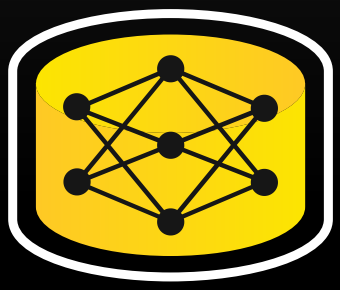
H2O.ai



# H2O FEATURE STORE

DOCUMENTATION

For questions, please contact **support@h2o.ai**

# Contents



# What is Feature Store?

The H2O Feature Store is a centralized repository for storing, updating, retrieving, and sharing features used in machine learning models across different projects. For most machine learning and AI applications, raw data must be transformed into features that are optimized for capturing information from the data. H2O AI Feature Store allows data scientists and engineers to easily organize, govern, share, and reuse these features. With H2O AI Feature Store, organizations can improve collaboration, increase the efficiency of the model development process, and deliver impactful AI outcomes faster.

This page contains the downloadable artifacts users need for connecting to the Feature Store server.

## Feature Store artifacts

### Clients

The following are the clients you can use to connect to Feature Store. You only need one depending on which environment you are using.

- Python
- Snowflake
- GRPC API

**Python** Python 3.7 or later is required to use the Feature Store Python client. You can install the Python client through several different methods. The simplest is via pip:

```
pip install h2o-featurestore
```

You can also install the Python client using of the of the following packages:

- Python client zip
- Python client wheel

**GRPC API** Feature Store uses GRPC as its communication protocol. If you use Java, you can use the GRPC API to connect to Feature Store. This is the download link for Feature Store's Java GRPC API client library:

- Java GRPC Feature Store API

### Azure Gen2 Spark dependencies

If you are using Azure Gen2 as the Feature Store storage cache, you will need Spark.

- Azure Gen2 Spark Dependencies

## Kubernetes deployment

You can use helm charts to deploy Feature Store in a Kubernetes cluster.

### K8s helm charts

This is the file you need to install Feature Store in Kubernetes:

- Feature Store Helm Charts

# Architecture

This section describes the main components of the Feature Store.

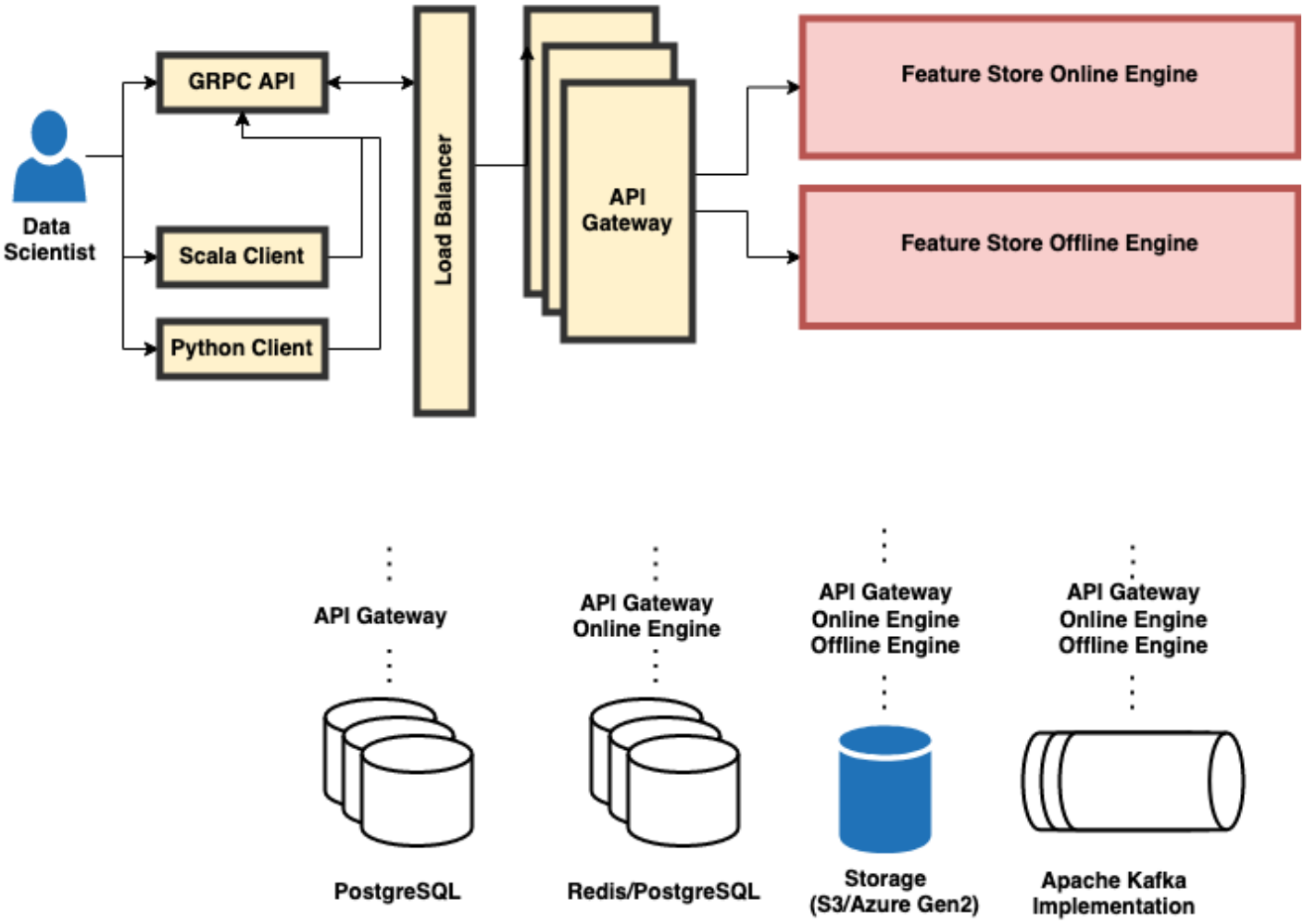
## High-level architecture

This subsection provides a basic overview of how the Feature Store runs.

