Tempus Edge Documentation

Release 0.1

Randy Pitcher

May 25, 2018

Contents:

TEMPUS EDGE
ANALYTICS INTELLIGENCE
AT THE EDGE

Tempus Edge is an edge framework for deploying microservices (edge applications) to the edge.

Getting Started Tutorial

This document discusses how IoFog can be used to filter data at the edge. The IoFog platform provides a way to run code remotely.The code/logic which we want to run is packaged as small micro services which are provided as a docker images.IoFog provide interface to communicate between these micro services. Flow of data is assume to be starting from a microservices which generate time series data. This data is then filtered using Tempus Filter service. Using MQTT service the filtered data is sent to ThingsBoard server.

1.1 Features

Tempus IoFog services has following feature:

- Create Micro-service for creating simulated timeseries data.
- Create a filter service which filter out data at source.
- Send only filtered data to the ThingsBoard.

1.2 Requirements

- JDK 1.8 at a minimum
- Maven 3.1 or newer
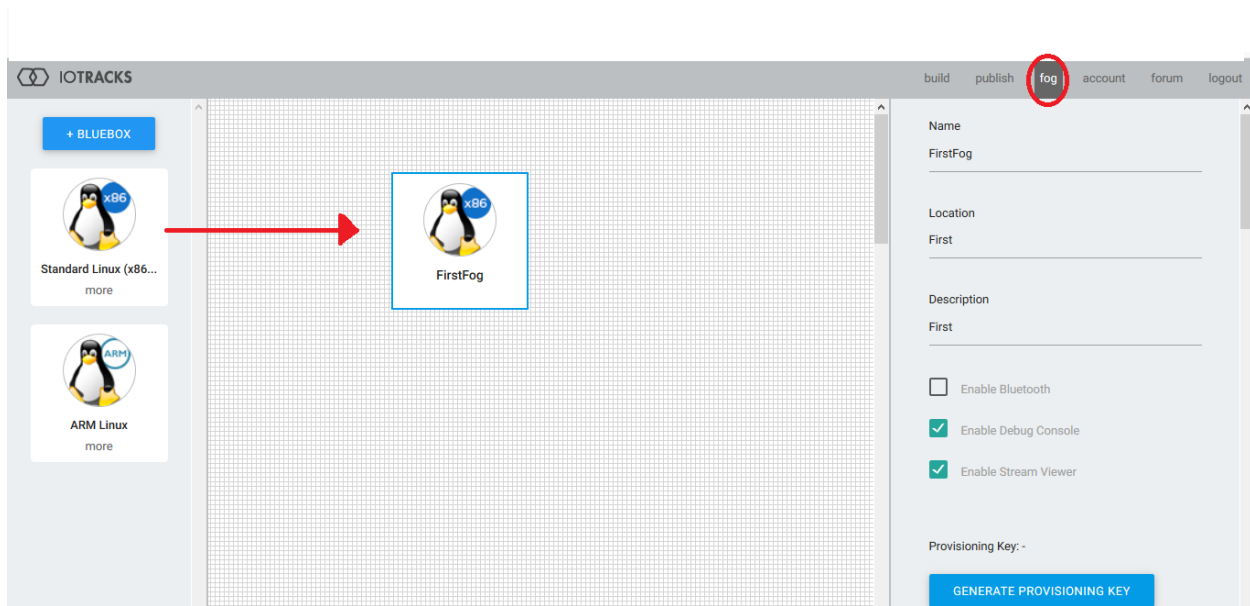- Git client (to build locally)

