Time Series Service

Contents

Copyright GE Digital

Time Series Service Overview

About the Time Series Service

Time Series data is a sequence of data points collected at set time intervals over a continuous period of time. Sensor data is an example of a common way to generate time series data. A Time Series data store requires a measurement with a corresponding timestamp. The Time Series service provides an attributes field to include additional relevant details about that specific data point, such as units or site, for example, "Site":"SanFrancisco".

Time Series data can consist of regular data (data sampled at regular time intervals), or irregular data, for example, data that is recorded only when a certain event occurs (so always at random times).

Note: Time Series is not an archival service. The time to live (TTL) period for data stored in the Time Series service is two years from the date of ingestion, after which the data will be deleted. You should back up your data if you need to keep it longer.

The Time Series service provides the following benefits:

- Efficient storage of time series data.
- Indexing the data for quick retrieval.
- High availability so you can access and query your data from anywhere via HTTP.
- Horizontal scalability.
- Millisecond data point precision.

Component	Description
Data Ingestion	The data-ingestion layer provides the ability to ingest real-time streaming data.
Data Query	The query API allows you to query data, with support for grouping of data points by tags, time ranges, or values, as well as aggregations. You can also filter by attributes to narrow your results.

The below high-level architecture diagram shows different ways you can send data to the Time Series service for ingestion, as well as how users can interact with the query service.

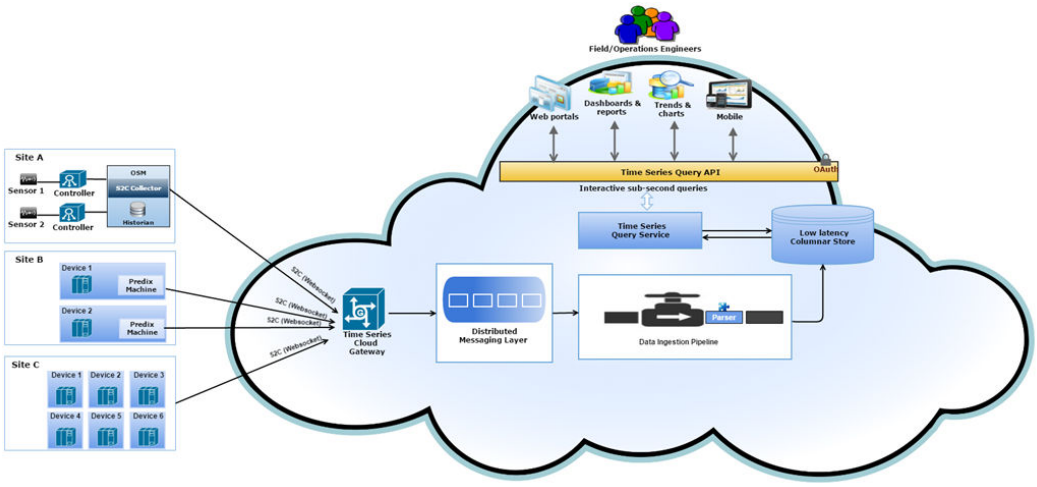

Figure 1: Time Series Service Architecture

The service ingests data through a WebSocket connection, which can then be queried using a REST API with HTTP requests. The below are some example ingestion configurations:

- **Site A – Historian**
  Sensors produce time series data associated with a tag name and send it to Historian for storage and management. S2C subscribes to tags and collects generated data from those tags only. S2C requests a web socket connection from a gateway application that is used for data ingestion.
- **Site B – WebSocket River utility**
  The WebSocket River establishes a connection when you first attempt a data transfer and keeps that channel open for as long as possible. Each data transfer verifies that the websocket connection is open. If the connection has been closed, the service opens a new connection. For more information about WebSocket River, see WebSocket River.
  To communicate with the Time Series gateway, the data must be structured as shown in #unique_6.
- **Site C – Direct WebSocket connection**
  Devices use an application to communicate directly with the websocket.

Additional Information

Exploring Time Series Tutorials

Setting up and Configuring the Time Series Service

Time Series Service Setup

Like other Predix platform services, authentication for the Time Series service is controlled by the designated trusted issuer and is managed by the User Account and Authentication (UAA) web service. You must set up a UAA service instance as the trusted issuer before getting started with the Time Series service. For information about authentication and authorization in Predix services, see About Security Services.

Accounts

You should have the following accounts to use Predix services:

- A Predix.io account. See #unique_11.
  When you register for a Predix.io account, an org and space is created for you in Cloud Foundry.
- A Github account. Go to https://github.com/join.

Software

Software	Version	Description
Cloud Foundry CLI	Latest stable binary version	Use the Cloud Foundry CLI to deploy and manage applications and services. Download the latest stable binary from *https://github.com/cloudfoundry/cli#downloads*.
[AED1] (Optional) Java SE Development Kit (JDK)	8	If you would like to use the Time Series Client Library, you will need Java 8. Download the JDK from *https://www.oracle.com/downloads/index.html*.
(Optional) Maven		If you would like to use the Time Series Client Library, you can use Maven to download that dependency. Other build tools like Gradle should work as well. You can download Maven from *https://maven.apache.org/download.cgi*.

Note: Git is not required to use the project. While you should be using some form of version control, the type of version control is up to you.

Task Roadmap

Step	Description
(Optional) Configure your proxy settings.	Depending on your location and network configuration, you may need to configure your proxy settings to access remote resources. See #unique_12.
(Optional) Update your Maven settings to use the Predix platform Artifactory.	To use the Time Series Java client library, you need to include it in your project. See #unique_13.
(Optional) Deploy a Predix Hello World Web application.	#unique_14.
Create a UAA client.	See #unique_15.
Create the Time Series service instance.	See #unique_16.
Bind your application to the service instance.	See #unique_17.
Create an OAuth client for the Time Series service.	See #unique_18.
Update the OAuth2 client to use Time Series.	See #unique_19.
Add the required Time Series scopes.	See #unique_20.
Add Predix zone token scopes to your application.	See #unique_21.

Best Practices

Goals

•

Design Best Practices

Do not differentiate data with tag names. It is better to differentiate data with attributes, rather than with detailed tag names. For example, instead of...

Preparing Data

Sizing Guidelines

Expected data points, SLA times expected

Guidelines for figuring out how many web socket connections you need. Formula to create # of data sockets, types of data at the lowest end assuming no attributes, and one tag you can assume 15K datapoints max for 512KB size message adding tag would add one time 80KB per tag if attribute exist then we recommend reduce the number of datapoints accordingly 512 KB messages would need 20K requests to be fulfilled we can consume 2 requests per sec

Expected Response Time	Number of Web Sockets
10000	One connection
1000	100 connections
10	1000 connections

Data Interpretation Patterns

Creating a UAA Service Instance

You can create multiple instances of the UAA service in your space.

About This Task

As a best practice, first delete any older unused instances before creating a new one.

Procedure

1. Sign into your Predix account at https://www.predix.io.
2. Navigate to **Catalog** > **Services**, then click the **User Account and Authentication** tile.
3. Click **Subscribe** on the required plan.
4. Complete the fields on the **New Service Instance** page.

Field	Description
Org	Select your organization.
Space	Select the space for your application.
Service instance name	Enter a unique name for this UAA service instance.
Service plan	Select a plan.
Admin client secret	Enter a client secret (this is the admin password for this UAA instance). The client secret can be any alphanumeric string. **Note:** Record the client secret in a secure place for later use.
Subdomain	(Optional) Enter a subdomain you might need to use in addition to the domain created for UAA. You must not add special characters in the name of the subdomain. The value of sub-domain is case-insensitive.

5. Click **Create Service**.

Results

Your UAA instance is created with the following specifications:

- A client identifier (`admin`).

  Note: An `admin` client is required for bootstrap purposes. You can create additional clients to use with your application.
- A client secret (that you specified while creating the service).

To retrieve additional details of your instance, you can bind an application to your instance.

Using the Command Line to Create a UAA Service Instance

Optional procedure for using the command line instead of the graphical user interface to create a UAA service instance.

About This Task

You can create up to 10 instances of UAA service in your space. If you need additional instances, you must delete an older unused instance and create a new one.

Procedure

1. Use the Cloud Foundry CLI to log into Cloud Foundry.

   ```
   cf login -a <API_Endpoint>
   ```

   Note: If you are a GE employee, you must use the `cf login --sso` command to log into Cloud Foundry. After you enter your SSO, you will receive a one-time passcode URL. Copy this URL and paste it in a browser to retrieve your one-time passcode. Use this code with the `cf` command to complete the CF login process.

   Depending on your Predix.io registration, the value of `<API_Endpoint>` is one of the following:

   - Predix US-West
     `https://api.system.aws-usw02-pr.ice.predix.io`
   - Predix Europe
     `https://api.system.aws-eu-central-1-pr.ice.predix.io`

   For example,

   ```
   cf login -a https://api.system.aws-usw02-pr.ice.predix.io
   ```

2. List the services in the Cloud Foundry marketplace by entering the following command.

   ```
   cf marketplace
   ```

   The UAA service, `predix-uaa`, is listed as one of the available services.

3. Create a UAA instance by entering the following command.

   ```
   cf create-service predix-uaa <plan> <my_uaa_instance> -c
   '{"adminClientSecret":"<my_secret>","subdomain":"<my_subdomain>"}'
   ```

   where:

   - `cf` stands for the CLI command, `cloud foundry`
   - `cs` stands for the CLI command `create-service`
   - `<plan>` is the plan associated with a service. For example, you can use the `tiered` plan for the `predix-uaa` service.
   - `-c` option is used to specify following additional parameters.
     - `adminClientSecret` specifies the client secret.
     - `subdomain` specifies a sub-domain you might need to use in addition to the domain created for UAA. This is an optional parameter. You must not add special characters in the name of the sub-domain. The value of sub-domain is case insensitive.

Note: Cloud Foundry CLI syntax can differ between Windows and Linux operating systems. See the Cloud Foundry help for the appropriate syntax for your operating system. For example, to see help for the `create service` command, run `cf cs`.

Results

Your UAA instance is created with the following specification:

- A client identifier (`admin`).

  Note: An `admin` client is created for bootstrap purposes. You can create additional clients to use with your application.
- A client secret (that you specified while creating the service).

To retrieve additional details of your instance, you can bind an application to your instance.

Example

Create a predix-uaa service instance with client secret as admin and sub-domain as ge-digital:

```
cf cs predix-uaa tiered test-1 -c
'{"adminClientSecret":"admin","subdomain":"ge-digital"}'
```

This is how it appears in VCAP SERVICES when using the `cf env <app_name>` command:

```
"VCAP_SERVICES": {
"predix-uaa": [
   {
    "credentials": {
     "dashboardUrl": "https://uaa-dashboard.run.asv-
pr.ice.predix.io/#/login/04187eb1-
e0cf-4874-8218-9fb77a8b4ed9",
     "issuerId": "https://04187eb1-
e0cf-4874-8218-9fb77a8b4ed9.predix-uaa.run.asv-
pr.ice.predix.io/oauth/token",
     "subdomain": "04187eb1-e0cf-4874-8218-9fb77a8b4ed9",
     "uri": "https://04187eb1-
e0cf-4874-8218-9fb77a8b4ed9.predix-uaa.run.asv-
pr.ice.predix.io",
     "zone": {
      "http-header-name": "X-Identity-Zone-Id",
      "http-header-value": "04187eb1-
e0cf-4874-8218-9fb77a8b4ed9"
     }
    },
    "label": "predix-uaa",
    "name": "testuaa",
    "plan": "Tiered",
    "provider": null,
    "syslog_drain_url": null,
    "tags": [],
    "volume_mounts": []
   }
  ],
```

Creating a Time Series Service Instance

Before You Begin

Complete the tasks in #unique_26.

Note: If you are registered on the Predix Azure domain, you must use the command-line instructions to create your service.

Procedure

1. Sign into your Predix account at https://www.predix.io.
2. Navigate to **Catalog** > **Data Management**, and click the **Time Series** tile.
3. Choose the plan, and click **Subscribe**.
4. On the **New Service Instance** page, enter:
5. (Optional) You can also use the Cloud Foundry CLI to create a Time Series service instance.

   Note: If you are a GE employee, you must use the `cf login --sso` command to log into Cloud Foundry. After you enter your SSO, you will receive a one-time passcode URL. Copy this URL and paste it in a browser to retrieve your one-time passcode. Use this code with the `cf` command to complete the CF login process.

   On Mac OS and Linux, use the following syntax:

   ```
   cf create-service predix-timeseries <plan> <my_time_series_instance>
   -c '{"trustedIssuerIds":["<uaa_instance1_host>/oauth/token",
   "<uaa_instance2_host>/oauth/token"]}'
   ```

   On Windows, use the following syntax:

   ```
   cf create-service predix-timeseries <plan> <my_time_series_instance>
   -c "{\"trustedIssuerIds\":[\"<uaa_instance1_host/oauth/token\",
   \"<uaa_instance2_host/oauth/token\"]}"
   ```

   where:

   - `<plan>` – The plan associated with a service.
   - <my_time_series_instance> – the service instance you are creating.
   - "trustedIssuerIds" – The issuer ID of your trusted issuer (UAA instance), such as `https://13fa0384-9e2a-48e2-9d06-2c95a1f4f5ea.predix-uaa.grc-apps.svc.ice.ge.com/oauth/token`. You can use a comma-separated list to specify multiple trusted issuers. You can retrieve this URL from the `VCAP_SERVICES` environment variable after you bind your UAA instance to an application.

Next Steps

#unique_17

Binding an Application to the Time Series Service Instance

About This Task

You must bind your application to the Time Series service instance to provision connection details and credentials for your Time Series service instance in the VCAP_SERVICES environment variable. Cloud Foundry runtime uses VCAP_SERVICES environment variables to communicate with a deployed application about its environment.

Procedure

1. Bind your Time Series application to your service instance:

   ```
   cf bind-service <application_name> <my_time_series_instance>
   ```

2. Restage your application to ensure the environment variable changes take effect:

   ```
   cf restage <application_name>
   ```

3. To view the environment variables for your application, enter the following command:

   ```
   cf env <application_name>
   ```

   The command shows the environment variables, which contain your basic authorization credentials, client ID, and the ingestion and query endpoint URIs.

   ```
   {
      "VCAP_SERVICES":{
         "predix-timeseries":[
            {
               "credentials":{
                  "ingest":{
                     "uri":"wss://<ingestion_url>",
                     "zone-http-header-name":"Predix-Zone-Id",
                     "zone-http-header-value":"<Predix-Zone-Id>",
                     "zone-token-scopes":[
                        "timeseries.zones.<Predix-Zone-Id>.user",
                        "timeseries.zones.<Predix-Zone-Id>.ingest"
                     ]
                  },
                  "query":{
                     "uri":"https://<query_url>",
                     "zone-http-header-name":"Predix-Zone-Id",
                     "zone-http-header-value":"<Predix-Zone-Id>",
                     "zone-token-scopes":[
                        "timeseries.zones.<Predix-Zone-Id>.user",
                        "timeseries.zones.<Predix-Zone-Id>.query"
                     ]
                  }
               },
               "label":"predix-timeseries",
               "name":"time-series-service-instance-predix-data-
   services",
               "plan":"beta",
               "tags":[
                  "timeseries",
                  "time-series",
   ```

```
                "time series"
            ]
        }
    ]
  }
}
```

Creating an OAuth2 Client

You can create OAuth2 clients with specific permissions for your application to work with Predix Platform services. Often this is the first step after creating an instance of a service.

About This Task

When you create an instance of UAA, the UAA Dashboard is available for configuring that instance of UAA. You can use the Client Management tab in the UAA Dashboard to create the OAuth2 clients.

If you are prefer using the UAA command-line interface (UAAC) instead of UAA Dashboard to create an OAuth2 client, see #unique_29

Procedure

1. In the Predix.io Console view, select the Space where your services are located.
2. In the Services Instances page, select the UAA instance to configure.
3. Select the **Configure Service Instance** option.
4. In the UAA Dashboard login page, specify your admin client secret and click **Login**.
5. In UAA Dashboard, select the **Client Management** tab.

   The Client Management tab has two views, **Clients** and **Services** . The **Services** view displays the service instances that you have created for your services.

   Note: The service instances displayed in the Services view were created while using the UAA that you are trying to configure. Service instances that you created using other UAA instances are not displayed on this page.
6. Click **Create Client** to open the **Create Client** form.
7. Complete the **Create Client** form.

Field	Description
Client ID	Specify a name for the OAuth2 client you are creating.
Authorized Grant Types	Choose one or more of the following grant types: • **authorization_code** When you use the authorization code grant type, the client directs the resource owner to UAA, which in turn directs the resource owner back to the client with the authorization code. • **client_credentials** When you use the client credentials grant type, the OAuth2 endpoint in UAA accepts the client ID and client secret and provides Access Tokens. • **password** When you use the resource owner password credentials grant type, the OAuth2 endpoint in UAA accepts the username and password and provides Access Tokens. • **refresh_token** The refresh tokens are credentials used to obtain access tokens. You can choose this option to obtain refresh token from UAA. You can then use the refresh token to obtain a new access token from UAA when the current access token becomes invalid or expires, or to obtain additional access tokens with identical or narrower scope. • **implicit** When you use the implicit grant type, UAA directly issues an Access Token to the client without authenticating the client. This reduces the number of round trips required to obtain an access token. For more information on grant types, see RFC 6749.
Client Secret	Specify the password. It is important that you keep a note of this password. If lost, this password cannot be retrieved.
Confirm Client Secret	Reenter the client secret.