You can push data to Feature Store through one of the Clients. The load-balancer acts as an ingress router for the cloud and routes the traffic to the Feature Store service that exposes the API. This lets you connect to one of the engines.

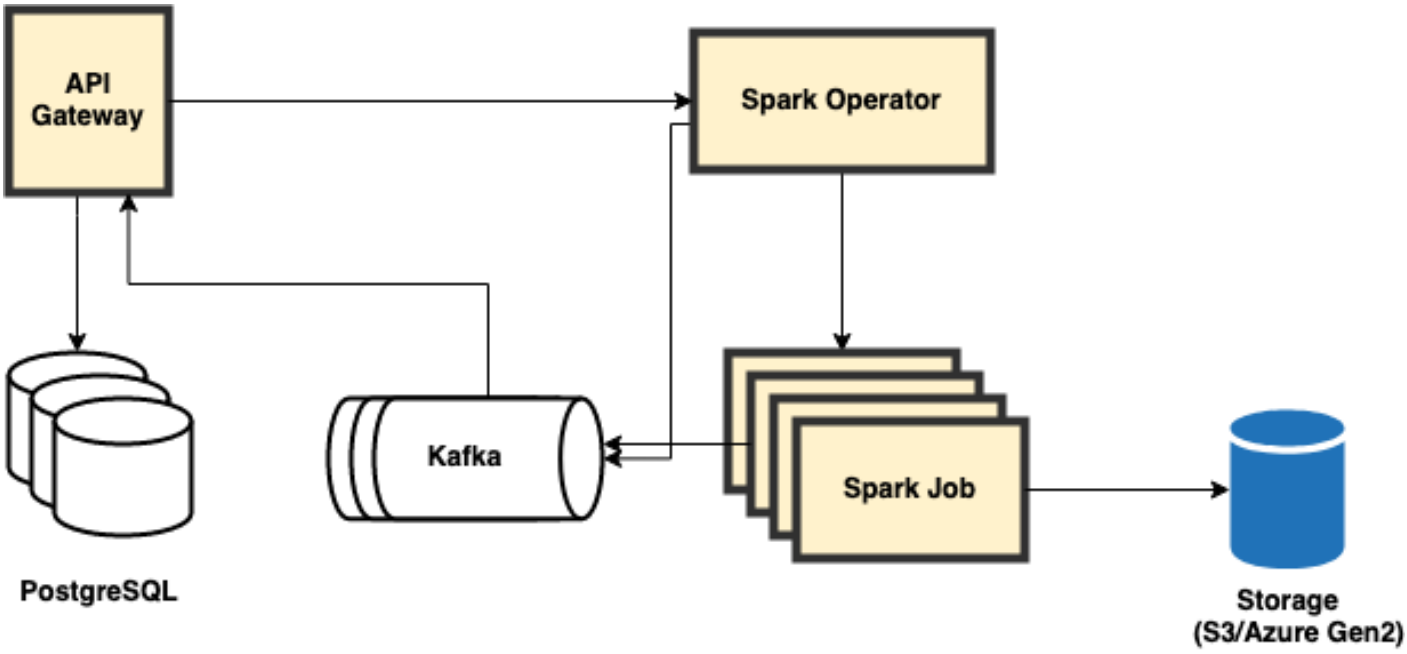
There are two engines: offline and online. The offline engine is a core service that handles API requests. It stores the metadata (that is, feature sets) used to train the model in a PostgreSQL database. The online engine stores the users' data and metadata with better accessibility. It uses this information to inference the model. The online engine can store online records data in Redis or in PostgreSQL database. You can sync data between the offline and online engine depending on how you want to use it. The main difference between the two engines is the real data location.

Persistent storage can be located in either S3 or in Azure Gen2. Apache Kafka queues requests for the offline and online engines and passes them to the Feature Store.




### Feature Store offline engine

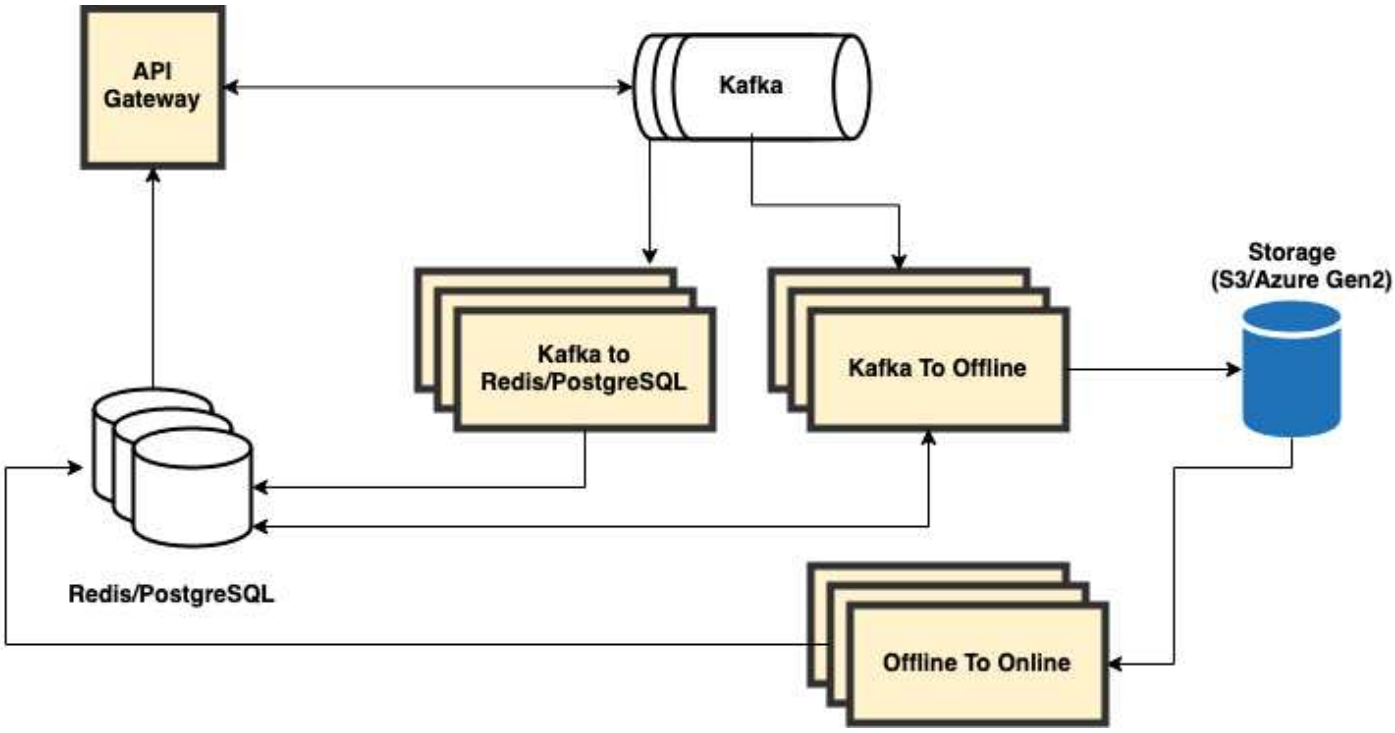
The core (pictured here as the offline engine) is a service that handles API requests.