1.3 Install IoFog

Create an account with IOTRACKS to use IoFog service here

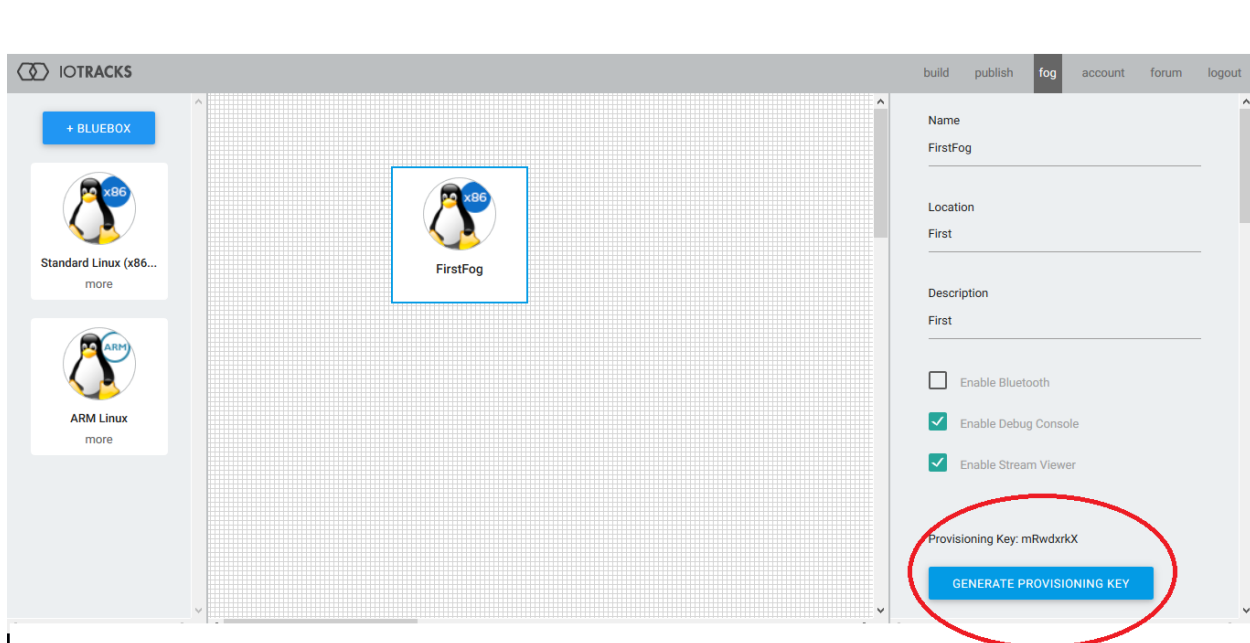*Next, setup your docker-based development fog here.*

1.4 Provision the Fog

Login to IoFog to access tracks and fogs.

After login go to the fog tab drag a fog instance on fog page.And click on new fog instance.

Click on new fog instance. On right hand you will see properties of fog.

Press the generate key to get id of the fog for provisioning.

Go to your Linux command line, type `sudo iofog provision ABCDWXYZ` and replace the ABCDWXYZ with

