signed certificate generation

Note This step requires Linux based OS with Java installed.

Set and export the following environment variables. Updating the values where needed. For example:

```
DOMAIN_SUFFIX="$(hostname) "
ORGANIZATIONAL_UNIT=Tempus
ORGANIZATION=Tempus
CITY=Roswell
STATE_OR_PROVINCE=GA
TWO_LETTER_COUNTRY_CODE=US

SERVER_KEYSTORE_PASSWORD=server_ks_password
SERVER_KEY_PASSWORD=server_key_password

SERVER_KEY_ALIAS="serveralias"
SERVER_FILE_PREFIX="mqttserver"
SERVER_KEYSTORE_DIR="/etc/tempus/conf/"

CLIENT_KEYSTORE_PASSWORD=password
CLIENT_KEY_PASSWORD=password

CLIENT_TRUSTSTORE="client_truststore"
CLIENT_KEY_ALIAS="clientalias"
CLIENT_FILE_PREFIX="mqttclient"
```

where

- **DOMAIN_SUFFIX** - Corresponds to **CN** value of the certificate. Must correspond to the target server domain (wildcards are allowed). Defaults to the current hostname
- **ORGANIZATIONAL_UNIT** - Corresponds to **OU** value of the certificate.
- **ORGANIZATION** - Corresponds to **O** value of the certificate.
- **CITY** - Corresponds to **L** value of the certificate.
- **STATE_OR_PROVINCE** - Corresponds to **ST** value of the certificate.
- **TWO_LETTER_COUNTRY_CODE** - Corresponds to **C** value of the certificate.
- **SERVER_KEYSTORE_PASSWORD** - Server Keystore password
- **SERVER_KEY_PASSWORD** - Server Key password. May or may not be the same as **SERVER_KEYSTORE_PASSWORD**
- **SERVER_KEY_ALIAS** - Server key alias. Must be unique within the keystore
- **SERVER_FILE_PREFIX** - Prefix to all server keygen-related output files

- **SERVER_KEYSTORE_DIR** - The default location where the key would be optionally copied. Can be overridden by -d option in server.keygen.sh script or entered manually upon the scrip run

The rest of the values are not important for the server keystore generation To run the server keystore generation, use following commands.

```
keytool -genkeypair -v \
-alias $SERVER_KEY_ALIAS \
-dname "CN=$DOMAIN_SUFFIX, OU=$ORGANIZATIONAL_UNIT, O=$ORGANIZATION, L=$CITY, ST=
→$STATE_OR_PROVINCE, C=$TWO_LETTER_COUNTRY_CODE" \
-keystore $SERVER_FILE_PREFIX.jks \
-keypass $SERVER_KEY_PASSWORD \
-storepass $SERVER_KEYSTORE_PASSWORD \
-keyalg RSA \
-keysize 2048 \
-validity 9999

keytool -export \
-alias $SERVER_KEY_ALIAS \
-keystore $SERVER_FILE_PREFIX.jks \
-file $SERVER_FILE_PREFIX.pub.pem -rfc \
-storepass $SERVER_KEYSTORE_PASSWORD

keytool -export \
-alias $SERVER_KEY_ALIAS \
-file $SERVER_FILE_PREFIX.cer \
-keystore $SERVER_FILE_PREFIX.jks \
-storepass $SERVER_KEYSTORE_PASSWORD \
-keypass $SERVER_KEY_PASSWORD

mkdir -p $SERVER_KEYSTORE_DIR
cp $SERVER_FILE_PREFIX.jks $SERVER_KEYSTORE_DIR
```

The keytool will used the configuration specified and will generate the following output files:

- **SERVER_FILE_PREFIX.jks** - Java keystore file. This is the file which will be used by Tempus MQTT Service
- **SERVER_FILE_PREFIX.cer** - Server public key file. It will be then imported to client's .jks keystore file.
- **SERVER_FILE_PREFIX.pub.pem** - Server public key in **PEM** format, which can be then used as a keystore or imported by non-Java clients.

To copy the keystore file, upload it manually to a directory which is in server's classpath. You may want to modify owner and permissions for the keystore file:

```
sudo chmod 400 /etc/tempus/conf/mqttserver.jks
sudo chown tempus:tempus /etc/tempus/conf/mqttserver.jks
```

Server configuration

Locate your **tempus.yml** file and uncomment the lines after “*# Uncomment the following lines to enable ssl for MQTT*”:

```
# MQTT server parameters
mqtt:
bind_address: "${MQTT_BIND_ADDRESS:0.0.0.0}"
bind_port: "${MQTT_BIND_PORT:8883}"
```

(continues on next page)

```

adaptor: "${MQTT_ADAPTOR_NAME:JsonMqttAdaptor}"
timeout: "${MQTT_TIMEOUT:10000}"
# Uncomment the following lines to enable ssl for MQTT
ssl:
  key_store: mqttserver.jks
  key_store_password: server_ks_password
  key_password: server_key_password
  key_store_type: JKS

```

You may also want to change **mqtt.bind_port** to 8883 which is recommended for MQTT over SSL servers. The **key_store** Property must point to the **.jks** file location. **key_store_password** and **key_password** must be the same as were used in keystore generation.

NOTE: Tempus supports **.p12** keystores as well. if this is the case, set **key_store_type** value to 'PKCS12' After these values are set, launch or restart your tempus server.

Client Examples

See following resources:

- [Device authentication options](#) for authentication options overview
- [Access Token based authentication](#) for example of one-way SSL connection
- [X.509 Certificate Based Authentication](#) for example of two-way SSL connection

Device authentication options

Device credentials are used in order to connect to the Tempus server by applications that are running on the device. Tempus is designed to support different device credentials. There are two supported credentials types at the moment:

- [Access Tokens](#) - general purpose credentials that are suitable for wide range of devices. Access Token based authentication may be used in not encrypted or one-way SSL mode.
 - **Advantages:** supported by resource constrained devices. Low network overhead. Easy to provision and use.
 - **Disadvantages:** may be easily intercepted while using un-encrypted network connection (HTTP instead of HTTPS, MQTT without TLS/SSL, etc).
- [X.509 Certificates](#) - [PKI](#) and [TLS](#) standard. X.509 Certificate based authentication is used in two-way SSL mode.
 - **Advantages:** high level of security using the encrypted network connection and public key infrastructure.
 - **Disadvantages:** not supported by some resource constrained devices. Affects battery and CPU usage.

Device credentials need to be provisioned to corresponding device entity on the server. There are multiple ways to do this:

- **Automatically**, using Tempus [Swagger](#). For example during manufacturing, QA or purchase order fulfilment.
- **Manually**, using Tempus [Web UI](#). For example for development purposes, or by system administrator.

Access Token based authentication

Access Token Based Authentication is the default device authentication type. Once the device is created in Tempus, the default access token is generated. It can be changed afterwards. In order to connect the device to a server using Access Token based authentication, the client must specify the access token as part of request URL (for HTTP and CoAP) or as a user name in MQTT connect message. See [supported protocols API](#) for more details.

One-Way MQTT SSL

One-way SSL authentication is a standard authentication mode, where your client device verifies the identity of a server using server certificate. In order to run one-way MQTT SSL, the server certificate chain should be signed by authorized CA or client must import the self-signed server certificate (.cer or .pem) to its trust store. Otherwise, a connection will fail with the 'Unknown CA' error.

X.509 Certificate Based Authentication

X.509 Certificate Based Authentication is used in Two-Way SSL connection. In this case, the certificate itself is the client's ID, thus, Access Token is no longer needed.

Instructions below will describe how to generate a client-side certificate and connect to the server that is running MQTT over SSL. You will need to have the public key of the server certificate in PEM format. See [following instructions](#) for more details on server-side configuration.

Set keygen variables

Export the following environment variable, and update the values if needed:

```
DOMAIN_SUFFIX="$(hostname) "
ORGANIZATIONAL_UNIT=Tempus
ORGANIZATION=Tempus
CITY=Roswell
STATE_OR_PROVINCE=GA
TWO_LETTER_COUNTRY_CODE=US

SERVER_KEYSTORE_PASSWORD=server_ks_password
SERVER_KEY_PASSWORD=server_key_password

SERVER_KEY_ALIAS="serveralias"
SERVER_FILE_PREFIX="mqttserver"
SERVER_KEYSTORE_DIR="/etc/tempus/conf/"

CLIENT_KEYSTORE_PASSWORD=password
CLIENT_KEY_PASSWORD=password

CLIENT_KEY_ALIAS="clientalias"
CLIENT_FILE_PREFIX="mqttclient"
```

Generating SSL Key Pair

Execute the following command to generate the SSL key pair.

```

keytool -genkeypair -v \
  -alias $SERVER_KEY_ALIAS \
  -dname "CN=$DOMAIN_SUFFIX, OU=$ORGANIZATIONAL_UNIT, O=$ORGANIZATION, L=$CITY, ST=
  →$STATE_OR_PROVINCE, C=$TWO_LETTER_COUNTRY_CODE" \
  -keystore $SERVER_FILE_PREFIX.jks \
  -keypass $SERVER_KEY_PASSWORD \
  -storepass $SERVER_KEYSTORE_PASSWORD \
  -keyalg RSA \
  -keysize 2048 \
  -validity 9999

keytool -export \
  -alias $SERVER_KEY_ALIAS \
  -keystore $SERVER_FILE_PREFIX.jks \
  -file $SERVER_FILE_PREFIX.pub.pem -rfc \
  -storepass $SERVER_KEYSTORE_PASSWORD

keytool -export \
  -alias $SERVER_KEY_ALIAS \
  -file $SERVER_FILE_PREFIX.cer \
  -keystore $SERVER_FILE_PREFIX.jks \
  -storepass $SERVER_KEYSTORE_PASSWORD \
  -keypass $SERVER_KEY_PASSWORD

mkdir -p $SERVER_KEYSTORE_DIR
cp $SERVER_FILE_PREFIX.jks $SERVER_KEYSTORE_DIR

```

The keytool commands outputs the following files:

- **CLIENT_FILE_PREFIX.jks** - Java Keystore file with the server certificate imported
- **CLIENT_FILE_PREFIX.nopass.pem** - Client certificate file in PEM format to be used by non-java client
- **CLIENT_FILE_PREFIX.pub.pem** - Client public key

Provision Client Public Key as Device Credentials

Go to **Tempus Web UI -> Devices -> Your Device -> Device Credentials**. Select X.509 Certificate device credentials, insert the contents of **CLIENT_FILE_PREFIX.pub.pem** file and click save. Alternatively, the same can be done through the REST API.

3.6 Administration UI

Tempus Cloud provides and Administration user interface

Administration

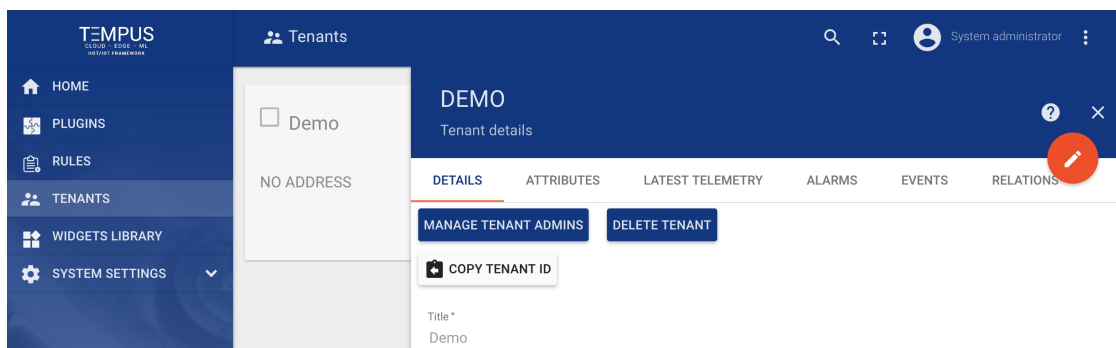
Tenants

Tempus support [Multitenancy](#) out-of-the-box. You can treat Tempus tenant as a separate business-entity: individual or organization who owns or produce devices.

System administrator is able to create tenant entities.



System administrator is also able to create multiple users with **Tenant Administrator** role for each tenant by pressing “Manage Tenant Admins” button in Tenant details.



Tenant Administrator is able to do following actions:

- Provision and Manage *Devices*
- Provision and Manage *Assets*
- Create and Manage *Business Units*
- Create and Manage *Dashboards*
- Configure *Rules* and *Plugins*
- Add or modify default widgets using *Widget Library*

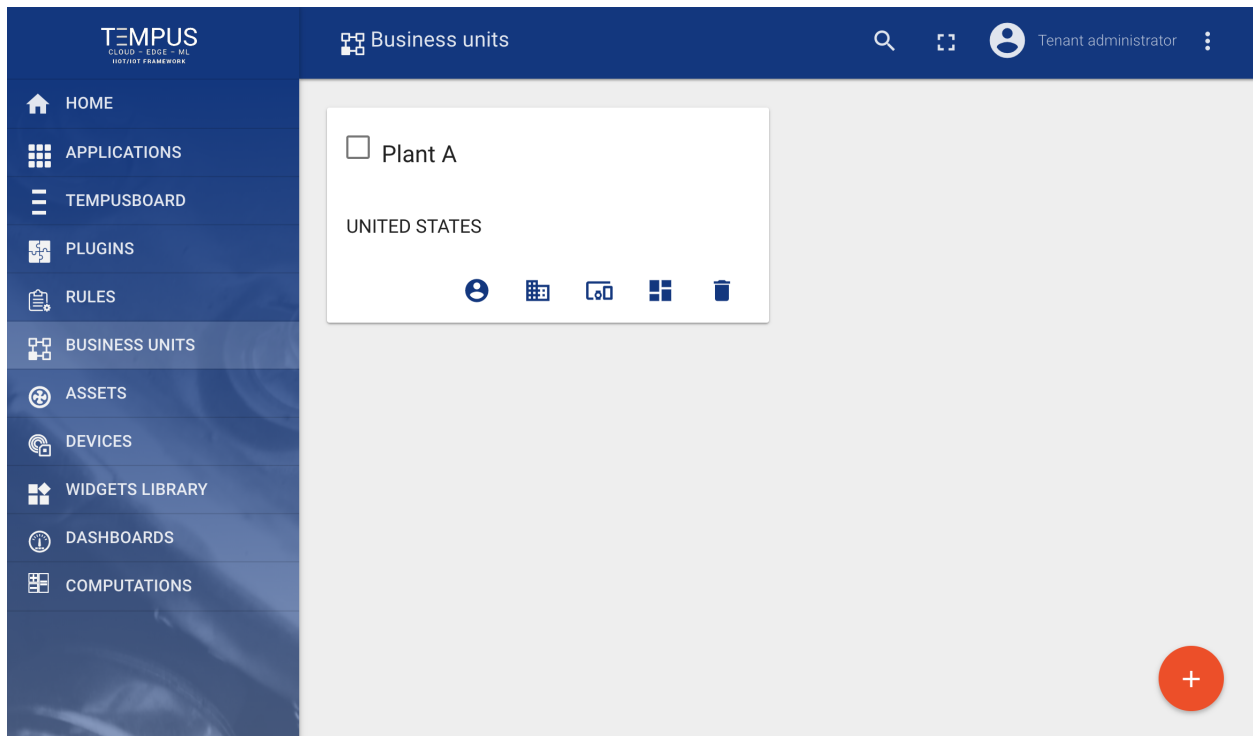
All actions listed above are available using *Swagger*

Business Units

Tempus supports following asset management features using Web UI.

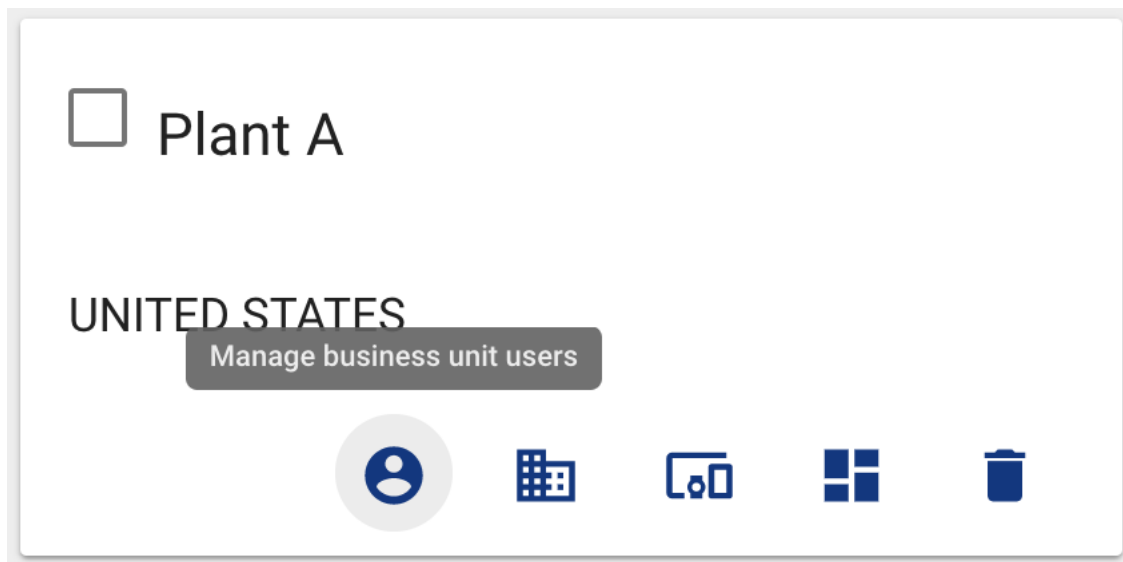
Add and delete business units

Tenant administrator is able create new business unit or delete them from Tempus.



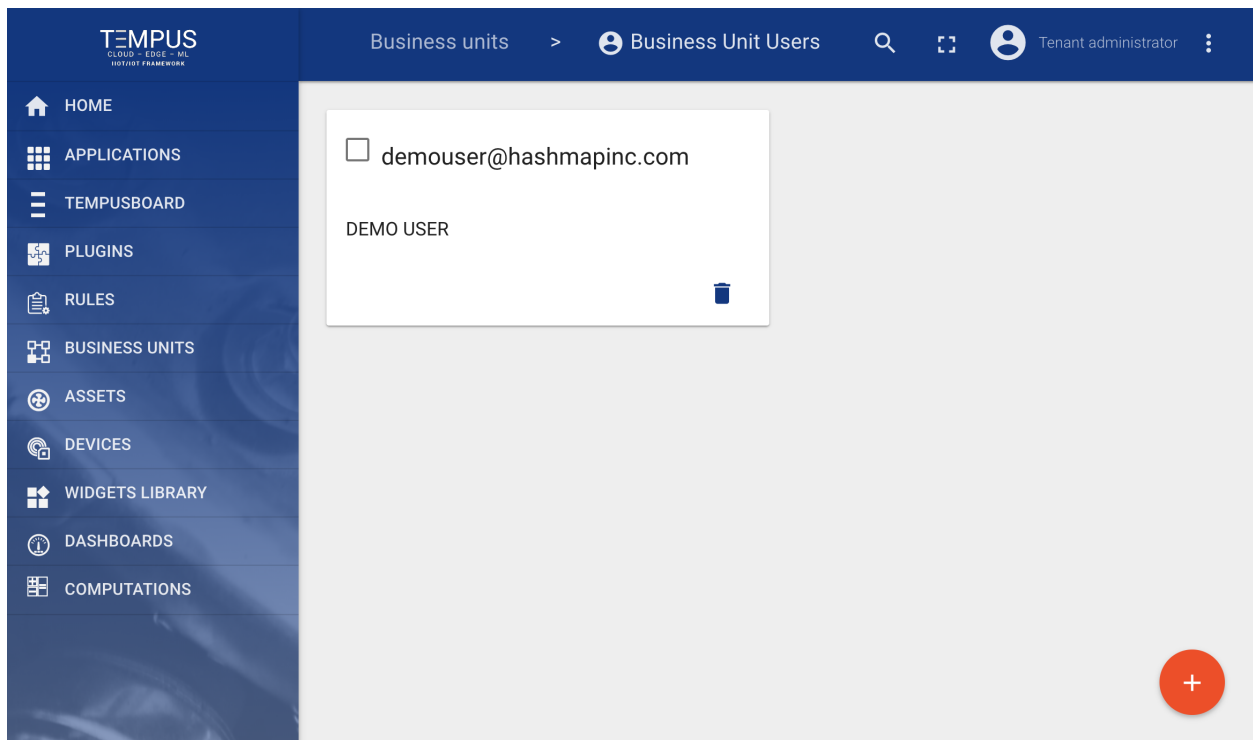
Manage business unit users

Tenant administrator is able to manage business unit users.



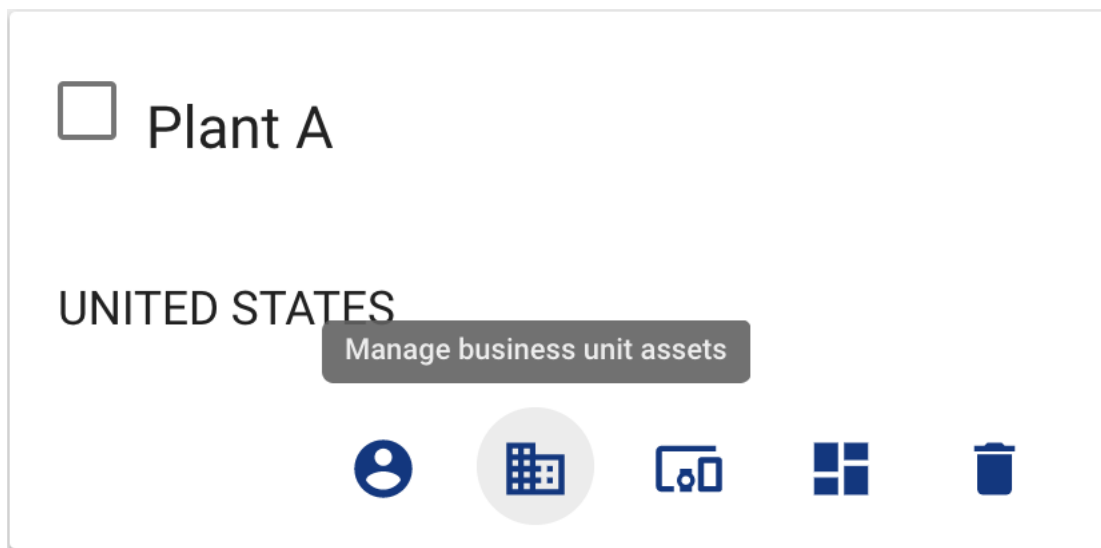
Add users to business unit

Tenant administrator is able add Users to a business unit.



Manage business unit assets

Tenant administrator is able to manage business unit assets.



Assign assets to business unit

Tenant administrator is able assign assets to a business unit.

Assign Asset(s) To business unit



Please select the assets to assign to the business unit

 Enter search

☐ Tool 1

☐ Tool 2

ASSIGN

CANCEL

Manage business unit devices

Tenant administrator is able to manage business units devices.

 Plant A

UNITED STATES

Manage business unit devices



Assign devices to business unit

Tenant administrator is able assign assets to a business unit.

Assign Device(s) To Business Unit



Please select the devices to assign to the business unit



☐

Tank 123

☐

Tank 456

☐

Demo123

ASSIGN

CANCEL

Manage business unit dashboards

Tenant administrator is able to manage business unit dashboards.

☐ Plant A

UNITED STATES

Manage business unit dashboards



Assign dashboards to business unit

Tenant administrator is able assign assets to a business unit.

Assign Device(s) To Business Unit



Please select the devices to assign to the business unit



☐

Tank 123

☐

Tank 456

☐

Demo123

ASSIGN

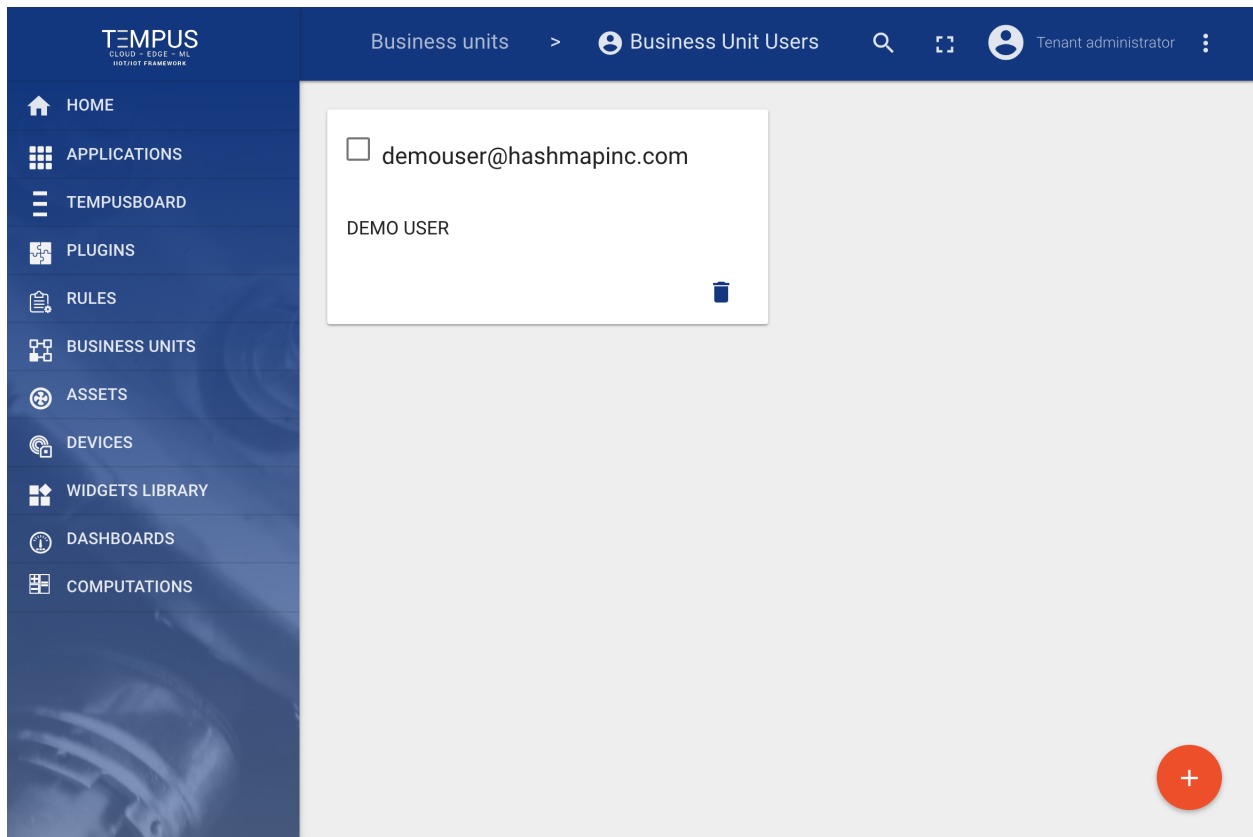
CANCEL

Users

Tempus supports following user management features using Web UI.

Add and delete users

Tenant administrator is able add and delete users to business unit.



User activation links

Tenant administrator is display or resend a user thier activation link

DEMOUSER@HASHMAPINC.COM

User details



DISPLAY ACTIVATION LINK

RESEND ACTIVATION

DELETE USER

Email *

demouser@hashmapinc.com



First Name

Demo



Last Name

User



Description

Default dashboard

Select dashboard



Always fullscreen

Display activation link

Activation link is provided for a user to activate an account and create a password.

User activation link



In order to activate user use the following [activation link](#) :

```
http://ec2-13-59-114-28.us-east-2.compute.amazonaws.com:8080/api/noauth/a
```




OK

Create user password


The user activation link prompts the user to create a password.