**Feature Store online engine**

The online engine is a scalable component that provides support for main three functionalities:

- storing data into the online storage - supported databases are Redis and PostgreSQL.
- saving data regularly from the online databases into the offline Feature Store storage ( `online to offline` ).
- updating Redis after each ingestion into the offline Feature Store storage with missing values ( `offline to online` ).




## Technical components

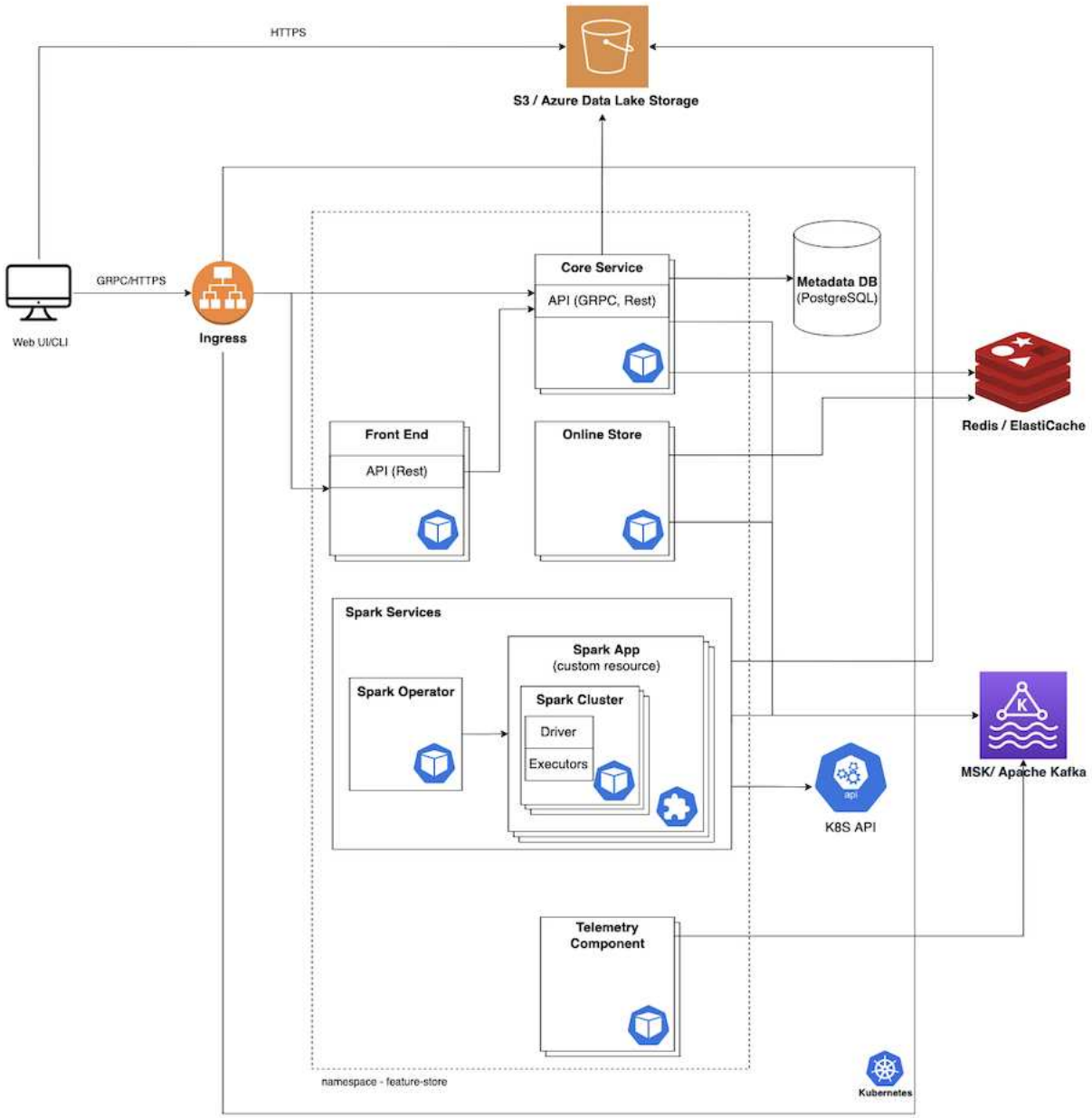
- **Authorization** - PostgreSQL for storing the authorization data
- **Authentication** - OpenIdConnect
- **Database** - PostgreSQL
- **Execution framework** - Apache Spark

- **Deployment environment** - Kubernetes
- **API definition** - ProtoBuf & GRPC

## Running Feature Store in production

Feature Store itself requires four components to be up and running inside a Kubernetes cluster. These main services will be installed automatically when you use our Helm charts.