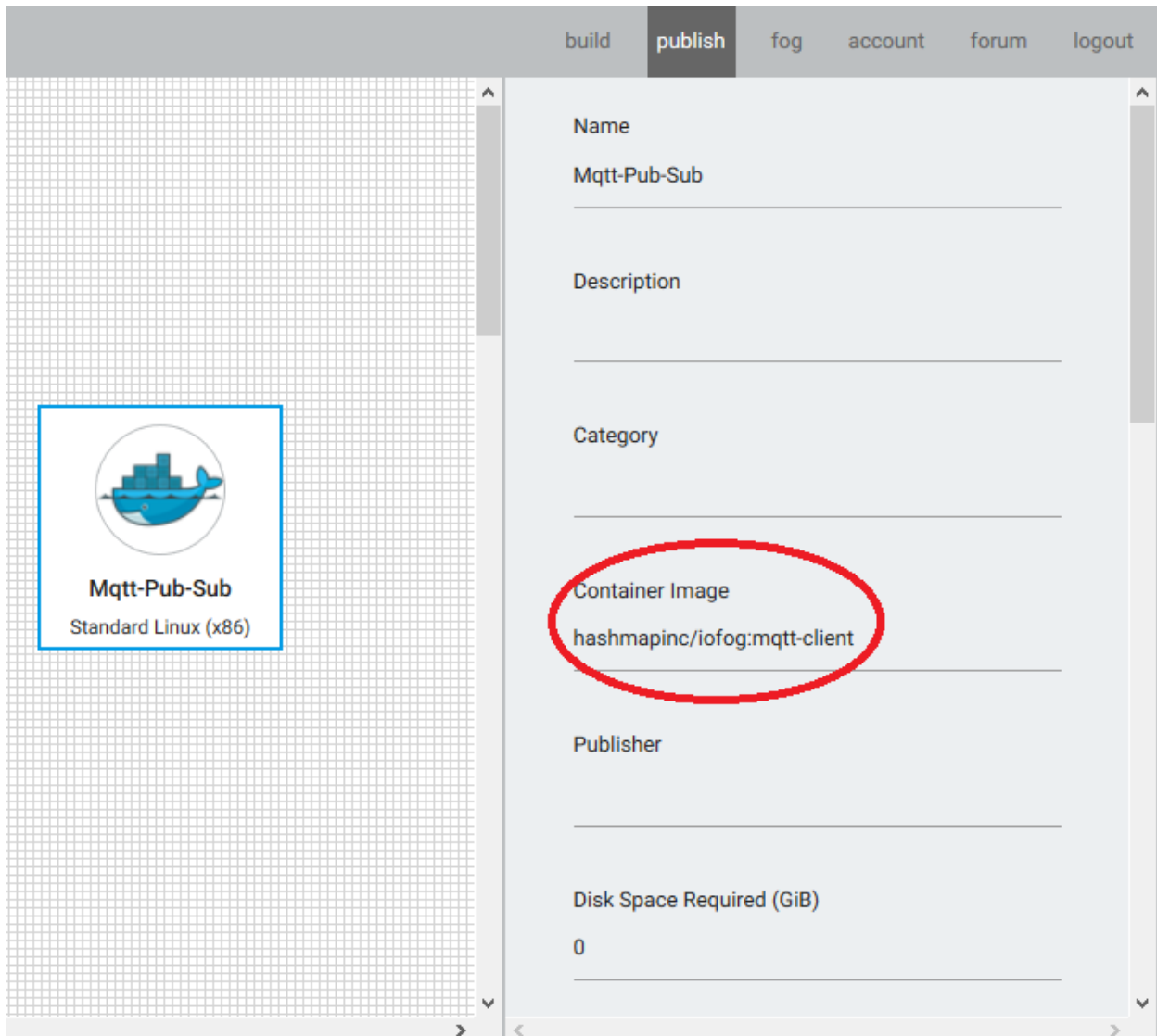your provisioning key (it is case sensitive) and verify the results

1.5 Creating edge applications

IoFog provides an SDK in four languages to create edge applications:

1. Java

2. Python

3. NodeJS

4. Go

1.6 Publishing edge applications

1. Log in to the ioAuthoring page.

2. Use the "Publish" menu to access the publishing portal.

3. Drag and drop the proper hardware element to the main area. Make sure the element matches the hardware you want your applicaiton to run on.

4. Add Name to your Element.

5. Add Container Image URL. It is the path to repository where the container is posted. Be sure to enter your container image string properly.

Click here to see in detail how to create and publish microservices.

1.7 Configuring Services

1. **Time Series Service:** No Configuration Needed.

2. **Json Filter Service:** Basic building block of the service has 3 terms:

```
{
    "term": "value.density",
    "OP" : "GTE",
    "value":    0
}
```

term: The term in json on which filter needs to be applied. The child term can be accessec using dot.

Example:

```
{
"range": {
    "start":0,
    "end":100
}
}
```

if you want to filter on "start" mention "range.start" in the term.

OP: Operation is what operation you need to perform on the the term.Allowed operation

- LEQ : less than equal
- GEQ : greater than equal
- LT : less than
- GT : greater tham
- EQ : equals
- NEQ : not equal

value: Value with which operation on term need to be done

Filters are furthur divided into

- String Filters
- Double Filters
- Boolean Filters

```
{
"DOUBLE":{
    "term": "value.density",
    "OP" : "GTE",
    "value":   0
}
}
```

This tells filter type of term.