Create Password

Password



Password again



CREATE PASSWORD

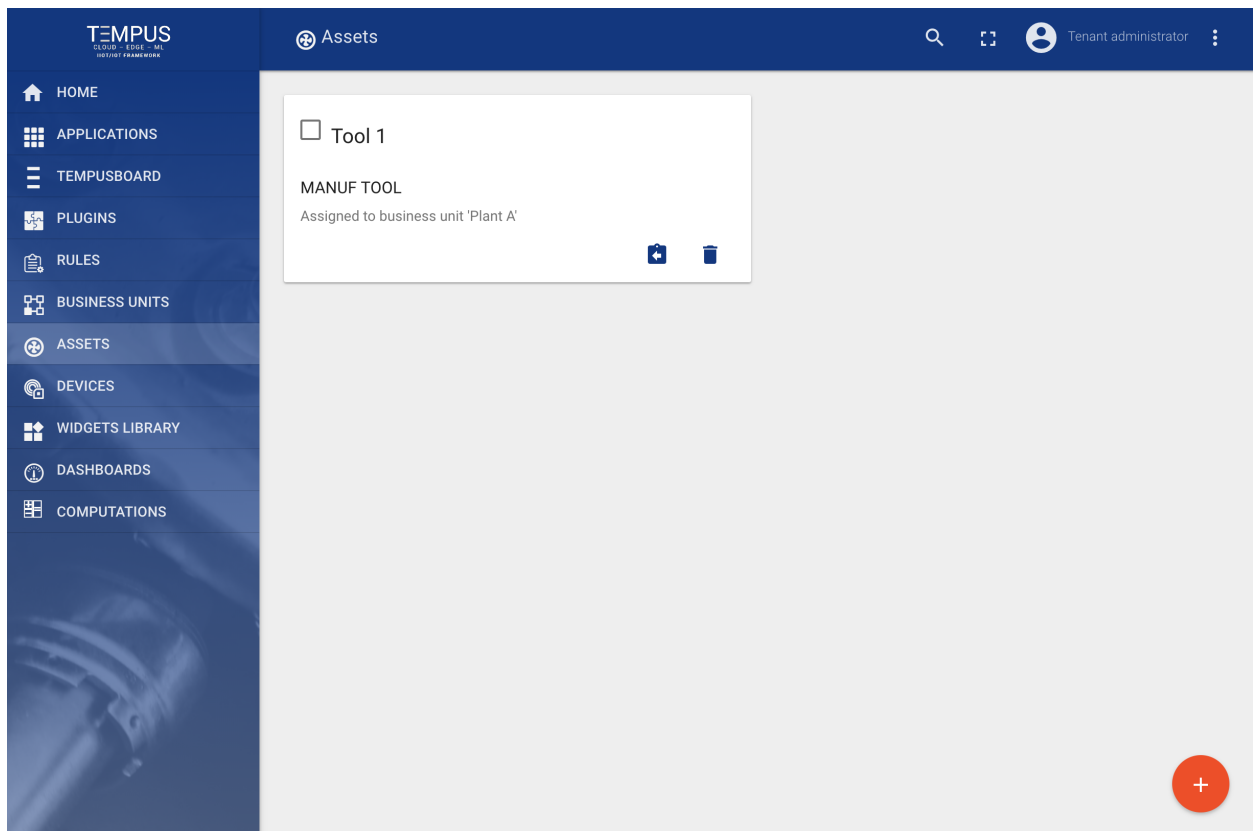
CANCEL

Assets

Tempus supports following asset management features using Web UI and *Swagger*.

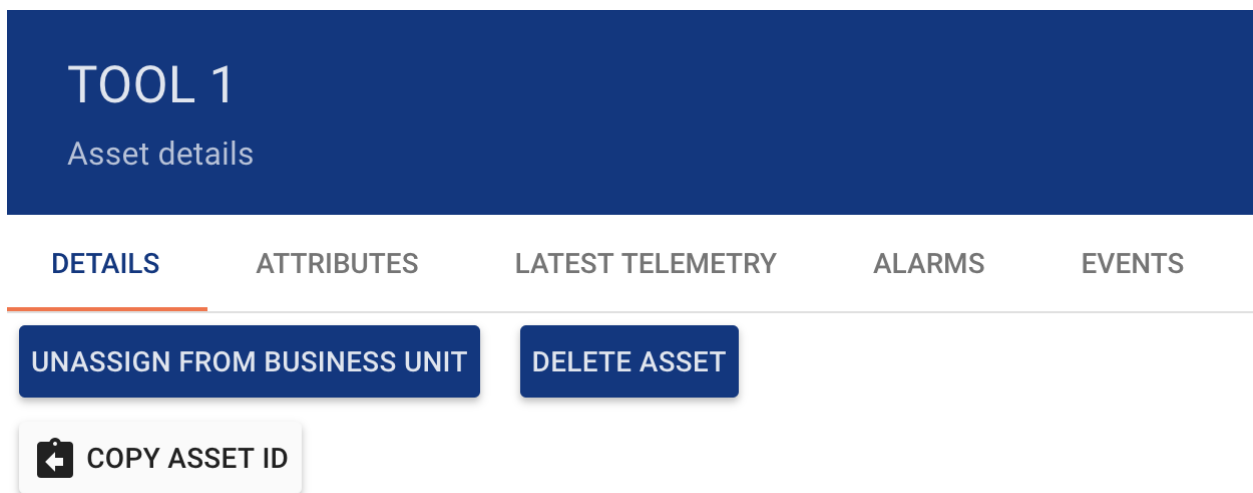
Add and delete assets

Tenant administrator is able to register new assets or delete them from Tempus.



Get Asset Id

Tenant administrator and business unit users are able to copy asset id to clipboard using “Copy Asset Id” button.



Assign assets to business units

Tenant administrator is able to assign assets to certain business unit. This will allow Customer users to fetch asset data using REST APIs or Web UI.

Assign Asset(s) To business unit



Please select the business unit to assign the asset(s)

☐

Plant A

ASSIGN

CANCEL

Manage asset attributes

Tenant administrator and business unit users are able to manage asset server-side attributes.

TOOL 1

Asset details

?

×

DETAILS

ATTRIBUTES

LATEST TELEMETRY

ALARMS

EVENTS

RELATIONS

Entity attributes scope

Server attributes

Server attributes

+

🔍

↻

<input type="checkbox"/>	Last update time	Key	↑	Value	
<input type="checkbox"/>	2018-03-13 14:58:52	firmware		2.3	<div></div>

Page: 1 Rows per page: 5 1 - 1 of 1 < >

Browse asset alarms

Tenant administrator and business unit users are able to browse asset alarms.

DEVICE C

Device details

?

×

<

DETAILS

ATTRIBUTES

LATEST TELEMETRY

LATEST DEPTH

ALARMS

>

Alarm status

Any

⌚

LAST 30 DAYS

Created time	Originator	Type	Severity	Status	Details
2018-03-05 11:56:46	Device C	Firmware Alarm	Critical	Cleared Acknowledged	...
2018-03-05 11:52:29	Device C	Firmware Alarm	Critical	Cleared Acknowledged	...
2018-03-02 09:52:05	Device C	Firmware Alarm	Critical	Cleared Acknowledged	...
2018-03-01 19:10:27	Device C	Firmware Alarm	Critical	Cleared Acknowledged	...
2018-03-01 18:46:26	Device C	Firmware Alarm	Critical	Cleared Acknowledged	...
2018-03-01 18:41:27	Device C	Firmware Alarm	Critical	Cleared Acknowledged	...
2018-03-01 18:38:26	Device C	Firmware Alarm	Critical	Cleared Acknowledged	...
2018-03-01 18:04:31	Device C	Firmware Alarm	Critical	Cleared Acknowledged	...

Browse asset events

Tenant administrator and business unit users are able to browse events related to particular asset using “Events” tab. Lifecycle events and statistics are coming soon.

Manage asset relations

Tenant administrator and business unit users are able to manage asset relations.

TOOL 1

Asset details

?

×

DETAILS

ATTRIBUTES

LATEST TELEMETRY

ALARMS

EVENTS

RELATIONS

Direction

From

Outbound relations

+

🔍

↻

<input type="checkbox"/> Type ↑	To entity type	To entity name	
<input type="checkbox"/> Contains	Business Unit	Plant A	<div><div>✎</div><div>🗑</div></div>

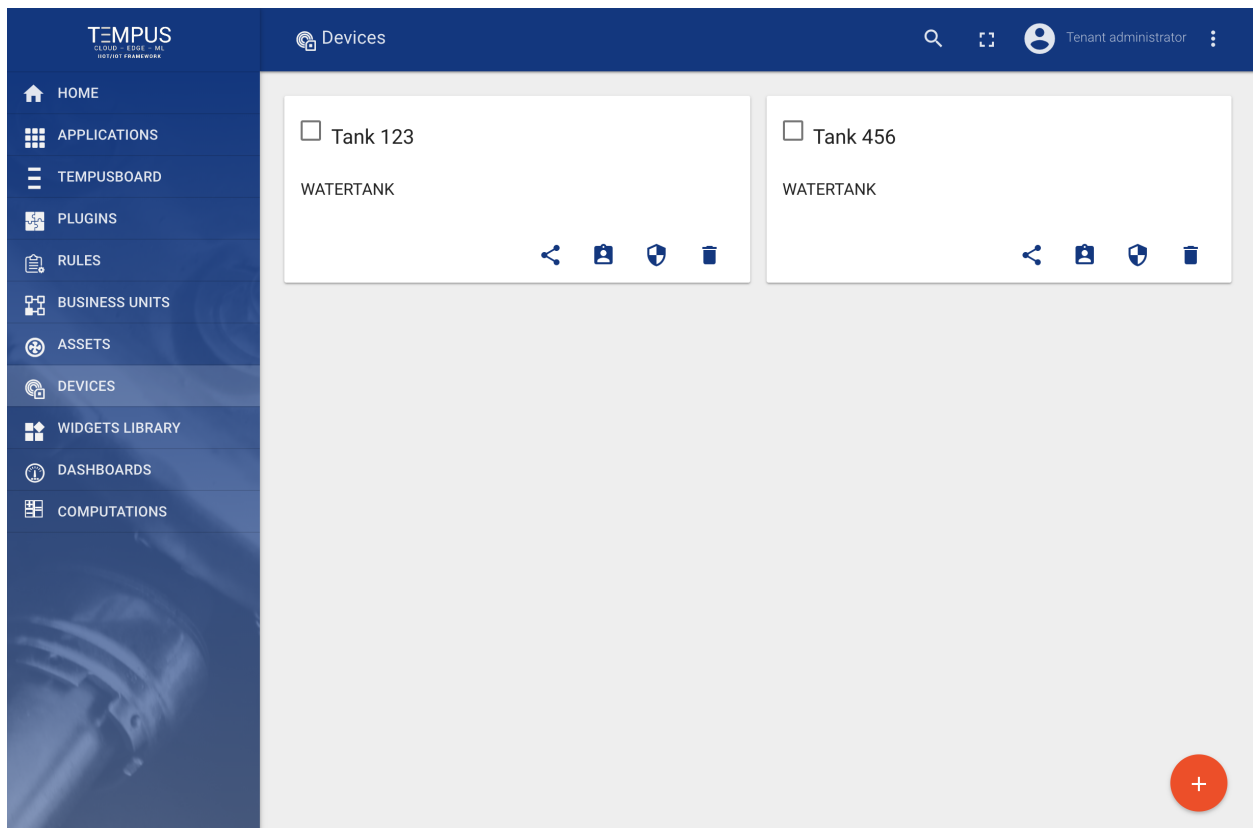
Page: 1 ▾ Rows per page: 5 ▾ 1 - 1 of 1 < >

Devices

Tempus supports following device management features using Web UI and *Swagger*.

Add and delete devices

Tenant administrator is able to register new devices or delete them from Tempus.



Manage device credentials

Tenant administrator is able to manage device credentials. Current release supports Access Token and X.509 Certificates based credentials.

Device Credentials

✕

Credentials type

Access token

Access token *

SECRET_TOKEN

12 / 20

SAVE

CANCEL

Get Device Id

Tenant administrator and business unit users are able to copy device id to the clipboard using “Copy Device Id” button.

TANK 123

Device details

DETAILS

ATTRIBUTES

LATEST TELEMETRY


ALARMS


EVEN

MAKE DEVICE PUBLIC

ASSIGN TO BUSINESS UNIT

MANAGE CREDENTIALS

 COPY DEVICE ID

 COPY ACCESS TOKEN

Assign devices to business units

Tenant administrator is able to assign devices to certain business unit. This will allow Customer users to fetch device data using REST APIs or Web UI.

Assign Device(s) To Business Unit



Please select the business unit to assign the device(s)

☐

Plant A

ASSIGN

CANCEL

Browse device attributes

Tenant administrator and business unit users are able to browse device attributes.

TANK 123

Device details

?

×

DETAILS

ATTRIBUTES

LATEST TELEMETRY

ALARMS

EVENTS

RELATIONS

Client attributes

Server attributes

Shared attributes

		Key	Value
	2018-03-13 16:24:30	deviceType	WaterTank
	2018-03-13 16:24:30	tankId	123

Page:

1

Rows per page:

5

1 - 2 of 2

<

>

Browse device telemetry

Tenant administrator and business unit users are able to browse device telemetry data.

TANK 123

Device details

?

×

DETAILS

ATTRIBUTES

LATEST TELEMETRY

ALARMS

EVENTS

RELATIONS

Latest telemetry

Q

<input type="checkbox"/>	Last update time	Key ↑	Value
<input type="checkbox"/>	2018-03-13 16:28:23	Attn	-0.0241897442711536
<input type="checkbox"/>	2018-03-13 16:28:23	waterTankLevel	14.363773148148148

Page:

1 ▼

Rows per page:

5 ▼

1 - 2 of 2

<

>

Browse device alarms

Tenant administrator and business unit users are able to browse device alarms.

TANK 123

Device details

?

×

DETAILS

ATTRIBUTES

LATEST TELEMETRY

ALARMS

EVENTS

RELATIONS

Alarm status

Any

⌚ LAST 30 DAYS

Created time	Originator	Type	Severity	Status	Details
NO ALARMS FOUND					

Browse device events

Tenant administrator and business unit users are able to browse events related to a particular device using “Events” tab. Lifecycle events and statistics are coming soon.

Manage device relations

Tenant administrator and business unit users are able to manage device relations.

TANK 123

Device details

?

×

DETAILS

ATTRIBUTES

LATEST TELEMTRY

ALARMS

EVENTS

RELATIONS

Direction

From

Outbound relations

+

🔍

↻

<input type="checkbox"/> Type ↑	To entity type	To entity name	
<input type="checkbox"/> Contains	Business Unit	Plant A	<div>✎</div> <div>🗑</div>

Page:

1

Rows per page:

5

1 - 1 of 1

<

>

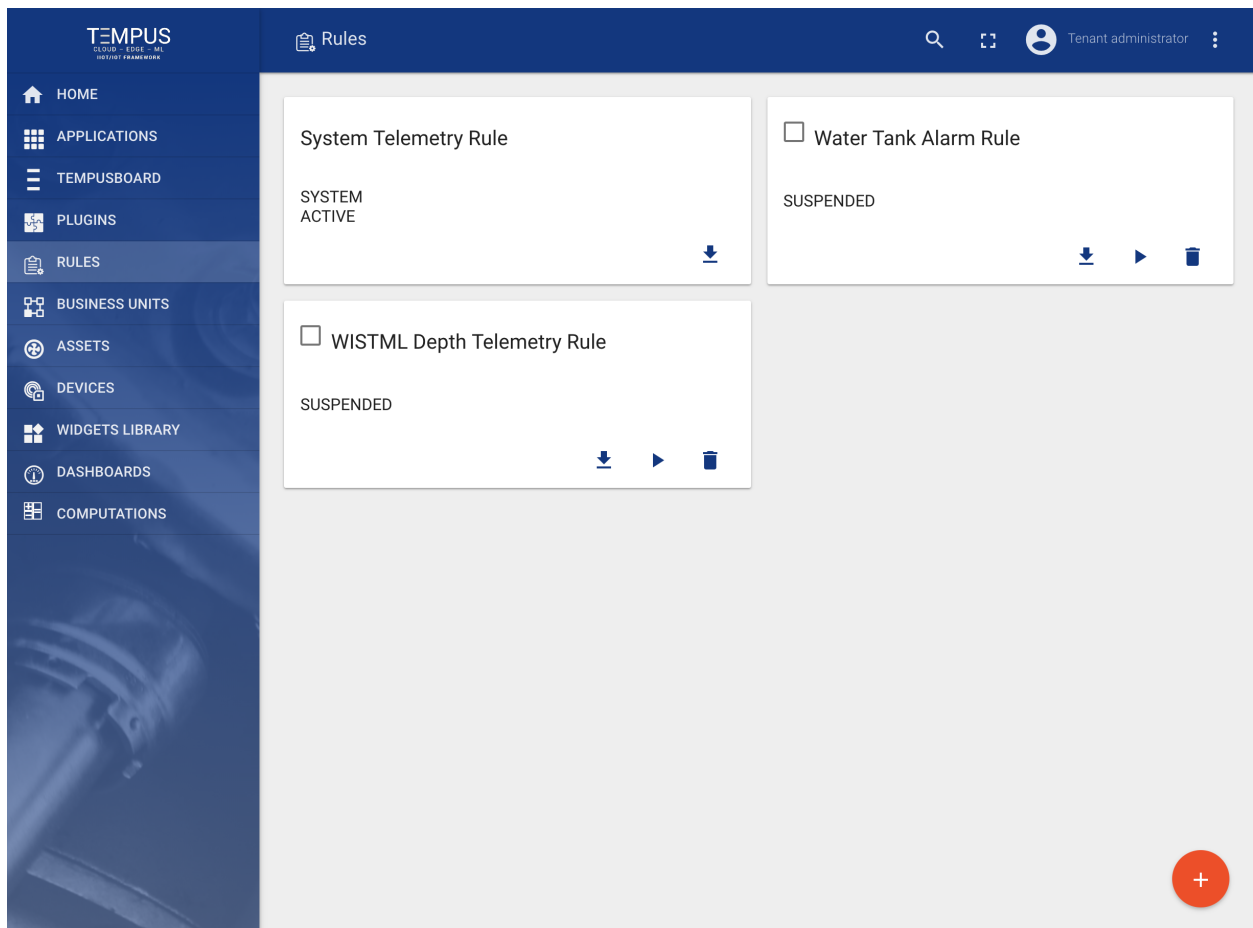
Rules

Rules page

Rules Administration UI page displays a table of system and tenant specific rules. Each rule has a separate card. You are able to do following operations:

- Import Or Create new Rule
- Export Rule to JSON
- Suspend and Activate particular Rule
- Delete the Rule

See *Rule Engine* documentation for more details.



Rules details

Each rule is represented as a separate card. You are able to edit rule components and review the rule events in the Rule details panel.

WATER TANK ALARM RULE

Rule details

?

×

DETAILS

ATTRIBUTES

LATEST TELEMETRY

ALARMS

EVENTS

RELATIONS

ACTIVATE RULE

EXPORT RULE

DELETE RULE

COPY RULE ID

Name*

Water Tank Alarm Rule

Description

Filters

	Filter name	Filter type	
1.	Message Filter	Message Type Filter	...
2.	Attribute Filter	Device Attributes Filter	...

Processor

Processor name	Processor type	
Below 2.0 Alarm	Alarm Processor	...

Plugin

Kafka Plugin

Plugin action

Action name	Action type	
Create Workorder	Kafka Plugin Action	...

You are also able to review rule life-cycle events, rule stats, and errors during message processing. Please note that in case of frequent errors the error messages are sampled.

109

WATER TANK ALARM RULE

Rule details

?

×

DETAILS

ATTRIBUTES

LATEST TELEMETRY

ALARMS

EVENTS

RELATIONS

Event type

Lifecycle event

LAST DAY

Event time	Server	Event	Status	Error
2018-03-13 16:42:38	[localhost:9001]	UPDATED	Success	
2018-03-13 16:41:52	[localhost:9001]	SUSPENDED	Success	

Rules import/export

Rule export

You are able to export your rule to JSON format and import it to the same or another Tempus instance. In order to export rule, you should navigate to the **Rules** page and click on the export button located on the particular rule card.

☐

Water Tank Alarm Rule

SUSPENDED

Export rule

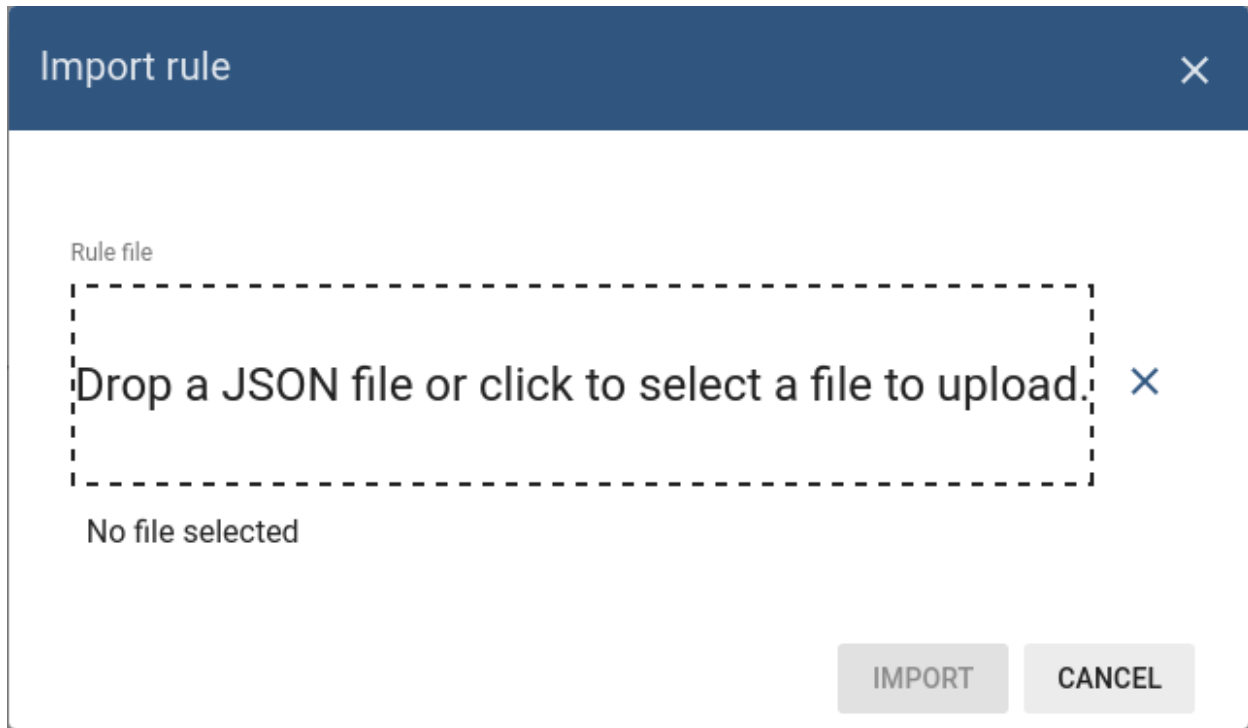
↓

▶

🗑

Rule import

Similar, to import the rule you should navigate to the **Rules** page and click on the big “+” button in the bottom-right part of the screen and then click on the import button.



Note All imported rules are in the suspended state. Don’t forget to **activate** your rule after import.

Troubleshooting

Possible issues while importing the rule:

- The corresponding target plugin is not imported.
- The corresponding target plugin is not activated.

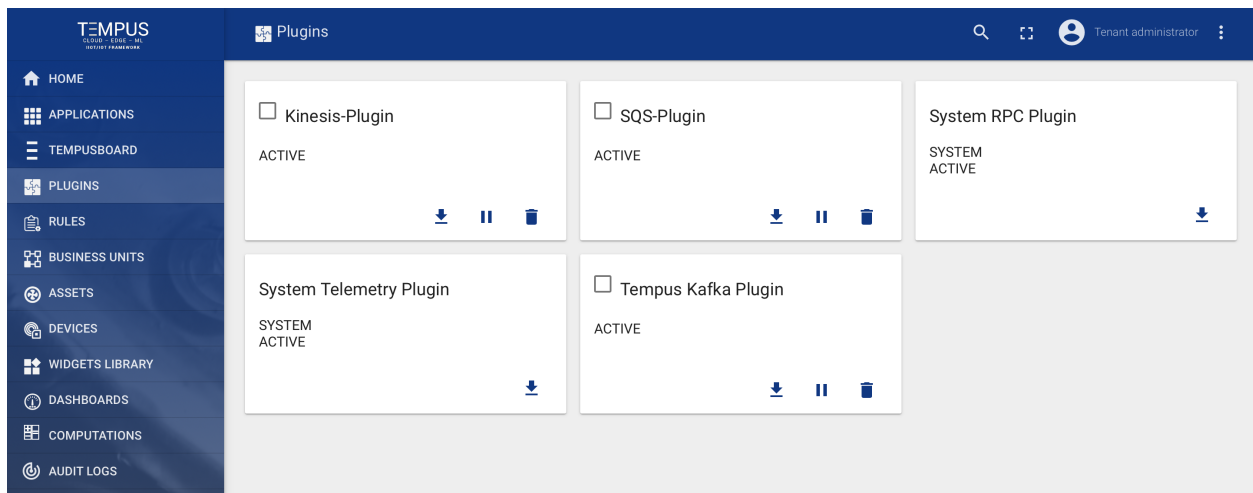
Plugins

Plugins page

Plugins Administration UI page displays a table of system and tenant specific plugins. Each plugin has a separate card. You are able to do following operations:

- Import Or Create new Plugin
- Export Plugin to JSON
- Suspend and Activate particular Plugin
- Delete the Plugin

See *Rule Engine* documentation for more details.



Plugin details

Each plugin is represented as a separate card. You are able to edit plugin configuration and review the plugin events in the Plugin details panel.

KAFKA PLUGIN FOR SPARK STREAMING

Plugin details

?

✕

DETAILS

ATTRIBUTES

LATEST TELEMETRY

ALARMS

EVENTS

RELATIONS

ACTIVATE PLUGIN

EXPORT PLUGIN

DELETE PLUGIN

COPY PLUGIN ID

Name *

Kafka Plugin for Spark Streaming

Description

API token *

1234

Plugin type *

Kafka Plugin

Plugin configuration

Bootstrap Servers *

localhost:9092

Automatically Retry Times If Fails

Producer Batch Size On Client

16384

You are also able to review plugin life-cycle events, stats, and errors during message processing. Please note that in case of frequent errors the error messages are sampled.

KAFKA PLUGIN FOR SPARK STREAMING

Plugin details

?

×

DETAILS

ATTRIBUTES

LATEST TELEMETRY

ALARMS

EVENTS

RELATIONS

Event type

Lifecycle event

LAST DAY

Event time	Server	Event	Status	Error
2018-03-13 16:57:37	[localhost:9001]	CREATED	Success	
2018-03-13 16:57:37	[localhost:9001]	STARTED	Success	

Plugin import/export

Plugin export

You are able to export your plugin to JSON format and import it to the same or another Tempus instance. In order to export plugin, you should navigate to the Plugins page and click on the export button located on the particular plugin card.

☐

Kafka Plugin for Spark Streaming

SUSPENDED

Export plugin

↓

▶

Plugin import

Similar, to import the plugin you should navigate to the Plugins page and click on the big “+” button in the bottom-right part of the screen and then click on the import button.

Import plugin



Plugin file

Drop a file or click to select a file to upload.



No file selected

IMPORT

CANCEL

SQS Plugins

SQS-PLUGIN

Plugin details

?

×

<

DETAILS

ATTRIBUTES

LATEST TELEMETRY

ALARMS

EVENTS

RELATIONS >

Name *

SQS-Plugin

Description

Plugin that is able to send messages to SQS Queues

API token *

sqs-plugin

Plugin type *

SQS Plugin

Plugin configuration

Access Key ID *

Secret Access Key *

Region *

us-east-1

You are also able to review plugin life-cycle events, stats, and errors during message processing. Please note that in case of frequent errors the error messages are sampled.

SQS-PLUGIN

Plugin details

?

×

<

DETAILS

ATTRIBUTES

LATEST TELEMETRY

ALARMS

EVENTS

RELATIONS

>

Event type

Lifecycle event

LAST DAY

↺

Event time	Server	Event	Status	Error
2018-05-10 21:47:03	[localhost:9001]	STARTED	Success	
2018-05-10 09:55:38	[localhost:9001]	STARTED	Success	

Plugin import/export

Plugin export

You are able to export your plugin to JSON format and import it to the same or another Tempus instance. In order to export plugin, you should navigate to the Plugins page and click on the export button located on the particular plugin card.

TEMPUS

CLOUD - EDGE - ML

IIOT/IIOT FRAMEWORK

HOME

APPLICATIONS

TEMPUSBOARD

PLUGINS

RULES

Plugins

□

SQS-Plugin

ACTIVE

Export plugin

↓

||

🗑

Plugin import

Similar, to import the plugin you should navigate to the Plugins page and click on the big “+” button in the bottom-right part of the screen and then click on the import button.