Preferred architecture used in production:




**Note:**

This graph does not present OAuth integration. To authenticate a user in the system, it has to be integrated with SSO (which supports OpenID protocol).

### Feature Store services

**Core API** The Feature Store Core is a Feature Store component that is always deployed within the container of the Kubernetes cluster. It uses GRPC to define the API, allowing it to be easily consumed by the clients. It performs several

important functions:

- registers the features in the database.
- triggers the start of data manipulation tasks on the Spark cluster.
- performs authentication and queries for authorization permissions.

This service provides a customer-facing API. It can be exposed via Kubernetes Ingress or any other Load Balancer. The Ingress must be able to route traffic to a GRPC service. We recommend to start at least two pods of this component to ensure high availability. We also suggest to assign at least 2 CPUs and 2GB of RAM for each pod, but the values depend on the client needs.

**Online store** This service is responsible for ingesting data into Redis and managing online-to-offline processes. We recommend starting at least two pods of this component to ensure high availability. We also suggest assigning at least 2 CPUs and 2GB of RAM for each pod, but the values depend on the client needs.

**Spark operator** The Kubernetes operator which manages the Spark jobs. The Feature Store Core submits a Spark job specification to the Spark operator. The Spark operator fully manages the life cycle of the Spark job. It also pushes job status change events to a messaging queue.

Only one pod can be present in the cluster. We suggest assigning at least 2 CPUs and 2GB of RAM to the pod.

**Spark job** The Spark cluster started dynamically by the operator to process your data. The Spark cluster is used for data manipulation tasks (such as registering new features or creating output data from selected features). The Feature Store Core takes care of starting and stopping separate Spark clusters for each new job. The communication between the Core and Spark is also implemented via GRPC.

We recommend using dedicated Kubernetes nodes with auto-scaling to host Spark clusters. The number of resources in the production environment depends on the feature set sizes, but we recommend assigning no less than 2 CPUs and 8GB of RAM. By default, the Spark Cluster will use 1 driver pod + 3 executors.

### Third-party software

Feature Store also requires third-party software to be present and accessible. This third-party software is seen lining the outside of the preceding production image.

**PostgreSQL** Feature Store uses PostgreSQL as the database. The database stores information about features, feature sets, projects, and users (also known as the metadata).

Postgres can also be used as a backend for online Feature Store (Redis is the default).

**Redis** By default, the Feature Store online engine uses Redis as a backend to store online data. It is recommended to use the Redis cluster setup, but any other configuration should work. Memory allocated to the cluster depends on the client needs and features stored in it.

**Main storage** The system supports 2 types of storage:

- S3 (and S3 compatible, e.g., Minio)
- Azure Data Lake Storage Gen2

**Event platform** We recommend you use Kafka.

# Concepts

This page explains the main concepts of Feature Store.

## Projects

Projects are the repository that contain feature sets which are comprised of features. Projects can be used to separate work by department (e.g., engineering and accounting).

### Project Access Modifiers

Access to projects can be further modified by access modifiers. For more information, please see Projects access modifiers.

### Features

Features are columns of highly curated data. Features are used to enhance the performance of ML models because features are measurable data. Features can be seen when you call the schema, and the printout will be in the order of `<column title> <feature>`. For example:

```
category STRING, jobtitle STRING
```

### Feature sets

A feature set is a collection of features. Feature sets are created via registration from the feature set schema. Registering a feature set simply means you are creating a new feature set. This information comes from a schema that you have extracted from a raw data source that you ingested into Feature Store.

The data sources for ingestion are available on the Supported data sources page.

**Derived feature sets** Feature Store has the ability to create derived feature sets. Derived feature sets are created from a parent feature set that has applied transformations. When the parent feature set is ingested to or reverted from, it automatically triggers the ingesting and/or reverting changes for its derived feature set.

The supported ways of transformation are:

- Spark Pipeline
- Driverless AI MOJO
- Join

**Keys** A feature in the feature set can be marked as a primary key. This primary key can be used to search for a specific item in your data. Primary keys must have a unique value (e.g., a social security number). When you want to create data from more feature sets, these are the keys used for the joining process.

**Tags** Tags can be attached to feature sets for filtering purposes.

**Types of Feature Store users** There are several types of user permissions in Feature Store. For more information please see Permissions.

## Storage

Feature Store uploads outputted data to a data store. You can obtain the data by downloading it using the pre-signed URL link.

### Storage backend

Multiple storage backends are supported:

- Any system exposing S3 API (AWS, Google Cloud, Minio)
- Azure Data Lake Gen 2

**Storage file format** Files are written in delta format.

### Output data

Output data results from the materialization of the features. The data can then be used inside any ML platform.

### Incremental ingest

Incremental ingestion is a consistent ingestion that takes place over time. Instead of ingesting all the data at once, it ingests new data over time (e.g., every five hours or every day). This can be done through scheduled ingestion.

Feature Store maintains one entry in storage for each major version of a feature set. New data are appended to storage during each new data ingest. Only unique values are appended.

# Prerequisites

1. Ensure that you have **Kubernetes running** and **kubectl configured** to access this cluster.
2. **Install the Helm CLI** on your local machine. Minimal supported version is *3.3.4*.
3. **Download helm charts** from the Downloads page.
4. Ensure you have your **identity provider running** and that it is configured correctly. Refer to IDP configuration for more details.
5. Ensure that you have **Azure Event Hubs** or **AWS MSK** or **Kafka preconfigured** with at least 1 topic.
    a. Feature Store Core asynchronously communicates with the Spark operator and Spark jobs by interfacing with a message queue system.

In the case of using Snowflake as a offline storage, please follow the Snowflake prerequisites instructions.