To create complex query **AND** and **OR** Filters are also provided.

```
{
    "EXP1":"..",
    "EXP2":".."
}
```

Both EXP1/EXP2 both can be again AND ,OR filter or and of String,Double or Boolean Filter

```
{
    "OR": {
        "EXP1": {
            "AND": {
                "EXP1": {
                    "DOUBLE": {
                        "term": "value.density",
                        "OP": "GTE ",
                        "value": 0
                    }
```

```
                },
                "EXP2": {
                    "DOUBLE": {
                        "term": "value.density",
                        "OP": "LTE",
                        "value": 1
                    }
                }
            }
        },
        "EXP2": {
            "STRING": {
                "term": "value.TYPE.id",
                "OP": "NEQ",
                "value": "TEST"
            }
        }
    }
}
```

3. **MQTT-Service** (iofog default)

   For MQTT service you need to define:

   - Publishers

   - Broker

   - User of device in thingsboard

```
{
    "subscriptions": [{
        "topic": "v1/devices/me/telemetry",
        "qos": 2
    }],
    "publishers": [{
        "topic": "v1/devices/me/telemetry",
        "qos": 2
    }],
    "broker": {
        "host": "192.168.1.183",
        "port": 1883
    },
    "user": {
        "username": "ioFogToken",
        "password": " "
    }
}
```

1.8 Usage

1. Publish the 3 tempus services as discussed above.

2. Configure the services.

3. Create a track with services as shown below.
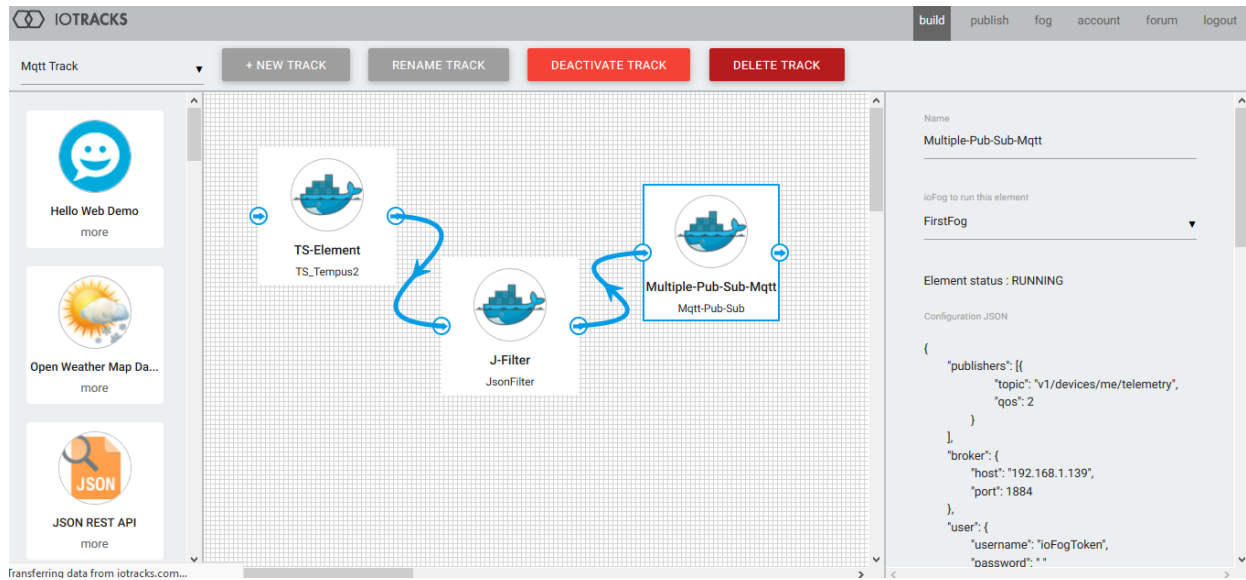
4. You can check data flowing into ThingsBoard.