Import plugin

Plugin file

Drop a file or click to select a file to upload.

No file selected

IMPORT

CANCEL

Kinesis Plugins

KINESIS-PLUGIN

Plugin details

? ×

< DETAILS ATTRIBUTES LATEST TELEMETRY ALARMS **EVENTS** RELATIONS >

Event type
Lifecycle event ▼

🕒 LAST DAY ↺

Event time	Server	Event	Status	Error
2018-05-16 16:26:02	[localhost:9001]	ACTIVATED	Success	
2018-05-16 16:23:09	[localhost:9001]	CREATED	Success	
2018-05-16 16:23:09	[localhost:9001]	STARTED	Success	

Plugin import/export

Plugin export

You are able to export your plugin to JSON format and import it to the same or another Tempus instance. In order to export plugin, you should navigate to the Plugins page and click on the export button located on the particular plugin card.

☐ Kinesis-Plugin

ACTIVE

Export plugin

↓ || 🗑️

Plugin import

Similar, to import the plugin you should navigate to the Plugins page and click on the big “+” button in the bottom-right part of the screen and then click on the import button.

Import plugin

Plugin file

Drop a file or click to select a file to upload.

No file selected

IMPORT

CANCEL

Note All imported plugins are in the suspended state. Don't forget to **activate** your plugin after import.

Troubleshooting

Possible issues while importing the plugin:

- The corresponding plugin API Token is **already reserved**. You can choose another token and edit it directly in the source json.
- The corresponding plugin implementation is **not available** in the server classpath.

Widget Library

Introduction

All IoT *Dashboards* are constructed using Tempus widgets that are defined in Widget Library. Each widget provides end-user functions such as data visualization, remote device control, alarms management and displaying static custom html content.

Widget Types

According to the provided features, each widget definition represents specific widget type. At the moment there are five widget types:

- *Latest values*
- *Time-series*
- *RPC (Control widget)*
- *Alarm Widget*
- *Static*

Each widget type has own specific datasource configuration and corresponding widget API. Each widget requires datasource for data visualization. Types of the available datasource depend on widget type of the widget:

- Target device - this datasource type is used in RPC. Basically, you need to specify target device for RPC widget
- Alarm source - this datasource type is used in Alarm widgets. This datasource requires source entity to display related alarms and corresponding alarm fields.
- Entity - this datasource type is used in both time-series and latest values widgets. Basically, you need to specify target entity and timeseries key or attribute name.
- Function - this datasource type is used in both time-series and latest values widgets for debug purposes. Basically, you are able to specify a javascript function that will emulate data from a device in order to tune visualization.

Latest values

Displays latest values of particular entity attribute or timeseries data point (for ex. any Gauge Widget or Entities Table widget). This kind of widgets uses values of entity attribute(s) or timeseries as datasource.

DATA

SETTINGS

ADVANCED

ACTIONS

Datasources

Maximum 1 datasource is allowed.

Type

Parameters

1.

Entity

Smart

×

energy: energy

✎

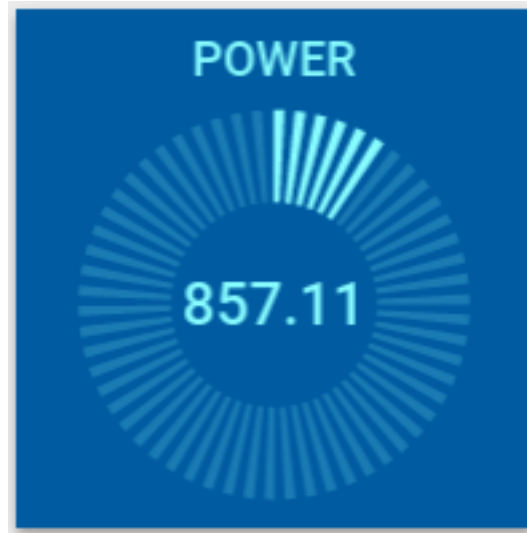
×

Timeseries

Attributes

Maximum 1 timeseries/attribute is allowed.

Below is an example of latest values widget - Digital Gauge displaying current power value.



Time-series

Displays historical values for the selected time period or latest values in the certain time window (for ex. “Timeseries - Flot” or “Timeseries table”). This kind of widgets uses only values of entity timeseries as datasource. In order to specify the time frame of displayed values, Timewindow settings are used. Timewindow can be specified on the dashboard level or on the widget level. It can be either realtime - dynamically changed time frame for some latest interval, or history - fixed historical time frame. All these settings are part of Time-series widget configuration.

DATA

SETTINGS

ADVANCED

ACTIONS

☒ Use dashboard timewindow

Timewindow

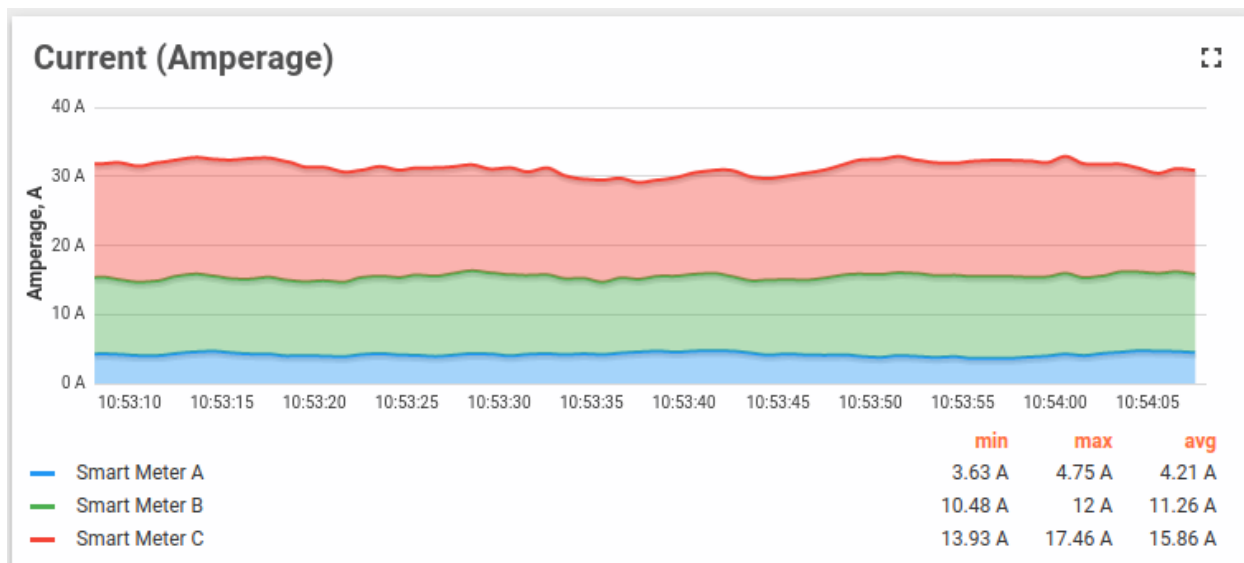
⌚ REALTIME - LAST MINUTE

Datasources

Type	Parameters
1. Entity	<div><div>Smart</div><div><div>amperage: amperage</div><div>voltage: voltage</div></div></div> <div>Timeseries</div>

+ ADD

Below is an example of time series widget - “Timeseries - Flot” displaying amperage values of three devices in real-time.



RPC (Control widget)

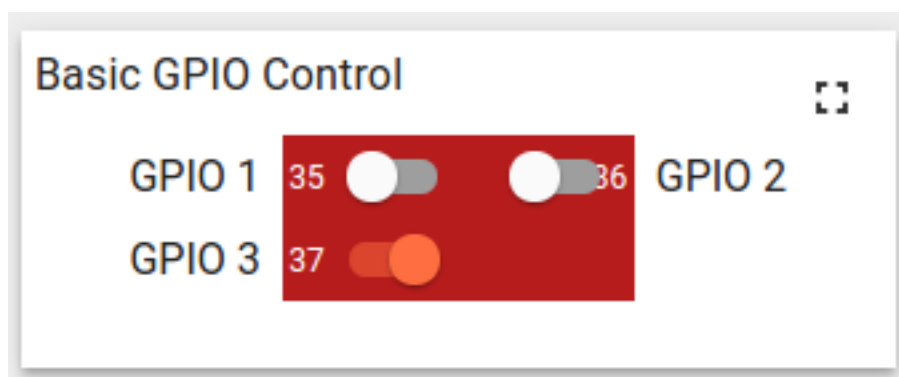
Allows to send RPC commands to devices and handles/visualize reply from the device (for ex. “Raspberry Pi GPIO Control”). RPC widgets are configured by specifying target device as target endpoint for RPC commands.

DATA
SETTINGS
ADVANCED
ACTIONS

Target device

state

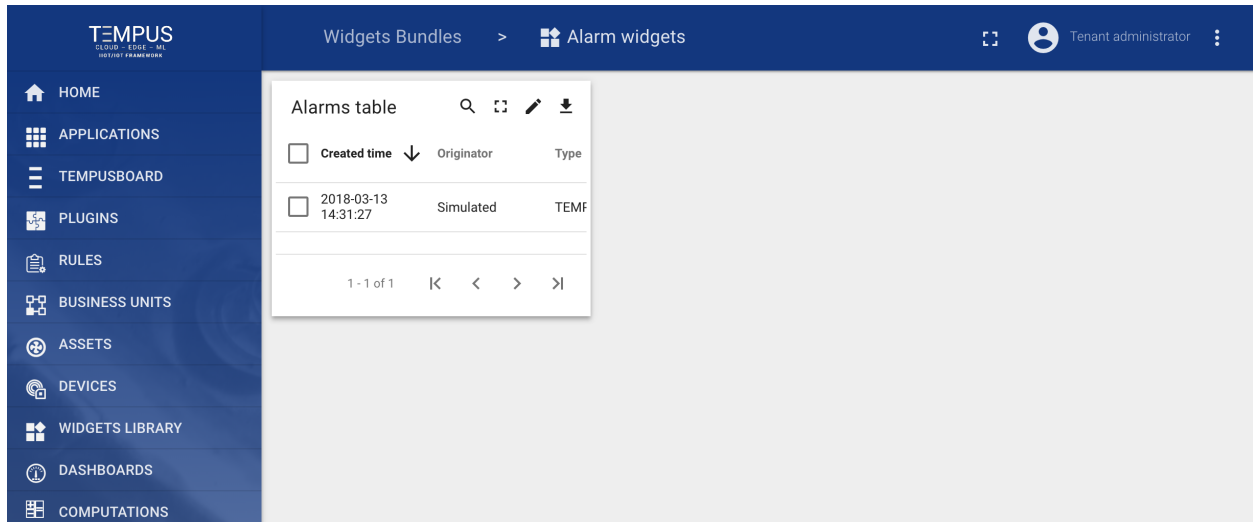
Below is an example of RPC widget - “Basic GPIO Control” - sending GPIO switch commands and detecting current GPIOs switch status.



Alarm Widget

Display alarms related to the specified entity in the certain time window (for ex. “Alarms table”). Alarm widgets are configured by specifying entity as alarms source and corresponding alarm fields. Like Time-series widgets alarm widgets have the timewindow configuration in order to specify the time frame of displayed alarms. Additionally

configuration contains “Alarm status” and “Alarms polling interval” parameters. “Alarm status” parameter specifies the status of alarms being fetched. “Alarms polling interval” controls alarms fetching frequency in seconds.



The screenshot shows the TEMPUS Cloud Edge AI Platform interface. The left sidebar contains navigation links: HOME, APPLICATIONS, TEMPUSBOARD, PLUGINS, RULES, BUSINESS UNITS, ASSETS, DEVICES, WIDGETS LIBRARY, DASHBOARDS, and COMPUTATIONS. The main header shows 'Widgets Bundles' and 'Alarm widgets'. The 'Alarms table' widget is displayed, showing a table with columns: Created time, Originator, and Type. The table contains one row of data: 2018-03-13 14:31:27, Simulated, and TEMP. The widget also includes a search icon, a refresh icon, and a download icon.

Below is an example of Alarm widget - “Alarms table” displaying latest alarms for the asset in real-time.

Alarms: Silo A						
<input type="checkbox"/> Created time ↑	Originator	Type	Severity	Status		
<input type="checkbox"/> 2017-06-12 19:25:46	Silo A	TEMPERATURE	Major	Cleared Acknowledged	...	
<input type="checkbox"/> 2017-06-12 20:01:44	Silo A	HUMIDITY	Major	Cleared Acknowledged	...	
<input type="checkbox"/> 2017-06-14 16:26:12	Silo A	HUMIDITY	Major	Cleared Acknowledged	...	
<input type="checkbox"/> 2017-06-14 17:40:21	Silo A	HUMIDITY	Major	Cleared Acknowledged	...	
<input type="checkbox"/> 2017-06-14 19:22:37	Silo A	HUMIDITY	Major	Cleared Acknowledged	...	
Page: 1 Rows per page: 10 1 - 7 of 7						

Static

Displays static customizable html content (for ex. “HTML card”). Static widgets don’t use any datasources and usually configured by specifying static html content and optionally css styles.

CSS

```
1 .card {  
2   font-weight: bold;  
3   font-size: 32px;  
4   color: #999;  
5   width: 100%;  
6   height: 100%;  
7   display: flex;  
8   align-items: center;  
9   justify-content: center;  
10 }
```

CSS

TIDY

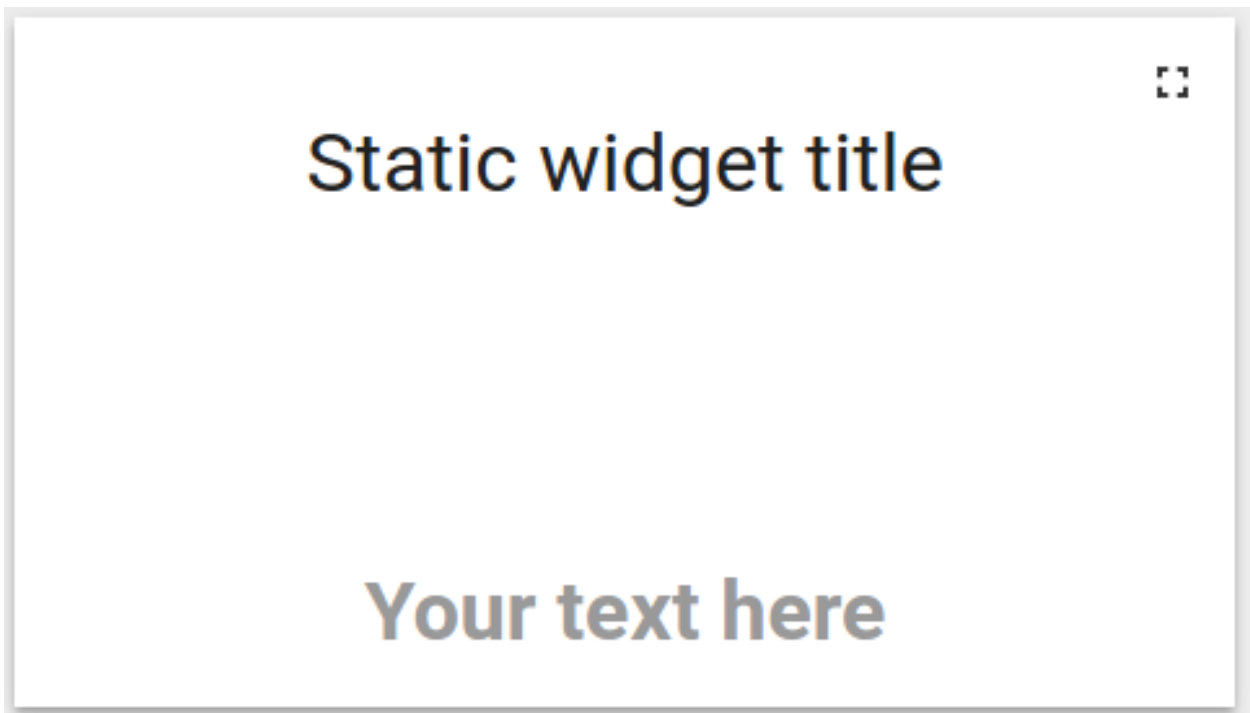
HTML *

```
i 1 <h1>Static widget title</h1>  
2 <div class='card'>Your text here</div>
```

HTML

TIDY

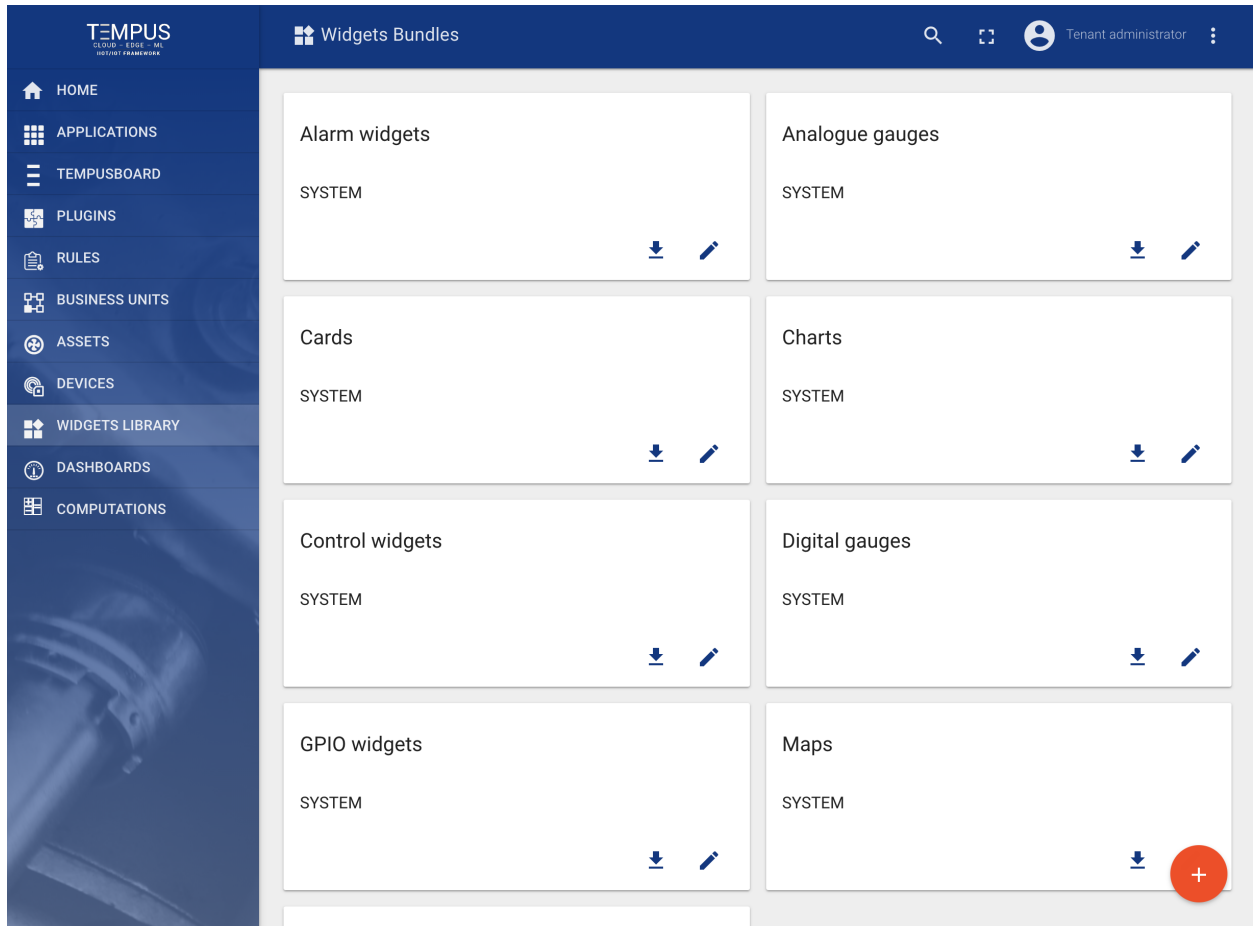
Below is an example of a Static widget - “HTML card” displaying specified html content.



Widgets Library (Bundles)

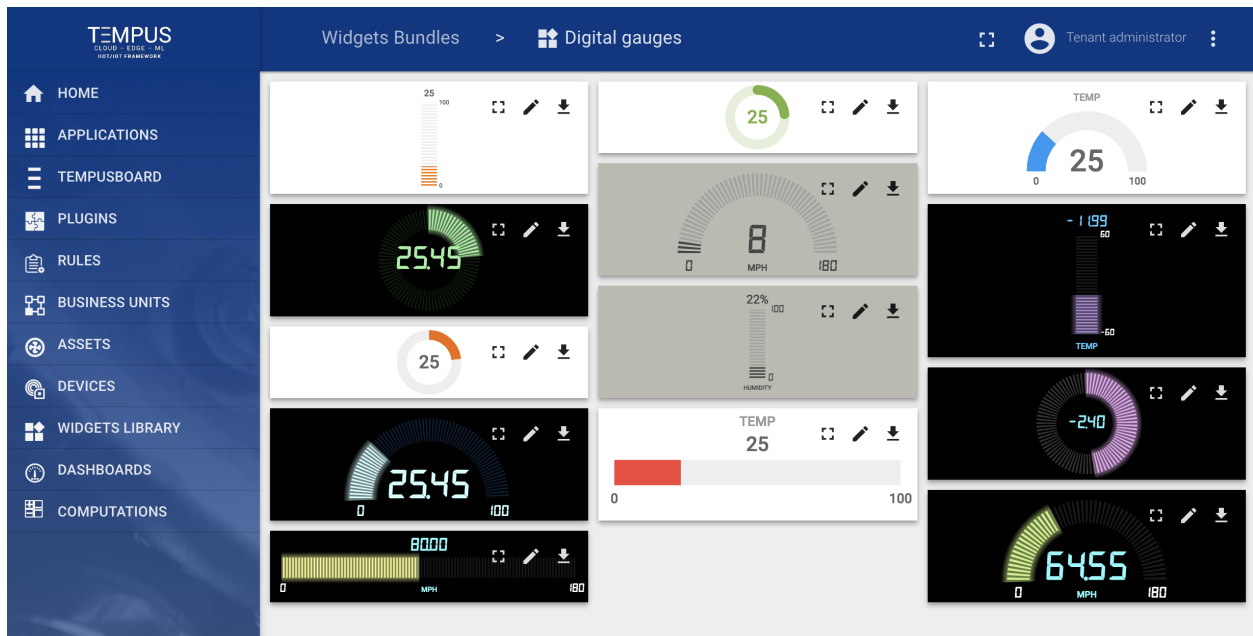
Widget definitions are grouped into widget bundles according to their purpose. There are System level and Tenant level **Widgets Bundles**. Initial Tempus installation is shipped with the basic set of system level **Widgets Bundles**. There are more than thirty widgets in seven widget bundles available out-of-the-box. System level bundles can be

managed by a **System administrator** and are available for use by any tenant in the system. Tenant level bundles can be managed by a **Tenant administrator** and are available for use only by this tenant and its business units. You can always implement and add your widgets by following this guide.



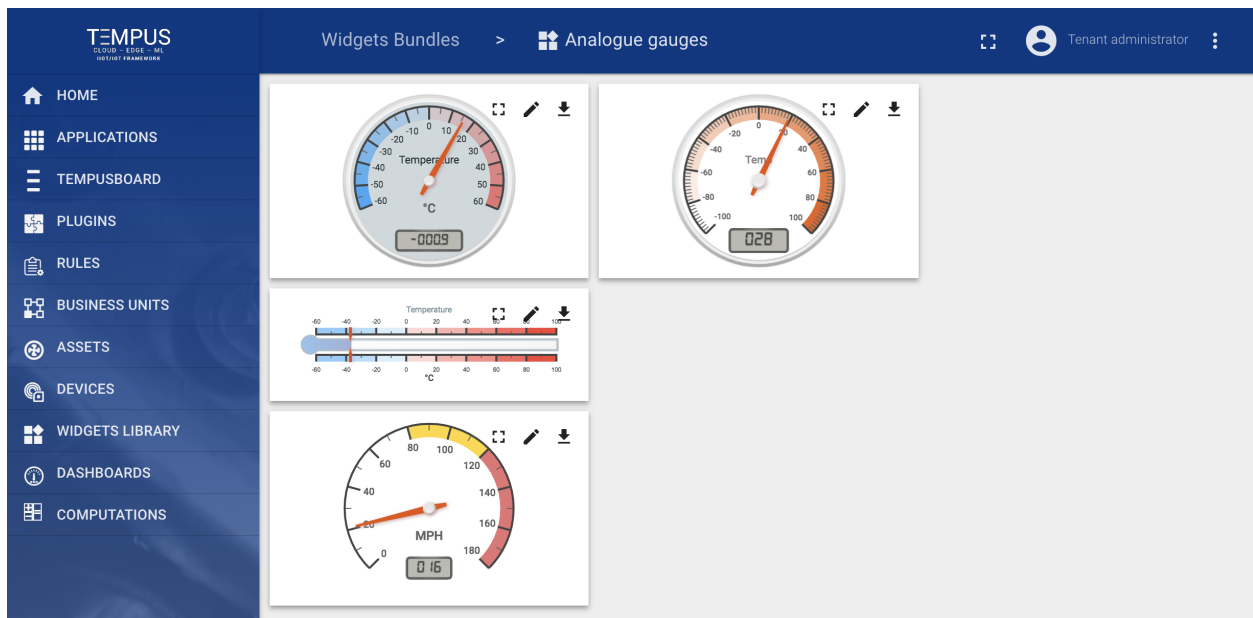
Digital Gauges

Useful for visualization of temperature, humidity, speed and other integer or float values.



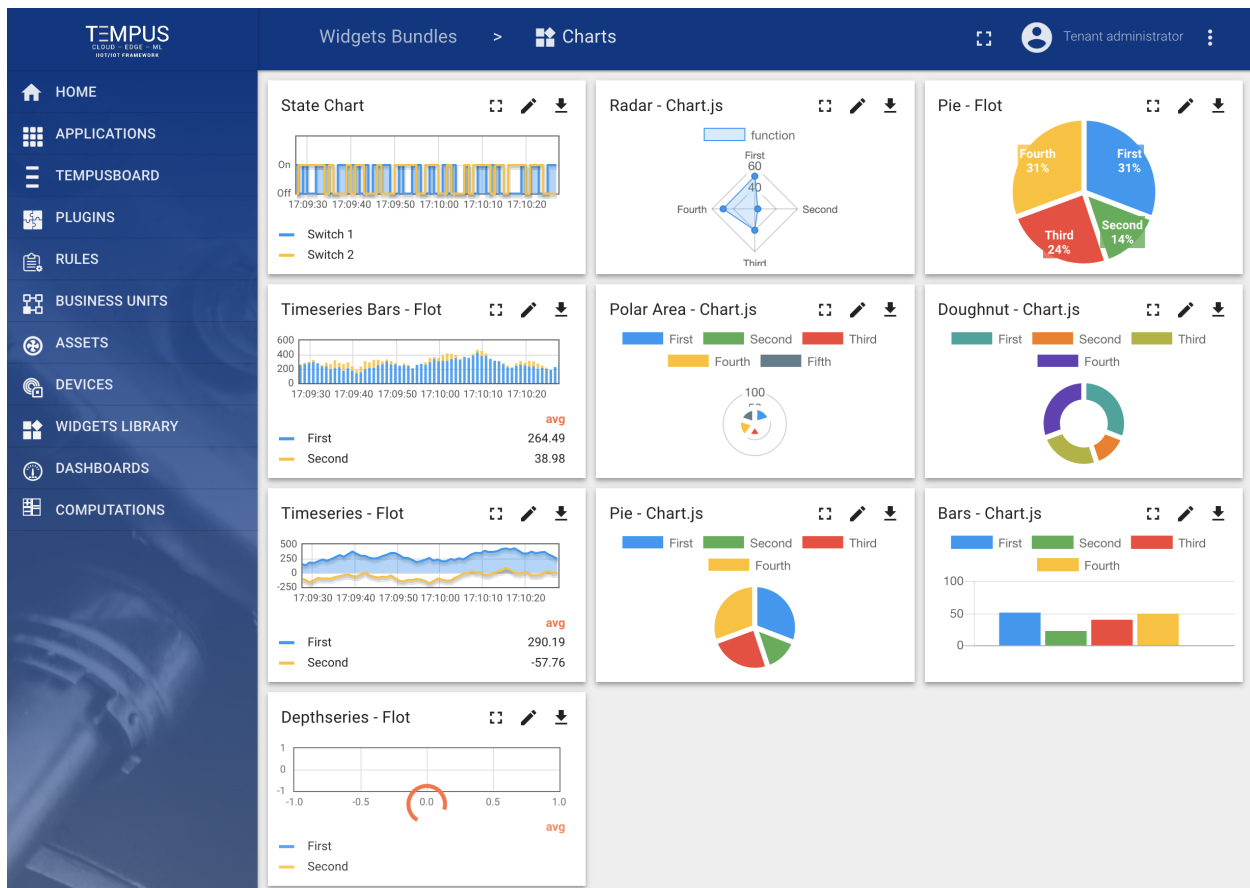
Analog Gauges

Similar to digital gauges, but have a different style.