Field	Description
Redirect URI	Specify a redirect URI to redirect the client after login or logout (for example, `http://example-app.com/callback`). Use this URI when you start using UAA as the service provider for your external Identity provider. UAA uses the value of Redirect URI for `/oauth/authorize` and `/logout` endpoints. You must specify a Redirect URI value if you use the Authorization Code or Implicit authorization grant type. When you use the Authorization Code grant type, the Redirect URI is your application's endpoint or callback that expects user authorization code. When you use the Implicit grant type, the Redirect URI is the end point where UAA sends the bearer token. Unique Resource Identifier consists of: • Access Protocol, `http` or `https` • Domain or IP address • Access Port such as 80 or 443 • Path If you have a specific URL for your application callback, you can use that to set the Redirect URI value for the related client. For example, `https://your-app-domain.run.aws-usw02-pr.ice.predix.io/path1/path2/callback`. You can specify multiple values for Redirect URI as a list of allowed destinations that UAA server can redirect the users. For example, `https://yourappdomain1.run.aws-usw02-pr.ice.predix.io/path1/path2/callback`, `https://yourappdomain2.run.aws-usw02-pr.ice.predix.io/path1/path2/callback`. If the subdomain of your application is dynamic, you can set the value of Redirect URI using wilcards. For example, `https://*.your-app-domain.run.aws-usw02-pr.ice.predix.io/path1/path2/callback`. **Note:** You must only use '*' for a domain that is exclusive to your application (Such as `your-app-domain` in example above). This prevents the redirect to be routed to an application that you do not own. You cannot use `*` in the top domain and sub domain (such as `predix.io` in the example above).
Scopes	Scopes are permissions associated with an OAuth Client to determine user access to a resource through an application. The user permissions are for authorization grant types `authorization_code`, `password` and `implicit`. By default, the admin client is assigned all required scopes. For a new client, an administrator can select the scopes to be added based on client requirements. For a list of available scopes, see Scopes Authorized by the UAA. To use an OAuth2 client for your Predix Platform service instance, you must update your OAuth2 client to add scopes that are specific to each service after adding the client to the service instance.

Field	Description
Authorities	Authorities are permissions associated with the OAuth Client when an application or API is acting on its own behalf to access a resource with its own credentials, without user involvement. The permissions are for the `client_credentials` authorization grant type. By default, the admin client is assigned all required authorities. For a new client, an administrator can select the authorities to be added based on client requirements. The list of authorities matches the list of scopes. For a list of available UAA scopes, see Scopes Authorized by the UAA. To use an OAuth2 client for your Predix Platform service instance, you must update your OAuth2 client to add authorities that are specific to each service after adding the client to the service instance. **Note:** An admin client is not assigned the default authority to change the user password. To change the user password, you must add the `uaa.admin` authority to your admin client.
Auto Approved Scopes	Specify scopes that can be approved automatically for the client without explicit approval from a resource owner.
Allowed Providers	Specifies the names of the external identity providers, if any. This field is required if you are using external identity providers with UAA as a service provider.
Access Token Validity	Specifies the access-token expiration time in ms.
Refresh Token Validity	Specifies the refresh-token expiration time in ms.

Next Steps

#unique_19 for your service specific information.

Updating the OAuth2 Client for Services

To use an OAuth2 client for secure access to your Predix Platform service instance from your application, you must update your OAuth2 client to add additional authorities or scopes that are specific to each service.

About This Task

To enable your application to access a platform service, your JSON Web Token (JWT) must contain the scopes required for a platform service. For example, some of the scope required for Access Control service are `acs.policies.read acs.policies.write`.

The OAuth2 client uses an authorization grant to request an access token. Based on the type of authorization grant that you have used, you must update your OAuth2 client to generate the required JWT. For more information on how the OAuth2 client is created, see Creating OAuth2 client.

If you use the UAA Dashboard to create additional clients, the client is created for the default `client_credentials` grant type. Some required authorities and scopes are automatically added to the client. You must add additional authorities or scopes that are specific to each service.

In addition, the admin client is not assigned the default authority to change the user password. To change the user password, you must add the `uaa.admin` authority to your admin client.

Use the following procedure to update the OAuth2 client.

Procedure

1. In the Console view, select the Space where your services are located.

2. In the Services Instances page, select the UAA instance to configure.
3. Select the **Configure Service Instance** option.
4. In the UAA Dashboard login page, specify your admin client secret and click **Login**.
5. In UAA Dashboard, select the **Client Management** tab.

   The Client Management tab has two views, **Clients** and **Services**. The **Services** view displays the service instances that you have created for your services.

   Note: The service instances displayed in the **Services** view are the instances that you created using the UAA that you are trying to configure. The service instances that you created using some other UAA instance are not displayed on this page.
6. Select the **Switch to Services View** option.
7. In the **Services** view, select the service that you need to update.
8. Choose an existing client or choose the **Create a new client** option. If you chose to create a new client, follow the steps in #unique_18.
9. Click **Submit**.
10. Click on the **Switch to Clients View** option.
11. In the **Clients** view, click the edit icon corresponding to the client added in the previous step.
12. Complete the **Edit Client** form.

Field	Description
Authorized Grant Types	Choose one or more of the following grant types: • **authorization_code** When you use the authorization code grant type, the client directs the resource owner to UAA, which in turn directs the resource owner back to the client with the authorization code. • **client_credentials** When you use the client credentials grant type, the OAuth2 endpoint in UAA accepts the client ID and client secret and provides Access Tokens. • **password** When you use the resource owner password credentials grant type, the OAuth2 endpoint in UAA accepts the username and password and provides Access Tokens. • **refresh_token** The refresh tokens are credentials used to obtain access tokens. You can choose this option to obtain refresh token from UAA. You can then use the refresh token to obtain a new access token from UAA when the current access token becomes invalid or expires, or to obtain additional access tokens with identical or narrower scope. • **implicit** When you use the implicit grant type, UAA directly issues an Access Token to the client without authenticating the client. This reduces the number of round trips required to obtain an access token. For more information on grant types, see RFC 6749.
Redirect URI	Specify a redirect URI to redirect the client after login (for example, `http://example-app.com/welcome`). This URI is used when you start using UAA as service provider for your external Identify provider.
Scopes	By default, the client is assigned a few required scopes. For a new client, an administrator can select the scopes to be added based on the selected grant type. If you select the `authorization_code`, `password` and `implicit` grant type, you must update the scopes with service specific scopes. For a complete list of required scopes, see #unique_30. For a list of available UAA scopes, see Scopes Authorized by the UAA.

Field	Description
Authorities	By default, the client is assigned a few required authorities. For a new client, an administrator can select the authorities to be added based on the selected grant type. If you select the `client_credentials` grant type, you must update the authorities with service specific authorities. For a complete list of scopes to be added for each service, see #unique_30. For a list of available UAA authorities, see Scopes Authorized by the UAA.
Auto Approved Scopes	Specify scopes that can be approved automatically for the client without explicit approval from the resource owner.
Allowed Providers	Specify the names of the external identity providers, if any. This field is required if you are using external identity providers with UAA as a service provider.
Access Token Validity	Specifies the access token expiration time in ms.
Refresh Token Validity	Specifies the refresh token expiration time in ms.

Next Steps

You can complete the following additional tasks in UAA Dashboard:

- If you are using authorization grant type as Authorization Code, Implicit, or Resource Owner Password, you can manage users in UAA.
- You can create password policies for user passwords.
- You can set up external identity provider or use UAA as an identity provider. See Managing Identity Providers.

If you have completed your OAuth2 client setup, you can bind your application to your service instance.

Authorities or Scopes Required for Time Series

When you create a new OAuth2 client, the client is assigned default scopes and authorities. You must add additional authorities or scopes that are specific to each service. To enable applications to access the Time Series service, your JSON Web Token (JWT) must contain the following Predix zone token scopes:

- For ingestion requests:
  - timeseries.zones.<Predix-Zone-Id>.user
  - timeseries.zones.<Predix-Zone-Id>.ingest
- For query requests:
  - timeseries.zones.<Predix-Zone-Id>.user
  - timeseries.zones.<Predix-Zone-Id>.query
- For delete requests:

  Note: The hard delete feature is currently available only in select environments. Contact Support to determine if this feature is enabled in your environment.

  - timeseries.zones.<Predix-Zone-Id>.user
  - timeseries.zones.<Predix-Zone-Id>.delete

The OAuth2 client uses an authorization grant to request an access token. OAuth2 defines four grant types. Based on the type of authorization grant that you use, you must update your OAuth2 client to generate the required JWT. For more information on how the OAuth2 client is created, see #unique_18.

Adding Zone Token Scopes to Applications

About This Task

The way you add zone token scopes depends on whether your application is a server-side or single-page web application.

Procedure

1. To add zone token scopes to a server-side application:
   a) Create two OAuth clients (one for ingestion and one for querying) on any of the trusted issuers that were provided when you created the service instance.
   b) Add the zone-token-scopes from the VCAP `credentials` for the respective clients to their authorities.

      The below example shows ingestion zone-token-scopes from VCAP credentials:

      ```
      "zone-token-scopes":[
                           "timeseries.zones.<Predix-Zone-Id>.user",
                           "timeseries.zones.<Predix-Zone-Id>.ingest"
                           "timeseries.zones.<Predix-Zone-Id>.delete"
                         ]
      ```

      The below example shows query zone-token-scopes from VCAP credentials:

      ```
      "zone-token-scopes":[
                           "timeseries.zones.<Predix-Zone-Id>.user",
                           "timeseries.zones.<Predix-Zone-Id>.query"
                           "timeseries.zones.<Predix-Zone-Id>.delete"
                         ]
      ```

2. To add zone token scopes to a single-page web application that uses client-side JavaScript to consume the time-series service instance:
   a) Create two user groups for both data ingestion and querying.
   b) Add the user who needs access to the application to the appropriate groups for data ingestion and queries.

      For example, for data ingestion, create two groups with the following names:

      ```
      "timeseries.zones.<Predix-Zone-Id>.user"
      "timeseries.zones.<Predix-Zone-Id>.ingest"
      ```

   c) For users who query data, create two groups with the following names:

      ```
      "timeseries.zones.<Predix-Zone-Id>.user"
      "timeseries.zones.<Predix-Zone-Id>.query"
      ```

   d) For users who delete data, create two groups with the following names:

      Note: The hard delete feature is currently available only in select environments. Contact Support to determine if this feature is enabled in your environment.

      ```
      "timeseries.zones.<Predix-Zone-Id>.user"
      "timeseries.zones.<Predix-Zone-Id>.delete"
      ```

3. You must also update your OAuth client scopes with the respective zone token scopes.

   See #unique_18 and #unique_32.
4. Push your time-series data for streaming ingestion using the WebSocket protocol.

   See #unique_6.

Using the Time Series Client Library

Using the Time Series Client Library

The Time Series service provides a REST-based API that you can access programatically using Java.

About This Task

In addition to a RESTful API that can be accessed using any language, the Time Series service provides a Java-based client library interface to the service. Use this library to help save time with configuring the WebSocket and HTTP connections, for example.

To use the Time Series client library, you should have a basic understanding of how to develop Java applications. You can use the Time Series client library to make calls to the Time Series API.

Note: This is a beta version of the Time Series client library. If you have questions or comments about the Time Series client library, post them on the Predix Forum at `http://www.predix.io/community/forum`.

This version of the Time Series client library supports filtering, as well as the following aggregators:

- Average
- Interpolation
- Scale

To use the client library, complete the tasks in #unique_26, then complete the following steps:

Procedure

1. Include the Time Series client in your application project. See #unique_13.
2. Download the Time Series library from https://artifactory.predix.io/artifactory/PREDIX-EXT/.

   Note: You must have a Predix account to access the download site.
3. Configure the Time Series library. See #unique_41.
4. Build the ingestion request. See #unique_42.
5. Build the query request. See #unique_43.

Including the Time Series Client in Your Project

The Time Series client must be included in your project.

Procedure

1. Update your Maven settings to use the Predix platform Artifactory by copying the following into the `<servers>` block of your `settings.xml` file, located in your `~/.m2` folder:

   ```
   <server>
       <id>artifactory.external</id>
       <username><predix-user-name></username>
       <password><predix-password></password>
   </server>
   ```

   where `<predix-user-name>` and `<predix-password>` are your Predix.io user name and password.

   Optionally, you can encrypt your password instead of using plain text. See Password Encryption.

2. Add the repository to the `POM` file located in your project's root directory:

```
<repositories>
      <repository>
         <id>artifactory.external</id>
         <name>GE external repository</name>
         <url>https://artifactory.predix.io/artifactory/PREDIX-EXT</
url>
      </repository>
</repositories>
```

3. Add the Time Series client as a dependency in your `POM` file:

```
<dependency>
    <groupId>com.ge.predix.timeseries</groupId>
    <artifactId>timeseries-client</artifactId>
    <version>0.0.2-SNAPSHOT</version>
</dependency>
```

Configuring the Time Series Client Library

About This Task

The application bound to your Time Series service contains environmental variables needed to configure the application with the Time Series client library using the `predix-timeseries.properties` file.

Procedure

1. Provide the configuration information to use the Time Series client library.
   a) Bind your application to your Time Series service instance (#unique_17).
   b) To retrieve the values needed to configure the Time Series client library, enter the following command at the Cloud Foundry CLI:

      ```
      cf env <application_name>
      ```

      The environment variables for your application (including the ingestion and query URLs) are returned to you. Make note of the environment variables.
   c) Create a file called `predix-timeseries.properties` and add the configuration information retrieved in the previous step.

      Note: The `predix-timeseries.properties` file is required to use the Time Series client library. However, if you are using the `createTenantContextFromProvidedProperties` method (shown in Step 2), the URIs, client id, and secrets properties in the "Execution properties" section are not used.

      ```
      # Predix Timeseries configuration. >>>>>
      # DO NOT MODIFY WITHOUT CONSULTING PREDIX SUPPORT
      predix.timeseries.maxTagsPerQuery=5
      predix.timeseries.maxIngestionMessageSize=512000
      # Predix Timeseries configuration. <<<<<

      #Purchased Plan restrictions. >>>>>
      # MODIFY TO SUIT YOUR PLAN. GOING OVER MIGHT RESULT IN ADDED
      CHARGES
      plan.ingestion.concurrent.connections.max=100
      plan.query.concurrent.connections.max=100
      #Purchased Plan restrictions. <<<<<
      ```

```
#Execution properties. >>>>>
# MODIFY AS APPROPRIATE

#Execution properties. <<<<<

uaa.uri=<your uaa uri. (without the /oauth/token)>
ingestion.uri=<ingestion uri obtained from the binding to Predix
time-series>
ingestion.zone-http-header-name=Predix-Zone-Id
ingestion.client.id=<the client id that has scope access to
'timeseries.zones.<your zone id>.ingest'>
ingestion.client.secret.env.variable=<environment variable that
contains the ingestion client secret>
query.uri=<query uri obtained from the binding to Predix time-
series>
query.zone-http-header-name=Predix-Zone-Id
query.client.id=<client id that has scope access to
'timeseries.zones.<your zone id>.query'>
query.client.secret.env.variable=<environment variable that
contains the query client secret>
```

Note: The client secret is an environment variable that you set on your local machine, which contains a password that you set. Add the name of the environment variable you created as the client secret. If you have two different client IDs (one for ingestion, one for querying), create two different environment variables with two different passwords.

d) Save the `predix-timeseries.properties` file and add it to your project.

e) To provide the path to the `predix-timeseries.properties` file, add the following line to your code:

```
TenantContext
tenant=TenantContextFactory.createTenantContextFromPropertiesFile<p
ropertiesPath>;
```

2. If you already have the Time Series URLs and authorization tokens available in your code, you can directly provide the configuration information programatically.

   To use both the ingestion and query client, add the following code:

```
TenantContext
tenant=TenantContextFactory.createTenantContextFromProvidedProperties
<queryUri, queryAccessToken, ingestionUri, ingestionAccessToken,
zoneHeaderName, zoneId>
```

   To use the ingestion client only, add the following code:

```
TenantContext tenant =
TenantContextFactory.createIngestionTenantContextFromProvidedProperti
es(ingestionUrl, authToken, predixZoneIdHeaderName,
predixZoneIdHeaderValue);
```

   To use the query client only, add the following code:

```
TenantContext tenant =
TenantContextFactory.createQueryTenantContextFromProvidedProperties(q
ueryUrl, authToken, predixZoneIdHeaderName, predixZoneIdHeaderValue);
```

Creating an Ingestion Request Using the Time Series Client Library

Learn how to create a time series data ingestion request using the example in this topic.

About This Task

The client library offers an Ingestion Request Builder that interfaces with the Time Series service and handles the WebSocket connection. The library uses a builder, which allows flexibility in adding whatever values you would like to the Ingestion Request.

The following shows an example data ingestion request using one good, one bad, and one uncertain data point. These values are derived from an active data source, such as a sensor.

```
Integer sensorValueAsInt = (int) Math.random();
Double sensorValueAsDouble = Math.random();
IngestionRequestBuilder ingestionBuilder =
IngestionRequestBuilder.createIngestionRequest()
        .withMessageId("<MessageID>")
        .addIngestionTag(IngestionTag.Builder.createIngestionTag()
                .withTagName("TagName")
                .addDataPoints(
                        Arrays.asList(
                                new DataPoint(new Date().getTime(),
sensorValueAsInt, Quality.GOOD),
                                new DataPoint(new Date().getTime(),
sensorValueAsDouble, Quality.NOT_APPLICABLE),
                                new DataPoint(new Date().getTime(),
"Bad Value", Quality.BAD),
                                new DataPoint(new Date().getTime(),
null, Quality.UNCERTAIN)
                        )
                )
                .addAttribute("AttributeKey", "AttributeValue")
                .addAttribute("AttributeKey2", "AttributeValue2")
                .build());

String json = ingestionBuilder.build().get(0);
IngestionResponse response =
ClientFactory.ingestionClientForTenant(ingestionTenant).ingest(json);
String responseStr = response.getMessageId() +
response.getStatusCode();
```

Note: The Time Series service now accepts compressed (GZIP) JSON payloads. The size limit for the actual JSON payload is 512 KB regardless of the ingestion request format. For compressed payloads, this means that the decompressed payload cannot exceed 512 KB.