1.9 License

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at:

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

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

Edge Applications

Tempus Edge currently supports the following edge applications:

JSON Translator	service for converting back and forth between JSON and Tempus Edge messages.
Golang MQTT Client	MQTT client implemented in golang.
Python MQTT Client	Python implementation of an MQTT client.
OPC Client	OPC client for monitoring and publishing to OPC UA servers.
OPC Tag Filter	service for searching OPC UA servers for monitorable tags.
Timeseries Generator	generates changing timeseries data.
Track Manager	core track management service.

2.1 JSON Translator

This edge application provides interoperability between Tempus Edge applications that use a protobuf message serialization and default IoFog applications that serialize to JSON.

This scala application accepts incoming ioFog messages. If the incoming message is a JSON, a protobuf wrapper is created around the JSON and the message is then written to the ioFog message stream. If the incoming message is a tempus edge protobuf message, it is deserialized and converted to a JSON string that is written to the ioFog message stream.

Because Tempus Edge messages also contain 2 descriptive bytes at the beginning of incoming ioFog messageContent byte arrays, the Translator will treat all protobuf'd JSON strings as messages using the *DATA* protocol and of type *JSON*.

2.1.1 Expected IoFog Config

There is no expected config for this edge app.

2.1.2 Building

Build the project and docker image using:

```
mvn package
```

2.1.3 Usage

Deploy this image to dockerhub and publish to ioFog. From there, the image can be used in your tracks. Put it in between default IoFog services (like the stream viewer or the JSON REST API) and Tempus Edge applications (like the track-manager).

One use case would be to connect track-manager outputs to the json-translator and feed translations to the JSON REST API element so configuration events could be monitored by an external web GUI or dashboard application.

2.1.4 Help

If you need any help, please reach out to Randy Pitcher.

2.2 Golang MQTT Client

This edge application is designed to act as a 2 way client between ioFog messages and MQTT messages. It is responsible for:

- accepting incoming MQTT messages and converting them to outgoing ioFog messages
- accepting incoming ioFog messages and converting them to outgoing MQTT messages

At this time, MQTT client is the core communication service between Tempus Edge and Tempus Cloud. Because of the way configuration is currently done, each Tempus Edge track will only support 1 MQTT client, which will always need to be Tempus Cloud.

In the future, this edge application may be split into the following 2 applications:

- a dedicated Tempus Cloud client
- a standard MQTT client

This will also result in separate TrackConfig entries for MqttConfig and TempusConfig based on the underlying MqttConfig.proto message definition.

2.2.1 Building

Make sure your `$GOPATH` contains the `src` directory when you're ready to build.

Build the project and docker image using:

```
make docker
```

2.2.2 Usage

Build and deploy this image to dockerhub using:

```
make deploy
```

Publish the image to ioFog. From there, the image can be used in your tracks.

Any configurations you need should be provided through the track-manager.

2.2.3 Supported Messages

The client supports the following incoming tempus edge messages:

Protocol	Type	MQTT-CLIENT Behavior
`CONFIG`	`UPDATE`	updates current configs
`DATA`	`JSON`	attempts to convert the json string value into a `MQTT` message type then sends to the current broker.
`DATA`	`MQTT`	uses the `qos`, `topic`, and `payload` fields of the message to send an mqtt message to the current broker.
`DATA`	`OPC`	attempts to convert this message into an `MQTT` message type. `qos` is set to 2, `topic` is set to the value of the `deviceName` field, and `payload` is the byte array resulting from an `fmt.Sprintf` of the `value` oneof field where the formatting is based on the type of the oneof value. No set value will send an empty byte array as the `payload`

The client converts all messages from the current broker into `DATA MQTT` tempus edge messages and sends them to the iofog message bus.

`payloads` from the broker are not altered and are sent as the byte arrays that they are.

2.2.4 Help

If you need any help, please reach out to Randy Pitcher.

2.3 Python MQTT Client

This edge application is designed to act as a 2 way client between ioFog messages and MQTT messages. It is responsible for:

- accepting incoming MQTT messages and converting them to outgoing ioFog messages
- accepting incoming ioFog messages and converting them to outgoing MQTT messages

At this time, MQTT client is the core communication service between Tempus Edge and Tempus Cloud. Because of the way configuration is currently done, each Tempus Edge track will only support 1 MQTT client, which will always need to be Tempus Cloud.

