TIBCO Mashery® Local
Installation and Configuration Guide for Docker

Software Release 4.2
November 2017

Two-Second Advantage®

TIBCO®

Important Information

Contents

TIBCO Documentation and Support Services

Documentation for this and other TIBCO products is available on the TIBCO Documentation site. This site is updated more frequently than any documentation that might be included with the product. To ensure that you are accessing the latest available help topics, visit:

https://docs.tibco.com

Product-Specific Documentation

The following document for this product can be found on the TIBCO Documentation site:

- TIBCO Mashery® Local Installation and Configuration Guide

For information on TIBCO Cloud Integration with Mashery, refer to Integrating with Mashery.

TIBCO Mashery Professional customers will not have access to all of the features documented here. The following is a list of capabilities that are not available and as such will not be visible within the API Control Center for these customers:

- Distributed API Management (managing Organizations)
- Enriched Call Log Export
- HTTPS Client Profiles
- Mashery Local (Deploy)
- Event Triggers

Additionally, TIBCO Mashery Professional customers will not have access to the Mashery V2 API and as such will be able to use only the OAuth2 Accelerator feature.

Additionally, TIBCO Mashery Professional includes 8M QPM (Queries per month) and all traffic purchased is limited to a max of 100 QPS (Queries per second). TIBCO Mashery Professional customers can create a max of 25 APIs and 25 packages.

How to Contact TIBCO Support

For comments or problems with this manual or the software it addresses, contact TIBCO Support:

- For an overview of TIBCO Support, and information about getting started with TIBCO Support, visit this site:

  http://www.tibco.com/services/support

- If you already have a valid maintenance or support contract, visit this site:

  https://support.tibco.com

  Entry to this site requires a user name and password. If you do not have a user name, you can request one.

How to Join TIBCO Community

TIBCO Community is an online destination for TIBCO customers, partners, and resident experts. It is a place to share and access the collective experience of the TIBCO community. TIBCO Community offers forums, blogs, and access to a variety of resources. To register, go to the following web address:

https://community.tibco.com

Introduction

This guide provides an overview of the installation, requirements and configuration for Mashery® Local for Docker.

Mashery Local for Docker is a set of Docker images for running Mashery Local. These images can be customized for custom configurations. Mashery Local for Docker allows customers to perform hybrid traffic management on premise to run the API traffic inside data-centers. Mashery Local securely interacts with the Mashery Cloud hosted Developer Portal, Administration Dashboard and API Reporting and Analytics modules.

Mashery Local includes commercial support for Project Mashling, an open-source, event-driven microgateway. You can publish an endpoint exposed by Mashling to Mashery for access to broader API management functions. For more information about Mashling, please see https://www.mashling.io/home or https://community.tibco.com/products/project-mashling.

Assumptions

This guide assumes that you are using Docker version 1.12 or later. If you have an internal cloud, established best practices will be applied (for example disk alignment). If you are using different servers and clients, the underlying concepts implied by the installation and configuration steps still apply.

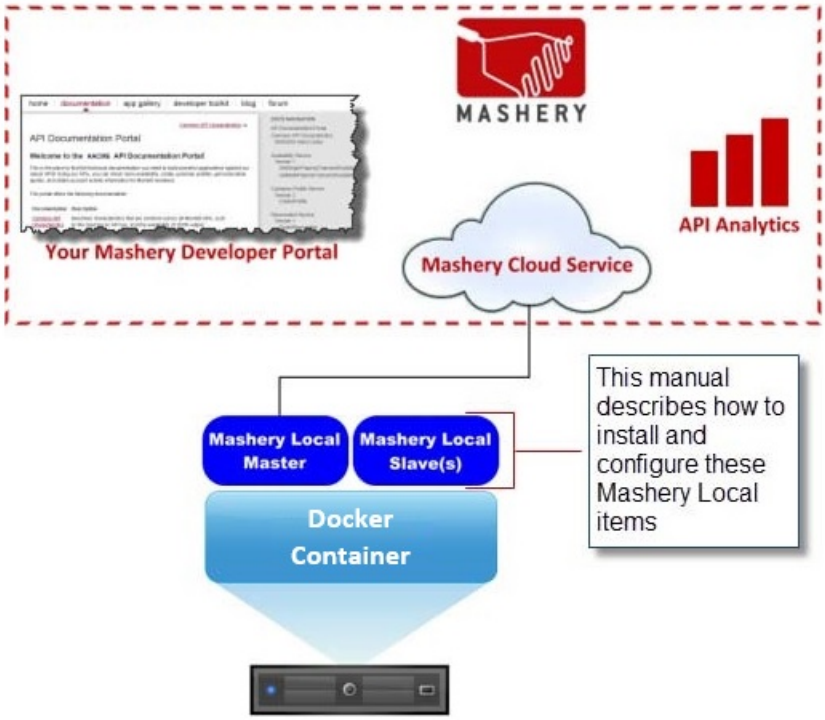
Conventions

This guide uses the following conventions:

- Keys you press simultaneously appear with a plus (+) sign between them (for example, **Ctrl+P** means press the **Ctrl** key first, and while holding it down, press the **P** key).
- Field, list, folder, window, and dialog box names have initial caps (for example, City, State).
- Tab names are **bold** and have initial caps (for example, **People** tab).
- Names of buttons and keys that you press on your keyboard are in bold and have initial caps (for example, **Cancel**, **OK**, **Enter**, **Y**).

Deployment Topology

The following diagram depicts a typical deployment topology for Mashery Local.



Overview of Installation and Configuration Process

This section provides a roadmap of the installation process for Mashery Local.

Procedure

1. Download the Mashery Local Docker artifact from TIBCO® eDelivery and create the Mashery Local Docker Image set as described in Installing and Configuring Mashery Local for Docker.
2. Configure a Mashery Local Master as described in Configuring a Mashery Local Master.
3. Configure slaves to the Mashery Local Master as described in Configuring Slaves to the Local Master. It is best practice to set up production with no less than 2 slaves per master.
4. Configure the load balancer as described in Configuring the Load Balancer.
5. Perform advanced configuration such as enabling notifications, as described in Advanced Configuration.

Mashery Local Failover Strategy Recommendations

Many of TIBCO customers rely on TIBCO Mashery Local to manage and distribute their revenue-generating and business-critical API traffic. The nature of the usage warrants that Mashery Local is always on without any downtime. Proper planning for redundancy and failover is recommended when high availability is expected of a mission-critical system.

The current Mashery Local architecture relies on four entities:

1. Mashery Cloud - This is where you make all your service configuration changes, through the Mashery Control Center API dashboard.
2. Mashery On-Prem Manager (MoM) - This is your Mashery Local's gateway to the Mashery Cloud. You must have received a secured key and secret, which provides each of your clusters its unique identity. You should always have a separate MoM key for each cluster, even if they are connecting to the same Mashery area.
3. A Mashery Master node synching with Cloud for API keys, OAuth Tokens, Service Configuration, User details, etc.. The time taken for synchronization of your configuration and token data is a function of the amount of data. Each customer implementation is unique and each network topology is different, so there isn't any formula that can correctly project the amount of time taken.
4. Slaves within the cluster that replicate the API Key, OAuth Token and Service Configuration data from Master. This happens within the cluster and in your environment, so the replication speed is slightly faster than Cloud Sync, but yet depends on various other environmental and data components.

TIBCO Recommendations to Achieve High Availability for TIBCO Mashery Local Deployment

Failover and redundancy can be achieved at many levels and should be considered while building your high availability strategy. Customers should also maintain updated runbooks so that their environment specific nuances are captured for their internal teams for faster deployment and recovery of systems. Failover systems should be tested and monitored in periodic intervals to ensure that they are in sync with production. Failure to do so will result in loss of traffic at the time of need. The following are some recommendations for redundancy – redundancy within a cluster and cross datacenter redundancy.

Redundancy within a Cluster:

Each Cluster has two type of Nodes – a Master Node that syncs the cluster with Cloud and many Slave Nodes that replicate from the Master. Though both type of Nodes are capable of serving traffic, TIBCO's recommendation is to keep the Master out of rotation in high traffic, high OAuth type implementations.

Keep one Slave extra than what is needed for optimum capacity to achieve within cluster redundancy.

If Master runs into problem due to disk, VM, or Network issues, then you can easily promote the spare Slaves to a Master and point the rest of the Slaves to the new Master. This can be achieved in minutes and will have a very low impact on the traffic. Except for newly synched up OAuth tokens, Slaves should be able to successfully service traffic during the promotion and pointing to the new Master. Fix the old Master Node and you can bring it back as a Slave into the cluster after re-imaging the VM.

If a Slave runs into problems, then take that Slave out of rotation from load balancer level. That way, you will not experience any traffic loss. Fix the issue and bring the slave back into rotation.

Cross Datacenter Redundancy:

If there is an issue with the datacenter, or if the whole cluster is having problems, having an Active-Active or Active-Passive cluster strategy is very beneficial in this scenario:

- **Active-Active Strategy**: Both Clusters with their unique Mashery On Prem Manager (MoM) key would connect to the same Mashery area and continue to sync. Nodes in both clusters can be used to serve traffic but both would have enough capacity (Disk Space, Caching configuration, etc.) to serve total traffic from both clusters combined and act like a failover if needed.

- **Active-Passive Strategy**: Both Clusters with their unique Mashery On Prem Manager (MoM) key would connect to the same Mashery area and will sync. Nodes from only one cluster would serve traffic. If needed traffic can be routed to the other non-traffic serving cluster without any blip in service. Both clusters should be identical in their configuration and capacity (Disk Space, Caching configuration, etc.) to serve total traffic.

Please note that TIBCO Mashery's license policy in cluster-based, so please discuss this with your Sales or Mashery Customer Success team. Having cluster redundancy is absolutely essential to avoid any traffic loss. Master sync and Slave replication takes time when done from scratch, and without cluster failover, you will encounter traffic loss.

Cluster Management in Mashery Local

The following sections describe how to set up and manage a Mashery Local Cluster:

- Setting up a new Mashery Local Cluster
- Adding a Slave to a Running Mashery Local Cluster
- Changing the Master in a Mashery Local Cluster

Setting up a New Mashery Local Cluster

The following section describes how to set up a new Mashery Local Cluster.

Procedure

Prepare mysqldump from Existing Mashery Local Cluster. (Optional)

1. For Mashery Local prior to 4.2.0, root can generate mysqldump in the Master Instance under the /mnt/ folder. Because proxy service needs to be stopped when mysqldump is being generated, this Mashery Local cluster cannot handle traffic.

   If you are using Docker-Machine, make sure you are talking to the right one. Execute the command `docker-machine ls` to find which one is currently active. It is also recommended to always redo the command:

   ```
   eval "$(docker-machine env <docker machine name>)"
   ```

   a) docker exec -it ml-tm nohup service javaproxy stop

   b)
      1. docker exec -it ml-db /bin/bash
      2. mysqldump -u masherybackup -p'password_for_masherybackup'--opt --master-data --single-transaction onprem > /mnt/onprem.sql
      3. md5sum /mnt/onprem.sql
      4. Exit the ml-db container

         Below step copies the mysqldump from container to host's file system in present working directory. You can specify other location by replacing '.' at the end.

      5. docker cp ml-db:/mnt/onprem.sql

   c) docker exec -it ml-tm nohup service javaproxy start

   For Mashery Local 4.2.0 or later, root can generate mysqldump in the Slave or Master Instance under /mnt/dump/ folder.

   Because proxy service needs to be stopped when mysqldump is being generated, this Master/Slave Instance cannot handle traffic.

   a) docker exec -it ml-tm nohup service javaproxy stop

   b)
      1. docker exec -it ml-db /bin/bash
      2. mysqldump -u masherybackup -p'password_for_masherybackup'--opt --dump-slave --single-transaction onprem > /mnt/dump/onprem.sql
      3. md5sum /mnt/dump/onprem.sql
      4. Exit the ml-db container

         Below step copies the mysqldump from container to host's file system in present working directory. You can specify other location by replacing '.' at the end

      5. docker cp ml-db:/mnt/dump/onprem.sql.

c) docker exec -it ml-tm nohup service javaproxy start

Set up Mashery Local Master Instance

2. Perform the following steps:
   a) Create and configure a Mashery Local Master instance as described in the topic "Configuring a Mashery Local Master", in this Guide.

      If special tuning is needed on MySQL, for example, expanding buffer pool size in "/etc/my.cnf" (login into ml-db container first):

      ```
      innodb_buffer_pool_size = 512M
      ```

      Mashery Local customers should consult TIBCO Support and request assistance in tuning MySQL. After tuning of MySQL, the MySQL service should be restarted by the Administrator:

      ```
      service mysqld restart
      ```

      When disk expansion is needed, the Administrator should follow the instructions in the topic "Expanding the Disk Space of a Mashery Local Instance" in this Guide.
   b) Import mysqldump from a previous Mashery Local Cluster. (Optional)

      Importing mysqldump from an existing Mashery Local Cluster can minimize the amount data to synchronize from Cloud, greatly reducing the amount of time required to setup the Mashery Local Instance.

      For example, suppose the remote Mashery Local instance is "remote_host".

      For Mashery Local prior to 4.2.0, root can copy mysqldump from remote host.

      If you are using Docker-Machine, make sure your are talking to the right one. Execute the command docker-machine ls to find which one is currently active. It is also recommended to always redo the command:

      ```
      eval "$(docker-machine env <docker machine name>)"
      ```

      1. First transfer the mysqldump from remote host to the current host. scp root@remote_host:<Absolute path of onprem.sql on remote host> <LOCAL DEST DIR>
      2. Now copy the mysqldump file from host machine to DB container's file system docker cp <LOCAL DEST DIR>/onprem.sql ml-db:/mnt/

      For Mashery Local 4.2.0 or later, root can copy mysqldump from remote host:

      1. First transfer the mysqldump from remote host to the current host. scp root@remote_host:<Absolute path of onprem.sql on remote host> <LOCAL DEST DIR>
      2. Now copy the mysqldump file from host machine to DB container's file system docker cp <LOCAL DEST DIR>/onprem.sql ml-db:/mnt/

      Verify the checksum of mysqldump file (Login into ml-db container:

      ```
      md5sum /mnt/dump/onprem.sql
      ```

      Stop "proxy" service:

      ```
      docker exec -it ml-tm nohup service javaproxy stop
      ```

      Import mysqldump:

      ```
      mysql -u masheryonprem -p'password_for_masheryonprem' onprem < /mnt/dump/
      onprem.sql
      ```

      After importing is done, restart MySQL:

      ```
      service mysqld restart
      ```

      Start "proxy" service:

      ```
      docker exec -it ml-tm nohup service javaproxy start
      ```
   c) Register the Mashery Local Instance as Master.

      Follow the instructions in the topic "Configuring a Local Mashery Master" in this guide to register the Mashery Local Instance as Master, to finish the settings for the Master in Cluster Manager.

When synchronizing the Master for the first time, allow the Master to finish the synchronization. This ensures the Master Instance is set up properly and synchronization with the Cloud is normal.

d) Stop "proxy" Service (Optional).

After "proxy" service is stopped, there will be no activity in MySQL. This enables Mashery Local slaves to replicate faster. To stop "proxy" service, run the following command:

```
docker exec -it ml-tm nohup service javaproxy stop
```

Set up the Mashery Local Slave Instance

3. Perform the following steps:
   a) Create and configure a Mashery Local Slave instance as described in the topic "Configuring a Mashery Local Slave", in this Guide.

   If special tuning is needed on MySQL, for example, expanding buffer pool size in "/etc/my.cnf":

   ```
   innodb_buffer_pool_size = 512M
   ```

   Mashery Local customers should consult TIBCO Support and request assistance in tuning MySQL. After tuning of MySQL, the MySQL service should be restarted:

   ```
   service mysqld restart
   ```

   When disk expansion is needed, the Administrator should follow the instructions in the topic "Expanding the Disk Space of a Mashery Local Instance" in this Guide.

   b) Register the Mashery Local Instance as a Slave.

   Follow the instructions in the topic "Configuring a Local Mashery Slave" in this guide to register the Mashery Local Instance as Slave, to finish the settings for the Slave in Cluster Manager.

   c) Import mysqldump from a previous Mashery Local Cluster. (Optional)

   Importing mysqldump from an existing Mashery Local Cluster can minimize the amount data to synchronize from Cloud, greatly reducing the amount of time required to setup the Mashery Local Instance.

   For example, suppose the remote Mashery Local instance is "remote_host".

   For Mashery Local prior to 4.2.0, the user can copy mysqldump from remote host.

   If you are using Docker-Machine, make sure you are talking to the right one. Execute the command docker-machine ls to find which one is currently active. It is also recommended to always redo the command:

   ```
   eval "$(docker-machine env <docker machine name>)"
   ```

   1. First transfer the mysqldump from remote host to the current host. scp root@remote_host:<Absolute path of onprem.sql on remote host> <LOCAL DEST DIR>
   2. Now copy the mysqldump file from host machine to DB container's file system docker cp <LOCAL DEST DIR>/onprem.sql ml-db:/mnt/

   For Mashery Local 4.2.0 or later, root can copy mysqldump from remote host:

   1. First transfer the mysqldump from remote host to the current host. scp root@remote_host:<Absolute path of onprem.sql on remote host> <LOCAL DEST DIR>
   2. Now copy the mysqldump file from host machine to DB container's file system docker cp <LOCAL DEST DIR>/onprem.sql ml-db:/mnt/

   Verify the checksum of mysqldump file (Login into ml-db container) :

   ```
   md5sum /mnt/dump/onprem.sql
   ```

   Stop "proxy" service:

   ```
   docker exec -it ml-tm nohup service javaproxy stop
   ```

Import mysqldump:

```
mysql -u masheryonprem -p'password_for_masheryonprem' onprem < /mnt/dump/
onprem.sql
```

After importing is done, restart MySQL:

```
service mysqld restart
```

Start "proxy" service:

```
docker exec -it ml-tm nohup service javaproxy start
```

Ensure MySQL Slave replicates well from MySQL Master:

```
mysql -u masheryonprem -p'password_for_masheryonprem' onprem
show slave status\G
```

d) Start "proxy" Service in Master after all Slaves are set. (Optional)

Run the following command:

```
docker exec -it ml-tm nohup service javaproxy start
```

Adding a Slave to a Running Mashery Local Cluster

The following section describes how to add a Slave to a running Mashery Local Cluster.

Procedure

1. Register the Mashery Local Instance as new Slave.

   Follow the instructions in the topic "Configuring a Local Mashery Slave" in this guide to register the Mashery Local Instance as Slave, to finish the settings for the Slave in Cluster Manager.

2. Exclude existing Slave or Master Instance from taking traffic.

   In order to prepare mysqldump, it is recommended that the user exclude an existing Slave Instance from Load Balancer, so that the Master Instance and other Slave Instances can keep taking traffic. In the case of high availability setup, the user can also switch traffic from the primary cluster to a backup cluster, then stop proxy service in the Master Instance of the primary cluster.

3. Stop "proxy" Service in existing Slave or Master Instance.

   Use the following command:

   ```
   docker exec -it ml-tm nohup service javaproxy stop
   ```

4. Initialize New Slave Database.

   a) Initialize New Slave Database by importing mysqldump file.

   The user should prepare mysqldump in the existing Slave Instance.

   1. The user generates mysqldump in existing Slave instance:

      ```
      mysqldump -u masherybackup -p'password_for_masherybackup' --opt --dump-
      slave --master-data --single-transaction onprem > /mnt/dump/onprem.sql


      md5sum /mnt/dump/onprem.sql
      ```

   2. Or, the user generates mysqldump in existing Master instance:

      ```
      mysqldump -u masherybackup -p'password_for_masherybackup' --opt --master-
      data --single-transaction onprem > /mnt/dump/onprem.sql


      md5sum /mnt/dump/onprem.sql
      ```

   b) Import mysqldump to new Slave instance.

   Copy mysqldump to new Slave instance:

   1. First transfer the mysqldump from remote host to the current host.scp root@remote_host:<Absolute path of onprem.sql on remote host> <LOCAL DEST DIR>

2. Now copy the mysqldump file from host machine to DB container's file systemdocker cp <LOCAL DEST DIR>/onprem.sql ml-db:/mnt/dump/
3. md5sum /mnt/dump/onprem.sql

Stop MySQL Slave:

```
mysql -u masheryonprem -p'password_for_masheryonprem' onprem
stop slave
```

Import mysqldump to new Slave Instance:

```
mysql -u masheryonprem -p'password_for_masheryonprem' onprem < /mnt/dump/
onprem.sql
```

c) Initialize New Slave Database by streaming mysqldump.

1. Stop Slave in new Slave Instance:

```
mysql -u masheryonprem -p'password_for_masheryonprem' onprem
stop slave
```

2. Import mysqldump to new Slave Instance via Stream:

- Stream from existing Slave Instance:

```
mysqldump -h remote_host_internal_ip -u mashonpremrepl -
p'password_for_mashonpremrepl' \
    --opt --dump-slave --single-transaction onprem \
    | mysql -u masheryonprem -p'password_for_masheryonprem' onprem
```

- Stream from existing Master Instance:

```
mysqldump -h remote_host_internal_ip -u mashonpremrepl -
p'password_for_mashonpremrepl' \
    --opt --master-data --single-transaction onprem \
    | mysql -u masheryonprem -p'password_for_masheryonprem' onprem
```

5. Restart MySQL Service:

```
service mysqld restart
```

6. Start "proxy" Service:

```
docker exec -it ml-tm nohup service javaproxy start
```

Changing the Master in a Mashery Local Cluster

The following section describes how to change the Master in a Mashery Local Cluster.

Procedure

1. Shut down the old Master in the Mashery Local Cluster.
   Follow the steps in the topic Shutting down a Master.
2. Promote one Slave to be the new Master.
   Follow the steps in the topic Promoting a Slave to Master.
3. Repoint other Slaves to the new Master.
   Follow the steps in the topic Repointing Other Slaves to a New Master.

Installing and Configuring Mashery Local for Docker

The following sections describe how to install and configure some basic environments complete with a master, one or more slaves, and load balancing.

Mashery Local for Docker includes a script that will download and install third-party software from third-party websites, including but not necessarily limited to CentOS and EPEL repositories located here:

- https://hub.docker.com/_/centos/
- http://vault.centos.org/
- https://dl.fedoraproject.org/pub/epel/

Such third-party software is subject to third-party software licenses that may be available on such third-party websites. For more information on CentOS repositories and EPEL, see:

- https://wiki.centos.org/AdditionalResources/Repositories
- https://fedoraproject.org/wiki/EPEL

Required Docker Images

Three images are needed to install Mashery Local for Docker:

1. On-premise database: ml-db
2. Memcache: ml-mem
3. TIBCO Mashery Local Core - Traffic Manager plus Cluster Manager UI: ml-core

Installing Mashery Local for Docker

To install Mashery Local for Docker:

Procedure

1. Install Docker Engine, Docker Machine (optional), and docker-compose (optional, and not needed if on Kubernetes) on your operating system.

   Refer to the Docker documentation for the operating system of your choice:
   - https://docs.docker.com/engine/
   - https://docs.docker.com/machine/
   - https://docs.docker.com/compose/

   For Mac OS installations, it's recommended to install Docker Toolbox so that multiple docker hosts can be run on the same box.

2. TIBCO Mashery Local for Docker is available as a TIB_mash-local**.tar.gz file. Download this file from TIBCO eDelivery and extract the file contents.
3. Create the TIBCO Mashery Local Docker Image set:
   a) Drop in custom configurations:

   - Modify examples/set-user-variables.sh and drop it in the resource/addons directory

   - (Optional) To use a custom https server PEM file for ML Cluster Manager, drop the PEM file to the resource/addons/certs directory and name it server.pem.

   If planning to run on Kubernetes, additional customization may be required. Please see the section Installing and Running Mashery Local for Docker with Kubernetes.

b) Navigate to the root folder of the extracted contents and run the following command to build the Mashery Local image set (comprising three images): `./build-docker.sh 2>&1 |tee /tmp/build-docker.log`

This will increment the image tag revision number. You can use the command "docker images" to check it out. You will need to modify the docker-compose.yml file to use the new tag if you build more than once in the same directory. However, if you would like to keep the same revision number, you will need to remove the file BUILD_NUMBER.txt in the current directory before starting the next build.

c) Verify three images are created: ml-db.tar.gz, ml-mem.tar.gz, ml-core.tar.gz. (The image sizes are about 400MB, 120MB, and 850MB, respectively.)

If the size of any image is significantly less than the numbers above, then the image build might have some problems. Check the build-docker.log generated from the previous step. If you see several errors, such as:

```
Could not retrieve mirrorlist http://mirrorlist.centos.org/?..... error
was14: PYCURL ERROR 22 - "The requested URL returned error: 503 Service
Unavailable"
```

then you probably have some firewall issues on your network. Switch to a network without firewall restriction to do the build.

4. (If planning to run on Kubernetes, the remaining steps do not apply. Please go to the section Installing and Running Mashery Local for Docker with Kubernetes to continue installation.) Navigate to the **examples** folder and copy the docker-compose.yml and the three image .gz files to the target Docker host machine.

The docker-compose.yml may need additional edits, depending on what ports need to be exposed or for other customization. For example, to add "extra hosts" if there are any extra host names and IP mapping that need to be added for a container.

Note: The indents and dash in the docker-compose yml file are important.

Run the following commands:

- `docker load -i` <each of the three image files, one by one>
- `docker-compose up -d`

5. Verify that four Docker containers are up:
`docker ps` to make sure the four containers are running.
6. Repeat Steps 4-5 for each Docker host that will run a Mashery Local instance.
7. Go to the instance in a browser:

   `https://<docker host-IP>:5480.`

8. Complete Master registration to TIBCO MOM (Mashery On-Prem Manager) or complete Slave registration to Master.

Additional Installation Tips

- Installation Steps with Docker Toolbox
- Working with Amazon EC2

Installing with Docker Toolbox

Docker Toolbox is a tool that lets you manage Docker engines on multiple virtual instances, and is used with Docker Machine. If you need to setup slaves for the cluster on different virtual instances, images built in the previous set of instructions (Step 3 of Installating Mashery Local for Docker) can be reused below.

1. Install Docker Toolbox from https://www.docker.com/products/docker-toolbox.
2. Use docker-machine *create* command to create Docker engines on virtual instances.

Drivers are available for various cloud provider platforms. Refer to https://docs.docker.com/machine/ for the latest information. Also refer to individual cloud provider documentation for more details on authentication details and other parameters you can use to customize your Docker Machine.

Some example commands are below:

a. To create a Docker Machine on a VirtualBox setup on your machine (prerequisite: VirtualBox 5+ ideal):

```
docker-machine create --driver virtualbox <docker machine name>
```

b. To create a Docker Machine on a VMware Fusion setup on your machine:

```
docker-machine create --driver vmwarefusion <docker machine name>
```

c. To create a Docker Machine on AWS (prerequisite: AWS signup, create an IAM administrator user and a key pair: AWS access key, AWS secret key):

```
docker-machine create --driver amazonec2 --amazonec2-access-key <your aws
access key> --amazonec2-secret-key <your aws secret key> <name for your new
AWS instance>
```

d. To create a Docker Machine on Microsoft Azure (prerequisite: Microsoft Azure signup):

```
docker-machine create --driver azure --azure-subscription-id <your
subscription id> <name for your new azure instance>
```

e. To create a Docker Machine on Google Cloud (prerequisite: Google Cloud signup, recommend installing and configuring gcloud tools locally to manage authentication. Refer to GCE documentation.):

```
docker-machine create --driver google --google-project <google project id> -
google-zone "us-west1-a" <name for your new google instance>
```

3. List all your available machines and make sure the one you just created shows up:

```
docker-machine ls
```

4. Connect your shell to a machine:

```
eval $(docker-machine env <docker machine name>)
```

```
docker-machine ls
```

(confirm the machine you are connecting to has an * to it to show that it's active)

5. You can use the three images you created via running the build-docker.sh script above:

a. Copy or move the images to the Amazon instance:

b. Run load command to load the images to the docker host.

c. Run `docker compose up -d`.

Working with Amazon EC2 Instances

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud.

Procedure

1. Install Docker Engine: http://docs.aws.amazon.com/AmazonECS/latest/developerguide/docker-basics.html
2. Install Docker Compose: https://docs.docker.com/compose/install

3. Install Docker Machine: https://docs.docker.com/machine/install-machine
4. TIBCO Mashery Local for Docker is available as a TIB_mash-local**.tar.gz file. Download this file from TIBCO eDelivery and extract the file contents.
5. Navigate to the root folder of the extracted contents and run the following command to build the Mashery Local image set (comprising three images):

   a) `./build-docker.sh`

   This will increment the image tag revision number. You can use the command "docker images" to check it out. You will need to modify the docker-compose.yml file to use the new tag if you build more than once in the same directory.

   b) Verify three images are created: ml_db.tar.gz, ml-mem.tar.gz, ml_core.tar.gz.
6. Navigate to the **examples** folder and copy the docker-compose.yml and the three image .gz files previously built to a target directory.

   The docker-compose.yml may need additional edits, depending on what ports need to be exposed or for other customization. For example, to add "extra hosts" if there are any extra host names and IP mapping that need to be added for a container.

   Note: The indents and dash in the docker-compose.yml file are important.

   Run the following command: `docker load -i` <each of the three image files, one by one>

   At this point, this AWS instance can be saved as an AM1 that can be re-used for any Mashery Local instance.
7. Run the following command: `docker-compose up -d`

   Verify that four Docker containers are up:

   `docker ps` to make sure the four containers are running.
8. Go to the instance in a browser:

   `https://<docker host-IP>:5480.`
9. Complete Master registration to TIBCO MOM (Mashery On-Prem Manager) or complete Slave registration to Master.

   If you are using docker-machine to create the docker container, then use the following command:

   ```
   docker-machine create --driver amazonec2 --amazonec2-access-key <Your
   AWS Key> --amazonec2-secret-key <Your AWS Key Secret> --amazonec2-region
   <Region You want to create the instance>  <Instance Name>
   ```

   You will need to open the following ports under the security group for registering Slave machine with Master: 22, 443, 2200, 2376 (TCP), 3306 , 5480, 11212.

Installation Troubleshooting Tips

Use the following tips in this section to troubleshoot your installation.

Changing the Traffic Manager Port

To change the Traffic Manager port in Mashery Local for Docker, modify the docker-compose.yml file to change the `80:80` under **services:/ml-tm:/ports:** to `<host port>:<container port>`, where the container port is the port you configured for the proxy.

Note that the host port could be different from the container port. The host port is the port that would be used to access the proxy from outside. After changing the ports in the docker-compose.yml, you will need to do docker-compose down and up to take them into effect. If you know the ports you are planning to switch in the future, you can add them in advance. Then, later when you decide to switch the port, you can simply change it from the UI (under **Instance Management > Instance Settings > HTTP/HTTPS port**).

There could be two scenarios for changing the proxy port:

Scenario 1	• Add the new port mapping to docker-compose.yml
	• Execute the command below if the Mashery Local Docker instance is running:
	`docker-compose down`
	• Execute
	`docker-compose up -d`
	• Change port from UI
	• Check whether port is in effect:
	`docker exec -it ml-tm netstat -nlp |grep LISTEN|grep tcp`
	• If the new port is not being listened, execute the command:
	`docker exec -it ml-tm nohup service javaproxy restart`

Scenario 2	• Change port from UI
	• Add the new port mapping to docker-compose.yml
	• Execute
	`docker-compose down`
	• Execute
	`docker-compose up -d`

How to Enable Additional Features That Require a New Port

To enable features, such as HTTPS, that requires a port other than 443, the port must be mapped in the docker-compose.yml file. If not, add it to the .yml file. Normally, it would be associated with Traffic Manager. So add it under the **services:/ml-tm:/ports**. Then, you access from outside through the Docker host IP address.

The example docker-compose.yml file already has most needed ports mapped. However, to change the ports to be used (for example HTTP/HTTPS ports), it would be better to make the changes in the docker-compose.yml file before starting the containers so that the mapping are in place. Then later, you can modify the UI to change the ports. However, if new port was not in effect after the UI change, try restarting the javaproxy. This can be done with command `docker exec -it ml-tm nohup service javaproxy restart`.

How to Telnet Memcache Port

Currently, only port 11212 is exposed to the outside for the memcache. You can telnet to the memcache port 11212 with the command:

```
telnet <docker host IP> 11212
```

The docker host IP can be found from the command echo `$DOCKER_HOST` if docker-machine is used. Otherwise, it's the machine/vm IP.

If you need to telnet to other memcache ports, you need to add the port mapping in the docker-compose.yml file and restart the docker-compose. Then, use the command:

```
telnet <docker host IP> <port number>
```

Alternatively, you could get in the memcache container with the command:

```
docker exec -it ml-mem /bin/bash
```

Install the telnet there and then use the command:

```
telnet localhost <port number>
```

How to Troubleshoot 596 Error Caused by Memcache

The `596 Service not found` error may be caused by memcache and you may see the following errors in proxy log:

```
[2017-03-21T19:16:42+00:00] WARN  [Memcached IO over {MemcachedConnection to ml-mem/
172.19.0.2:11214}] n.spy.memcached.MemcachedConnection  - Closing, and reopening
{QA sa=ml-mem/172.19.0.2:11214, #Rops=0, #Wops=2, #iq=0, topRop=null, topWop=Cmd:
get Keys: ENDPOINTS_digital-api.biogen.comExp: 0, toWrite=0, interested=0}, attempt
2.


[2017-03-21T19:16:44+00:00] WARN  [proxy-server-71] c.m.p.i.m.MemcachedClientImpl
- Operation timed out; retrying...


net.spy.memcached.internal.CheckedOperationTimeoutException: Timed out waiting for
operation - failing node: ml-mem/172.19.0.2:11214
```

First check if whether the memcached is running:

```
docker exec -it ml-mem ps -ef
```

Then, look for the memcached process. If not running, get in the ml-mem container to start it and see whether there's any error:

```
docker exec -it ml-mem /bin/bash
then


service memcached start
```

If it failed to start because of running out of file limit, following the instructions in the section How to Change Ulimits on Containers to fix it.

How to Change Ulimits for a Container

To override the fault ulimits for a container, you can either specify a single limit as an integer or soft/hard limits as a mapping. For example:

```
ulimits:
  nproc: 65535
  nofile:
    soft: 65535
    hard: 65535
```

How to Use NFS for Verbose Log

To use NFS for verbose log:

1. Mount the NFS to a host directory, for example, `/mnt/nfs`.
2. Add the volume mapping in the docker-compose.yml file under the **services:/ml-tm:/volumes**, for example:

   ```
   - /mnt/nfs:/var/log/tm_verbose_log
   ```

   Use the same indent as the existing entry `- mldata:/mnt`.
3. Execute

   ```
   docker-compose down
   ```
4. Execute

   ```
   docker-compose up -d
   ```
5. Modify the UI to set the Verbose Logs Location to `/var/log/tm_verbose_log` but leave the flag **Use NFS** unchecked.

6. Enable the verbose log.
7. Execute

```
docker exec -it ml-tm nohup service
javaproxy restart
```

Creating a Larger Memory for Memory Allocation

The **Memory Allocation** factor setting (in the **Management Options** of **Instance Management**) resizes the memory of the instance. It will not resize memory to less than 1024MB, as it is the minimum required memory for Docker.

On some platforms, for example Mac OS, if Docker host is created by Docker Machine, Docker creates an instance with 1024MB by default. For the Memory Allocation factor to have an effect, the Docker Machine should be created with a larger memory than 1024MB.

For example, to create a Docker Machine with 2GB of memory, use the following command:

```
$docker-machine create -d virtualbox --virtualbox-memory 2048 <docker-machine-name>
```

For example, to create a Docker Machine with 4GB of memory, use the following command:

```
$docker-machine create -d virtualbox --virtualbox-memory 4096 <docker-machine-name>
```

How to Monitor the Health of Docker Containers

To check the container logs, use the following example command:

```
docker logs ml-tm
```

To check the container status, use the following example commands:

```
docker stats ml-tm
docker top ml-tm
```

How to Increase the CPU Share and Memory of a Container

To increase the Docker CPU share and the memory of a container, use the following example command:

```
docker update --cpu-shares 5120 -m 3000M ml-tm
```

See Docker CPU share constraints and memory constraints for more information.

How to Do a Clean Restart of a Docker Instance

If you are using Docker-Machine, make sure your are talking to the right one. Execute the command `docker-machine ls` to find which one is currently active. It is also recommended to always redo the command:

```
eval "$(docker-machine env <docker machine name>)"
```

Deleting volumes will wipe out their data. Back up any data that you need before deleting a container.

Procedure

1. Stop the container(s) using the following command:

```
docker-compose down
```

2. Delete all containers using the following command:

```
docker rm -f $(docker ps -a -q)
```

3. Delete all volumes using the following command:

```
docker volume rm $(docker volume ls -q)
```

4. Restart the containers using the following command:

```
docker-compose up -d
```

How to Register Master or Slave with Commands

You can register the master or slave with the following example commands:

- On Docker host for master:

```
docker exec -it ml-cm /etc/ml.sh register_master '{ "api_key": "chainsproxykey",
"api_secret": "DVUVwqjXqQ", "node_name": "ML_Master", "master_address":
"192.168.99.100", "ntp": "false", "ntp_address": ""}'
```

- On Docker host for slave1:

```
docker exec -it ml-cm /etc/ml.sh register_slave '{ "api_key": "chainsproxykey",
"api_secret": "DVUVwqjXqQ", "node_name": "ML_Slave1", "master_address":
"192.168.99.100", "ntp": "false", "ntp_address": "", "slave_address":
"192.168.99.101"}'
```

If NTP needs to be configured (recommended) during registration, use the following commands:

- On the Docker host for master:

```
docker exec -it ml-cm /etc/ml.sh register_master '
{ "api_key": "chainsproxykey", "api_secret": "DVUVwqjXqQ", "node_name":
"ML_Master", "master_address": "192.168.99.100", "ntp": "true", "ntp_address":
"0.centos.pool.ntp.org","ntp_address1": "1.centos.pool.ntp.org","ntp_address2":
"2.centos.pool.ntp.org","ntp_address3": "3.centos.pool.ntp.org"}
'
```

- On Docker host for slave1:

```
docker exec -it ml-cm /etc/ml.sh register_slave '
{ "api_key": "chainsproxykey", "api_secret": "DVUVwqjXqQ", "node_name":
"ML_Slave1", "master_address": "192.168.99.100", "ntp": "true", "ntp_address":
"0.centos.pool.ntp.org","ntp_address1": "1.centos.pool.ntp.org","ntp_address2":
"2.centos.pool.ntp.org","ntp_address3": "3.centos.pool.ntp.org",
"slave_address": "192.168.99.101"}
```

How to Promote a Slave to Master with CLI

To promote a Slave to Master without using Mashery Local Cluster Manager administrator, you can use Command Line Interface (CLI).

Copy the following code to a temporary file (for example, `/tmp/promote_to_master.py`):

```
import sys;
sys.path.append('/var/www/htdocs/service/mashery/cgi');
import Mashery;
Mashery.make_master()
Mashery.backup_mysql_server()
```

Then, execute the following command:

```
docker cp /tmp/promote_to_master.py ml-cm:/tmp
docker exec -it ml-cm python/tmp/promote_to_master.py
```

How to Change Master to Slave with CLI

To change a Master to a Slave without using Mashery Local Cluster Manager administrator, you can use Command Line Interface (CLI).

Create a temporary file (for example, `/tmp/change_master.py`) with the following code (fill in the **<old master IP>** and **<new master IP>** fields):

```
import sys;
sys.path.append('/var/www/htdocs/service/mashery/cgi');
import debug;
debug.change_master_to('<old master IP>','<new master IP>')
```

Then, execute the following command:

```
docker cp /tmp/change_master.py ml-cm:/tmp
docker exec -it ml-cm python/tmp/change_master.py
```

Managing Docker Containers

Use the following commands to manage the Docker containers:

Action	Command
Pause	`docker-compose pause`
Unpause	`docker-compose unpause`
Restart	`docker-compose restart`
Shut down	`docker-compose down`
Complete Cleanup (remove persistent data)	`docker volume rm $(docker volume ls -q)` This will clean up all the database content and configurations. Then, you will need to redo and register the master and slave after re-running Mashery Local for Docker.  This command removes *all* volumes for a docker host. If you have other volumes besides those used by Mashery Local for Docker, you must remove the volumes for Mashery Local for Docker individually.

Installing and Running Mashery Local for Docker Using Kubernetes

To install and run Mashery Local for Docker using Kubernetes on Amazon Web Services (AWS) cloud, ensure your configuration meets the proper pre-requisites, then follow the steps below. The following instructions use Kubernetes Operations (kops), the recommended tool for creating and managing Kubernetes clusters.

Prerequisites

- Mac OS or Linux local working environment
- AWS account with full access to the AWS APIs
- AWS EC2 Console
- AWS Command Line Interface (CLI) installed and configured
  - AWS configurations set up, such as default region, access key, and secret key
  - Verify the command "aws" is on your path and that you can do some simple AWS CLI commands, for example:

    ```
    aws ec2 describe-vpcs
    ```
- Local Docker environment ready (either connect to a docker-machine that is up and running, or run docker host on the machine).

 You need this to upload docker images even if you are using pre-built images from S3.

- Mashery Local for Docker images built locally (For instructions on building Mashery Local for Docker images, see Installing Mashery Local for Docker.

  If you have custom adapters, see the Customizing for Kubernetes section.

- Docker images verified (no critical errors during the build and the images can be seen with the command "docker images".

Procedure

1. Install Kops.

   On OS X, use the following command:

   ```
   brew update && brew install kops
   ```

2. Create the cluster.

   a) Use the following commands as an example to create the cluster definition:

   ```
   export NAME=kubeml411.rkdemo.com
   export KOPS_STATE_STORE=s3://rkdemo-state-store
   kops create cluster --zones us-east-1a $NAME
   ```

   The cluster name must use a public domain suffix. In this example, `rkdemo.com` is a registered domain.

   b) Update the slave nodes' instance type by editing the cluster node configuration:

   ```
   kops edit ig --name=$NAME nodes
   ```

   The default size listed under the **spec.machineType** property is `t2.medium`; change it to **m3.large** or higher. Additionally, the values of the **maxSize** and **minSize** properties should be increased from `2` to `3`, so that each Mashery Local node (the master and two slaves) will be deployed on a separate host.

   Warning! Leaving the **minSize** and **maxSize** property values at `2` will result in both slaves being deployed to the same physical host, resulting in TCP port conflicts.

   Tip! Use vi commands to save and exit the editor.