# Requirements

## Kubernetes cluster

Feature Store must be deployed on top of Kubernetes. Try using the Helm charts delivered by H2O.ai.

## Identity provider

Please make sure your identity provider, such as Keycloak or Azure Active Directory, is running.

For Azure, we provide additional information available at Configuration of Azure Active Directory client.

## PostgreSQL

PostgreSQL is used to store all Feature Store metadata.

## Database for online records

Either Redis or PostgreSQL can be used for storing online features.

## Main storage

Feature Store supports Azure Data Lake Gen2 or any Amazon S3 API supported cloud storage services.

## Messaging system

Feature Store uses Kafka as messaging protocol. Therefore, implementations of Kafka protocol in Cloud are natively supported (e.g., Azure EventHub or Amazon MSK).

# Kubernetes Helm charts

## Charts

Feature Store Helm charts are designed to be a lightweight way to install Feature Store official services. It is the official way to install the services.

## Supported configurations

We recommend matching the versions of the Helm chart and the Feature Store you're installing.

The Helm chart which matches this version of Feature Store is available in the kubernetes section of the download page.

## Deploying Feature Store with Helm

You have to prepare the values file before running the `install` command because it is required to configure the deployment. Default values are documented in the Helm values section.

Install Helm charts:

```
helm install feature-store PATH_TO_DOWNLOADED_CHART --values helm-values.yaml -n K8S_NAMESPACE
```

The output should look like this:

```
NAME: feature-store
LAST DEPLOYED: Wed Feb 16 13:52:34 2022
NAMESPACE: fs-local
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

# System events

Feature Store `core` service notifies 3rd party systems about your actions by sending events to a pointed Kafka topic or logs.

## Enable notifications

To enable notifications users have to set the `notifications.channels` parameter to one of the following values:

- `kafka`
- `logs`
- `kafka,logs`

Then, configure the Kafka topic with the proper parameter `messaging.kafka.notifications.*`.

We recommend providing that configuration through Helm values:

| Helm Value | Default | Description |
|---|---|---|
| `global.notifications.channels` | `empty` | Set value (e.g. `kafka` to enable notifications) |
| `global.messaging.kafka.topics.notifications` | (no default value) | Name of notifications topic |

**Note:** It is possible to automatically configure new topics by the Feature Store. Please review the `global.config.messaging.kafka` parameters described in Helm values.

## Events producer

The `core` service will send an event every time the user makes an API call to the backend. Based on that information, consumer service has knowledge about all actions happening in the Feature Store. That information can be used through an alert system to notify the operator about users' actions.

The following is an example triggered while creating a new project:

```
project = client.projects.create("test")
```

```
{
  "seq": "2022-07-20T08:15:46.861903-62d7b9b2d209b07c1d88f720",
  "method": "CreateProject",
  "timestamp": "2022-07-20T08:15:46.861903",
  "requestId": "fdb98051-c732-49cd-8e5b-8ba48fb325a2",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "id": "62d7b9b2d209b07c1d88f720",
  "projectName": "test1"
}
```

```
project = client.projects.get("test")
```

```
{
  "seq": "2022-10-03T16:57:58.300710-633af8600423a6148368033e",
  "method": "GetProject",
  "timestamp": "2022-10-03T16:57:58.30071",
  "requestId": "a02d7618-869f-4330-8b91-cc523d2d6ad1",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "id": "62d7b9b2d209b07c1d88f720",
  "projectName": "test"
}
```

```
project.delete()
```

```
{
  "seq": "2022-10-04T08:35:06.673156-633bd413708da32636a7cdbe",
  "method": "DeleteProject",
  "timestamp": "2022-10-04T08:35:06.673156",
  "requestId": "855981af-c82a-48d3-8b32-828f21e86c27",
  "userId": "8cf44396-1e80-4bd4-8d8d-c5e6befb4f2d",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "id": "62d7b9b2d209b07c1d88f720",
  "projectName": "test"
}
```


```
client.projects.list()
```


```
{
  "seq": "2022-10-04T08:37:35.166016-8cf44396-1e80-4bd4-8d8d-c5e6befb4f2d",
  "method": "ListProjects",
  "timestamp": "2022-10-04T08:37:35.166016",
  "requestId": "5996c73a-3179-4ad4-aec1-82a215606283",
  "userId": "8cf44396-1e80-4bd4-8d8d-c5e6befb4f2d",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)"
}
```


```
project.add_consumers(["dev@h2o.ai"]) # add permissions
```


```
{
  "seq": "2022-10-04T08:41:06.803045-633bd481708da32636a7cdc8",
  "method": "AddProjectPermission",
  "timestamp": "2022-10-04T08:41:06.803045",
  "requestId": "a628c9fd-f849-4b91-b764-88e11260ac4f",
  "userId": "8cf44396-1e80-4bd4-8d8d-c5e6befb4f2d",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "id": "633bd481708da32636a7cdc8",
  "projectName": "test3",
  "additionalData": {
    "permissionType": "Consumer",
    "users": "dev@h2o.ai"
  }
}
```


```
project.remove_consumers(["dev@h2o.ai"])
```


```
{
  "seq": "2022-10-04T08:42:50.347749-633bd481708da32636a7cdc8",
  "method": "RemoveProjectPermission",
  "timestamp": "2022-10-04T08:42:50.347749",
  "requestId": "ef174341-84ce-46e7-a3a7-4eb92e023fa4",
  "userId": "8cf44396-1e80-4bd4-8d8d-c5e6befb4f2d",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "id": "633bd481708da32636a7cdc8",
  "projectName": "test3",
  "additionalData": {
    "permissionType": "Consumer",
    "users": "dev@h2o.ai"
  }
}
```


```
project.description = "better description" # similar will be for other fields update
```