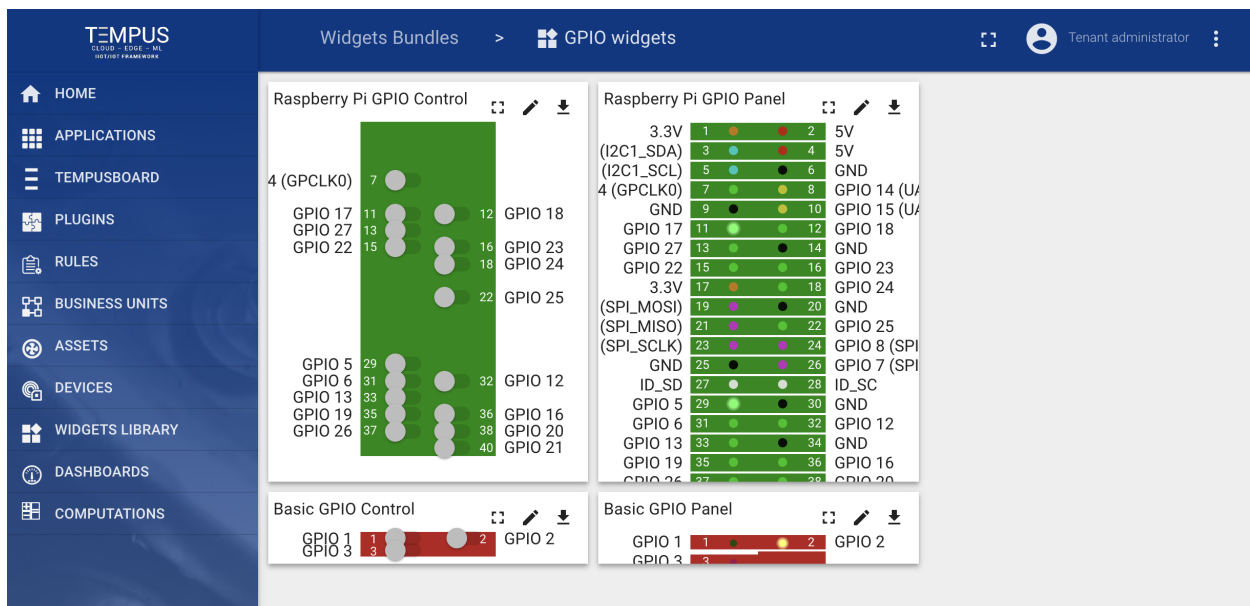
Charts

Useful for visualization of historical or real-time data with a time window.



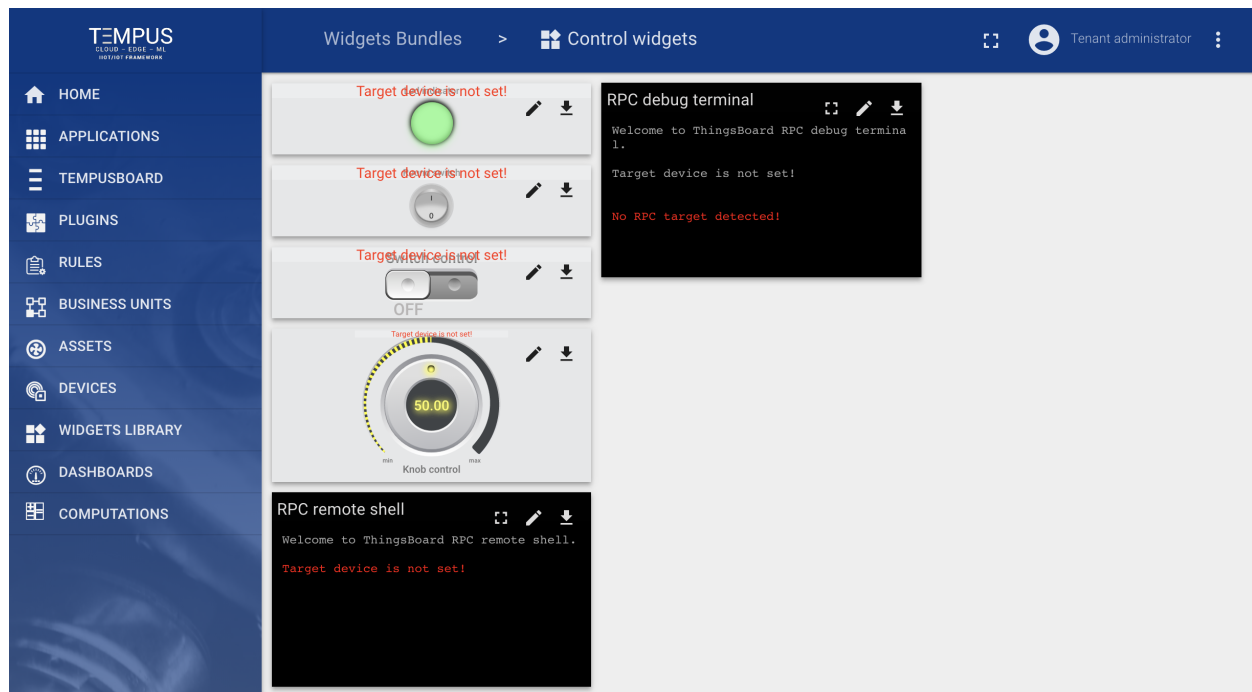
GPIO widgets

Useful for visualization and control of GPIO state for target devices.



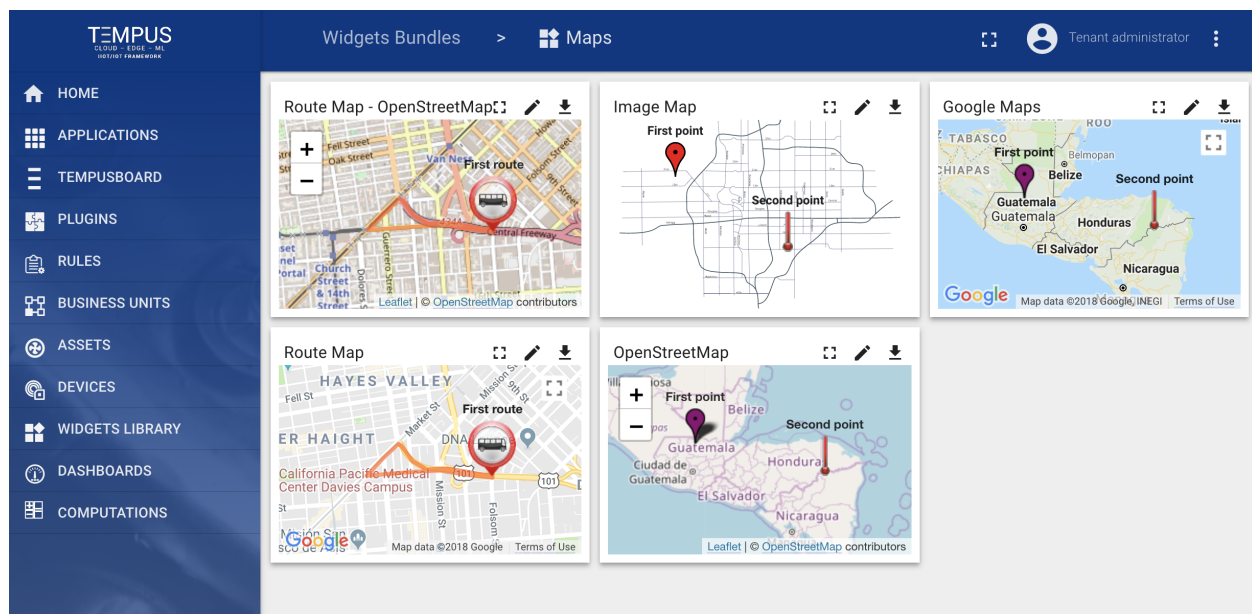
Control widgets

Useful for visualize current state and send RPC commands to target devices.



Maps widgets

Useful for visualization of devices geo locations and track devices routes both in real-time and history mode.



Cards

Useful for visualization of timeseries data or attributes in a table or card widget.

The screenshot shows the Tempus dashboard interface. The left sidebar contains navigation links: HOME, APPLICATIONS, TEMPUSBOARD, PLUGINS, RULES, BUSINESS UNITS, ASSETS, DEVICES, WIDGETS LIBRARY, DASHBOARDS, and COMPUTATIONS. The main area is titled 'Widgets Bundles > Cards'. It displays several widget bundles:

- Timeseries table**: A table with columns 'Timestamp' and 'Temperature °C'.

Timestamp	Temperature °C
2018-03-13 17:13:15	-26.6
2018-03-13 17:13:14	-14
2018-03-13 17:13:13	-3
2018-03-13 17:13:12	-19.4
- Label widget**: A card with a placeholder for an image background and a value of -33.96 units.
- Attributes card**: A card with a function 'Random' and a value of -33.96.
- HTML code here**: A card with a placeholder for HTML code.
- VALUE TITLE**: A card displaying '1.41 units.' with a value description text.
- Entities table**: A table with columns 'Entity name', 'Entity type', and 'Sin'. It shows 1 - 1 of 1 results.

Alarm widgets

Useful for visualization of alarms for specific entities both in real-time and history mode.

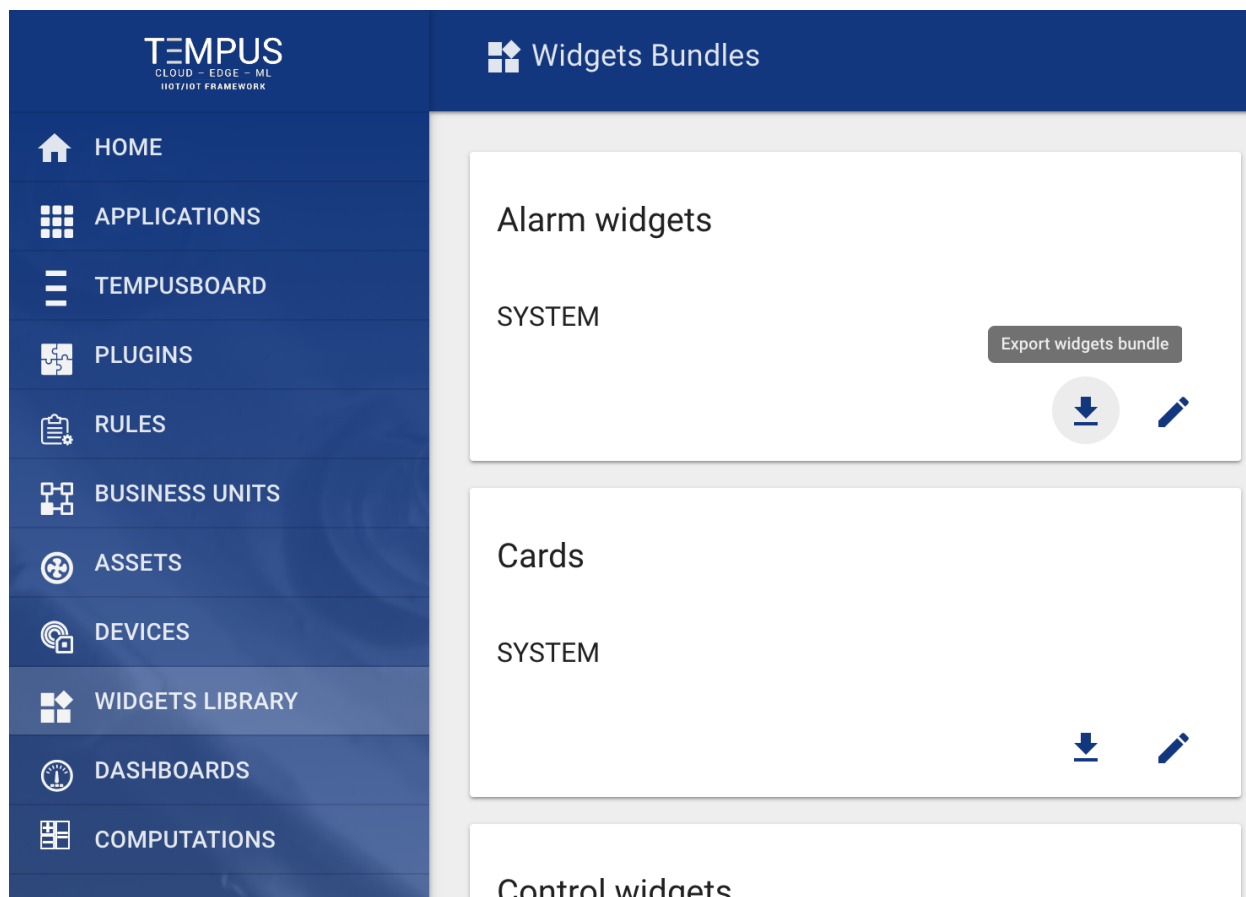
The screenshot shows the Tempus dashboard interface. The left sidebar contains navigation links: HOME, APPLICATIONS, TEMPUSBOARD, PLUGINS, RULES, BUSINESS UNITS, ASSETS, DEVICES, WIDGETS LIBRARY, DASHBOARDS, and COMPUTATIONS. The main area is titled 'Widgets Bundles > Alarm widgets'. It displays an 'Alarms table' widget:

<input type="checkbox"/> Created time	Originator	Type
<input type="checkbox"/> 2018-03-13 14:31:27	Simulated	TEMP

Widgets Bundles import/export

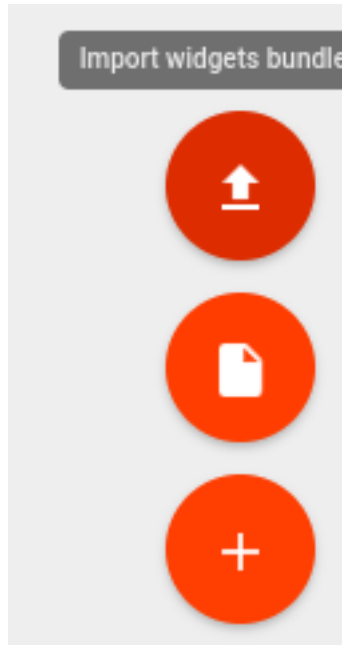
Widgets Bundle export

You are able to export widgets bundle to JSON format and import it to the same or another Tempus instance. In order to export widgets bundle, you should navigate to the Widgets Library page and click on the export button located on the particular widgets bundle card.

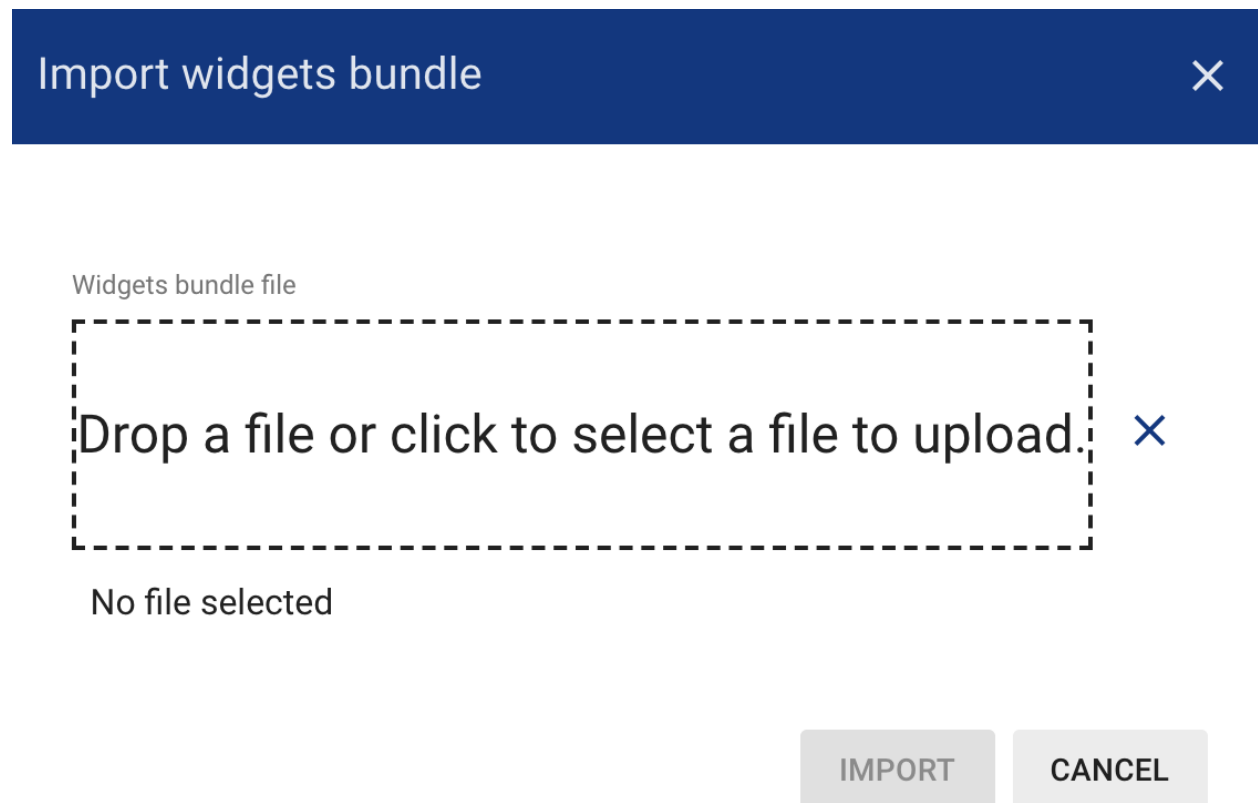


Widgets Bundle import

Similar, to import the widgets bundle you should navigate to the Widgets Library page and click on the big “+” button in the bottom-right part of the screen and then click on the import button.



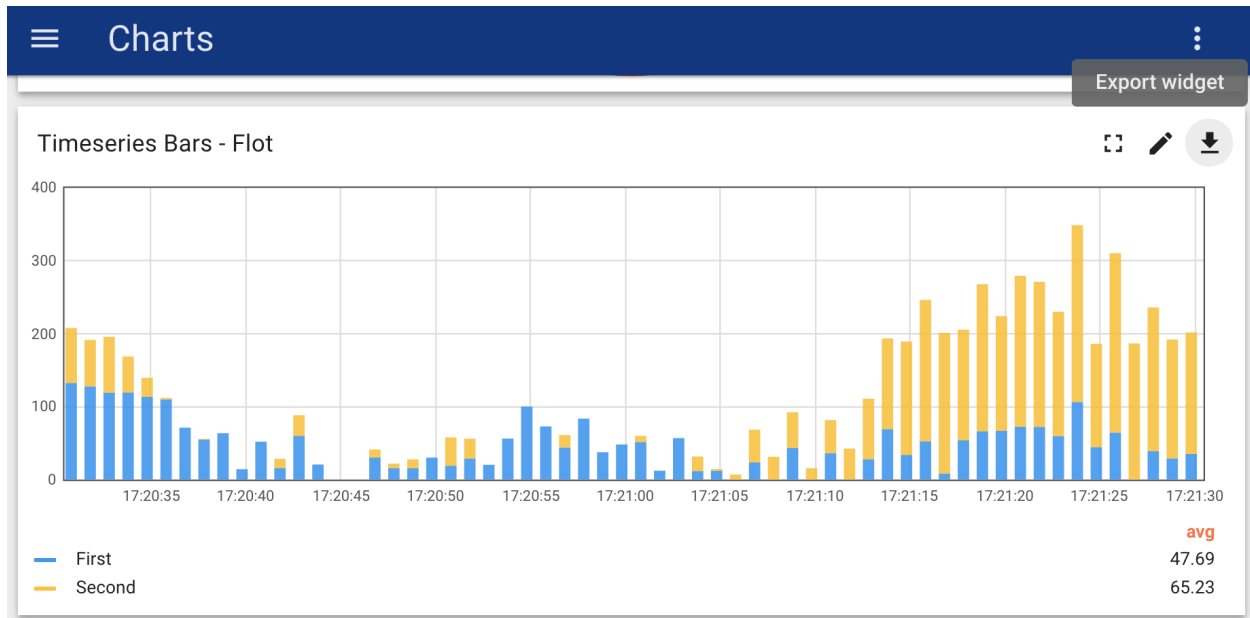
The widgets bundle import window should a popup and you will be prompted to upload the json file.



Widgets Types import/export

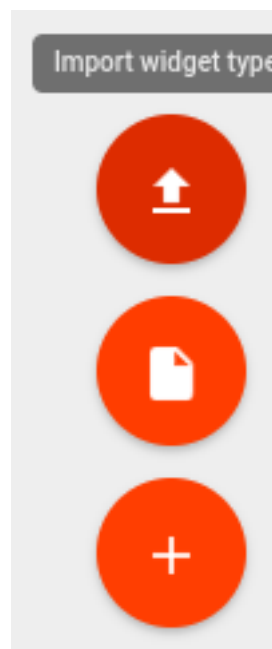
Widget Type export

You are able to export particular widget type from widgets bundle to JSON format and import it to the same or another Tempus instance. In order to export widget type, you should navigate to the Widgets Library page, then open desired widgets bundle and finally click on the export button located on the particular widget type card.



Widget Type import

Similar, to import the widget type you should navigate to the Widgets Library page, then open your widgets bundle and click on the big “+” button in the bottom-right part of the screen and then click on the import button.



The widget type import window will show a popup and you will be prompted to upload the json file.

Import widgets bundle



Widgets bundle file

Drop a file or click to select a file to upload.



No file selected

IMPORT

CANCEL

Dashboards

Default dashboard for business unit user

Tempus allow you to define default IoT dashboard for your business unit users in 2 simple steps:

Step 1. Assign dashboard to business unit

See embedded video tutorial above on tips how to do this.

Step 2. Open business unit user details

Navigate to “**Business Unit** -> Your business unit -> **Buiness Unit Users**” and toggle edit mode using ‘pencil’ button in the top-right corner of the screen.

Step 3. Select dashboard

Select the IoT dashboard from the list and apply changes. Please note that you can also check the “Always Fullscreen” mode to prevent a user from navigating to different dashboards/screens.

CUSTOMER@THINGSBOARD.ORG

User details



Email *

customer@thingsboard.org

First Name

Last Name

Description

Default dashboard

Smart energy monitoring

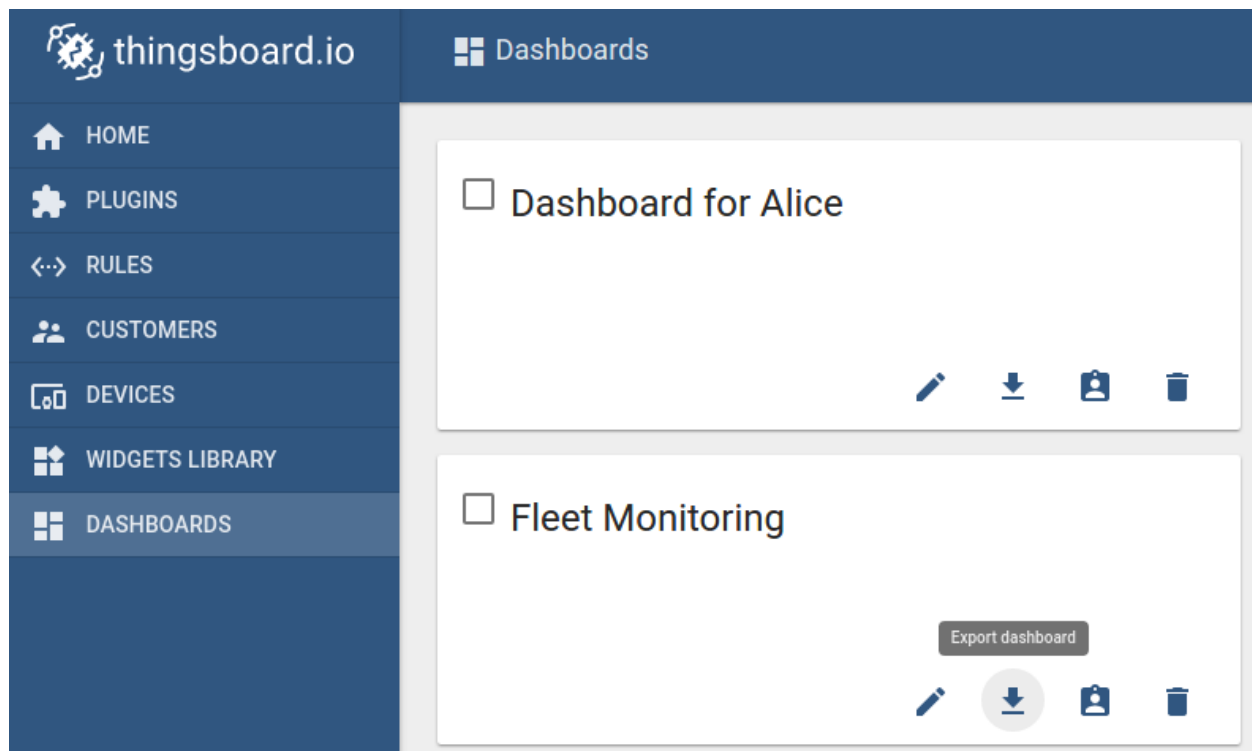


Always fullscreen

IoT Dashboard import/export

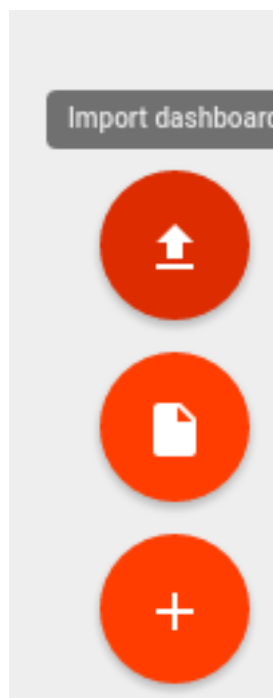
Dashboard export

You are able to export your dashboard to JSON format and import it to the same or another Tempus instance. In order to export dashboard, you should navigate to the Dashboards page and click on the export button located on the particular dashboard card.



Dashboard import

Similar, to import the dashboard you should navigate to the Dashboards page and click on the big “+” button in the bottom-right part of the screen and then click on the import button.



The dashboard import window should popup and you will be prompted to upload the json file.

Import widgets bundle



Widgets bundle file

Drop a file or click to select a file to upload. 

No file selected



IMPORT

CANCEL

Once you click on the “import” button you will need to specify the device aliases. This basically allows you to set what device(s) correspond to dashboard alias.