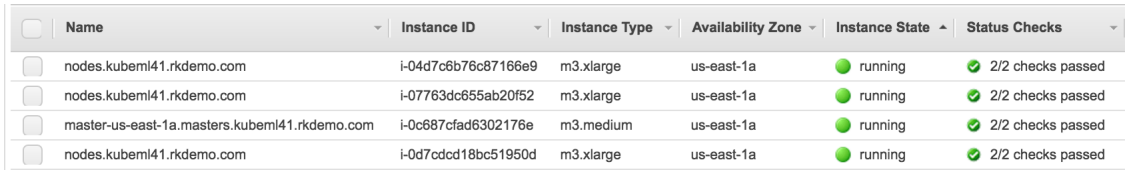
Edit cluster node configuration

```
apiVersion: kops/v1alpha2
kind: InstanceGroup
metadata:
  creationTimestamp: 2017-07-20T22:55:30Z
  labels:
    kops.k8s.io/cluster: kubeml41.rkdemo.com
  name: nodes
spec:
  image: kope.io/k8s-1.6-debian-jessie-amd64-hvm-ebs-2017-05-02
  machineType: m3.large
  maxSize: 3
  minSize: 3
  role: Node
  subnets:
  - us-east-1a
```

Finally, apply the changes using the following command:

```
kops update cluster $NAME --yes
```

If successful, your EC2 Console on AWS should look like this:



Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks
nodes.kubeml41.rkdemo.com	i-04d7c6b76c87166e9	m3.xlarge	us-east-1a	running	2/2 checks passed
nodes.kubeml41.rkdemo.com	i-07763dc655ab20f52	m3.xlarge	us-east-1a	running	2/2 checks passed
master-us-east-1a.masters.kubeml41.rkdemo.com	i-0c687cfad6302176e	m3.medium	us-east-1a	running	2/2 checks passed
nodes.kubeml41.rkdemo.com	i-0d7cdcd18bc51950d	m3.xlarge	us-east-1a	running	2/2 checks passed

3. Install the Kubernetes Dashboard UI
   a) Use the following command to install the Dashboard service:

   ```
   kubectl create -f https://rawgit.com/kubernetes/dashboard/master/src/deploy/kubernetes-dashboard.yaml
   ```

   b) Access the dashboard app at https://api.<cluster-name>/ui, for example, https://api.kubeml411.rkdemo.com/ui

   Use the following command to obtain the password for the admin user:

   ```
   export PATH=<path to kubernetes-directory>/client/bin:.:$PATH
   ```

   c) You can access the Kubernetes console UI with the following URL: `<cluster server url>/ui`

   In the previous example, this is: `https://34.205.42.112/ui`

4. Deploy Mashery Local.
   a) Create a private Amazon EC2 Container Registry (ECR) for Mashery Local for Docker, for example:

   ```
   aws ecr create-repository --repository-name <registry name>

   for example
   aws ecr create-repository --repository-name tibco/mlce
   ```

   If you have never used AWS ECS before, you will need to go to the AWS ECS dashboard and follow the "Getting Started" step.

   b) Go to the directory **examples/kubernetes** extracted from the Mashery Local for Docker 4.1 (or later) release, modify the aws-env.sh with the planned configuration, and set the environment variables with the command:

   ```
   . aws-env.sh
   ```

   The ML_REGISTRY_NAME is the registry name used in step 4a.

   The ML_REGISTRY_HOST can be found with the following command:

   ```
   aws ecr get-login --registry-ids `aws ecr describe-repositories --repository-names "$ML_REGISTRY_NAME" |grep registryId |cut -d ":" -f 2|tr -d ' ",'`|awk -F'https://' '{print $2}'
   ```

   Or, from the AWS ECS dashboard, go to `Repositories > Repository URI`. For example, with repository URI "`12345603243.dkr.ecr.us-east-1.amazonaws.com/tibco/mlce`", the ML_REGISTRY_NAME is `tibco/mlce`, and the ML_REGISTRY_HOST is `12345603243.dkr.ecr.us-east-1.amazonaws.com`.

   c) Add or set login credentials in `<home>/.docker/config.json` using the command:

   ```
   aws ecr get-login --no-include-email | sh -
   ```

   For the `get-login` command, the `--no-include-email` option must be specified for Docker version 17.06 or later, otherwise the command will fail.

d) Load Docker images.

1. Verify Mashery Local for Docker images with the correct tag are in the current docker host with the command:

```
docker images
```

The tag should match the env. variable **ML_IMAGE_TAG**.

2. Execute the following script to load images to the ECR Docker registry:

```
upload-images.sh
```

e) Execute the following script to store the Docker registry key as Kubernetes "Secret":

```
set-registry-key.sh
```

f) Execute the following script to store MOM host and key as Kubernetes "Secret":

```
set-mom-secret.sh create <MOM key> <MOM secret>
```

If you want to enable HTTPS or OAuth, see the section Customizing for Kubernetes for additional configuration steps.

g) Create storage classes for Mashery Local for Docker persistent stores:

```
set-storage-classes.sh
```

h) Create Mashery Local Traffic Manager service and Mashery Local Master service:

```
set-ml-services.sh
```

You can check the services with the following commands:

```
kubectl describe service ml-traffic-manager
```

```
kubectl describe service ml-master
```

The ml-traffic-manager is configured with load balancer. You can find the load balancer DNS name with the following command:

```
kubectl describe service ml-traffic-manager|grep Ingress|awk -F' ' '{print $3}'
```

The load balancer can also be found on the AWS EC2 dashboard **Load Balancers** list.

API invocation should be done solely via the AWS ELB (Elastic Load Balancer). The ELB configuration uses the *internal* IPs of the customer nodes for load balancing, so invoking API calls directly via the public IP addresses of the master or slave nodes is not an option.

i) Deploy Mashery Local master instance:

```
deploy-master.sh
```

You can check the ML instance pods with the command:

```
kubectl get pods
```

The ML master pod has a name like `ml-master-.....`. When it's fully up, you should see 4/4 under the **READY** column with **STATUS** "Running" for the master instance pod.

You can check the startup init instance log with the following command:

```
kubectl exec -ti `kubectl get pods |grep ml-master |cut -d " " -f 1` -c ml-cm
-- cat /var/log/mashery/init-instance.log
```

When it's fully ready to serve traffic, you should see something like the following:

```
....

Register status: Content-Type: application/json Status: 200 {"results":
[{"results": [{"address": "10.0.22.98"}], "error": null}, {"results":
[{"area_name": "Roger"}], "error": null}, {"results": [{"credentials_updated":
true}], "error": null}, {"results": [{"name": "ml-master-4209822619-sxq40",
"id": 0}], "error": null}, {"results": [{"is_master": true}], "error": null}],
"error": null}

****  04/06 05:27:38  Register instance succeeded

Load service result:

Load service result:
```

```
Load service result: 70a0b42e-2b9a-4f60-a4d6-8c5503894043 [SERVICES] 04/06/17
05:27:45 - 04/06/17 05:27:47: 254 records (Success) 70a0b42e-2b9a-4f60-
a4d6-8c5503894043 [KEYS] 04/06/17 05:27:47 - 04/06/17 05:27:55: 10963 records
(Success) 70a0b42e-2b9a-4f60-a4d6-8c5503894043 [APPS] 04/06/17 05:27:55 -
04/06/17 05:28:23: 6884 records (Success) 70a0b42e-2b9a-4f60-a4d6-8c5503894043
[CLASSES] 04/06/17 05:28:23 - 04/06/17 05:28:23: 0 records (Success)
70a0b42e-2b9a-4f60-a4d6-8c5503894043 [PACKAGES] 04/06/17 05:28:23 - 04/06/17
05:29:54: 28824 records (Success) 70a0b42e-2b9a-4f60-a4d6-8c5503894043
[PACKAGEKEYS] 04/06/17 05:29:54 - 04/06/17 05:30:17: 5553 records (Success)

****  04/06 05:30:17  Service info loaded

Load cache output first ten lines: - Trying to load mapi data for spkey:
m8hxx3wxy5wjyjhfzc328wqh key: MAPI_m8hxx3wxy5wjyjhfzc328wqh::
2011w25DeveloperJay key: MAPI_m8hxx3wxy5wjyjhfzc328wqh::2011w25DeveloperRoger
key: MAPI_m8hxx3wxy5wjyjhfzc328wqh::3skjegt4ddpam6a5r8sfgpkz key:
MAPI_m8hxx3wxy5wjyjhfzc328wqh::4q5t7z4gduy388z9nk5tmptm key:
MAPI_m8hxx3wxy5wjyjhfzc328wqh::4tzw5p5h5mx8gr8ez6m34wak key:
MAPI_m8hxx3wxy5wjyjhfzc328wqh::5s8ds7dcyj7cjz4h9h5tv7ev key:
MAPI_m8hxx3wxy5wjyjhfzc328wqh::5yy6dkjbq7sr922j4wt6u2hc key:
MAPI_m8hxx3wxy5wjyjhfzc328wqh::6mbcz48nabrz682xn2hdmhzn key:
MAPI_m8hxx3wxy5wjyjhfzc328wqh::8tng6tk5bzhpfqexn525cqnj

****  04/06 05:31:01  Cache Loaded

****  04/06 05:31:01  Ping Traffic Manager succeeded

****  04/06 05:31:01  Setting status ready
```

When the ML master instance containers are up, you can find the ML master instance node public IP with the following command:

```
kubectl describe node `kubectl get pods -o wide |grep ml-master |awk -F' ' 'ml-
master'`|\
awk '/Addresses/ {for(i=1; i<=6; i++) {getline; print}{print "\n"}}'


InternalIP:    172.20.41.66
LegacyHostIP:  172.20.41.66
ExternalIP:    184.73.16.126
InternalDNS:   ip-172-20-41-66.ec2.internal
ExternalDNS:   ec2-184-73-16-126.compute-1.amazonaws.com
Hostname:      ip-172-20-41-66.ec2.internal


InternalIP:    172.20.54.44
LegacyHostIP:  172.20.54.44
ExternalIP:    54.160.43.6
InternalDNS:   ip-172-20-54-44.ec2.internal
ExternalDNS:   ec2-54-160-43-6.compute-1.amazonaws.com
Hostname:      ip-172-20-54-44.ec2.internal


InternalIP:    172.20.58.180
LegacyHostIP:  172.20.58.180
ExternalIP:    34.207.81.153
InternalDNS:   ip-172-20-58-180.ec2.internal
ExternalDNS:   ec2-34-207-81-153.compute-1.amazonaws.com
Hostname:      ip-172-20-58-180.ec2.internal
```

If you need to access the Mashery Local instance Cluster Manager UI, you need to open the port 5480 for UI access. For convenience, you can open the port for all minion nodes in the cluster with the following command:

```
aws ec2 authorize-security-group-ingress --group-id `aws ec2 describe-security-
groups \
--filters "Name=group-name,Values=nodes.${NAME}" | jq -r
'.SecurityGroups[0].GroupId'` \
--protocol tcp --port 5480 --cidr 0.0.0.0/0
```

Or you can open the port individually as needed with additional security group through AWS UI or CLI.

Then you can login to the ML master instance Cluster Manager UI with https://< ML master instance node ip>:5480.

You can get into any ML master instance container with the following command:

```
kubectl exec -ti `kubectl get pods |grep ml-master |cut -d " " -f 1` -c
<container name> -- /bin/bash
```

The container names are: ml-db, ml-mem, ml-tm, ml-cm.

You can also execute some simple remote command on a container directly:

```
kubectl exec -ti `kubectl get pods |grep ml-master |cut -d " " -f 1` -c
<container name> -- <remote command>

for example:

kubectl exec -ti `kubectl get pods |grep ml-master |cut -d " " -f 1` -c ml-tm
-- ls -l /var/log/trafficmgr/access
```

At any time, you could also get in the Kubernetes dashboard UI to check the progress, such as checking the deployment, replica sets, services, pods, containers and their logs.

j) Deploy Mashery Local slave instances:

```
deploy-slaves.sh
```

You can check the Mashery Local instance pods with the command:

```
kubectl get pods
```

The Mashery Local slaves instance pods are named with ml-slave-0, ml-slave-1, ml-slave-2.

When it's fully up, you should see 4/4 under the **READY** column with **STATUS** "Running" for the slave instance pod.

You can check the startup init instance log with the following command:

```
kubectl exec -ti `kubectl get pods |grep <slave pod name> |cut -d " " -f 1` -c
ml-cm -- cat /var/log/mashery/init-instance.log

for example:

kubectl exec -ti `kubectl get pods |grep ml-slave-0 |cut -d " " -f 1` -c ml-cm
-- cat /var/log/mashery/init-instance.log
```

You can find the Mashery Local slave instance node IP with the following command:

```
kubectl describe node `kubectl get pods -o wide |grep <slave pod name> |awk -
F' ' '{print $7}'` |grep Addresses |cut -d "," -f 3
```

Then, login to the ML slave instance Cluster Manager UI with https://<ML slave instance node ip>:5480

If you didn't open the port 5480 for all nodes in the previous step, you need to open the port for each ML slave instance individually with additional security group through AWS UI or CLI.

You can get into any ML slave instance container with the following command:

```
kubectl exec -ti `kubectl get pods |grep <slave pod name> |cut -d " " -f 1` -c
<container name> -- /bin/bash
```

The container names are: ml-db, ml-mem, ml-tm, ml-cm.

You can also execute some simple remote command on a container directly:

```
kubectl exec -ti `kubectl get pods |grep <slave pod name> |cut -d " " -f 1` -c
<container name> -- <remote command>

for example:

kubectl exec -ti `kubectl get pods |grep ml-slave-0 |cut -d " " -f 1` -c ml-tm
-- ls -l /var/log/trafficmgr/access
```

At any time, you could also get into the Kubernetes dashboard UI to check the progress, such as checking the stateful sets, services, pods, and containers and their logs.

By default, it's configured to run two slave instances.

You can use the following command to increase or reduce the number of slaves:

```
kubectl patch statefulset ml-slave --type='json' -p='[{"op": "replace",
"path": "/spec/replicas", "value":<the desired replica number>}]'
```

However, you must have enough worker nodes to run all the slave instances.

k) Test the traffic, using the following example commands:

```
export LB=`kubectl describe service ml-traffic-manager|grep Ingress|awk -F'
' '{print $3}'` && echo $LB

curl -H 'Host: roger.api.perfmom.mashspud.com' http://$LB/testep?
api_key=funjsgx8m5bsew2jngpdanxf
```

l) Cleanup or undeploy Mashery Local instances.

To undeploy Mashery Local slave instances:

```
deploy-slaves.sh delete
```

To undeploy Mashery Local master instances:

```
deploy-master.sh delete
```

m) Shut down Kubernetes cluster using the following command:

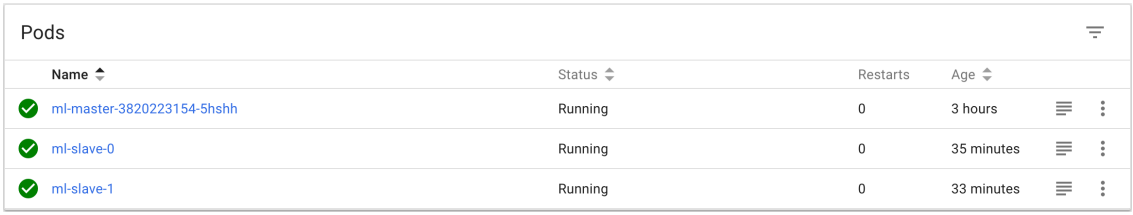
```
kubernetes/cluster/kube-down.sh
```

Verifying Installation on Kubernetes

To verify the Mashery Local installation on Kubernetes:

Procedure

1. Navigate to the Pods view in the Kubernetes Dashboard.



You can also use the following command:

```
kubectl get pods
```

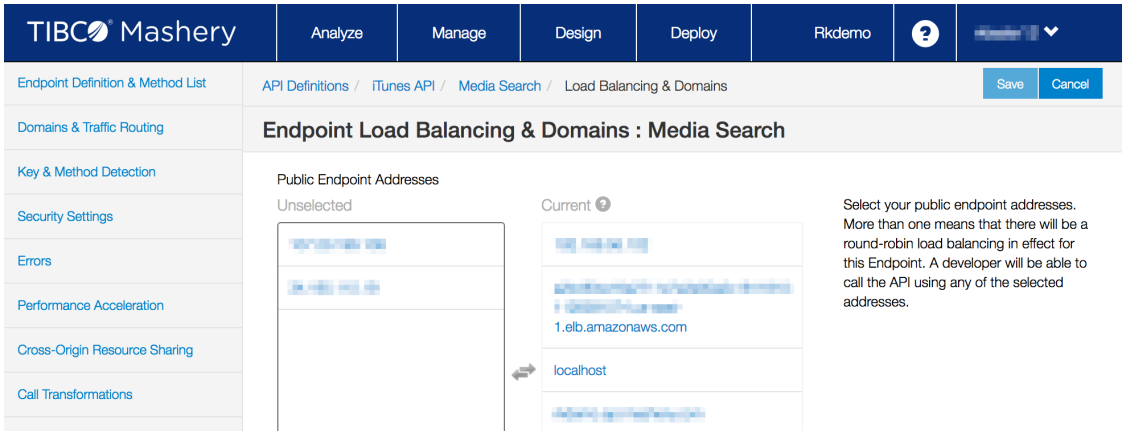
```
kubectl get pods

NAME                          READY     STATUS    RESTARTS   AGE
ml-master-3820223154-5hshh    4/4       Running   0          2h
ml-slave-0                    4/4       Running   0          5m
ml-slave-1                    4/4       Running   0          3m
```

The values under the **READY** column should all read `4/4`.

2. The Mashery Local Cluster Manager console should look like this:

Invoking APIs via AWS ELB

In the Mashery API Control Center dashboard, make sure to add the AWS ELB host name to your endpoint's domain list and synchronize the Mashery Local cluster. For more information, see Endpoint Load Balancing and Domains.



When you invoke your API via the ELB, you can tell which traffic manager node responded to the call by looking at the HTTP response headers, for example:

```
X-Mashery-Responder: ml-slave-1
```

Customizing for Kubernetes

When Kubernetes does autoscaling or auto-repairing, you are not able to get into the Mashery Local Cluster Manager UI to change the configuration, so all customizations should be set up at image build time.

Disabling HTTP or using a different HTTP Port

To disable HTTP or to use a different HTTP port, modify the following variables in the examples/set-user-variables.sh and drop it in the `resource/addons` directory before building docker images:

```
export HTTP_ENABLED=true
```

```
export HTTP_PORT=80
```

Also, make the corresponding changes in the **aws-env.sh** file.

Adding a Custom Adapter

To add a custom adapter, put the adapter zip file in the `resources/addons` directory and modify the following variable in the examples/set-user-variables.sh and drop it in the resource/addons directory before building docker images:

```
export CUSTOM_ADAPTER_ZIP_FILE_NAME=<custom adapter zip file name>
```

Enabling OAuth

To enable OAuth, in the **aws-env.sh** file, set:

```
export OAUTH_ENABLED=true
```

Then, execute the following command as an additional step in Step 7 of Installing and Running Mashery Local for Docker with Kubernetes:

```
set-oauth-secret.sh <create|replace|delete> <OAuth authorization user> <OAuth
authorization user password>
```

For the first argument in the above command, use "create" for the first time, then use "replace" for subsequent changes.

Enabling HTTPS

To enable OAuth, in the **aws-env.sh** file, set:

```
export HTTPS_ENABLED=true
```

```
export HTTPS_PORT=<port number>
```

Next, put the server certificate file in the `resources/addons/certs` directory before building the images.

Then, execute the following command as an additional step in Step 7 of Installing and Running Mashery Local for Docker with Kubernetes:

```
set-https-secret.sh <create|replace|delete> <server certificate file name> <server
certificate password>
```

For the first argument in the above command, use "create" for the first time, then use "replace" for subsequent changes.

Installing and Running Mashery Local for Docker Using GCP

To install and run Mashery Local for Docker using GCP (Google Cloud Platform), follow the steps below.

Procedure

1. Install the Google Cloud SDK.

   Follow the instructions on the Google Cloud SDK page. Verify your installation using the following command:

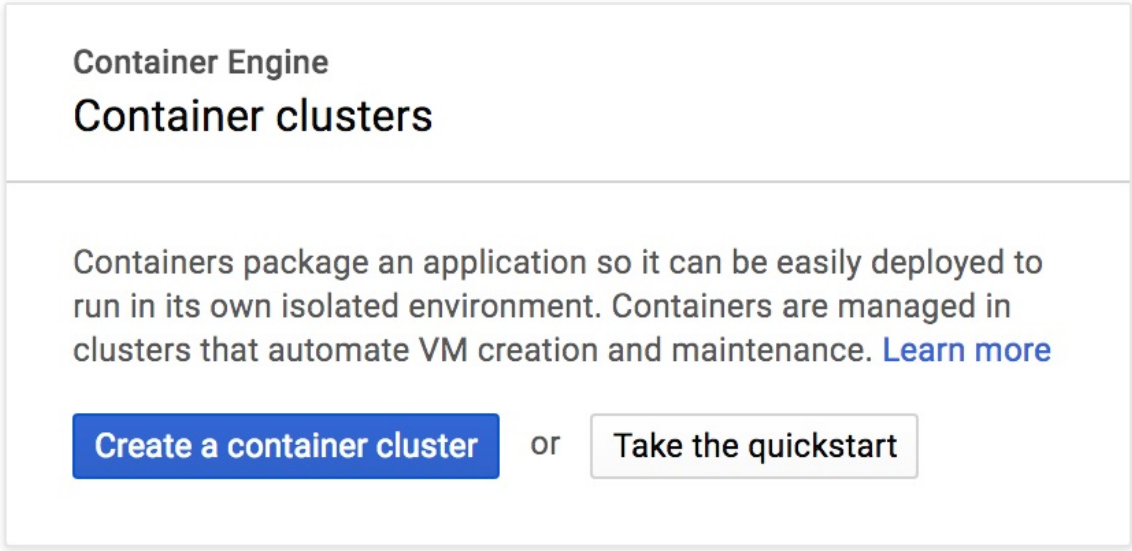
   ```
   gcloud -v
   ```

   The output should look like the following:

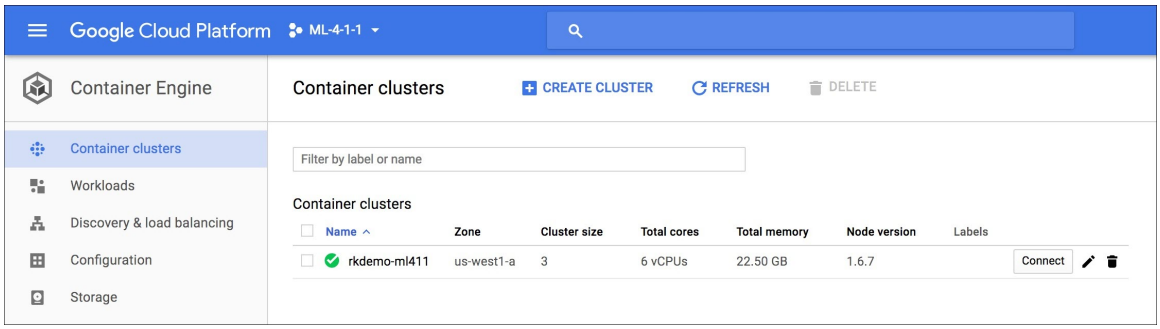
   ```
   Google Cloud SDK 169.0.0
   bq 2.0.25
   core 2017.08.28
   gcloud
   gsutil 4.27
   ```

2. Create the cluster.

   Use the GCP console to create a new cluster:



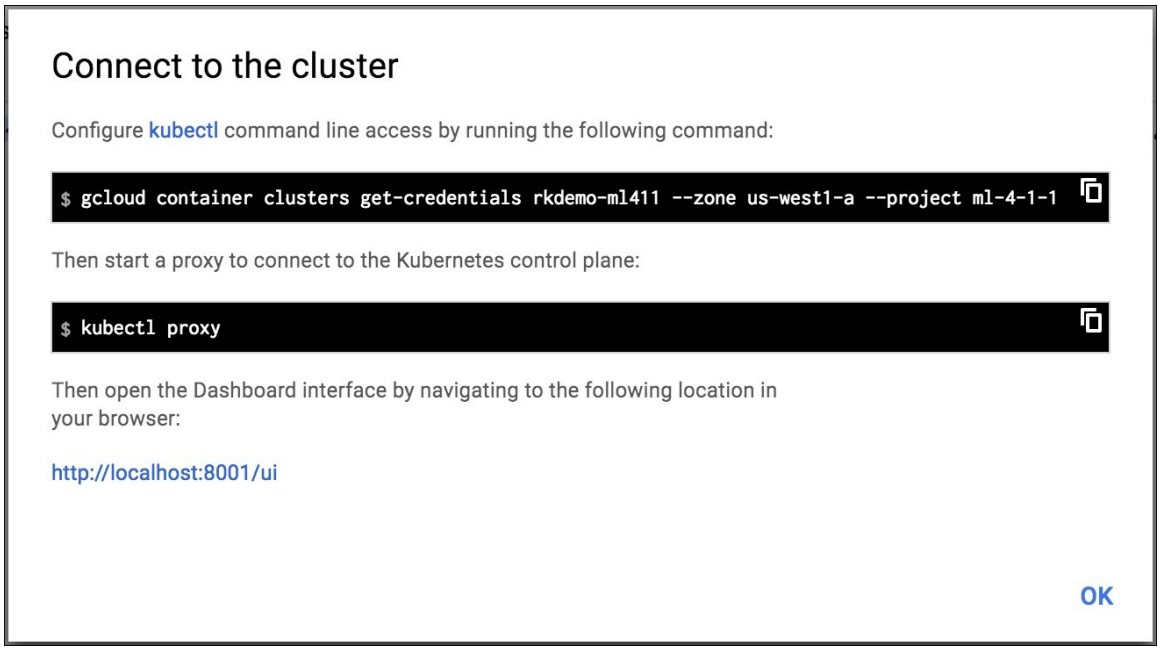
When your cluster is ready, it should be listed as follows:



Use n1-standard-2 as the *minimum* machine type for the cluster (2 vCPUs and 7.5 GB of memory).

3. Install the Kubernetes Dashboard UI.

   Click the **Connect** button in the console to reveal the **gcloud** commands to execute (the following is an excerpt):



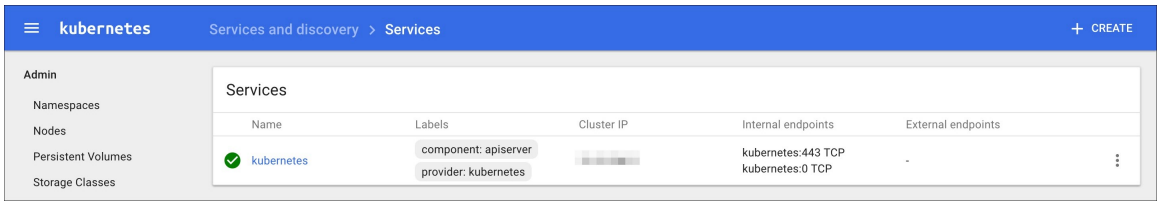
The output from the first command should look like the following:

```
Fetching cluster endpoint and auth data.
kubeconfig entry generated for rkdemo-ml411.
```

The output from the second command should look like the following:

```
Starting to serve on 127.0.0.1:8001
```

Access the dashboard app at `https://localhost:8081/ui`.



4. Upload the Docker Images to the GCP Container Registry.
   a) In order to deploy Mashery Local to your Kubernetes cluster on GCP, you will need to upload the images to your GCP Container Registry. The registry console is available at https://cloud.google.com/container-registry. The ML Docker images need to be tagged first before they can be uploaded to the registry. Use the following command to tag them: `docker tag [IMAGE] [HOSTNAME]/[PROJECT-ID]/[IMAGE]`, for example:

```
docker tag ml-core:v4.1.1.0 us.gcr.io/ml-4-1-1/ml-core:v4.1.1.0
docker tag ml-mem:v4.1.1.0 us.gcr.io/ml-4-1-1/ml-mem:v4.1.1.0
docker tag ml-db:v4.1.1.0 us.gcr.io/ml-4-1-1/ml-db:v4.1.1.0
```

The following is the list of available gcr.io (container registry) hostnames:

- us.gcr.io hosts your images in the United States
- eu.gcr.io hosts your images in the European Union
- asia.gcr.io hosts your images in Asia
- gcr.io without a prefix hosts your images in the United States, but this behavior may change in a future release.

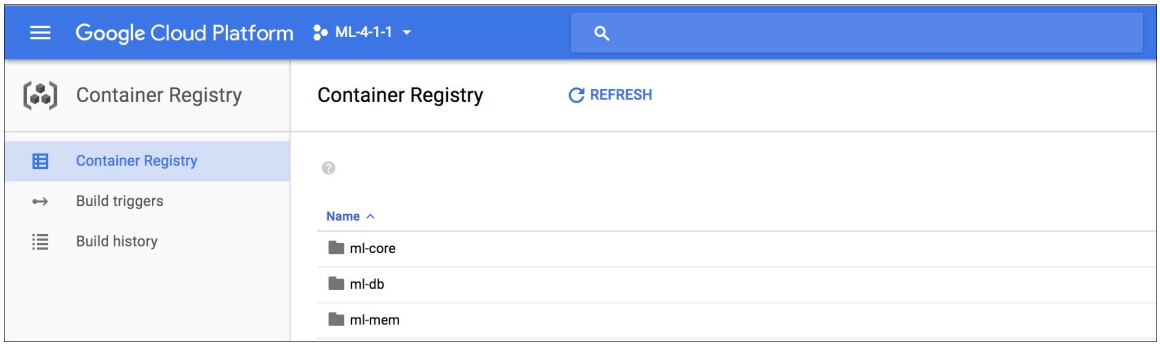
See https://cloud.google.com/container-registry/docs/pushing-and-pulling for more details.

b) Upload the images using the `gcloud docker push` command:

```
gcloud docker -- push us.gcr.io/ml-4-1-1/ml-core
gcloud docker -- push us.gcr.io/ml-4-1-1/ml-mem
gcloud docker -- push us.gcr.io/ml-4-1-1/ml-db
```

The space separating the '--' characters from the push parameter in the above commands is required.

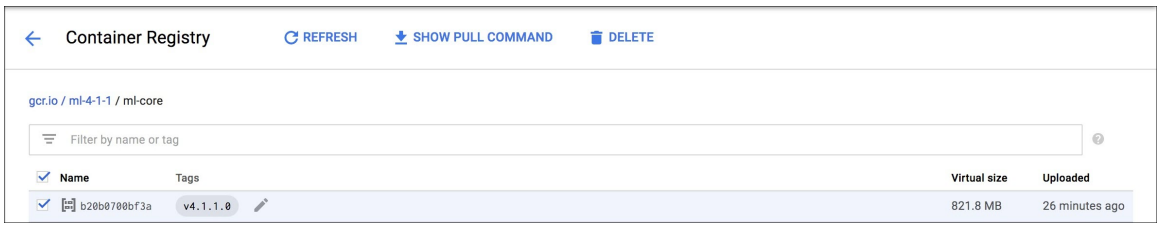
c) Once the push commands complete, the new images should be listed in the container registry console.



You can also verify the pushed images using the `gcloud container images` command:

```
gcloud container images list-tags us.gcr.io/ml-4-1-1/ml-core
DIGEST        TAGS      TIMESTAMP
b20b0700bf3a  v4.1.1.0  2017-08-03T22:33:02
```

d) With the Docker images in the container registry, you can inspect the images in the container registry console.



e) Execute the `pull` command for each image:

```
gcloud docker -- pull us.gcr.io/ml-4-1-1/ml-core:v4.1.1.0
gcloud docker -- pull us.gcr.io/ml-4-1-1/ml-mem:v4.1.1.0
gcloud docker -- pull us.gcr.io/ml-4-1-1/ml-db:v4.1.1.0
```

```
Welcome to Cloud Shell! Type "help" to get started.
rkiesler@ml-4-1-1:~$ gcloud docker -- pull gcr.io/ml-4-1-1/ml-core:v4.1.1.0
v4.1.1.0: Pulling from ml-4-1-1/ml-core

4b1edaad48b8: Pull complete
8f05be9d49e7: Pull complete
279031214595: Pull complete
0135b2a4b65b: Extracting [==========>                                        ] 118.7 MB/512.4 MB
403667c93ec2: Download complete
1caf2cdcf15b: Download complete
0b7049652cef: Download complete
871d1547648a: Download complete
```

f) Create a new shell script to setup GCP environment variable (replaces aws-env.sh):

```
gcp-env.sh
```

Source the new shell script:

```
source gcp-env.sh
```

Copy deploy-master.sh to deploy-master-gcp.sh. Edit deploy-master-gcp.sh and change all filename references from:

```
cat >> ml-master.yaml << EOF
```

to:

```
cat >> ml-master-gcp.yaml << EOF
```

g) Additionally, change all image registry references from:

```
image: $ML_REGISTRY_HOST/$ML_REGISTRY_NAME:ml-core-$ML_IMAGE_TAG
image: $ML_REGISTRY_HOST/$ML_REGISTRY_NAME:ml-mem-$ML_IMAGE_TAG
image: $ML_REGISTRY_HOST/$ML_REGISTRY_NAME:ml-db-$ML_IMAGE_TAG
```

to:

```
image: $ML_REGISTRY_HOST/$ML_REGISTRY_NAME/ml-core:$ML_IMAGE_TAG
image: $ML_REGISTRY_HOST/$ML_REGISTRY_NAME/ml-mem:$ML_IMAGE_TAG
image: $ML_REGISTRY_HOST/$ML_REGISTRY_NAME/ml-db:$ML_IMAGE_TAG
```

h) Execute the following script to store the Docker registry key as Kubernetes "Secret":

```
set-registry-key.sh
```

i) Execute the following script to store MOM host and key as Kubernetes "Secret":

```
set-mom-secret.sh create <MOM key> <MOM secret>
```

If you want to enable HTTPS or OAuth, see the section Customizing for Kubernetes for additional configuration steps.

j) Create storage classes for Mashery Local for Docker persistent stores:

```
set-storage-classes.sh
```

k) Create Mashery Local Traffic Manager service and Mashery Local Master service:

```
set-ml-services.sh
```

You can check the services with the following commands:

```
kubectl describe service ml-traffic-manager
```

```
kubectl describe service ml-master
```

The ml-traffic-manager is configured with load balancer. You can find the load balancer DNS name with the following command:

```
kubectl describe service ml-traffic-manager|grep Ingress|awk -F' ' '{print $3}'
```

The load balancer can also be found on the AWS EC2 dashboard **Load Balancers** list.

API invocation should be done solely via the AWS ELB (Elastic Load Balancer). The ELB configuration uses the *internal* IPs of the customer nodes for load balancing, so invoking API calls directly via the public IP addresses of the master or slave nodes is not an option.

l) Deploy Mashery Local master instance:

```
deploy-master-gcp.sh
```

You can check the ML instance pods with the command:

```
kubectl get pods
```

The ML master pod is named `ml-master-<name>`. When it's fully up, you should see 4/4 under the **READY** column with **STATUS** "Running" for the master instance pod.
You can check the startup init instance log with the following command:

```
kubectl exec -ti `kubectl get pods |grep ml-master |cut -d " " -f 1` -c ml-cm
-- cat /var/log/mashery/init-instance.log
```

When it's fully ready to serve traffic, you should see something like the following:

```
....

Register status: Content-Type: application/json Status: 200 {"results":
[{"results": [{"address": "10.0.22.98"}], "error": null}, {"results":
[{"area_name": "Roger"}], "error": null}, {"results": [{"credentials_updated":
true}], "error": null}, {"results": [{"name": "ml-master-4209822619-sxq40",
"id": 0}], "error": null}, {"results": [{"is_master": true}], "error": null}],
"error": null}

****  04/06 05:27:38  Register instance succeeded

Load service result:

Load service result:

Load service result: 70a0b42e-2b9a-4f60-a4d6-8c5503894043 [SERVICES] 04/06/17
05:27:45 - 04/06/17 05:27:47: 254 records (Success) 70a0b42e-2b9a-4f60-
a4d6-8c5503894043 [KEYS] 04/06/17 05:27:47 - 04/06/17 05:27:55: 10963 records
(Success) 70a0b42e-2b9a-4f60-a4d6-8c5503894043 [APPS] 04/06/17 05:27:55 -
04/06/17 05:28:23: 6884 records (Success) 70a0b42e-2b9a-4f60-a4d6-8c5503894043
[CLASSES] 04/06/17 05:28:23 - 04/06/17 05:28:23: 0 records (Success)
70a0b42e-2b9a-4f60-a4d6-8c5503894043 [PACKAGES] 04/06/17 05:28:23 - 04/06/17
05:29:54: 28824 records (Success) 70a0b42e-2b9a-4f60-a4d6-8c5503894043
[PACKAGEKEYS] 04/06/17 05:29:54 - 04/06/17 05:30:17: 5553 records (Success)

****  04/06 05:30:17  Service info loaded

Load cache output first ten lines: - Trying to load mapi data for spkey:
m8hxx3wxy5wjyjhfzc328wqh key: MAPI_m8hxx3wxy5wjyjhfzc328wqh::
2011w25DeveloperJay key: MAPI_m8hxx3wxy5wjyjhfzc328wqh::2011w25DeveloperRoger
key: MAPI_m8hxx3wxy5wjyjhfzc328wqh::3skjegt4ddpam6a5r8sfgpkz key:
MAPI_m8hxx3wxy5wjyjhfzc328wqh::4q5t7z4gduy388z9nk5tmptm key:
MAPI_m8hxx3wxy5wjyjhfzc328wqh::4tzw5p5h5mx8gr8ez6m34wak key:
MAPI_m8hxx3wxy5wjyjhfzc328wqh::5s8ds7dcyj7cjz4h9h5tv7ev key:
MAPI_m8hxx3wxy5wjyjhfzc328wqh::5yy6dkjbq7sr922j4wt6u2hc key:
MAPI_m8hxx3wxy5wjyjhfzc328wqh::6mbcz48nabrz682xn2hdmhzn key:
MAPI_m8hxx3wxy5wjyjhfzc328wqh::8tng6tk5bzhpfqexn525cqnj

****  04/06 05:31:01  Cache Loaded

****  04/06 05:31:01  Ping Traffic Manager succeeded

****  04/06 05:31:01  Setting status ready
```

When the ML master instance containers are up, you can find the ML master instance node public IP with the following command:

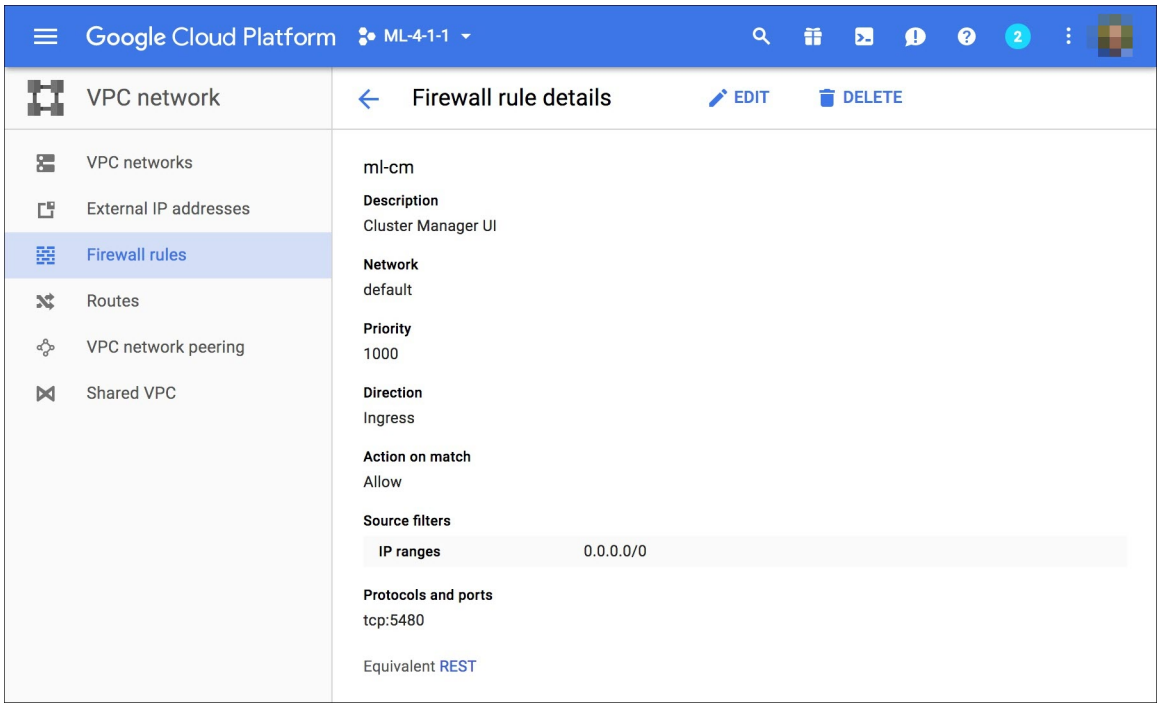
```
kubectl describe node `kubectl get pods -o wide |grep ml-master |awk -F' ' 'ml-
master'`|\
awk '/Addresses/ {for(i=1; i<=6; i++) {getline; print}{print "\n"}}'


  InternalIP:   10.138.0.2
  ExternalIP:   104.198.13.169
  Hostname:     gke-rkdemo-ml411-default-pool-ca67e8bf-c9vp
Capacity:
 cpu:           2
 memory:        7664944Ki


  InternalIP:   10.138.0.4
```

```
  ExternalIP:   35.197.13.37
  Hostname:     gke-rkdemo-ml411-default-pool-ca67e8bf-f5s0
Capacity:
 cpu:           2
 memory:        7664944Ki


  InternalIP:   10.138.0.3
  ExternalIP:   35.185.213.163
  Hostname:     gke-rkdemo-ml411-default-pool-ca67e8bf-x9g4
Capacity:
 cpu:           2
 memory:        7664944Ki
```

If you need to access the Mashery Local instance Cluster Manager UI, you need to open the port 5480 for UI access. For convenience, you can open the port for all minion nodes in the cluster with the following GCP command:

```
gcloud compute firewall-rules create
```

Alternatively, you can create the firewall rule in the GCP console. For more information, see https://cloud.google.com/sdk/gcloud/reference/compute/firewall-rules/create.



Then you can login to the ML master instance Cluster Manager UI with https://< ML master instance node ip>:5480.

You can get into any ML master instance container with the following command:

```
kubectl exec -ti `kubectl get pods |grep ml-master |cut -d " " -f 1` -c
<container name> -- /bin/bash
```

The container names are: ml-db, ml-mem, ml-tm, ml-cm.

You can also execute some simple remote command on a container directly:

```
kubectl exec -ti `kubectl get pods |grep ml-master |cut -d " " -f 1` -c
<container name> -- <remote command>

for example:

kubectl exec -ti `kubectl get pods |grep ml-master |cut -d " " -f 1` -c ml-tm
-- ls -l /var/log/trafficmgr/access
```

At any time, you could also get in the Kubernetes dashboard UI to check the progress, such as checking the deployment, replica sets, services, pods, containers and their logs.

m) Copy `deploy-slaves.sh` to `deploy-slaves-gcp.sh`. Edit `deploy-slaves-gcp.sh` and change all filename references from:

```
cat >> ml-slave.yaml << EOF
```

to:

```
cat >> ml-slave-gcp.yaml << EOF
```

n) Additionally, change all image registry references from:

```
image: $ML_REGISTRY_HOST/$ML_REGISTRY_NAME:ml-core-$ML_IMAGE_TAG
image: $ML_REGISTRY_HOST/$ML_REGISTRY_NAME:ml-mem-$ML_IMAGE_TAG
image: $ML_REGISTRY_HOST/$ML_REGISTRY_NAME:ml-db-$ML_IMAGE_TAG
```

to:

```
image: $ML_REGISTRY_HOST/$ML_REGISTRY_NAME/ml-core:$ML_IMAGE_TAG
image: $ML_REGISTRY_HOST/$ML_REGISTRY_NAME/ml-mem:$ML_IMAGE_TAG
image: $ML_REGISTRY_HOST/$ML_REGISTRY_NAME/ml-db:$ML_IMAGE_TAG
```

o) Deploy Mashery Local slave instances:

```
deploy-slaves-gcp.sh
```

You can check the Mashery Local instance pods with the command:

```
kubectl get pods
```

The Mashery Local slaves instance pods are named with ml-slave-0, ml-slave-1, ml-slave-2.

When it's fully up, you should see 4/4 under the **READY** column with **STATUS** "Running" for the slave instance pod.