```
{
```

```
  "seq": "2022-10-04T08:45:46.111069-633bd481708da32636a7cdc8",
  "method": "UpdateProjectFields",
  "timestamp": "2022-10-04T08:45:46.111069",
  "requestId": "e6d7d6df-ef0d-4221-9536-dd9fb175ee6c",
  "userId": "8cf44396-1e80-4bd4-8d8d-c5e6befb4f2d",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "id": "633bd481708da32636a7cdc8",
  "projectName": "test3",
  "additionalData": {
    "updatedValue": "better description"
  }
}
```


```
fs = project.feature_sets.register(csv_schema, "fs_test")
```


```
{
  "seq": "2022-07-20T10:30:04.639687-62d7bd0c93f57739745041a5",
  "sourceRequest": "",
  "method": "OfflineFeatureSetRegister",
  "timestamp": "2022-07-20T10:30:04.639687",
  "requestId": "737ed314-bf29-4d01-aecd-3c4f1ecb21c2",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "id": "62d7bd0c93f57739745041a5",
  "projectName": "test1",
  "featureSetName": "fs_test",
  "additionalData": {
    "partitionBy": "time_travel_column_auto_generated",
    "timeTravelColumn": "",
    "primaryKey": "",
    "createdDateTime": "2022-07-20T08:30:04.446074",
    "owner": "dev@h2o.ai"
  }
}
```


```
fs = project.feature_sets.get("fs_test")
```


```
{
  "seq": "2022-10-04T09:08:02.048966-633af7b80423a61483680335",
  "sourceRequest": "",
  "method": "GetFeatureSet",
  "timestamp": "2022-10-04T09:08:02.048966",
  "requestId": "0709aeb8-e322-4c22-9bff-b16dbe4dba5b",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "id": "633af7b80423a61483680335",
  "projectName": "test_project",
  "featureSetName": "fs_test",
  "additionalData": {
    "featureSetVersion": "1.1"
  }
}
```


```
fs_new = fs.create_new_version(csv_schema,"new version")
```


```
{
  "seq": "2022-10-04T09:09:56.742789-633af7b80423a61483680335",
  "sourceRequest": "",
  "method": "CreateNewFeatureSetVersion",
```

```
  "timestamp": "2022-10-04T09:09:56.742789",
  "requestId": "0dd81fd4-8f54-48f6-b816-7a81fccdd377",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "id": "633af7b80423a61483680335",
  "projectName": "test_project_123",
  "featureSetName": "test_fs",
  "additionalData": {
    "updatedVersion": "2.0"
  }
}
```


```
fs.delete()
```


```
{
  "seq": "2022-10-04T09:31:57.332711-633af7b80423a61483680335",
  "sourceRequest": "",
  "method": "DeleteFeatureSet",
  "timestamp": "2022-10-04T09:31:57.332711",
  "requestId": "4f0c0f44-9b39-48aa-8695-7f3beb079645",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "id": "633af7b80423a61483680335",
  "projectName": "test_project_123",
  "featureSetName": "test_fs"
}
```


```
fs.description = "test description" # similar will be for other fields update
```


```
{
  "seq": "2022-10-04T09:15:55.695864-633af7b80423a61483680335",
  "sourceRequest": "",
  "method": "UpdateFeatureSetFields",
  "timestamp": "2022-10-04T09:15:55.695864",
  "requestId": "fe6a1129-c9d6-4ac9-9442-9b4f39cc35dc",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "id": "633af7b80423a61483680335",
  "projectName": "test_project_123",
  "featureSetName": "test_fs1",
  "additionalData": {
    "description": "test description"
  }
}

{
  "seq": "2022-10-04T09:15:55.924144-633af7b80423a61483680335",
  "sourceRequest": "",
  "method": "GetFeatureSetLastMinor",
  "timestamp": "2022-10-04T09:15:55.924144",
  "requestId": "4bc17a4f-4818-497f-956d-ae5fb75824a2",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "id": "633af7b80423a61483680335",
  "projectName": "test_project_123",
  "featureSetName": "test_fs1",
  "additionalData": {
```

```
    "featureSetVersion": "1.2"
  }
}
```


```
fs.list_versions()
```


```
{
  "seq": "2022-10-04T09:20:08.376487-17a91bc8-3733-4d40-af33-e729cce6823a",
  "method": "ListFeatureSetVersions",
  "timestamp": "2022-10-04T09:20:08.376487",
  "requestId": "c66fe245-e7a8-4c27-9272-63f264cd0e63",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)"
}
```


```
project.feature_sets.list()
```


```
{
  "seq": "2022-10-03T16:58:40.040700-17a91bc8-3733-4d40-af33-e729cce6823a",
  "method": "ListFeatureSets",
  "timestamp": "2022-10-03T16:58:40.0407",
  "requestId": "1cd48fff-ba6f-4602-bbb8-f22b8fa1cf82",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "additionalData": {
    "projectNames": "test_project2,test_project_123",
    "featureSetIds": "633af8750423a61483680341,633af7b80423a61483680335"
  }
}
```


```
feature = fs.features["id"]
feature.description = "test description" # similar will be for other fields update
```


```
{
  "seq": "2022-10-04T09:27:33.904475-633af7b80423a61483680335",
  "sourceRequest": "",
  "method": "UpdateFeatureFields",
  "timestamp": "2022-10-04T09:27:33.904475",
  "requestId": "cae1e009-5096-4c47-bf62-8f500ec9a7f1",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "id": "633af7b80423a61483680335",
  "projectName": "test_project_123",
  "featureSetName": "test_fs1",
  "additionalData": {
    "featureName": "id",
    "updatedField": "description",
    "updatedValue": "test_description"
  }
}
```


```
fs.add_consumers(["test@h2o.ai"])
```


```
{
  "seq": "2022-10-04T09:11:59.740232-633af7b80423a61483680335",
  "sourceRequest": "",
  "method": "AddFeatureSetPermission",
  "timestamp": "2022-10-04T09:11:59.740232",
  "requestId": "b159d492-ca02-4717-814e-16b494963b70",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
```