In the future, this edge application may be split into the following 2 applications:

- a dedicated Tempus Cloud client
- a standard MQTT client

This will also result in separate TrackConfig entries for MqttConfig and TempusConfig based on the underlying MqttConfig.proto message definition.

2.3.1 Building

Build the project and docker image using:

```
make
```

2.3.2 Supported Messages

The client supports the following incoming tempus edge messages:

Pro-to-col	Type	MQTT-CLIENT Behavior
`CONFIG`	`UPDATE`	updates current configs
`DATA`	`JSON`	attempts to convert the json string value into a `MQTT` message type then sends to the current broker.
`DATA`	`MQTT`	uses the `qos`, `topic`, and `payload` fields of the message to send an mqtt message to the current broker.
`DATA`	`OPC`	attempts to convert this message into an `MQTT` message type. `qos` is set to 2, `topic` is set to the value of the `deviceName` field, and `payload` is the byte array resulting from an `fmt.Sprintf` of the `value` oneof field where the formatting is based on the type of the oneof value. No set value will send an empty byte array as the `payload`

The client converts all messages from the current broker into `DATA MQTT` tempus edge messages and sends them to the iofog message bus.

`payload`s from the broker are not altered and are sent as the byte arrays that they are.

2.3.3 Help

If you need any help, please reach out to Randy Pitcher.

2.4 OPC Client

This edge application uses the track's `config.pb` file to connect to an OPC UA server.

It performs 2 main functions:

- subscribes to OPC nodes and writes value changes to the IoFog message bus.
- writes OPC values when a new `DATA OPC` message arrives.

On startup and when new `CONFIG UPDATE` messages arrive, this edge application will use the latest `config.pb` to refresh its OPC UA subscriptions.

The latest values for each subscribed OPC node will be written to the IoFog message bus each time subscriptions are updated, regardless of whether the values are actually new or not.

2.4.1 Building

Build the edge application and docker image using:

```
mvn package
```

2.4.2 Help

If you need any help, please reach out to Randy Pitcher.

2.5 OPC Tag Filter

This edge application uses the track's `config.pb` file to connect to an OPC UA server and search for subscriptions.

Subscriptions are defined by any OPC node that matches the `whitelist` and does not match the `blacklist` regex arrays in the `OpcConfig` of the `TrackConfig`.

The matching subscriptions are then assigned a `deviceName` based on a `deviceMap` structure in the `OpcConfig`.

On startup and when new `CONFIG UPDATE` messages arrive, this edge application will use the latest `config.pb` to find new subscriptions and will send them to the Track Manager.

2.5.1 Building

Build the edge application and docker image using:

```
mvn package
```

2.5.2 Help

If you need any help, please reach out to Randy Pitcher.

2.6 Timeseries Generator

This edge application generates test timeseries data in `JSON` format continuously.

It requires no configuraiton.

2.6.1 Usage

Simply add this element to a track and attach it to a desired endpoint. Data will begin flowing immediately.

2.7 Track Manager

This edge application is core to Tempus Edge. It is the "brain" that allows for seamless control of individual tracks within a fog from both ioFog and Tempus.

This scala application takes initial configs from ioFog and stores them as a protobuf file.

This file is accessible to all track elements through a shared volume (mounted at `/iofog/config/<YOUR_TRACK_NAME>` on the host).

As updated configurations come from ioFog and Tempus (through the mqtt client element), Track Manager updates the protobuf config file and sends newConfig messages to the iofog message bus. Other elements in the track are then responsible for handling new configs in response to this newConfig message. Most often, this will involve loading configs from the protobuf file.