You can check the startup init instance log with the following command:

```
kubectl exec -ti `kubectl get pods |grep <slave pod name> |cut -d " " -f 1` -c
ml-cm -- cat /var/log/mashery/init-instance.log

for example:

kubectl exec -ti `kubectl get pods |grep ml-slave-0 |cut -d " " -f 1` -c ml-cm
-- cat /var/log/mashery/init-instance.log
```

You can find the Mashery Local slave instance node IP with the following command:

```
kubectl describe node `kubectl get pods -o wide |grep <slave pod name> |awk -
F' ' '{print $7}'` |grep Addresses |cut -d "," -f 3
```

Then, login to the ML slave instance Cluster Manager UI with https://<ML slave instance node ip>:5480

If you didn't open the port 5480 for all nodes in the previous step, you need to open the port for each ML slave instance individually with additional security group through AWS UI or CLI.

You can get into any ML slave instance container with the following command:

```
kubectl exec -ti `kubectl get pods |grep <slave pod name> |cut -d " " -f 1` -c
<container name> -- /bin/bash
```

The container names are: ml-db, ml-mem, ml-tm, ml-cm.

You can also execute some simple remote command on a container directly:

```
kubectl exec -ti `kubectl get pods |grep <slave pod name> |cut -d " " -f 1` -c
<container name> -- <remote command>

for example:

kubectl exec -ti `kubectl get pods |grep ml-slave-0 |cut -d " " -f 1` -c ml-tm
-- ls -l /var/log/trafficmgr/access
```

At any time, you could also get into the Kubernetes dashboard UI to check the progress, such as checking the stateful sets, services, pods, and containers and their logs.

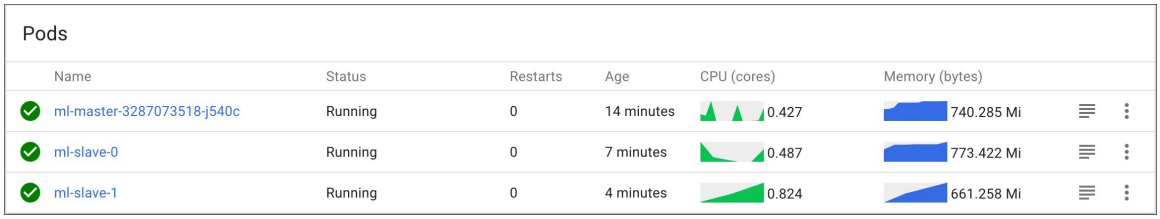
By default, it's configured to run two slave instances.

You can use the following command to increase or reduce the number of slaves:

```
kubectl patch statefulset ml-slave --type='json' -p='[{"op": "replace",
"path": "/spec/replicas", "value":<the desired replica number>}]'
```

However, you must have enough worker nodes to run all the slave instances.

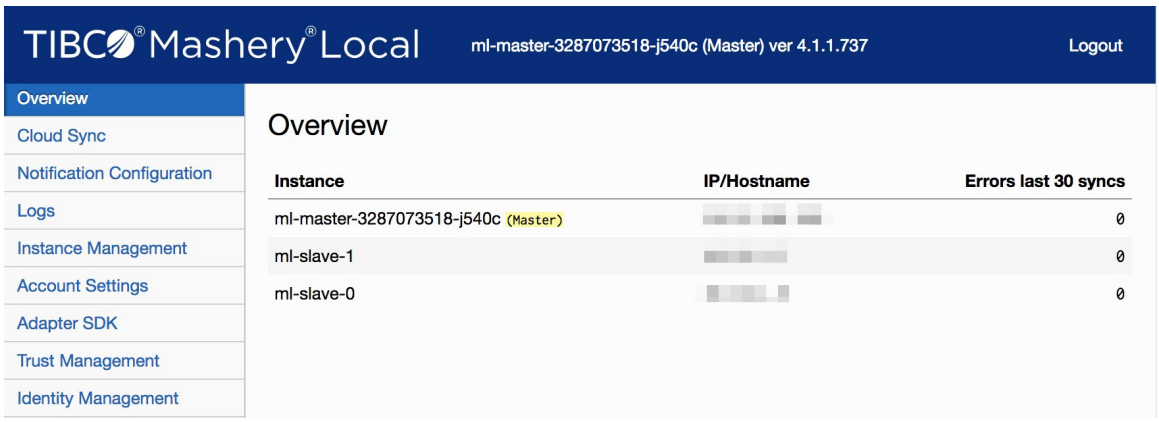
p) If everything works properly, you should have a fully deployed cluster with one master and two slaves.



q) You can also use the following command (the values under the READY column should all read 4/4):

```
kubectl get pods

NAME                              READY     STATUS    RESTARTS   AGE
ml-master-3287073518-j540c        4/4       Running   0          26m
ml-slave-0                        4/4       Running   0          19m
ml-slave-1                        4/4       Running   0          15m
```

r) Finally, the Mashery Local Cluster Manager console should look like this:



Invoking APIs via GCP Load Balancer

Be sure to add the LB host name or IP address to your endpoint's domain list and synchronize the Mashery Local cluster. To find the LB IP address, execute the command:

```
export LB=`kubectl describe service ml-traffic-manager|grep Ingress|awk -F'
' '{print $3}'` && echo $LB
```

Notes for Docker Installations Using GCP

The following section are additional notes for Docker installations using GCP.

- The process/steps for pulling the image can be done manually as indicated. However, if you skip this step, it will be done automatically when running the `deploy-master-gcp` script. Note that the spin-up time will be slightly longer since all the images will need to be pulled as a part of the deployment process. So you'll see "CreatingContainer" for a few minutes until anything happens. Logs in local dashboard for the deployment will track progress on the pull requests
- The `deploy-slaves-gcp` script is based on the Master deployment script, which has persistent storage. This is ideal for a Master, since it handles synchronizations between MOM and also

contains local settings around sync times, etc. This may not be ideal for the Slave deployment script, particularly if you're testing or switching between different Mashery clusters within a K8s cluster. The persistent storage for Slaves will keep the original MOM Key/Secret used during the first deployment, which can lead to issues later on, if you switch to a new Master or cluster (for another POC/environment). You can always just manually delete the PVs (persistent volumes) for the Slaves but long term non-persistent seems to be the best option for Slaves.

- Init logs for the containers (`ML-CM`, in particular) under `/var/log/mashery/` are your most helpful way for troubleshooting why a container may not be fully initializing. For example, looking at these logs were used to determine that while a key/secret were being passed to MOM for authentication, the process was ultimately failing. This may have been due to explicitly defining a MOM-Host during the set-mom-secret script process. Deleting the existing secret, and re-creating/replacing it, helped the container to initialize without issues.
- During initialization of the containers you may, almost immediately, see warnings or errors of persistent volume health checks failing. This is caused by health-check scripts running a little too quickly, which draws these flags. Give the containers a little more time (2 to 4 minutes on average for a decently equipped K8s cluster). You can refresh your dashboard or check "kubectl get pods" to see if all 4 containers come on line. The ML-CM (4th container) tends to take a little longer than the rest.

Configuring the Mashery Local Cluster

Mashery Local may run configured in a cluster of one master and multiple slaves.

To configure the Mashery Local cluster, you need to:

- Configure a Mashery local master
- Configure slave(s) to the local master

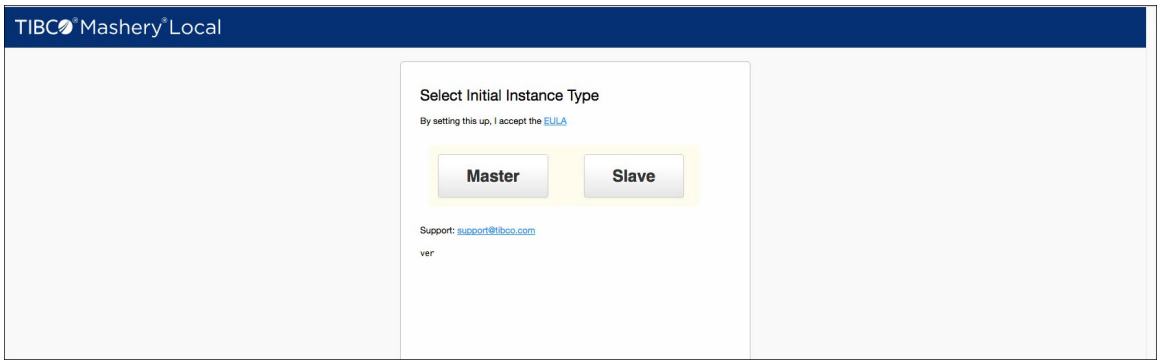
If you run Mashery Local with Kubernetes, the Master and Slave configuration are done automatically.

Configuring a Mashery Local Master

To configure a Mashery Local master:

Procedure

1. Browse to the Mashery Local Cluster Manager of the master by using the Docker Host IP address of the instance:
   `https://<IP_address_of_instance>:5480`
2. Login with username **administrator** and the password configured in **set-user-variables.sh**.
   Click **Master**.



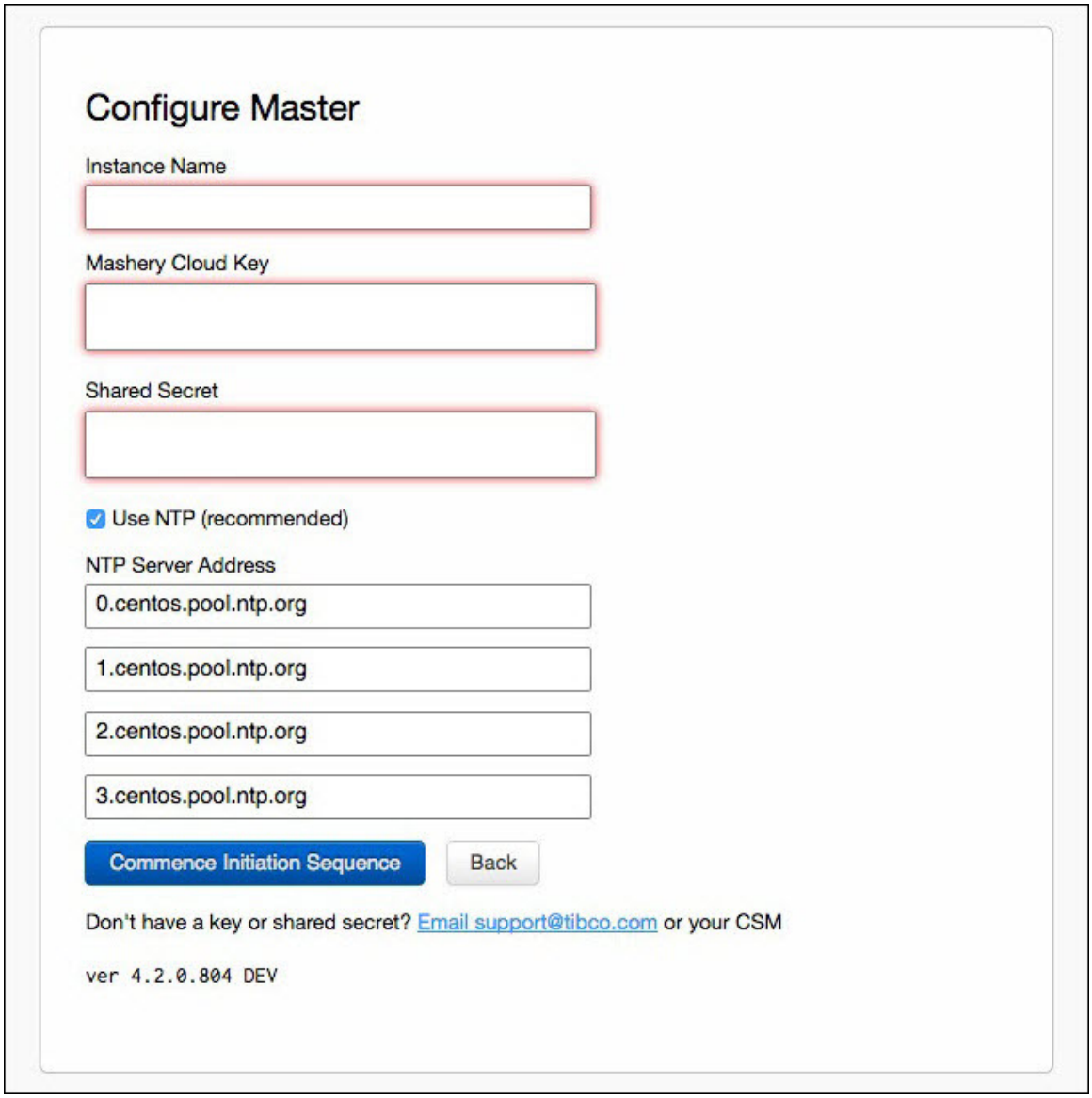
The **Configure Master** window appears.

Enter an instance name (this name will eventually display in the Mashery Admin Dashboard) that is meaningful to your operation, the Mashery Cloud Key and shared secret provided by TIBCO Mashery, and the NTP server address, if used.

The **Instance Name** you choose should be a unique name; other Master or Slaves instances in your cluster cannot be given the same name.

The **Use NTP (recommended)** checkbox is selected by default, and four NTP servers can be configured.

If you have multiple clusters, the Mashery Cloud Key and shared secret provided by TIBCO Mashery should be unique to each of your clusters. Mashery Local clusters should not share keys.

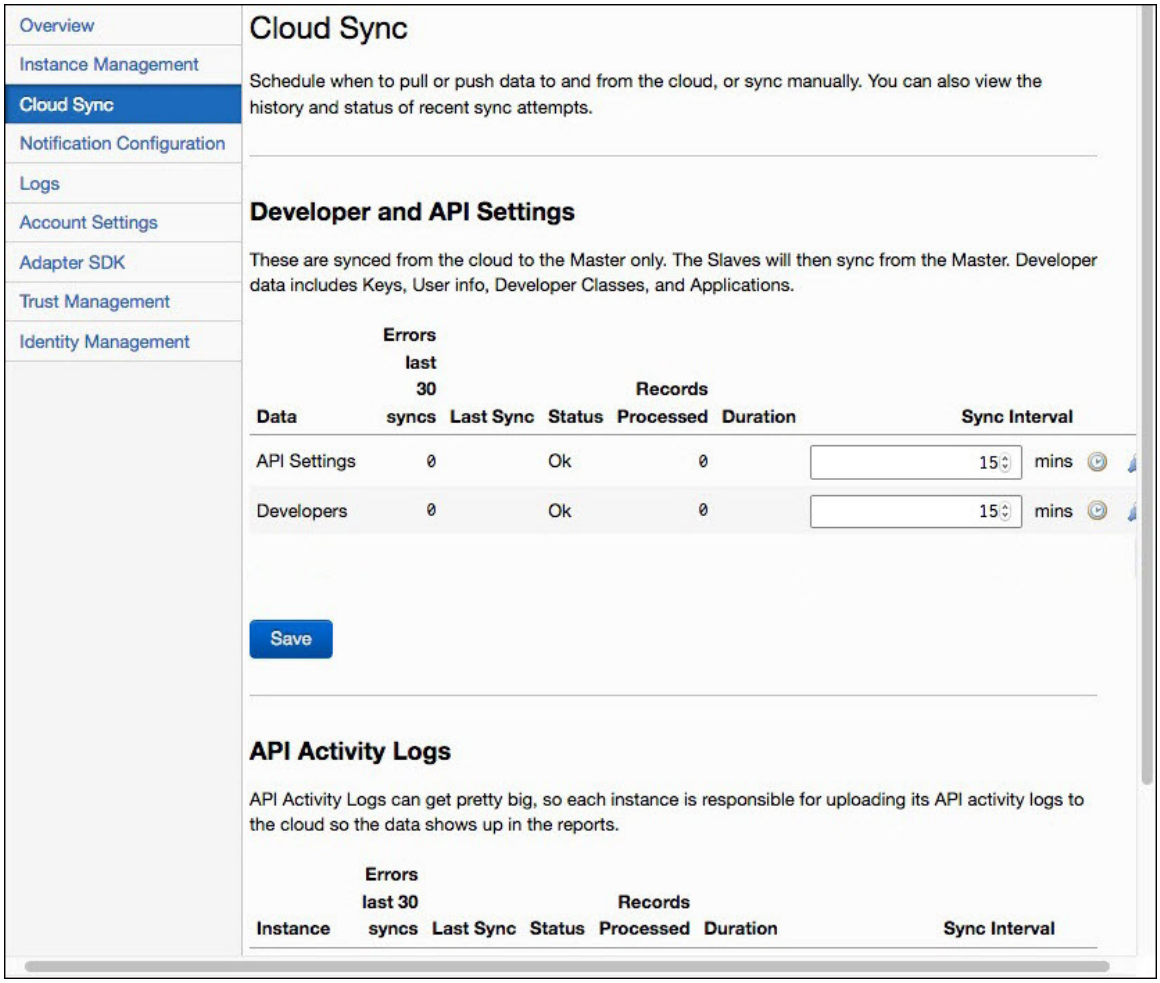


3. Click **Commence Initiation Sequence**.

   After the Master initializes with the Mashery cloud service, a completion page appears.

4. Click **Continue**.

5. Navigate to the **Cloud Sync** page and perform manual syncs for **API Settings** and **Developers** by clicking the adjacent icons:



6. Test the instance as described in Testing a New Instance.
7. See the instructions in Advanced Configuration for how to enable notifications, if desired.

Configuring Slaves to the Local Master

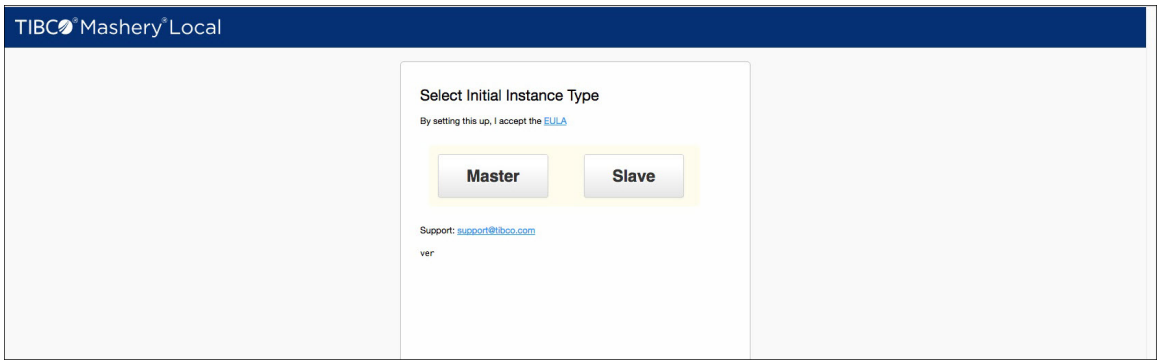
Mashery Local may run configured in a cluster of one master and multiple slaves.
To configure slaves to the master:

Procedure

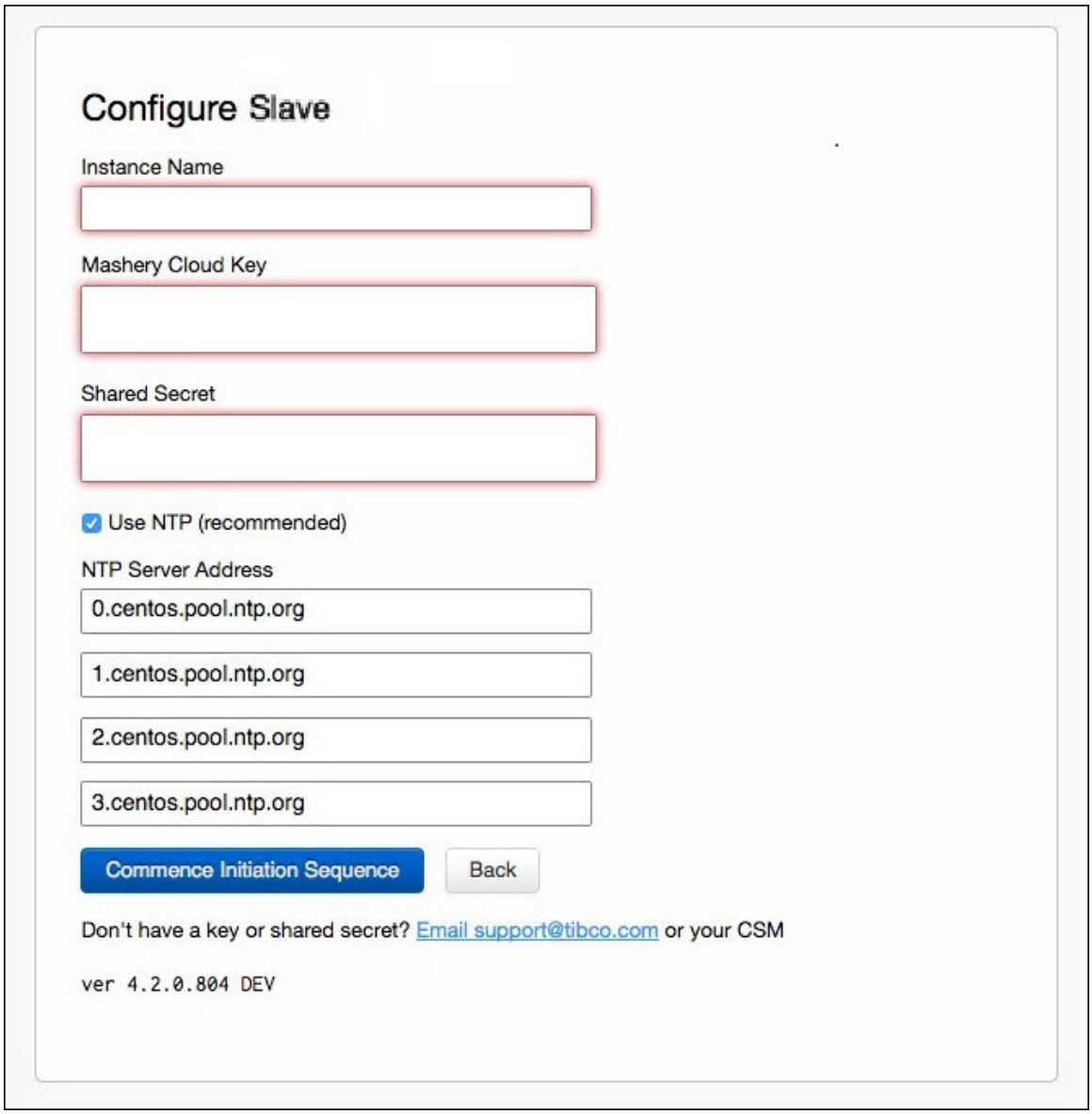
1. Browse to the Mashery Local Cluster Manager of the slave by using the Docker Host IP address of the instance:

   `https://<IP_address_of_instance>:5480`

2. Login with username **administrator** and the password provided by TIBCO Mashery.
3. Click **Slave**.



4. Enter an instance name (this name will eventually display in the Mashery Admin Dashboard) that is meaningful to your operation, the Mashery Cloud Key and shared secret provided by TIBCO Mashery, and the NTP server address, if used.



The **Use NTP (recommended)** checkbox is selected by default, and four NTP servers can be configured.

5. Click **Register with Mashery and Master**.
6. Click **Continue**.
7. Test the instances as described in Testing a New Instance.
8. See the instructions in Advanced Configuration for how to enable notifications, and API and JMX reporting access, if desired.

Configuring the Load Balancer

TIBCO Mashery recommends using a Load Balancer to best utilize the cluster, although this is not required because you may route your API traffic directly to each instance.

Each instance hosts a service called /mashping. Configure the Load Balancer to access the following address, without the host header:

```
http://<IP_address_of_instance>/mashping
```

If the Load Balancer and the cluster is working correctly, /mashping returns the following response:

```
HTTP/1.1 200 OK
Server: Mashery Proxy
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
{"status":200,"time":1315510300,"message":"success"}
```

If /mashping returns any other response, then the load balancer should remove the instance from the cluster and either retry after a period of time or alert operations to investigate.

Mashery Local has two instance types: Master and Slave. Should the Load Balancer pull the Master out of the cluster pool, an Operations engineer should immediately investigate whether it can be recovered, and, if not, promote a Slave to Master. Taking offending Slaves out of rotation through the Load Balancer can mitigate any traffic impact. If no Master exists in the pool, data synchronization with the Mashery Cloud Service will not occur with the exception of API event activity. Access Tokens, Keys, Applications, Classes and Services will not be synchronized.

For steps on how to promote a Slave to Master, see Promoting a Slave to Master.

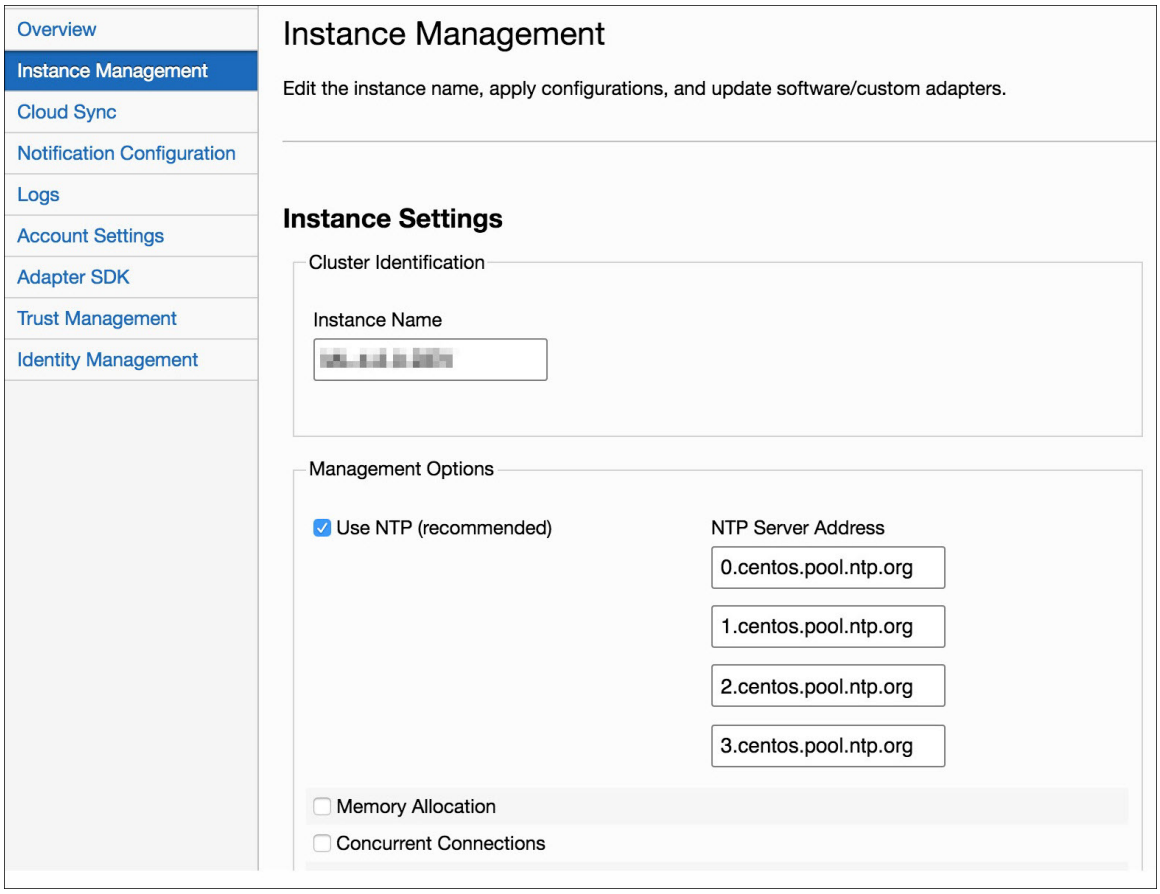
Configuring the Instance

The **Instance Management** tab allows you to configure additional settings for that particular instance. You can edit the instance name, configure instance settings, and update software and custom adapters. Additional system-level parameters can be tuned here such as application memory allocation, configuration cache size, maximum concurrent connections, and connection pool size for the database.

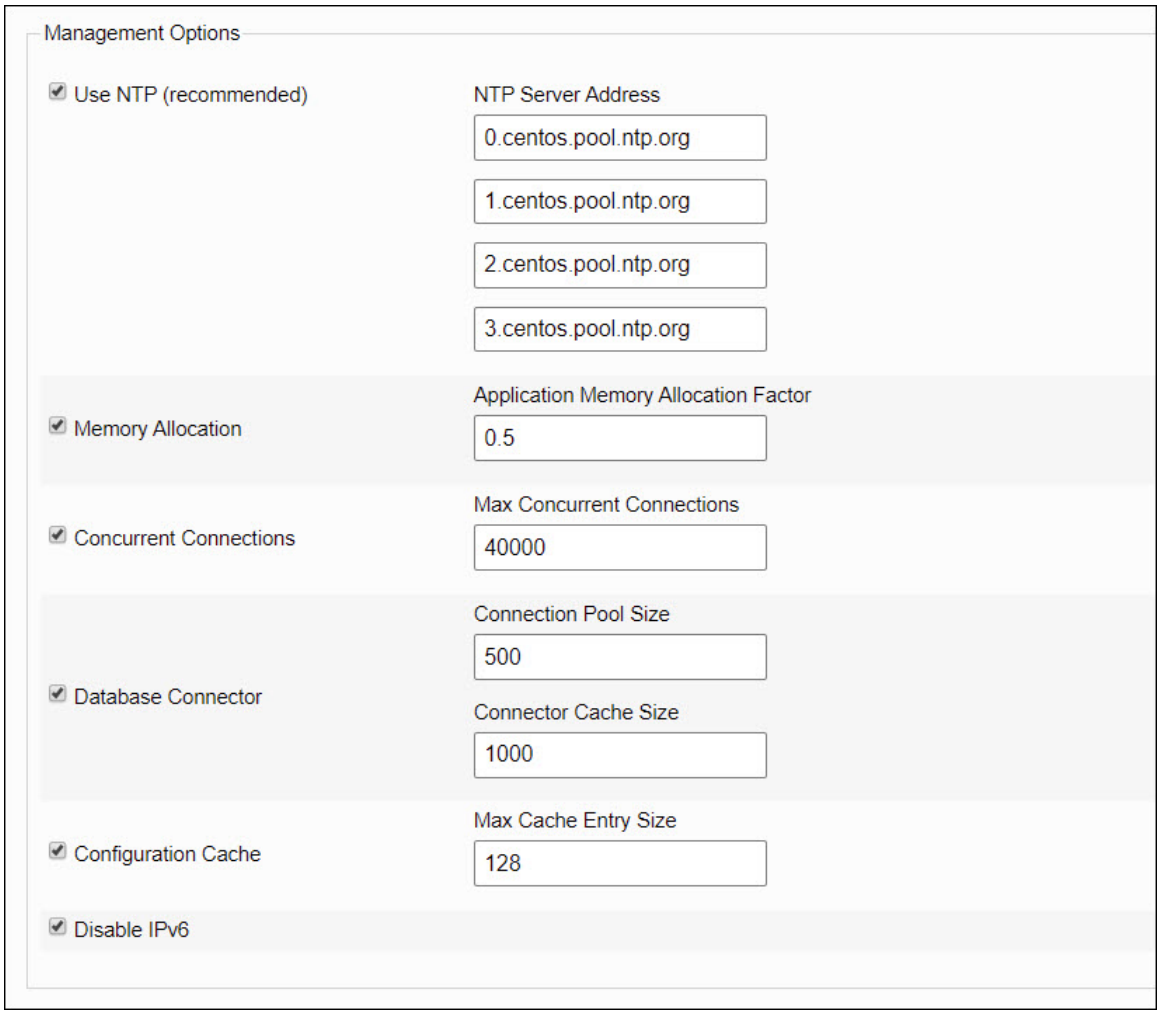
To configure an instance:

Procedure

1. Click **Instance Management**.



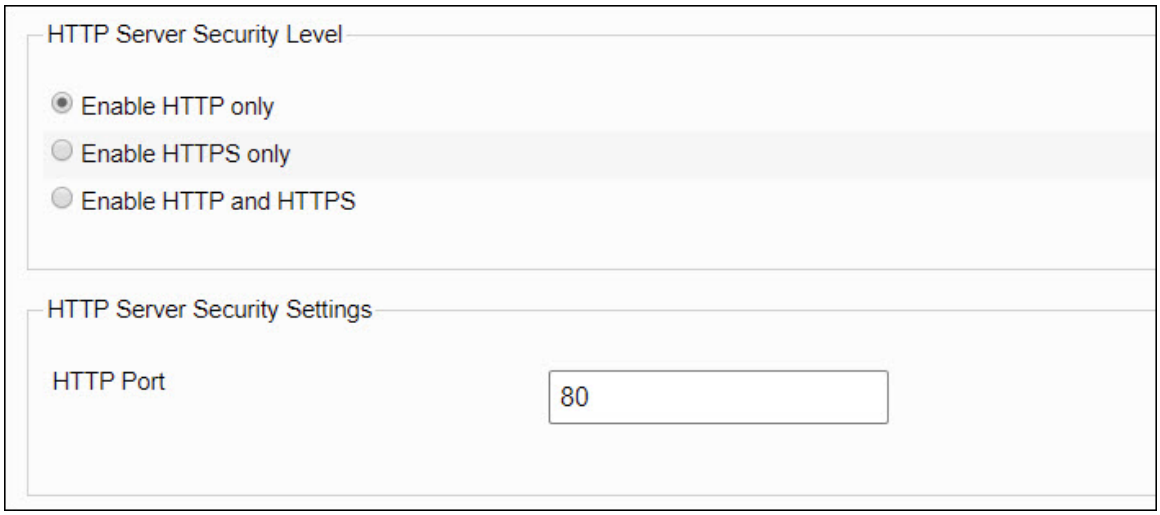
2. Click the **Management Options** for which you want to configure the settings. A text box is displayed for the selected **Management Options.**
3. Enter the details for the following fields to configure the instance.



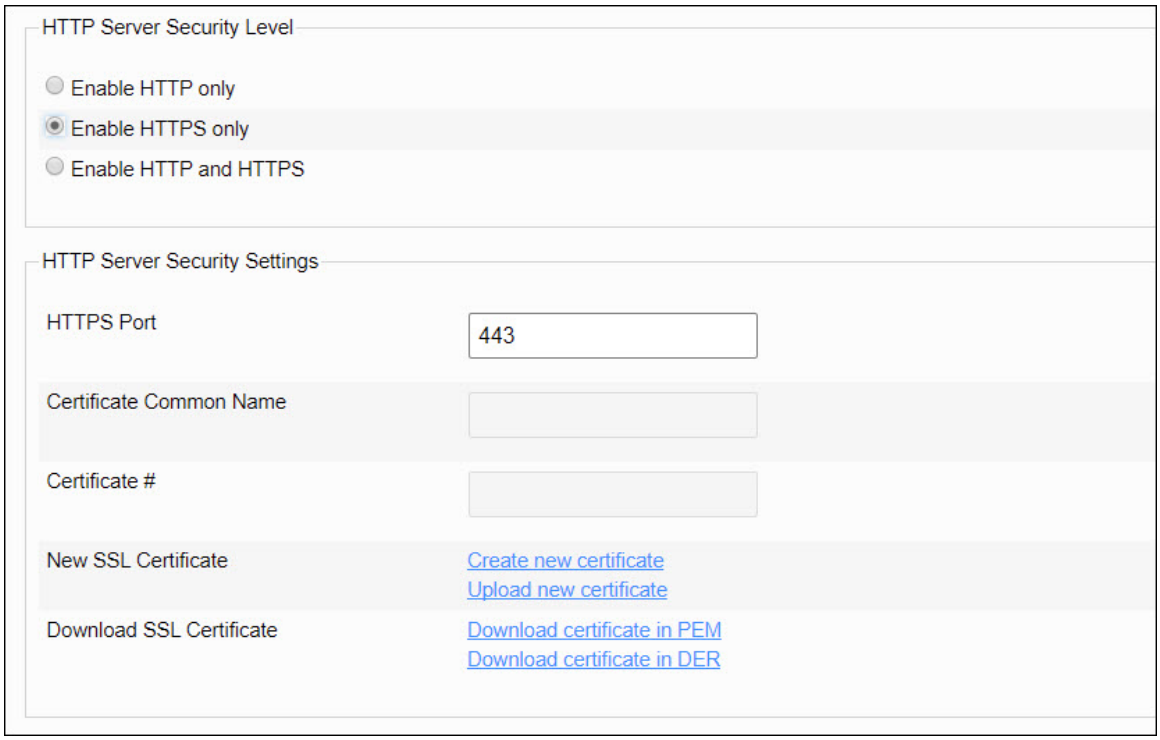
Field	Description
Use NTP (recommended)	Enabled by default. Specify one to four NTP server addresses. NTP Address for time synchronizing is required if using NTP.
Memory Allocation	Specify application memory size as a fraction of the available memory, between 0 and 1.
Concurrent Connections	Sets the maximum number of concurrent connections to this service instance, must be at least 1.
Database Connector	The **Connection Pool Size** sets the maximum number of concurrent connections this instance will make to its database, between 1 and 100,000. The **Connector Cache Size** sets the cache size for the data connector, between 1 and 524,288.

Field	Description
Configuration Cache	Specify the maximum entry size (in MB) for configuration cache, at least 1.
Disable IPv6	Select this option to disable IPv6 if IPv6 traffic should not be allowed to the backend. By default, Mashery Local supports both IPv4 and IPv6.

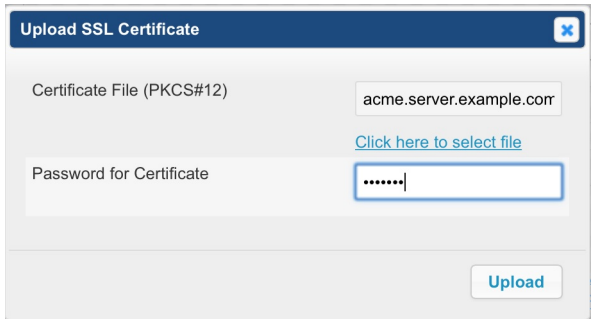
4. Select the appropriate **HTTP Server Security Level**:

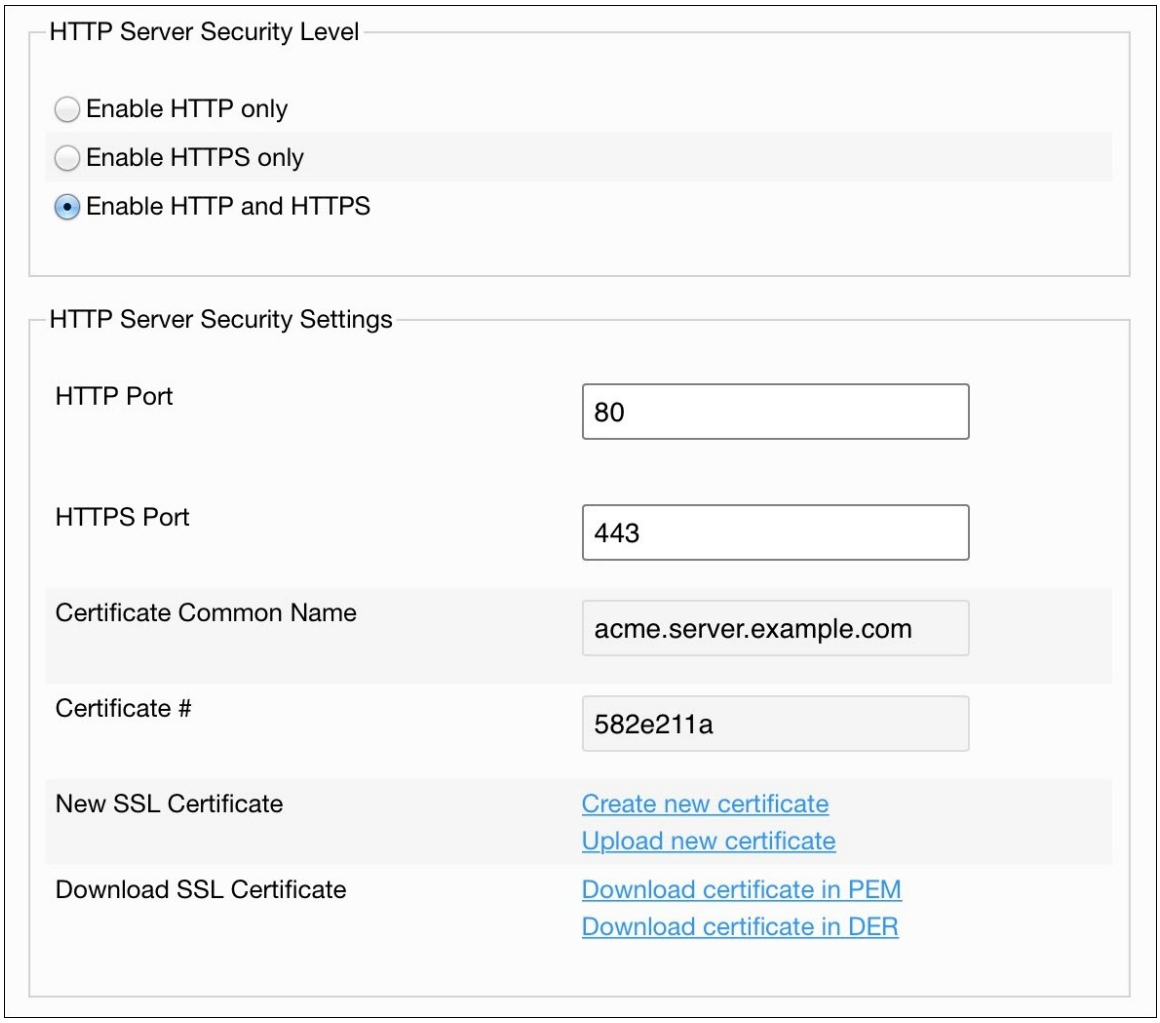


- **Enable HTTP only**: If selected, the default **HTTP Port** for **HTTP Server Security Settings** is `80`.



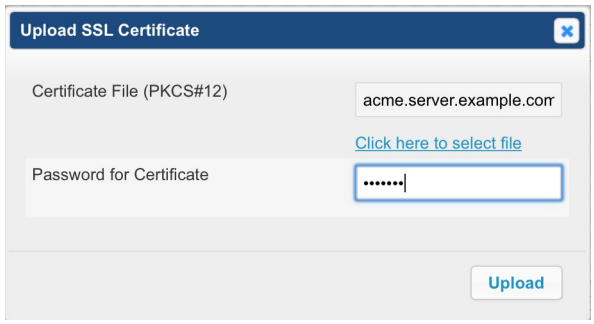
- **Enable HTTPS only**: If selected, enter the details for the following fields:

Field	Description
HTTPS Port	Specify the HTTPS port. The default is `443`.
Certificate Common Name (display only)	Automatically displays the name of the selected certificate.
Certificate # (display only)	Automatically displays the number of the selected certificate.
New SSL Certificate	Select from: • **Create new certificate**: If selected, enter a Certificate Common name in the **Create SSL Certificate** window, then click **Create**.  • **Upload new certificate**: If selected, in the **Upload SSL Certificate** window, browse to the SSL certificate using the **Click here to select file** link, enter the **Password for Certificate**, then click **Upload**. 
Download SSL Certificate	Select from: • **Download certificate in PEM**: downloads the current certificate in PEM format. • **Download certificate in DER**: downloads the current certificate in DER format.



- **Enable HTTP and HTTPS**: If selected, enter the details for the following fields:

Field	Description
HTTP Port	Specify the HTTP port. The default is `80`.
HTTPS Port	Specify the HTTPS port. The default is `443`.
Certificate Common Name (display only)	Displays the name of the selected certificate.
Certificate # (display only)	Displays the number of the selected certificate.

Field	Description
New SSL Certificate	Select from: • **Create new certificate**: If selected, enter a Certificate Common name in the **Create SSL Certificate** window, then click **Create**.  • **Upload new certificate**: If selected, in the **Upload SSL Certificate** window, browse to the SSL certificate using the **Click here to select file** link, enter the **Password for Certificate**, then click **Upload**. 
Download SSL Certificate	Select from: • **Download certificate in PEM**: downloads the current certificate in PEM format. • **Download certificate in DER**: downloads the current certificate in DER format.