Creating a Query Using the Time Series Client Library

The client library also offers a Query Builder that interfaces with the Time Series service and return a QueryResponse object.

About This Task

The Query Response can then be accessed either programmatically, or converted to a JSON string. The library again uses a builder, which allows flexibility in adding whatever values and filters are needed in the Query Request.

Procedure

Create a data query request.

The following shows an example data query request using numerous filters.

```
QueryBuilder builder = QueryBuilder.createQuery()
        .withStartAbs(1427463525000L)
        .withEndAbs(1427483525000L)
        .addTags(
                QueryTag.Builder.createQueryTag()
                        .withTagNames(Arrays.asList("ALT_SENSOR",
"TEMP_SENSOR"))
                        .withLimit(1000)
                        .addAggregation(Aggregation.Builder.averageWith
Interval(1, TimeUnit.HOURS))
                        .addFilters(FilterBuilder.getInstance()
                                .addAttributeFilter("host",
Arrays.asList("<host>")).build())
                        .addFilters(FilterBuilder.getInstance()
                                .addAttributeFilter("type",
Arrays.asList("<type>")).build())
                        .addFilters(FilterBuilder.getInstance()
                                .addMeasurementFilter(FilterBuilder.Con
dition.GREATER_THAN_OR_EQUALS, Arrays.asList("23.1")).build())
                        .addFilters(FilterBuilder.getInstance()
                                .withQualitiesFilter(Arrays.asList(Qual
ity.BAD, Quality.GOOD)).build())
                        .build());
QueryResponse response =
ClientFactory.queryClientForTenant(tenant).queryAll(builder.build());
```

Using the Time Series Service

Time Series Data Model

A time series uses tags, which are often used to represent sensors (for example, a temperature sensor).

Data Ingestion

Time series data sets are based on the idea of tags, which represent different types of values measured by sensors (for example, temperature or pressure). A data record consists of one Tag Name (Sensor type), and multiple datapoints, each comprised of a Timestamp (Time) and a Measurement (Value). As additional options, each datapoint can also contain the quality of its data, and the overall record can contain multiple attributes (key/value pairs) describing the asset.

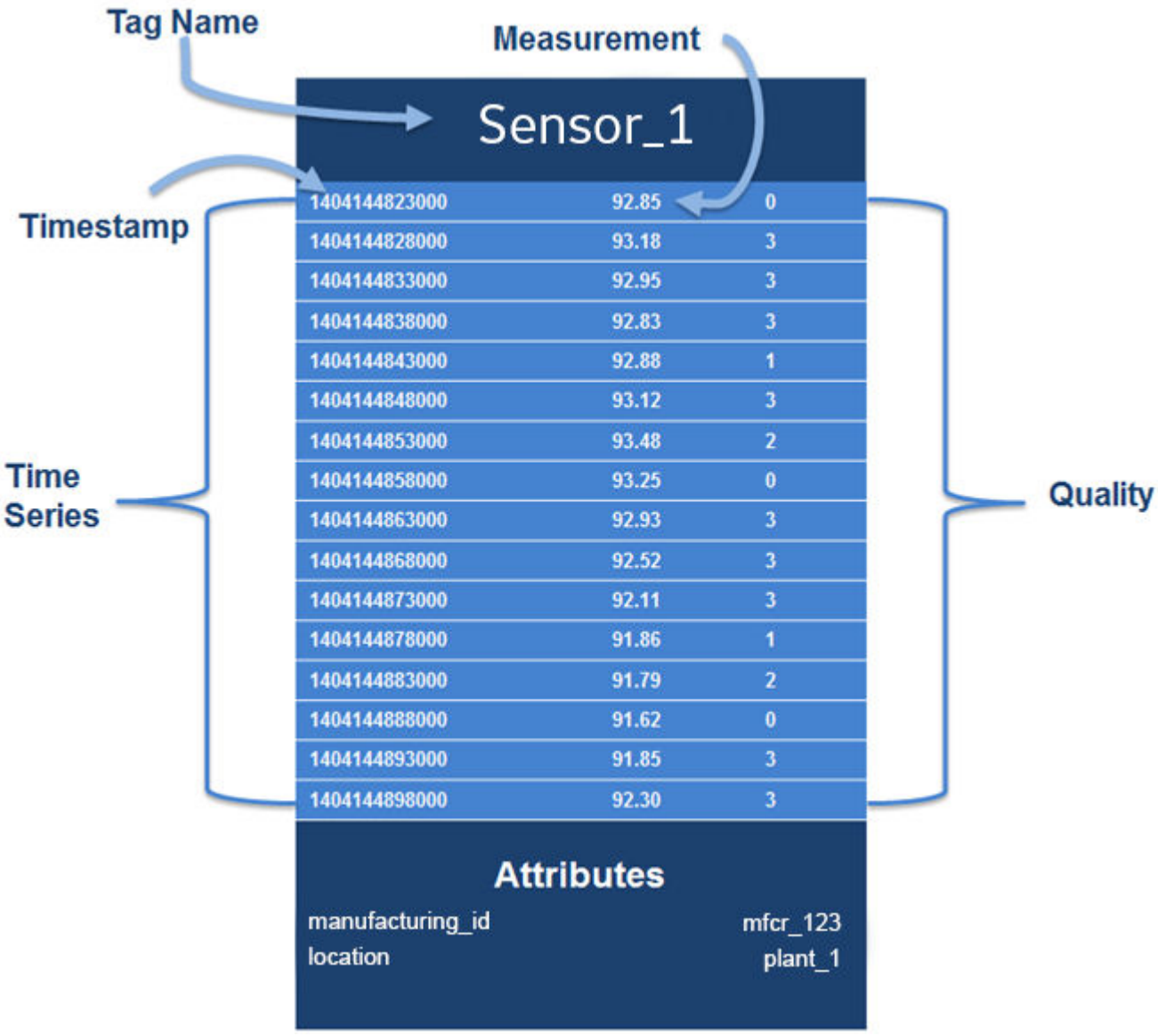

Figure 2: Time Series Data Model

Table 1: Time Series Data Model Structure

Term	Definition
Tag name	(Required) Name of the tag (for example, "Temperature"). This does not need to be a unique value, and typically numerous sensors representing the same tag type will share this same tag name. Do not use the tag name to store attributes; you should instead store attributes like the sensor ID in the Attributes section. The tag name can contain alphanumeric characters, periods (.), slashes (/), dashes (-), underscores (_) and spaces and is limited to a maximum 256 characters.
Timestamp	(Required) The date and time in UNIX epoch time, with millisecond precision.
Measurement	(Required) The value produced from your sensor. See *Data Types* for more information on what is accepted. Do not include the unit of measurement in the field, for example, `98'C`).
Qualities	(Optional) Quality of the data, represented by the values 0, 1, 2, 3. See *Data Quality* below for an explanation of data quality values.
Attribute	(Optional) Attributes are key/value pairs used to store data associated with a tag (for example: "ManufacturerGE":"AircraftID230"). This is useful for filtering data. **Note:** Attributes allow you to store additional relevant details about that specific data point—they should not be used to store large amounts of data. Do not use null values.
BackFill	(Optional) BackFill is a boolean flag that can be used to indicate whether this ingestion request includes backFill or non-live data. Data ingested with this option enabled will not be available to query immediately, but will be available eventually. Example: backFill: true **Note:** "backFill" field is a boolean and setting it to any other data type would return 400 (Bad Request).

Data Quality

You can tag data with values that represent data quality, which enables you to query by quality in cases where you want to retrieve or process only data of a specific level of quality. Using data quality can help to ensure that the data you get back in the query result is the most relevant for your purpose.

Value	Description
0	Bad quality.
1	Uncertain quality.
2	Not applicable.
3	Good quality. If you do not specify quality, the default is good (3).

Data Types

The Time Series ingestion service accepts mixed data types.

The following table shows sample input and output for mixed variable types (text, number, and null). When designing your data model:

- Do not use attributes as tag names. Keep the tag name as the type of data produced by the sensor (for example, "Temperature" or "Pressure"), and use attributes to distinguish between distinct sensors.
- Do not use timestamps as measurement values in your application.

Ingestion JSON Input	Ingestion JSON Input Type	Query Output	Datapoint Type
27	long	27	number
"34"	string (max 256 characters)	34	number
3.345	float	3.345	number
"1.123"	string (max 256 characters)	1.123	number
"1.0E-2"	string (256 characters)	0.01	number
"abc"	string (max 256 characters)	"abc"	text
"true"	string (max 256 characters)	"true"	text
true / false	boolean	"true" / "false"	text
"null"	string (max 256 characters)	"null"	text
null	null	null	Null
""	string (max 256 characters)	""	null
" "	string (max 256 characters)	" "	null

Data Consumption

The Time Series service provides REST APIs for querying and aggregating time series data. The API allows you to:

- Query time series data specifying tags (sensors, etc) and a time window.
- Filter by attribute values.
- Retrieve all tags and attribute keys from the time series database.
- Add aggregates and interpolate data points in a given time window.
- Query the time series database for the latest values.

  Note: When you use the GET latest API, the query returns the latest data point.
- Delete tags and all the associated data points.

For more information about these APIs, see the API Documentation.

Ingesting Time Series Data

The Time Series service ingests data using the WebSocket protocol. You may use the Java client library to facilitate ingestion and to avoid setting up a WebSocket, or create your own Websocket connection if you prefer to use a language other than Java.

Ingest with the Java Client Library

After setting up your client, you may create Ingestion Requests using the Ingestion Request Builder. The following shows an example data ingestion request using the Java client. Library.

```
IngestionRequestBuilder ingestionBuilder =
IngestionRequestBuilder.createIngestionRequest()
        .withMessageId("<MessageID>")
        .addIngestionTag(IngestionTag.Builder.createIngestionTag()
            .withTagName("TagName")
            .addDataPoints(
                Arrays.asList(
                                new DataPoint(new
Date().getTime(),Math.random(), Quality.GOOD),
                                new DataPoint(new Date().getTime(),
"BadValue", Quality.BAD),
                                new DataPoint(new Date().getTime(),
null,Quality.UNCERTAIN))
                            )
            .addAttribute("AttributeKey", "AttributeValue")
            .addAttribute("AttributeKey2", "AttributeValue2")
            .build());

String json = ingestionBuilder.build().get(0);
IngestionResponse response =
ClientFactory.ingestionClientForTenant(tenant).ingest(json);
String responseStr = response.getMessageId() +
response.getStatusCode();
```

Ingest with your own WebSocket Connection

You may also setup your own WebSocket in whatever language you prefer. It is recommended you find a WebSocket library in the language you are using to facilitate this process. Once you have a WebSocket library connected, point the request to the endpoint `wss://<ingestion_url>`. You must also provide the following headers:

- Authorization: Bearer <token from UAA>
- Predix-Zone-Id: <your zone id>
- Origin: `http://<your IP address or “localhost”>`

An example of a WebSocket implementation for Time Series ingestion is available at `https://github.com/PredixDev/timeseries-bootstrap/blob/master/src/main/java/com/ge/predix/solsvc/timeseries/bootstrap/client/TimeseriesClientImpl.java`

Note: The ingestion URI and `<Predix-Zone-Id>` are included with the environment variables for your application when you bind your application to your Time Series service instance. To view the environment variables, on a command line, enter:

```
cf env <application_name>
```

Example Data Ingestion Request

The following shows an example of the JSON payload for an ingestion request:

```
URL: wss://ingestion_url
Headers:
```

```
   Authorization: Bearer <token from trusted issuer>
   Predix-Zone-Id: <Predix-Zone-Id>
   Origin: http://<origin-hostname>/


Request Payload:
{
   "messageId": "<MessageID>",
   "body":[
      {
         "name":"<TagName>",
         "datapoints":[
            [
               <EpochInMs>,
               <Measure>,
               <Quality>
            ]
         ],
         "attributes":{
            "<AttributeKey>":"<AttributeValue>",
            "<AttributeKey2>":"<AttributeValue2>"
         }
      }
   ]
}
```

Ingesting backFill data

When ingesting historical or non-live data, it is recommended to use the backFill option to avoid overloading the live data stream. Data ingested using this option will not be available to query immediately.

Note: This feature is not yet supported in the Java client.

The following JSON is an example of how to use the backFill option.

```
URL: wss://ingestion_url
Headers:
   Authorization: Bearer <token from trusted issuer>
   Predix-Zone-Id: <Predix-Zone-Id>
   Origin: http://<origin-hostname>/


Request Payload:
{
   "messageId": "<MessageID>",
   "body":[
      {
         "name":"<TagName>",
         "datapoints":[
            [
               <EpochInMs>,
               <Measure>,
               <Quality>
            ]
         ],
         "attributes":{
            "<AttributeKey>":"<AttributeValue>",
```

```
            "<AttributeKey2>":"<AttributeValue2>"
        }
      }
   ],
   "backFill": true
}
```

Ingestion Nuances

Consider the following when creating an ingestion request:

- In previous releases, quality was supported as an attribute, but starting with this release,you must explicitly provide quality in each datapoint, along with the timestamp and measurement
- The Time Series service now accepts compressed (GZIP) JSON payloads. The size limit for the actual JSON payload is 512 KB regardless of the ingestion request format. For compressed payloads, this means that the decompressed payload cannot exceed 512 KB.
- The <Message Id> can be a string or integer, and must be unique. When using an integer the <*MessageID*> should be between 0 and 2^{64} (18446744073709551616).
- The `<BackFill>` must be a boolean. The ingestion request will return a 400 (Bad Request) status code if the `<BackFill>` is sent as any other data type. This is an optional parameter, and its value will be **false** if not specified.

Acknowledgement Message

An acknowledgement message is sent back to the client for each message packet. The following example shows an acknowledgement message:

```
{
        "messageId": <MessageID>,
        "statusCode": <AcknowledgementStatusCode>
}
```

The acknowledgement status codes are:

Code	Message
202	Accepted successfully
400	Bad request
401	Unauthorized
413	Request entity too large **Note:** The payload cannot exceed512KB.
503	Failed to ingest data

Tips for Data Ingestion

Spread ingestion requests over multiple connections, as sending numerous payloads over the same connection increases wait time for data availability over the query service. Also, be aware of our data plan when using multiple connections.

Pushing Time Series Data

Use the command-line interface as a simple way to interact with the Time Series service gateway. The time series gateway uses the WebSocket protocol for streaming ingestion. The ingestion endpoint format is: `wss://ingestion_url`.

The Web Socket protocol is used rather than HTTP because it can ingest a higher volume of data, which increases performance.

You can see an example of a WebSocket implementation for time series ingestion at https://github.com/predixdev/timeseries-bootstrap.

Note: The ingestion URI and `<Predix-Zone-Id>` are included with the environment variables for your application when you bind your application to your time series service instance. To view the environment variables, on a command line, enter:

```
cf env <application_name>
```

For all ingestion requests, use the token you received from UAA in the `Authorization:` section of the HTTP Header in the form of `Bearer <token from trusted issuer>`.

Note: In previous releases, quality was supported as an attribute, but starting with this release, you must explicitly provide quality in each datapoint, along with the timestamp and measurement.

Example Data Ingestion Request

The following shows an example of the JSON payload for an ingestion request:

```
URL: wss://ingestion_url
Headers:
   Authorization: Bearer <token from trusted issuer>
   Predix-Zone-Id: <Predix-Zone-Id>
   Origin: http://<origin-hostname>/


Request Payload:
{
   "messageId": "<MessageID>",
   "body":[
      {
         "name":"<TagName>",
         "datapoints":[
            [
               <EpochInMs>,
               <Measure>,
               <Quality>
            ]
         ],
         "attributes":{
            "<AttributeKey>":"<AttributeValue>",
            "<AttributeKey2>":"<AttributeValue2>"
         }
      }
   ]
}
```

The following shows an example of the JSON payload for an ingestion request to send it in backFill mode.

```
URL: wss://ingestion_url
Headers:
   Authorization: Bearer <token from trusted issuer>
   Predix-Zone-Id: <Predix-Zone-Id>
   Origin: http://<origin-hostname>/


{
   "messageId": "<MessageID>",
   "body":[
      {
         "name":"<TagName>",
         "datapoints":[
            [
               <EpochInMs>,
               <Measure>,
               <Quality>
            ]
         ],
         "attributes":{
            "<AttributeKey>":"<AttributeValue>",
            "<AttributeKey2>":"<AttributeValue2>"
         }
      }
   ],
   "backFill": true
}
```

The following shows an example data ingestion request if you are using a Java client.

```
IngestionRequestBuilder ingestionBuilder =
IngestionRequestBuilder.createIngestionRequest()
        .withMessageId("<MessageID>")
        .addIngestionTag(IngestionTag.Builder.createIngestionTag()
                .withTagName("TagName")
                .addDataPoints(
                        Arrays.asList(
                                new DataPoint(new Date().getTime(),
Math.random(), Quality.GOOD),
                                new DataPoint(new Date().getTime(),
"Bad Value", Quality.BAD),
                                new DataPoint(new Date().getTime(),
null, Quality.UNCERTAIN)
                        )
                )
                .addAttribute("AttributeKey", "AttributeValue")
                .addAttribute("AttributeKey2", "AttributeValue2")
                .build());

String json = ingestionBuilder.build().get(0);
IngestionResponse response =
ClientFactory.ingestionClientForTenant(tenant).ingest(json);
String responseStr = response.getMessageId() +
response.getStatusCode();
```

Note: The `<MessageID>` can be a string or integer, and must be unique. When using an integer the <*MessageID*> should be between 0 and 2^{64} (18446744073709551616).