```
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "id": "633af7b80423a61483680335",
  "projectName": "test",
  "featureSetName": "test_fs1",
  "additionalData": {
    "permissionType": "Consumer",
    "users": "test@h2o.ai"
  }
}
```


```
fs.remove_consumers(["test@h2o.ai"])
```


```
{
  "seq": "2022-10-04T09:14:07.807106-633af7b80423a61483680335",
  "sourceRequest": "",
  "method": "RemoveFeatureSetPermission",
  "timestamp": "2022-10-04T09:14:07.807106",
  "requestId": "886835ee-bc45-4914-abdc-89317101df14",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "id": "633af7b80423a61483680335",
  "projectName": "test_project_123",
  "featureSetName": "test_fs1",
  "additionalData": {
    "permissionType": "Consumer",
    "users": "test@h2o.ai"
  }
}
```


```
fs.ingest(csv)
```


```
{
  "seq": "2022-07-20T10:34:17.366153-62d7bd0c93f57739745041a5",
  "sourceRequest": "",
  "method": "StartIngest",
  "timestamp": "2022-07-20T10:34:17.366153",
  "requestId": "301bb8e6-8e64-4d8e-92e6-96a436e29e89",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "id": "62d7bd0c93f57739745041a5",
  "projectName": "test1",
  "featureSetName": "fs_test",
  "additionalData": {
    "jobId": "62d7be0993f57739745041a9"
  }
}
```



```
{
  "seq": "2022-07-20T11:45:06.221177-62d7ce6305797005c7dadbad",
  "sourceRequest": "RawData(RawDataLocation(Csv(CSVFileSpec(s3a://feature-store-test-
data/smoke_test_data/training.csv,,,UnknownFieldSet(Map())),UnknownFieldSet(Map())))",
  "method": "EndIngest",
  "timestamp": "2022-07-20T11:45:06.221177",
  "id": "62d7ce6305797005c7dadbad",
  "ingestionCount": 3,
  "ingestionStartTime": "2022-07-20T09:44:40.635830",
  "ingestionEndTime": "2022-07-20T09:44:43.273830",
  "projectName": "test1",
```

```
  "featureSetName": "fs_test"
}
```

```
{
  "seq": "2022-10-03T16:55:16.215758-633af7c00423a61483680339",
  "method": "JobStatus",
  "timestamp": "2022-10-03T16:55:16.215758",
  "id": "633af7c00423a61483680339",
  "additionalData": {
    "eventId": "00a10969-f48f-4f3e-b125-6071f5c4633f"
  },
  "jobStatus": "Running"
}
```


```
fs.get_preview()
```


```
{
  "seq": "2022-10-03T17:13:19.862989-633af7b80423a61483680335",
  "method": "GetPreview",
  "timestamp": "2022-10-03T17:13:19.862989",
  "requestId": "94d0f5de-5c34-4197-bce3-8e2480c5ba76",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "id": "633af7b80423a61483680335",
  "projectName": "test",
  "featureSetName": "test_fs"
}
```


```
ref.as_spark_frame(spark_session)
```


```
{
  "seq": "2022-10-03T17:19:04.772158-633af7b80423a61483680335",
  "sourceRequest": "",
  "method": "RetrieveAsSpark",
  "timestamp": "2022-10-03T17:19:04.772158",
  "requestId": "8ad0f47f-a057-4e29-aeb6-d3e761421fd7",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "id": "633af7b80423a61483680335",
  "projectName": "test",
  "featureSetName": "test_fs"
}
```


```
ref.download()
```


```
{
  "seq": "2022-10-03T17:14:17.676281-17a91bc8-3733-4d40-af33-e729cce6823a",
  "method": "RetrieveAsLinks",
  "timestamp": "2022-10-03T17:14:17.676281",
  "requestId": "aaf0f2d9-7922-4c00-be28-bf7302d2160d",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "additionalData": {
    "jobId": "633afc1d0423a61483680347"
  }
}
```


```
ref = ingest.retrieve()
ref.download()
```

```
{
  "seq": "2022-10-04T09:41:23.219196-633be24c708da32636a7cdda",
  "sourceRequest": "",
  "method": "StartRetrieveJob",
  "timestamp": "2022-10-04T09:41:23.219196",
  "requestId": "27568494-7765-494e-9c83-a19ae73c7ddb",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "id": "633be24c708da32636a7cdda",
  "projectName": "test_project",
  "featureSetName": "test_fs",
  "additionalData": {
    "jobId": "633be3a3708da32636a7cde8"
  }
}
```


```
ingest.revert()
```


```
{
  "seq": "2022-10-04T09:43:51.959670-633be24c708da32636a7cdda",
  "sourceRequest": "",
  "method": "StartRevertIngest",
  "timestamp": "2022-10-04T09:43:51.95967",
  "requestId": "7ec74e9e-660f-40ae-be3a-0a54fc3b9e7a",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)",
  "id": "633be24c708da32636a7cdda",
  "projectName": "test_project",
  "featureSetName": "test_fs",
  "additionalData": {
    "jobId": "633be437708da32636a7cdeb"
  }
}
```


```
client.auth.login()
```


```
{
  "seq": "2022-10-03T16:42:36.891337-17a91bc8-3733-4d40-af33-e729cce6823a",
  "method": "UserLogin",
  "timestamp": "2022-10-03T16:42:36.891337",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)"
}
```


```
client.auth.logout()
```


```
{
  "seq": "2022-10-03T16:43:51.724728-17a91bc8-3733-4d40-af33-e729cce6823a",
  "method": "UserLogout",
  "timestamp": "2022-10-03T16:43:51.724728",
  "requestId": "bf4b8a63-1ee1-416e-a347-2c04c7af485b",
  "userId": "17a91bc8-3733-4d40-af33-e729cce6823a",
  "userAgent": "feature-store-py-cli/SUBST_FS_VERSION grpc-python/1.43.0 grpc-
c/21.0.0 (osx; chttp2)"
}
```

### Consume events

All events sent by the `core` service are in JSON format. The consumers need to pull the data from the topic specified in the configuration.

Kafka has a command line consumer that dumps out messages to standard output.