5. Click **Save**.

   You may be reminded that Mashery Local needs to restart proxy service.

   The instance is configured for the specified settings.

What to do next

For detailed steps on setting up HTTPS Server, please refer to the following sections in the Appendix:

- Setting up HTTPS Server using Self-Signed Certificate
- Setting up HTTPS Server using Customer-Provided Certificate

Shutting Down a Master

The following section describes how to shut down a Master in a Mashery Local Cluster.

Procedure

- Use `docker-compose down` to shut down the Master.

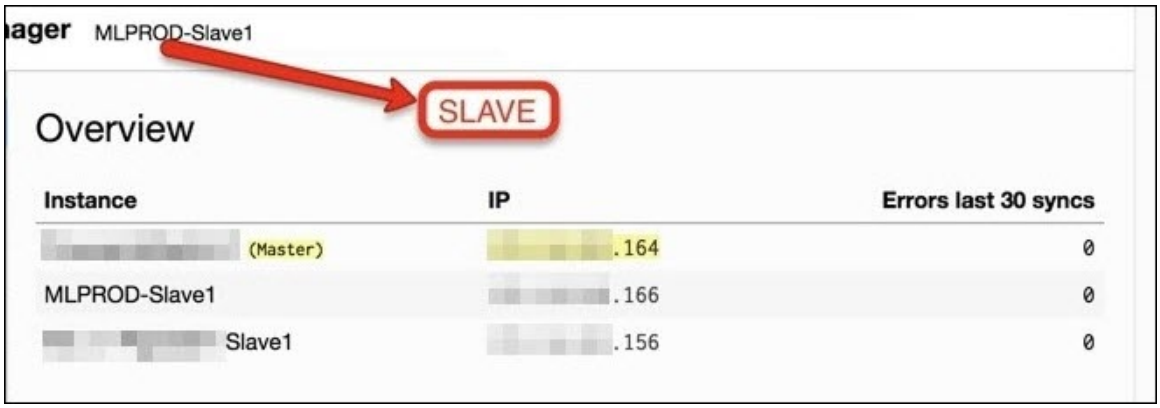
Promoting a Slave to Master

Promoting a Slave to Master is important from within a cluster, and having multiple clusters (using unique MoM keys) connecting to the same area is High Availability. Taking offending Slaves out of rotation through the Load Balancer can also mitigate any traffic impact.

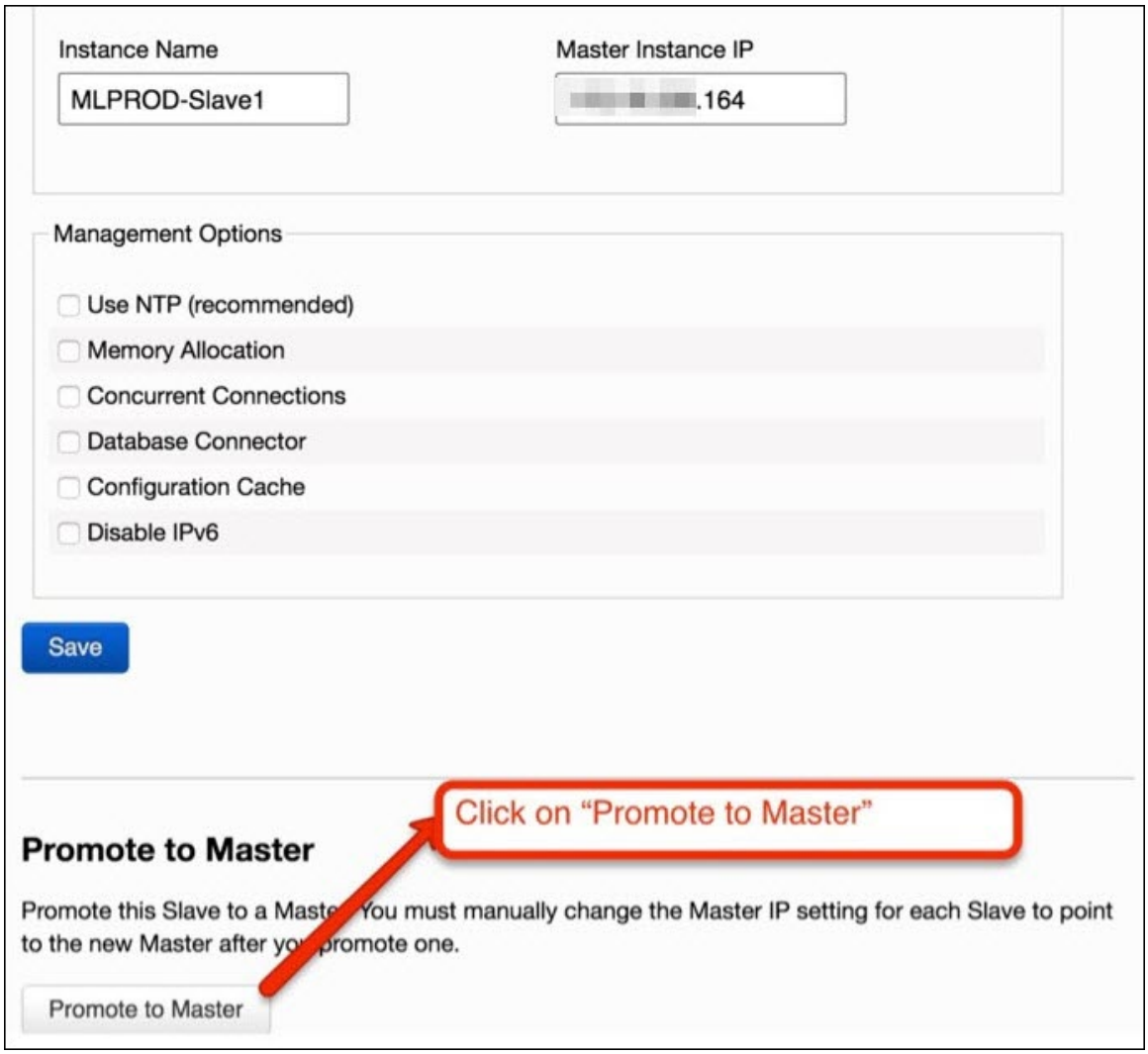
To promote a Slave to Master:

Procedure

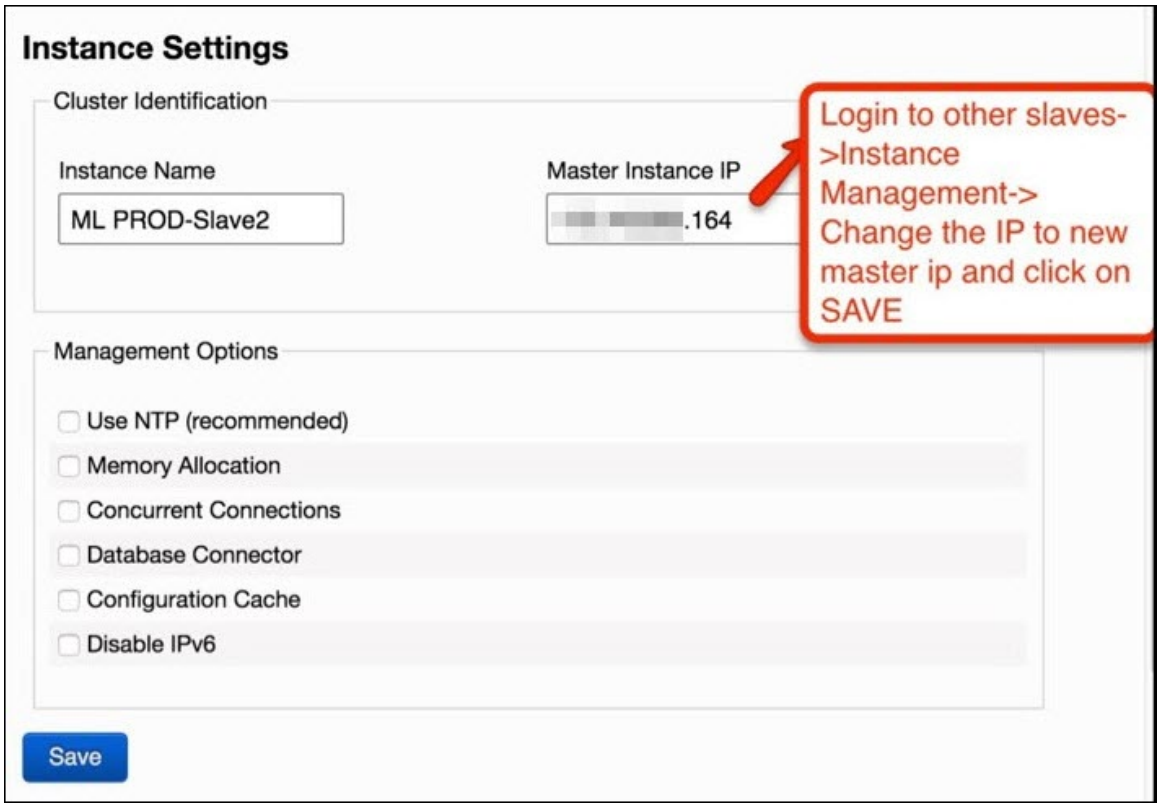
1. Log into the Slave instance.



2. Click **Instance Management**.
3. In the **Promote to Master** section, click **Promote to Master**.



4. Log into the other Slaves, go to **Instance Settings** in **Instance Management**, and in the change the **Master Instance IP** address to the new Master's IP address.



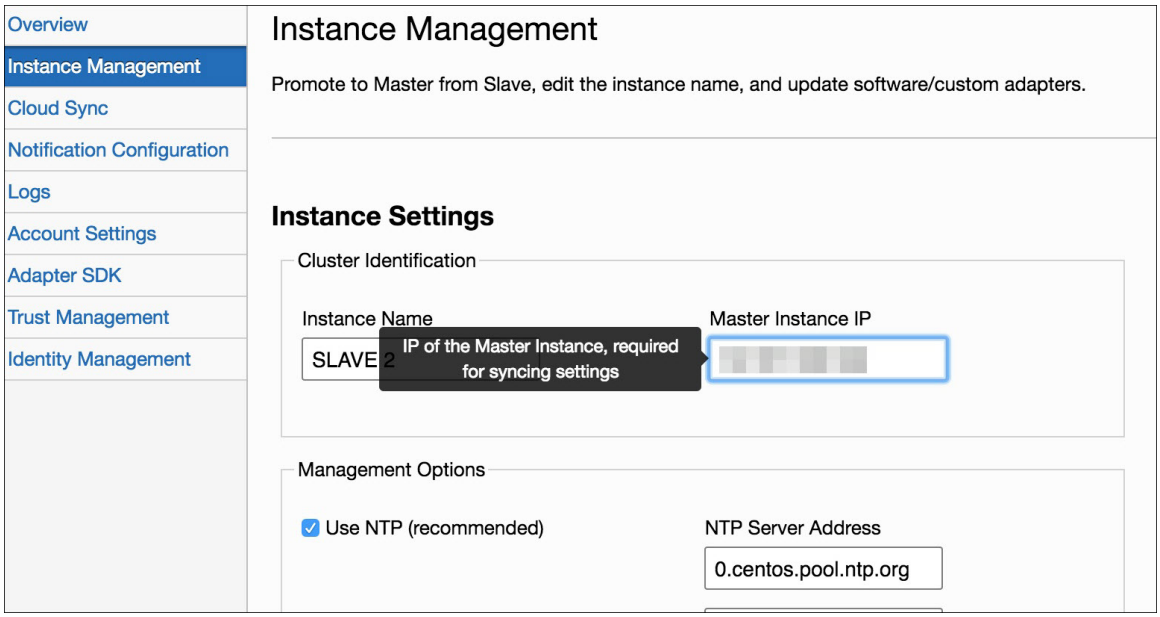
5. (Optional) Delete the old Master or shut down that Virtual Machine.

Repointing Other Slaves to a New Master

The following section describes how to repoint other slaves to a new Master.

Procedure

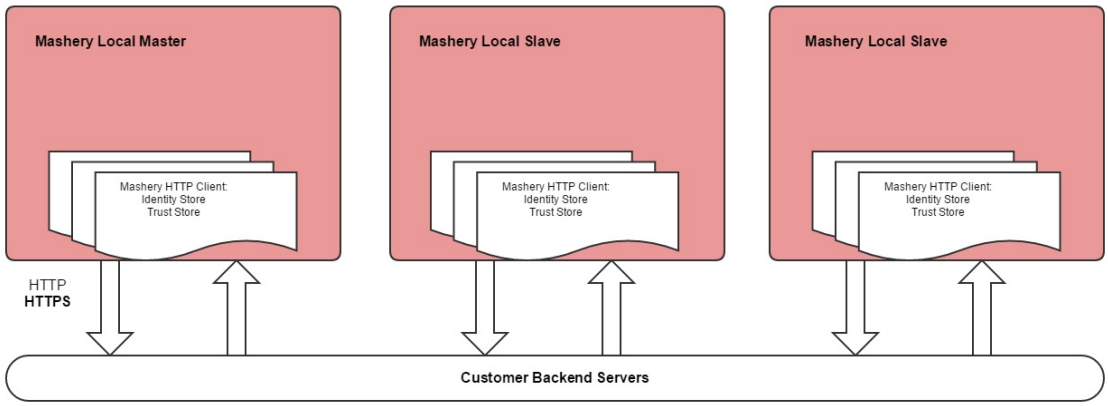
1. Log into the Slave Instance.
2. On the **TIBCO Mashery Local** page, click **Instance Management**.



3. In the **Instance Settings** section, enter the IP address of the new Master in the **Master Instance IP** field.
4. Click **Save**.

HTTPS Client Feature Overview

The HTTP Client in Mashery Local is used for connecting to customer backend servers.



The HTTPS Client supports the following features:

- Verification of backend server certificate
- HTTPS Client authentication (Mutual SSL Authentication)

These features are configured via the HTTPS Client Profile feature. With an HTTPS Client profile, you can configure one or more trusted CA certificates. These CA certificates are used for verifying backend customer server certificates. You can configure only one identity, which will be used in HTTPS Client Authentication (Mutual SSL Authentication). The HTTPS Client profile can be applied to one or more endpoints.

Summary of How to Use the HTTPS Client Profile Feature

The HTTPS Client Profile feature is implemented as follows:

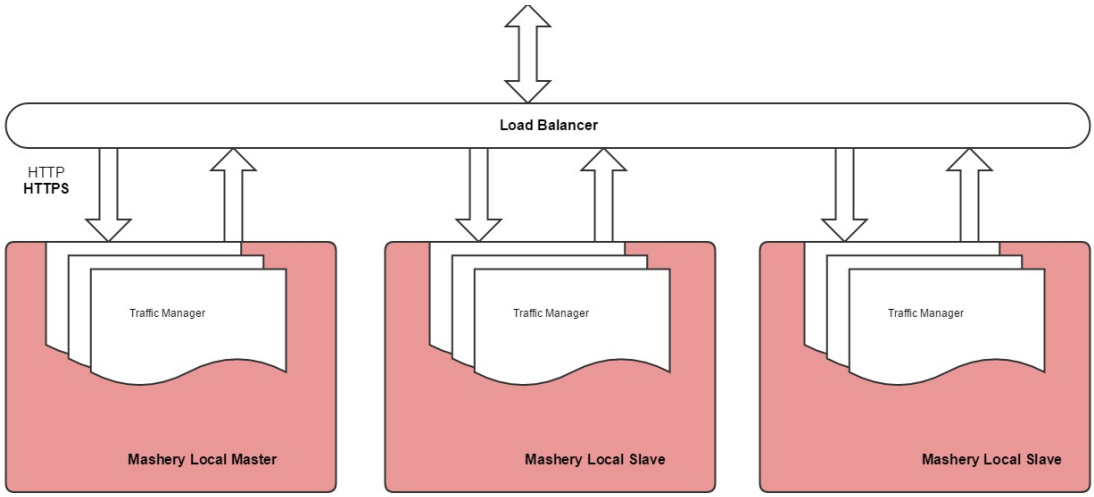
1. Certificates and identities are uploaded or updated in Mashery Local using Mashery Cluster Manager.
2. Metadata of certificates and identity are synchronized to Mashery SaaS (Control Center). Then, they are available for creating HTTPS Client profiles in Mashery SaaS.
3. An HTTPS Client profile is created In Mashery SaaS, and the profile is applied to the endpoint.
4. The HTTPS Client Profile is synchronized to Mashery Local, then the list of certificates and one identity will be used in HTTPS Client connection to backend servers.

For detailed steps on setting up the HTTPS Client feature, please refer to following sections in the Appendix:

- Configuring and using the HTTPS Client Feature without Mutual Authentication
- Configuring and using the HTTPS Client Feature with Mutual Authentication

HTTPS Server Feature Overview

Mashery Local supports HTTPS requests. The following picture illustrates a typical TIBCO Mashery Local deployment, which consists of one master node and two slave nodes. The communication between the Load Balancer and Mashery Local nodes can be either HTTP or HTTPS.



Administrators can configure the certificates and port number for the Mashery Local HTTPS Server. Mashery Local supports using self-signed certificates or customer-provided certificates. The configuration for certificates and port number must be done on a per-node basis using Mashery Cluster Manager.

For detailed steps on setting up an HTTPS Server, please refer to following sections in the Appendix:

- Setting up HTTPS Server using Self-Signed Certificate
- Setting up HTTPS Server using Customer-Provided Certificate

Advanced Configuration and Maintenance

This section describes how you can extend your installation by adding the following capabilities:

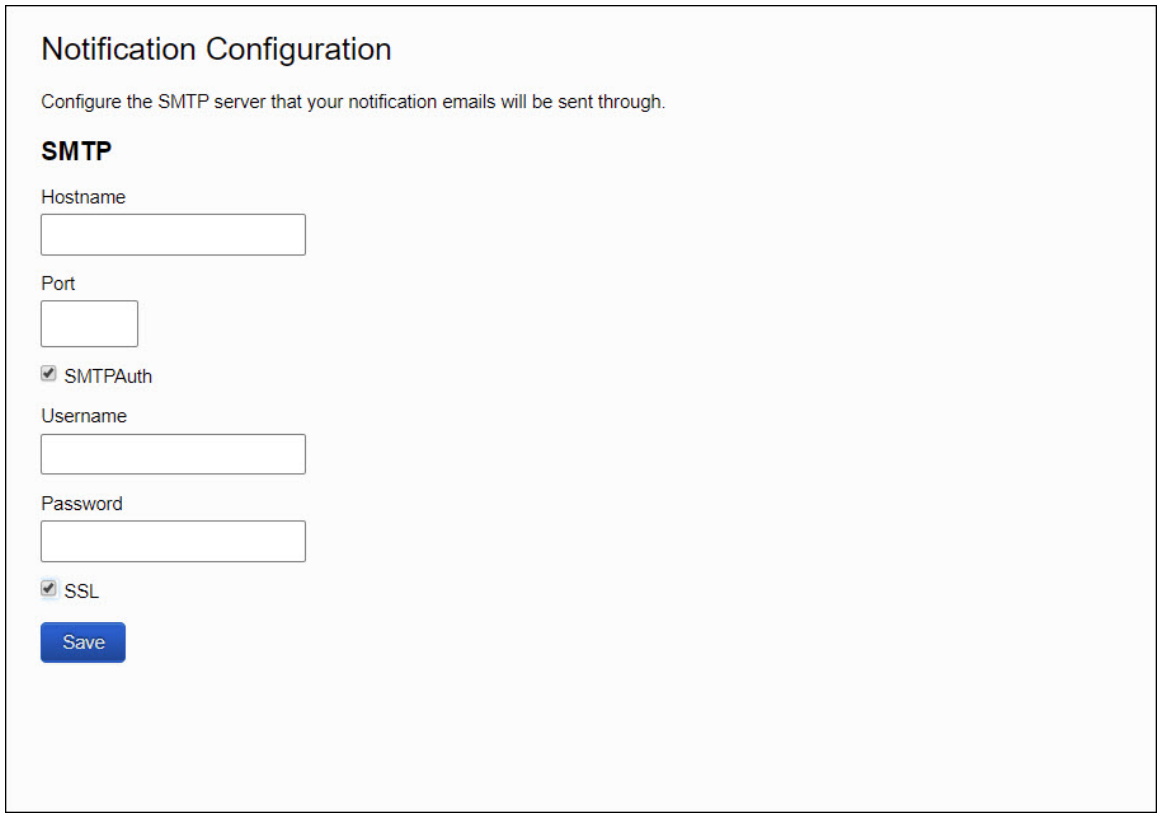
- Quota Notifications
- Configuring OAuth 2.0 API Access
- API and JMX Reporting

Configuring Quota Notifications

You can configure Mashery Local to send Over Throttle Limit, Over Quota Limit and Near Quota Limit Warning notifications when an API key exceeds or nears its limits.

To configure quota notifications, follow the steps below:

1. Click **Notification Configuration**.



2. In the **Notification Configuration** page, configure/enable the following fields for the SMTP server:
   - Hostname
   - Port
   - SMTPAuth
   - Username
   - Password
   - SSL
3. Click **Save**.

Similar notifications settings are available on the slave instances as well.

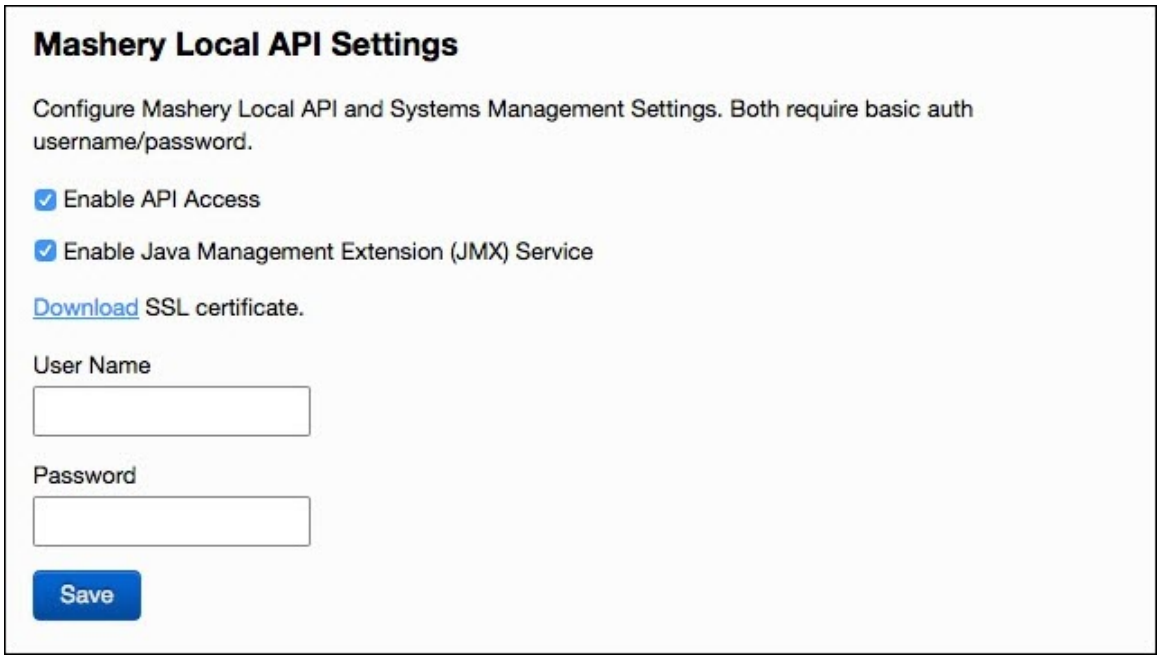
Configuring OAuth 2.0 API Access

Mashery Local provides a local OAuth 2.0 API. The API can be available on any computer within the cluster, regardless if the instance is a Master or Slave.

To configure and use the OAuth 2.0 API, follow the instructions below:

Procedure

1. Click **Account Settings**.



2. In the **Mashery Local API Settings** section, enable/configure the following fields to enable the API:
   - **Enable API Access**
   - **User Name** and **Password**: You need to enter a username and password to access the API using basic HTTP authentication.
3. Click **Save**.
4. Similar API settings are available on the slave instances as well. On all slave instances, navigate to **Account Settings**, and under **Mashery Local API Settings**, select the **Enable API Access** check box.

Making OAuth 2.0 Calls

The Mashery Local OAuth 2.0 API is available over SSL on port 8083 of your local instance.

Authorization is handled via HTTP authentication using the credentials you specified in the steps for Configuring OAuth 2.0 API Access.

Sample Call

The following is a sample call for OAuth 2.0.

```
curl -v -d '{
"method":"oauth2.fetchApplication",
"id": "2",
"params":{
```

```
        "service_key" : "<service_key>",
         "client" : {"client_id":"<api_key>",
                          "client_secret":"<api_secret>"},
                  "response_type" : null,
         "uri" : {"redirect_uri": "http://sometest.test.com/error?key=foo",
                      "state":"bar"}
        }
}' 'https://<masherylocal_host>:8083/v2/json-rpc/' -u '<api_access_user>:
<api_access_password>' -k
```

For host name, the IP Address of the instance where the OAuth API is enabled can be used. If the OAuth API on TIBCO Mashery® Local is to be used in conjunction with traffic outside the firewall, it is recommended that the TIBCO Mashery® Local cluster be fronted with a load balancer with a host name is associated with it (in addition to mapping port 8083 to an acceptable externally accessible port, such as 443).

Understanding the OAuth 2.0 API

The complete technical documentation for the Mashery OAuth 2.0 API is available online at http://support.mashery.com/docs/read/mashery_api/20/OAuth_Supporting_Methods. To see this documentation, request access by contacting your client services contact.

The following table describes the API at a high level:

API Method	Purpose
fetchApplication	Used during the Authorization step when the service provider's authorization server presents the resource owner with information about the client requesting access to the resource owner's data. The API calls is used to verify if the client is valid and fetches the client application data (name, attributes, redirection url) which will be used to provide information to the end user.
createAuthorizationCode (Authz Code grant type only)	After the resource owner has successfully authenticated against the service provider's authorization server and authorized the client, the authz server will make this API call to TIBCO Mashery to generate the authz code which can be subsequently used to obtain an access token. As a part of this API call, the service provider will also supply the user-context (userid) for the authenticated user. The service provider returns the authz code to the client using the redirection url.
createAccessToken	API call used to generate the access token. • For the authz code grant type, a valid authz code must be presented. • For implicit and resource owner grant types, this occurs after the resource owner has been authenticated (user-context should be supplied). Service provider initiates the API call. • For Client Credentials flow, only the client credentials are verified. • When exchanging a refresh token, a valid refresh token must be presented.  Both client id and secret must be presented when requesting an access token except in the case of Implicit grant type.

API Method	Purpose
fetchAccessToken	May be used by the service provider to validate access tokens and may be used as an additional layer of security or when certain API calls are sent directly to the provider instead of through TIBCO Mashery.
fetchUserApplications	Used by the service provider to present the resource owner with the client applications that been authorized by that resource owner. This is typically used in the **Account** section of the service provider's site where the resource owner can view the list.
revokeAccessToken	Used by the service provider to allow the resource owner to revoke access to specific client applications that been authorized by that resource owner. This is typically used in the "Account" section of the service provider's site where the resource owner can view the list authorized applications and select which application should no longer be allowed access.
revokeUserApplication	Revokes all tokens for an application for the specified user.

Configuring JMX Reporting Access

JMX Monitoring is not supported in Mashery Local for Docker 4.1.0.

Using the Adapter SDK

This section outlines the development process for writing custom adapters using the Adapter SDK for Mashery Local Traffic Manager. This section also provides the list of areas of extension provided in the SDK, along with code samples to illustrate the extension process.

Adapter SDK Package

The Adapter SDK defines the Traffic Manager domain model, tools and APIs and provides extension points to inject custom code in the processing of a call made to the Traffic Manager.

DIY SDK adapters need to be coded and compiled using JDK 1.6 or lower.

The Adapter SDK package contains the following:

- TIBCO Mashery Domain SDK
- TIBCO Mashery Infrastructure SDK

TIBCO Mashery Domain SDK

TIBCO Mashery Domain SDK packaged in com.mashery.trafficmanager.sdk identifies the traffic manager SDK and provides access to the TIBCO Mashery domain model which includes key objects such as Members, Applications, Developer Classes, Keys, Packages.

TIBCO Mashery Infrastructure SDK

TIBCO Mashery Infrastructure SDK provides the ability to handle infrastructure features and contains the following:

- **TIBCO Mashery HTTP Provider**

  The HTTP provider packaged as com.mashery.http provides HTTP Request/Response processing capability and tools to manipulate the HTTP Request, Response, their content and headers.

- **TIBCO Mashery Utility**

  The utility packaged as com.mashery.util provides utility code which handles frequently occurring logic such as string manipulations, caching, specialized collection handling, and logging.

SDK Domain Model

The Traffic Manager domain model defines the elements of the Traffic Manager runtime.

The following table highlights some of the key elements:

Element	Description	Usage
User	A user or member subscribing to APIs and accesses the APIs.	`com.mashery.trafficmanager.model.User`
API	An API represents the service definition. A service definition has endpoints defined for it.	`com.mashery.trafficmanager.model.API`

Element	Description	Usage

Endpoint	An Endpoint is a central resource of an API managed within Mashery. It is a collection of configuration options that defines the inbound and outbound URI's, rules, transformations, cache control, security, etc. of a unique pathway of your API.
	An Endpoint is specialized as either an API Endpoint or a Plan Endpoint. This specialization provides context to whether or not the Endpoint is being used as part of a Plan or not.
		• Generic endpoint entity representation:
		`com.mashery.trafficmanager.model.Endpoint`
		• API endpoint entity representation:
		`com.mashery.trafficmanager.model.APIEndpoint`
		• Plan endpoint entity representation:
		`com.mashery.trafficmanager.model.PlanEndpoint`

Method	A method is a function that can be called on an endpoint and represents the method currently being accessed/requested from the API request. A method could have rate and throttle limits specified on it to dictate the volume of calls made using a specific key to that method.
	A Method is specialized as either an API Method or Plan Method. The specialization provides context to whether or not the Method belong to a Plan.
		• Generic method entity representation:
		`com.mashery.trafficmanager.model.Method`
		• API method entity representation:
		`com.mashery.trafficmanager.model.APIMethod`
		• Plan method entity representation:
		`com.mashery.trafficmanager.model.PlanMethod`

Package	A Package is a mechanism to bundle or group API capability allowing the API Manager to then offer these capabilities to customers/users based on various access levels and price points. A Package represents a group of Plans.	`com.mashery.trafficmanager.model.Package`
Plan	A Plan is a collection of API endpoints, methods and response filters to group functionality so that API Product Managers can manage access control and provide access to appropriate Plans to different users.	`com.mashery.trafficmanager.model.Plan`

Element	Description	Usage
API Call	The API Call object is the complete transaction of the incoming request received by the Traffic Manager and the outgoing response as processed by the Traffic Manager. It provides an entry point into all other entities used in the execution of the request.	`com.mashery.trafficmanager.model.core.APICall`
Key	A key is an opaque string allowing a developer to access the API functionality. A key has rate and throttle controls defined on it and dictates the volume of calls that can be made to the API by the caller. A Key can be specialized as an API key or Package Key. This specialization provides context to whether the key provides access to an API or a specific Plan in a Package.	• Generic key entity representation: `com.mashery.trafficmanager.model.Key` • API key entity representation: `com.mashery.trafficmanager.model.APIKey` • Package key entity representation: `com.mashery.trafficmanager.model.PackageKey`
Application	An application is a developer artifact that is registered by the developer when he subscribes to an API or a Package.	`com.mashery.trafficmanager.model.Application`
Rate Constraint	A Rate Constraint specifies how the amount of traffic is managed by limiting the number of calls per a time period (hours, days, months) that may be received.	`com.mashery.trafficmanager.model.RateConstraint`
Throttle Constraint	A Throttle Constraint specifies how the velocity of traffic is managed by limiting the number of calls per second that may be received.	`com.mashery.trafficmanager.model.ThrottleConstraint`
Customer Site	A customer specific area configured through the developer portal.	`com.mashery.trafficmanager.model.CustomerSite`

Extended Attributes

The traffic manager model allows defining name-value pairs on different levels of the model. The levels are identified here:

- Application
- Customer Site
- Key (both API Key and Package Key)
- Package

- Plan
- User

Pre and Post Processor Extension Points

This version of the SDK allows extensions for Processors only. This means that only pre and post processing of requests prior to invocation of the target host are allowed.

Listener Pattern

The extension API leverages a listener pattern to deliver callbacks to extension points to allow injecting custom logic.

A call made to the traffic manager is an invocation to a series of tasks. Each step in the workflow accomplishes a specific task to fulfill the call. The current API release only allows customization of the tasks prior to invoking the API server (pre-process) and post receipt of the response from the API server (post-process). The callback API handling these extensions is called a Processor.

The pre-process step allows a processor to receive a fully-formed HTTP request targeted to the API server. The processor is allowed to alter the headers or the body of the request prior to the request being made to the server. Upon completion of the request and receiving the response the Traffic Manager allows the processor to alter the response content and headers prior to the response flowing back through a series of exit tasks out to the client.

Event Types and Event

The transition of the call from one task to the next is triggered through 'events' and an event is delivered to any subscriber interested in receiving the event. The SDK supports two event-types which are delivered synchronously:

- **Pre-Process Event type**: This event is used to trigger any pre-process task.
- **Post-Process Event type**: This event is used to trigger any post-process task.
- **Authentication Event type**: This event is used to trigger any custom authentication.

The subscribers in this case will be Processors registered in a specific manner with the Traffic Manager API.

Event Listener API

The Traffic Manager SDK provides the following interface and is implemented by custom processors to receive Processor Events.

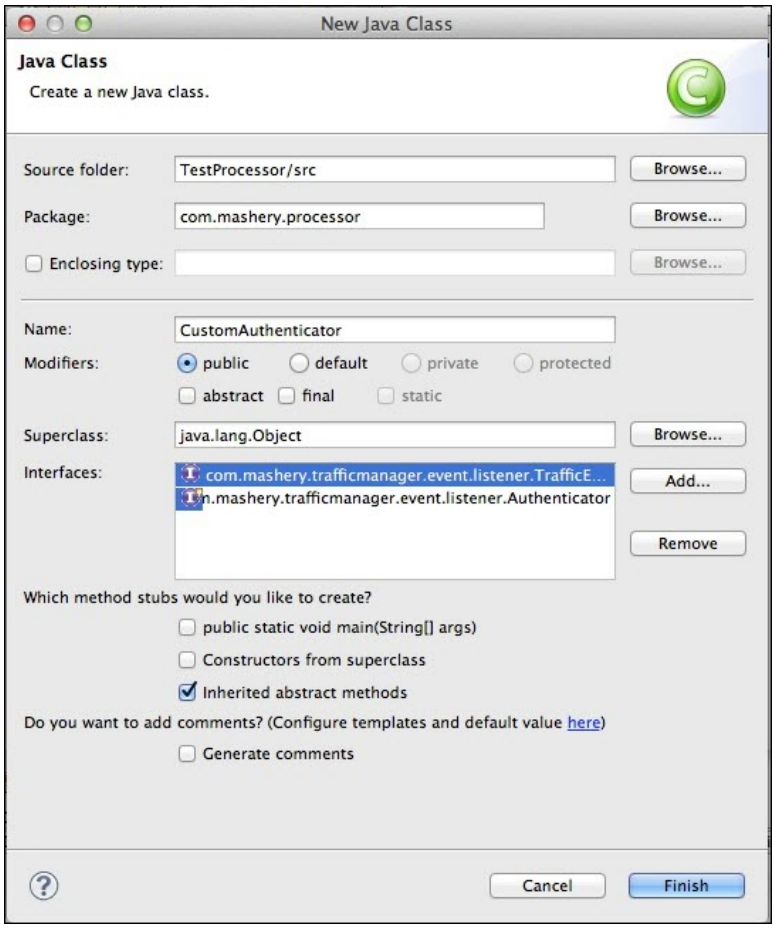
```
package com.mashery.trafficmanager.event.listener;
import com.mashery.trafficmanager.event.model.TrafficEvent;
/*** Event listener interface which is implemented by listeners which wish to
handle Traffic events. Traffic events will be delivered via this callback
synchronously to handlers implementing the interface.
The implementers of this interface subscribe to events via annotations. E.g.
Processor events need to handle events by using annotations in the
com.mashery.proxy.sdk.event.processor.annotation */
public interface TrafficEventListener {
   /*** The event is delivered to this API @param event*/
   void handleEvent(TrafficEvent event);
}
```

Creating a Custom Authenticator

This version of the SDK allows you to create a custom authenticator. This triggers an authentication event that a custom processor can handle for the custom authentication.

To create and use a custom authenticator:

1. Download the SDK, then add the SDK to the class path for **com.mashery.trafficmanager.event.listener.TrafficEventListener** and **com.mashery.trafficmanager.event.listener.Authenticator**.
2. Create a new class. In Eclipse, select the Java project, right-click and select `New > Class` in the context menu.
3. Set the following fields as shown in the image below, implement the **com.mashery.trafficmanager.event.listener.TrafficEventListener** and **com.mashery.trafficmanager.event.listener.Authenticator** interface, then click **Finish**.



4. The new class should contain the following content:

```
package com.mashery.processor;
import com.mashery.trafficmanager.event.listener.Authenticator;
import com.mashery.trafficmanager.event.listener.TrafficEventListener;
import com.mashery.trafficmanager.event.model.TrafficEvent;

public class CustomAuthenticator implements TrafficEventListener, Authenticator {
    @Override
    public void handleEvent(TrafficEvent arg0) {
        // TODO Auto-generated method stub
    }
}
```

5. Add the following class annotation, as shown:

```
package com.mashery.processor;
import com.mashery.trafficmanager.event.listener.Authenticator;
import com.mashery.trafficmanager.event.listener.TrafficEventListener;
import com.mashery.trafficmanager.event.model.TrafficEvent;
import com.mashery.trafficmanager.processor.ProcessorBean;
```

```
@ProcessorBean(enabled = true, name = "CustomAuthProcessor", immediate = true)
public class CustomAuthenticator implements TrafficEventListener, Authenticator {
    @Override
    public void handleEvent(TrafficEvent arg0) {
        // TODO Auto-generated method stub
    }
}
```

The "name" attribute on the ProcessorBean annotation, `CustomAuthProcessor`, is what will be used to reference this class from the endpoint definition.

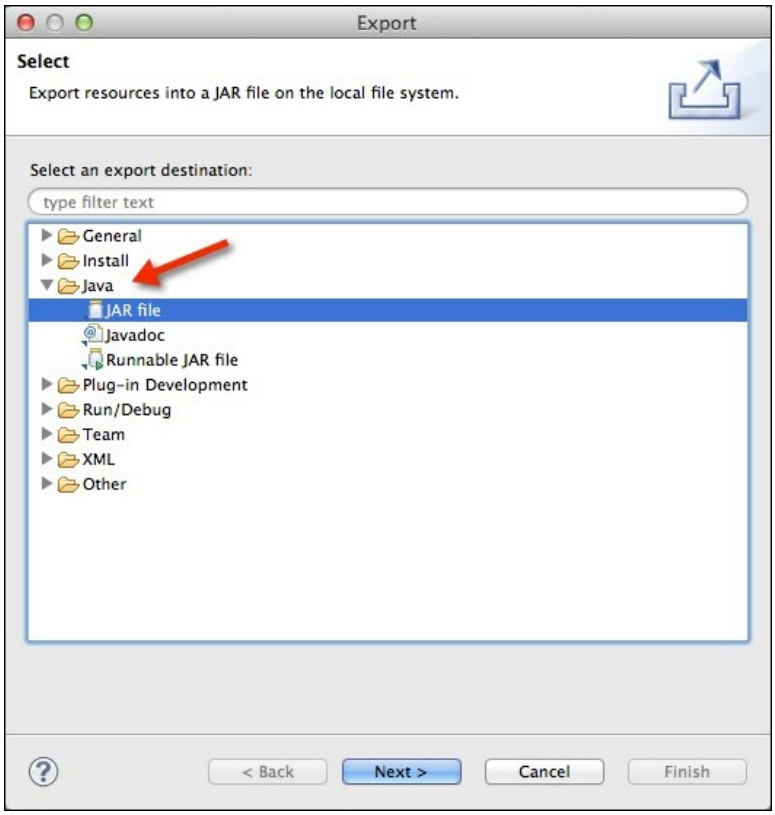
6. Update the **CustomAuthenticator** class with the following code snippet:

```
package com.mashery.processor;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.mashery.http.server.HTTPServerRequest;
import com.mashery.trafficmanager.event.listener.Authenticator;
import com.mashery.trafficmanager.event.listener.TrafficEventListener;
import com.mashery.trafficmanager.event.model.TrafficEvent;
import com.mashery.trafficmanager.event.processor.model.AuthenticationEvent;
import com.mashery.trafficmanager.event.processor.model.PostProcessEvent;
import com.mashery.trafficmanager.event.processor.model.PreProcessEvent;
import com.mashery.trafficmanager.processor.ProcessorBean;
import com.mashery.trafficmanager.processor.ProcessorException;

@ProcessorBean(enabled = true, name = "CustomAuthProcessor", immediate = true)
public class CustomAuthenticator implements TrafficEventListener, Authenticator {

    private final Logger log =
LoggerFactory.getLogger(CustomAuthenticator.class);
    @Override
    public void handleEvent(TrafficEvent event) {
        try {
            if (event instanceof PreProcessEvent) {
                //preProcess((PreProcessEvent) event);
            } else if(event instanceof PostProcessEvent){
                //postProcess((PostProcessEvent) event);
            } else if(event instanceof AuthenticationEvent){
                authCallprocess((AuthenticationEvent)event);
            }
        } catch (ProcessorException e) {
            log.error("Exception occurred when handling processor event");
        }
    }

    private void authCallprocess(AuthenticationEvent event) throws
ProcessorException{
        HTTPServerRequest httpRequest = event.getServerRequest();
        //add code to perform authentication of the request.
    }

}
```

Use **HttpServerRequest** to perform any authentication activity.

All the headers, status code and status messages set in the custom authentication would not be returned as part of response in case of Authentication Event handling (Custom authenticator). If you want to fail authentication request from custom authenticator, then you need to terminate the call in order to throw "ERR_403_NOT_AUTHORIZED" for a request. Refer to the example in Terminating a Call During Processing of an Event for more information.

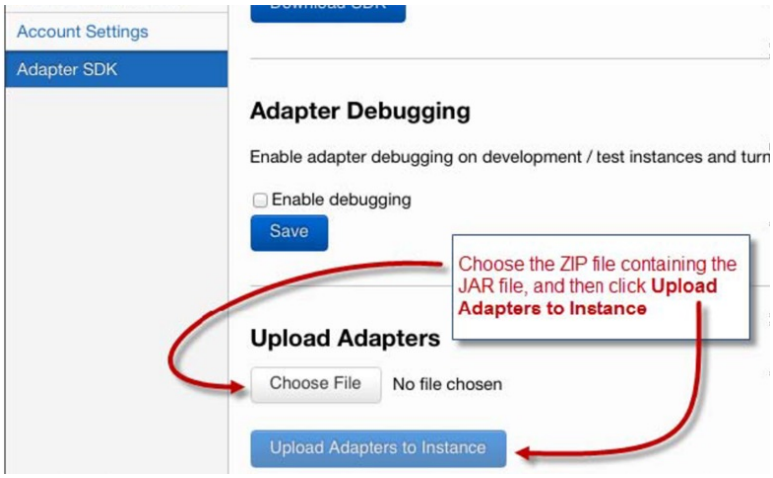
7. Ensure there are no compilation errors and export the JAR file. In Eclipse, select the Java project, right click and select **Export** from the context menu.
8. In the Export wizard, select and export destination of type `Java > Jar` file.