Note: `<BackFill>` must be a boolean value. A 400 (Bad Request) status code will be returned if `<BackFill>` is set as any other data type.

See #unique_53 for more information about the structure of a time series tag.

Acknowledgement Message

An acknowledgement message is sent back to the client for each message packet. The following example shows an acknowledgement message:

```
{
    "messageId": <MessageID>,
    "statusCode": <AcknowledgementStatusCode>
}
```

Acknowledgement status codes include:

Code	Message
202	Accepted successfully
400	Bad request
401	Unauthorized
413	Request entity too large **Note:** The payload cannot exceed 512KB.
503	Failed to ingest data

Related Information

Troubleshoot Time Series Queries on page 95

Data Types

The Time Series ingestion service accepts mixed data types.

The following table shows sample input and output for mixed variable types (text, number, and null).

Ingestion JSON Input	Ingestion JSON Input Type	Query Output	Datapoint Type
27	long	27	number
“34”	string (max 256 characters)	34	number
3.345	float	3.345	number
“1.123”	string (max 256 characters)	1.123	number
“1.0E-2”	string (256 characters)	0.01	number
“abc”	string (max 256 characters)	“abc”	text
“true”	string (max 256 characters)	“true”	text
true / false	boolean	“true” / “false”	text

“null”	string (max 256 characters)	“null”	text
null	null	null	null
""	string (max 256 characters)	""	null
" "	string (max 256 characters)	" "	null

Querying Time Series Data

Before You Begin

Before you begin, perform all of the tasks in *Setting Up and Configuring the Time Series Service*.

About This Task

You can create asynchronous or synchronous queries using either the Java Client Library or via HTTP requests to the REST API. The input JSON for an asynchronous query request is the same as a synchronous query request. Both the asynchronous and synchronous query APIs allow you to do the follow: The API allows you to:

- Query time series data specifying by tags (sensors, etc) and a time window.
- Filter by attribute values.
- Add aggregates and interpolate data points in a given time window.
- Query the time series database for the latest values.

Only the synchronous API allows you to retrieve all tags and attribute keys from the time series database.

Procedure

1. View the environment variables for your application by entering the following on a command line:

   ```
   cf env <application_name>
   ```

   The query URI, trusted issuer token, and `<Predix-Zone-Id>` are included with the environment variables for your application. You will use this information in the HTTP header of your query.
2. Create your query.

   The header for your query must be in the following format:

   ```
   URL: https://query_url/v1/datapoints
   Method: POST
   Headers:
       Authorization: Bearer <token from trusted issuer>
       Predix-Zone-Id: <Predix-Zone-Id>
   ```

   Note: You can use both the `GET` and `POST` methods to query time series data. See #unique_57 for more information about how to form a query request.
3. Create your query. Choose either synchronous or asynchronous query method.

   a) **Synchronous query request.**

   The header for your synchronous query must be in the following format:

   ```
   URL: https://query_url/v1/datapoints
   Method: POST
   Headers:
   ```

```
    Authorization: Bearer <token from trusted issuer>
    Predix-Zone-Id: <Predix-Zone-Id>
```

b) **Asynchronous query request.**

Note: The asynchronous queries feature is currently available only in select environments. Contact Support to determine if this feature is enabled in your environment

The header for your asynchronous query must be in the following format:

```
URL: http://query_url/ /v1/datapoints/async
Method: POST
Headers:
    Authorization: Bearer <token from trusted issuer>
    Predix-Zone-Id: <Predix-Zone-Id>
```

Note: You can use both the `GET` and `POST` methods to query time series data. See #unique_57 for more information about how to form a query request.

Tips for Time Series Data Queries

Attributes

- Attributes should not contain high cardinality values for either the attribute key or value.
- Do not store counters as attributes, as it degrades query performance.

GET vs POST Methods

- Use the GET method for less complex queries. Responses can be cached in proxy servers and browsers.
- Use the POST method as a convenience method when the query JSON is very long, and you do not need the request to be cached.

Note: The GET latest API returns the latest data point.

When using the POST API, the following rules apply:

- Filters are the only supported operation, and they are optional.
- Specifying a start time is optional, however, if you specify a start time, you must also specify an end time.
- If you do not define a time window, the query retrieves the latest data points from the current time (now).

Performance

- Multiple queries with fewer tags in each query leads to better performance than fewer queries that include more tags.

Data Points Limit

Data points in the query result cannot exceed the maximum limit of 500,000.

Time Series Query Optimization

To optimize query response times and service performance, keep the following recommendations and best practices in mind.

- When deciding which tags to use for your data, keep in mind that tags are the primary means to ingest and query data in Time Series. Because Time Series data is indexed by tags, queries based on tags run

much faster than attribute-based queries. For optimum performance, it is best to use tags to track data that you need to frequently query.

For example, to query data from California, New York, and Texas locations separately, you might create three queries. If you define an `asset1` tag with a `state` attribute and query by attribute, run time can be slow. To optimize query performance, you can instead define a distinct tag for each query, for example, `asset1_CA`, `asset1_NY`, and `asset1_TX`.

- Be aware that querying without using filters returns results faster. To minimize the time cost of query processing, it is best to avoid filtering to the greatest extent possible. For example, to find the latest data point, avoid using filters in your query.
- To obtain the best performance, query by tags rather than by attributes. Tags are indexed, which speeds response time.
- For fastest processing, specify the shortest time intervals that are useful to retrieve the data you need. For example, to find only one data point, set a one-month time window. Six queries with one-month time windows will run faster than a single query with a six-month time window.

Query Properties and Examples

Query Properties

Your query must specify a start field, which can be either absolute or relative. The end field is optional, and can be either absolute or relative. And you do not specify the `end` time, the query uses the current date and time.

- Absolute start time – Expressed as an integer timestamp value in milliseconds.
- Relative start time – Calculated relative to the current time. For example, if the start time is 15 minutes, the query returns all matching data points for the last 15 minutes. The relative start time is expressed as a string with the format `<value><time-unit>-ago`, where:
  - `<value>` must be an integer greater than zero (0).
  - `<time-unit>` – must be represented as follows:

Time Unit	Represents
ms	milliseconds
s	seconds
mi	minutes
h	hours
d	days
w	weeks
mm	months
y	years

Example General Query

You can use the time series APIs to list tags and attributes, as well as to query data points. You can query data points using start time, end time, tag names, time range, measurement, and attributes.

Note: The input JSON for an asynchronous query request is the same format as a synchronous query request. The header format is different, depending up which query API (asynchronous or synchronous) you specify in the URL. For more information, see #unique_61.

The following example shows a synchronous JSON data query with header using the POST method, with a limit of 1000 set for the number of data points to return in the query result:

```
URL: https://query_url/v1/datapoints
Method: POST
Headers:
    Authorization: Bearer <token from trusted issuer>
    Predix-Zone-Id: <Predix-Zone-Id>
Payload:
{
    "start": 1427463525000,
    "end": 1427483525000,
    "tags": [
       {
            "name": [
                "ALT_SENSOR",
                "TEMP_SENSOR"
            ],
            "limit": 1000,
            "aggregations": [
              {
                "type": "avg",
                "interval": "1h"
              }
            ],
            "filters": {
                "attributes": {
                    "host": "<host>",
                    "type": "<type>"
                },
                "measurements": {
                    "condition": "ge",
                    "values": "23.1"
                },
                "qualities": {
                    "values": [
                        "0",
                        "3"
                    ]
                }
            },
            "groups": [
                {
                    "name": "attribute",
                    "attributes": [
                        "attributename1",
                        "attributename2"
                    ]
                }
            ]
        }
    ]
}
```

The following example shows an asynchronous JSON data query with header using the POST method, with a limit of 1000 set for the number of data points to return in the query result. The same payload is passed, note the difference in the header URL.

```
URL: https://query_url/v1/datapoints/async
Method: POST
Headers:
    Authorization: Bearer <token from trusted issuer>
    Predix-Zone-Id: <Predix-Zone-Id>
Payload:
{
    "start": 1427463525000,
    "end": 1427483525000,
    "tags": [
       {
            "name": [
                "ALT_SENSOR",
                "TEMP_SENSOR"
            ],
            "limit": 1000,
            "aggregations": [
              {
                "type": "avg",
                "interval": "1h"
              }
            ],
            "filters": {
                "attributes": {
                    "host": "<host>",
                    "type": "<type>"
                },
                "measurements": {
                    "condition": "ge",
                    "values": "23.1"
                },
                "qualities": {
                    "values": [
                        "0",
                        "3"
                    ]
                }
            },
            "groups": [
                {
                    "name": "attribute",
                    "attributes": [
                        "attributename1",
                        "attributename2"
                    ]
                }
            ]
        }
    ]
}
```

The following is an example asynchronous query response.

```
{
    "statusCode": 202,
"correlationId": "00000166-36cc-b44b-5d85-db5918aeed67",
```

```
"expirationTime" : 1533666448748
}
```

The following shows an example of a query request if you are using the Java time series client library:

```
QueryBuilder builder = QueryBuilder.createQuery()
        .withStartAbs(1427463525000L)
        .withEndAbs(1427483525000L)
        .addTags(
                QueryTag.Builder.createQueryTag()
                        .withTagNames(Arrays.asList("ALT_SENSOR",
"TEMP_SENSOR"))
                        .withLimit(1000)
                        .addAggregation(Aggregation.Builder.averageWith
Interval(1, TimeUnit.HOURS))
                        .addFilters(FilterBuilder.getInstance()
                                .addAttributeFilter("host",
Arrays.asList("<host>")).build())
                        .addFilters(FilterBuilder.getInstance()
                                .addAttributeFilter("type",
Arrays.asList("<type>")).build())
                        .addFilters(FilterBuilder.getInstance()
                                .addMeasurementFilter(FilterBuilder.Con
dition.GREATER_THAN_OR_EQUALS, Arrays.asList("23.1")).build())
                        .addFilters(FilterBuilder.getInstance()
                                .withQualitiesFilter(Arrays.asList(Qual
ity.BAD, Quality.GOOD)).build())
                        .build());
QueryResponse response =
ClientFactory.queryClientForTenant(tenant).queryAll(builder.build());
```

Use the token you receive from the trusted issuer in the HTTP Header '`Authorization`' for all query requests in the form of '`Bearer <token from trusted issuer>`'.

Note:

You can use both the GET and POST methods to query time series data. Use the query API with the GET HTTP method by passing the query JSON as a URL query parameter. The GET method allows requests and responses to be cached in proxy servers and browsers.

In cases where the query JSON is very long and exceeds query parameter length (for some browsers), use POST as an alternate method to query the time series API.

POST requests have no restrictions on data length. However, requests are never cached. The Time Series service API design supports complex, nested time series queries for multiple tags and their attributes. For example, more complex queries can include aggregations and filters.

For more information about this API, see the API Documentation.

Note: Tag names, measurement, and attributes are case-sensitive and can contain only alphanumeric characters, periods (.), slashes (/), dashes (-), and underscores (_).

Requesting Asynchronous Query Status

You can request the status of an asynchronous query only, this type of request is not supported for synchronous queries. The asynchronous status query request must be in the following format.

```
URL: http://query_uri/v1/datapoints/query/status
    Method: GET/POST
    Headers:
```

```
        Authorization: Bearer <token from trusted issuer>
        Predix-Zone-Id: <Predix-Zone-Id>
                 Parameters:
        correlationID: <correlationID>
        jobStatusOnly: True
```

The `jobStatusOnly` field is not required. The `jobStatusOnly` parameter will default to True if not provided.

The following is an example of an asynchronous query response.

```
HTTP Code: 200

{
    "request":{
      "correlationID":"aaa-aaa-aaa",
      "tenantID":"zzz-zzz-zzz",
      "operationType":"query"
   },
   "lastUpdated":12345678,
   "created":12345678,
   "status":"Completed",
   "statusText":"Successfully completed",
   "message":{
   }
}
```

Requesting Asynchronous Query Results

The request for asynchronous query results must be in the following format.

```
URL: http://query_uri/v1/datapoints/query/status
    Method: GET/POST
    Headers:
        Authorization: Bearer <token from trusted issuer>
        Predix-Zone-Id: <Predix-Zone-Id>
                 Parameters:
        correlationID: <correlationID>
        jobStatusOnly: False
```

The `jobStatusOnly` field is not required. The `jobStatusOnly` parameter will default to True if not provided. This field indicates if the status query response should contain the time series data (`False`) or only the requested job status (`True`).

The following example shows asynchronous query results.

```
HTTP Code: 200

{
    "request":{
      "correlationID":"aaa-aaa-aaa",
      "tenantID":"zzz-zzz-zzz",
      "operationType":"query"
   },
   "lastUpdated":12345678,
   "created":12345678,
   "statusText":"finished”,
   "message” : “”,
   "queryResults":
```

```
      "{
         "tags": [{
            "name": "test.async",
            "results": [{
               "groups": [{
                  "name": "type",
                  "type": "number"
               }],
               "attributes": {},
               "values": [
                  [1435776300000, 28.875, 3]
               ]
            }],
            "stats": {
               "rawCount": 4
            }
         }]
      }"
}
```

Valid response for `statusText` can be one of the following.

- created
- paused
- processing
- finished
- not found
- expired
- failed

If the query completes successfully, the `queryResults` field contains the JSON response from the query.

Using Relative Start and End Times in a Query

This example shows a JSON query using a relative start time, with a value of "15" and a unit of "minutes", and an absolute end time in milliseconds.

```
{
    "start": "15mi-ago",
    "end": 1427483525000,
    "tags": [
        {
            "name": "ALT_SENSOR",
            "limit": 1000
       }
   ]
}
```

This example shows a query using a relative start time using the time series client library:

```
QueryBuilder builder = QueryBuilder.createQuery()
        .withStartRelative(15, TimeUnit.MINUTES, TimeRelation.AGO)
        .withEndAbs(1427483525000L)
        .addTags(
                QueryTag.Builder.createQueryTag()
                        .withTagNames(Arrays.asList("ALT_SENSOR"))
                        .withLimit(1000)
```

```
                        .build());

QueryResponse queryResponse =
ClientFactory.queryClientForTenant(tenant).queryAll(builder.build());
```

This example shows a JSON query using the absolute start time in milliseconds, and a relative end time with a value of "5" and a unit of "minutes".

```
{
    "start": 1427463525000,
    "end": "5mi-ago",
    "tags": [
        {
            "name": "ALT_SENSOR",
            "limit": 1000
       }
   ]
}
```

This example shows a query using an absolute start time and relative end time using the time series client library:

```
QueryBuilder builder = QueryBuilder.createQuery()
        .withStartAbs(1427463525000L)
        .withEndRelative(5, TimeUnit.MINUTES, TimeRelation.AGO)
        .addTags(
                QueryTag.Builder.createQueryTag()
                        .withTagNames(Arrays.asList("ALT_SENSOR"))
                        .withLimit(1000)
                        .build());

QueryResponse queryResponse =
ClientFactory.queryClientForTenant(tenant).queryAll(builder.build());
```

Data Points Limit

The following limits apply to data queries:

- Data points in the query result cannot exceed the maximum limit of 500,000.
  Narrow your query criteria (for example, time window) to return a fewer number of data points.

Limiting the Data Points Returned by a Query

When you create a query, you can specify a limit for the number of data points returned in the query results. This example shows a JSON query with the `"limit"` property set to 1000.

```
{
    "start": 1427463525000,
    "end": 1427483525000,
    "tags": [
        {
            "name": "ALT_SENSOR",
            "limit": 1000
       }
   ]
}
```

This example shows the same "limit" query using the time series client library.

```
QueryBuilder builder = QueryBuilder.createQuery()
        .withStartAbs(1427463525000L)
        .withEndAbs(1427483525000L)
        .addTags(
                QueryTag.Builder.createQueryTag()
                        .withTagNames(Arrays.asList("ALT_SENSOR"))
                        .withLimit(1000)
                        .build());

QueryResponse queryResponse =
ClientFactory.queryClientForTenant(tenant).queryAll(builder.build());
```

Specifying the Order of Data Points Returned by a Query

The default order for query results is ascending, and it is ordered by timestamp. You can use the `order` property in your query request to specify the order in which you want the data points returned. This example returns the data points in descending order:

```
{
    "start": 1427463525000,
    "end": 1427483525000,
    "tags": [
        {
            "name": "ALT_SENSOR",
            "order": "desc"
       }
   ]
}
```

This example shows the same "order" query using the time series client library:

```
QueryBuilder builder = QueryBuilder.createQuery()
        .withStartAbs(1427463525000L)
        .withEndAbs(1427483525000L)
        .addTags(
                QueryTag.Builder.createQueryTag()
                        .withTagNames(Arrays.asList("ALT_SENSOR"))
                        .withOrder(QueryTag.Order.DESCENDING)
                        .build());

QueryResponse queryResponse =
ClientFactory.queryClientForTenant(tenant).queryAll(builder.build());
```

Related Information

#unique_53

Query for Latest Data Point Example

The Time Series API provides both a `GET` and `POST` method for retrieving the most recent data point within the defined time window. By default, unless you specify a start time, the query goes back 15 days. When using the POST API, the following rules apply:

- Filters are the only supported operation, and they are optional.
- Specifying a start time is optional. However, if you do specify the start time, you must also specify an end time.