2.7.1 Expected IoFog Config

The following config JSON shows the expected format for an IoFog Container configuration. Each root-level field is optional:

```
{
  "trackMetadata": {
    "trackName" : "my-track",
    "trackId": 0,
    "metadata": "{myMetadata: meta}"
  },

  "mqttConfig": {
    "subscriptions": [{
      "topic": "v1/devices/me/telemetry",
      "qos": 2
    }],
    "publishers": [{
      "topic": "v1/devices/me/telemetry",
      "qos": 2
    }],
    "broker": {
      "host": "192.168.1.183",
      "port": 1883
    },
    "securityType": 0,
    "user": {
      "username": "ioFogToken",
      "password": " "
    }
  },

  "opcConfig": {
    "endpoint": "opc.tcp://myHost:8080/myEndpoint",
    "securityType": 0
  }
}
```

The following Volume Mappings config is expected for this element:

```
{
  "volumemappings": [
    {
      "hostdestination": "/iofog/config/<YOUR_TRACK_NAME>",
      "containerdestination": "/mnt/config",
      "accessmode": "rw"
    }
  ]
}
```

Do not forget to update <YOUR_TRACK_NAME> and make sure that all elements in this track have this same volume mapping config to ensure configs can be shared between containers.

2.7.2 Building

Build the project and docker image using:

```
mvn package
```

2.7.3 Usage

Deploy this image to dockerhub and publish to ioFog. From there, the image can be used in your tracks.

Any configurations you need to make available to your track elements should be provided through ioAuthoring to this element.

2.7.4 Help

If you need any help, please reach out to Randy Pitcher.

Developer Documentation

Welcome, developers!

To make it easy to use and contribute to Tempus Edge, we have provided the following documentation:

Repository Standards	describe general guidelines for the Tempus Edge repository.
Development Fog	describes how to create a docker-based development fog environment.
Track Configuration	describes how tracks are configured.
Tempus Edge Messaging	describes how to use Tempus Edge messages.

3.1 Repository Standards

- Application folders should be all lower case with '-' to separate words; no underscores, no spaces, no camelCase.

- Application folders should contain Dockerfiles defining the applicaiton image.

    - Docker images should be named using the pattern `hashmapinc/ tempus-edge-<APPLICATION-NAME>:<VERSION>`

- Application folders should contain a `README.md` describing the application and usage. This includes instructions for building and running both the Docker image and the source code.

- Build files / directories should not be committed to the repository. This means no `.jar` files, no `\*.pyc` files, and no mvn `**/target/` build directories.

- All commits should result in buildable code. The code may be buggy, but please do not commit unbuildable code if you can help it.

NOTE: Please submit pull requests to udpate these rules if you think something else makes more sense!

3.2 Development Fog

For development purposes, it is often easier to run iofog in a docker container locally.

To run this docker container, clone this repository, cd into this directory (development-fog), and execute the following commands

```
mvn package
sh startFog.sh
```

3.2.1 Development Notes

This container will run a full docker instance inside itself. This may have security implications and is therefore not suitable for anything other than local development on a dev machine.

To make this work, 3 special steps were necessary:

1. The Dockerfile must include `VOLUME /var/lib/docker` in order for the internal docker daemon to be able to create internal containers.

2. The container must be ran with the `--privileged` argument to allow the docker daemon to start.

3. The docker service must be started manually. This is acheived by the [entrypoint.sh](entrypoint/entrypoint.sh) script that is executed at container launch.

3.2.2 Help

Please reach out to Randy Pitcher if you have any questions.

3.3 Track Configuration

Configuration for each application will be managed by an instance of the `track-manager` service.

Configurations will be made available through a shared `volume` in each application's docker container.

3.3.1 `config.pb` Files

Configurations are stored as `protobuf` files. The `track-manager` will store incoming configurations in the shared volume using the following convention:

```
/mnt/config/config.pb
```

When new configurations are available, the `track-manager` will send a config protocol message to the track and each application will be responsible for updating their configs with the new `config.pb` contents.

3.3.2 Application Volume Mapping

The `config.pb` file in each **application container** is accessible through a shared volume at:

```
/mnt/config/config.pb
```

The **host machine** where the ioFog agent is running will store shared configs at:

```
/iofog/config/<YOUR_TRACK_NAME>/config.pb
```

This allows multiple tracks to run on the same `fog` host.