9. Provide the JAR file name and click **Finish**.

> You can ignore all warnings during the export process.

10. Create a zip archive, adding the exported JAR file created in the previous step.
11. Sign into the Mashery Local Admin UI and select **Adapter SDK** from the left menu.
12. In the **Upload Adapters** section, click **Choose File** and select the zip file created in Step 9, then click **Upload Adapters to Instance**.

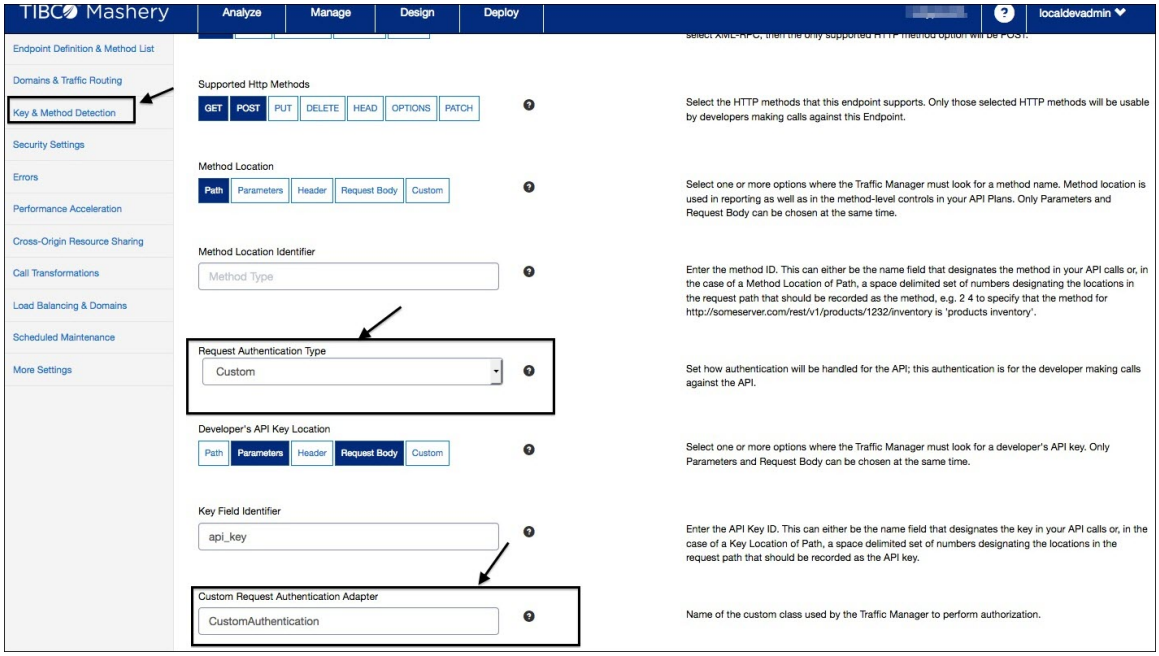


If the upload is successful, a message appears that the adapters were updated successfully.

13. Update the endpoint's authentication configuration to use the custom authenticator. Log into TIBCO Mashery SaaS (Control Center), navigate to `Design > API > Select Endpoint > Key & Method`

`Detection`, select **Request Authentication Type** as **Custom**, then enter the name of the custom authenticator in the **Custom Request Authentication Adapter** field.

The name used for the **Custom Request Authentication Adapter** field is the `ProcessorBean` name used in Step 4.



14. Save the configuration in the portal dashboard.

Implementing and Registering Processors

Writing custom processors involves the following general steps:

- Downloading the SDK
- Implementing the event listener
- Implementing lifecycle callback handling
- Adding libraries to the classpath

The following sections describe these steps in more detail.

Downloading the SDK

To download the SDK:

Procedure

1. Click **Adapter SDK**.
2. Click **Download SDK**:

3. Use your favorite IDE and put the SDK jars in your classpath.
4. Create a project and a new java class. The details of that process are skipped here and assumed that the developer will use the relevant IDE documentation to accomplish this.

Implementing the Event Listener

To implement the event listener:

Procedure

1. Employ the Traffic Event Listener interface (introduced in Event Listener API) as shown in the following example:

```
package com.company.extension;
public class CustomProcessor implements TrafficEventListener{
     public void handleEvent(TrafficEvent event){
        //write your custom code here
      }
}
```

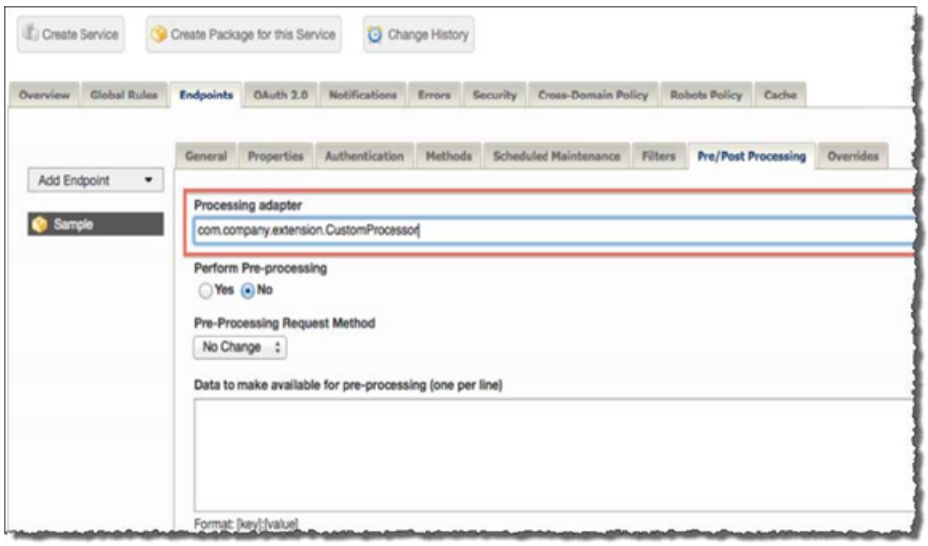
2. Annotate your code to ensure that the processor is identified correctly for callbacks on events related to the specific endpoints it is written to handle:

```
@ProcessorBean(enabled=true, name="com.company.extension.CustomProcessor",
immediate=true)
public class CustomProcessor implements TrafficEventListener{
     public void handleEvent(TrafficEvent event){
        //write your custom code here
    }
}
```

The annotation identifies the following properties:

- **enabled**: Identifies if the processor is to be enabled.
- **name**: Identifies the unique name of the processor as configured in API Settings (see marked area in 'red' in the following screenshot).
- **immediate**: Identifies if the processor is enabled immediately.

The name used in the annotation for the Processor MUST be the same as configured on the portal for the Endpoint>Pre/Post Processing, as shown in the following screenshot:



Implementing Lifecycle Callback Handling

If you wish to have some initialization work done once and only once for each of the processors, then implement the following interface:

`package com.mashery.trafficmanager.event.listener`:

```
/*** The lifecycle callback which gets called when the processor gets loaded when
installed and released*/
public interface ListenerLifeCycle {
  /*** The method is called once in the life-cycle of the processor before the
processor is deemed ready to handle requests. If the processor throws an exception,
the activation is assumed to be a failure and the processor will not receive any
requests @throws ListenerLifeCycleException*/
  public void onLoad(LifeCycleContext ctx) throws ListenerLifeCycleException;

 /*** The method is called once in the life-cycle of the processor before the
processor is removed due. The processor will not receive any requests upon
inactivation.*/
  public void onUnLoad(LifeCycleContext ctx);
}
```

The **onLoad** call is made once prior to the processor handling any requests and **onUnLoad** call is made before the processor is decommissioned and no more requests are routed to it.

The lifecycle listener can be implemented on the Processor class or on a separate class. The annotation needs to add a reference to the lifecycle-class if the interface is implemented (see highlighted property in **bold**).

```
package com.company.extension;
@ProcessorBean(enabled=true, name="com.company.extension.CustomProcessor",
immediate=true, lifeCycleClass="com.company.extension.CustomProcessor")
public class CustomProcessor implements TrafficEventListener, ListenerLifeCycle{
  public void handleEvent(TrafficEvent event){
    //write your custom code here
  }
  public void onLoad(LifeCycleContext ctx) throws ListenerLifeCycleException{
  }
  public void onUnLoad(LifeCycleContext ctx){
  }
}
```

The `lifeCycleClass` property should point to the class implementing the Listener LifeCycle interface. This also allows having a separate lifecycle listener interface as follows (note the different lifeCycleClass name).

The following example shows a different class implementing the LifeCycle callback:

```
package com.company.extension;
@ProcessorBean(enabled=true, name="com.company.extension.CustomProcessor",
immediate=true, lifeCycleClass="com.company.extension.CustomProcessorLifeCycle")
public class CustomProcessor implements TrafficEventListener {
   public void handleEvent(TrafficEvent event){
     //write your custom code here
   }
   public void onLoad(LifeCycleContext ctx) throws ListenerLifeCycleException{
   }
   public void onUnLoad(LifeCycleContext ctx){
   }
}
public class CustomProcessorLifeCycle implements ListenerLifeCycle{
   public void onLoad(LifeCycleContext ctx) throws ListenerLifeCycleException{
   }
   public void onUnLoad(LifeCycleContext ctx){
   }
}
```

Adding Libraries to Classpath

If the processor needs third-party libraries, those can be used in development and packaged with the processors, as described in Deploying Processors to Runtime.

Deploying Processors to Runtime

Deploying a custom processor involves the following general steps:

- Packaging the custom processor
- Uploading the custom processor
- Enabling Debugging

The following sections describe these steps in more detail.

Packaging the Custom Processor

To package your custom processor once the processor code is written, perform the following steps:

1. Compile the classes and create a JAR file with all the classes.
2. Specify third party libraries used in the `MANIFEST.MF` of the JAR containing the processor classes. and should be introduced as follows:
   - Third party libraries should be listed as values of the property **Class-Path** in the `MANIFEST.MF`.
   - Only third party library names are used, do not include paths to libraries.
   - Third party library names should be separated by spaces.

   Example: In `META-INF/MANIFEST.MF`:

   ```
   Class-Path: org.apache.commons.codec_1.4.0.v201209201156.jar
   com.amazonaws_1.1.9.jar
   ```
3. Create a folder "`lib`" alongside the JAR file, then copy all third party libraries to the "`lib`" folder.
4. Zip the JAR file and all contents of the "`lib`" folder to a ZIP package.

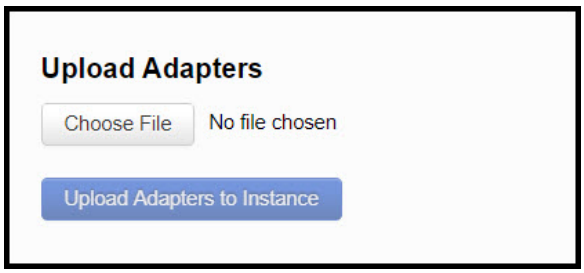
For more information, please refer to the Adapter SDK Usage and Examples section in the Appendix.

Uploading the Custom Processor

To upload the custom processor:

Procedure

1. In the **Upload Adapters** section, click **Choose file** to navigate and select the zip file containing the JAR file.
2. Click **Upload Adapters to Instance** to upload the package .ZIP file to the Mashery Local instance.



If the upload is successful, a message appears that the adapters were uploaded successfully.

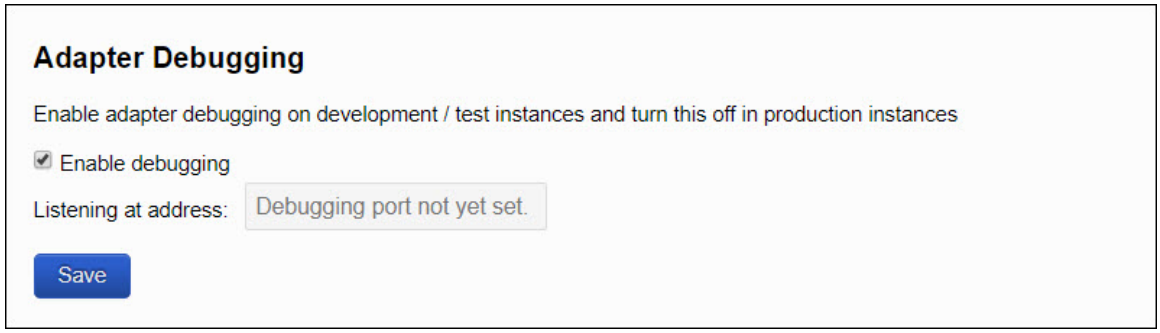
How Custom Processors are Updated

- Custom processors are updated each time a new package ZIP file is uploaded, the existing custom processors installed, with last package ZIP file are cleaned.
- To keep an existing custom processor, include it in the new package even if the custom processor is not updated.
- To update a custom processor, include it in the new package, do not change the attributes of *ProcessorBean*.

Enabling Debugging

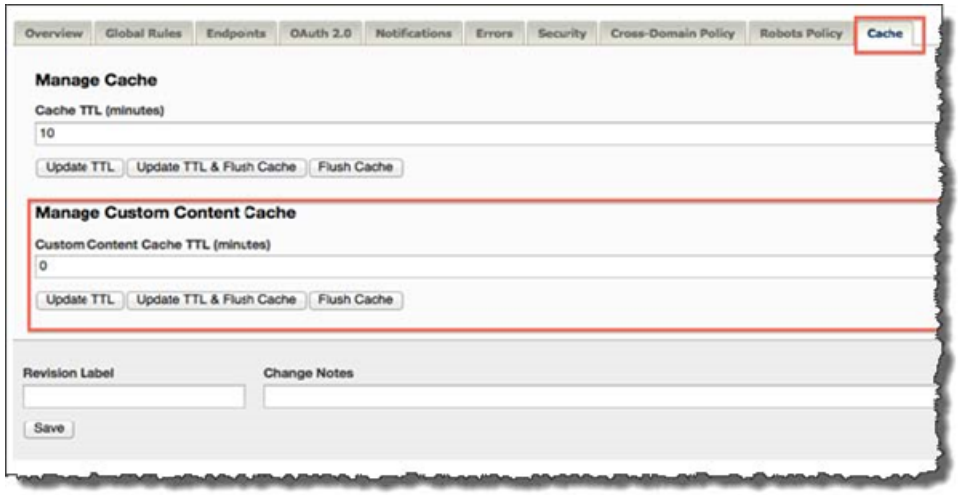
During development, it is sometimes necessary to enable debugging on the Mashery Local instance.

To enable debugging, click the **Enable debugging** check box, indicate the port number to which you will connect your debugger, and then click **Save**:



Caching Content

The custom endpoints can cache content during the call handling. The cache configuration is found in the **Manage Custom Content Cache** section on the **API Settings** page.



Manage Custom Content Cache provides the following options:

- **Custom TTL**: A default TTL provided for the cache.
- **Update TTL**: Provides ability to save any TTL changes.
- **Update TTL & Flush Cache**: Updates the database with the updated TTL and flushes the cache contents.
- **Flush Cache**: Allows the cache contents to be flushed.

The SDK provides references to a Cache where all this data is stored. The cache interface provided in the callback to the `TrafficEventListener` is:

```
package com.mashery.trafficmanager.cache;
/*** Cache API which allows extensions to store and retrieve data from cache*/
public interface Cache {
  /**
  * Retrieves the value from the cache for the given key
  * @param key
  * @return
  * @throws CacheException
  */
  Object get(String key) throws CacheException;
  /**
  * Puts the value against the key in the cache for a given ttl
  * @param key
  * @param value
  * @param ttl
  * @throws CacheException
  */
  void put(String key, Object value, int ttl) throws CacheException;
}
```

A reference to the cache can be found on the `ProcessorEvent` which is reported on the callback. Here is an example of how to access cache on callback:

```
package com.company.extension;
@ProcessorBean(enabled=true, name="com.company.extension.CustomProcessor",
immediate=true
public class CustomProcessor implements TrafficEventListener, ListenerLifeCycle{
  public void handleEvent(TrafficEvent event){
       ProcessorEvent processorEvent = (ProcessorEvent) event;
       Cache cacheReference = processorEvent.getCache();
       //Add data to cache
       try{
```

```
            cacheReference.put(“testkey”, “testValue”, 10)
        }catch(CacheException e){
         //load data or load default data
        }
    //write your custom processor code here
    }
}
```

A reference to cache is also available on the lifecycle callback:

```
package com.company.extension;
public class CustomProcessorLifeCycle implements ListenerLifeCycle{
   public void onLoad(LifeCycleContext ctx) throws ListenerLifeCycleException{
      Cache cache = ctx.getCache();
      // perform cache operations
   }
   public void onUnLoad(LifeCycleContext ctx){
   }
}
```

Terminating a Call During Processing of an Event

This version of the SDK allows a user to terminate a call during pre or post processing, or in authentication event handling. For example, if the request does not have a required URL parameter, Mashery Local can be configured to terminate the call in the pre-processing.

All the headers, status code and status messages set in the custom processing is returned to the client as part of the response in pre processing and post processing.

All the headers, status code and status messages set in the custom authentication would not be returned as part of response in case of Authentication Event handling (Custom authenticator). If you want to fail authentication request from the custom authenticator, then you need to terminate the call in order to throw "ERR_403_NOT_AUTHORIZED" for a request.

For example, if you want to terminate the call in authenticator, if request doesn't contain the authorization header, then the call can be terminated by marking the response as complete as shown in the following example:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.mashery.http.HTTPHeaders;
import com.mashery.trafficmanager.debug.DebugContext;
import com.mashery.trafficmanager.event.listener.Authenticator;
import com.mashery.trafficmanager.event.listener.TrafficEventListener;
import com.mashery.trafficmanager.event.model.TrafficEvent;
import com.mashery.trafficmanager.event.processor.model.AuthenticationEvent;
import com.mashery.trafficmanager.event.processor.model.PostProcessEvent;
import com.mashery.trafficmanager.event.processor.model.PreProcessEvent;
import com.mashery.trafficmanager.processor.ProcessorBean;
import com.mashery.trafficmanager.processor.ProcessorException;

@ProcessorBean(enabled = true, name = "CustomAuthentication", immediate = true)

public class CustomAuthentication implements TrafficEventListener,Authenticator {

    @Override
    public void handleEvent(TrafficEvent event) {
        try {
            if (event instanceof AuthenticationEvent) {
                authenticate((AuthenticationEvent) event);
            }
        } catch (ProcessorException e) {
        }
    }

    private void authenticate(AuthenticationEvent event)
            throws ProcessorException {
        //For example request doesn't contain the authorization header then user
can terminate the call by marking response as complete
```

```
        // in order to thrown 403 ERR_403_NOT_AUTHORIZED for the incoming request.
        if (headers != null) {
            String authorization = headers.get(HEADER_AUTHORIZATION);
            if ((null == authorization || authorization == "")
                    || !authorization.startsWith(AUTH_BASIC)) {
                debugContext.logEntry("Final Value", "DIY-CUSTOM-AUTH-HEADER-
FAILIURE");
                event.getCallContext().getResponse().setComplete();
            }
    }
}
```

If you want to terminate the call in pre or post processing, refer to the following example:

```
package com.mashery.processor;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.mashery.trafficmanager.event.listener.TrafficEventListener;
import com.mashery.trafficmanager.event.listener.Authenticator;
import com.mashery.trafficmanager.event.model.TrafficEvent;
import com.mashery.trafficmanager.event.processor.model.PostProcessEvent;
import com.mashery.trafficmanager.event.processor.model.PreProcessEvent;
import com.mashery.trafficmanager.model.core.ExtendedAttributes;
import com.mashery.trafficmanager.processor.ProcessorBean;
import com.mashery.trafficmanager.processor.ProcessorException;

@ProcessorBean(enabled = true, name = "PrePostProcessing", immediate = true)

public class PrePostProcessing implements TrafficEventListener{

    private final Logger log = LoggerFactory.getLogger(PrePostProcessing.class);

    @Override
    public void handleEvent(TrafficEvent event) {
        try {
            if (event instanceof PreProcessEvent) {
                preProcess((PreProcessEvent) event);
            } else if (event instanceof PostProcessEvent) {
                postProcess((PostProcessEvent) event);
            }
        } catch (ProcessorException e) {
            log.error("Exception occurred when handling processor event");
        }
    }

   //In the below example we checking the query parameter's value to decide whether
to terminate the call or not.
    private void preProcess(PreProcessEvent event) throws ProcessorException {
        String complete =
event.getCallContext().getRequest().getQueryData().get("preComplete");
        if (complete != null) {
            event.getCallContext().getResponse().getHTTPResponse().setBody(new
StringContentProducer("{\"response\": \"Terminated the call in pre-processing\"}"));
            event.getCallContext().getResponse().setComplete();
        }
    }
   //In the below example we checking the query parameter's value to decide whether
to terminate the call or not.
    private void postProcess(PostProcessEvent event) throws ProcessorException {
       String complete =
event.getCallContext().getRequest().getQueryData().get("postComplete");
        if (complete != null) {
            event.getCallContext().getResponse().getHTTPResponse().setBody(new
StringContentProducer("{\"response\": \"Terminated the call in post-processing
\"}"));
            event.getCallContext().getResponse().setComplete();
        }
    }
}
```

Accessing Package Key EAVs in the Custom Processor

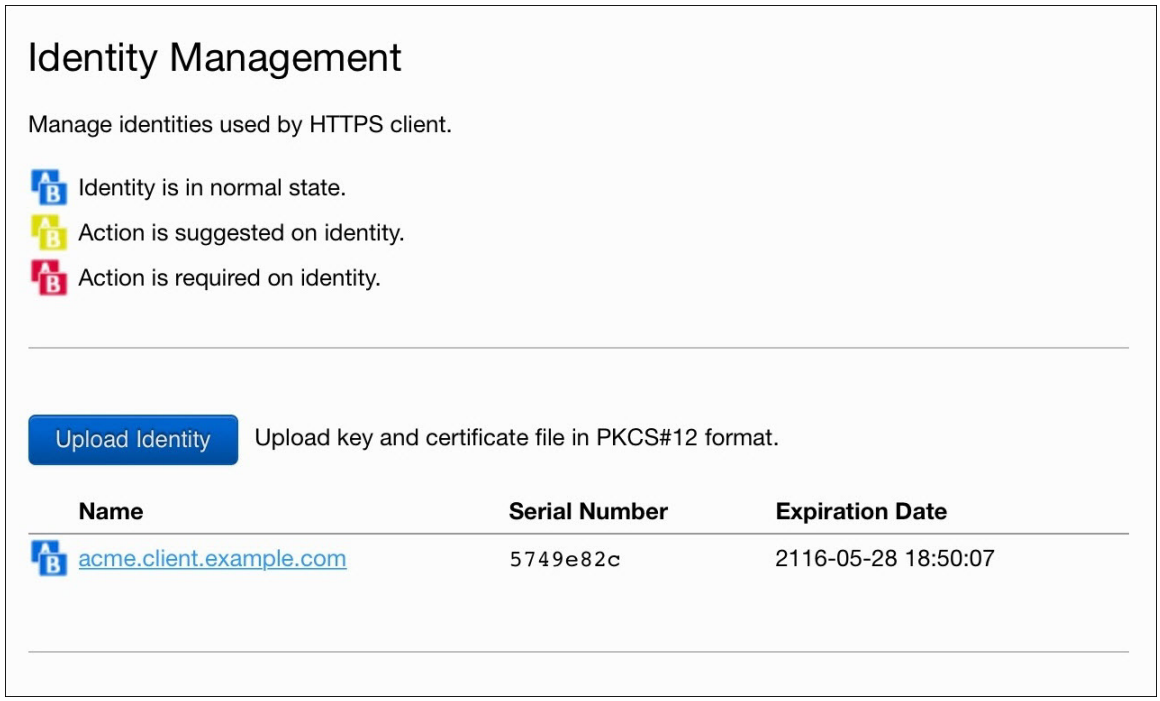
This version of the SDK includes an example custom processor that can access package EAVs (Extended Attribute Values).

For example, this sample processor looks for "user_defined_key" EAVs for a package key:

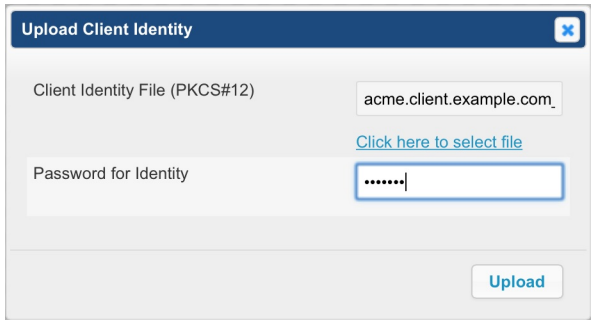
```
package com.mashery.processor;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.mashery.trafficmanager.event.listener.TrafficEventListener;
import com.mashery.trafficmanager.event.listener.Authenticator;
import com.mashery.trafficmanager.event.model.TrafficEvent;
import com.mashery.trafficmanager.event.processor.model.PostProcessEvent;
import com.mashery.trafficmanager.event.processor.model.PreProcessEvent;
import com.mashery.trafficmanager.model.core.ExtendedAttributes;
import com.mashery.trafficmanager.processor.ProcessorBean;
import com.mashery.trafficmanager.processor.ProcessorException;

@ProcessorBean(enabled = true, name = "PrePostProcessing", immediate = true)

public class PrePostProcessing implements TrafficEventListener,Authenticator {

    private final Logger log = LoggerFactory.getLogger(PrePostProcessing.class);

    @Override
    public void handleEvent(TrafficEvent event) {
        try {
            if (event instanceof PreProcessEvent) {
                preProcess((PreProcessEvent) event);
            } else if (event instanceof PostProcessEvent) {
                postProcess((PostProcessEvent) event);
            } else if (event instanceof AuthenticationEvent) {
               authenticate((AuthenticationEvent) event)
            }
        } catch (ProcessorException e) {
            log.error("Exception occurred when handling processor event");
        }
    }

    //In the below snippet we are extracting the package key and checking its value.

    private void preProcess(PreProcessEvent event) throws ProcessorException {
        ExtendedAttributes attrs = (event).getKey().getExtendedAttributes();
        String strAllowed = attrs.getValue("user_defined_key");
    }
    private void postProcess(PostProcessEvent event) throws ProcessorException {
        ExtendedAttributes attrs = (event).getKey().getExtendedAttributes();
        String strAllowed = attrs.getValue("user_defined_key");
    }
    private void authenticate(AuthenticationEvent event)
            throws ProcessorException {
        ExtendedAttributes attrs = (event).getKey().getExtendedAttributes();
        String strAllowed = attrs.getValue("user_defined_key");
    }
}
```

Configuring Identity Management

The **Identity Management** page allows the administrator to add or update identities used by the HTTPS client. The HTTPS client profile references these identities.



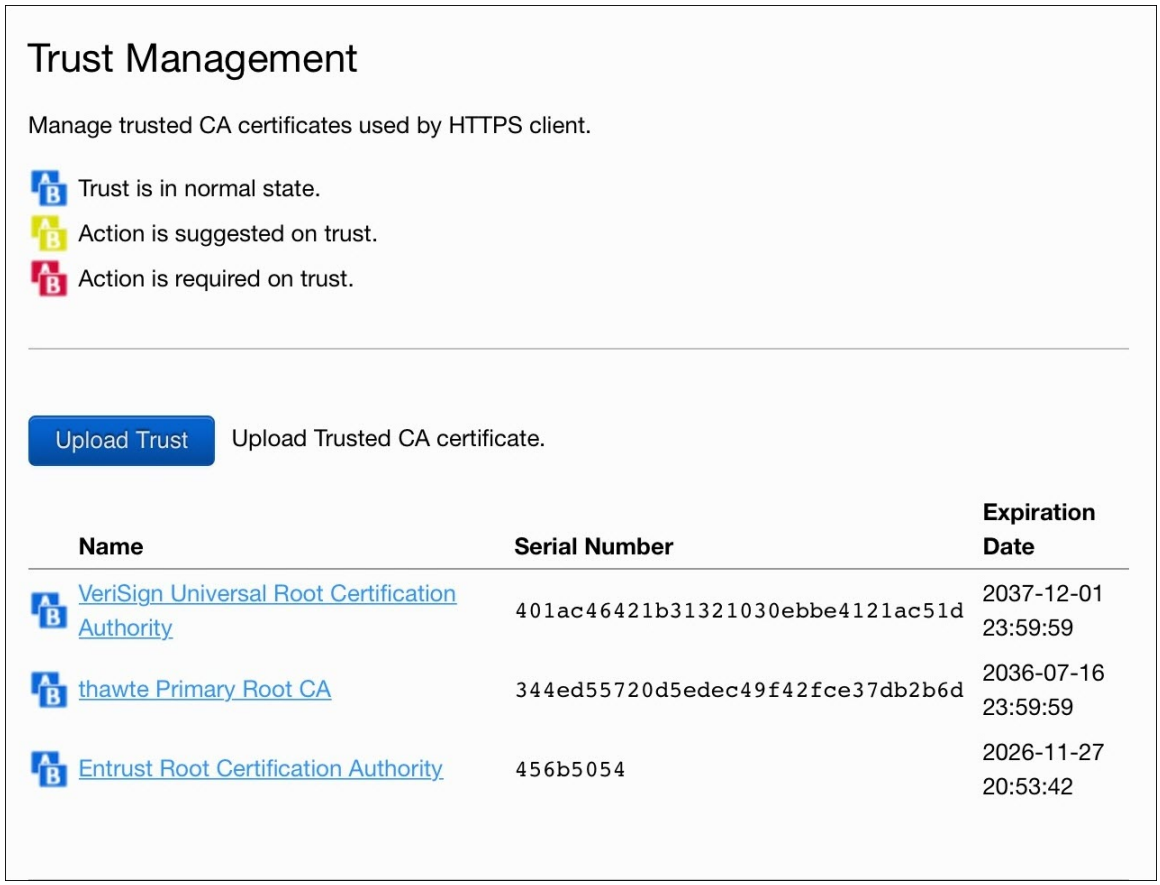
The following table describes the fields in the **Identity Management** page.

Field or button	Description
Upload Identity	Opens the **Upload a Trusted CA Certificate** window.  To upload an identity, click **Click here to select file**, enter the **Password**, then click **Upload**.
Name	The name of the identity.
Serial Number	The serial number of the identity.

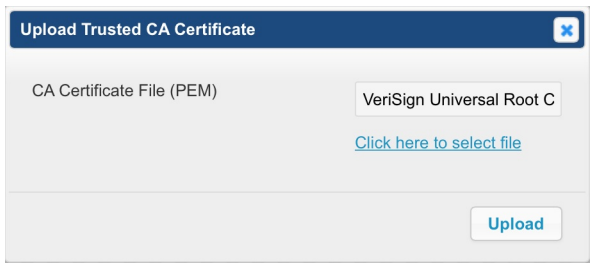
Field or button	Description
Expiration Date	The date and time the identity expires.
State	Identifies the following information: • The state of the identity: – Certificate manifest will be synchronized with TIBCO Mashery SaaS – Certificate manifest has been synchronized with TIBCO Mashery SaaS – Certificate is about to expire - The expiration warning is shown one month before expiration date. – Certificate expired. – Certificate manifest update will be synchronized with TIBCO Mashery SaaS – Certifcate in TIBCO Mashery Local is outdated for TIBCO Mashery SaaS – Certificate is not present in TIBCO Mashery Local • The number of profile(s) using the identity. • The number of endpoint(s) using the identity. • The available action suggested or required for the identity.

Configuring Trust Management

The **Trust Management** page allows the administrator to add or update certificates used by the HTTPS client. The HTTPS client profile references these certificates.



The following table describes the fields in the **Trust Management** page.

Field or button	Description
Upload Trust	Opens the **Upload a Trusted CA Certificate** window.  To upload a certificate, click **Click here to select file**, then click **Upload**. In case of a CA certificate chain, you can upload the immediate CA certificate.

Field or button	Description
Name	The name of the certificate.
Serial Number	The serial number of the certificate.
Expiration Date	The date and time the certificate expires.
State	Identifies the following information: • The state of the certificate: – Certificate manifest will be synchronized with TIBCO Mashery SaaS – Certificate manifest has been synchronized with TIBCO Mashery SaaS – Certificate is about to expire - The expiration warning is shown one month before expiration date. – Certificate expired. – Certificate manifest update will be synchronized with TIBCO Mashery SaaS – Certifcate in TIBCO Mashery Local is outdated for TIBCO Mashery SaaS – Certificate is not present in TIBCO Mashery Local • The number of profile(s) using the certificate. • The number of endpoint(s) using the certificate. • The available action suggested or required for the certificate.

Testing the New Instance

You should test a new instance after installing and creating it.

Testing a New Instance

One approach to test a new instance is:

Procedure

1. Find the API to test in the **API Settings** area of the Mashery Admin Dashboard and identify an associated endpoint that is ready for testing.
2. Create a test API key for the API identified in the previous step. You accomplish this in the **Users** area accessed by clicking the **Users** tab of the Mashery Admin Dashboard.
3. Perform a manual sync of the Services and Developers in the **Cloud Sync** page of the Mashery Local Cluster Manager, as described in step 7 of Configuring Slaves to the Master.
4. Construct a test API call for the API you wish to test.
5. Execute the API call against the instance. Unless you have set up a domain name for the instance, your API call will need to be made against the IP address of the instance directly.

   Should you use a hostname or IP in your test call? When a service is setup in the dashboard, the hostnames (IP addresses as well could be used) that will consume the service are defined. When a call is made to the proxy, the hostname used for the call must match one of the hostnames setup in the dashboard for the service, otherwise the call will fail. If you make a call directly to one of the instances using its IP address and that IP address was not configured in the service definition, then the proxy returns a 596 error.

   If you receive the expected response from the API, then your instance is working properly.

Tracking the Database Restore and Replication Status

TIBCO Mashery Local slave node registration process provides a status endpoint that helps to track the asynchronous steps of database restore and replication status. However, these steps are processed in the background, so there is no active feedback on completion or failure of processes.

The status endpoint (registration_status.py) is an experimental API and is subject to change in later implementations.

There is no need to Enable API Access on the node for this endpoint to function.

To obtain the database restore and replication status, type the URL of the endpoint in your browser. For example: `https://<Mashery_Local_Slave_IP>:5480/service/mashery/cgi/replication_status.py`

The endpoint returns the following JSON:

```
{
  "replication_status":
  {
    "restore": {
                "error": false,
                "errors": "",
                "log": "1. transfer backup from master\nMon Aug 22 18:38:04
UTC
                        2016\n\n2. unzip backup\nMon Aug 22 18:38:41 UTC 2016\n\n3.
                        stop slave\nSlave stopped\nMon Aug 22 18:38:41 UTC
                        2016\n\n4. restore from backup\nMon Aug 22 20:36:18 UTC
                        2016\n\n5. start slave\nSlave started\nMon Aug 22 20:36:18
                        UTC 2016\n\n6. done\nMon Aug 22 20:36:18 UTC 2016\n\n",
                "complete": true
```

```
                },
  "replication": {
                  "last_error": "",
                  "seconds_behind_master": "250832\n",
                  "slave_io_running": "Yes\n",
                  "slave_sql_running": "Yes\n"
                  }
  },
  "error": null
}
```

The following table provides details about the JSON that is returned by the status endpoint:

JSON Node	Value in JSON	Description
Database restore log		
`replication_status.restore.log`		Provides database restore log.
`replication_status.restore.complete`	true	Implies that the database restore step is done.
	false	Implies that the database restore step is not complete.
`replication_status.restore.error`	true	Implies that there are errors during the process. Refer to the `replication_status.restore.errors` node to get more details for the errors.
	false	Implies that there are errors during the process.
Database replication status		
`replication_status.replication`		Provides replication-specific information from “show slave status”.
`replication_status.replication.last_error`	<error>	Provides the details for the errors (if any).
`replication_status.replication.seconds_behind_master`	<time in milliseconds>	Provides an estimate of how long it takes the slave node to catch up to the master.

JSON Node	Value in JSON	Description
`replication_status.replication.slave_io_running` or `replication_status.replication.slave_sql_running`	No Yes	Replication is not running and `last_error` provides the details for the errors (if any). `Replication starts, seconds_behind_master` provides an estimate of how long it takes the slave node to catch up to the master.

When the restore step is in process, replication is disabled. Therefore, the value of `slave_io_running` and `slave_sql_running` is `No`.

Troubleshooting

Mashery Local provides a set of tools that help an administrator to debug issues with API calls as they flow through TIBCO Mashery, troubleshoot networking issues with the system, identify issues with cloud synchronization, and collect system logs to facilitate Operations and Support staff to identify root-cause faster. This section outlines the tools available and their usage scenarios.

Verbose Logs

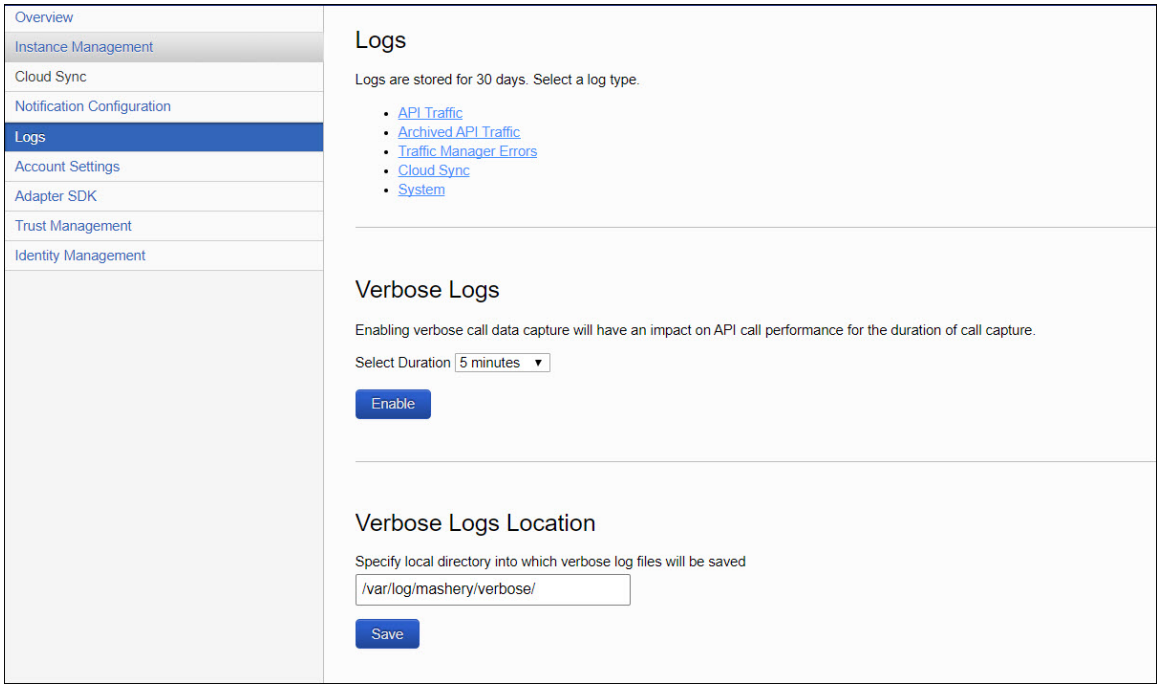
The Mashery Local administrator can troubleshoot issues related to call payloads or identify any inconsistencies as API call data flows through TIBCO Mashery by enabling verbose logs on API calls. This feature is not enabled as an "always on" feature as producing these verbose logs may have some impact on API call performance. Instead, options are provided on the **Cluster Manager** UI to enable and disable verbose logs.

Using the Verbose Logs Feature

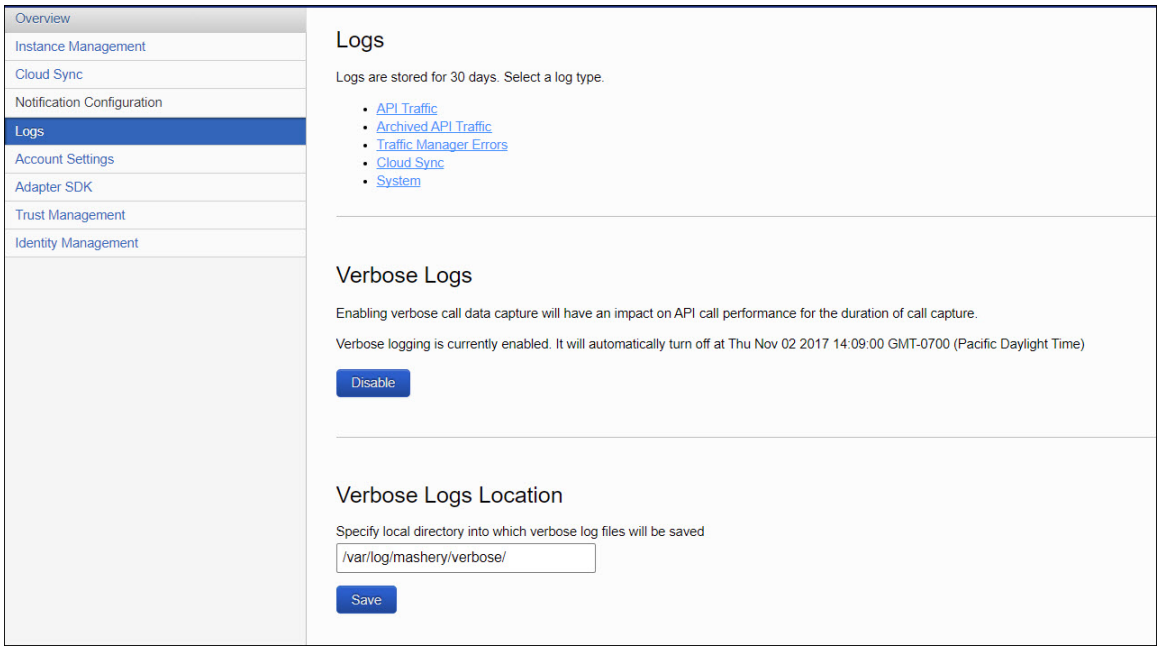
To use the Verbose Log feature:

Procedure

1. Click **Logs**
2. Specify the Verbose Logs location. The default directory is `</var/log/mashery/verbose>`.



3. Enable Verbose Logs.
   a) Select duration for capturing the logs (05, 10, 15, or 30 minutes).
   b) Click **Enable**.



After you enable verbose logs, Mashery Local writes the call data logs that include inbound request data, inbound processed data, outbound response data, and outbound processed data. Verbose logs (call data capturing) is disabled after the selected time duration expires.

You must set the Verbose Logs Location on each node in the cluster including Master and all Slaves. Enabling or disabling verbose logs can only occur on the Master node. The Slave nodes just inherit the current verbose log enablement status from the Master.

Working with Verbose Logs

A directory is created every minute with the name format as YYYY-MM-DD-HH-MM. All the calls that are logged in a minute become part of one directory and so on. For each call, a sub-directory is created using the name <timestamp>-<Mashery Message ID>.

Mashery Message ID is a globally unique ID generated for every API call that is processed by TIBCO Mashery. The Mashery Message ID provides a possible mechanism for administrators to create a golden thread for debugging issues between TIBCO Mashery, your partners and your backend system. To be able to include this GUID in request and response headers, you can toggle on **Include X-Mashery-Message-ID in Request** and **Include X-Mashery-Message-ID in Response** properties on in the **Services>Endpoint>Properties** page in the TIBCO Mashery Administration Dashboard.

Within each sub-directory, four log files are `InboundRequest.log`, `InboundProcessed.log`, `OutboundResponse.log`, `OutboundProcessed.log`.

- `InboundRequest` contains the request data on the API call as it is originally received by Mashery Local from the client application.
- `InboundProcessed` contains the Mashery processed version of the inbound request as sent to API server (or to cache if enabled).
- `OutboundResponse` contains the response data as it is originally received by Mashery from the API server (or from cache if enabled).
- `OutboundProcessed` contains the Mashery processed version of the outbound response as sent to the client application.

Each of the four files contain some important metadata written as key-value pairs with 1 pair on one line. After the metadata, a new delimiter line is written followed by the actual message.

The metadata included are:

- Key: Key a developer uses to get access to a Package Plan or API.
- Service ID: TIBCO Mashery generated unique ID to identify an API.
- Endpoint ID: TIBCO Mashery generated unique ID to identify an Endpoint.
- Site ID: TIBCO Mashery generated unique ID to identify your Site within the TIBCO Mashery Network.
- IP address: IP address of the client application invoking the API call.
- Method (if available): Method that was being accessed in the API call (available if appropriate **Method Configuration** settings are specified in **Services**>**End-points**>**Properties** tab in the TIBCO Mashery Administration dashboard).
- Cache hit: 1 if cache is enabled and response is met from cache, 0 otherwise.
- Error message (if any): TIBCO Mashery generated error message on that API call (if any).

Mapping Endpoint IDs

TIBCO Mashery Local provides a script that allows fetching a list of endpoints with details such as the Endpoint ID and the Endpoint name. The Endpoints associated with a service are displayed. The Service ID is the parameter used to fetch the Endpoint details.