- If you do not define a time window, the query retrieves the latest data points from the current time (now).

The following shows an example of a JSON query for the latest data point with no time window defined:

```
{
    "tags": [
        {
            "name": "ALT_SENSOR",
            "filters": {
                "measurements": {
                    "condition": "le",
                    "value": 10
                }
            }
        }
    ]
}
```

This example shows the same query using the time series client library:

```
QueryBuilder builder = QueryBuilder.createQuery()
        .addTags(
                QueryTag.Builder.createQueryTag()
                        .withTagNames(
                                Arrays.asList("ALT_SENSOR")
                        )
                        .addFilters(FilterBuilder.getInstance()
                            .addMeasurementFilter(FilterBuilder.Conditi
on.LESS_THAN_OR_EQUALS, Arrays.asList("10"))
                                .build())
                        .build());
QueryResponse queryResponse =
ClientFactory.queryClientForTenant(tenant).queryForLatest(builder.buil
d());
```

Query for Data Points Relative to a Time Stamp Examples

You can query for data points relative to a given time stamp by passing the parameter "direction" and specifying whether you want results *after* ("forward") or *before* ("backward") that point. Use the "limit" filter to specify the number of data points to return.

Query for Data Points After a Given Time Stamp

This example shows how you can query for data points for results after ("forward") a given time stamp. This example shows a JSON ingestion data query.

```
[{
    "name": "test.forward.integer",
    "datapoints": [
        [1435776300000, 2, 3],
        [1435776400000, 10, 3],
        [1435776500000, 1, 3],
        [1435776550000, 2, 3],
        [1435776600000, 6, 3],
        [1435776700000, 2, 3],
        [1435776800000, 13, 3],
        [1435776900000, 3, 0]
```

```
    ]
}]
```

This example shows a query using the parameter "direction" with the value "forward" with a given "start" time. The "limit" filter specifies "3" data points are to be returned.

```
{
    "start": 1435776550000,
    "direction": "forward",
        "tags": [{
        "name": "test.forward.integer",
            "limit": 3
    }]
}
```

This example shows the results returned.

```
{
    "tags": [{
        "name": "test.forward.integer",
        "results": [{
            "values": [
                            [1435776550000, 2, 3],
                    [1435776600000, 6, 3],
                    [1435776700000, 2, 3]
            ],
                        "groups": [{
                "name": "type",
                "type": "number"
            }],
            "attributes": {}
        }],
        "stats": {
            "rawCount": 8
        }
    }]
}
```

Query for Data Points Before a Given Time Stamp

This example shows how you can query for data points for results before ("backward") a given time stamp. This example shows a JSON ingestion data query.

```
[{
    "name": "test.backward.integer",
    "datapoints": [
        [1435776300000, 2, 3],
        [1435776400000, 10, 3],
        [1435776500000, 1, 3],
        [1435776550000, 2, 3],
        [1435776600000, 6, 3],
        [1435776700000, 2, 3],
        [1435776800000, 13, 3],
        [1435776900000, 3, 0]
    ]
}]
```

This example shows a query using the parameter "direction" with the value "backward" with a given "start" time. The "limit" filter specifies "3" data points are to be returned.

```
{
    "start": 1435776550000,
    "direction": "backward",
        "tags": [{
        "name": "test.backward.integer",
            "limit": 3
    }]
}
```

This example shows the results returned.

```
{
    "tags": [{
        "name": "test.backward.integer",
        "results": [{
            "values": [
                            [1435776400000, 10, 3],
                    [1435776500000, 1, 3],
                    [1435776550000, 2, 3]
            ],
                        "groups": [{
                "name": "type",
                "type": "number"
            }],
            "attributes": {}
        }],
        "stats": {
            "rawCount": 8
        }
    }]
}
```

Data Aggregators

Aggregation functions allow you to perform mathematical operations on a number of data points to create a reduced number of data points. Aggregation functions each perform distinct mathematical operations and are performed on all the data points included in the sampling period.

Note: Data points that are string or null types within the time range are ignored in data aggregations and not returned in the response.

You can use the `<query url>/v1/aggregations` endpoint to get a list of the available aggregations.

For more information about these APIs, see the API Documentation.

Aggregations are processed in the order specified in the query. The output of an aggregation is passed to the input of the next until they are all processed.

Aggregator Parameters

The `unit` and `sampling` aggregators are common to more than one aggregator.

Unit is a string and must be one of the following:

- MILLISECONDS

- SECONDS
- MINUTES
- HOURS
- DAYS
- WEEKS

The `sampling` aggregator is a JSON object that contains a `value (integer)` and a `unit`.

Range Aggregator

You can set the following parameters on any range aggregator:

- `sampling` – JSON object that contains a unit and value.
- `align_start_time` (boolean) – Set to true if you want the time for the aggregated data point for each range to fall on the start of the range instead of being the value for the first data point within that range.
- `align_sampling` (boolean) – Set to true if you want the aggregation range to be aligned based on the sampling size and you want the data to take the same shape, when graphed, as you refresh the data.
- `start_time` (long) – Start time from which to calculate the ranges (start of the query).
- `time_zone` (long time zone format) – Time zone to use for time-based calculations.

Aggregators

Name	Type	Parameters	Description
`avg`	Average	Extends the `range` aggregator.	Returns the average of the values.
`count`	Count	Extends the `range` aggregator.	Counts the number of data points.
`dev`	Standard Deviation	Extends the `range` aggregator.	Returns the standard deviation of the time series.
`diff`	Difference	Extends the `range` aggregator.	Calculates the difference between successive data points.
`div`	Divide	The `divisor` (double) parameter is required. This is the value by which to divide data points.	Returns each data point divided by the specified value of the `divisor` property.
`gaps`	Gaps	Extends the `range` aggregator.	Marks gaps in data with a null data point, according to the sampling rate.
`interpolate`	Interpolate	Extends the `range` aggregator.	Does linear interpolation for the chosen window.
`least_squares`	Least Squares	Extends the `range` aggregator.	Returns two points that represent the best fit line through the set of data points for the range.

Name	Type	Parameters	Description
max	Maximum	Extends the range aggregator.	Inherits from the range aggregator. Returns the most recent largest value.
min	Minimum	Extends the range aggregator.	Returns the most recent smallest value.
percentile	Percentile	Extends the range aggregator. percentile - Percentile to count.	Calculates a probability distribution, and returns the specified percentile for the distribution. The “percentile” value is defined as 0 < percentile <= 1 where .5 is 50% and 1 is 100%.
rate	Rate	• sampling – Sets the sampling for calculating the rate. • unit – Required. Set the sampling duration as MILLISECONDS, SECONDS, MINUTES, HOURS, DAYS, or WEEKS. For example, if you set the unit to HOUR, the sampling rate is over one hour. • time_zone – Must use the long format, for example:	Returns the rate of change between a pair of data points.
sampler	Sampler	Optional parameters include: • unit – Set the sampling duration as MILLISECONDS, SECONDS, MINUTES, HOURS, DAYS, or WEEKS. For example, if you set the unit to HOUR, the sampling rate is over one hour. • time_zone – Must use the long format, for example:	Calculates the sampling rate of change for the data points.
scale	Scale	factor (double) – Scaling value.	Scales each data point by a factor.
sum	Sum	Extends the range aggregator.	Returns the sum of all values.
trendmode	Trend Mode	• sampling – Sets the sampling for calculating the min and max within each interval.	Returns the min and max values within each interval. **Note:** Trend Mode converts all longs to double. This causes a loss of precision for very large long values.

Aggregation Examples

Data Interpolation

You can interpolate data between raw data samples when data is missing and query for results based on a time interval (for example, every one hour). The following shows a sample request for interpolating data for every hour for the given time window:

```
{
   "start":1432667547000,
   "end":1432667552000,
   "tags":[
      {
         "name":"phosphate.level",
         "aggregations":[
            {
               "type":"interpolate",
               "interval": "1h"
            }
         ]
      }
   ]
}
```

Querying to Have a Specific Number of Data Points Returned

This shows an example of querying data using interpolation mode and specifying the time window and number of data points to return:

```
{
   "start":1432667547000,
   "end":1432667552000,
   "tags":[
      {
         "name":"phosphate.level",
         "aggregations":[
            {
               "type":"interpolate",
               "sampling":{
                  "datapoints":500
               }
            }
         ]
      }
   ]
}
```

Querying for the Average Value

This shows an example of querying for the average value with a 30-second interval on which to aggregate the data:

```
{
    "tags": [{
        "name": "Compressor-2015",
```

```
        "aggregations": [{
            "type": "avg",
            "sampling": {
                "unit": "s",
                "value": "30"
            }
        }]
    }],
    "start": 1452112200000,
    "end": 1453458896222
}
```

Querying for Values Between Boundaries

This shows an example of querying data using interpolation mode between "start", "end", "interval", and "extension" (24 hours or less/in milliseconds) parameter values. Predix Time Series will interpolate between start - extension and end + extension and return values between start + interval to end, inclusive.

Example Ingestion JSON

```
[{
    "name": "test.interpolation.extension.integer",
    "datapoints": [
        [1200, 3.0, 3],
        [2500, 4.0, 3],
        [3200, 2.0, 3],
    ]
}]
```

Example Query

```
{
    "start": 1000,
    "end": 3000,
    "tags": [{
        "name": "test.interpolation.extension.integer",
        "aggregations": [{
            "type": "interpolate",
                        "interval": "1000ms",
                        "extension": "1000"
                }]
    }]
}
```

Example Result

```
{
  "tags": [
    {
      "name": "test.interpolation.extension.integer",
      "results": [
        {
          "groups": [
            {
              "name": "type",
              "type": "number"
            }
```

```
          ],
          "values": [
            [2000, 3.6153846153846154, 3],
            [3000, 2.571428571428571, 3],
          ],
          "attributes": {}
        }
      ],
      "stats": {
        "rawCount": 3
      }
    }
  ]
}
```

Average With Mixed Data Types

The Average aggregator returns the average of the values.

Example Ingestion JSON

```
[{
    "name": "test.average.mixedType",
    "datapoints": [
        [1435776300000, 2, 1],
        [1435776400000, null],
        [1435776500000, 10.5, 3],
        [1435776550000, "100", 2],
        [1435776600000, "string"],
        [1435776700000, "string36"],
        [1435776800000, true],
        [1435776900000, 3, 0]
    ]
}]
```

Example Query

```
{
    "start": 1435766300000,
    "end": 1435777000000,
    "tags": [{
        "name": "test.average.mixedType",
        "aggregations": [{
            "type": "avg",
            "sampling": {
                "datapoints": 1
            }
        }]
    }]
}
```

The following shows an example of the average aggregator using the time series client library:

```
QueryBuilder builder = QueryBuilder.createQuery()
        .withStartAbs(1432671129000L)
        .withEndAbs(1432672230500L)
```

```
        .addTags(
                QueryTag.Builder.createQueryTag()
                        .withTagNames(
                                Arrays.asList("ALT_SENSOR")
                        )
                        .addAggregation(Aggregation.Builder.averageWith
Interval(3600, TimeUnit.SECONDS))
                        .build());
QueryResponse queryResponse =
ClientFactory.queryClientForTenant(tenant).queryForLatest(builder.buil
d());
```

Example Result

```
{
    "tags": [{
        "name": "test.average.mixedType",
        "results": [{
            "groups": [{
                "name": "type",
                "type": "number"
            }],
            "attributes": {},
            "values": [
                [1435776300000, 28.875, 3]
            ]
        }],
        "stats": {
            "rawCount": 4
        }
    }]
}
```

Average With Double Data Types

The Average aggregator returns the average of the values.

Example Ingestion JSON

```
[{
    "name": "test.average.doubleType",
    "datapoints": [
        [1435766300000, 2.0],
        [1435766400000, 2.5],
        [1435766500000, 3.0],
        [1435766550000, 3.5],
        [1435766600000, 4.0],
        [1435766700000, 50.0],
        [1435766800000, 100.0],
        [1435766900000, 150.5],
        [1435777000000, 175.0],
        [1435777100000, 175.5],
        [1435777200000, 200.0],
        [1435777300000, 225.0],
        [1435777400000, 225.5],
        [1435777500000, 250],
        [1435777600000, 300.0]
```

```
    ]
}]
```

Example Query

```
{
    "start": 1435766300000,
    "end": 1435787600000,
    "tags": [{
        "name": "test.average.doubleType",
        "aggregations": [{
            "type": "avg",
            "sampling": {
                "datapoints": 1
            }
        }]
    }]
}
```

Example Result

```
{
    "tags": [{
        "name": "test.average.doubleType",
        "results": [{
            "groups": [{
                "name": "type",
                "type": "number"
            }],
            "attributes": {},
            "values": [
                [1435766300000, 124.43333333333334, 3]
            ]
        }],
        "stats": {
            "rawCount": 15
        }
    }]
}
```

Count With Mixed Data Types

The Count aggregator returns the most recent largest value.

Example 1

Example Ingestion JSON

```
[{
    "name": "test.count.mixedType",
    "datapoints": [
        [1435776300000, 2, 1],
        [1435776400000, null],
        [1435776500000, 10.5, 3],
        [1435776550000, "100", 2],
        [1435776600000, "string"],
        [1435776700000, "string36"],
```

```
        [1435776800000, true],
        [1435776900000, 3, 0]
    ]
}]
```

Example Query

```
{
    "start": 1435766300000,
    "end": 1435777000000,
    "tags": [{
        "name": "test.count.mixedType",
        "filters": {
            "qualities": {
                "values": ["3"]
            }
        },
        "aggregations": [{
            "type": "trendmode",
            "sampling": {
                "datapoints": 1
            }
        }, {
            "type": "count",
            "sampling": {
                "datapoints": 1
            }
        }]
    }]
}
```

Example Result

```
{
    "tags": [{
        "name": "test.count.mixedType",
        "results": [{
            "groups": [{
                "name": "type",
                "type": "number"
            }],
            "filters": {
                "qualities": {
                    "values": ["3"]
                }
            },
            "attributes": {},
            "values": [
                [1435776500000, 1, 3]
            ]
        }],
        "stats": {
            "rawCount": 1
        }
    }]
}
```

Example 2

Example Ingestion JSON

```
[{
    "name": "test.count.mixedType",
    "datapoints": [
        [1435776300000, 2, 1],
        [1435776400000, null],
        [1435776500000, 10.5, 3],
        [1435776550000, "100", 2],
        [1435776600000, "string"],
        [1435776700000, "string36"],
        [1435776800000, true],
        [1435776900000, 3, 0]
    ]
}]
```

Example Query

```
{
    "start": 1435766300000,
    "end": 1435777000000,
    "tags": [{
        "name": "test.count.mixedType",
        "aggregations": [{
            "type": "trendmode",
            "sampling": {
                "datapoints": 1
            }
        }, {
            "type": "count",
            "sampling": {
                "datapoints": 1
            }
        }]
    }]
}
```

Example Result

```
{
    "tags": [{
        "name": "test.count.mixedType",
        "results": [{
            "groups": [{
                "name": "type",
                "type": "number"
            }],
            "attributes": {},
            "values": [
                [1435776300000, 2, 3]
            ]
        }],
        "stats": {
            "rawCount": 4
        }
    }]
}
```

Example 3

Example Ingestion JSON

```
[{
    "name": "test.count.mixedType",
    "datapoints": [
        [1435776300000, 2, 1],
        [1435776400000, null],
        [1435776500000, 10.5, 3],
        [1435776550000, "100", 2],
        [1435776600000, "string"],
        [1435776700000, "string36"],
        [1435776800000, true],
        [1435776900000, 3, 0]
    ]
}]
```

Example Query

```
{
    "start": 1435766300000,
    "end": 1435777000000,
    "tags": [{
        "name": "test.count.mixedType",
        "aggregations": [{
            "type": "count",
            "sampling": {
                "datapoints": 1
            }
        }]
    }]
}
```

Example Result

```
{
    "tags": [{
        "name": "test.count.mixedType",
        "results": [{
            "groups": [{
                "name": "type",
                "type": "number"
            }],
            "attributes": {},
            "values": [
                [1435776300000, 4, 3]
            ]
        }],
        "stats": {
            "rawCount": 4
        }
    }]
}
```

Count With Double Data Types

The Count aggregator returns the most recent largest value.

Example Ingestion JSON

```
[{
    "name": "test.count.mode.doubleType",
    "datapoints": [
        [1435766300000, 2.0],
        [1435766400000, 2.5],
        [1435766500000, 3.0],
        [1435766550000, 3.5, 2],
        [1435766600000, 4.0],
        [1435766700000, 50.0],
        [1435766800000, 100.0],
        [1435766900000, 150.5],
        [1435777000000, 175.0],
        [1435777100000, 175.5],
        [1435777200000, 200.0],
        [1435777300000, 225.0],
        [1435777400000, 225.5],
        [1435777500000, 250],
        [1435777600000, 300.0]
    ]
}]
```

Example Query

```
{
    "start": 1435766300000,
    "end": 1435787600000,
    "tags": [{
        "name": "test.count.mode.doubleType",
        "aggregations": [{
            "type": "count",
            "sampling": {
                "datapoints": 1
            }
        }]
    }]
}
```

Example Result

```
{
    "tags": [{
        "name": "test.count.mode.doubleType",
        "results": [{
            "groups": [{
                "name": "type",
                "type": "number"
            }],
            "attributes": {},
            "values": [
                [1435766300000, 15, 3]
            ]
        }],
        "stats": {
            "rawCount": 15
        }
    }]
}
```

Difference With Mixed Data Types

The Difference aggregator calculates the difference between successive data points.