Configure aliases used by imported dashboard



	Alias	Devices	
1.	Arduino UNO Demo Device	Device list	<div>Use filter  </div>

SAVE

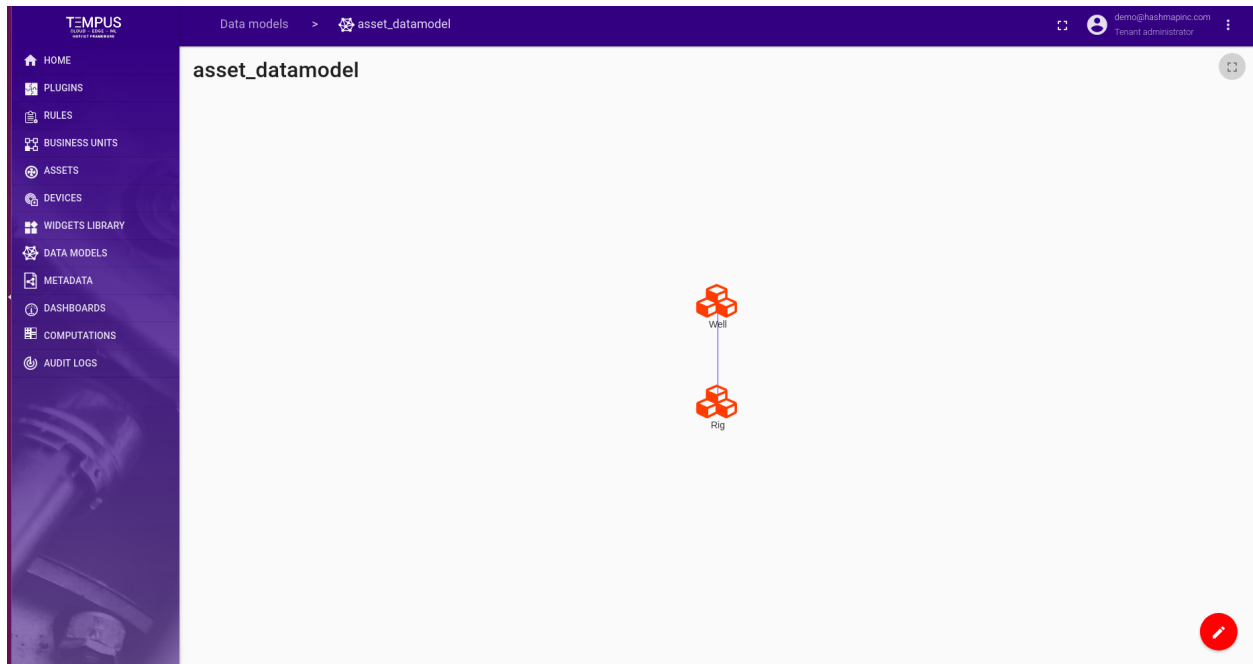
CANCEL

Asset Landing Dashboard

Tempus supports following asset landing dashboard features using Web UI and *Swagger*.

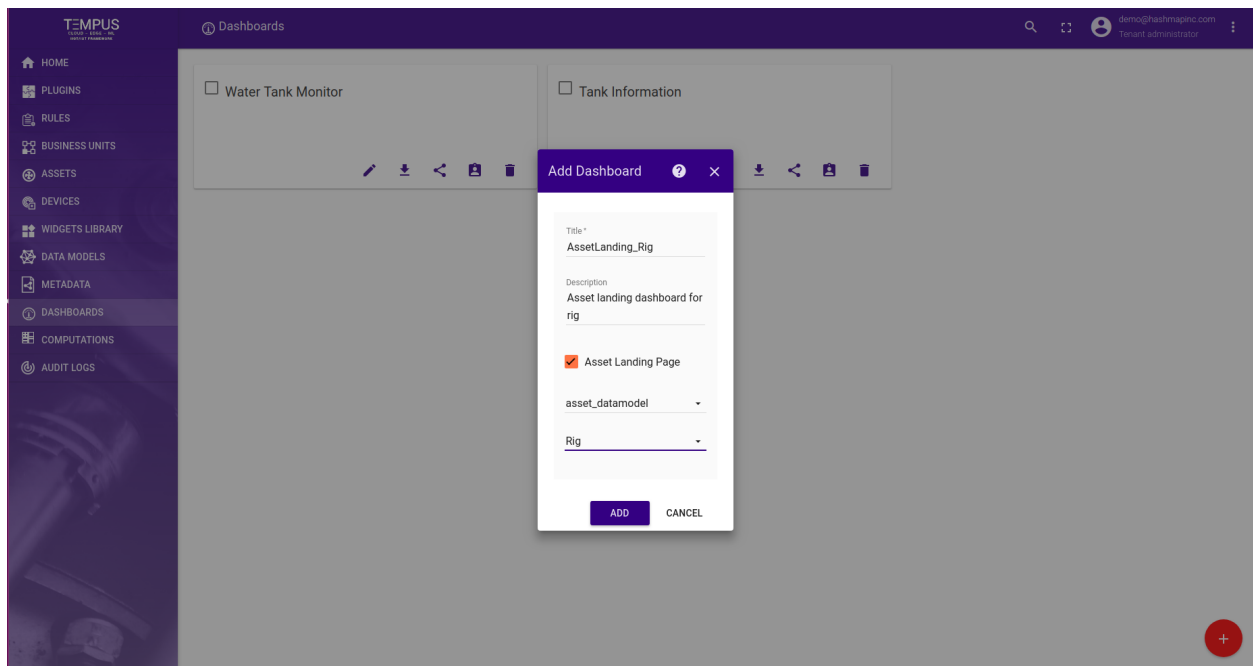
Create Asset Landing Dashboard

Data model is required to create asset landing dashboard. Created one data model named asset_datamodel which having well and rig assets.



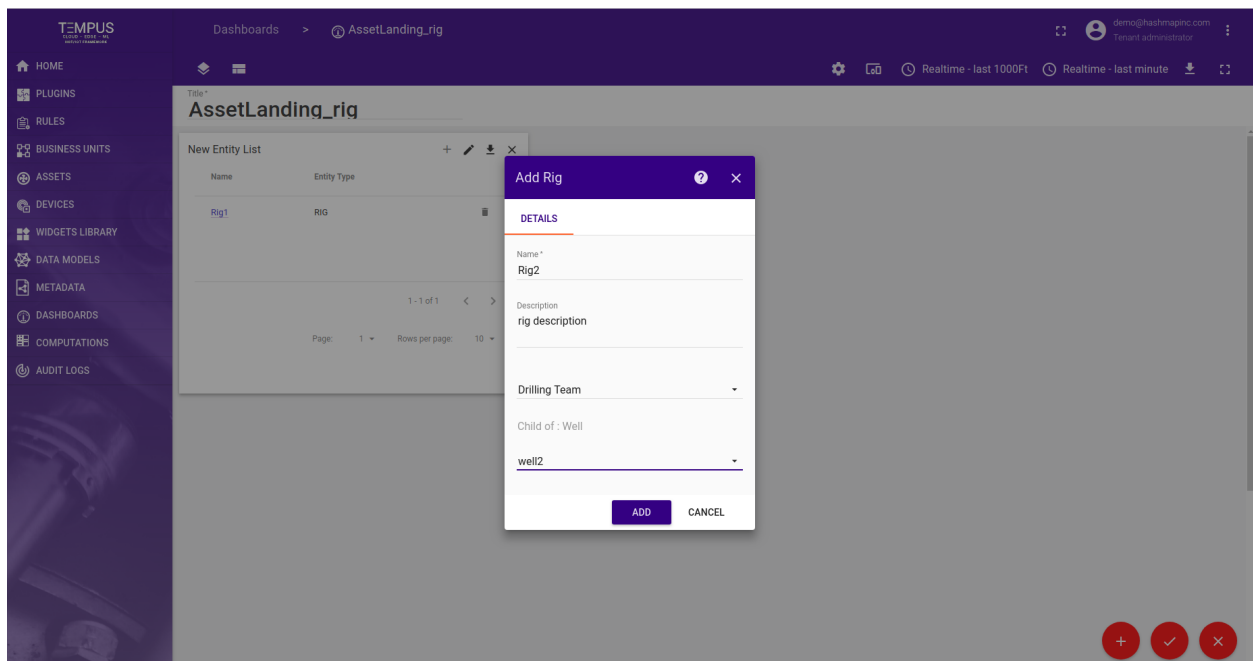
User can create asset landing dashboard from dashboard tab. To create asset landing dashboard need to give following:

- **Title** - the name of the asset landing dashboard.
- **Description** - the description of the asset landing dashboard.
- **Asset Landing Flag** - checked checkbox to set asset landing dashboard.
- **Data Model** - Select associated data model.
- **Asset of selected data model** - select asset from selected data model.



Add entity widget to create asset

To create asset from entity widget, add entity widget into asset landing dashboard. After adding widget, click on the + button to create rig type assets.

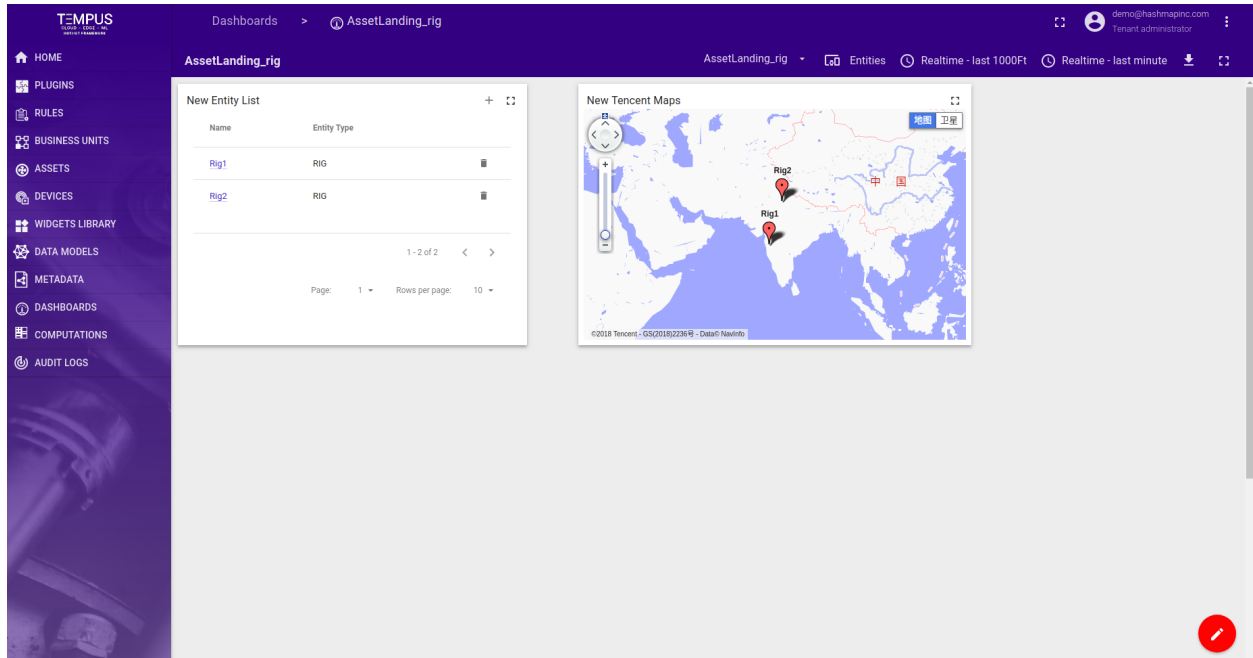


User can create asset from landing dashboard. To create asset, need to give the following:

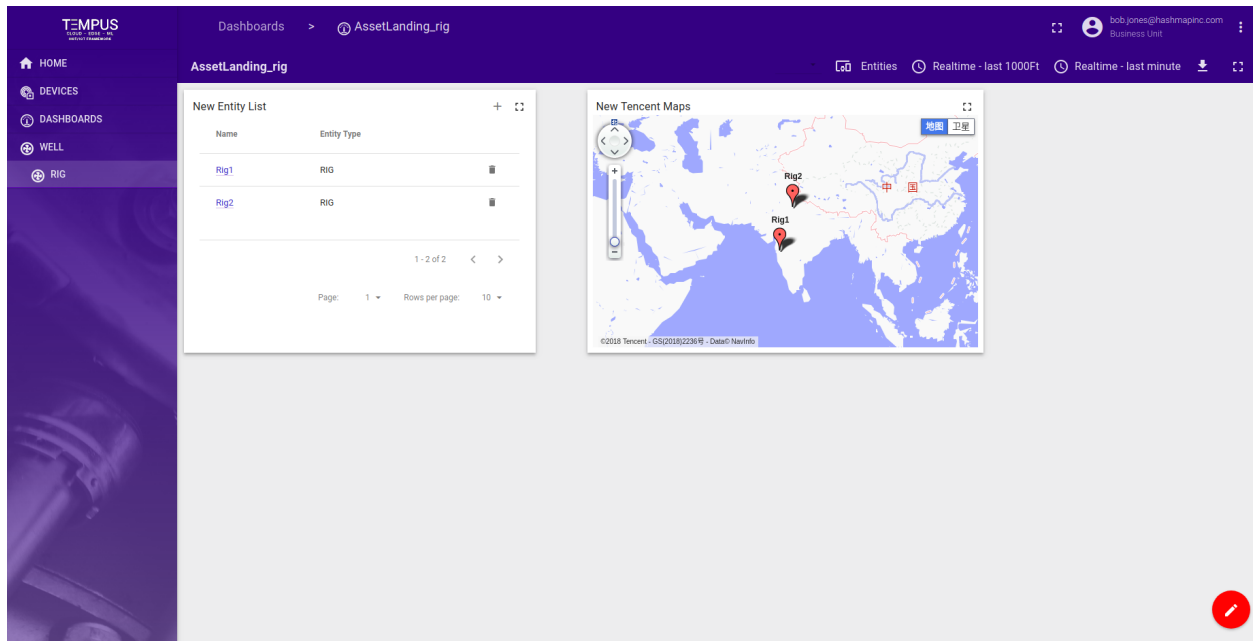
- **Name** - the name of the asset.
- **Description** - the description of the asset.
- **Assign Business Unit** - Assign asset to business unit.

- **Add relation** - If asset have parent, then in add parent relation.

Added rig with map widget.



This Landing page is assign to business Unit.If User logged in, will see asset stepper. As following



Cluster Information

Tempus supports following cluster information feature using Web UI.

System administrator is able to check how the load is distributed over different nodes in a cluster using **Cluster Info** option in side-bar menu.

Node Host	Node Port	Device Session Count	RPC Session Count	Node Status
localhost	9001	2	0	UP

System administrator is able to see the nodes details. Node details include the host, port, number of active device actor sessions, number of active rpc actor sessions and status of the node.

Cluster Information Web Ui will show the details of different nodes in cluster. To see updated values on cluster info Web UI page use refresh button on top right corner of the table.

Mail Settings

Tempus System Administrator is able to configure a connection to a SMTP server that will be used to distribute activation and password reset emails to users. This configuration step is required in production environments. If you are evaluating the platform, pre-provisioned demo accounts are sufficient in most of the use cases. NOTE System Mail settings are used only during user creation and password reset process and are controlled by a system administrator. Tenant administrator is able to setup email plugin to distribute alarms produced by rule engine.

- *Step 1. Login as system administrator*
- *Step 2. Change administrator email address*
- *Step 3. Open 'Outgoing Mail' and populate SMTP server settings*
 - *Step 3.1. Sendgrid configuration example*
 - *Step 3.2. Gmail configuration example*
- *Step 4. Save configuration*

Following steps are required to configure system mail settings.

Step 1. Login as system administrator

Login to your Tempus instance WEB UI as a system administrator using default account.

Step 2. Change administrator email address

Right click on the burger in the top-right corner of the WEB UI and select 'Profile'. Change 'sysadmin@hashmapinc.com' to your email address. Now re-login as administrator again.

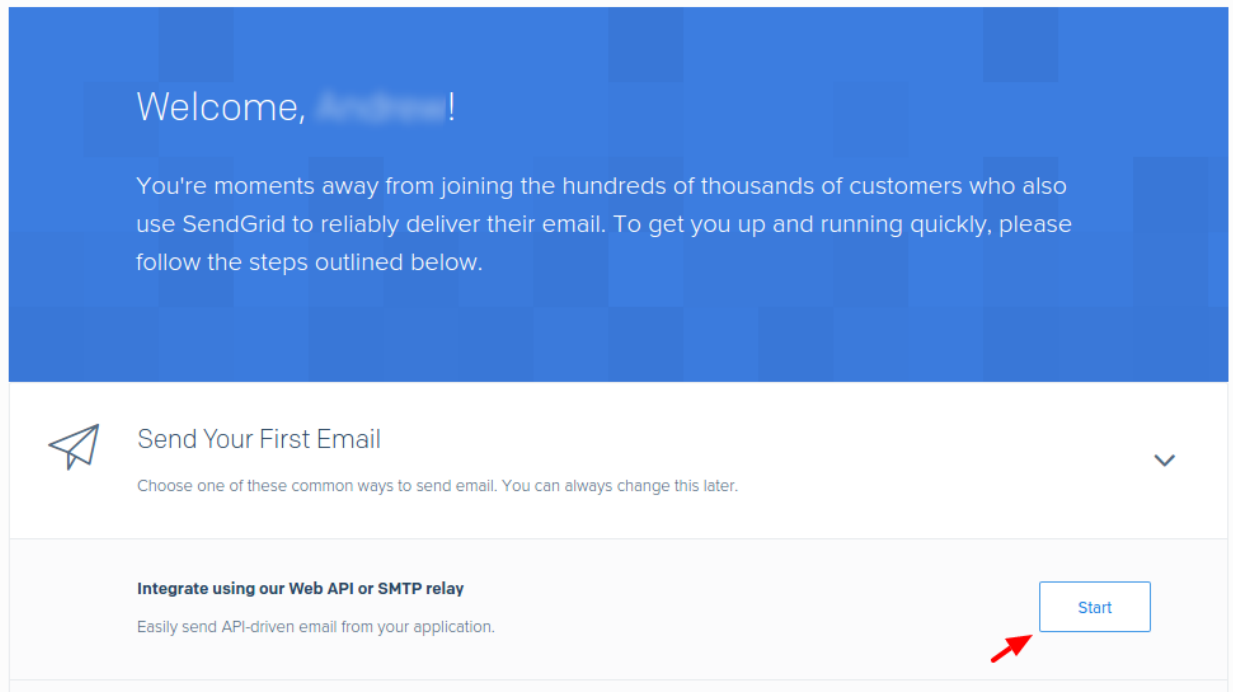
Step 3. Open 'Outgoing Mail' and populate SMTP server settings

Navigate to System Settings -> Outgoing Mail and populate the form. Click on 'Send Test Email' button. A test email will be sent to the email address that you have specified in 'Step 2'. In case of error in configuration, you should

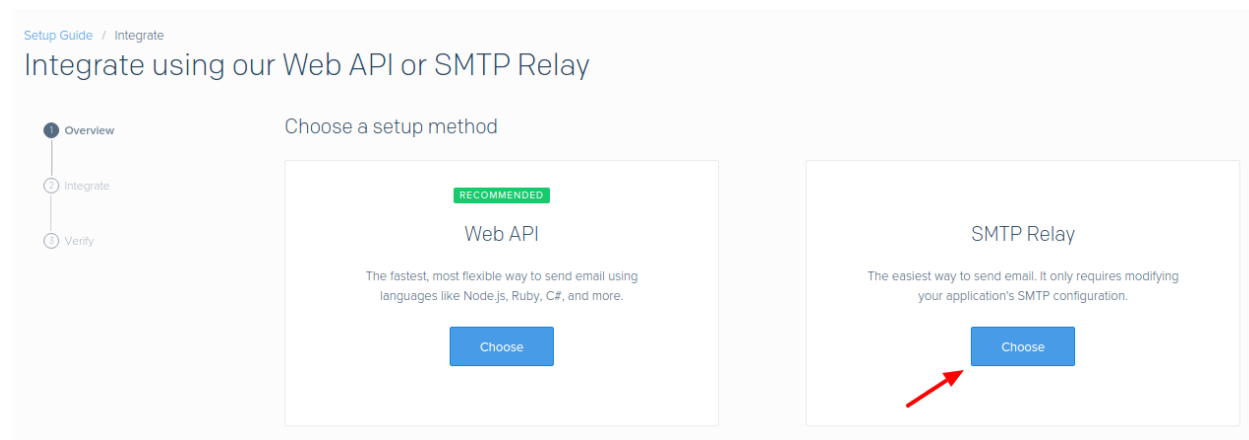
receive a popup with the error log.

Step 3.1. Sendgrid configuration example

SendGrid configuration is fairly simple and straightforward. First, you need to create [SendGrid](#) account. You can try it for free and the free plan is most likely enough for platform evaluation. Once you create your account, you will be forwarded to the welcome page. Now you can provision your SMTP Relay credentials. See the screen-shot below.



Please choose SMTP relay on the next page.



Once you populate the API key name and generate it, you will be able to copy-paste settings from the screen to Tempus mail settings form.

Integrate using our Web API or SMTP Relay

- Overview
- Integrate**
- Verify

How to send email using the SMTP Relay

1 Create an API key

This allows your application to authenticate to our API and send mail. You can enable or disable additional permissions on the [API keys page](#).

✔ "ThingsboardApiKey" was successfully created and added to the next step.

SG.7a [REDACTED] RTQ

2 Configure your application

Configure your application with the settings below.

Server	smtp.sendgrid.net
Ports	25, 587 (for unencrypted/TLS connections) 465 (for SSL connections)
Username	apikey
Password	SG.7a [REDACTED] RTQ

☐ I've updated my settings.

Next: Verify Integration

Copy-paste the settings, update 'Mail From' field and click on 'Send Test Mail' button.

- HOME
- PLUGINS
- RULES
- TENANTS
- WIDGETS LIBRARY
- SYSTEM SETTINGS
 - General
 - Outgoing Mail

System Settings > Outgoing Mail

Test mail was successfully sent! CLOSE

Mail From *
Thingsboard <name@company.com>

SMTP protocol
SMTPS

SMTP host *
smtp.sendgrid.net

SMTP port *
465

Timeout (msec) *
10000

☒ Enable TLS

Username
apikey

Password
.....

SEND TEST MAIL

SAVE


Once you receive the notification about a successful test, save populated data. You can also complete verification on the SendGrid website.

Integrate using our Web API or SMTP Relay

✓ Overview

✓ Integrate

3 Verify



It worked!

We successfully received your email. Your setup is complete!

[View Email Activity](#)

[View Documentation](#)

Step 3.2. Gmail configuration example

In order to use G-mail, you will need to do two extra steps. First, you need to [allow less secure apps](#). Second, you need to enable two-step verification and generate an [app password](#). Although the second step is not mandatory, it is highly recommended.

Generated app password

Email

securesally@gmail.com

Password

.....

Your app password for your device

How to use it

Go to the settings for your Google Account in the application or device you are trying to set up. Replace your password with the 16-character password shown above. Just like your normal password, this app password grants complete access to your Google Account. You won't need to remember it, so don't write it down or share it with anyone.

DONE

Once this is ready, you should be able to setup Gmail account using the information below

thingsboard.io

HOME
PLUGINS
RULES
TENANTS
WIDGETS LIBRARY
SYSTEM SETTINGS
General
Outgoing Mail

System Settings > Outgoing Mail

Outgoing Mail Settings

Mail From *
Thingsboard <[REDACTED]@gmail.com>

SMTP protocol
SMTPS

SMTP host *
smtp.gmail.com

SMTP port *
465

3 / 5

Timeout (msec) *
10000

5 / 6

☐ Enable TLS

Username
[REDACTED]@gmail.com

Password
.....

SEND TEST MAIL

SAVE

Similar settings are available for G-suite accounts, however, you may need to contact your system administrator to enable less secure apps, etc. Note that you can also enable/disable TLS using checkbox.

thingsboard.io

HOME
PLUGINS
RULES
TENANTS
WIDGETS LIBRARY
SYSTEM SETTINGS
General
Outgoing Mail

System Settings > Outgoing Mail

Test mail was successfully sent! CLOSE

Outgoing Mail Settings

Mail From *
[REDACTED]@thingsboard.io

SMTP protocol
SMTPS

SMTP host *
smtp-relay.gmail.com

SMTP port *
465

3 / 5

Timeout (msec) *
10000

5 / 6

☒ Enable TLS

Username
[REDACTED]@thingsboard.io

Password
.....

SEND TEST MAIL

SAVE

Step 4. Save configuration

Once you will receive test email you can save SMTP server configuration.

3.7 Tempus Rule Engine Reference

The rule engine consists of 4 distinct parts. They are discussed in the sections below.

Rule Engine Components

Processors

There are 5 filter types that allow a tenant administrator to perform a number of data driven actions.

Filter Types

Device Attributes Filter

Overview

This component allows filtering incoming messages by attributes of the device. This filter is very useful if you want to apply the rule only to the certain sub-set of your device. Filter expression is a javascript expression and basically defines this sub-set. You are able to use any attribute types.

Configuration

You are able to write boolean javascript expression using following bindings:

- **cs** - client-side attributes map.
- **ss** - server-side attributes map.
- **shared** - shared attributes map.

If you are not sure that certain attribute is present, you can add check it's type for undefined. For example, filter below will match if client-side attribute 'firmware_version' is set and equal to '1.0.0'

```
typeof cs.firmware_version !== 'undefined' && cs.firmware_version === '1.0.0'
```

Example

Assuming following device attributes and their types

- firmware_version - client-side
- country - client-side
- subscription_plan - shared
- balance - server-side

The following filter will match all premium subscription devices with positive balance that are located in the USA with firmware version equal to 1.1.0

```
cs.firmware_version=='1.1.0' && cs.country=='USA' && shared.subscription_plan==  
↪ 'premium' && ss.balance > 0
```

If you are not sure that all attributes are present for your device, you should use the following syntax that adds all necessary “null” checks

```
typeof cs.firmware_version !== 'undefined' &&  
typeof cs.country !== 'undefined' &&  
typeof shared.subscription_plan !== 'undefined' &&  
typeof ss.balance !== 'undefined' &&  
cs.firmware_version=='1.1.0' && cs.country=='USA' && shared.subscription_plan==  
↪ 'premium' && ss.balance > 0
```

Message Type Filter

Overview

This component allows filtering incoming messages by type. This filter is very efficient. We recommend using this filter in almost every rule to quickly ignore irrelevant messages.

Configuration

You are able to select multiple types: “Get Attributes”, “Post Attributes”, “Post Telemetry” and “RPC Request”.

Example

As a system administrator, you are able to review filter example inside Rules->System Telemetry Rule->Filters->Message Telemetry Filter.

Filter?×

Name *

TelemetryFilter

Type *

Message Type Filter ▼

Message types *

POST_TELEMETRY

POST_ATTRIBUTES

GET_ATTRIBUTES

SAVE

CANCEL

Device Telemetry Filter

Overview

This component allows filtering incoming telemetry messages by their values. This filter is very useful if you want to apply rule based on certain values of telemetry. For example, an engine controller may periodically report its temperature. When engine temperature is higher than 100 degrees you may raise an alert. The filter expression is written in javascript.

Configuration

You are able to write boolean javascript expression using bindings that match keys of your telemetry message. If you are not sure that certain key is present in your message, you can add check it's type for undefined. For example, filter below will match if temperature is higher then 100 degrees.

```
typeof temperature !== 'undefined' && temperature > 100
```

Assuming following telemetry message uploaded from engine controller device:

```
{ "temperature":1100, "enabled":true, "mode":"A" }
```

The following filter will match if device is enabled, operating in mode 'A' and temperature is greater than 1000 degrees


```
temperature > 1000 && enabled == true && mode == 'A'
```

If you are not sure that all telemetry data points are present in the message, you should use the following syntax that adds all necessary “null” checks

```
typeof temperature !== 'undefined' && typeof enabled !== 'undefined' && typeof mode !== 'undefined' &&  
temperature > 1000 && enabled == true && mode == 'A'
```

Example

Filter

?

×

Name *

waterTankLevel Filter

Type *

Device Telemetry Filter

▼

Filter*

i 1

typeof waterTankLevel !== 'undefined'

JAVASCRIPT

TIDY

CANCEL

Method Name Filter

Overview

This component allows filtering incoming RPC request messages by method name. This filter is very efficient and useful to forward RPC request to particular plugins that handle them.

Configuration

You are able to select multiple method names in one filter. For example, if you want to have two plugins (their functionality is just for the demo purposes):

- plugin A allows getting current time
- plugin B allows getting the weather forecast You may implement plugin A to handle getTime method and plugin B to handle getWeather method. In this case you will need to configure two rules:
- rule A that points to plugin A based on “getTime” method filter
- rule B that points to plugin B based on “getWeather” method filter

Device Type Filter

Overview

This component allows for filtering data based on the Device Type of the device that the data originated from.

Configuration

You are able to add one or more device types to the filter. Simply click the **+ NEW** button and add the device type to filter on.

Example

Filter

?

×

Name *

Water Tank Filter

Type *

Device Type Filter

▼

Device types

1.

×

Device Type

Device Type

WaterTank

+

NEW

SAVE

CANCEL

Actions

Actions are executions via plugins that allow Tempus to perform a task based on the result of one or more filters within a rule.

Action Types

Kafka Plugin Action

Overview

This component allows creating a kafka message by substitution of device attributes and message data into configurable templates.