```
//Request searching with a particular service id
python getEndpointNames.py --service 95bpf2jv3f8p5x3xqsu657x5

//Response in json formatter
{
   "services":[
      {
         "endpoints":[
            {
               "id":"7xwgjatahmuwgrz79cgw286a",
               "name":"CORS-disabled"
            },
            {
               "id":"2m4zz8nw4n9w36uau7j2bnqb",
               "name":"Custom CORS(custom rest as the API key source)"
            },
            {
               "id":"g2qx6vhxubu4d4w66egqnxsh",
               "name":"CORS-enabled- EIN-112-dontallow"
            },
            {
               "id":"uavv2nm6yy7j94nhp8zp5kjf",
               "name":"CORS-enabled-EIN-112"
            },
            {
               "id":"pgbrzzu89dtyvumqht4ncnt4",
               "name":"preflight-requestmultipledomainno"
            },
            {
               "id":"7qcpe6dsss4kxp4u8c6fy5pr",
               "name":"EIN-222"
            }
         ],
         "id":"95bpf2jv3f8p5x3xqsu657x5"
      }
   ]
}
```

Debugging Utility

A debug utility is provided that can be used to capture information about system health and configuration, connectivity to the cloud for synchronization, fix Slave registration issues, and resolve any replication issues between Master and Slave. A command line console is available to run the various options available. This utility is useful for gathering information to assist with trouble-shooting common system configuration errors with TIBCO Mashery Local.

Running the Debug Utility

Execute the following command to run the debug utility:

```
$ python /opt/mashery/utilities/debug_util.py
```

The following options are available. Some options are available to be run only on Master and some only on Slave.

```
Select from the following:
1: Collect Logs
2: Test connectivity to Cloud Sync
3: Show Slave Status
4: Check IP address
5: Update record or Master IP address in Master (Master IP address has changed and
registration of new Slave with cluster fails)
6: Fix slave corruption (Restart slave at last valid read position)
7: Update record of Master IP address in old Slave node (Master IP address has
changed and cluster is not updated)
8: System manager (Remove non-functional or unused slaves form Master)
9: Collect system state (Disk health, process health, time setting, network
settings)
menu: Show this menu
exit: Quit
```

For option **9: Collect system state**, the resulting files for this option are created in the home directory, depending on the login users (root/administrator).

Collect Logs

This tool produces a `tar.gz` file that collects Traffic Manager component logs, sync status with the cloud, the Slave and Master IP address checks, logs required to check replication issues between Master and Slave and verbose logs for the day (if any).

This option can be run on Master and Slave nodes.

Test Connectivity to Cloud Sync

This tool helps to determine if there are any errors connecting to the TIBCO Mashery Cloud system for synchronization.

This option can be run on Master and Slaves.

Show Slave Status

This option displays whether a Slave is functioning correctly, including its status, the Master systems IP address and any replication errors that are present between Master and Slave.

This option can be run on a Slave node.

Check IP Address

This option allows you to check the current IP address of the Master.

This option can be run on a Master node.

Update Record of Master IP Address in Master

Sometimes if the IP address of a Master node changes, new Slave registration with the Master fails. Running this option fixes the record of the Master IP address in the Master node for successful Slave registration.

This option can be run on a Master node.

Fix Slave Corruption

This option allows you to resolve Master Slave replication issues.

This option can be run on a Slave node.

Update Record of Master IP Address in Old Slave Node

This option updates the record of the Master IP address in the Slave nodes and is useful for resolving Master-Slave replication issues.

System Manager (Remove Non-functional or Unused Slaves from Master)

Sometimes Slave nodes are decommissioned and new Slave nodes are created. This option on Master system can be used to remove unused slaves.

System Level Troubleshooting

TIBCO Mashery Local administrators have the ability to run the following select commands to investigate and troubleshoot the network or system level issues.

- ping
- ping6
- tracepath
- tracepath6
- tcpdump
- traceroute
- arping
- tshark
- route
- ifconfig
- iptables
- dhclient

sudo edit for the following files:

- /etc/resolv.conf
- /etc/sysconfig/network-scripts/ifcfg-eth0
- /etc/sysconfig/network-scripts/ifcfg-eth1
- /etc/rc.local
- /etc/hosts
- /etc/securitylimits.conf
- /etc/nssswitch.conf

General Troubleshooting

The following table provides information for troubleshooting general Mashery Local (for Appliance and Docker) issues.

Form Factor	Issue	Notes
All	API call returns a 596 error	**Possible Cause** API is configured with specific supported HTTP Methods, and the HTTP Method used for this call is not allowed. **Diagnostic Steps** 1. Test the API call using the SaaS domain (`<customer>`.api.mashery.com) 2. If the call returns a 596 error, review the **Key & Method Detection** settings for this endpoint and confirm that the HTTP Method used in the API call is allowed. **Resolution** 1. If the HTTP method used in this call is not configured on the endpoint, update the supported HTTP Methods to include the HTTP method. 2. Run a manual Mashery Local sync to update the configuration in the on-prem traffic manager.

Form Factor	Issue	Notes

Appliance	API call returns a 596 error	**Possible Cause**
		Memcached is not running
		Diagnostic Steps
		Check that the API configuration is loaded into memcache:
		1. SSH into the Mashery Local Instance, for example: `ssh root@<IP ADDRESS OF THE INSTANCE>`
		2. TELNET to the Memcache port: **telnet localhost <port>**. Here are the port numbers for the various memcache pools:
		    1. "memservicePool": 11214
		    2. "memcachePool": 11211
		    3. "memcachePackaged": 11215
		    4. "contentCachePool": 11213
		    5. "memcountPool": 11212
		3. Run the **stats items** command.
		4. Identify the item number with more than 1 record.
		5. Run the command:
		`stats cachedump <ITEM NUMBER> <NUMBER OF RECORDS>`
		Resolution
		If the response is coming from the master and the settings are not in memcache, you likely have a synchronization issue.
		If the response is coming from the slave and the settings are not in memcache, you like have a replication issue. Force a memcache load of the service definitions:
		`/opt/javaproxy/proxy/memcacheloader --env production --verbose --service`

Form Factor	Issue	Notes

Docker	API call returns a 596 Error	**Possible Cause**
		Memcached container is not running.
		Diagnostic Steps
		1. Check the proxy.log for Memcached errors.
		2. Check whether the memcached is running:
		`docker exec -it ml-mem ps -ef`
		and look for the memcached process.
		Resolution
		1. If memcache is not running, ssh into the ml-mem container to start it and see whether there's any error:
		`docker exec -it ml-mem /bin/bash`
		then:
		`service memcached start`
		2. If it's caused by running out of file limit, increase the **ulimit** setting by editing the docker-compose.yml file, and if you are using docker-compose, do a docker-compose down followed by a docker-compose up to restart the containers.
		Please see https://docs.docker.com/compose/compose-file/#/ulimits.
		Add those to the docker-compose.yml file under the container section, most likelyml_memwould need this(if memcached failed to start). For example:
		`ulimits:`
		`  nproc: 65535`
		`  nofile:`
		`    soft: 20000`
		`    hard: 40000`
		  Watch out for the leading spaces. They must align with others using the correct indentation.

Form Factor	Issue	Notes
All	API call returns intermittent 596 error on a previously working slave.	**Possible Cause** Sync between master and one or more slaves is not working. **Diagnostic Steps** Errors are intermittent indicating that there is a problem with one slave. Use the following command: `python /opt/mashery/utilities/debug_util.py` Select Option 3 (Show Slave Status). This option displays whether a Slave is functioning correctly, including its status, the Master systems IP address and any replication errors that are present between Master and Slave. **Resolution** If errors are present, recreate the Slave instance.

All	API call returns 596 error on a new slave.	**Possible Cause**
		Sync between master and slave is not working.
		Diagnostic Steps
		When connecting a new slave to a Master, the customer sees this error:
		`Registering as Slave ERROR: Failed to configure the node as slave.`
		Resolution
		This can happen if the IP Address of the Master was changed after the initial installation of the Master. The built in Debug Utility (debug_util.py) should be run on the Master in order to fix this.
		Have the customer run the debug_util.py on the "Master", using the following command:
		`python /opt/mashery/utilities/debug_util.py`
		Select Option 5. (Update record or Master IP address in Master. (Master IP address has changed and registration of new Slave with cluster fails)).
		The customer should then be able to register the new Slave to the Master node.

Form Factor	Issue	Notes

All	Mashery Local Web Console is blank.	**Possible Cause**
		Disk is full.
		Diagnostic Steps
		Review disk space using the "df -h" command. This will give you a percentage usage of both disks (there are usually 2 disks, 1 "system" and the other "mnt" (mnt contains the logs and the mysql database, the rest is on system)
		`Filesystem Size Used Avail Use% Mounted on`
		`/dev/sda3 8.8G 1.4G 7.0G 17% /`
		`tmpfs 871M 0 871M 0% /dev/shm`
		`/dev/sda1 124M 63M 55M 54% /boot`
		`/dev/mapper/mnt_vg-mnt 40G 514M 37G 2% /mnt`
		Resolution
		If disk space utilization is over 90% for either disk, customer should ask their System Administrator to increase the size of the respective disk.

All	Mashery Local Web Console is blank.	**Possible Cause** Available memory is low. **Diagnostic Steps** Review free memory using the "free -h" command. **Resolution** If available memory is low or the system is using swap, customer should ask their System Administrator to increase the size of memory on this instance or add more nodes to the cluster so that this instance is not at capacity.

Form Factor	Issue	Notes
Appliance	Mashery Local Web Console is blank.	**Possible Cause** Basic processes are not running. **Diagnostic Steps** Review basic processes using the "ps aux | more" command. Check for: • memcached • javaproxy • mysqld • vami-sfcbd • lighttpd **Resolution** If any of these processes are not running, reboot Mashery Local instance.

All	Cannot synchronize API Settings.	**Possible Cause**
		Connection to Mashery On-Prem Manager (MOM) is not present.
		Diagnostic Steps
		Run the following command:
		`dig api-mom.mashery.com`
		If you get a response, then try:
		`curl -k https://api-mom.mashery.com/ping`
		Resolution
		If you get a response, then you do have a good connection to MOM.
		If you do not get a response, check your network configuration to ensure outbound HTTPS / 443 access is allowed.

Form Factor	Issue	Notes
All	Mashery Local returns a 503 Service Unavailable error.	**Possible Cause** Failsafe is being triggered for the endpoint in question. **Diagnostic Steps** Confirm that the error message of `503_Service_Unavailable_Proxy` is being returned. **Resolution** This means Mashery's failsafe has been triggered due to excessive 504 responses from the API over a short period of time. It could be that the customer's origin servers are now taking longer than the configured connection or response TTLs set on the endpoint. If those values are low, then the customer should increase the values. If they are already high, then the customer needs to improve performance on their origin server to alleviate the issue.

Docker	Docker Instance cannot be reached.	**Possible Cause**
		Docker containers need to be returned to a clean state.
		Diagnostic Steps
		Error checking TLS connection: Something went wrong running an SSH command!
		error getting ip address: host is not running
		Docker-Machine instances in Timeout state
		Resolution
		If you are connected to the VPN, disconnect VPN
		• Stop All containers
		`docker stop $(docker ps -a -q)`
		• Delete all containers
		`docker rm $(docker ps -a -q)`
		• Delete all images
		`docker rmi $(docker images -q)`
		• If using Virtualbox, remove host adapter -
		Open Virtualbox, click `File -> Preferences -> Network -> Host-only Network`, remove Vboxnet#
		• Unsetting DOCKER variables
		`unset ${!DOCKER*}`
		Restart Docker Terminal and start creating new instance.

Appendix

This appendix describes some configurations for using Mashery Local features.

Setup Examples

The following sections describe setup examples for Mashery Local deployments.

Example Cloud Deployments with CLI

The following examples demonstrate Mashery Local for Docker deployment to cloud environments (Azure and AWS for now) using Command Line Interface (CLI). The advantage with CLI is that the deployment process can be done repeatedly with scripts and an environment can be easily replicated reliably.

These examples are for illustrating the concepts, they are not production-grade deployments. The scripts used in these examples are in the **examples** directory.

In each example, two Mashery Local instances are created - one for Mashery Local master and one for Mashery Local slave. Each Mashery Local instance is deployed to a cloud virtual machine (VM) instance (each VM instance corresponding to a docker host). Then, both the master and slave are connected to Mashery On-Prem Manager and are ready to handle traffic.

Cloud Deployment Prerequisites

- Docker, docker-machine, and docker-compose installed locally (for example, on a Mac).
- AWS or Azure accounts have been set up and CLI installed.
- Verify the command `aws` or `az` is on your path and you can do some simple CLI commands, for example:
  - `az group list`
  - `aws ec2 describe-vpcs`

Deploying Mashery Local to Cloud Environment

1. Follow the instructions to build Mashery Local for Docker images locally with docker-machine. Verify the images are correct (no errors during the build and the images can be seen with the command `docker images`.
2. Prepare a directory that has the three image gz files, the modified (optional) docker-compose.yml file, and azure-ml.sh, aws-ml.sh and init-ml.sh.

   export MOM_KEY="<MOM key>" export MOM_SECRET="<MOM secret>"

   chmod a+x *.sh
3. Set the following environmental variables for Azure or AWS:

   For Azure:

   export AZURE_USER="<azure user>"

   export AZURE_PWD="<azure password>"

   export AZURE_SUBSCRIPTION_ID="<your azure subscription id>"

   export AZURE_IMAGE="<azure image for the location>"

   (Default value "canonical:UbuntuServer:16.04.0-LTS:latest" for location "westus" if not specified.)

   export ML_LOC_NAME="<azure location for ML>"

(Default value "westus" if not specified)

export ML_RESOURCE_GROUP="<azure resource group name for ML>"

(Default value "mlResourceGroup" if not specified)

export ML_DNS_NAME="<public IP DNS name>"

(Default value "testml" if not specified)

For AWS:

export AWS_ACCESS_KEY_ID="<AWS_ACCESS_KEY_ID>"

export AWS_SECRET_ACCESS_KEY="<AWS_SECRET_ACCESS_KEY>"

export AWS_DEFAULT_REGION="<default AWS region>"

(Default value "us-east-1" if not specified)

export AWS_ZONE="<AWS zone>"

(Default value "e" if not specified)

export AWS_AVAILABILITY_ZONE="<AWS availability zone>"

(Default value "us-east-1e" if not specified)

export AWS_AMI="<AWS AMI id>"

(Default value "ami-efe09bf8" if not specified)

Where the AWS AMI has to be an Ubuntu 16.04.* image available on the region

For eu-central-1, do NOT use ami-9346bcfc because there's some extra software running inside that would cause performance degradation. Any public AWI needs be carefully examined before being used.)

And the AWS availability zone must exist for that region.

Verify that the output format is json in your default configuration (check the file <home>/.aws/config or use the command "aws configure to verify).

4. Run the following scripts:

   For Azure, run the script azure-ml.sh.

   For AWS, run the script aws-ml.sh.

   It takes some time for the scripts to run (about 45 minutes for Azure and 100 minutes for AWS) because images must be loaded to the cloud twice - once each for the master and slave.

   It's better to save the output to a file for later examination, for example:

   ```
   ./aws-ml.sh 2>&1 |tee /tmp/aws-ml.out
   ```

   If the script hangs in docker loading images, try the following:

   1. Ctrl-C out of the script.
   2. Execute the following commands (they take more than an hour):

      ```
      execute the following commands (they take more than an hour)

      docker-machine scp ml_db.tar.gz aws-ml-master:/tmp
      docker-machine ssh aws-ml-master docker load -i /tmp/ml_db.tar.gz
      docker-machine scp ml-mem.tar.gz aws-ml-master:/tmp
      docker-machine ssh aws-ml-master docker load -i /tmp/ml-mem.tar.gz
      docker-machine scp ml_core.tar.gz aws-ml-master:/tmp
      docker-machine ssh aws-ml-master docker load -i /tmp/ml_core.tar.gz

      docker-machine scp ml_db.tar.gz aws-ml-slave1:/tmp
      docker-machine ssh aws-ml-slave1 docker load -i /tmp/ml_db.tar.gz
      ```

```
docker-machine scp ml-mem.tar.gz aws-ml-slave1:/tmp
docker-machine ssh aws-ml-slave1 docker load -i /tmp/ml-mem.tar.gz
docker-machine scp ml_core.tar.gz aws-ml-slave1:/tmp
docker-machine ssh aws-ml-slave1 docker load -i /tmp/ml_core.tar.gz

./init-ml.sh aws  2>&1 |tee /tmp/init-ml.out
```

If something goes wrong with AWS and you need to rerun the aws-ml.sh, perform the following cleanup first:

```
docker-machine rm aws-ml-master aws-ml-slave1
docker-machine rm -f aws-ml-master aws-ml-slave1

aws ec2 delete-key-pair --key-name aws-ml-master
aws ec2 delete-key-pair --key-name aws-ml-slave1

aws elb delete-load-balancer --load-balancer-name MLLB
aws ec2 delete-vpc --vpc-id <vpc id value>
```

If you just cleaned up the previous install and the AWS instances were "Terminated" but still not completely removed yet, you may get the following error when you rerun aws-ml.sh: `"An error occurred (InvalidInstance) when calling the RegisterInstancesWithLoadBalancer operation: EC2 instance i-xxxx... is not in running state."`

As a workaround if you don't want to wait till those AWS instances are completely removed, you can modify the aws-ml.sh and init-ml.sh to replace to words ML-Master and ML-Slave1 to something else, before executing the aws-ml.sh again.

Handling Errors on docker-machine create

Update your docker-machine version to 0.12.2+ if you encounter either of the following errors when aws-ml.sh or azure-ml.sh is doing "docker-machine create":

- `Error creating machine: Error running provisioning: Unable to verify the Docker daemon is listening: Maximum number of retries (10) exceeded`
- `Error creating machine: Error running provisioning: ssh command error:command : systemctl -f start dockererr : exit status 1output : Job for docker.service failed because the control process exited with error code. See "systemctl status docker.service" and "journalctl -xe" for details.`

See https://github.com/docker/machine/releases for the latest release and installation instructions.

Before you retry aws-ml.sh or azure-ml.sh, make sure to remove the old broken docker-machines first:

```
docker-machine rm -f <docker machine name>
```

where the docker-machine name could be aws-ml-master, aws-ml-slave1, azure-ml-master, or azure-ml-slave1.

Also, clean up the entities (AWS VPC or AZURE ResourceGroup ...) created on AWS or AZURE.

You may need to update your aws cli or azure cli to the latest version.

For reference, see:

- http://docs.aws.amazon.com/cli/latest/userguide/installing.html
- https://docs.microsoft.com/en-us/cli/azure/install-azure-cli

The following are the versions that work on Mac:

- ```
  aws --version

  returns

  aws-cli/1.11.124 Python/3.5.1 Darwin/13.0.2 botocore/1.5.87
  ```
- ```
  az --version

  returns
  ```

```
azure-cli (2.0.10)

...

Python (Darwin) 3.5.1 (default, Jan 22 2016, 17:08:33)

[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)]

Python location '/usr/local/opt/python3/bin/python3.5'
```

Recreating Master and Slave

If docker-machines (AWS or Azure instances) are fine and Docker images are loaded, and you just want to have a clean new master and slave, perform the following steps.

- For AWS: `./init-ml.sh aws 2>&1 |tee /tmp/init-ml.out`
- For Azure: `./init-ml.sh azure 2>&1 |tee /tmp/init-ml.out`

A load balancer is created for both Azure and AWS cases, and traffic would go through the load balancer.

For Azure, to access the UI, you can use the docker host IP or load balancer IP with port 5480 for master and port 5481 for slave1. For AWS, to access the UI, use the instance's public IP. You can find the public IP from AWS UI or use the command for Master:

```
aws ec2 describe-instances --filters 'Name=tag:Name,Values=ML-Master'|grep
PublicIpAddres |cut -d '"' -f 4
```

For Slave, (Slave1, for example), use the command:

```
aws ec2 describe-instances --filters 'Name=tag:Name,Values=ML-Slave1'|grep
PublicIpAddres |cut -d '"' -f 4
```

There are other enhancements that could be made, for example, modify the security rules to restrict internal traffics from within the virtual network only, or create docker-machine without public IP address (all required access from outside could go through load balancer).

Example Setup to Run Mashery Local Master and Slave on a Local Machine

Prerequisites

Docker, docker-machine, and docker-compose installed locally (for example, on a Mac).

Procedure

1. Follow the instructions to build Mashery Local for Docker images locally with docker-machine.
   Prepare a directory that has the three image gz files, the modified (optional) docker-compose.yml file and init-ml.sh.

   The docker-compose.yml and init-ml.sh can be found in the examples directory.

   ```
   export MOM_HOST="<MOM host>"
   export MOM_KEY="<MOM key>"
   export MOM_SECRET="<MOM secret>"
   chmod a+x *.sh
   ```

2. Create two docker machines: `xxx-ml-master` and `xxx-ml_slave1`, where `xxx` is replaced with your own machine names, then start both docker-machines.

The docker machine names must be `xxx-ml-master` and `xxx-ml_slave1` for the init-ml.sh to work without modification.

Both docker-machines must be running (Use "`docker-machine ls`" to check their status.)

3. Execute the command:

   `init-ml.sh xxx load`

   (for the first time); or:

   `init-ml.sh xxx`

   This cleanly restarts the master and slave re-using the already loaded images (with all previous data removed).

Adapter SDK Usage and Examples

The following sections describe a typical setup of a development environment for the Adapter SDK, as well as examples of Adapter SDK usage in various projects.

Adapter SDK Development Environment Example Setup

The following is an example setup used for the Adapter SDK development environment. The environment details are listed below for reference.

- **Operating System**: CentOS Linux release 7.3.1611
- **JDK**: OpenJDK version "1.8.0_121"
- **Eclipse**: Eclipse Java EE IDE for Web Developers
  - **Version**: Mars.2 Release (4.5.2)
  - **Build id**: 20160218-0600
- **Ant**: Apache Ant™ version 1.9.2 compiled on June 10 2014

  `yum install ant`
- **Maven**: Apache Maven 3.0.5 (Red Hat 3.0.5-17)

  `yum install maven`

Setting up the Adapter SDK for Maven

To set up the Adapter SDK for Maven, follow the steps below:

Procedure

1. Download the Adapter SDK from Mashery Local Cluster Manager.
2. Copy the Adapter SDK to your development environment, for example, `/home/beta/Documents/sdk.zip`.
3. Unzip the Adapter SDK:

   ```
   unzip sdk.zip -d sdk
   ```
4. Install Adapter SDK JARs in Maven local repository:

   ```
   cd /home/beta/Documents/sdk
   mvn install:install-file -Dfile=com.mashery.http_1.0.0.v20130130-0044.jar -
   DgroupId=com.mashery \
   -DartifactId=http -Dversion=1.0.0.v20130130-0044 -Dpackaging=jar -
   DgeneratePom=true
   mvn install:install-file -
   Dfile=com.mashery.trafficmanager.sdk_1.1.0.v20130214-0043.jar -
   DgroupId=com.mashery \
   ```

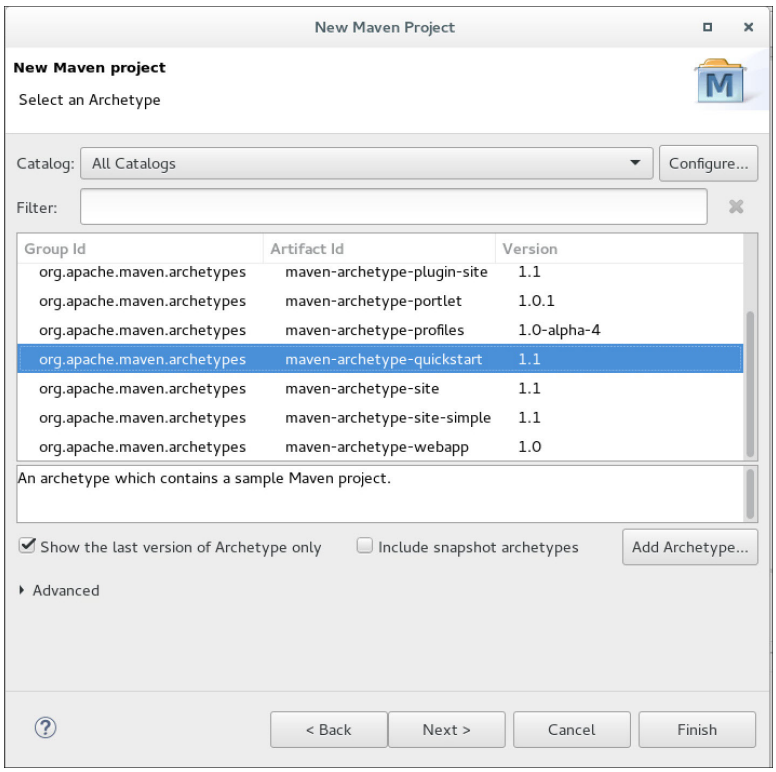
```
-DartifactId=trafficmanager -Dversion=1.1.0.v20130214-0043 -Dpackaging=jar -
DgeneratePom=true
mvn install:install-file -Dfile=com.mashery.util_1.0.0.v20130214-0015.jar -
DgroupId=com.mashery \
-DartifactId=util -Dversion=1.0.0.v20130214-0015 -Dpackaging=jar -
DgeneratePom=true
```

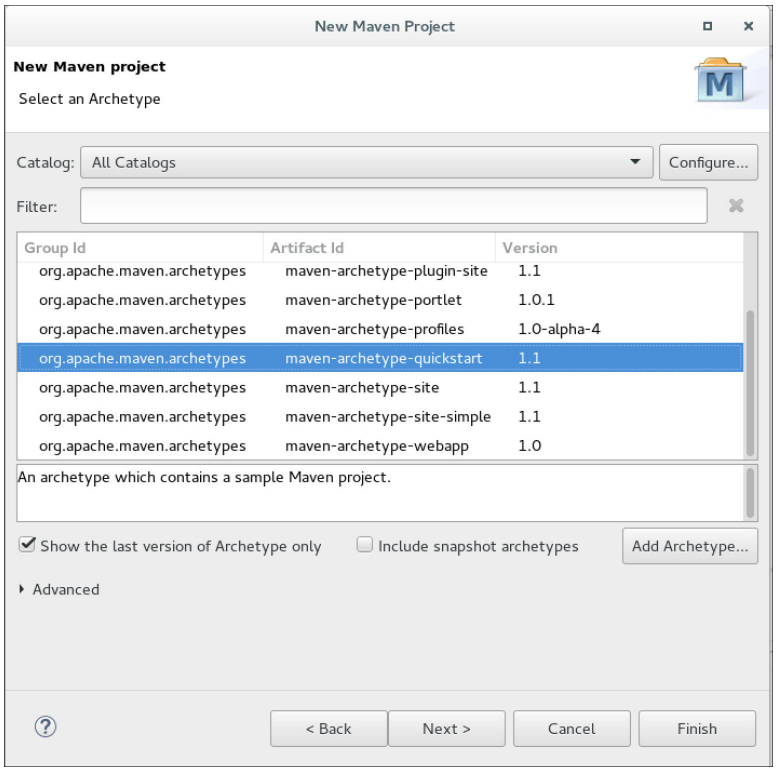
Using the Adapter SDK in Mashery Local with Single Processor

To use the Adapter SDK in Mashery Local with a single processor, follow the steps below:

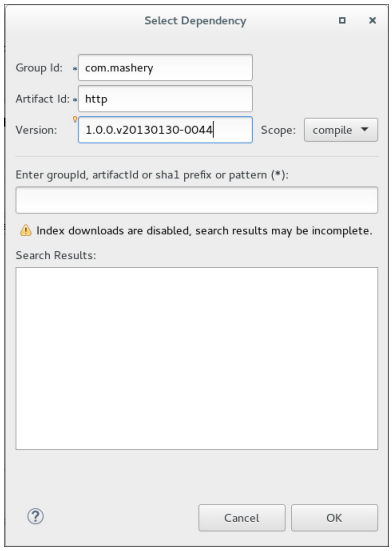
Procedure

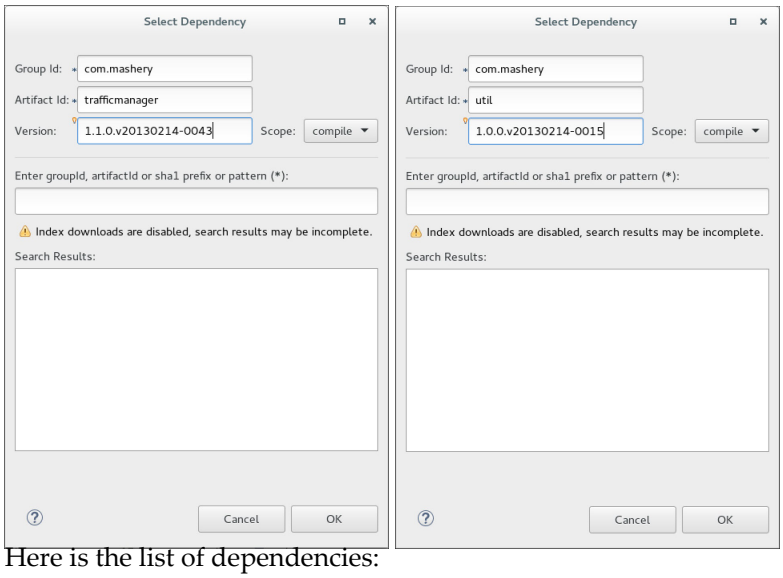
1. Create a new Maven Project in Eclipse. Go to `File > New > Maven Project`.



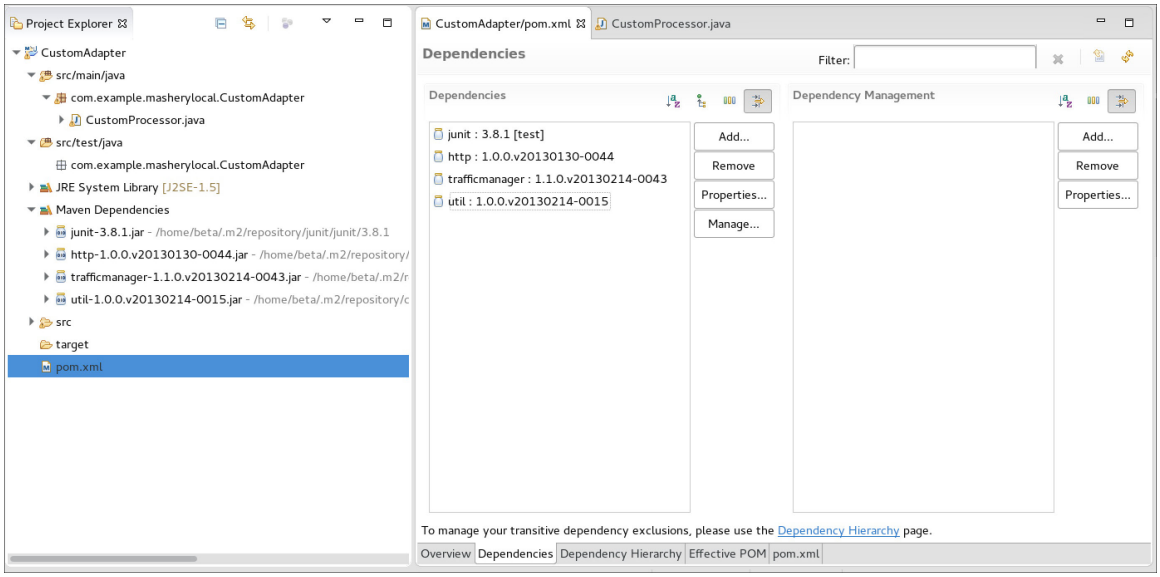


2. Add dependencies to the project. In Eclipse, go to `Project Explorer > pom.xml > Dependencies`.

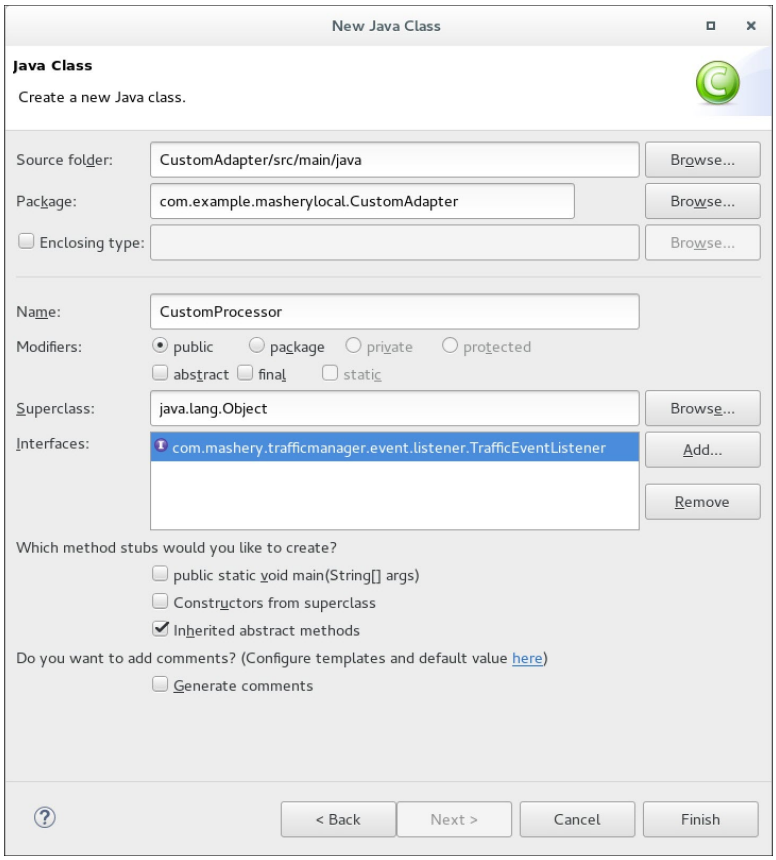




Here is the list of dependencies:



3. Create new Java Class. In Eclipse, go to `Project > New > Class`.
   In the package `com.example.masherylocal.CustomAdapter`, create the following class **`CustomProcessor`**.



This is the reference code:

```
package com.example.masherylocal.CustomAdapter;

import com.mashery.http.server.HTTPServerRequest;
import com.mashery.http.server.HTTPServerResponse;
import com.mashery.trafficmanager.event.listener.TrafficEventListener;
import com.mashery.trafficmanager.event.model.TrafficEvent;
import com.mashery.trafficmanager.event.model.TrafficEventCategory;
import com.mashery.trafficmanager.event.model.TrafficEventType;
import com.mashery.trafficmanager.event.processor.model.ProcessorEvent;
import com.mashery.trafficmanager.model.core.APICall;
import com.mashery.trafficmanager.model.core.ApplicationRequest;
import com.mashery.trafficmanager.model.core.TrafficManagerResponse;
import com.mashery.trafficmanager.processor.ProcessorBean;

@ProcessorBean(enabled=true,
name="com.example.masherylocal.CustomAdapter.CustomProcessor", immediate=true)
public class CustomProcessor implements TrafficEventListener {

    public void handleEvent(TrafficEvent event) {
        TrafficEventType eventType = event.getType();
        if (eventType.getCategory() != TrafficEventCategory.PROCESSOR)
            return;
        ProcessorEvent processorEvent = (ProcessorEvent) event;
        APICall apiCall = processorEvent.getCallContext();

        if (eventType.getName().contentEquals("Pre-Process Event")) {
            ApplicationRequest appRequest = apiCall.getRequest();
            HTTPServerRequest httpRequest = appRequest.getHTTPRequest();
        } else if (eventType.getName().contentEquals("Post-Process Event")) {
            TrafficManagerResponse tmResp = apiCall.getResponse();
            HTTPServerResponse httpResp = tmResp.getHTTPResponse();

httpResp.getHeaders().add("CustomAdapter.CustomProcessor::handleEvent", "Post-
Process Event");
        }
```

```
    }
}
```

4. Build the project in the folder, for example, `/home/beta/work_diysdk/CustomAdapter/`:

```
mvn package
```

5. Use a simple zip command to package the project in the folder, for example, `/home/beta/work_diysdk/CustomAdapter/target/`:

```
zip CustomAdapter-1.0.zip CustomAdapter-1.0.jar
```

6. Unload the zipped package using Mashery Local Cluster Manager.
7. Apply the Custom Processor on the Endpoint using Mashery SaaS Control Center.

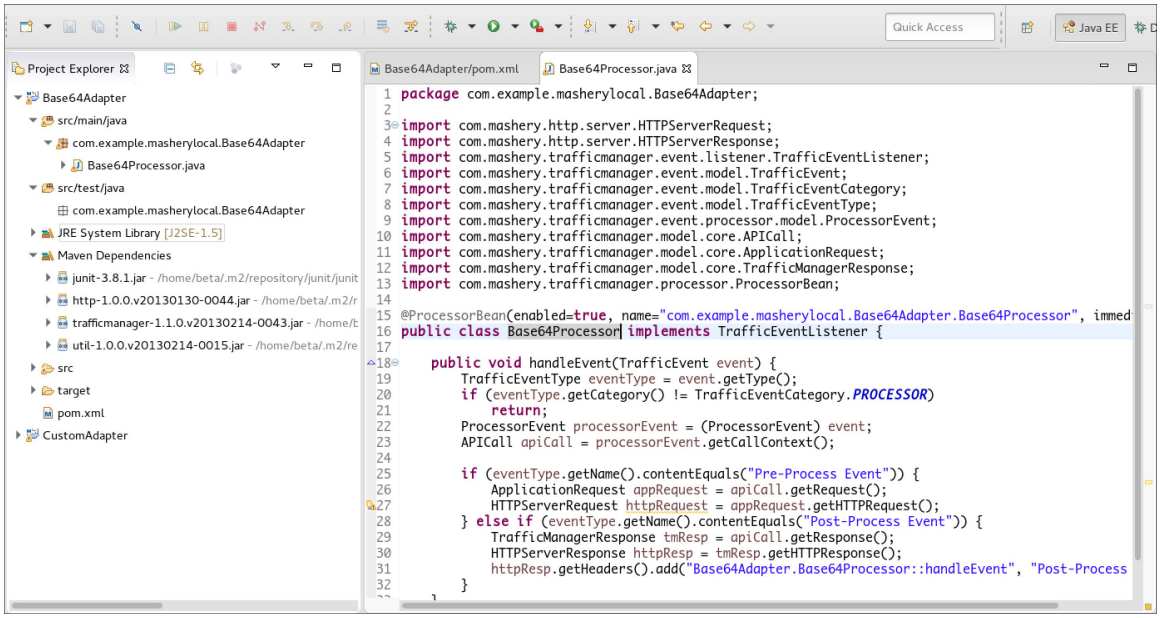
Using the Adapter SDK in Mashery Local with Third-Party Libraries

To use the Adapter SDK in Mashery Local with third-party libraries, follow the steps below:

Procedure

1. Create a new Maven Project "Base64Adapter" in Eclipse, then add dependencies and class, as described in Using the Adapter SDK in Mashery Local with a Single Processor.

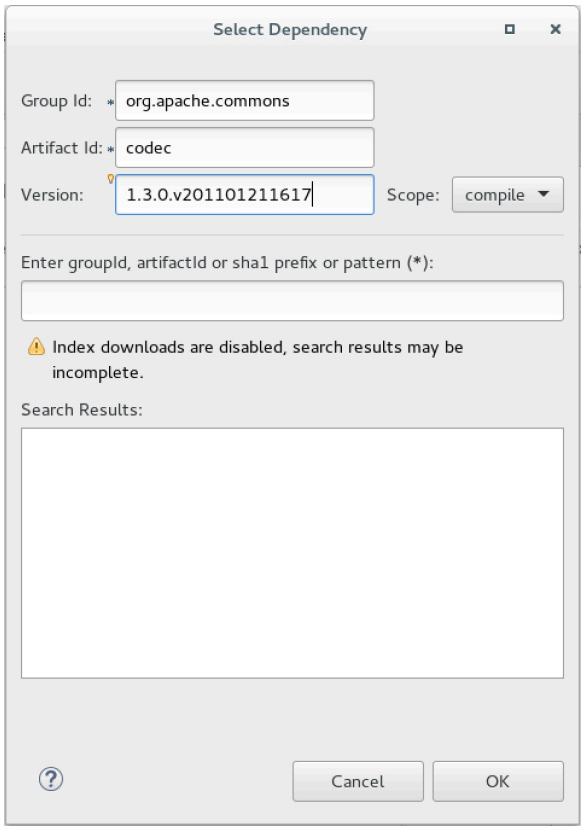
   The project layout should look like the following:



2. Install the third-party library in Maven local repository, for example:

```
mvn install:install-file -Dfile=org.apache.commons.codec_1.3.0.v201101211617.jar
-DgroupId=org.apache.commons \
    -DartifactId=codec -Dversion=1.3.0.v201101211617 -Dpackaging=jar -
DgeneratePom=true
```

3. Add dependency on third-party library in the project:



4. Use third-party library in the project, such as this example, which uses `org.apache.commons.codec_1.3.0.v201101211617.jar`:

```
package com.example.masherylocal.Base64Adapter;

import java.io.UnsupportedEncodingException;

import org.apache.commons.codec.binary.Base64;

import com.mashery.http.server.HTTPServerRequest;
import com.mashery.http.server.HTTPServerResponse;
import com.mashery.trafficmanager.event.listener.TrafficEventListener;
import com.mashery.trafficmanager.event.model.TrafficEvent;
import com.mashery.trafficmanager.event.model.TrafficEventCategory;
import com.mashery.trafficmanager.event.model.TrafficEventType;
import com.mashery.trafficmanager.event.processor.model.ProcessorEvent;
import com.mashery.trafficmanager.model.core.APICall;
import com.mashery.trafficmanager.model.core.ApplicationRequest;
import com.mashery.trafficmanager.model.core.TrafficManagerResponse;
import com.mashery.trafficmanager.processor.ProcessorBean;

@ProcessorBean(enabled=true,
name="com.example.masherylocal.Base64Adapter.Base64Processor", immediate=true)
public class Base64Processor implements TrafficEventListener {

    public void handleEvent(TrafficEvent event) {
        TrafficEventType eventType = event.getType();
        if (eventType.getCategory() != TrafficEventCategory.PROCESSOR)
            return;
        ProcessorEvent processorEvent = (ProcessorEvent) event;
        APICall apiCall = processorEvent.getCallContext();

        if (eventType.getName().contentEquals("Pre-Process Event")) {
            ApplicationRequest appRequest = apiCall.getRequest();
            HTTPServerRequest httpRequest = appRequest.getHTTPRequest();
        } else if (eventType.getName().contentEquals("Post-Process Event")) {
            TrafficManagerResponse tmResp = apiCall.getResponse();
```

```
            HTTPServerResponse httpResp = tmResp.getHTTPResponse();
            try {
                byte[] base64bytes = Base64.encodeBase64("Post-Process
Event".getBytes("UTF-8"));
                String base64str = new String(base64bytes, "UTF-8");

httpResp.getHeaders().add("Base64Adapter.Base64Processor_handleEvent",
base64str);
            } catch (UnsupportedEncodingException e) {
                // TODO Add logging
            }
        }
    }
}
```

5. Update build script to include Class-Path in MANIFEST.MF, as in this example Maven Project, this is done by adding `org.apache.maven.plugins.maven-jar-plugin` to pom.xml:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/
2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example.masherylocal</groupId>
  <artifactId>Base64Adapter</artifactId>
  <version>1.0</version>
  <packaging>jar</packaging>

  <name>Base64Adapter</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>3.0.2</version>
        <configuration>
          <archive>
            <index>true</index>
            <manifest>
              <addClasspath>true</addClasspath>
            </manifest>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>com.mashery</groupId>
        <artifactId>http</artifactId>
        <version>1.0.0.v20130130-0044</version>
    </dependency>
    <dependency>
        <groupId>com.mashery</groupId>
        <artifactId>trafficmanager</artifactId>
        <version>1.1.0.v20130214-0043</version>
    </dependency>
    <dependency>
```

```
        <groupId>com.mashery</groupId>
        <artifactId>util</artifactId>
        <version>1.0.0.v20130214-0015</version>
    </dependency>
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>codec</artifactId>
        <version>1.3.0.v201101211617</version>
    </dependency>
  </dependencies>
</project>
```

Here is an example of MANIFEST.MF:

```
Manifest-Version: 1.0
Built-By: beta
Class-Path: http-1.0.0.v20130130-0044.jar trafficmanager-1.1.0.v201302
 14-0043.jar util-1.0.0.v20130214-0015.jar codec-1.3.0.v201101211617.j
 ar
Created-By: Apache Maven 3.0.5
Build-Jdk: 1.8.0_121
```

6. Build the project in the folder, for example, `/home/beta/work_diysdk/Base64Adapter/`:

```
mvn package
```

7. Use a simple zip command to package the project in the folder, for example, `/home/beta/work_diysdk/Base64Adapter/target/`:

```
cd /home/beta/work_diysdk/Base64Adapter/target/
mkdir lib
cp /home/beta/.m2/repository/org/apache/commons/codec/1.3.0.v201101211617/
codec-1.3.0.v201101211617.jar lib/
zip Base64Adapter-1.0.zip Base64Adapter-1.0.jar lib/codec-1.3.0.v201101211617.jar
```

Here is an example of the zipped contents:

8. Unload the zipped package using Mashery Local Cluster Manager.
9. Apply the Custom Processor on the Endpoint using Mashery SaaS Control Center.