Example Ingestion JSON

```
[{
    "name": "test.diff.mixedType",
    "datapoints": [
        [1435776300000, 2, 1],
        [1435776400000, null],
        [1435776500000, 10.5, 3],
        [1435776550000, "100", 2],
        [1435776600000, "string"],
        [1435776700000, "string36"],
        [1435776800000, true],
        [1435776900000, 3, 0]
    ]
}]
```

Example Query

```
{
    "start": 1435766300000,
    "end": 1435777000000,
    "tags": [{
        "name": "test.diff.mixedType",
        "aggregations": [{
            "type": "diff"
        }]
    }]
}
```

Example Result

```
{
    "tags": [{
        "name": "test.diff.mixedType",
        "results": [{
            "groups": [{
                "name": "type",
                "type": "number"
            }],
            "attributes": {},
            "values": [
                [1435776500000, 8.5, 3],
                [1435776550000, 89.5, 3],
                [1435776900000, -97, 3]
            ]
        }],
        "stats": {
            "rawCount": 4
        }
    }]
}
```

Divide With Mixed Data Types

The Divide aggregator returns each data point divided by the specified value of the `divisor` property.

Example Ingestion JSON

```
[{
    "name": "test.div.mixedType",
    "datapoints": [
        [1435776300000, 2, 1],
        [1435776400000, null],
        [1435776500000, 10.5, 3],
        [1435776550000, "100", 2],
        [1435776600000, "string"],
        [1435776700000, "string36"],
        [1435776800000, true],
        [1435776900000, 3, 0]
    ]
}]
```

Example Query

```
{
    "start": 1435766300000,
    "end": 1435777000000,
    "tags": [{
        "name": "test.div.mixedType",
        "aggregations": [{
            "type": "div",
            "divisor": 3
        }]
    }]
}
```

Example Result

```
{
    "tags": [{
        "name": "test.div.mixedType",
        "results": [{
            "groups": [{
                "name": "type",
                "type": "number"
            }],
            "attributes": {},
            "values": [
                [1435776300000, 0.6666666666666666, 3],
                [1435776500000, 3.5, 3],
                [1435776550000, 33.333333333333336, 3],
                [1435776900000, 1, 3]
            ]
        }],
        "stats": {
            "rawCount": 4
        }
```

```
    }]
}
```

Divide With Double Data Types

The Divide aggregator returns each data point divided by the specified value of the `divisor` property.

Example Ingestion JSON

```
[{
    "name": "test.div.doubleType",
    "datapoints": [
        [1435766300000, 2.0],
        [1435766400000, 2.5],
        [1435766500000, 3.0],
        [1435766550000, 3.5],
        [1435766600000, 4.0],
        [1435766700000, 50.0],
        [1435766800000, 100.0],
        [1435766900000, 150.5],
        [1435777000000, 175.0],
        [1435777100000, 175.5],
        [1435777200000, 200.0],
        [1435777300000, 225.0],
        [1435777400000, 225.5],
        [1435777500000, 250],
        [1435777600000, 300.0]
    ]
}]
```

Example Query

```
{
    "start": 1435766300000,
    "end": 1435787600000,
    "tags": [{
        "name": "test.div.doubleType",
        "aggregations": [{
            "type": "div",
            "divisor": 3
        }]
    }]
}
```

Example Result

```
{
    "tags": [{
        "name": "test.div.doubleType",
        "results": [{
            "groups": [{
                "name": "type",
                "type": "number"
            }],
            "attributes": {},
            "values": [
```

```
                [1435766300000, 0.6666666666666666, 3],
                [1435766400000, 0.8333333333333334, 3],
                [1435766500000, 1, 3],
                [1435766550000, 1.1666666666666667, 3],
                [1435766600000, 1.3333333333333333, 3],
                [1435766700000, 16.666666666666668, 3],
                [1435766800000, 33.333333333333336, 3],
                [1435766900000, 50.166666666666664, 3],
                [1435777000000, 58.333333333333336, 3],
                [1435777100000, 58.5, 3],
                [1435777200000, 66.66666666666667, 3],
                [1435777300000, 75, 3],
                [1435777400000, 75.16666666666667, 3],
                [1435777500000, 83.33333333333333, 3],
                [1435777600000, 100, 3]
            ]
        }],
        "stats": {
            "rawCount": 15
        }
    }]
}
```

Gaps With Mixed Data Types

The Gaps aggregator marks gaps in data with a null data point, according to the sampling rate.

Example Ingestion JSON

```
[{
    "name": "test.gaps.mixedType",
    "datapoints": [
        [1435776300000, 2, 1],
        [1435776400000, null],
        [1435776500000, 10.5, 3],
        [1435776550000, "100", 2],
        [1435776600000, "string"],
        [1435776700000, "string36"],
        [1435776800000, true],
        [1435776900000, 3, 0]
    ]
}]
```

Example Query

```
{
    "start": 1435766300000,
    "end": 1435777000000,
    "tags": [{
        "name": "test.gaps.mixedType",
        "aggregations": [{
            "type": "gaps",
            "interval": "100000ms"
        }]
    }]
}
```

Example Result

```
{
    "tags": [{
        "name": "test.gaps.mixedType",
        "results": [{
            "groups": [{
                "name": "type",
                "type": "number"
            }],
            "attributes": {},
            "values": [
                [1435776300000, 2, 1],
                [1435776400000, null, 3],
                [1435776500000, 10.5, 3],
                [1435776550000, 100, 2],
                [1435776600000, null, 3],
                [1435776700000, null, 3],
                [1435776800000, null, 3],
                [1435776900000, 3, 0]
            ]
        }],
        "stats": {
            "rawCount": 4
        }
    }]
}
```

Gaps With Double Data Types

The Gaps aggregator marks gaps in data with a null data point, according to the sampling rate.

Example Ingestion JSON

```
[{
    "name": "test.gaps.doubleType",
    "datapoints": [
        [1435766300000, 2.0],
        [1435766400000, 2.5],
        [1435766500000, 3.0],
        [1435766550000, 3.5],
        [1435766600000, 4.0],
        [1435766700000, 50.0],
        [1435766800000, 100.0],
        [1435766900000, 150.5],
        [1435777000000, 175.0],
        [1435777100000, 175.5],
        [1435777200000, 200.0],
        [1435777300000, 225.0],
        [1435777400000, 225.5],
        [1435777500000, 250],
        [1435777600000, 300.0]
    ]
}]
```

Example Query

```
{
    "start": 1435766300000,
    "end": 1435777000000,
    "tags": [{
        "name": "test.gaps.doubleType",
        "aggregations": [{
            "type": "gaps",
            "interval": "500000ms"
        }]
    }]
}
```

Example Result

```
{
  "tags": [
    {
      "name": "test.gaps.doubleType",
      "results": [
        {
          "groups": [
            {
              "name": "type",
               "type": "number"
            }
          ],
          "values": [
            [1435766300000, 2, 3],
            [1435766400000, 2.5, 3],
            [1435766500000, 3, 3],
            [1435766550000, 3.5, 3],
            [1435766600000, 4, 3],
            [1435766700000, 50, 3],
            [1435766800000, 100, 3],
            [1435766900000, 150.5, 3],
            [1435767300000, null, 3],
            [1435767800000, null, 3],
            [1435768300000, null, 3],
            [1435768800000, null, 3],
            [1435769300000, null, 3],
            [1435769800000, null, 3],
            [1435770300000, null, 3],
            [1435770800000, null, 3],
            [1435771300000, null, 3],
            [1435771800000, null, 3],
            [1435772300000, null, 3],
            [1435772800000, null, 3],
            [1435773300000, null, 3],
            [1435773800000, null, 3],
            [1435774300000, null, 3],
            [1435774800000, null, 3],
            [1435775300000, null, 3],
            [1435775800000, null, 3],
            [1435776300000, null, 3],
            [1435777000000, 175, 3]
          ],
          "attributes": {}
        }
```

```
      ],
      "stats": {
        "rawCount": 9
      }
    }
  ]
```

Interpolation With Mixed Data Types

Example Ingestion JSON

```
[{
    "name": "test.iterpolation.mixedType",
    "datapoints": [
        [1435776300000, 2, 1],
        [1435776400000, null],
        [1435776500000, 10.5, 3],
        [1435776550000, "100", 2],
        [1435776600000, "string"],
        [1435776700000, "string36"],
        [1435776800000, true],
        [1435776900000, 3, 0]
    ]
}]
```

Example Query

```
{
    "start": 1435776000000,
    "end": 1435777000000,
    "tags": [{
        "name": "test.iterpolation.mixedType",
        "aggregations": [{
            "type": "interpolate",
            "interval": "100000ms"
        }]
    }]
}
```

The following shows an example of a query using the interpolation aggregator with interval using the time series client library:

```
QueryBuilder builder = QueryBuilder.createQuery()
        .withStartAbs(1432671129000L)
        .withEndAbs(1432672230500L)
        .addTags(
                QueryTag.Builder.createQueryTag()
                        .withTagNames(
                                Arrays.asList("ALT_SENSOR")
                        )
                        .addAggregation(Aggregation.Builder.interpolate
WithInterval(3600, TimeUnit.SECONDS))
                        .build());
QueryResponse queryResponse =
ClientFactory.queryClientForTenant(tenant).queryForLatest(builder.buil
d());
```

The following shows an example of a query using the interpolation aggregator with data point count, using the time series client library:

```
QueryBuilder builder = QueryBuilder.createQuery()
        .withStartAbs(1432671129000L)
        .withEndAbs(1432672230500L)
        .addTags(
                QueryTag.Builder.createQueryTag()
                        .withTagNames(
                                Arrays.asList("ALT_SENSOR")
                        )
                        .addAggregation(Aggregation.Builder.interpolate
ForDatapointCount(100))
                        .build());
QueryResponse queryResponse =
ClientFactory.queryClientForTenant(tenant).queryForLatest(builder.buil
d());
```

Example Result

```
{
  "tags": [
    {
      "name": "test.iterpolation.mixedType",
      "results": [
        {
          "groups": [
            {
              "name": "type",
              "type": "number"
            }
          ],
          "values": [
            [1435776100000, 0, 0],
            [1435776200000, 0, 0],
            [1435776300000, 2, 1],
            [1435776400000, 2, 1],
            [1435776500000, 10, 3],
            [1435776600000, 100, 2],
            [1435776700000, 100, 2],
            [1435776800000, 100, 2],
            [1435776900000, 0, 0],
            [1435777000000, 0, 0]
          ],
          "attributes": {}
        }
      ],
      "stats": {
        "rawCount": 4
      }
    }
  ]
}
```

Interpolation With Double Data Types

Example Ingestion JSON

```
[{
    "name": "test.interpolation.doubleType",
    "datapoints": [
        [1435766300000, 2.0],
        [1435766400000, 2.5],
        [1435766500000, 3.0],
        [1435766550000, 3.5],
        [1435766600000, 4.0],
        [1435766700000, 50.0],
        [1435766800000, 100.0],
        [1435766900000, 150.5],
        [1435777000000, 175.0],
        [1435777100000, 175.5],
        [1435777200000, 200.0],
        [1435777300000, 225.0],
        [1435777400000, 225.5],
        [1435777500000, 250],
        [1435777600000, 300.0]
    ]
}]
```

Example Query

```
{
    "start": 1435766300000,
    "end": 1435767300000,
    "tags": [{
        "name": "test.interpolation.doubleType",
        "aggregations": [{
            "type": "interpolate",
            "interval": "50000ms"
        }]
    }]
}
```

Example Result

```
{
  "tags": [
    {
      "name": "test.interpolation.doubleType",
      "results": [
        {
          "groups": [
            {
              "name": "type",
              "type": "number"
            }
          ],
          "values": [
            [1435766350000, 2.25, 3],
            [1435766400000, 2.5, 3],
```

```
            [1435766450000, 2.75, 3],
            [1435766500000, 3, 3],
            [1435766550000, 3.5, 3],
            [1435766600000, 4, 3],
            [1435766650000, 27,3],
            [1435766700000, 50, 3],
            [1435766750000, 75, 3],
            [1435766800000, 100, 3],
            [1435766850000, 125.25, 3],
            [1435766900000, 150.5, 3],
            [1435766950000, 150.5, 3],
            [1435767000000, 150.5, 3],
            [1435767050000, 150.5, 3],
            [1435767100000, 150.5, 3],
            [1435767150000, 150.5, 3],
            [1435767200000, 150.5, 3],
            [1435767250000, 150.5, 3],
            [1435767300000, 150.5, 3]
          ],
          "attributes": {}
        }
      ],
      "stats": {
        "rawCount": 8
      }
    }
  ]
}
```

Minimum With Mixed Data Types

The Minimum aggregator returns the most recent smallest value.

Example Ingestion JSON

```
[{
    "name": "test.min.mixedType",
    "datapoints": [
        [1435776300000, 2, 1],
        [1435776400000, null],
        [1435776500000, 10.5, 3],
        [1435776550000, "100", 2],
        [1435776600000, "string"],
        [1435776700000, "string36"],
        [1435776800000, true],
        [1435776900000, 3, 0]
    ]
}]
```

Example Query

```
{
    "start": 1435766300000,
    "end": 1435777000000,
    "tags": [{
        "name": "test.min.mixedType",
        "aggregations": [{
            "type": "min",
```

```
            "sampling": {
                "datapoints": 1
            }
        }]
    }]
}
```

Example Result

```
{
    "tags": [{
        "name": "test.min.mixedType",
        "results": [{
            "groups": [{
                "name": "type",
                "type": "number"
            }],
            "attributes": {},
            "values": [
                [1435776300000, 2, 1]
            ]
        }],
        "stats": {
            "rawCount": 4
        }
    }]
}
```

Minimum With Double Data Types

The Minimum aggregator returns the most recent smallest value.

Example Ingestion JSON

```
[{
    "name": "test.min.doubleType",
    "datapoints": [
        [1435766300000, 2.0],
        [1435766400000, 2.5],
        [1435766500000, 3.0],
        [1435766550000, 3.5],
        [1435766600000, 4.0],
        [1435766700000, 50.0],
        [1435766800000, 100.0],
        [1435766900000, 150.5],
        [1435777000000, 175.0],
        [1435777100000, 175.5],
        [1435777200000, 200.0],
        [1435777300000, 225.0],
        [1435777400000, 225.5],
        [1435777500000, 250],
        [1435777600000, 300.0]
    ]
}]
```

Example Query

```
{
    "start": 1435766300000,
    "end": 1435787600000,
    "tags": [{
        "name": "test.min.doubleType",
        "aggregations": [{
            "type": "min",
            "sampling": {
                "datapoints": 1
            }
        }]
    }]
}
```

Example Result

```
{
    "tags": [{
        "name": "test.min.doubleType",
        "results": [{
            "groups": [{
                "name": "type",
                "type": "number"
            }],
            "attributes": {},
            "values": [
                [1435766300000, 2, 3]
            ]
        }],
        "stats": {
            "rawCount": 15
        }
    }]
}
```

Maximum With Mixed Data Types

The Maximum aggregator returns the most recent largest value.

Example Ingestion JSON

```
[{
    "name": "test.max.mixedType",
    "datapoints": [
        [1435776300000, 2, 1],
        [1435776400000, null],
        [1435776500000, 10.5, 3],
        [1435776550000, "100", 2],
        [1435776600000, "string"],
        [1435776700000, "string36"],
        [1435776800000, true],
        [1435776900000, 3, 0]
    ]
}]
```

Example Query

```
{
    "start": 1435776300000,
    "end": 1435779600000,
    "tags": [{
        "name": "test.max.mixedType",
        "aggregations": [{
            "type": "max",
            "sampling": {
                "datapoints": 1
            }
        }]
    }]
}
```

Example Result

```
{
    "tags": [{
        "name": "test.max.mixedType",
        "results": [{
            "groups": [{
                "name": "type",
                "type": "number"
            }],
            "attributes": {},
            "values": [
                [1435776550000, 100, 2]
            ]
        }],
        "stats": {
            "rawCount": 4
        }
    }]
}
```

Maximum With Double Types

The Maximum aggregator returns the most recent largest value.

Example Ingestion JSON

```
[{
    "name": "test.max.mode.doubleType",
    "datapoints": [
        [1435766300000, 2.0],
        [1435766400000, 2.5],
        [1435766500000, 3.0],
        [1435766550000, 3.5, 2],
        [1435766600000, 4.0],
        [1435766700000, 50.0],
        [1435766800000, 100.0],
        [1435766900000, 150.5],
        [1435777000000, 175.0],
        [1435777100000, 175.5],
        [1435777200000, 200.0],
```

```
        [1435777300000, 225.0],
        [1435777400000, 225.5],
        [1435777500000, 250],
        [1435777600000, 300.0]
    ]
}]
```

Example Query

```
{
    "start": 1435766300000,
    "end": 1435777800000,
    "tags": [{
        "name": "test.max.mode.doubleType",
        "aggregations": [{
            "type": "max",
            "sampling": {
                "datapoints": 1
            }
        }]
    }]
}
```

Example Result

```
{
    "tags": [{
        "name": "test.max.mode.doubleType",
        "results": [{
            "groups": [{
                "name": "type",
                "type": "number"
            }],
            "attributes": {},
            "values": [
                [1435777600000, 300, 3]
            ]
        }],
        "stats": {
            "rawCount": 15
        }
    }]
}
```

Scale With Mixed Data Types

The Scale aggregator scales each data point by a factor.