Configuration

During action configuration you are able to specify following:

- Set flag to confirm delivery
- Kafka topic name
- Kafka body template The Body Template syntax is based on Velocity

RabbitMQ Plugin Action

Overview

This component allows creating RabbitMQ message by substitution of device attributes and message data into configurable templates.

Configuration

During action configuration you are able to specify following: * set flag to confirm delivery * rabbitmq exchange name * rabbitmq queue name * rabbitmq message properties (BASIC, MINIMAL_BASIC, MINIMAL_PERSISTENT_BASIC, PERSISTENT_BASIC, PERSISTENT_TEXT_PLAIN, TEXT_PLAIN) * rabbitmq message body template The Body Template syntax is based on Velocity

REST API Call Plugin Action

Overview

This component allows creating a POST/PUT request body by substitution of device attributes and message data into configurable templates.

Configuration

During action configuration you are able to specify following:

- set flag to confirm delivery
- action path of the http endpoint
- request method - POST or PUT
- expected result code in http response

- body template of the request The Body Template syntax is based on Velocity and is already described in alarm processor documentation.

Send Mail Action

Overview

This component allows building email message by substitution of device attributes and message data into configurable templates.

Configuration

During action configuration you are able to specify following templates: from, to, cc, bcc, subject and body. The template syntax is based on Velocity and is already described in alarm processor documentation. Additionally, you can specify Send Flag property. This is an optional property that may be used in combination with isNewAlarm or left blank.

Telemetry Plugin Action

Overview

This component allows forwarding incoming attributes and timeseries requests to telemetry plugin.

Configuration

There are two additional fields added in configuration i.e

- **Tag Quality Time Window** : It is the time period over which the quality parameters like avg, mean, median etc will be calculated for a tag(i.e. key) present in telemetry data.
- **Tag Quality Depth Window** : It is the depth period over which the quality parameters like avg, mean, median etc will be calculated for a tag(i.e. key) present in depth telemetry data.

Example

As a system administrator, you are able to review action example inside Rules->System Telemetry Rule->Actions->Telemetry Plugin Action.

Plugins

Plugins allow Tempus to interact with other systems through various means. The currently supported plugins are below.

Platforms

Device Messaging Plugin

Overview

This RPC plugin enables communication between various IoT devices through the Tempus cluster. The plugin introduces basic security features: devices are able to exchange messages only if they belong to the same customer. The plugin implementation can be customized to cover more complex security features. Configuration

You can specify following configuration parameters:

- Maximum amount of devices per customer
- Default request timeout
- Maximum request timeout

Device RPC API

The plugin handles two rpc methods: `getDevices` and `sendMsg`. The examples listed below will be based on demo account and MQTT protocol. Please note that you are able to use other protocols - CoAP and HTTP.

Get Device List API

In order to send a message to other devices, you will need to know their identifiers. A device can request a list of other devices that belong to the same customer using `getDevices` RPC call.

mqtt-get-device-list.sh

```
export TOKEN=A1_TEST_TOKEN
node mqtt-get-device-list.js
```

mqtt-get-device-list.js

```
var mqtt = require('mqtt');
var client = mqtt.connect('mqtt://127.0.0.1', {
  username: process.env.TOKEN
});

client.on('connect', function () {
  console.log('connected');
  client.subscribe('v1/devices/me/rpc/response/+');
  var requestId = 1;
  var request = {
    "method": "getDevices",
    "params": {}
  };
  client.publish('v1/devices/me/rpc/request/' + requestId, JSON.stringify(request));
});

client.on('message', function (topic, message) {
  console.log('response.topic: ' + topic);
  console.log('response.body: ' + message.toString());
});
```

RESPONSE

```
[
  {
    "id": "aa435e80-9fce-11e6-8080-808080808080",
    "name": "Test Device A2"
  },
  {
    "id": "86801880-9fce-11e6-8080-808080808080",
    "name": "Test Device A3"
  }
]
```

Send Message API

A device can send a message to other device that belongs to the same customer using `sendMsg` RPC call. The example below will attempt to send a message from device “Test Device A1” to device “Test Device A2”.

mqtt-send-msg.sh

```
export TOKEN=A1_TEST_TOKEN
node 'mqtt-send-msg.js'
```

mqtt-send-msg.js

```
var mqtt = require('mqtt');
var client = mqtt.connect('mqtt://127.0.0.1', {
  username: process.env.TOKEN
});

client.on('connect', function () {
  console.log('connected');
  client.subscribe('v1/devices/me/rpc/response/+');
  var requestId = 1;
  var request = {
    method: "sendMsg",
    params: {
      deviceId: "aa435e80-9fce-11e6-8080-808080808080",
      timeout: 2000,
      oneway: false,
      body: {
        param1: "value1"
      }
    }
  };
  client.publish('v1/devices/me/rpc/request/' + requestId, JSON.stringify(request));
});

client.on('message', function (topic, message) {
  console.log('response.topic: ' + topic);
  console.log('response.body: ' + message.toString());
});
```

As a result, you should receive the following error:

```
{"error": "No active connection to the remote device!"}
```

Let's launch emulator of target device and send message again:

mqtt-recieve-msg.sh

```
export TOKEN=A2_TEST_TOKEN
node mqtt-receive-msg.js
```

mqtt-recieve-msg.sh

```
var mqtt = require('mqtt');
var client = mqtt.connect('mqtt://127.0.0.1', {
  username: process.env.TOKEN
});

client.on('connect', function () {
  console.log('connected');
  client.subscribe('v1/devices/me/rpc/request/+');
});

client.on('message', function (topic, message) {
  console.log('response.topic: ' + topic);
  console.log('response.body: ' + message.toString());
  client.publish(topic.replace('request', 'response'), '{"status":"ok"}');
});
```

As a result, you should receive following response from device:

```
{ "status": "ok" }
```

Note that target device id, access tokens, request and response bodies are hardcoded into scripts and correspond to devices that must be created beforehand.

Telemetry Plugin

Overview

Telemetry plugin is responsible for: * persisting attribute updates to internal data storage; * persisting timeseries data to internal data storage; * provides server-side API to query and subscribe for data updates. Since Telemetry plugin functionality is critical for data visualization purposes in dashboards, it is configured on the system level by a system administrator. Advanced users or platform developers can customize telemetry plugin functionality.

Configuration

There is no specific configuration for this component.

Server-side API

Telemetry plugin API description is available in corresponding attributes and telemetry guides.

Example

As a system administrator, you are able to review plugin example inside Plugins->System Telemetry Plugin.

RPC Plugin

Overview

RPC plugin is responsible for: * providing REST API to send RPC request from server-side applications to devices; * pushing RPC request to devices via one of available protocols: MQTT, CoAP or HTTP;

By default, this plugin is configured on the system level by a system administrator. You are able to configure your own instance of the plugin on tenant level. Advanced users or platform developers can customize rpc plugin functionality.

Configuration

You can specify default RPC timeout for plugin instance in the plugin configuration.

Server-side API

RPC plugin API description is available in corresponding rpc guides.

As a system administrator, you are able to review plugin example inside Plugins->System RPC Plugin

Mail Plugin

Overview

Mail plugin is responsible for sending email messages that are triggered by corresponding rule actions.

Configuration

You can specify following parameters:

- mail server host
- mail server port
- user name
- password
- map with other properties. See java mail sender for more details.

Server-side API

This plugin does not provide any server-side API.

Kafka Plugin

Overview

Kafka plugin is responsible for sending messages to Kafka brokers triggered by specific rules

Configuration

You can specify following configuration parameters:

- *bootstrap servers* - list of kafka brokers
- *number of attempts to reconnect to kafka* if connection fails
- *number of messages to unit* into batch on client
- *time to buffer locally* before sending to kafka broker (in ms)
- *buffer max size* on client
- *minimum number of replicas that must acknowledge a write*
- *topic key serializer* by default - org.apache.kafka.common.serialization.* StringSerializer
- *topic value serializer* by default - * org.apache.kafka.common.serialization.StringSerializer
- any other additional properties could be provided for kafka broker connection

Server-side API

This plugin does not provide any server-side API.

Example

In this example, we are going to demonstrate how you can configure this extension to be able to send a message to Kafka topic every time new telemetry message for the device arrives.

Prerequisites before continuing Kafka extension configuration:

- Kafka broker is up and running
- Appropriate Kafka Topic created
- Tempus is up and running

Kafka Plugin Configuration

Let's configure Kafka plugin first. Go to Plugins menu and create new plugin:

Please set correctly Kafka Bootstrap Servers URL and any other parameters located in plugin configuration section that is suitable for your case so Kafka extension is able to connect to Kafka broker.

Click on **Activate** plugin button:

Kafka Rule Configuration

Now it's time to create appropriate Rule.

Add filter for **POST_TELEMETRY** message type:

Click **Add** button to add filter.

Then select 'Kafka Plugin' in the drop-down box for the Plugin field:

Add action that will send temperature telemetry of device to particular kafka topic.

Click **Add** button and then activate Rule. Sending Temperature Telemetry

Now you can send Telemetry message that contains 'temp' telemetry for any of your devices:

```
{ "temp": 73.4 }
```

You should see **73.4** message in appropriate Kafka topic once you'll post this message. Here is an example of a command that publish single telemetry message to locally installed Tempus:

```
mosquitto_pub -d -h "localhost" -p 1883 -t "v1/devices/me/telemetry" -u "$ACCESS_TOKEN"  
↪ -m '{"temp":73.4}'
```

Time RPC Plugin

Overview

Simple RPC plugin that is responsible for handling “getTime” RPC request from devices. This plugin is a part of default Tempus installation for demo purposes. It demonstrates that devices can send RPC request via various connectivity protocols to execute server-side logic and get the result.

Configuration

You can specify time format configuration parameter. See DateFormatter API for more details.

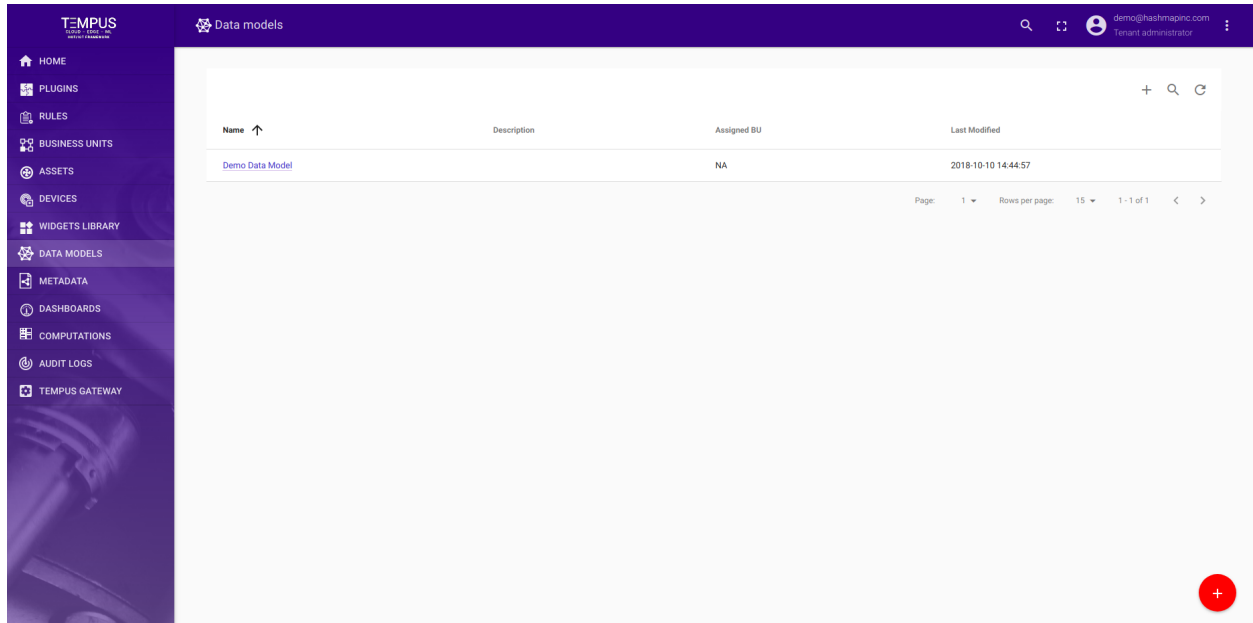
Server-side API

This plugin does not provide any server-side API.

3.8 Data Model Reference

This reference document explains different actions and operation related to Data Model. The document includes creation, updation, deletion and assignment of Data Model to different Business Units and creating Asset Landing Pages for DataModel Objects.

Data Model Information

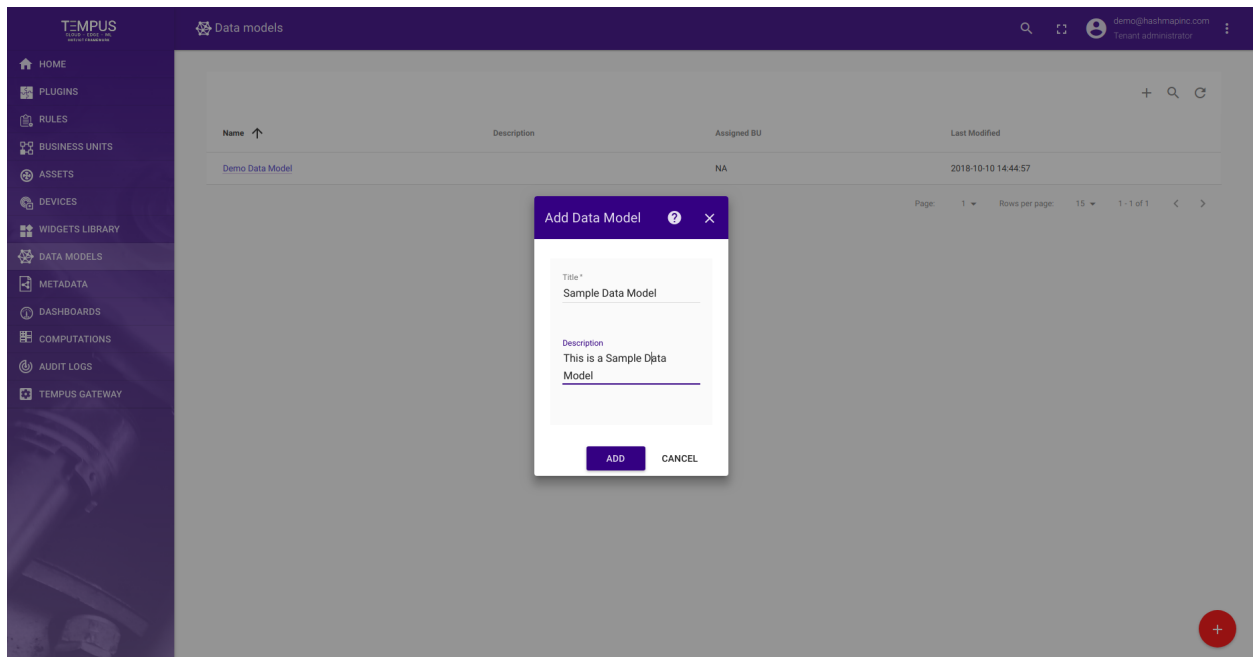


The screenshot shows the 'Data models' page in the Tempus application. The left sidebar contains a navigation menu with items: HOME, PLUGINS, RULES, BUSINESS UNITS, ASSETS, DEVICES, WIDGETS LIBRARY, DATA MODELS (highlighted), METADATA, DASHBOARDS, COMPUTATIONS, AUDIT LOGS, and TEMPUS GATEWAY. The main content area displays a table with the following columns: Name, Description, Assigned BU, and Last Modified. A single row is visible with the name 'Demo Data Model' and an assigned BU of 'NA'. The last modified date is '2018-10-10 14:44:57'. At the bottom right of the table area, there is a red circular button with a white plus sign.

Name	Description	Assigned BU	Last Modified
Demo Data Model		NA	2018-10-10 14:44:57

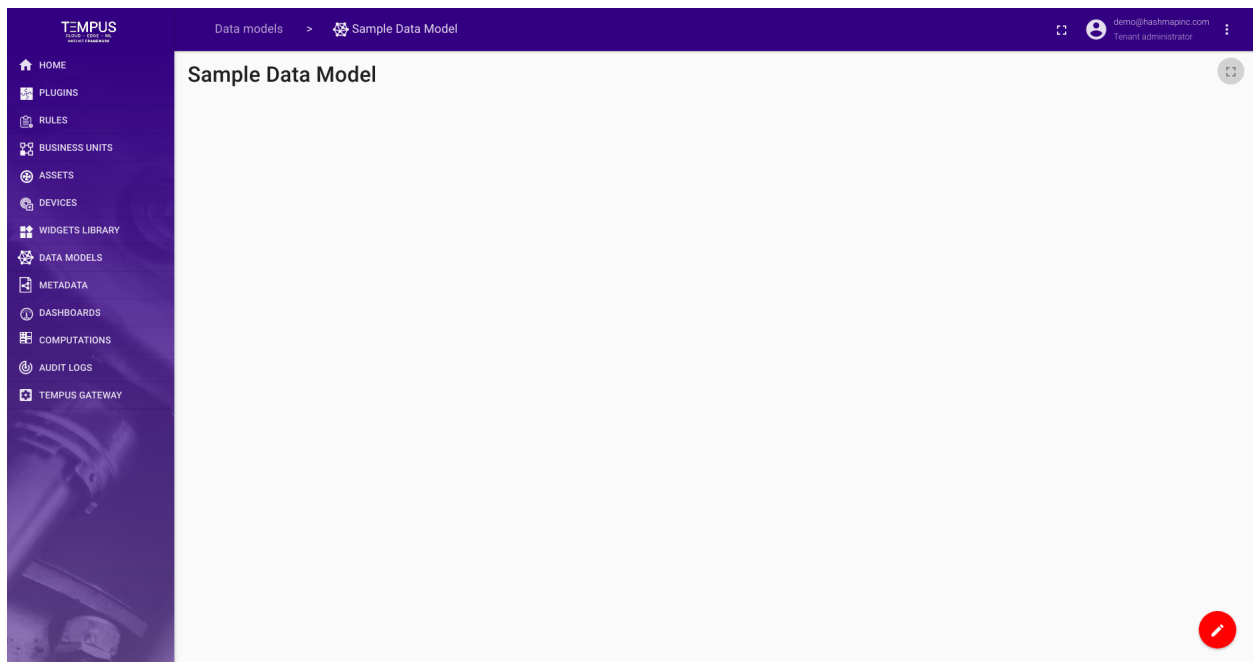
The page displays the overall created Data Model for login tenant admin.

Data Model Creation



This screenshot shows the same 'Data models' page as the previous one, but with an 'Add Data Model' pop-up dialog open in the center. The dialog has a title bar with a question mark icon and a close button. It contains two text input fields: 'Title' with the value 'Sample Data Model' and 'Description' with the value 'This is a Sample Data Model'. At the bottom of the dialog are two buttons: 'ADD' and 'CANCEL'. The background table and sidebar are dimmed.

Click on orange button to Add a Data Model, a pop-up will open, fill in the details related to Data Model. On clicking on Add button, you will be redirected to canvas for Adding DataModel Objects.



For adding DataModel Object refer DataModel Object Reference documentation.

DataModel Object Reference

This reference explains the different parts and operation related DataModel Object like creation, updation and deletion.

Object Information

The screenshot shows the Tempus Data Model Object creation interface. The 'New Datamodel Object' dialog is open, displaying the 'OBJECT INFORMATION' tab. The form includes the following fields:

- Object Name ***: Developer (9 / 256 characters)
- Description**: This object represent a Developer (33 / 500 characters)
- Tempus Type ***: Asset (dropdown menu)
- Icon**: Developer.png (upload area with a dashed border and a close button)

A note at the bottom of the dialog states: "Note: If No Icon Uploaded Then Default Icon Of Assets Will Be Applied." The dialog has tabs for OBJECT INFORMATION, ATTRIBUTES, RELATIONSHIPS, and REVIEW, and a NEXT button at the bottom right.

The first page of the DataModel Object creation stepper is for entering general object information. This includes:

- **Object Name** - the name of the object. This must be unique within a datamodel.
- **Object Description** - the optional description of the object.
- **Object Type** - the type of object this is. This can be either `Device` or `Asset`.
- **Object Icon** - the icon of object. This can only be provided for `Asset`.

For our example, let's create a `Developer` object.

Object Attributes

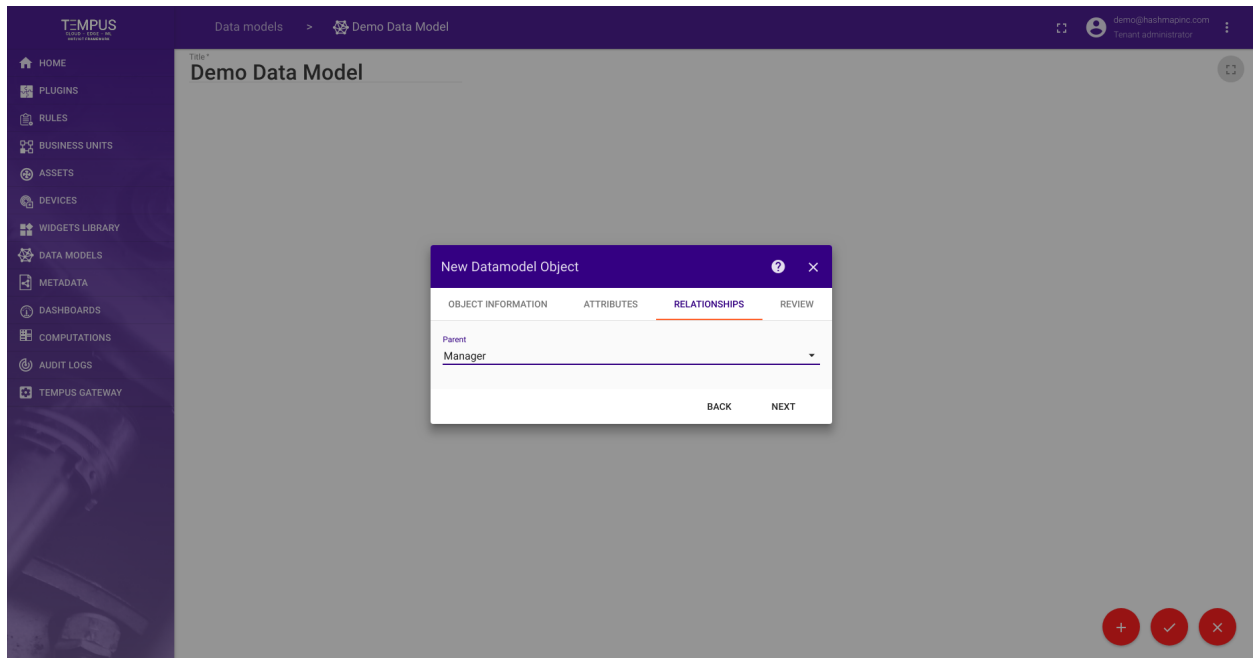
The screenshot displays the TEMPUS Data Model interface. A modal window titled 'New Datamodel Object' is open, showing the 'ATTRIBUTES' tab. The attributes listed are Name, Age, Heart Rate, and Location. Below the list is a text input field labeled 'Attribute Name' with a character count '8 / 256'. The modal has 'BACK' and 'NEXT' buttons at the bottom. The background shows the 'Demo Data Model' page with a sidebar menu containing options like HOME, PLUGINS, RULES, BUSINESS UNITS, ASSETS, DEVICES, WIDGETS LIBRARY, DATA MODELS, METADATA, DASHBOARDS, COMPUTATIONS, AUDIT LOGS, and TEMPUS GATEWAY.

The Object Attributes section is where the attribute fields of this object are defined. For our *Developer* example object, we have created attributes for:

- *Name* - the name of the developer. This could be manually entered for each developer.
- *Age* - the age of the developer. This could be updated based on an external birthday metadata source.
- *Heart Rate* - the developer's heart rate. This could come from a fitness *Device* associated with the developer.
- *Location* - the location of the developer. This could come from a mobile *Device* associated with the developer.

These attributes are of course just examples to illustrate how to use object attributes. We don't actually monitor the developers this closely (yet).

Object Relationships



The Relationships section is where the DataModel Object can define how it relates to other objects in the DataModel. At this point, only a `Parent` relationship is supported, but more general relationships will be supported in the future. For our example, let's assume we have already created a `Manager` datamodel object that we would like to assign as the `Parent` of our `Developer` object.

Object Review

The screenshot shows the Tempus Data Model interface. A modal window titled "New Datamodel Object" is open, displaying the "REVIEW" tab. The dialog contains the following information:

- Datamodel Object**
 - Name: Developer
 - Description: This object represent a Developer
 - Type: Asset
- Object Attributes**
 - Name
 - Age
 - Heart Rate
 - Location
- Relationships**
 - Parent: Manager
- Image**
 - Icon:

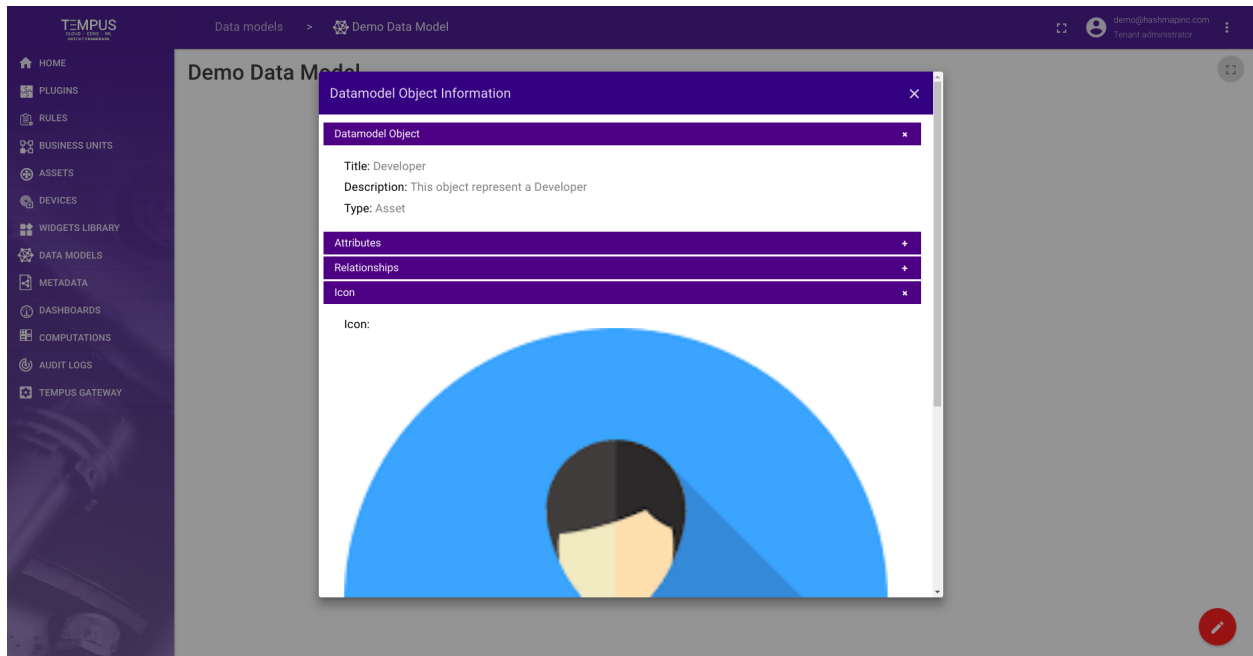
At the bottom of the dialog are "BACK" and "SUBMIT" buttons. In the bottom right corner of the main interface, there are three red circular buttons: a plus sign (+), a checkmark (✓), and a close sign (X).

The Review section allows us to review the DataModel Object we have defined.

For our example, this looks right for our `Developer` object. We will accept by clicking `Submit` to save our object to the datamodel.