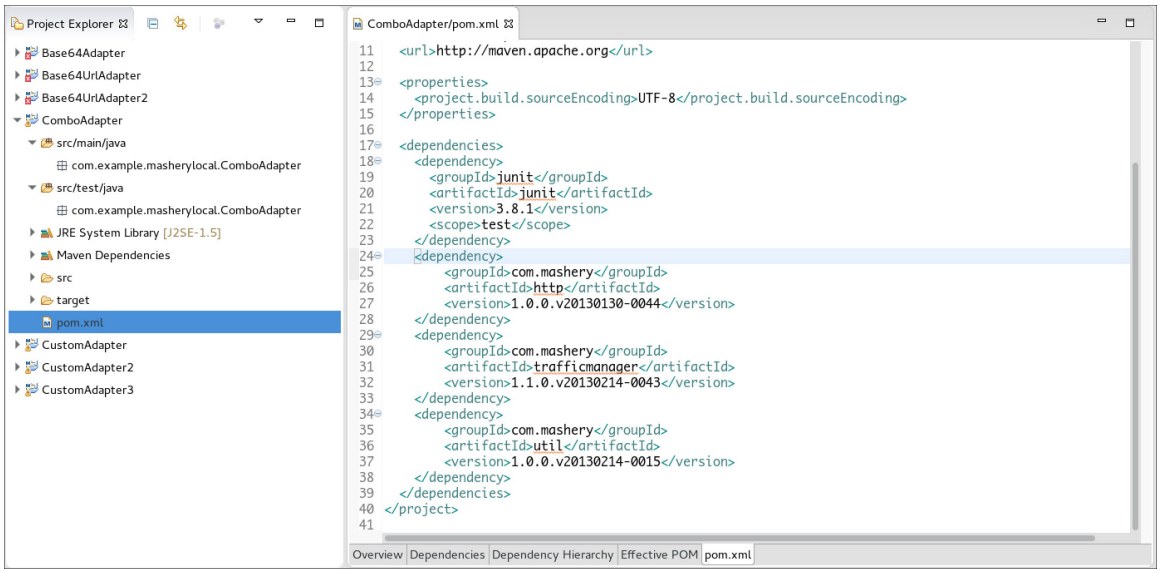
Using the Adapter SDK in Mashery Local with Multiple Processors in One Eclipse Project

To use the Adapter SDK in Mashery Local with multiple processors in one Eclipse project, follow the steps below:

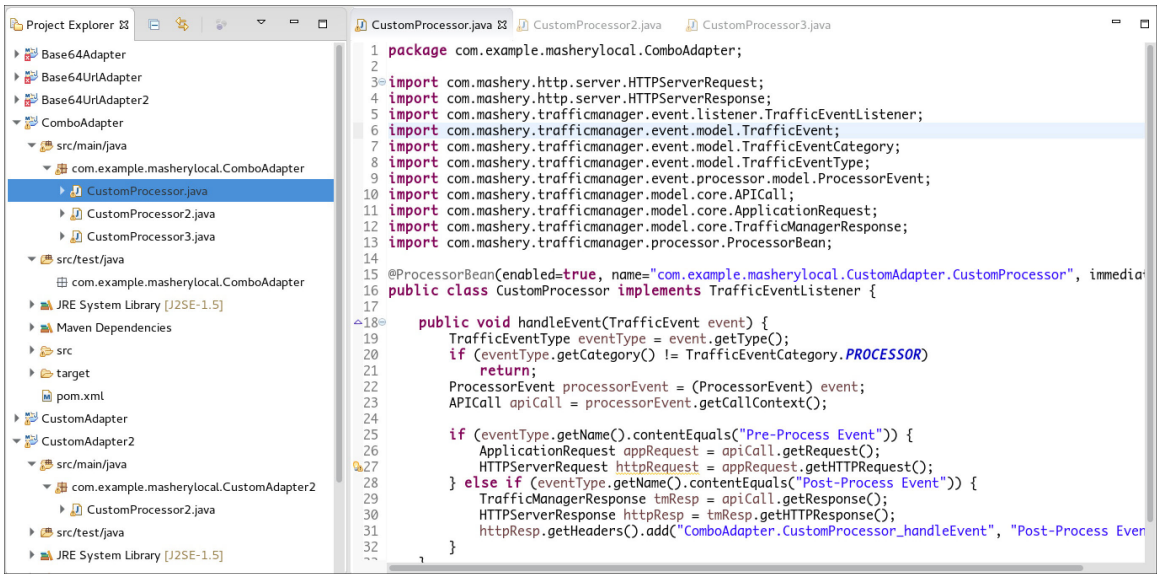
Procedure

1. Create a new Maven Project "ComboAdapter" in Eclipse, then add dependencies, as described in Using the Adapter SDK in Mashery Local with a Single Processor.

   The project layout should look like the following:



2. Add the classes for three custom processors as shown below:



3. Build the project in the folder, for example:

```
cd /home/beta/work_diysdk/ComboAdapter
mvn package
```

4. Use a simple zip command to package the project in the folder, for example:

```
/home/beta/work_diysdk/ComboAdapter/target
zip ComboAdapter-1.0.zip ComboAdapter-1.0.jar
```

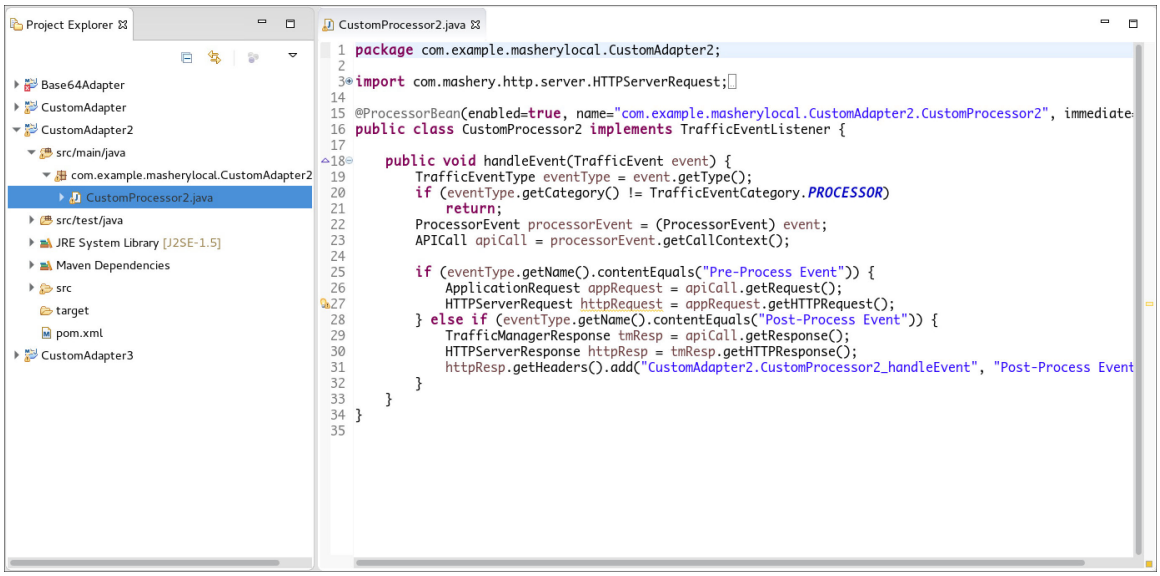
5. Unload the zipped package using Mashery Local Cluster Manager.
6. Apply the Custom Processor on the Endpoint using Mashery SaaS Control Center.

Using the Adapter SDK in Mashery Local with Multiple Processors in One Zip Package

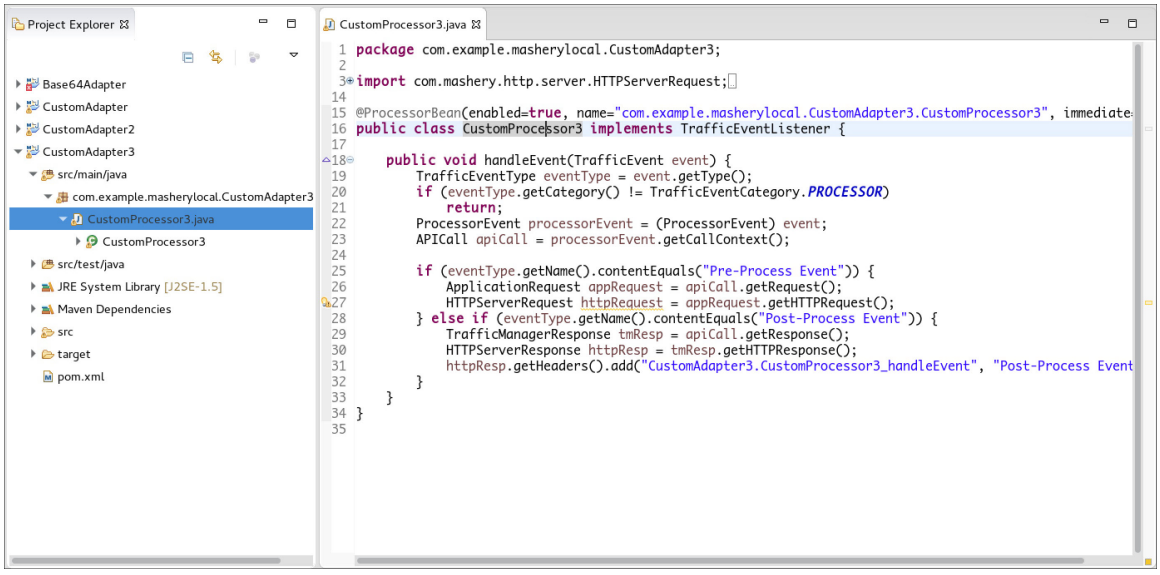
To use the Adapter SDK in Mashery Local with multiple processors in one zip package, follow the steps below:

Procedure

1. Create a new Maven project "CustomAdapter2" in Eclipse, then add dependencies and class, as described in Using the Adapter SDK in Mashery Local with a Single Processor.
   The project layout should look like the following:



2. Create a new Maven Project "CustomAdapter3" in Eclipse, then add dependencies and class, as described in Using the Adapter SDK in Mashery Local with a Single Processor.
   The project layout should look like the following:



3. Build the projects in the folders, for example, `/home/beta/work_diysdk/CustomAdapter2/` and `/home/beta/work_diysdk/CustomAdapter3/`:

```
cd /home/beta/work_diysdk/CustomAdapter2/
mvn package


cd /home/beta/work_diysdk/CustomAdapter3/
mvn package
```

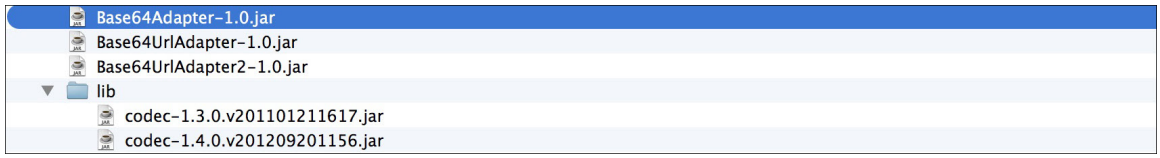
4. Use a simple zip command to package the project in the folder, for example, `/home/beta/work_diysdk/CustomAdapter/target/`:

```
cd /home/beta/work_diysdk/CustomAdapter/target
cp /home/beta/work_diysdk/CustomAdapter2/target/CustomAdapter2-1.0.jar .
cp /home/beta/work_diysdk/CustomAdapter3/target/CustomAdapter3-1.0.jar .
zip CustomAdapters-1.0.zip CustomAdapter-1.0.jar CustomAdapter2-1.0.jar
CustomAdapter3-1.0.jar
```

5. Unload the zipped package using Mashery Local Cluster Manager.
6. Apply the Custom Processor on the Endpoint using Mashery SaaS Control Center.

Using the Adapter SDK in Mashery Local with Multiple Processors in One Package and Third Party Libraries

To use the Adapter SDK in Mashery Local with multiple processors in one package and third party libraries, note that the following package contains three custom processors: `Base64Adapter-1.0.jar references codec-1.3.0.v201101211617.jar`, `Base64UrlAdapter-1.0.jar` and `Base64UrlAdapter2-1.0.jar reference codec-1.4.0.v201209201156.jar`.

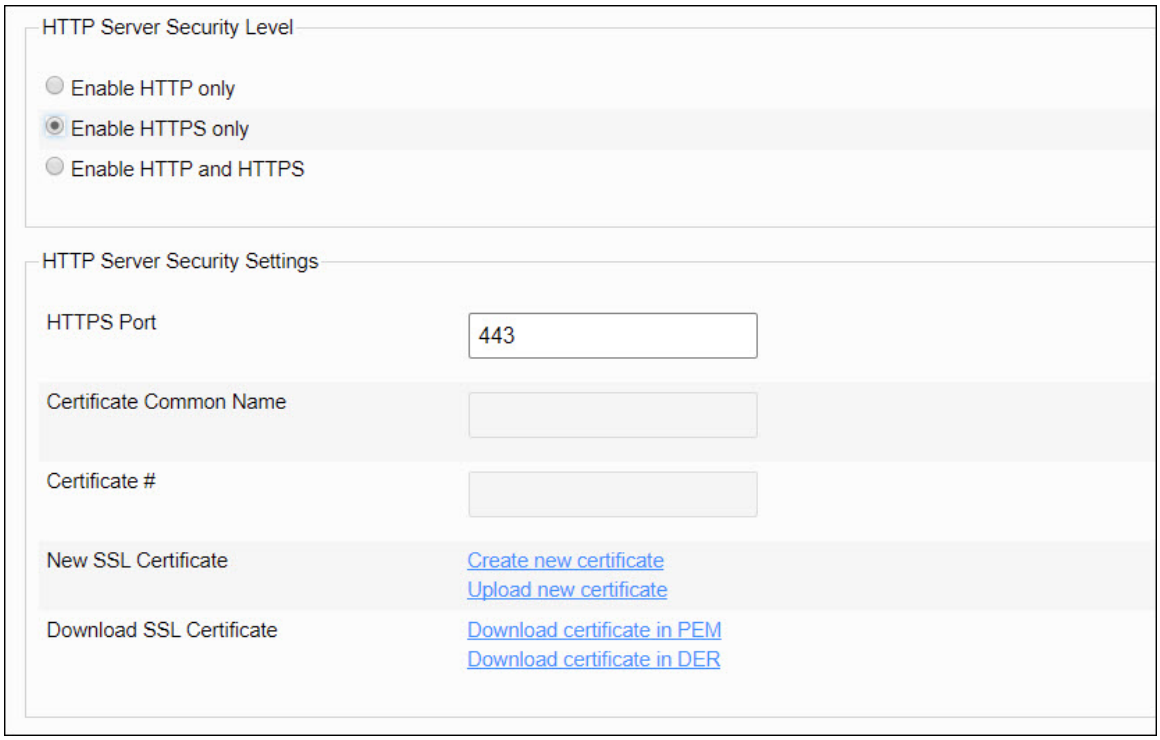


Setting up HTTPS Server using Self-Signed Certificate

To set up HTTPS Server to use a self-signed certificate, follow the steps below:

Procedure

1. On the **Mashery Cluster Manager** tab, click **Instance Management**.
2. Scroll down to **HTTP Server Security Level** section and select **Enable HTTPS only**.



3. In the **HTTP Server Security Settings** section, specify the **HTTPS Port** number (default is 443).

Administrators can change the port number to another number, such as `8443`. For Linux installations, it is not advised to choose a port number below 1000. In addition, the following reserved port numbers are used by TIBCO Mashery Local: 3306, 8081, 8082, 8083, 5489, 11211, 11212, 11213, 11214, 11215, and 11216.

4. Use your server self-signed certificate. Click **Create new certificate**.
The **Create SSL Certificate** window is displayed.

5. Enter a name in the **Certificate Common Name** field, for example, `acme.example.com`, then click **Create**.
6. Click **Save** to save all changes. It will take a few minutes for Mashery Local service to restart.

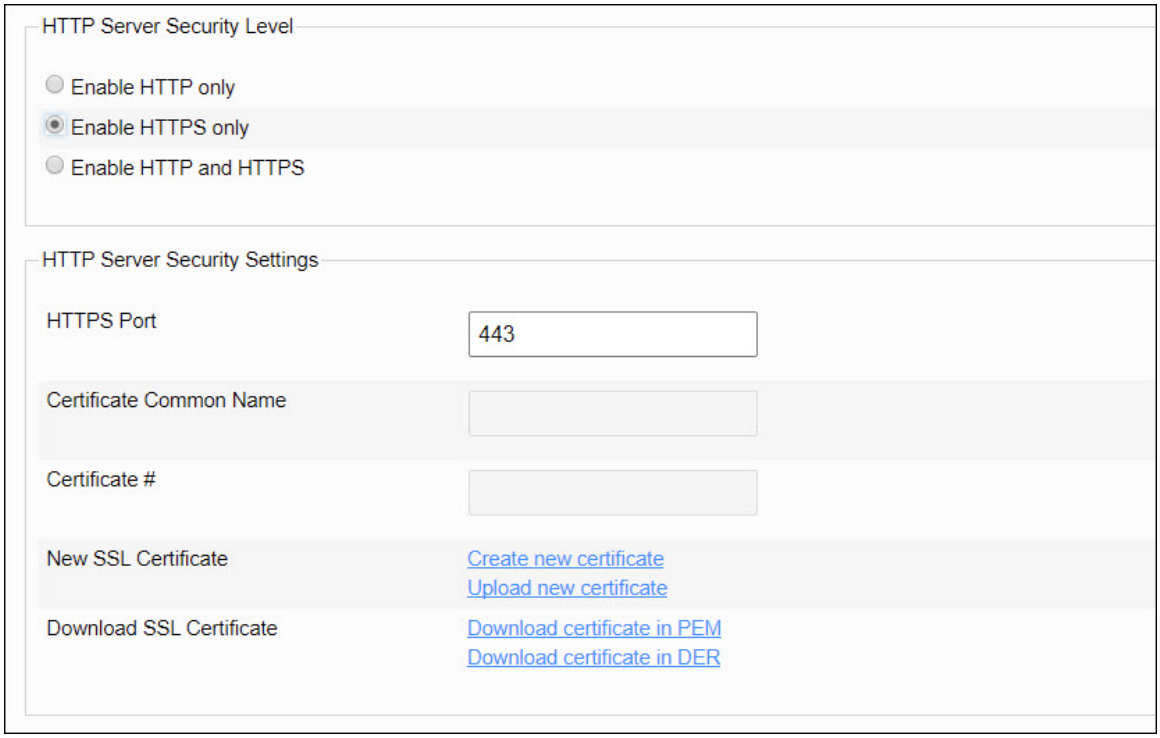
Setting up HTTPS Server using Customer-Provided Certificate

To set up HTTPS Server to use a self-signed certificate, follow the steps below:

Procedure

1. On the **Mashery Cluster Manager** tab, click **Instance Management**.

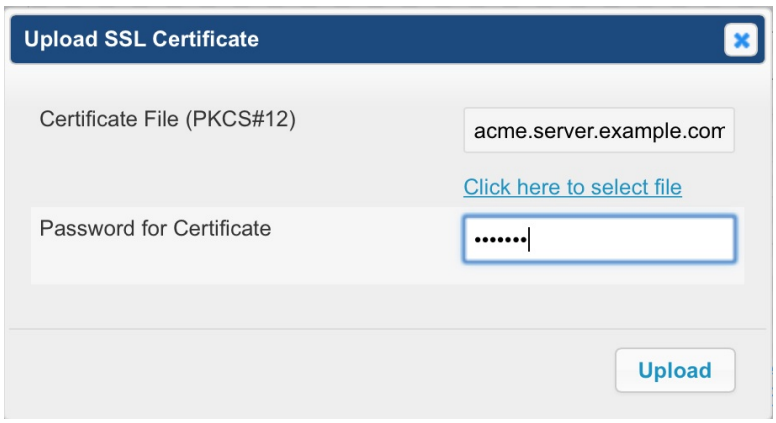
2. Scroll down to **HTTP Server Security Level** section and select **Enable HTTPS only**.



3. In the **HTTP Server Security Settings** section, specify the **HTTPS Port** number (default is 443).

> Administrators can change the port number to another number, such as `8443`. For Linux installations, it is not advised to choose a port number below 1000. In addition, the following reserved port numbers are used by TIBCO Mashery Local: 3306, 8081, 8082, 8083, 5489, 11211, 11212, 11213, 11214, 11215, and 11216.

4. Use your CA certificate. Click **Upload new certificate**.
The **Upload SSL Certificate** window is displayed.



5. In the **Upload SSL Certificate** window, click the **Click here to select file** link, browse to the CA certificate file, then click **Upload**.

> The Certificate File should be in PKCS#12 format.

6. Click **Save** to save all changes. It will take a few minutes for Mashery Local service to restart.

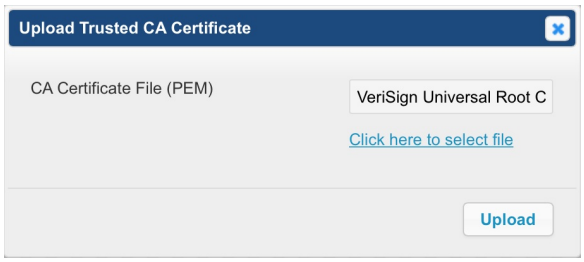
Configuring and Using the HTTPS Client Feature without Mutual Authentication

To use the HTTPS Client Feature without Mutual Authentication, you will need to configure **Trust Management** settings in Mashery Cluster Manager and configure an HTTPS Client Profile in Mashery SaaS (Control Center).

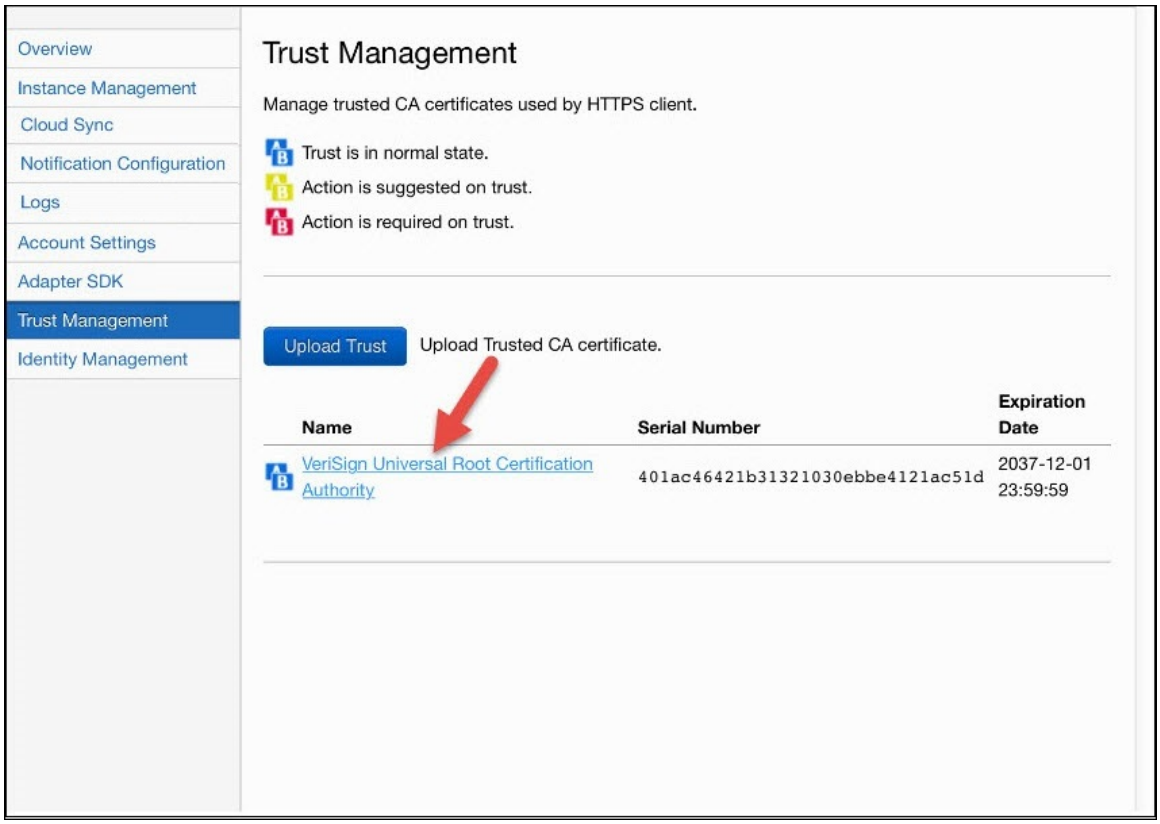
For Mashery Local:

Procedure

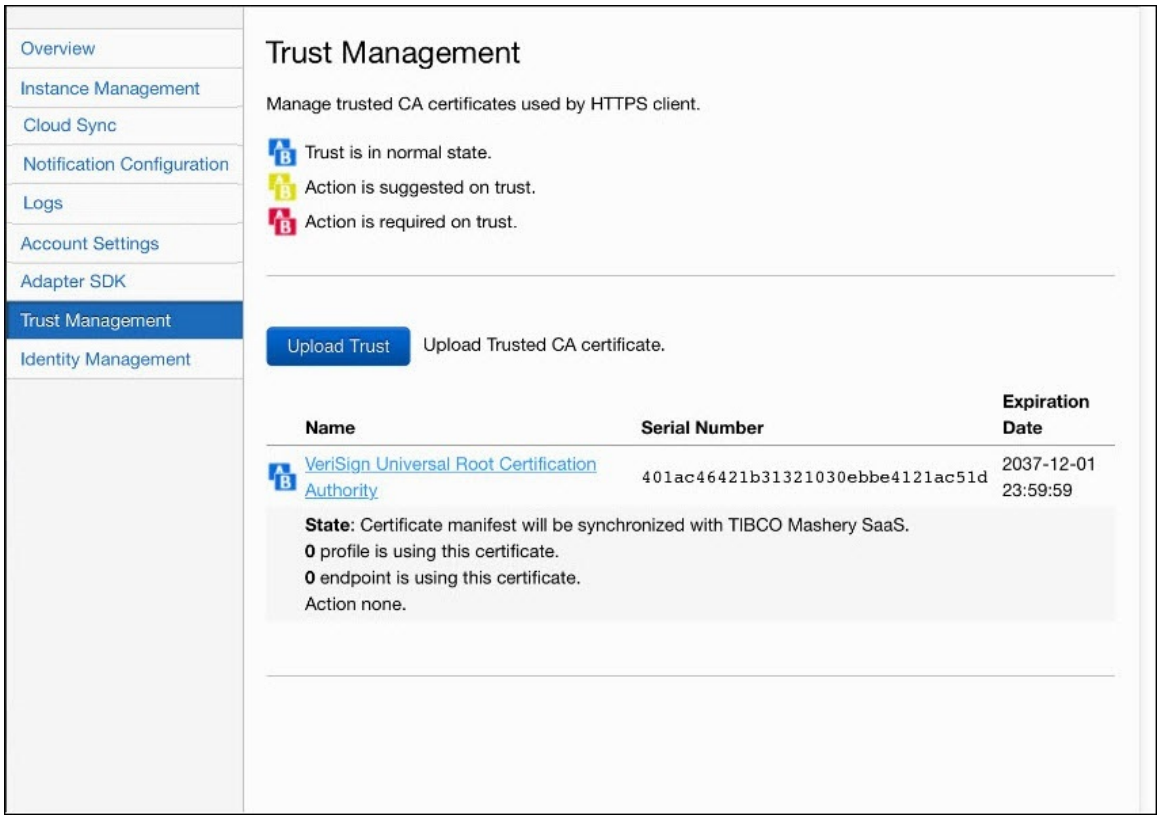
1. On the **Mashery Cluster Manager** tab, click **Trust Management**.
2. Click **Upload Trust**. The **Upload Trusted CA Certificate** window is displayed.



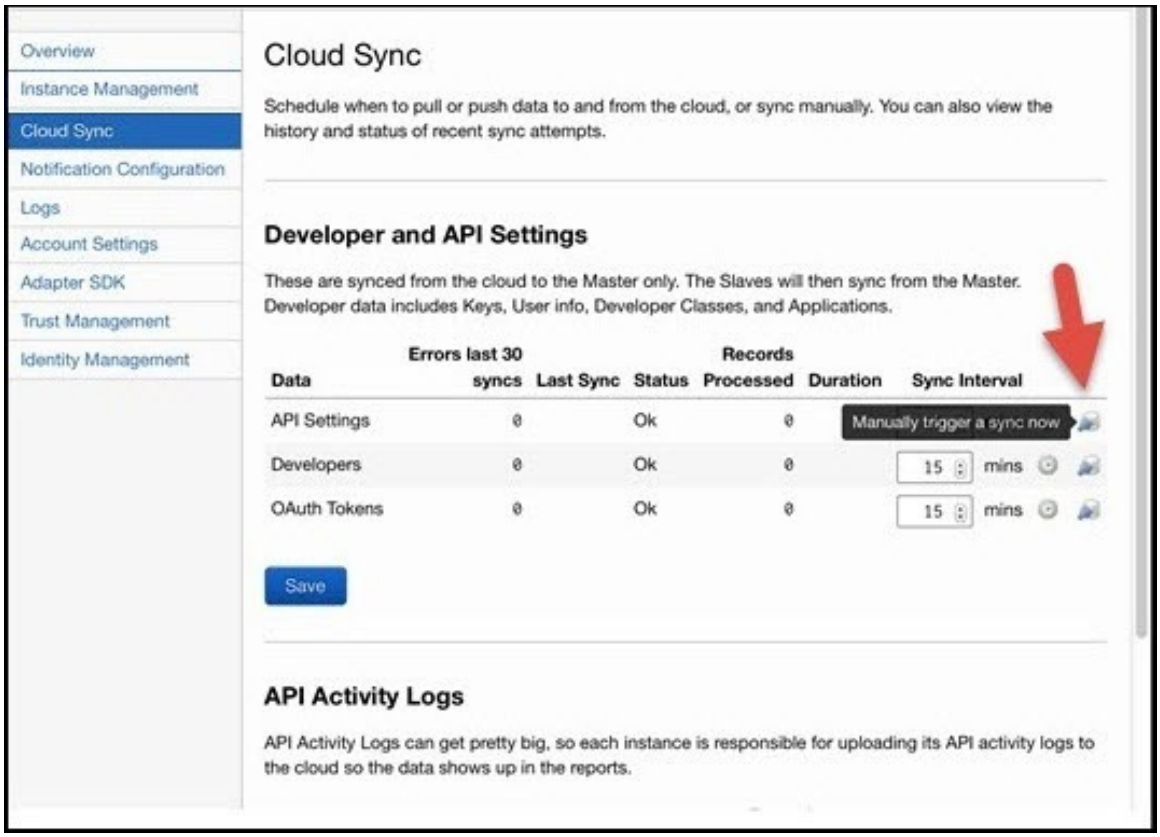
3. In the **Upload Trusted CA Certificate** window, click the **Click here to select file** link, browse to the CA certificate, then click **Upload**.
4. The CA Certificate is now added as a trusted certificate. Click the link next to the certificate name to view the state.



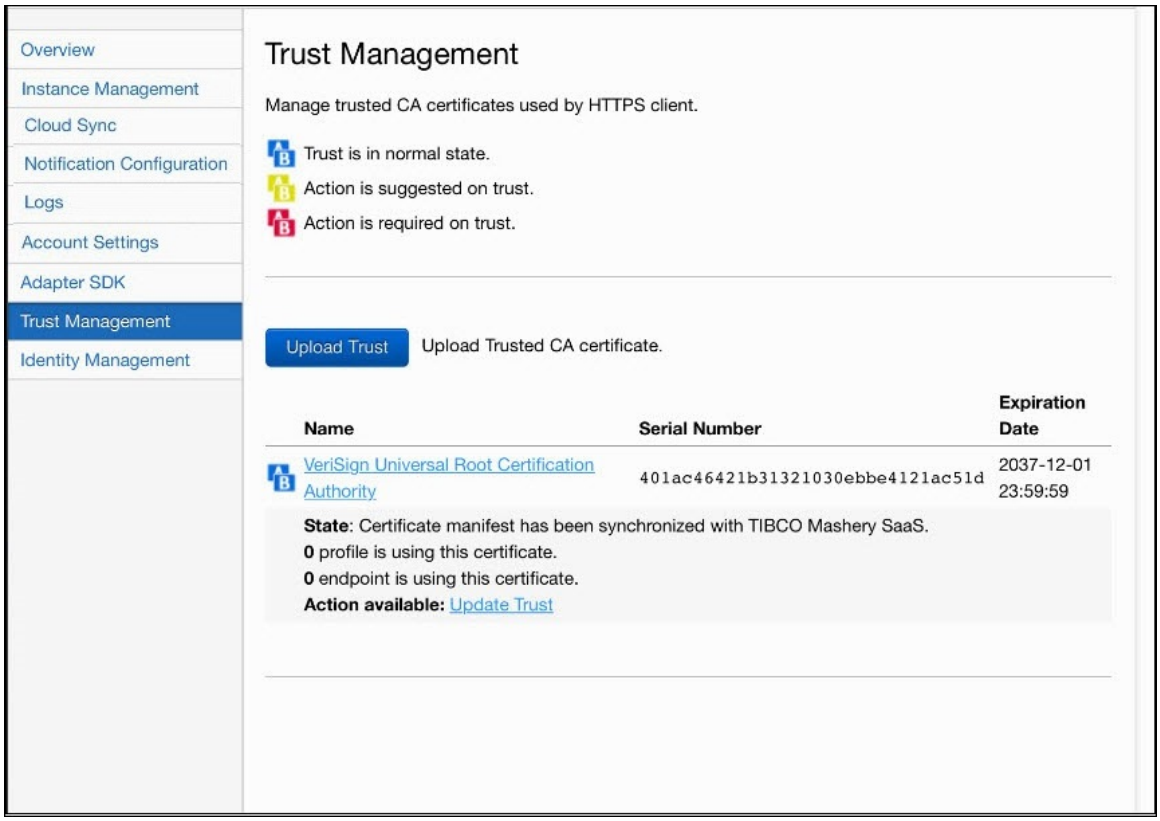
5. In this example, the State is `Certificate manifest will be synchronized with TIBCO Mashery SaaS`. Mashery Local will synchronize automatically, or you can manually trigger a sync in **Cloud Sync** settings.



6. To manually trigger a sync, on the **Mashery Cluster Manager** tab, click **Cloud Sync**.



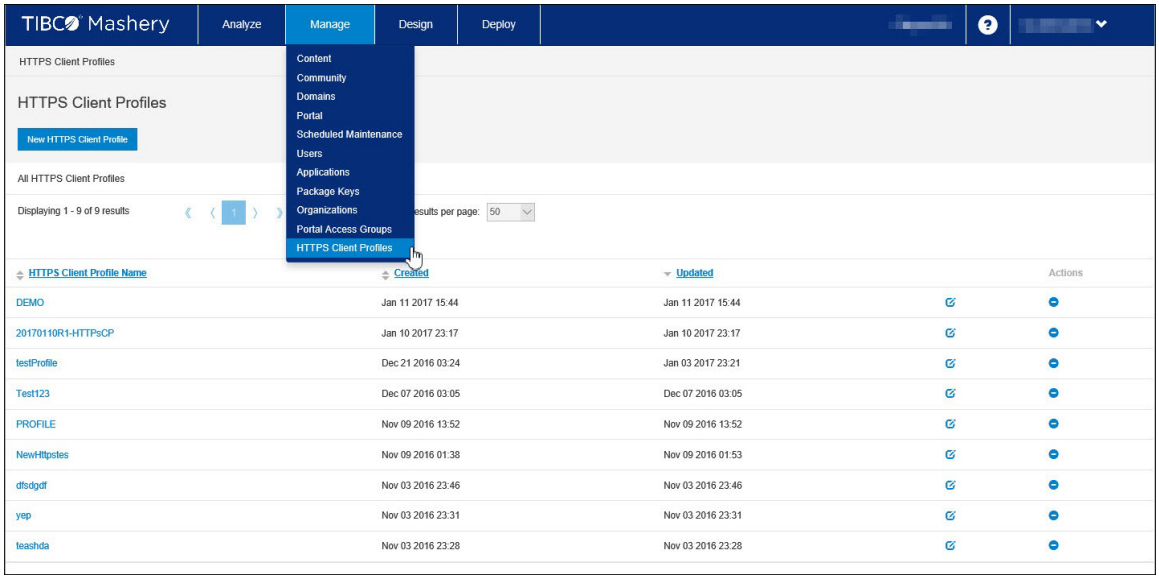
7. In the **Developer and API Settings** section, for API Settings, click the **Sync** button to manually trigger a sync. This will make the certificate metadata available instantly in Mashery SaaS; otherwise, the sync will occur according to the minutes defined for the **Sync Interval** setting.
8. After the certificate becomes available in SaaS, the State changes to `Certificate manifest has been synchronized with TIBCO Mashery SaaS`.



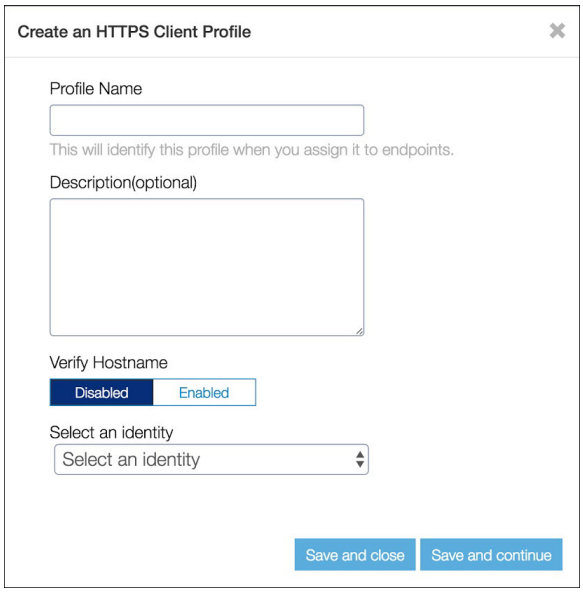
What to do next

To create an HTTPS Client Profile in TIBCO Mashery SaaS, follow the steps below:

1. Click `Manage > HTTPS Client Profiles`. The **HTTPS Client Profiles** window is displayed.



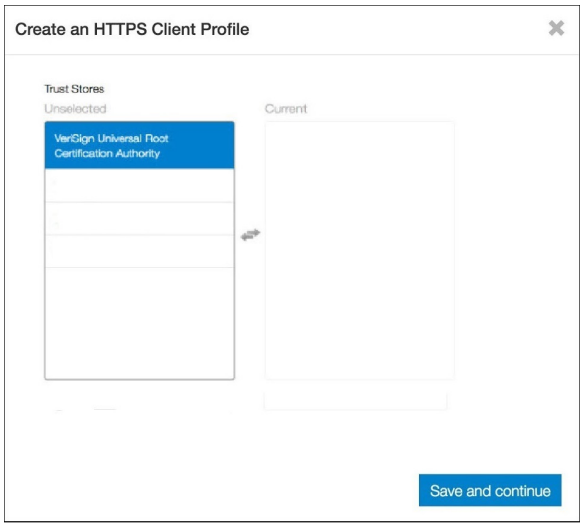
2. Click the **New HTTPS Client Profile** button. The **Create an HTTPS Client Profile** window is displayed.



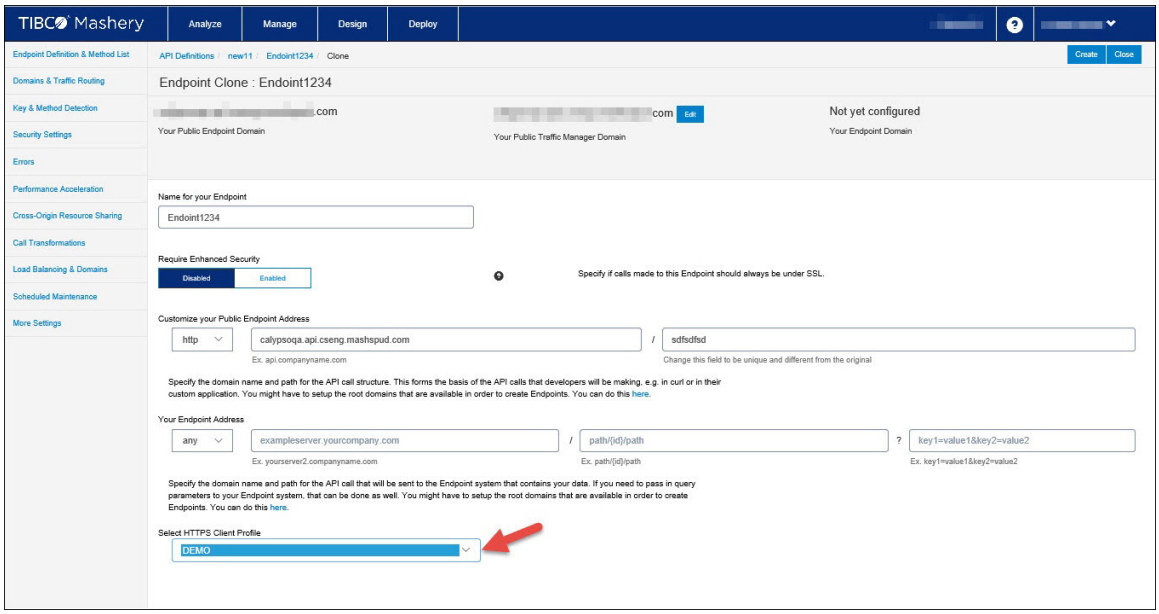
3. On the **Create an HTTPS Client Profiles** window, enter information in the following fields:

Field	Description
Profile name	Enter a name for the HTTPS client profile.
Description	(Optional) Enter a description for the HTTPS client profile.
Verify Hostname	Select one of the following: • **Disabled**: Click to not have the hostname verified. • **Enabled**: Click to have the hostname verified.
Select an identity	Select an identity for the HTTPS client profile.