Example Ingestion JSON

```
[{
    "name": "test.scale.mixedType",
    "datapoints": [
        [1435776300000, 2, 1],
        [1435776400000, null],
        [1435776500000, 10.5, 3],
```

```
        [1435776550000, "100", 2],
        [1435776600000, "string"],
        [1435776700000, "string36"],
        [1435776800000, true],
        [1435776900000, 3, 0]
    ]
}]
```

Example Query

```
{
"start": 1435766300000,
"end": 1435777000000,
"tags": [{
"name": "test.scale.mixedType",
"aggregations": [{ "type": "scale",
"factor": 3
}]
}]
}
```

Example Result

```
{
    "tags": [{
        "name": "test.scale.mixedType",
        "results": [{
            "groups": [{
                "name": "type",
                "type": "number"
            }],
            "attributes": {},
            "values": [
                [1435776300000, 6, 1],
                [1435776500000, 31.5, 3],
                [1435776550000, 300, 2],
                [1435776900000, 9, 0]
            ]
        }],
        "stats": {
            "rawCount": 4
        }
    }]
}
```

Scale With Double Data Types

The Scale aggregator scales each data point by a factor.

Example Ingestion JSON

```
[{
    "name": "test.scale.doubleType",
    "datapoints": [
        [1435766300000, 2.0],
        [1435766400000, 2.5],
```

```
        [1435766500000, 3.0],
        [1435766550000, 3.5, 2],
        [1435766600000, 4.0],
        [1435766700000, 50.0],
        [1435766800000, 100.0],
        [1435766900000, 150.5],
        [1435777000000, 175.0],
        [1435777100000, 175.5],
        [1435777200000, 200.0],
        [1435777300000, 225.0],
        [1435777400000, 225.5],
        [1435777500000, 250],
        [1435777600000, 300.0]
    ]
}]
```

Example Query

```
{
    "start": 1435766300000,
    "end": 1435787600000,
    "tags": [{
        "name": "test.scale.doubleType",
        "aggregations": [{
            "type": "scale",
            "factor": 3
        }]
    }]
}
```

The following shows an example query using the scale aggregator with factor, using the time series client library:

```
QueryBuilder builder = QueryBuilder.createQuery()
        .withStartAbs(1432671129000L)
        .withEndAbs(1432672230500L)
        .addTags(
                QueryTag.Builder.createQueryTag()
                        .withTagNames(
                                Arrays.asList("ALT_SENSOR")
                        )
                        .addAggregation(Aggregation.Builder.scaleWithFa
ctor((double) 10))
                        .build());
QueryResponse queryResponse =
ClientFactory.queryClientForTenant(tenant).queryForLatest(builder.buil
d());
```

Example Result

```
{
    "tags": [{
        "name": "test.scale.doubleType",
        "results": [{
            "groups": [{
                "name": "type",
                "type": "number"
            }],
            "attributes": {},
```

```
            "values": [
                [1435766300000, 6, 3],
                [1435766400000, 7.5, 3],
                [1435766500000, 9, 3],
                [1435766550000, 10.5, 2],
                [1435766600000, 12, 3],
                [1435766700000, 150, 3],
                [1435766800000, 300, 3],
                [1435766900000, 451.5, 3],
                [1435777000000, 525, 3],
                [1435777100000, 526.5, 3],
                [1435777200000, 600, 3],
                [1435777300000, 675, 3],
                [1435777400000, 676.5, 3],
                [1435777500000, 750, 3],
                [1435777600000, 900, 3]
            ]
        }],
        "stats": {
            "rawCount": 15
        }
    }]
}
```

Sum With Mixed Data Types

The Sum aggregator returns the sum of all values.

Example Ingestion JSON

```
[{
    "name": "test.sum.mixedType",
    "datapoints": [
        [1435776300000, 2, 1],
        [1435776400000, null],
        [1435776500000, 10.5, 3],
        [1435776550000, "100", 2],
        [1435776600000, "string"],
        [1435776700000, "string36"],
        [1435776800000, true],
        [1435776900000, 3, 0]
    ]
}]
```

Example Query

```
{
    "start": 1435766300000,
    "end": 1435777000000,
    "tags": [{
        "name": "test.sum.mixedType",
        "aggregations": [{
            "type": "sum",
            "sampling": {
                "datapoints": 1
            }
        }]
```

```
    }]
}
```

Example Result

```
{
    "tags": [{
        "name": "test.sum.mixedType",
        "results": [{
            "groups": [{
                "name": "type",
                "type": "number"
            }],
            "attributes": {},
            "values": [
                [1435776300000, 115.5, 3]
            ]
        }],
        "stats": {
            "rawCount": 4
        }
    }]
}
```

Trend Mode With Mixed Data Types

The Trend Mode aggregator returns the minimum and maximum value data point for the specified time range. Use the `sampling` parameter to specify the range.

Example Ingestion JSON

```
[{
    "name": "test.trend.mode.mixedType",
    "datapoints": [
        [1435776300000, 2, 1],
        [1435776400000, null],
        [1435776500000, 10.5, 3],
        [1435776550000, "100", 2],
        [1435776600000, "string"],
        [1435776700000, "string36"],
        [1435776800000, true],
        [1435776900000, 3, 0]
    ]
}]
```

Example Query

```
{
    "start": 1435776300000,
    "end": 1435779600000,
    "tags": [{
        "name": "test.trend.mode.mixedType",
        "aggregations": [{
            "type": "trendmode",
            "sampling": {
                "datapoints": 1
```

```
            }
        }]
    }]
}
```

Example Result

```
{
    "tags": [{
        "name": "test.trend.mode.mixedType",
        "results": [{
            "groups": [{
                "name": "type",
                "type": "number"
            }],
            "attributes": {},
            "values": [
                [1435776300000, 2, 1],
                [1435776550000, 100, 2]
            ]
        }],
        "stats": {
            "rawCount": 4
        }
    }]
}
```

Trend Mode With Double Data Types

The Trend Mode aggregator returns the minimum and maximum value data point for the specified time range. Use the `sampling` parameter to specify the range.

Example Ingestion JSON

```
[{
    "name": "test.trend.mode.doubleType",
    "datapoints": [
        [1435766300000, 2.0],
        [1435766400000, 2.5],
        [1435766500000, 3.0],
        [1435766550000, 3.5, 2],
        [1435766600000, 4.0],
        [1435766700000, 50.0],
        [1435766800000, 100.0],
        [1435766900000, 150.5],
        [1435777000000, 175.0],
        [1435777100000, 175.5],
        [1435777200000, 200.0],
        [1435777300000, 225.0],
        [1435777400000, 225.5],
        [1435777500000, 250],
        [1435777600000, 300.0]
    ]
}]
```

Example Query

```
{
    "start": 1435766300000,
    "end": 1435777800000,
    "tags": [{
        "name": "test.trend.mode.doubleType",
        "aggregations": [{
            "type": "trendmode",
            "sampling": {
                "datapoints": 1
            }
        }]
    }]
}
```

Example Result

```
{
    "tags": [{
        "name": "test.trend.mode.doubleType",
        "results": [{
            "groups": [{
                "name": "type",
                "type": "number"
            }],
            "attributes": {},
            "values": [
                [1435766300000, 2, 3],
                [1435777600000, 300, 3]
            ]
        }],
        "stats": {
            "rawCount": 15
        }
    }]
}
```

Filters

You can filter data points by attributes, qualities, and multiple tag names using a set of comparison operations, including the following:

Operation	Attribute
Greater than (>)	`gt`
Greater than (>) or equal to (=)	`ge`
Less than (<)	`lt`
Less than (<) or equal to (=)	`le`
Equal to (=)	`eq`
Not equal to (≠)	`ne`

Greater Than or Equal to Operation With Multiple Tags

The following code sample shows a query to the time series service instance, which includes multiple tags, a time window, and attributes for data points greater than (>), or equal to (=), a specific data point value. Example JSON request payload:

```
{
    "start": 1432671129000,
    "end": 1432671130500,
    "tags": [
        {
            "name": [
                "tag119",
                "tag121"
            ],
            "filters": {
                "measurements": {
                    "condition": "ge",
                    "values": "2"
                }
            }
        }
    ]
}
```

The following shows an example query filtering with multiple tags, using the time series client library:

```
QueryBuilder builder = QueryBuilder.createQuery()
        .withStartAbs(1432671129000L)
        .withEndAbs(1432671130500L)
        .addTags(
                QueryTag.Builder.createQueryTag()
                        .withTagNames(Arrays.asList("tag119",
"tag121"))
                        .addFilters(FilterBuilder.getInstance()
                                .addMeasurementFilter(FilterBuilder.Con
dition.GREATER_THAN_OR_EQUALS, Arrays.asList("2")).build())
                        .build());
QueryResponse queryResponse =
ClientFactory.queryClientForTenant(tenant).queryAll(builder.build());
```

Filtering by Attributes

The following example filters values for one or more tags:

```
{
    "start": 1349109376000,
    "end": 1349109381000,
    "tags": [
        {
            "name": [
                "<TagName1>",
                "<TagName2>"
            ],
            "filters": {
                "attributes": {
                    "< AttributeKey1>": "<AttributeValue1>"
                }
            }
        }
```

```
    ]
}
```

The following example filters a different value for each tag:

```
{
    "start": 1349109376000,
    "end": 1349109381000,
    "tags": [
        {
            "name": "<phosphate.level1>",
            "filters": {
                "attributes": {
                    "< AttributeKey1>": "<AttributeValue1>"
                }
            }
        },
        {
            "name": "<phosphate.level2>",
            "filters": {
                "attributes": {
                    "< AttributeKey2>": "<AttributeValue2>"
                }
            }
        }
    ]
}
```

Filtering by Qualities

The following example filters qualities for one or more tags:

```
{
    "start": 1349109376000,
    "end": 1349109381000,
    "tags": [
        {
            "name": [
                "<TagName1>",
                "<TagName2>"
            ],
            "filters": {
                "qualities": {
                    "values": ["0", "1"]
                }
            }
        }
    ]
}
```

The following example filters different qualities for each tag:

```
{
    "start": 1349109376000,
    "end": 1349109381000,
    "tags": [
        {
            "name": "<TagName1>",
            "filters": {
```

```
                "qualities": {
                    "values": ["0", "1"]
                }
            }
        },
        {
            "name": "<TagName2>",
            "filters": {
                "qualities": {
                    "values": ["0", "1"]
                }
            }
        }
    ]
}
```

Groups

Use the `groups` option in queries to return data points in specified groups.

Grouping by Attributes

You can group data points by attribute values as shown in the following example:

```
{
   "start":1432671128000,
   "end":1432671129000,
   "tags":[
      {
         "name":[
            "tagname1",
            "tagname2"
         ],
         "groups":[
            {
               "name":"attribute",
               "attributes":[
                  "attributename1"
               ]
            }
         ]
      }
   ]
}
```

Grouping by Quality

You can group data point results based on qualities as shown in the following example:

```
{
  "start": 1432671128000,
  "end": 1432671129000,
  "tags": [
    {
      "name": [
         "tagname1",
         "tagname2"
```

```
      ],
      "groups": [
           {
               "name": "quality"
           }
        ]
    }
  ]
}
```

Grouping by Measurement

You can group data point results based on measurement as shown in the following example:

```
{
    "start": "15d-ago",
    "end": "1mi-ago",
    "tags": [
        {
            "name": "tagname",
            "groups": [
                {
                    "name": "measurement",
                    "rangeSize": 3
                }
             ]
        }
    ]
}
```

The `rangeSize` is the number of values to place in a group. For example, a range size of 10 puts measurements between 0-9 in one group, 10-19 in the next group, and so on.

Grouping by Time

You can group data point results by a time range, beginning at the start time of the query as shown in the following example:

```
{
    "start": 1432671128000,
    "end": 1432671138000,
    "tags": [
        {
            "name": ["tagname1","tagname2"],
            "groups": [
                {
                    "name": "time",
                    "rangeSize": "1h",
                    "groupCount": 24
                }
            ]
        }
    ]
}
```

The `groupCount` property defines the maximum number of groups to return. In the above example, with a `rangeSize` of "1h" and a `groupCount` of "24", the return values are a range of one day, with one-hour groups.

Data Interpolation With Good and Bad Quality

This section shows examples of interpolating data points when there is bad quality data for a specific tag (of type integer), by time range.

Table 2: Data Quality Values

Value	Description
0	Bad quality.
1	Uncertain quality.
2	Not applicable.
3	Good quality. If you do not specify quality, the default is good (3).

Interpolation Between one Good Value and One Bad Value Data Point

The following shows an ingestion request where several data points are ingested, with one good data point followed by one bad data point with a later timestamp. Example request payload:

Tag Name	Timestamp	Measure	Quality
tagName	1432671129000	541	q=3
tagName	1432671130000	843	q=0

The following shows the query with a start time and end time that covers the data points created in the ingestion request above, with an aggregation interval that is less than the time windows between the two data points. Example request payload:

```
{
   "start":1432671129000,
   "end":1432671130500,
   "tags":[
      {
         "name":"coolingtower.corrosion.level",
         "aggregations":[
            {
               "type":"interpolate",
               "interval": "500ms"
            }
         ]
      }
   ]
}
```

The results show interpolated points between the good and bad data points have the value of the good data point. Example result:

```
{
   "tags": [{
      "name": "coolingtower.corrosion.level",
      "results": [{
         "groups": [{
            "name": "type",
```

```
            "type": "number"
         }],
         "attributes": {},
         "values": [
            [
               1432671129500,
               541,
               3
            ],
            [
               1432671130000,
               0,
               0
            ],
            [
               1432671130500,
               0,
               0
            ]
         ]
      }],
      "stats": {
         "rawCount": 2
      }
   }]
}
```

Interpolation Between One Bad Value and One Good Value Data Point

The following shows an ingestion request where several data points are ingested, with one bad data point followed by one good data point with a later timestamp. Example request payload:

Tag Name	Timestamp	Measure	Quality
tagName	1432671129000	541	q=0
tagName	1432671130000	843	q=3

The following shows the query with a start time and end time that covers the data points created in the ingestion request above, with an aggregation interval that is less than the time windows between the two data points. Example request payload:

```
{
   "start":1432671129000,
   "end":1432671130500,
   "tags":[
      {
         "name":"coolingtower.corrosion.level",
         "aggregations":[
            {
               "type":"interpolate",
               "interval": "500ms"
            }
         ]
      }
   ]
}
```

The expected result is that the interpolated points between the bad and good data points have a value of 0 (zero). Example result:

```
{
   "tags": [{
      "name": "coolingtower.corrosion.level",
      "results": [{
         "groups": [{
            "name": "type",
            "type": "number"
         }],
         "attributes": {},
         "values": [
            [
               1432671129500,
               0,
               0
            ],
            [
               1432671130000,
               843,
               3
            ],
            [
               1432671130500,
               843,
               3
            ]
         ]
      }],
      "stats": {
         "rawCount": 2
      }
   }]
}
```

Interpolation Between Two Bad Values

The following shows an ingestion request with one bad data point followed by one bad data point with a later timestamp:

Tag Name	Timestamp	Measure	Quality
tagName	1432671129000	541	q=0
tagName	1432671130000	843	q=0

The following shows a query where the start time and end time fully covers the two data points created in the previous samples, with an aggregation interval that is less than the time window between the data points. Example request payload:

```
{
   "start":1432671129000,
   "end":1432671130500,
   "tags":[
      {
         "name":"coolingtower.corrosion.level",
         "aggregations":[
            {
               "type":"interpolate",
```

```
                "interval": "500ms"
            }
        ]
    }
]
}
```

The expected result is that the interpolated points between the bad data points have a value of 0 (zero). Example result:

```
{
   "tags": [{
      "name": "coolingtower.corrosion.level",
      "results": [{
         "groups": [{
            "name": "type",
            "type": "number"
         }],
         "attributes": {},
         "values": [
            [
               1432671129500,
               0,
               0
            ],
            [
               1432671130000,
               0,
               0
            ],
            [
               1432671130500,
               0,
               0
            ]
         ]
      }],
      "stats": {
         "rawCount": 2
      }
   }]
}
```

Interpolation of a Bad Data Point

The following shows an ingestion request where several data points are injected, with at least one bad data point. Example request payload:

Tag Name	Timestamp	Measure	Quality
tagName	1432671129000	541	q=0
tagName	1432671130000	843	q=3

The following shows a data query with a start time that is equal to the timestamp of the bad data point. Example request payload:

```
{
   "start":1432671129000,
   "end":1432671130500,
```

```
   "tags":[
      {
         "name":"coolingtower.corrosion.level",
         "aggregations":[
            {
               "type":"interpolate",
               "interval": "500ms"
            }
         ]
      }
   ]
}
```

The expected result is that the interpolated points at the bad data point have a value of 0 (zero). Example result:

```
{
   "tags": [{
      "name": "coolingtower.corrosion.level",
      "results": [{
         "groups": [{
            "name": "type",
            "type": "number"
         }],
         "attributes": {},
         "values": [
            [
               1432671129500,
               0
            ],
            [
               1432671130000,
               843,
               3
            ],
            [
               1432671130500,
               843,
               3
            ]
         ]
      }],
      "stats": {
         "rawCount": 2
      }
   }]
}
```

Data Interpolation With Good and Bad Quality With Double Data Type

This section shows examples of interpolating data points when there is bad quality data for a specific tag of floating point (double) type, by time range.