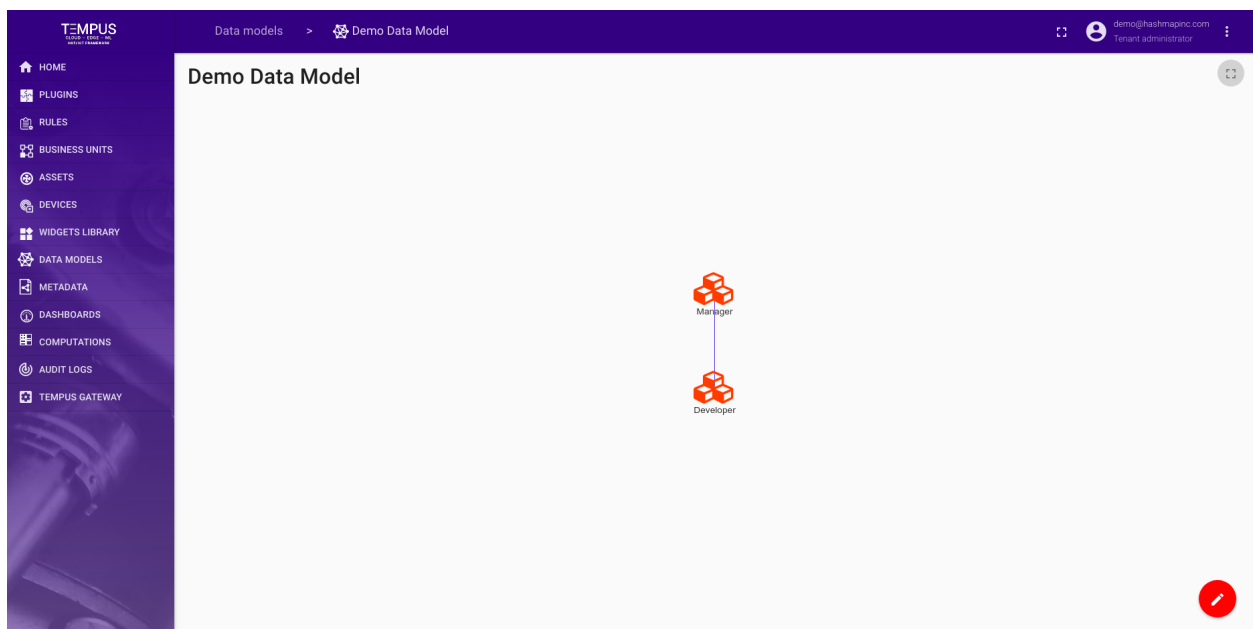
NOTE : After adding DataModel Objects click on the orange tick button on buttom right corner, otherwise your changes will not be saved.

Object Saved Review



The final DataObject Model view for Developer Object after submitting and saving changes of DataModel.

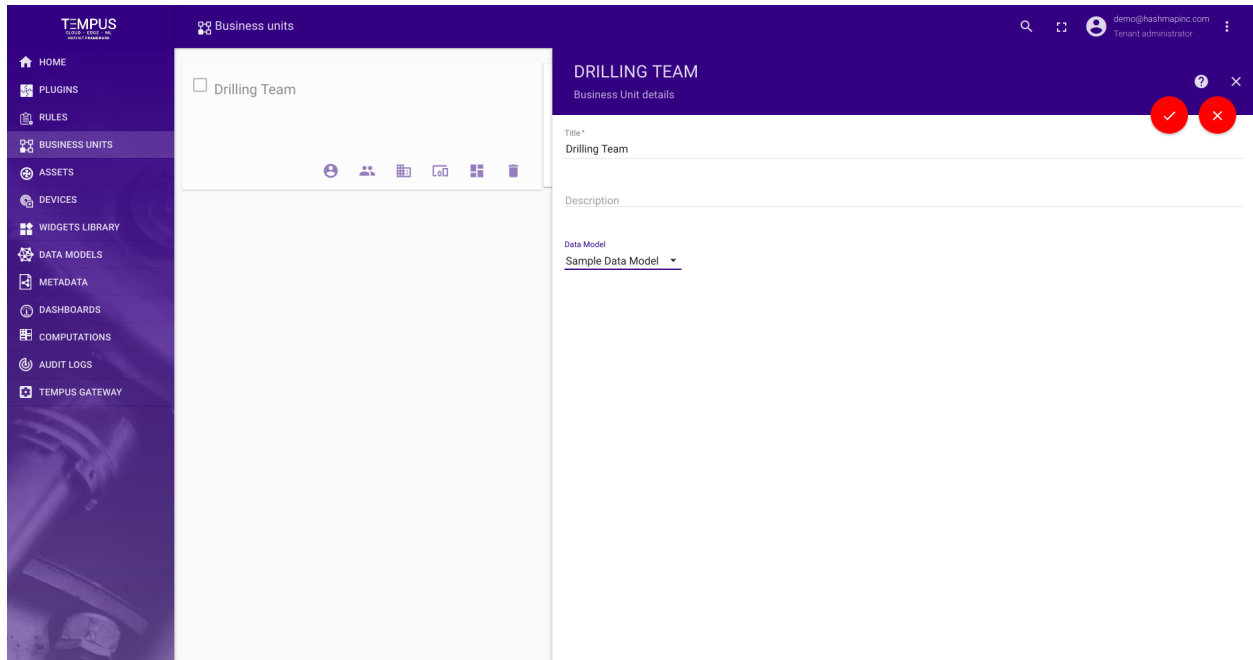
Results



With that, we have created our first small DataModel Object. This can be expanded to account for the devices associ-

ated with a Developer (macbook, fitness tracker, phone, etc...).

Assign Data Model to Business Unit

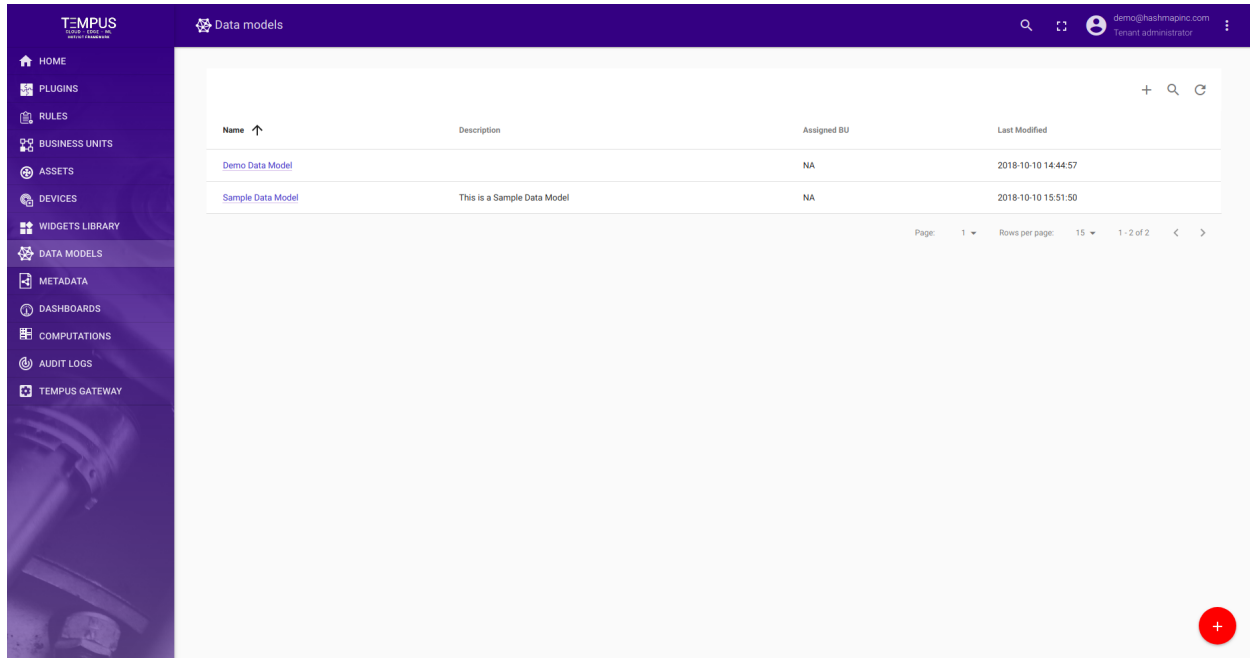


To assign a Data Model to a Business Unit, move to Business Unit tab and click on particular Business Unit card. This will open a side navigation for Business Unit, now click on edit and select the Data Model from DataModel drop-down and finally apply changes.

Asset Landing Pages for Data Model Object

For creating Asset Landing Pages refer to Asset Landing Page document.

Results



The screenshot shows the Tempus Data Models interface. On the left is a sidebar with navigation links: HOME, PLUGINS, RULES, BUSINESS UNITS, ASSETS, DEVICES, WIDGETS LIBRARY, DATA MODELS (selected), METADATA, DASHBOARDS, COMPUTATIONS, AUDIT LOGS, and TEMPUS GATEWAY. The main area is titled 'Data models' and contains a table with the following data:

Name ↑	Description	Assigned BU	Last Modified
Demo Data Model		NA	2018-10-10 14:44:57
Sample Data Model	This is a Sample Data Model	NA	2018-10-10 15:51:50

At the bottom right of the table area, there is a red circular button with a white plus sign. Below the table, pagination information is visible: Page: 1, Rows per page: 15, 1 - 2 of 2.

With that, we have created our Data Model.

3.9 Development

This section will help you start developing on Tempus Cloud

Tempus Developer Quickstart

Source Code

The Tempus source code is version controlled using Git version control at [GitHub](#)

The Tempus documentation source code is available at [GitHub](#)

Issue Tracking

Track issues on [GitHub](#)

Building

Configure your git client

We recommend running the following git config commands in order to ensure that git checks out the repository in a consistent manner. These changes are particularly important if running on Windows, as the git client has trouble with long filenames otherwise. Additionally, in Windows, the default behavior of the git client, when installed, is to set the core.autocrlf configuration option to true, which can cause some of the unit tests to fail.

```
git config --global core.longpaths true
git config --global core.autocrlf false
```

Checking out from Git

To check out the code:

```
git clone https://github.com/hashmapinc/Tempus.git
```

Then checkout the 'master' branch

```
git checkout master
```

The master branch currently represents the last release of Tempus. If you would like to work on the active / stable branch you need to check out the 'dev' branch.

```
git checkout dev
```

Tempus Development Environment Setup

Pre-requisites

- Docker
- Docker Compose
- Make
- Maven > 3.1
- JDK 11

Source Code

The Tempus source code is version controlled using Git version control at [GitHub](#)

The Tempus documentation source code is available at [GitHub](#)

Issue Tracking

Track issues on [GitHub](#)

Introduction

The TempusDevEnvironment orchestrates the setup of a developer environment suitable for demos or development work. This setup requires 3 repositories to be pulled:

1. TempusDevEnvironment (this one)
2. Tempus (described above)
3. Nifi-simulator-bundle (described below)

Checking out the TempusDevEnvironment from Git

To clone the TempusDevEnvironment code:

```
git clone https://github.com/hashmapinc/TempusDevEnvironment.git
```

To clone out the simulator bundle:

```
git clone https://github.com/hashmapinc/nifi-simulator-bundle.git
```

Once you have cloned both repositories edit the Makefile in the root of the TempusDevEnvironment project.

Modify the top 2 lines that specify `PROJECT_DIR` (this is the root of the Tempus project) and the `SIM_PROJECT_DIR` (this is the simulator bundle that was cloned with the command above).

There are 6 options in this make file:

1. **Install**

- This will only build Tempus and the Simulator bundle following the steps in the first section

2. **Copy**

- This will copy the files to the appropriate place so the docker images can be built

3. **Build**

- This will build and start the docker compose

4. **Build-ldap**

- This will build the environment with openLdap and ldapAdmin

To get started run

```
make all
```

Widgets Development Guide

Introduction

Tempus Cloud widgets are additional UI modules that can be easily integrated into any Dashboard and provide end-user functions such as data visualization, remote device control, alarms management and displaying static custom html content. According to the provided features each widget definition represents specific Widget Type.

Creating a Widget Definition

In order to create a new widget definition, navigate to the “Widget Library”, then open existing “Widgets Bundle” or create a new one. In the “Widgets Bundle” view, click on the big “+” button in the bottom-right part of the screen and then click on the “Create new widget type” button.

Select widget type window should popup and you will be prompted to select corresponding widget type that you are going to develop.

After that the “Widget Editor” page will be opened pre-populated with the starter widget template according to the previously selected widget type.

Widget Editor overview

Widget Editor view represents mini IDE designed to develop custom widget definitions. It consists of top toolbar and four main sections:

- Resources/HTML/CSS
- JavaScript
- Settings schema
- Widget preview

Widget Editor Toolbar

Widget Editor Toolbar consists of the following items:

- **Widget Title field** - used to specify title of the widget definition
- **Widget Type selector** - used to specify type of the widget definition
- **Run button** - used to run widget code and view result in the Widget preview section
- **Undo button** - reverts all editor sections to latest saved state
- **Save button** - saves widget definition
- **Save as button** - allows to save a new copy of widget definition by specifying new widget type name and target Widgets Bundle

Resources/HTML/CSS section

This section consists of three tabs. The first Resources tab is used to specify external JavaScript/CSS resources used by the widget.

Second HTML tab contains widget html code (Note: some widgets create html content dynamically and initial html content can be empty).

Third CSS tab contains widget specific CSS style definitions.

JavaScript section

This section contains all widget related JavaScript code according to the Widget API.

Settings schema section

This section consists of two tabs. The first Settings schema tab is used to specify json schema of widget settings in order to auto-generate UI form using react-schema-form builder. This generated UI form is displayed in the Advanced tab of widget settings. Settings Object serialized by this schema is used to store specific widget settings and accessible from widget JavaScript code.

Second Data key settings schema tab is used to specify json schema of particular data key settings in order to auto-generate UI form using react-schema-form builder. This generated UI form is displayed in Advanced tab of the Data key configuration dialog. Settings Object serialized by this schema is used to store specific settings for each data key of the datasource defined in the widget. These settings are accessible from widget JavaScript code.

Widget preview section

This section is used to preview and test widget definition. It is presented as mini dashboard containing one widget instantiated from the current widget definition. It has mostly all functionality provided by usual Tempus dashboard but has some limitations. For example, only “Function” can be selected as datasource type in widget datasources section for debug purposes.

Basic widget API

All widget related code is located in the JavaScript section. The built-in variable `self` that is a reference to the widget instance is also available. Each widget function should be defined as a property of the `self` variable. `self` variable has property `ctx` - reference to widget context that has all necessary API and data used by widget instance. Below is brief description of widget context properties:

Property	Type	Description
<code>\$container</code>	jQuery Object	Container element of the widget. Can be used to dynamically access or modify widget DOM using jQuery API.
<code>\$scope</code>	Object	Angular scope object of the current widget element. Can be used to access/modify scope properties when widget is built using Angular approach.
<code>width</code>	Number	Current width of widget container in pixels.
<code>height</code>	Number	Current height of widget container in pixels.
<code>isEdit</code>	Boolean	Indicates whether the dashboard is in the view or editing state.
<code>isMobile</code>	Boolean	Indicates whether the dashboard view is less then 960px width (default mobile break-point).
<code>widgetConfig</code>	Object	Common widget configuration containing properties such as color (text color), backgroundColor (widget background color), etc.
<code>settings</code>	Object	Widget settings containing widget specific properties according to the defined settings json schema
<code>units</code>	String	Optional property defining units text of values displayed by widget. Useful for simple widgets like cards or gauges.
<code>decimals</code>	Number	Optional property defining how many positions should be used to display decimal part of the value number.
<code>hideTitlePanel</code>	Boolean	Manages visibility of widget title panel. Useful for widget with custom title panels or different states.
<code>defaultSubscription</code>	Object	See Subscription object
<code>timewindow-Functions</code>	Object	See Timewindow functions
<code>controlApi</code>	Object	See Control API
<code>actionsApi</code>	Object	See Actions API
<code>stateController</code>	Object	See State Controller

In order to implement new widget the following JavaScript functions should be defined (Note: each function is optional and can be implemented according to the widget specific/behaviour):

Function	Description	
onInit()	The first function which is called when widget is ready for initialization. Should be used to prepare widget DOM, process widget settings and initial subscription information.	
onDataUpdated()	Called when the new data is available from the widget subscription. Latest data can be accessed from the defaultSubscription object of widget context (ctx).	
onResize()	Called when widget container is resized. Latest width and height can be obtained from widget context (ctx).	
onEditModeChanged()	Called when dashboard editing mode is changed. Latest mode is handled by isEdit property of ctx.	
onMobileModeChanged()	Called when dashboard view width crosses mobile breakpoint. Latest state is handled by isMobile property of ctx.	
onDestroy()	Called when widget element is destroyed. Should be used to cleanup all resources if necessary.	
getSettingsSchema()	Optional function returning widget settings schema json as alternative to Settings tab of Settings schema section.	
getDataKeySettingsSchema()	Optional function returning particular data key settings schema json as alternative to Data key settings schema tab of Settings schema section.	
typeParameters()	Returns object describing widget datasource parameters. See Type parameters object.	
actionSources()	Returns object describing available widget action sources used to define user actions. See Action sources object.	

Widget subscription object contains all subscription information including current data according to the widget type. Depending on widget type subscription object provides different data structures. For Latest values and Time-series widget types it provides the following properties:

- **datasources** - array of datasources used by this subscription and has the following structure:

```

datasources = [
  { // datasource
    type: 'entity', // type of the datasource. Can be "function" or "entity"
    name: 'name', // name of the datasource (in case of "entity" usually Entity_
↳name)
    aliasName: 'aliasName', // name of the alias used to resolve this particular_
↳datasource Entity
    entityName: 'entityName', // name of the Entity used as datasource
    entityType: 'DEVICE', // datasource Entity type (for ex. "DEVICE", "ASSET",
↳"TENANT", etc.)
    entityId: '943b8cd0-576a-11e7-824c-0b1cb331ec92', // entity identifier_
↳presented as string uuid.
    dataKeys: [ // array of keys (attributes or timeseries) of the entity used to_
↳fetch data
      { // dataKey
        name: 'name', // the name of the particular entity attribute/
↳timeseries
        type: 'timeseries', // type of the dataKey. Can be "timeseries",
↳"attribute" or "function"
        label: 'Sin', // label of the dataKey. Used as display value (for ex._
↳in the widget legend section)
        color: '#ffffff', // color of the key. Can be used by widget to set_
↳color of the key data (for ex. lines in line chart or segments in the pie chart).
        funcBody: "", // only applicable for datasource with type "function"_
↳and "function" key type. Defines body of the function to generate simulated data.
        settings: {} // dataKey specific settings with structure according to_
↳the defined Data key settings json schema. See "Settings schema section".
      },

```

(continues on next page)

```

        //...
    ]
},
//...
]

```

- **data** - array of latest data received in scope of this subscription and has the following structure:

```

data = [
  {
    datasource: {}, // datasource object of this data. See datasource structure_
    ↪above.
    dataKey: {}, // dataKey for which the data is held. See dataKey structure_
    ↪above.
    data: [ // array of data points
      [ // data point
        1498150092317, // unix timestamp of datapoint in milliseconds
        1, // value, can be either string, numeric or boolean
      ],
      //...
    ],
  },
  //...
]

```

For Alarm widget type it provides the following properties: * **alarmSource** - information about entity for which alarms are fetched and has the following structure:

```

alarmSource = {
  type: 'entity', // type of the alarm source. Can be "function" or "entity"
  name: 'name', // name of the alarm source (in case of "entity" usually Entity_
  ↪name)
  aliasName: 'aliasName', // name of the alias used to resolve this particular_
  ↪alarm source Entity
  entityName: 'entityName', // name of the Entity used as alarm source
  entityType: 'DEVICE', // alarm source Entity type (for ex. "DEVICE", "ASSET",
  ↪"TENANT", etc.)
  entityId: '943b8cd0-576a-11e7-824c-0b1cb331ec92', // entity identifier_
  ↪presented as string uuid.
  dataKeys: [ // array of keys indicating alarm fields used to display alarms data
    { // dataKey
      name: 'name', // the name of the particular alarm field
      type: 'alarm', // type of the dataKey. Only "alarm" in this case.
      label: 'Severity', // label of the dataKey. Used as display value (for_
      ↪ex. as a column title in the Alarms table)
      color: '#ffffff', // color of the key. Can be used by widget to set_
      ↪color of the key data.
      settings: {} // dataKey specific settings with structure according to_
      ↪the defined Data key settings json schema. See "Settings schema section".
    },
    //...
  ],
}

```

- **alarms** - array of alarms received in scope of this subscription and has the following structure:

```
alarms = [
  { // alarm
    id: { // alarm id
      entityType: "ALARM",
      id: "943b8cd0-576a-11e7-824c-0b1cb331ec92"
    },
    createTime: 1498150092317, // Alarm created time (unix timestamp)
    startTs: 1498150092316, // Alarm started time (unix timestamp)
    endTs: 1498563899065, // Alarm end time (unix timestamp)
    ackTs: 0, // Time of alarm acknowledgment (unix timestamp)
    clearTs: 0, // Time of alarm clear (unix timestamp)
    originator: { // Originator - id of entity produced this alarm
      entityType: "ASSET",
      id: "ceb16a30-4142-11e7-8b30-d5d66714ea5a"
    },
    originatorName: "Originator Name", // Name of originator entity
    type: "Temperature", // Type of the alarm
    severity: "CRITICAL", // Severity of the alarm ("CRITICAL", "MAJOR", "MINOR",
    ↪ "WARNING", "INDETERMINATE")
    status: "ACTIVE_UNACK", // Status of the alarm
                                // ("ACTIVE_UNACK" - active unacknowledged,
                                // "ACTIVE_ACK" - active acknowledged,
                                // "CLEARED_UNACK" - cleared unacknowledged,
                                // "CLEARED_ACK" - cleared acknowledged)
    details: {} // Alarm details object derived from alarm details json.
  }
]
```

For other widget types like RPC or Static subscription object is optional and does not contain necessary information.

Timewindow functions

Object with timewindow functions used to manage widget data time frame. Can be used by Time-series or Alarm widgets.

Function	Description
onUpdateTimewindow(startTimeMs, end-TimeMs)	This function can be used to update current subscription time frame to historical one identified by startTimeMs and endTimeMs arguments.
onResetTimewindow()	Resets subscription time frame to default defined by widget timewindow component or dashboard timewindow depending on widget settings.

Control API

Object that provides API functions for RPC (Control) widgets.

Function	Description
sendOneWay-Command(method, params, timeout)	Sends one way (without response) RPC command to the device. Returns command execution promise. method - RPC method name, string, params - RPC method params, custom json object, timeout - maximum delay in milliseconds to wait until response/acknowledgement is received.
sendTwoWay-Command(method, params, timeout)	Sends two way (with response) RPC command to the device. Returns command execution promise with response body in success callback.

Actions API

Set of API functions to work with user defined actions.

Function	Description
getActionDescriptors(actionSourceId)	Returns the list of action descriptors for provided actionSourceId
handleWidgetAction(\$event, descriptor, entityId, entityName)	Handles action produced by particular action source. \$event - event object associated with action, descriptor - action descriptor, entityId and entityName - current entity id and name provided by action source if available.

State Controller

Reference to Dashboard state controller providing API to manage current dashboard state.

Function	Description
openState(id, params, openRightLayout)	Navigates to new dashboard state. id - id of the target dashboard state, params - object with state parameters to use by the new state, openRightLayout - optional boolean argument to force open right dashboard layout if present in mobile view mode.
updateState(id, params, openRightLayout)	Updates current dashboard state. id - optional id of the target dashboard state to replace current state id, params - object with state parameters to update current state parameters, openRightLayout - optional boolean argument to force open right dashboard layout if present in mobile view mode.
getStateId()	Returns current dashboard state id.
getStateParams()	Returns current dashboard state parameters.
getStateParamsByStateId(id)	Returns state parameters for particular dashboard state identified by id.

Type parameters object

Object describing widget datasource parameters. It has the following properties:

```
return {
  maxDatasources: -1, // Maximum allowed datasources for this widget, -1 - unlimited
  maxDataKeys: -1 //Maximum allowed data keys for this widget, -1 - unlimited
}
```

Action sources object

Map describing available widget action sources to which user actions can be assigned. It has the following structure:

```
return {
  'headerButton': { // Action source Id (unique action source identifier)
    name: 'Header button', // Display name of action source, used in widget
    ↪ settings ('Actions' tab).
    multiple: true // Boolean property indicating if this action source supports
    ↪ multiple action definitions (for ex. multiple buttons in one cell, or only one
    ↪ action can be assigned on table row click.)
  }
};
```

Creating simple widgets

Below is the set of simple tutorials how to create minimal widgets of each type. In order to minimize the amount of code, the Angular framework will be used, on which Tempus UI is actually based. By the way, you can always use pure JavaScript or jQuery API in your widget code.

Latest Values widget

In the Widgets Bundle view click on the big “+” button in the bottom-right part of the screen and then click on the “Create new widget type” button. Click on the Latest Values button in the Select widget type popup. The Widget Editor will be opened pre-populated with the content of the default Latest Values template widget.

- Clear content of the CSS tab of “Resources” section.
- Put the following HTML code inside the HTML tab of “Resources” section:

```
<div flex layout="column" style="height: 100%;" layout-align="center stretch">
  <div>My first latest values widget.</div>
  <div flex layout="row" ng-repeat="dataKeyData in data" layout-align="space-around
  ↪ center">
    <div>{{dataKeyData.dataKey.label}}:</div>
    <div>{{dataKeyData.data[0][0] | date : 'yyyy-MM-dd HH:mm:ss'}}</div>
    <div>{{dataKeyData.data[0][1]}}</div>
  </div>
</div>
```

- Put the following JavaScript code inside the “JavaScript” section:

```
self.onInit = function() {

  self.ctx.$scope.data = self.ctx.defaultSubscription.data;

}
```

- Click on the Run button in the Widget Editor Toolbar in order to see the result in Widget preview section.

In this example the data property of subscription is assigned to the \$scope and become accessible within HTML template. Inside the HTML a special ng-repeat angular directive is used in order to iterate over available dataKeys datapoints and render corresponding latest values with their timestamps.

Time-Series widget

In the Widgets Bundle view click on the big “+” button in the bottom-right part of the screen and then click on the “Create new widget type” button. Click on the Time-Series button in the Select widget type popup. The Widget Editor will be opened pre-populated with the content of default Time-Series template widget.

- Clear content of the CSS tab of “Resources” section.
- Put the following HTML code inside the HTML tab of “Resources” section:

```
<div flex layout="column" style="height: 100%;">
  <div>My first Time-Series widget.</div>
  <md-tabs md-border-bottom>
    <md-tab ng-repeat="datasource in datasources track by $index" label="{
    ↪{datasource.name}}">
      <table style="width: 100%;">
        <thead>
          <tr>
            <th>Timestamp</th>
            <th ng-repeat="dataKeyData in datasourceData[$index]">{
            ↪{dataKeyData.dataKey.label}}</th>
          <tr>
            </thead>
            <tbody>
              <tr ng-repeat="data in datasourceData[$index][0].data track by
              ↪$index">
                <td>{{data[0] | date : 'yyyy-MM-dd HH:mm:ss'}}</td>
                <td ng-repeat="dataKeyData in datasourceData[$parent.$index]">
                ↪{{dataKeyData.data[$parent.$index][1]}}</td>
              </tr>
            </tbody>
          </table>
        </md-tab>
      </md-tabs>
    </div>
```

- Put the following JavaScript code inside the “JavaScript” section:

```
self.onInit = function() {
  self.ctx.$scope.datasources = self.ctx.defaultSubscription.datasources;
  self.ctx.$scope.data = self.ctx.defaultSubscription.data;

  self.ctx.$scope.datasourceData = [];

  var currentDatasource = null;
  var currentDatasourceIndex = -1;

  for (var i=0; i<self.ctx.$scope.data.length; i++) {
    var dataKeyData = self.ctx.$scope.data[i];
    if (dataKeyData.datasource != currentDatasource) {
      currentDatasource = dataKeyData.datasource;
      currentDatasourceIndex++;
      self.ctx.$scope.datasourceData[currentDatasourceIndex] = [];
    }
    self.ctx.$scope.datasourceData[currentDatasourceIndex].push(dataKeyData);
  }
}
```

(continues on next page)

```

}

self.onDataUpdated = function() {
    self.ctx.$scope.$digest();
}

```

- Click on the Run button in the Widget Editor Toolbar in order to see the result in Widget preview section.

In this example the datasources and data property of subscription is assigned to the \$scope and become accessible within HTML template. The datasourceData scope property is introduced to map datasource specific dataKeys data by datasource index for flexible access within HTML template. Inside the HTML a special ng-repeat angular directive is used in order to iterate over available datasources and render corresponding tabs. Inside each tab the table is rendered using dataKeys data obtained from datasourceData scope property accessed by datasource index. Each table renders columns by iterating over all dataKeyData objects and renders all available datapoints by iterating over data array of each dataKeyData to render timestamps and values. Note that in this code onDataUpdated function is implemented with a call to angular \$digest function necessary to perform new rendering cycle when new data is received.