The volume mapping from the **host machine** to each **application container** is performed in ioAuthoring on each element in the volume mappings section.

The JSON that defines this volume mapping is:

```
{
  "volumemappings": [
    {"hostdestination": "/iofog/config/<YOUR_TRACK_NAME>",
     "containerdestination": "/mnt/config",
     "accessmode": "rw"}
  ]
}
```

3.4 Tempus Edge Messaging

Standard message formatting should be used by all edge applications. The purpose of standard message formatting is:

- reduce message parsing cost
- promote reusability between edge applications
- maintain flexibility for future edge application requirements
- improve the ease of application debugging

3.4.1 Message Anatomy

A Tempus Edge message is a single byte array that lives in the `contentdata` field of an IoFog message.

In general, Tempus Edge messages consist of 3 parts:

protocol first byte of the array defining the *Tempus Edge message protocol* to use.

type optional second byte of the array defining the *Tempus Edge message type* for the given `protocol`.

payload optional remaining byte array containing a serialized protobuf message of the type defined by `type`.

Message Protocols

Each IoFog message should contain a `messageProtocol` byte that describes the kind of message being sent.

This value is a `byte` or `uint8` depending on the language. It is the **first byte** in the messageContent field of the IoFog message.

The significance of each value is detailed below:

Pro-tocol Byte	Protocol Name	Description
000	Unde-fined protocol	This can indicate a message that didn't originate from a Hashmap application. This is used to allow support for JSON-based messaging used in default ioFog elements.
001	Config protocol	describes messages relating to configurations. This is generally either commanding a receiving element to accept new configs or alerting elements that new configs are available.
002	Data pro-tocol	used for standard data passing from element to element.

NOTE: new message protocols will be defined here as new protocols become necessary. Please send a pull request if you'd like to suggest other protocols!

Message Types

In general, a `messageType` will be similar to a `messageProtocol`. It will also be a `byte` value and will be the **second byte** in the ioFog message content field.

The type will serve to further specify how to decode the incoming message content or further specify what action a container should take when receiving a message.

The message types for each message protocol are defined below:

Configuration Message Types

Pro-to-col Byte	Type Byte	Pro-tobuf Enum	Description
001	000	UPDATE_CONFIG	informs the receiver that new configs are available. The receiver is expected to update their own configs.
001	001	TRACK_CONFIG	informs the receiver that a new track config is contained in this message and that the receiver should parse and persist the new track config. The `track-manager` is the intended consumer of this message type. **NOTE:** this config will be accepted as the new config by the track manager. This means changes will not be 'merged' with current configs. Use this message type when you want to submit a completely new config. Use other message types to update individual pieces of a track's configuration.
001	002	TRACK_METADATA	This message contains a TrackMetadata protobuf byte array and commands the receiver to update a track's metadata.
001	003	MQTT_CONFIG	This message contains an MqttConfig protobuf byte array and commands the receiver to update a track's MQTT configuration.
001	004	OPC_CONFIG	This message contains an OpcConfig protobuf byte array and commands the receiver to update a track's OPC configuration.
001	005	OPC_SUBSCRIPTIONS	this message contains an OpcConfig.subs value (OpcConfig.Subscriptions protobuf message) and commands the receiver to u pdate a track's OPC configuration subscriptions value.

Data Message Types

Protocol Byte	Type Byte	Protobuf Enum	Description
002	000	JSON	used to define a message that is a protobuf byte array that will decode into a JSON string that will need to be parsed by the receiver.
002	001	MQTT	used to define a message that is a protobuf byte array that will decode into an MQTT message. This message contains everything necessary for the `mqtt-client` to send this message in MQTT format.
002	002	OPC	used to define a message that is a protobuf byte array that will decode into an OPC message. This message contains everything necessary for the `opc-client` to send this message in OPC format.

NOTE: new message types will be defined here as new types become necessary. Please send a pull request if you'd like to suggest other types!