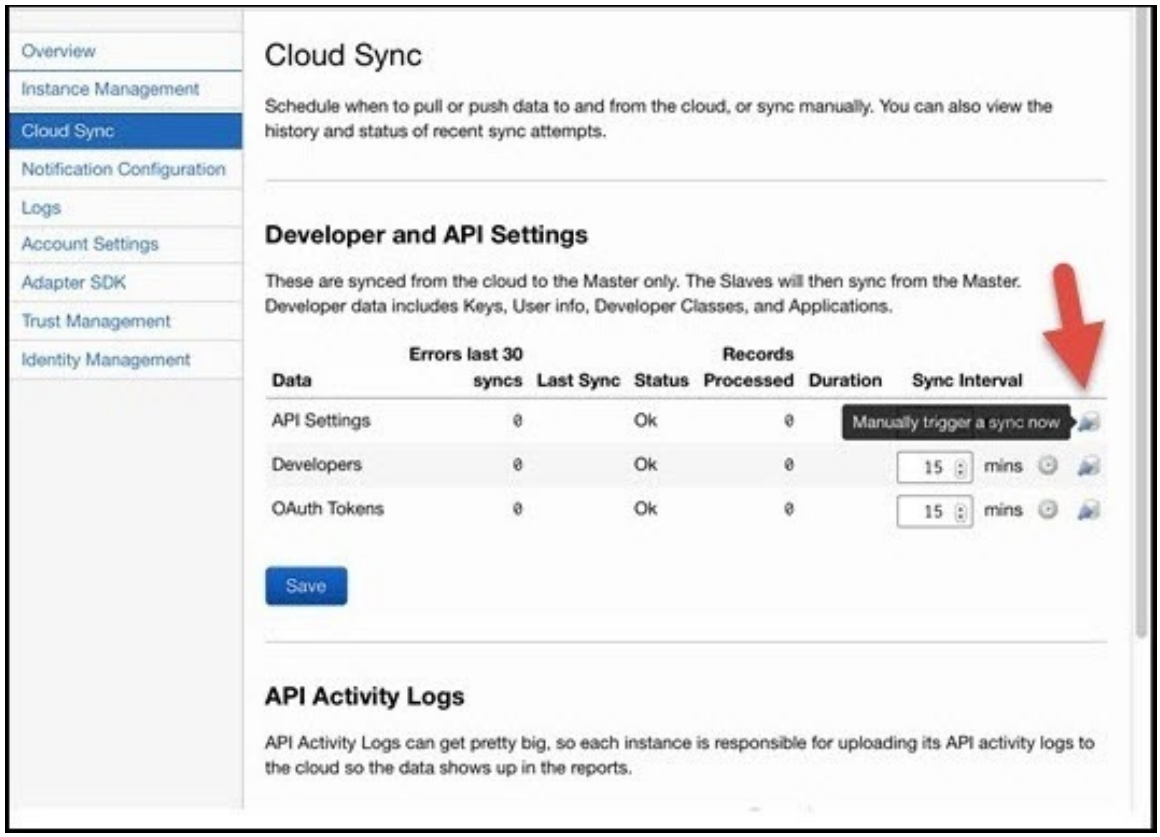
4. Click **Save and Continue**.
5. A second **Create an HTTPS Client Profile** page displays.



6. In the **Unselected** list of the **Trust stores** section, click the Trusted CA Certificate you uploaded in Mashery Cluster Manager to move it to the **Current** list, then click **Save and continue** to finish creating the HTTPS Client Profile. Once the HTTPS Client Profile is created, you can then select the profile when creating an endpoint on the **Endpoint Create: New Endpoint Definition** page.
7. To assign the HTTPS Client Profile to an endpoint, click `Design > API Definitions > Domains & Traffic Routing`.
8. On the **Domains & Traffic Routing** page for the existing endpoint of your API definition (or, **Endpoint Create: New Endpoint Definition** page for a new endpoint), use the **Select HTTP Client Profile** field to select the HTTPS Client Profile you just created, then click **Save** (or **Create**). The endpoint is now associated with the HTTPS Client Profile.



9. Log back into Mashery Cluster Manager, go to the **Cloud Sync** tab, and click the manual sync button. This syncs the HTTPS Client Profile and Endpoint configuration updates to Mashery Local, and the HTTP Client Profile is now in use for the customer.



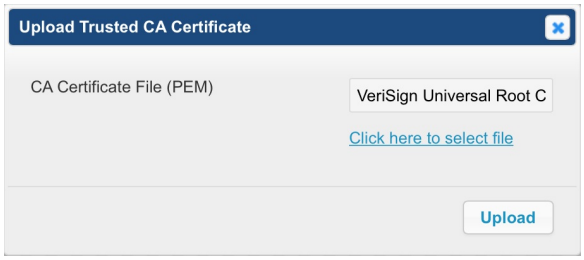
Configuring and Using the HTTPS Client Feature with Mutual Authentication

To use the HTTPS Client Feature without Mutual Authentication, you will need to configure **Trust Management** and **Identity Management** settings in Mashery Cluster Manager and configure an HTTPS Client Profile in Mashery SaaS (Control Center).

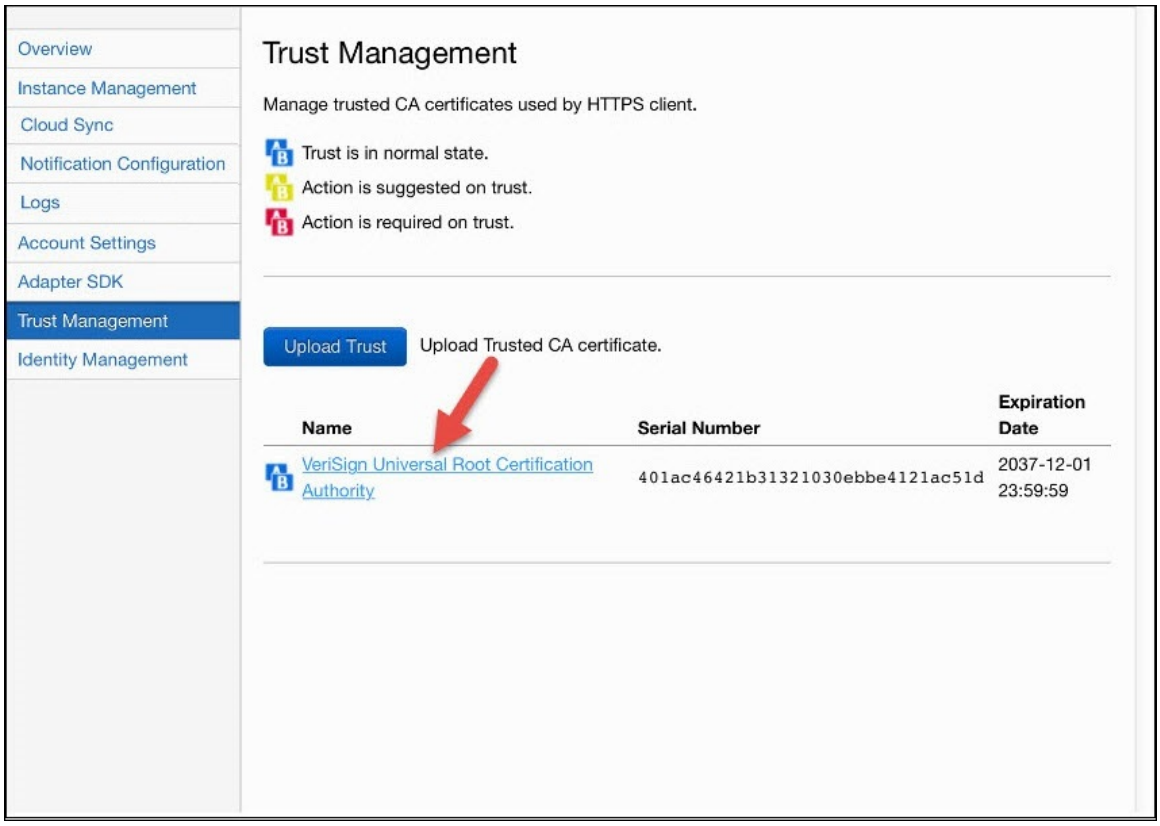
For Mashery Local:

Procedure

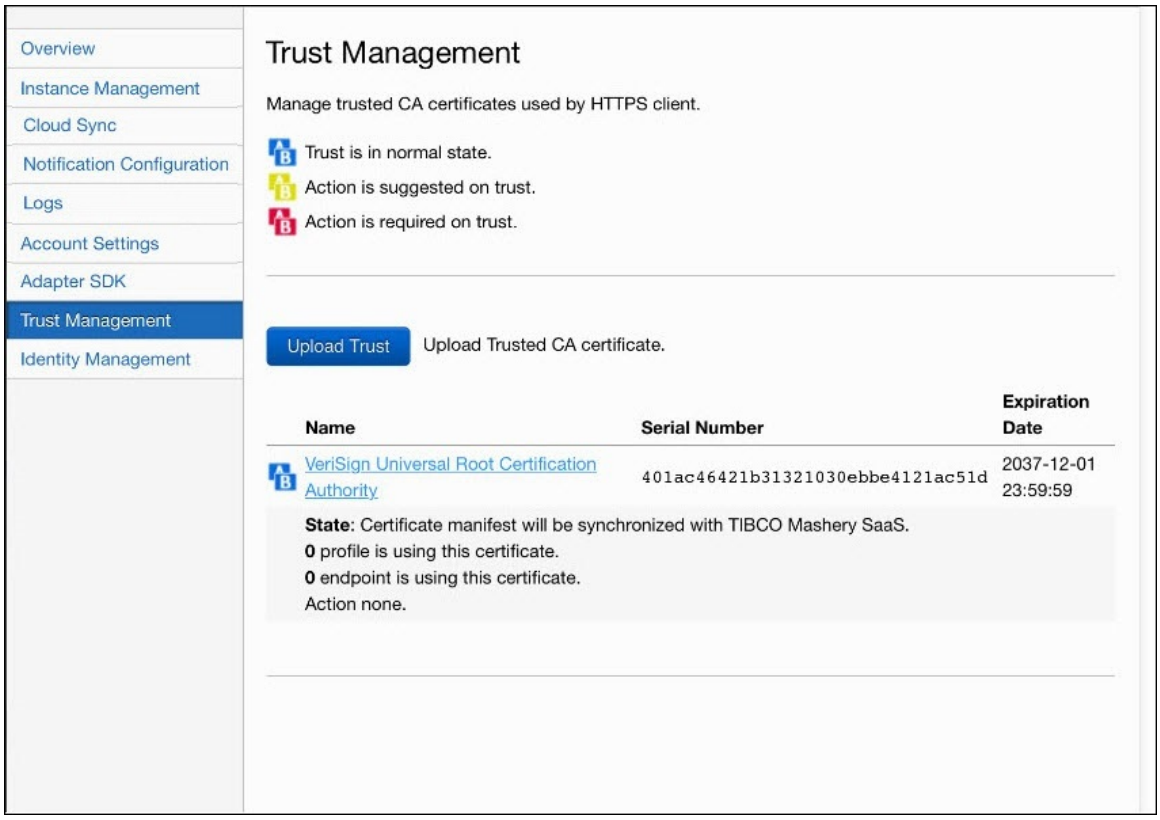
1. On the **Mashery Cluster Manager** tab, click **Trust Management**.
2. Click **Upload Trust**. The **Upload Trusted CA Certificate** window is displayed.



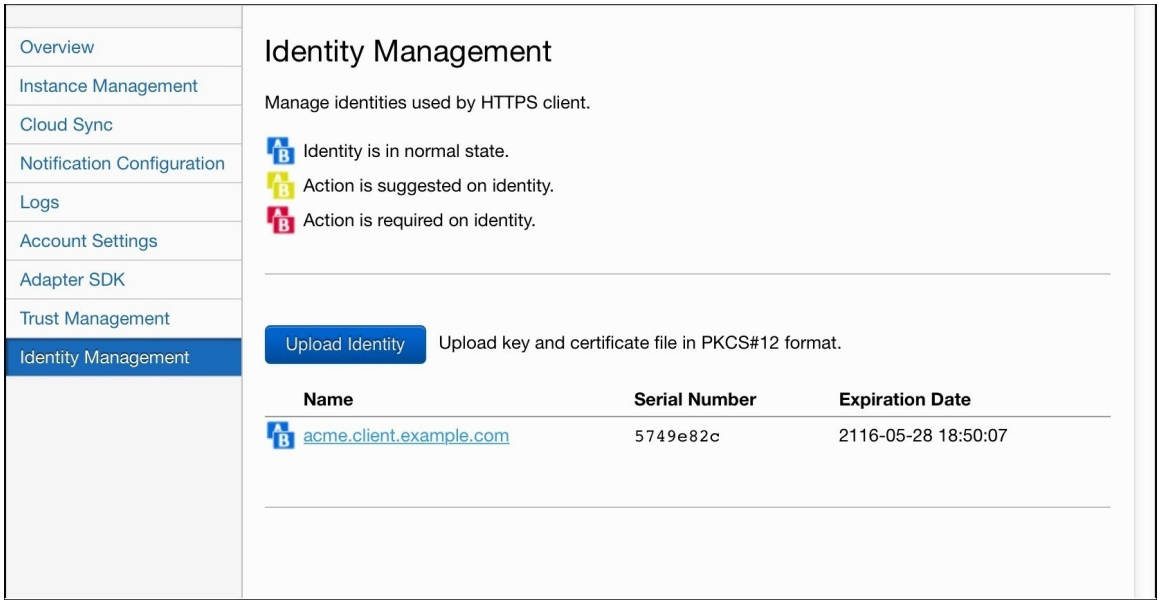
3. In the **Upload Trusted CA Certificate** window, click the **Click here to select file** link, browse to the CA certificate, then click **Upload**.
4. The CA Certificate is now added as a trusted certificate. Click the link next to the certificate name to view the state.



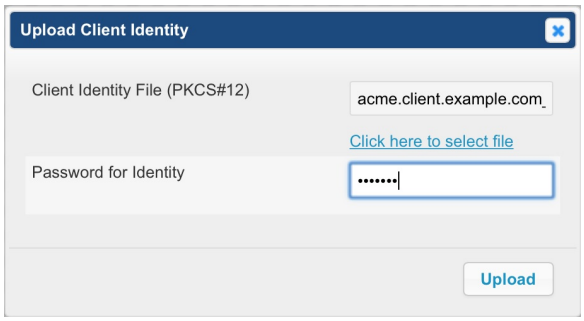
5. In this example, the State is `Certificate manifest will be synchronized with TIBCO Mashery SaaS`. Mashery Local will synchronize automatically, or you can manually trigger a sync in **Cloud Sync** settings.



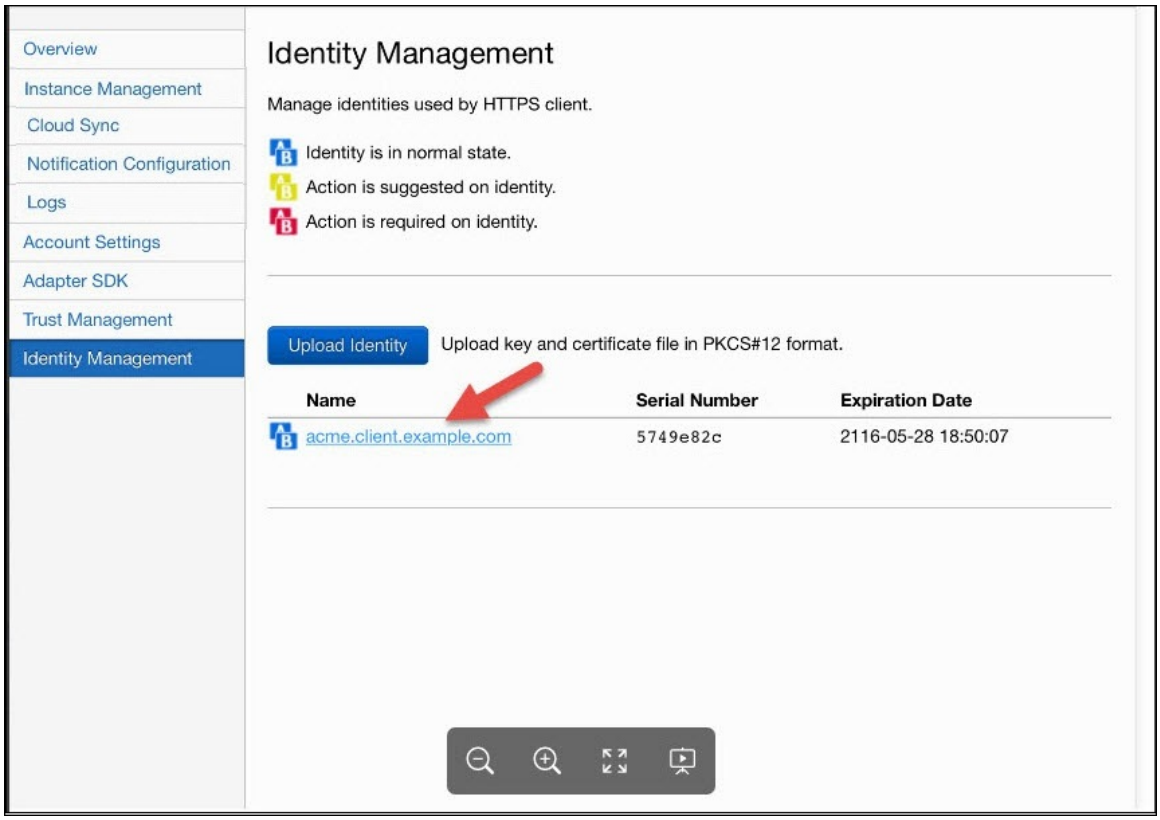
6. On the **Mashery Cluster Manager** tab, click **Identity Management**.



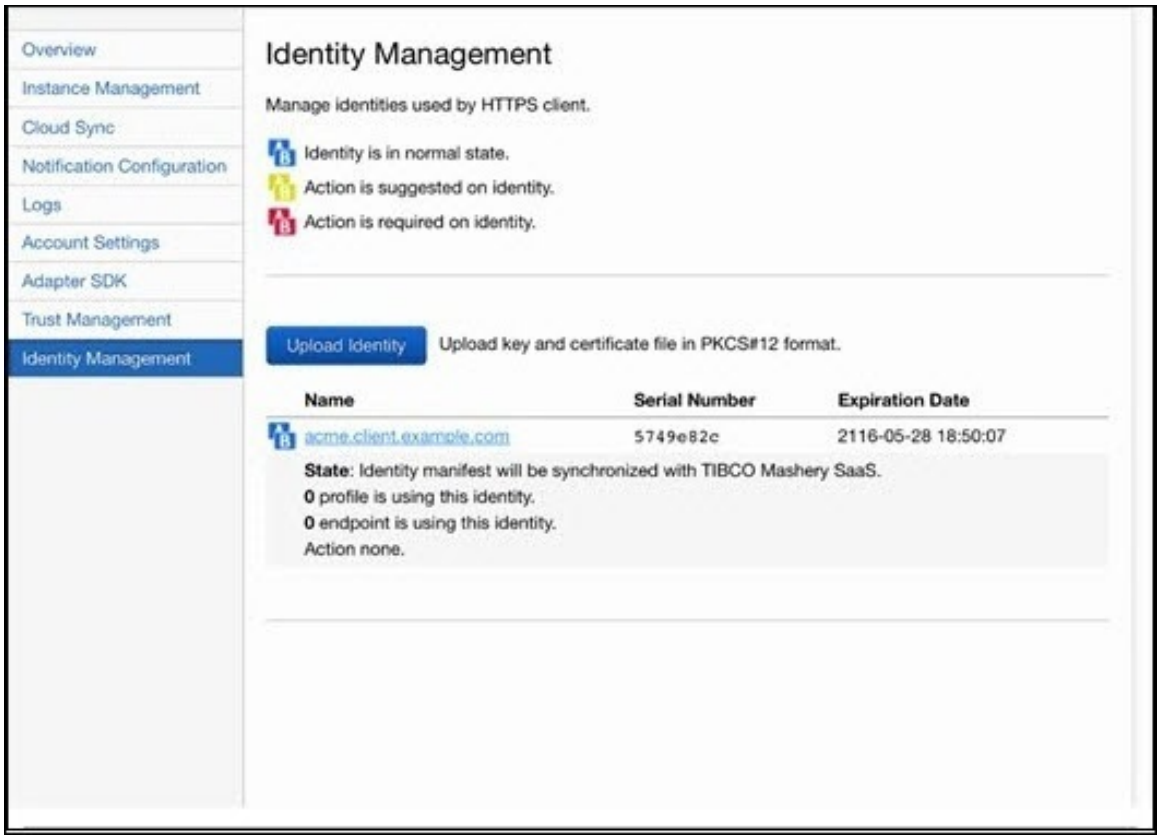
7. Click **Upload Identity**. The **Upload Client Identity** window is displayed.



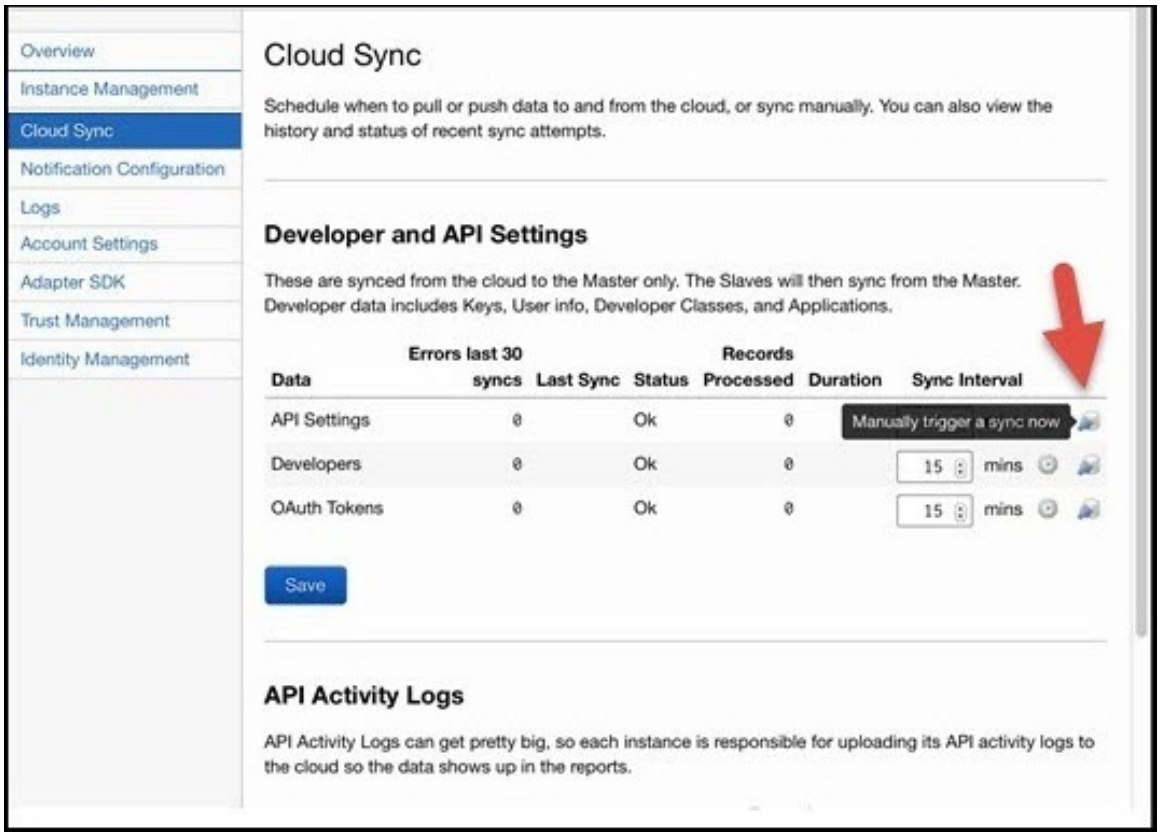
8. In the **Upload Client Identity** window, click the **Click here to select file** link, browse to the identity file (PKCS#12 format), enter the **Password for Identity**, then click **Upload**.
9. The Client Identity is now added as an Identity. Click the link next to the Identity name to view the state.



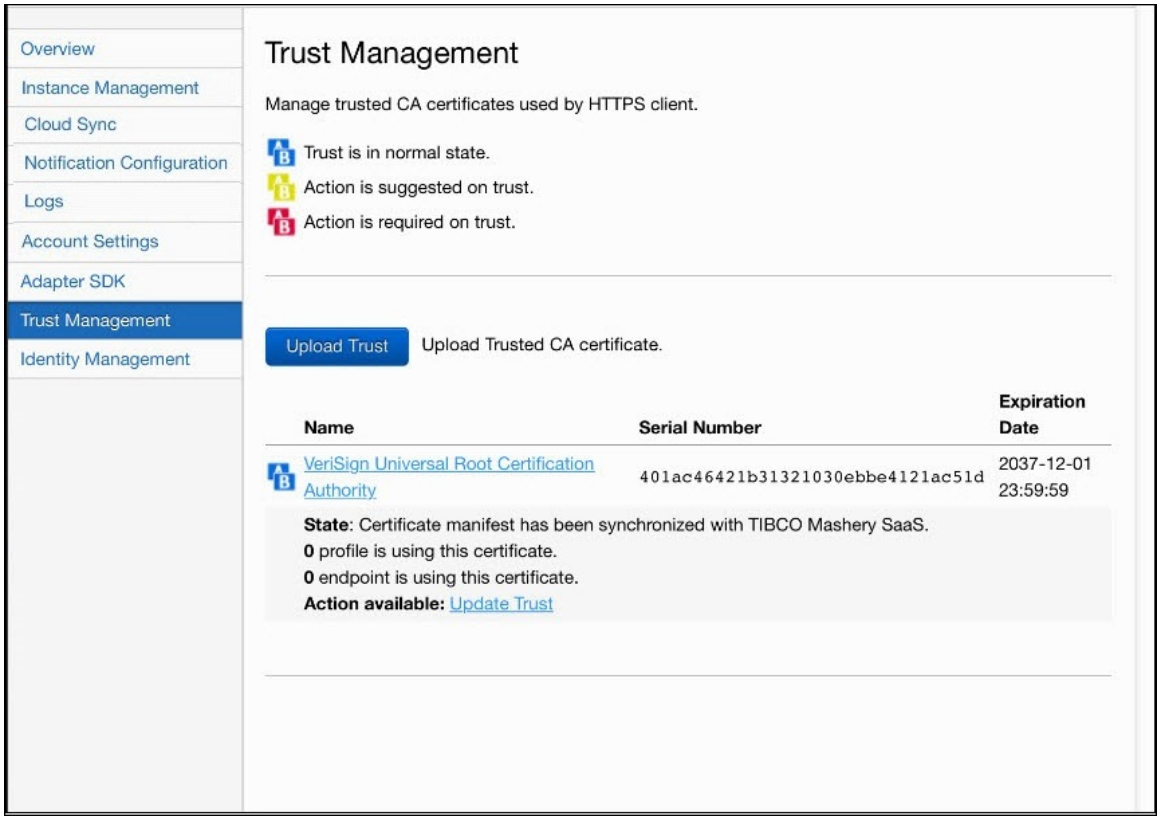
10. In this example, the State is `Identity manifest will be synchronized with TIBCO Mashery SaaS`. Mashery Local will synchronize automatically, or you can manually trigger the sync in **Cloud Sync** settings.



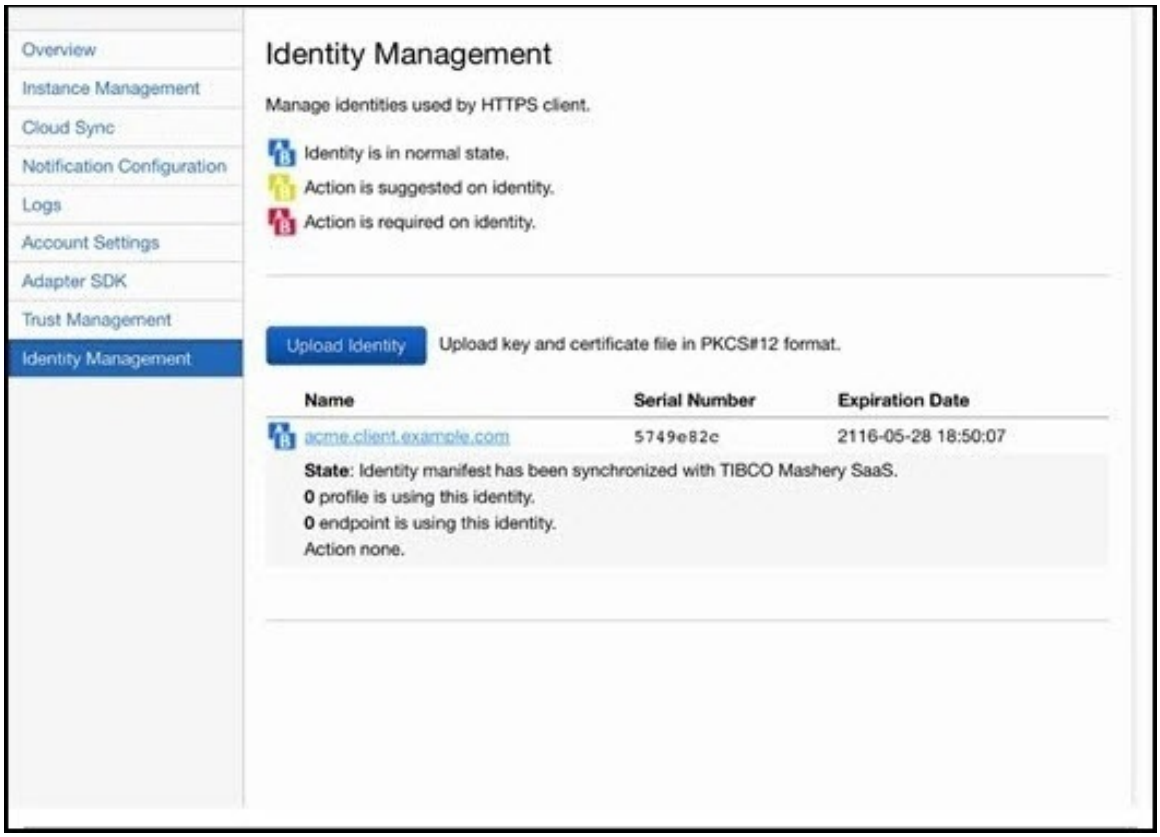
11. To manually trigger a sync, on the **Mashery Cluster Manager** tab, click **Cloud Sync**.



12. In the **Developer and API Settings** section, for API Settings, click the **Sync** button to manually trigger a sync. This will make the certificate and identity metadata available instantly in Mashery SaaS; otherwise, the sync will occur according to the minutes defined for the **Sync Interval** setting.
13. On the **Trust Management** tab, after the certificate becomes available in SaaS, the State changes to `Certificate manifest has been synchronized with TIBCO Mashery SaaS.`



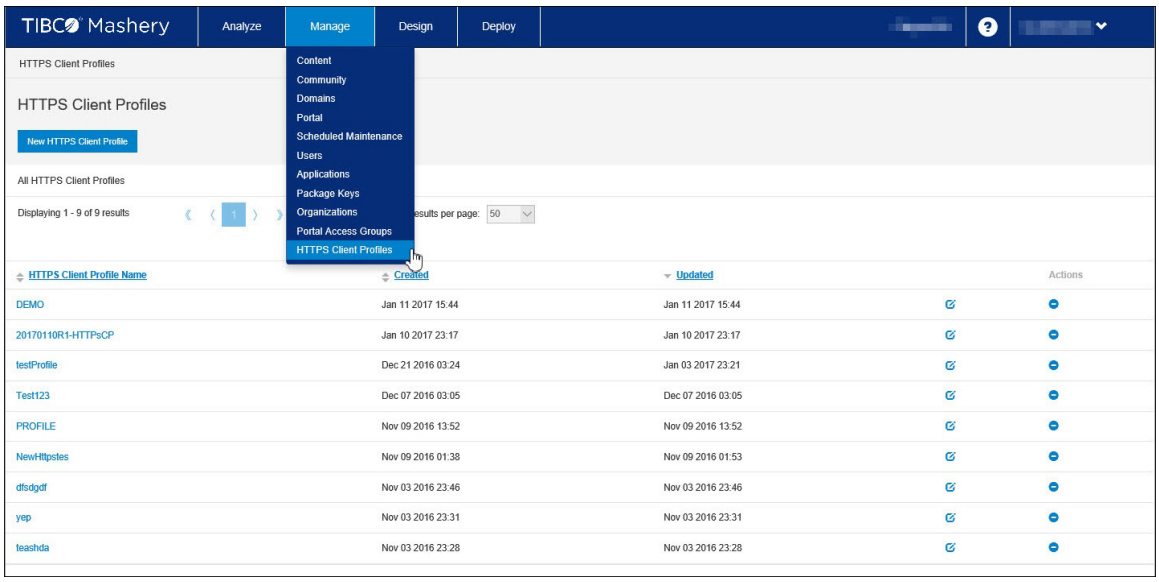
14. On the **Identity Management** tab, after the Identity becomes available in SaaS, the State changes to: `Identity manifest has been synchronized with TIBCO Mashery SaaS.`



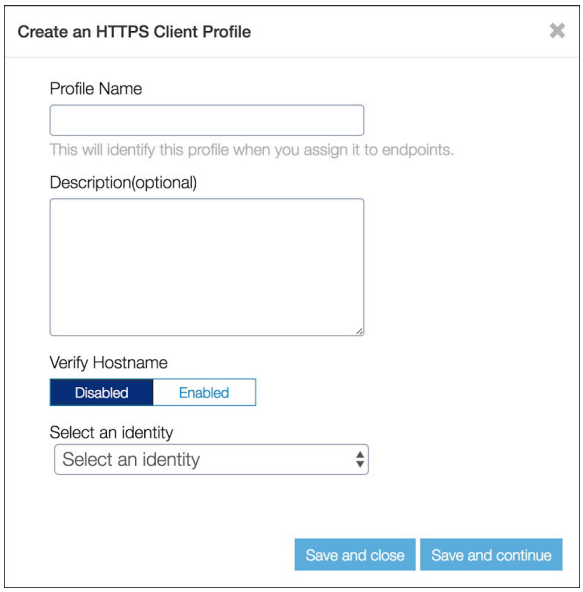
What to do next

To create an HTTPS Client Profile in TIBCO Mashery SaaS, follow the steps below:

1. Click `Manage > HTTPS Client Profiles`. The **HTTPS Client Profiles** window is displayed.



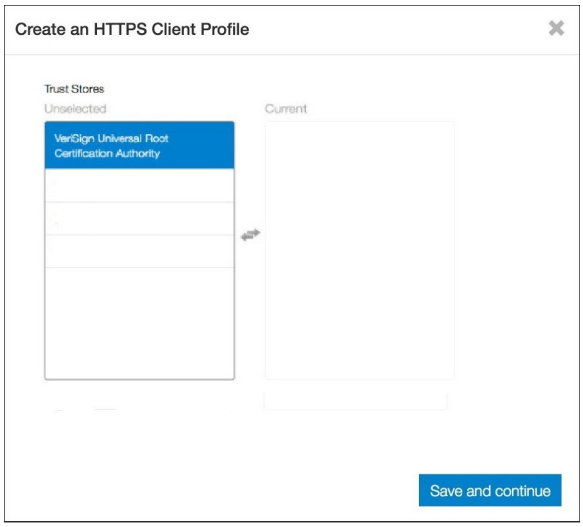
2. Click the **New HTTPS Client Profile** button. The **Create an HTTPS Client Profile** window is displayed.



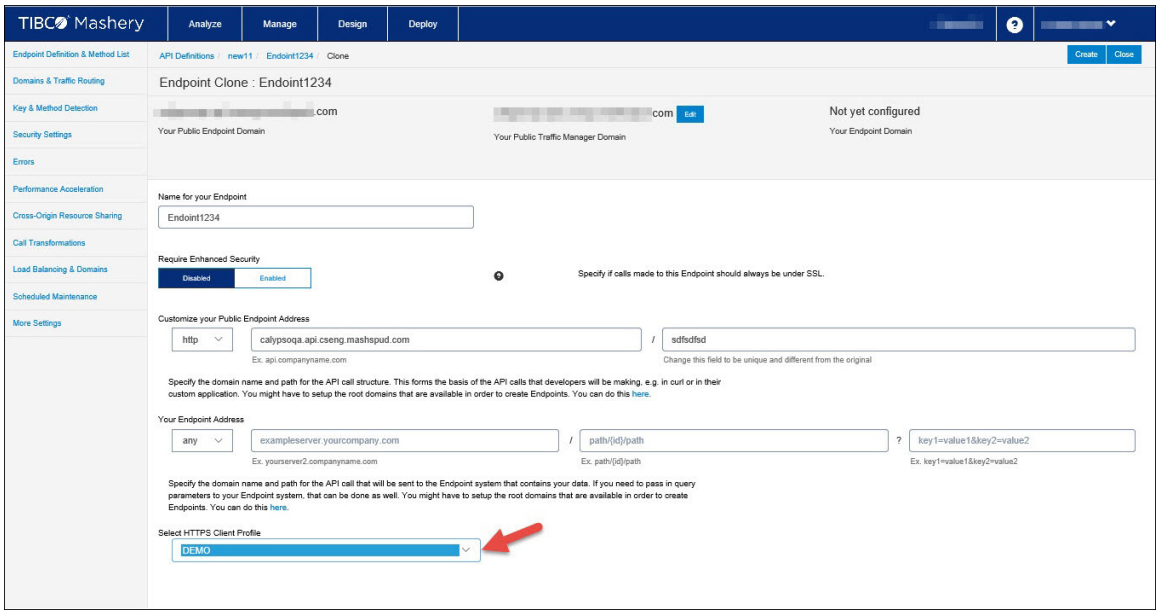
3. On the **Create an HTTPS Client Profiles** window, enter information in the following fields:

Field	Description
Profile name	Enter a name for the HTTPS client profile.
Description	(Optional) Enter a description for the HTTPS client profile.
Verify Hostname	Select one of the following: • **Disabled**: Click to not have the hostname verified. • **Enabled**: Click to have the hostname verified.
Select an identity	Select an identity for the HTTPS client profile.

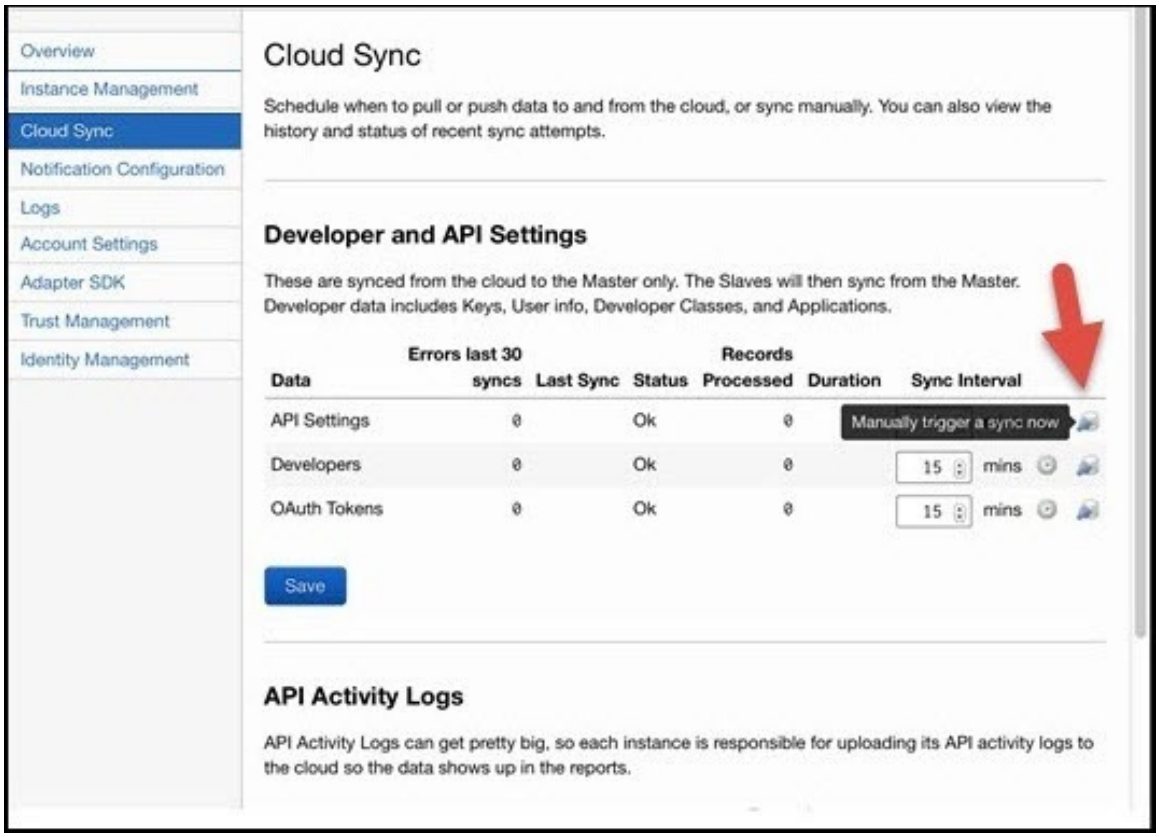
4. Click **Save and Continue**.
5. A second **Create an HTTPS Client Profile** page displays.



6. In the **Unselected** list of the **Trust stores** section, click the Trusted CA Certificate you uploaded in Mashery Cluster Manager to move it to the **Current** list, then click **Save and continue** to finish creating the HTTPS Client Profile. Once the HTTPS Client Profile is created, you can then select the profile when creating an endpoint on the **Endpoint Create: New Endpoint Definition** page.
7. To assign the HTTPS Client Profile to an endpoint, click `Design > API Definitions > Domains & Traffic Routing`.
8. On the **Domains & Traffic Routing** page for the existing endpoint of your API definition (or, **Endpoint Create: New Endpoint Definition** page for a new endpoint), use the **Select HTTP Client Profile** field to select the HTTPS Client Profile you just created, then click **Save** (or **Create**). The endpoint is now associated with the HTTPS Client Profile.



9. Log back into Mashery Cluster Manager, go to the **Cloud Sync** tab, and click the manual sync button. This syncs the HTTPS Client Profile and Endpoint configuration updates to Mashery Local, and the HTTP Client Profile is now in use for the customer.



Enabling Java SSL Debug Logging

To enable Java SSL debug logging for Mashery Local, follow the steps below:

Procedure

1. Add the following setting in "/opt/javaproxy/proxy/proxy.ini" in ml-tm container:
   ```
   -Djavax.net.debug=all
   ```
2. Restart javaproxy:
   ```
   service javaproxy restart
   ```
3. Send requests to Mashery Local, watch log in "/var/log/javaproxy-runtime.log". For example:
   ```
   tail -f /var/log/javaproxy-runtime.log
   ```