Alarm widget

In the Widgets Bundle view click on the big “+” button in the bottom-right part of the screen and then click on the “Create new widget type” button. Click on the Alarm Widget button in the Select widget type popup. The Widget Editor will be opened pre-populated with the content of the default Alarm template widget.

- Put the following HTML code inside the HTML tab of “Resources” section:

```

<div flex layout="column" style="height: 100%;">
  <div>My first Alarm widget.</div>
  <table style="width: 100%;">
    <thead>
      <tr>
        <th ng-repeat="dataKey in alarmSource.dataKeys">{{dataKey.label}}</th>
      <tr>
      </tr>
    </thead>
    <tbody>
      <tr ng-repeat="alarm in alarms">
        <td ng-repeat="dataKey in alarmSource.dataKeys"
          ng-style="getAlarmCellStyle(alarm, dataKey)">
          {{getAlarmValue(alarm, dataKey)}}
        </td>
      </tr>
    </tbody>
  </table>
</div>

```

- Put the following JSON content inside the “Settings schema” tab of Settings schema section:

```

{
  "schema": {
    "type": "object",
    "title": "AlarmTableSettings",
    "properties": {
      "alarmSeverityColorFunction": {
        "title": "Alarm severity color function: f(severity)",
        "type": "string",
        "default": "if(severity == 'CRITICAL') {return 'red';} else if_
, (severity == 'MAJOR') {return 'orange';} else return 'green'; "

```

(continues on next page)

```

        }
    },
    "required": []
},
"form": [
    {
        "key": "alarmSeverityColorFunction",
        "type": "javascript"
    }
]
}

```

- Put the following JavaScript code inside the “JavaScript” section:

```

self.onInit = function() {
    self.ctx.$scope.alarmSource = self.ctx.defaultSubscription.alarmSource;

    var alarmFields = self.ctx.$scope.$injector.get('types').alarmFields;
    var $filter = self.ctx.$scope.$injector.get('$filter');

    var alarmSeverityColorFunctionBody = self.ctx.settings.alarmSeverityColorFunction;
    if (angular.isUndefined(alarmSeverityColorFunctionBody) || !
↪alarmSeverityColorFunctionBody.length) {
        alarmSeverityColorFunctionBody = "if(severity == 'CRITICAL') {return 'red';}
↪else if (severity == 'MAJOR') {return 'orange';} else return 'green';";
    }

    var alarmSeverityColorFunction = null;
    try {
        alarmSeverityColorFunction = new Function('severity',
↪alarmSeverityColorFunctionBody);
    } catch (e) {
        alarmSeverityColorFunction = null;
    }

    self.ctx.$scope.getAlarmValue = function(alarm, dataKey) {
        var alarmField = alarmFields[dataKey.name];
        if (alarmField) {
            var value = alarm[alarmField.value];
            if (alarmField.time) {
                return $filter('date')(value, 'yyyy-MM-dd HH:mm:ss');
            } else {
                return value;
            }
        } else {
            return alarm[dataKey.name];
        }
    }

    self.ctx.$scope.getAlarmCellStyle = function(alarm, dataKey) {
        var alarmField = alarmFields[dataKey.name];
        if (alarmField && alarmField == alarmFields.severity &&
↪alarmSeverityColorFunction) {
            var severity = alarm[alarmField.value];
            var color = alarmSeverityColorFunction(severity);
            return {
                color: color
            }
        }
    }
}

```

(continues on next page)


```

        };
    }
    return {};
}

self.onDataUpdated = function() {
    self.ctx.$scope.alarms = self.ctx.defaultSubscription.alarms;
}

```

- Click on the Run button in the Widget Editor Toolbar in order to see the result in Widget preview section.

In this example the alarmSource and alarms properties of subscription is assigned to the \$scope and become accessible within HTML template. Inside the HTML a special ng-repeat angular directive is used in order to iterate over available alarm dataKeys of alarmSource and render corresponding columns. The table rows are rendered by iterating over alarms array and corresponding cells rendered by iterating over dataKeys. The function getAlarmValue is fetching alarm value using special alarmFields constants obtained from types which is part of Tempus UI and accessed via Angular \$injector. The function getAlarmCellStyle is used to assign custom cell style for each alarm cell. In this example, we introduced new settings property called alarmSeverityColorFunction that contains function body returning color depending on alarm severity. Inside the getAlarmCellStyle function there is corresponding invocation of alarmSeverityColorFunction with severity value in order to get color for alarm severity cell. Note that in this code onDataUpdated function is implemented in order to update alarms property with latest alarms from subscription.

Static widget

In the Widgets Bundle view click on the big “+” button in the bottom-right part of the screen and then click on the “Create new widget type” button. Click on the Static Widget button in the Select widget type popup. The Widget Editor will be opened pre-populated with the content of default Static template widget.

- Put the following HTML code inside the HTML tab of “Resources” section:

```

<div flex layout="column" style="height: 100%;" layout-align="space-around stretch">
  <h3 style="text-align: center;">My first static widget.</h3>
  <md-button class="md-primary md-raised" ng-click="showAlert()">Click me</md-
  ↪button>
</div>

```

- Put the following JSON content inside the “Settings schema” tab of Settings schema section:

```

{
  "schema": {
    "type": "object",
    "title": "Settings",
    "properties": {
      "alertContent": {
        "title": "Alert content",
        "type": "string",
        "default": "Content derived from alertContent property of widget_
        ↪settings."
      }
    }
  },
  "form": [
    "alertContent"
  ]
}

```

(continues on next page)

(continued from previous page)

```
    ]  
  }  
}
```

- Put the following JavaScript code inside the “JavaScript” section:

```
self.onInit = function() {  
  
    self.ctx.$scope.showAlert = function() {  
        var alertContent = self.ctx.settings.alertContent;  
        if (!alertContent) {  
            alertContent = "Content derived from alertContent property of widget_  
↪settings.";  
        }  
        window.alert(alertContent);  
    };  
}
```

- Click on the Run button in the Widget Editor Toolbar in order to see the result in Widget preview section.

This is just a static HTML widget so there is no subscription data or special widget API was used. Only custom showAlert function was implemented showing an alert with the content of alertContent property of widget settings. You can switch to dashboard edit mode in Widget preview section and change value of alertContent by changing widget settings in the “Advanced” tab of widget details. Then you can see that the new alert content is displayed.

Integrating existing code to create widget definition

Below are some examples demonstrating how external JavaScript libraries or existing code can be reused/integrated to create new widgets.

Using external JavaScript library

In this example Latest Values gauge widget will be created using external gauge.js library.

In the Widgets Bundle view click on the big “+” button in the bottom-right part of the screen and then click on the “Create new widget type” button. Click on the Latest Values button in the Select widget type popup. The Widget Editor will be opened pre-populated with the content of default Latest Values template widget.

- Open Resources tab and click “Add” then insert the following link:

```
http://bernii.github.io/gauge.js/dist/gauge.min.js
```

- Clear content of the CSS tab of “Resources” section.
- Put the following HTML code inside the HTML tab of “Resources” section:

```
<canvas id="my-gauge"></canvas>
```

- Put the following JavaScript code inside the “JavaScript” section:

```
var canvasElement;  
var gauge;  
  
self.onInit = function() {
```

(continues on next page)

(continued from previous page)

```
canvasElement = $('#my-gauge', self.ctx.$container)[0];
gauge = new Gauge(canvasElement);
gauge.minValue = -1000;
gauge.maxValue = 1000;
gauge.animationSpeed = 16;
self.onResize();
}

self.onResize = function() {
  canvasElement.width = self.ctx.width;
  canvasElement.height = self.ctx.height;
  gauge.update(true);
  gauge.render();
}

self.onDataUpdated = function() {
  var value = self.ctx.defaultSubscription.data[0].data[0][1];
  gauge.set(value);
}
```

- Click on the Run button in the Widget Editor Toolbar in order to see the result in Widget preview section.

In this example API of the external JS library was used that become available after injecting the corresponding URL in Resources section. The value displayed was obtained from subscription data property for the first dataKey.

Using existing JavaScript code

Another approach of creating widgets is to use existing bundled JavaScript code. In this case, you can create own JavaScript class or Angular directive and bundle it into the Tempus UI code. In order to make this code accessible within the widget, you need to register corresponding Angular module or inject JavaScript class to a global variable (for ex. window object). Some of the Tempus widgets already use this approach. Take a look at the widget.service.js. Here you can find how some bundled classes or modules are registered for later use in Tempus widgets. For example “Timeseries - Flot” widget (from “Charts” Widgets Bundle) uses TbFlot JavaScript class which is injected as window property inside widget.service.js:

```
...
import TbFlot from '../widget/lib/flot-widget';
...

$window.TbFlot = TbFlot;
...
```

Another example is “Timeseries table” widget (from “Cards” Widgets Bundle) that uses Angular directive tb-timeseries-table-widget which is registered as dependency of tempus.api.widget Angular module inside widget.service.js. Thereby this directive becomes available for use inside the widget template HTML.

```
...
import tempusTimeseriesTableWidget from '../widget/lib/timeseries-table-widget';
...

export default angular.module('tempus.api.widget', ['oc.lazyLoad', tempusLedLight,
  ↪tempusTimeseriesTableWidget,
```

(continues on next page)

...

Widget code debugging tips

The most simple method of debugging is Web console output. Just place `console.log(...)` function inside any part of widget JavaScript code. Then click Run button to restart widget code and observe debug information in the Web console.

Another and most effective method of debugging is to invoke browser debugger. Put `debugger;` statement into the place of widget code you are interested in and then click Run button to restart widget code. Browser debugger (if enabled) will automatically pause code execution at the debugger statement and you will be able to analyze script execution using browser debugging tools.

While developing client-side js, if you need to access the `document` or `window` objects, add this comment at the end of the line in your code referencing the object:

```
//eslint-disable-line
```

This will tell the linter to ignore the line. Only do this if you have to. In most cases, use the angular `$window` object instead.

Custom Widgets

See documentation on custom widgets below:

3D Trajectory Viewer

This widget displays witsml trajectory data in a 3D chart.

Usage

This widget expects data to exist in the first data source of the widget.

It will look for trajectory data objects with the following fields:

`azi` - azimuth in degrees `incl` - incline in degrees `md` - measured depth in any units. The metric of this measure will be used in the chart.

Help

If you need any help, please reach out to [Randy Pitcher](#).

UI Development Guide

Running UI container in hot redeploy mode

By default, the Tempus Cloud UI is served on port 8080. However, when developing the UI, developing in hot redeploy mode is significantly more efficient.

To start the UI container in hot redeploy mode you will need to install node.js first. Once node.js is installed you can start container by executing next command:

```
cd ${TEMPUS_WORK_DIR}/ui
mvn clean install -P npm-start
```

This will launch a server that will listen on 3000 port. All REST API and websocket requests will be forwarded to 8080 port.

Running server-side container

To start server-side container there are 2 options:

- Run the main method of `com.hashmapinc.server.TempusServerApplication` class that is located in the application module from the IDE.
- Start the server from command line as a regular Spring boot application:

```
cd ${TB_WORK_DIR}
java -jar application/target/tempus-${VERSION}-boot.jar
```

- Run the TempusDevEnvironment as described in the Developer Quick Start

Dry run

Navigate to <http://localhost:3000/> or <http://localhost:8080/> and login into Tempus Cloud using demo data credentials:

login `demo@hashmapinc.com` **password** `tenant`

Make sure that you are able to login and everything has started correctly.

Kubeless Client Guide

Introduction

Kubeless is a serverless framework on top of Kubernetes, which provides a way to deploy & run “functions” on K8s cluster.

Function can be any source code written in PHP, Java 1.8, Node.js, Golang, .Net, Ruby or Python, having a method which takes event and context as parameters. Check this [link](#) for more details on function.

Triggers are associated with functions to invoke them on specific event, examples are `HttpTrigger`, `KafkaTrigger` and `CronTrigger` where `Http` call will trigger execution of function having `HttpTrigger`, `KafkaTrigger` will invoke function once data is published to specified topic and `CronTrigger` will be invoked on specific schedule.

Kubeless java-client provides APIs to deploy and perform other operations on functions & triggers over k8s cluster.

Below is detailed usage examples.

Usage

API Client

Kubeless java-client provides `ApiClient` from Kubernetes Java client to communicate with K8s API server.

ApiClient can be used depending upon, from where you are going to deploy your function.

- If you will be deploying functions from code running in Pod from same cluster use below code

```
import io.kubernetes.client.util.Config;
ApiClient client = Config.fromCluster();
```

- From outside cluster

```
import io.kubernetes.client.util.Config;
String filename = "/path/to/kubeconfig";
ApiClient client = Config.fromConfig(filename)
```

If you use Config.defaultClient() it attempts to auto detect where the code is being used and creates a client object accordingly. It first looks for a kubeconfig file in environment variable \$KUBECONFIG, then falls back to path \$HOME/.kube/config. If that doesn't work, it falls back to an in-cluster mode by detecting a default service account. Lastly, if none of this works, it attempts to connect to the server at <http://localhost:8080>

Once ApiClient is built set it as Default client to use for further requests as

```
Configuration.setDefaultApiClient(client);
```

Function API

KubelessV1beta1FunctionApi contains APIs for communicating with k8s cluster with Kubeless. It contains APIs to

- list all functions
- fetch specific function by name
- delete specific function by name
- create new function
- update existing function

To create a new instance of FunctionApi client you need to pass "namespace" within which functions will be created

```
KubelessV1beta1FunctionApi functionApi = new KubelessV1beta1FunctionApi("default");
```

This will pick up ApiClient created in earlier step from Configuration object.

Below is complete code that will be used to list all functions from default namespace

```
import com.hashmapinc.kubeless.apis.KubelessV1beta1FunctionApi;
import com.hashmapinc.kubeless.models.V1beta1Function;
import com.hashmapinc.kubeless.models.V1beta1FunctionList;
import com.squareup.okhttp.Call;
import com.squareup.okhttp.Response;
import io.kubernetes.client.ApiClient;
import io.kubernetes.client.ApiException;
import io.kubernetes.client.Configuration;
import io.kubernetes.client.JSON;
import io.kubernetes.client.util.Config;
import java.io.IOException;

public class TestKubelessClient {

    public static void main(String[] args) throws IOException, ApiException {
```

(continues on next page)

(continued from previous page)

```
ApiClient client = Config.defaultClient();
Configuration.setDefaultApiClient(client);

KubelessV1beta1FunctionApi functionApi = new KubelessV1beta1FunctionApi(
↪ "default");
Call listFunctionsCall = functionApi.listFunctionsCall();

Response response = listFunctionsCall.execute();

JSON json = new JSON();
V1beta1FunctionList functionList = json.deserialize(response.body().string(), ↪
↪ V1beta1FunctionList.class);

    for (V1beta1Function item : functionList.getItems()) {
        System.out.println(item.getMetadata().getName());
    }
}
```

As you can see JSON object is used here to deserialize a response from K8s API server, which internally uses Gson registered with few Data converters like Date time.

All the models which are used as Request and Response needs to be Serialized and Deserialized, respectively, using JSON instance only.

Similarly API client can be used to create a function, which requires a target V1beta1Function instance

```
public Call createFunctionCall(V1beta1Function function) throws ApiException
```

Sample of creation of function object is as below

```
V1beta1AbstracType<V1beta1FunctionSpec> function = new V1beta1Function()
    .metadata(new V1ObjectMeta()
        .name("functionName")
        .namespace("default"))
    .spec(new V1beta1FunctionSpec()
        .checksum("sha256 of function")
        .function("function string or url")
        .dependencies("dependency file file as string if needed")
        .timeout("180")
        .handler("functionFilename.methodName")
        .runtime("Java1.8"));
KubelessV1beta1FunctionApi functionApi = new KubelessV1beta1FunctionApi("default");
Call createFunctionsCall = functionApi.createFunctionCall((V1beta1Function) function);

Response response = createFunctionsCall.execute();

JSON json = new JSON();
V1beta1Function functionCreated = json.deserialize(response.body().string(), ↪
↪ V1beta1Function.class);
```

Trigger API

Once function is created next step is to add a trigger which will trigger an execution of function. Let's look at an example of adding Kafka trigger

To create a Kafka trigger we need to provide few labels to match with a function to which we want to add a trigger like below:

```
V1beta1AbstractType<V1beta1KafkaTriggerSpec> trigger = new V1beta1KafkaTrigger()
    .metadata(new V1ObjectMeta()
        .name("TestKafkaTrigger"))
    .spec(new V1beta1KafkaTriggerSpec()
        .topic("test-topic")
        .labelSelector(new V1LabelSelector()
            .putMatchLabelsItem("created-by", "kubeless")
            .putMatchLabelsItem("function", "functionName")));
```

Then using Kubeless Trigger API client we can send a request to deploy this trigger on K8s cluster as below,

```
KubelessV1beta1KafkaTriggerApi triggerApi = new KubelessV1beta1KafkaTriggerApi(
    ↪ "default");
Call kafkaTriggerCall = triggerApi.createKafkaTriggerCall((V1beta1KafkaTrigger) ↪
    ↪ trigger);
Response result = kafkaTriggerCall.execute();
```

3.10 Gateway

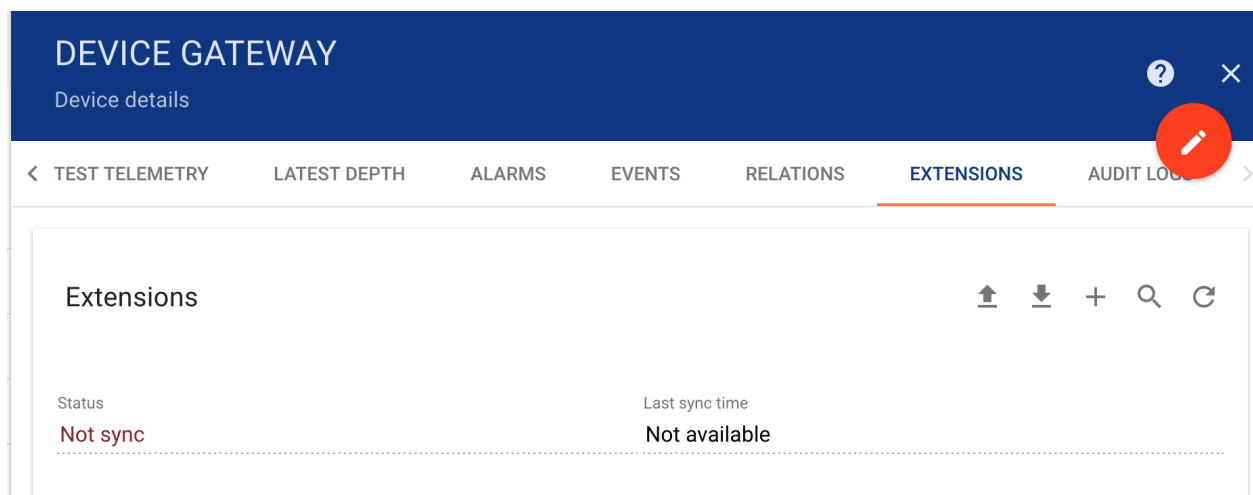
This guide describes how to configure device of type gateway to connect to Tempus-gateway. Tempus-gateway is a solution that allows you to integrate IoT devices connected to legacy and third-party systems with Tempus Cloud.

Gateway Extension

This section will help you to configure various gateway extensions.

How to add extension

Click on device of type gateway. Go to extension tab at top-right corner by clicking on '>'. Then, Click on the “+” button at the top-right corner of the page.



WITSML Extension

This section will help you to configure WITSML gateway extension.

Add WITSML Extension

After clicking on add extension, add name and type of extension as shown below:

Add extension

Extension id *

WITSML DEMO

HTTP

MQTT

OPC UA

WITS

WITSML

ioWell08	DEFAULT
ioWell09	DEFAULT

WITSML Extension Configuration

Following WITSML properties can be configure:

- **host** - WITSML server url (without http or https protocol)
- **port** - WITSML server port to connect
- **lowFrequencyInSeconds** - scanning interval for low frequency witsml objects
- **highFrequencyInMillis** - scanning interval for high frequency witsml objects (growing objects like log, trajectory)
- **username / password** - access credentials
- **version** - WITSML version (currently two versions are supported, 1.3.1.1 & 1.4.1.1)
- **objectTypes** - type of objects to be fetch
- **wellStatus** - status of well to be fetch
- **wellIdToBeScanned** - scanning witsml objects for single well id (Default is empty i.e. to scan all wells and related witsml objects)

Here is the sample configuration:

Configuration

Servers

1.

Host *

https://witsml.host.com/index.aspx

Port *

443

Low Frequency in seconds (Well, Wellbore, Rig) *

60

High Frequency in millis (Logs, Mudlogs, Trajectory, ...)

5000

Timeout in milliseconds *

5000

Well Status *

Active

Username *

username

Password *

.....

WITSML Version *

1.4.1.1

Object Types *

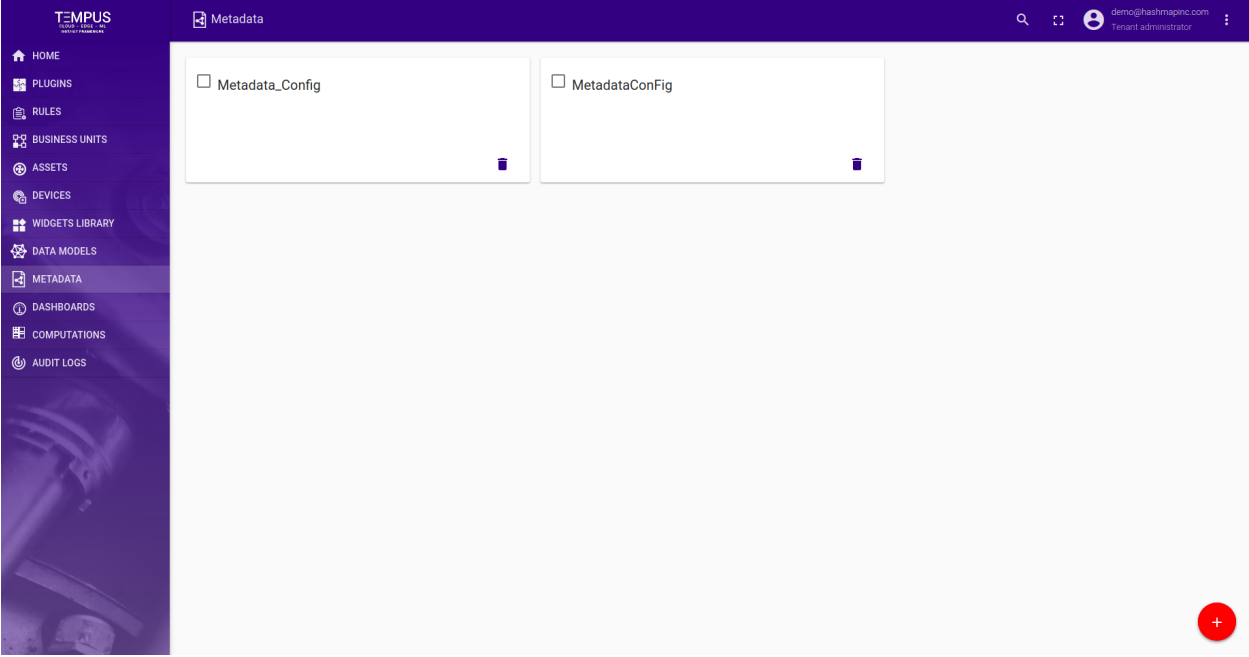
Wellbore, Logs, Messages, Rig, Trajectory

SAVE

CANCEL

3.11 Metadata

This guide describes how to configure metadata. In metadata User can set source and sink type. User can add queries in metadata. Metadata is microservice written in HashMap Analytic framework.



Add Metadata

Add Metadata?

Name *

MetadataConFig

Source Type *

JDBC

Database URL *

jdbc://postgres:postgres

Username *

Postgres

Password *

.....

Sink Type *

REST

URL *

http://localhost:8000

ADD

CANCEL

User can create metadata configuration by entering information. This includes:

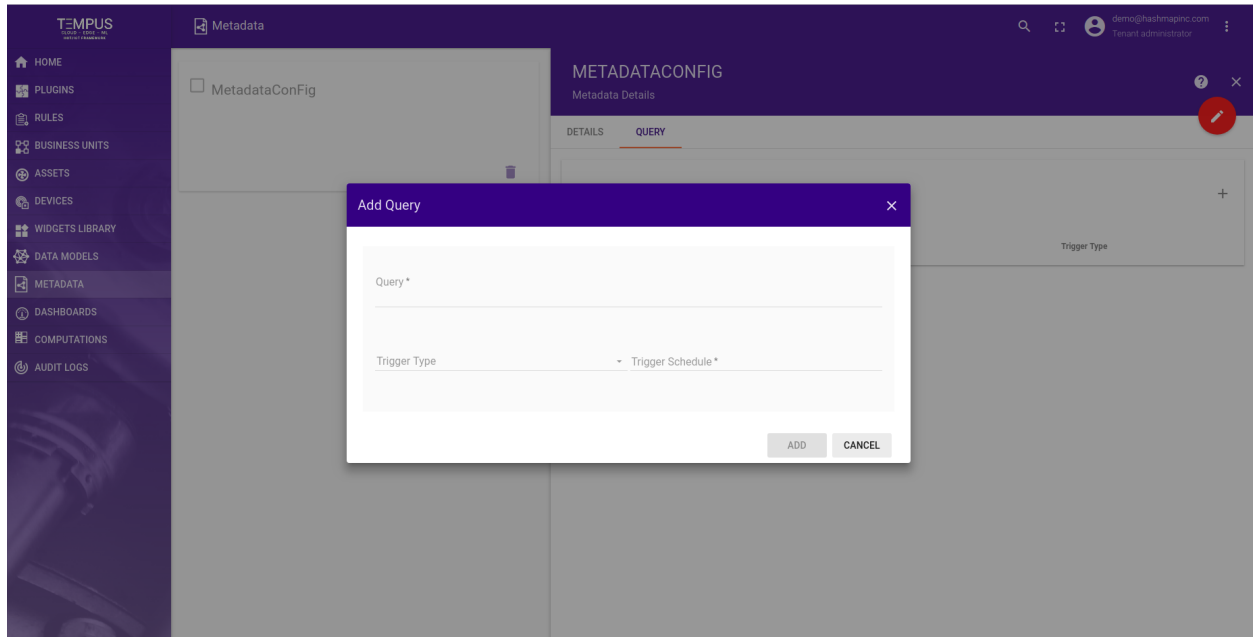
- **Metadata Name** - The name of the metadata configuration.
- **Source Type** - Metadata will be pull from source.User need to select source type.User can select JDBC.Once source type is selected user can add source detail which includes database url,username and password.
- **Sink Type** - Pulled metadata will be push in set sink details.User need to select sink type and set url of sink type.

Edit Metadata

The screenshot displays the 'TEMPUS' application interface. On the left is a dark sidebar with a navigation menu including: HOME, PLUGINS, RULES, BUSINESS UNITS, ASSETS, DEVICES, WIDGETS LIBRARY, DATA MODELS, METADATA (highlighted), DASHBOARDS, COMPUTATIONS, and AUDIT LOGS. The main content area is titled 'Metadata' and contains a list with one item, 'MetadataConFig'. To the right, a 'METADATACONFIG' modal window is open, showing 'Metadata Details'. The form includes the following fields: 'Name' (filled with 'MetadataConFig'), 'Source Type' (a dropdown menu currently showing 'JDBC'), 'Database URL' (filled with 'jdbc://postgres:postgres'), 'Username' (filled with 'Postgres'), 'Password' (masked with dots), a 'TEST CONNECTION' button, 'Sink Type' (a dropdown menu currently showing 'REST'), and 'URL' (filled with 'http://localhost:8000'). The top right of the application shows a user profile for 'demo@hashmapinc.com' with the role 'Tenant administrator'.

User can edit metadata configuration,delete metadata configuration and also Test connection with source.

Add Query



User can add queries by clicking on Query tab in detail sidenav. Query includes following:

- **Query** - Valid query.
- **Trigger Type** - By default CRON trigger type is supported in trigger type.
- **Trigger Schedule** - Cron expression can be set as trigger schedule.

User can edit and delete added queries.

3.12 Indices and tables

- genindex
- modindex
- search