Table 3: Data Quality Values

Value	Description
0	Bad quality.
1	Uncertain quality.
2	Not applicable.
3	Good quality. If you do not specify quality, the default is good (3).

Interpolation Between one Good Value and One Bad Value Data point

The following shows an ingestion request where several data points are ingested, with one good data point followed by one bad data point with a later timestamp. Example request payload:

Tag Name	Timestamp	Measure	Quality
tagName	1432671129000	541.26	q=3
tagName	1432671130000	843.25	q=0

The following shows the query with a start time and end time that covers the data points created in the ingestion request above, with an aggregation interval that is less than the time windows between the two data points. Example request payload:

```
{
   "start":1432671129000,
   "end":1432671130500,
   "tags":[
      {
         "name":"coolingtower.corrosion.level",
         "aggregations":[
            {
               "type":"interpolate",
               "interval": "500ms"
            }
         ]
      }
   ]
}
```

The results show interpolated points between the good and bad data points have the value of the good data point. Example result:

```
{
   "tags": [{
      "name": "coolingtower.corrosion.level",
      "results": [{
         "groups": [{
            "name": "type",
            "type": "number"
         }],
         "attributes": {},
         "values": [
            [
               1432671129500,
               541.26,
               3
```

```
            ],
            [
               1432671130000,
               0,
               0
            ],
            [
               1432671130500,
               0,
               0
            ]
         ]
      }],
      "stats": {
         "rawCount": 2
      }
   }]
}
```

Interpolation Between One Bad Value and One Good Value Data point

The following shows an ingestion request where several data points are ingested, with one bad data point followed by one good data point with a later timestamp. Example request payload:

Tag Name	Timestamp	Measure	Quality
tagName	1432671129000	541.26	q=0
tagName	1432671130000	843.25	q=3

The following shows the query with a start time and end time that covers the data points created in the ingestion request above, with an aggregation interval that is less than the time windows between the two data points. Example request payload:

```
{
   "start":1432671129000,
   "end":1432671130500,
   "tags":[
      {
         "name":"coolingtower.corrosion.level",
         "aggregations":[
            {
               "type":"interpolate",
               "interval": "500ms"
            }
         ]
      }
   ]
}
```

The expected result is that the interpolated points between the bad and good data points have a value of 0.0. Example result:

```
{
   "tags": [{
      "name": "coolingtower.corrosion.level",
      "results": [{
         "groups": [{
            "name": "type",
```

```
            "type": "number"
         }],
         "attributes": {},
         "values": [
            [
               1432671129500,
               0,
               0
            ],
            [
               1432671130000,
               843.25,
               3
            ],
            [
               1432671130500,
               843.25,
               3
            ]
         ]
      }],
      "stats": {
         "rawCount": 2
      }
   }]
}
```

Interpolation Between Two Bad Values

The following shows an ingestion request with one bad data point followed by one bad data point with a later timestamp:

Tag Name	Timestamp	Measure	Quality
tagName	1432671129000	541.26	q=0
tagName	1432671130000	843.25	q=0

The following shows a query where the start time and end time fully covers the two data points created in the previous samples, with an aggregation interval that is less than the time window between the data points. Example request payload:

```
{
   "start":1432671129000,
   "end":1432671130500,
   "tags":[
      {
         "name":"coolingtower.corrosion.level",
         "aggregations":[
            {
               "type":"interpolate",
               "interval": "500ms"
            }
         ]
      }
   ]
}
```

The expected result is that the interpolated points between the bad data points have a value of 0.0. Example result:

```
{
   "tags": [{
      "name": "coolingtower.corrosion.level",
      "results": [{
         "groups": [{
            "name": "type",
            "type": "number"
         }],
         "attributes": {},
         "values": [
            [
               1432671129500,
               0,
               0
            ],
            [
               1432671130000,
               0,
               0
            ],
            [
               1432671130500,
               0,
               0
            ]
         ]
      }],
      "stats": {
         "rawCount": 2
      }
   }]
```

Interpolation Between Two Good Values

The following shows an ingestion request where several data points are injected with one good data point followed by a good data point with a later timestamp:

Tag Name	Timestamp	Measure	Quality
tagName	1432671129000	541.26	q=3
tagName	1432671130000	843.25	q=3

The following shows a data query where the start and end time cover the two good data points.

```
{
    "start": 1432671129000,
    "end": 1432671130500,
    "tags": [
        {
            "name": "coolingtower.corrosion.level",
            "aggregations": [
                {
                    "type": "interpolate",
                    "interval": "500ms"
                }
            ]
```

```
        }
    ]
}
```

The below shows the expected result:

```
{
   "tags": [{
      "name": "test.interpolation.mixed.quality.integer",
      "results": [{
         "groups": [{
            "name": "type",
            "type": "number"
         }],
         "attributes": {},
         "values": [
            [
               1432671129500,
               692.255,
               3
            ],
            [
               1432671130000,
               843.25,
               3
            ],
            [
               1432671130500,
               843.25,
               3
            ]
         ]
      }],
      "stats": {
         "rawCount": 2
      }
   }]
}
```

Interpolation of a Bad Data Point

The following shows an ingestion request where several data points are injected, with at least one bad data point. Example request payload:

Tag Name	Timestamp	Measure	Quality
tagName	1432671129000	541.26	q=0
tagName	1432671130000	843.25	q=3

The following shows a data query with a start time that is equal to the timestamp of the bad data point. Example request payload:

```
{
   "start":1432671129000,
   "end":1432671130500,
   "tags":[
      {
         "name":"coolingtower.corrosion.level",
         "aggregations":[
```

```
            {
               "type":"interpolate",
               "interval": "500ms"
            }
         ]
      }
   ]
}
```

The expected result is that the interpolated points at the bad data point have a value of 0.0. Example result:

```
{
   "tags": [{
      "name": "coolingtower.corrosion.level",
      "results": [{
         "groups": [{
            "name": "type",
            "type": "number"
         }],
         "attributes": {},
         "values": [
            [
               1432671129500,
               0,
               0
            ],
            [
               1432671130000,
               843.25,
               3
            ],
            [
               1432671130500,
               843.25,
               3
            ]
         ]
      }],
      "stats": {
         "rawCount": 2
      }
   }]
}
```

Unbinding an Application From the Time Series Service

Procedure

- To unbind a service instance, run:

  ```
  cf unbind-service <application_name> <my_time_series_instance>
  ```

  where:

  - *application_name* is is the name of your application.
  - *my_time_series_instance* is the name of your service instance.

Data Removal

You can permanently remove Time Series data and check status using a HTTP request to the REST APIs.

Note: The hard delete feature is currently available only in select environments. Contact Support to determine if this feature is enabled in your environment.

Permanent Data Removal (Hard Delete)

The hard delete features enables the following actions:

- Delete time series data specifying tag(s) and a time window.
- Delete time series data when start time is the beginning of an hour and end time is the end of an hour (Unix Time).
- Delete all the times series data in a tag.
- Perform delete operations asynchronously.
- Perform multiple delete operations in one delete request.
- Retrieve the `correlationID` for each delete request in order to query for the delete request status.

These additional actions are supported for the Dedicated plan:

- Delete time series data by filtering attribute values.
- Delete any time series data point (that is, you are not restricted to only the beginning or end of an hour).

Tips for Permanent Data Removal

- Invalid delete requests (such as JSON parsing, validations, etc.) are returned only when using the delete status API.
- Start time cannot be greater than End time.
- Setting the `start` parameter to START_OF_TIME sets the start to the earliest time available.
- Setting the `end` parameter to END_OF_TIME sets the end to the latest time available.
- Tag information is not deleted when all data points are deleted.
- There is no guarantee of atomicity of a delete action. If a delete action fails, only some data points may be deleted but not all. Issue a retry if the action fails to ensure that a delete action succeeds.
- Delete requests do not have an SLA.

Important: There is no way to undo a delete. Use with caution.

Requirements for Permanent Data Removal — Free and Tiered Plans

- Start time must correspond to the start of an hour in milliseconds. For examples, see #unique_92/unique_92_Connect_42_table_25843228-3f7b-4188-9c51-557e171ff42d.
- End time must correspond to the end of an hour in milliseconds (or the start of the next hour minus one millisecond). For examples, see #unique_92/unique_92_Connect_42_table_75579766-3421-4a42-ad8c-8fe57182797c.

Table 4: Examples: Start Time

Valid Start Time	Invalid Start Time
1532653200000 (Friday, July 27, 2018 1:00:00 AM)	1532653442000 (Friday, July 27, 2018 1:04:02 AM)
901522800000 (Monday, July 27, 1998 7:00:00 AM)	901522802000 (Monday, July 27, 1998 7:00:02 AM)

Table 5: Examples: End Time

Valid End Time	Invalid End Time
15326567999999 (Sunday, September 5, 2455 7:59:59.999 PM	1532653442000 (Friday, July 27, 2018 1:04:02 AM)
	1532653200000 (Friday, July 27, 2018 1:00:00 AM)

Deleting Data Permanently (Hard Delete)

Note: The hard delete feature is currently available only in select environments. Contact Support to determine if this feature is enabled in your environment.

Before You Begin

You must have the `delete` scope.

Procedure

1. View the environment variables for your application by entering the following on a command line:

   ```
   cf env <application_name>
   ```

   The delete URI, trusted issuer token, and `<Predix-Zone-Id>` are included with the environment variables for your application. You will use this information in the HTTP header of your query.
2. Create the delete request.

   The header must be in the following format, using values from step 1:

   ```
   URL: http://delete_url/v1/datapoints/delete
   Method: POST
   Headers:
       Authorization: Bearer <token from trusted issuer>
       Predix-Zone-Id: <Predix-Zone-Id>

   {
       "body": {
           "tags": [
               {
                   "name": [
                       "<TagName1>",
                       "<TagName2>"
                   ],
                   "filters": {
                       "attributes": {
                           "<AttributeKey>": "<AttributeValue>",
                           "<AttributeKey2>": "<AttributeValue2>"
                       }
                   }
               },
   ```

```
            {
                "name": [
                    "<TagName3>",
                    "<TagName4>"
                ],
                "filters": {
                    "attributes": {
                        "<AttributeKey3>": "<AttributeValue3>",
                        "<AttributeKey4>": "<AttributeValue4>"
                    }
                }
            }
        ],
        "start": "START_OF_TIME",
        "end": "END_OF_TIME"
    }
}
```

3. The following is an example of the delete response.

```
{
    "statusCode": 202,
    "correlationId": "00000166-36cc-b44b-5d85-db5918aeed67"
}
```

Querying Data Delete Status

Note: The hard delete feature is currently available only in select environments. Contact Support to determine if this feature is enabled in your environment.

Before You Begin

You must have the `query` scope.

Procedure

1. View the environment variables for your application by entering the following on a command line.

```
cf env <application_name>”
```

   The query URI, trusted issuer token, and `<Predix-Zone-Id>` are included with the environment variables for your application. You will use this information in the HTTP header of your query.
2. Create the query request.

```
URL: http://query_uri/v1/datapoints/delete/status
    Method: GET
    Headers:
        Authorization: Bearer <token from trusted issuer>
        Predix-Zone-Id: <Predix-Zone-Id>
       Parameters:
        correlationID: <correlationID>
```

3. The following is an example of the status response.

```
HTTP Code: 200

{
    "correlationID": "<correlationID>",
```

```
    "operationType": "DELETE",
    "statusText": "<status text>”
}
```

`statusText` value will be one of the following:

- created
- paused
- processing
- finished
- not found
- expired
- failed

Deleting a Time Series Service Instance

Procedure

- To delete a service instance, run:

```
cf delete-service <service_instance_name>
```

Troubleshoot Time Series

Troubleshoot Time Series Queries

Error Message When Querying the Time Series Service Instance

When you submit a query using raw query mode or another aggregation, you receive an error message in this format:

```
"<tag_name> [0] exceeded the maximum number of data points that could
be retrieved (<count of data points found > query maximum limit).
Reduce the time window and try again."
```

Cause

The number of data points retrieved by a query cannot exceed the maximum limit of 500,000.

Solution

Narrow your query criteria (for example, the time window) to return a fewer number of data points.

401 Unauthorized Message Received When Making a Query Request

Cause

If the client cannot be authenticated, an HTTP response code `401: Unauthorized` is returned.

This can happen for the following reasons:

- The `<Predix-Zone-Id>` tag you are using is incorrect.
- The token is invalid or expired.
- The client ID is incorrect.
- The client is missing the OAuth2 scopes and authorities.

Solution

Check the environment variables for your application:

```
cf env <application_name>
```

Ensure that:

- You are using the correct `<Predix-Zone-Id>` for your service instance.
- You are using a valid token.
- You are using the correct client ID.
- You added the OAuth2 scopes and authorities to the client.

See Updating the OAuth2 Client Using UAAC for more information.

503 Acknowledgement Message

Cause

There could be an issue with the environment.

Solution

File a support ticket. See #unique_99.

Unable to Query the Data Points That Were Ingested

Cause

Possible causes include:

- The `<Predix-Zone-Id>` used in the ingestion request client is different from the `<Predix-Zone-Id>` used in the query client.
- The application that is bound to the time series instance used for data ingestion is not the same as the one bound to the application used for querying.
- Invalid characters used in tag name, or attribute names, or attribute values.
- Data does not comply to the required structure.

Solution

- Verify that the `<Predix-Zone-Id>` you are using in your query request is the same as the one used in the ingestion request.
- Verify that the application bound to the time series service instance for querying is the same as the one used for data ingestion.
- Verify that you are using the correct tag structure and valid characters.
  See *Time Series Tag Structure* in #unique_53.

Unable to Query Some of the Data Points That Were Ingested

Description

After sending an ingestion request, you are unable to query some of the data points.

Cause

Possible causes include:

- Network delays
- Missing data points

Solution

- Check the list of possible causes in *Unable to Query the Data Points That Were Ingested*.
- File a support ticket. See #unique_99.

Query Returns No Data Points

Description

When you submit a query, you receive a 200 response, indicating the query was successful, but no data points are returned.

Cause

This can happen if the Timestamp in your time series tag is not in milliseconds.

Solution

Use only UNIX epoch time in milliseconds in the time series tag Timestamp.

Related Information

Pushing Time Series Data on page 29

Troubleshoot Time Series Data Ingestion

The following are some issues you may encounter when sending ingestion requests to the Time Series service.

400 Acknowledgement Message

You receive a `400` acknowledgement message when making an ingestion request.

Cause

Possible causes are:

- The data points you are attempting to ingest do not conform to the message envelope JSON.
- You are using the Java WebSocket package and using the `sendText` method.
- The write buffer for your WebSocket client is not set to 512KB.

Solution

If you receive a `400` error, check the following:

- Check the message and make sure it is well-formed JSON is the message ID is returned with the 400 error.
- Verify that you are not chunking your ingestion request. Each message must conform to the JSON format specified.
- If you are using the Java WebSocket package (javax.websocket), use the `sendBinary` method instead of `sendText`.
- If you are using a WebSocket package in any language other than Java, verify the write buffer for your WebSocket client is set to 512KB (524,288 bytes).

401 Acknowledgement Message

If the client cannot be authenticated, an HTTP response code `401: Unauthorized` is returned.

Cause

When you receive a `401 Authorization` acknowledgement message when making a data ingestion request, it may be due to one of the following reasons:

- The `<Predix-Zone-Id>` you are using is incorrect.
- The token is invalid or expired.
- The client ID is incorrect.
- The client is missing the OAuth2 scopes and authorities.
- If you are using the Time Series client library on Linux, the Java program may not be picking up the environment variables you set for the client secret.

Solution

If you receive a `401: Unauthorized` error, try the following:

- Check the environment variables for your application:

```
cf env <application_name>
```

- Verify that you are using the correct `<Predix-Zone-Id>` for your service instance.
- Verify that you are using a valid token.
- Verify that you are using the correct client ID.

- Verify that you added the OAuth2 scopes and authorities to the client.
- If you are using the Time Series client library on Linux, make sure the Java program is picking up the environment variables you set for the client secret.
    - Run the following command:

      ```
      System.out.println(System.getenv())
      ```

      This prints all of the environment variables Java has access to. If the environment variable is not there, do the following:

        - Make sure it is set correctly, either by exporting it prior to runtime or setting it in `~/.bashrc` or
        - Set the environment variable in the run configuration of the IDE.

See Updating the OAuth2 Client Using UAAC for more information.

413 Acknowledgement Message

You receive a `413` acknowledgement message when making a data ingestion request.

Cause

The ingestion request payload exceeds the maximum limit of 512KB. Note that even though we now support compressed GZIP JSON payloads, the decompressed payload must still be less than 512KB.

Solution

Resubmit the ingestion request with a request payload that is less than 512KB.

Filing a Support Ticket for Time Series

Procedure

1. Go to the Predix portal at http://predix.io, and click **Support**.
2. Click **File a support ticket**, and provide the following information:
    - The `<Predix-Zone-Id>` of the ingestion request.
    - The message ID of the data point in the ingestion request.
    - The correlation ID from the query response. This is in the headers of every successful response and body of any error response.

Time Series Release Notes

Time Series

Q3 2016

New Features

Compressed Format Accepted for Data Ingestion

The Time Series service now accepts compressed (GZIP) JSON payloads. The size limit for the actual JSON payload is 512 KB regardless of the ingestion request format. For compressed payloads, this means that the decompressed payload cannot exceed 512 KB.

Q2 2016

New Features

Data Ingestion

You can use mixed data types, including null and string in data ingestion.