```
$ kafka-console-consumer.sh --bootstrap-server localhost:9094 --topic notifications --from-beginning

{
 "seq": String (Surrogate Key: Timestamp-id),
 "sourceRequest": String (in case of a feature set, it contains be data source domains, for ingest it contain
 "method": String (name of the action in Feature Store),
 "timestamp": LocalDateTime,
 "requestId": String*,
 "userId": String*,
 "userAgent": String*,
 "id": String* (project or feature set id),
 "ingestionCount": Long* (only available whit ingest statistics),
 "ingestionStartTime": String*,
 "ingestionEndTime": String*,
 "projectName": String*,
 "featureSetName": String*,
 "additionalData": {  }*
}
```

Keys in `additionalData` object are:

- `jobId`
- `permissionType`
- `users`
- `featureName`
- `updatedField`
- `updatedValue`
- `message`
- `featureSetVersion`
- `updatedVersion`
- `projectNames`
- `featureSetIds`
- `eventId`
- `description`
- `customData`
- `accessModifier`
- `featureSetType`
- `applicationName`
- `deprecated`
- `dataSourceDomains`
- `tags`
- `processInterval`
- `processIntervalUnit`
- `flow`
- `featureSetType`
- `applicationId`
- `approved`
- `notes`
- `onlineNamespace`
- `connectionType`
- `topic`
- `ttlOffline`
- `ttlOfflineInterval`
- `ttlOnline`

- `ttlOnlineInterval`
- `featureId`
- `status`
- `dataType`
- `importance`
- `classifiers`
- `special`

and values are strings.

The `method` field can have one of the following values:

- `CreateProject`
- `DeleteProject`
- `ListProjects`
- `GetProject`
- `AddProjectPermission`
- `RemoveProjectPermission`
- `OfflineFeatureSetRegister`
- `CreateNewFeatureSetVersion`
- `DeleteFeatureSet`
- `AddFeatureSetPermission`
- `RemoveFeatureSetPermission`
- `UpdateFeatureFields`
- `GetFeatureSetLastMinor`
- `GetFeatureSet`
- `ListFeatureSetVersions`
- `ListFeatureSets`
- `UpdateFeatureSetFields`
- `StartIngest`
- `StartRevertIngest`
- `StartRetrieveJob`
- `RetrieveAsSpark`
- `GetPreview`
- `RetrieveAsLinks`
- `JobStatus`
- `UserLogin`
- `UserLogout`
- `EndIngest`
- `UpdateProjectFields`

This value shows which action was triggered by the user.

**Note:** Fields are only available in JSON when not null. For example, the `id` field, depending on `method`, will be `ProjectId` when using Project API.

# Custom CA certificates

It's possible to add own CA certificates to components' trust store. To do that, config-map with CA bundle has to be present in kubernetes cluster.

## CA certificates bundle

User can load multiple certificates into Feature Store components. All certificates have to be bundled in one config item and store in k8s config map, eg:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: pem-ca-bundle
data:
  rootCA.pem: |
    -----BEGIN CERTIFICATE-----
    FjAUBgNVBAsMDUZlYXR1cmUtU3RvcmUxDzANBgNVBAMMBlJvb3RDQTCCASIwDQYJ
    ...
    KoZIhvcNAQEBBQADggEPADCCAQoCggEBAMaL6==
    -----END CERTIFICATE-----
    -----BEGIN CERTIFICATE-----
    MIIDTjCCAjYCCQDjradeTuANSjANBgkqhkiG9w0BAQsFADBpMQswCQYDVQQGEwJQ
    ...
    ftAwrrWU2poHRkQQY5CxatxMPgSxievLCwWq7qnzHpXtbw==
    -----END CERTIFICATE-----
    -----BEGIN CERTIFICATE-----
    TDEPMA0GA1UECAwGS3Jha293MQ8wDQYDVQQHDAZLcmFrb3cxDzANBgNVBAoMBkgy
    ...
    Jvo2e6md7u/SB0Rgy6TCbohRVmoqCbuiTfqjJpaLhNVFu==
    -----END CERTIFICATE-----
    ...
```

## Configure Feature Store

To add own CA certificates into Feature Store users have point the config map with the certificate through Helm values:

| Helm Value | Default | Description |
|---|---|---|
| `global.extraTrustedCertificates.configMapName` | `empty` | ConfigMap name with CA certificates bundle |
| `global.extraTrustedCertificates.caBundleKey` | `ca_bundle.pem` | ConfigMap key name with certificates list |

# H2O GPTE Integration

## Configuration possibility

The Feature Store can be integrated with H2O GPTe using presented configuration:

- `global.cloud.h2oGPTe.enabled`
- `global.cloud.h2oGPTe.url`
- `global.cloud.h2oGPTe.apiKey`
- `global.cloud.h2oGpteServiceName`

If the `global.cloud.h2oGPTe.url` is not provided, the Feature Store will attempt to retrieve the service address from service discovery by checking if service exists. In this case `global.cloud.h2oGpteServiceName` is used.

Regarding authentication, the Feature Store can use an API Key (`global.cloud.h2oGPTe.apiKey`) for the system account. In this case, this key will also be shared with the UI to facilitate chat service access. If the API key is not provided, both the Feature Store and H2O GPTe must use the same Identy Provider for the usage of Access Tokens.

# Logging

By default, all Feature Store services produce logs to `stdout` in JSON format.

## Log structure

We use the following JSON template:

```
{
  "time": {
    "$resolver": "timestamp",
    "pattern": {
      "format": "yyyy-MM-dd'T'HH:mm:ss.SS'Z'",
      "timeZone": "UTC"
    }
  },
  "level": {
    "$resolver": "level",
    "field": "name"
  },
  "message": {
    "$resolver": "message",
    "stringified": true
  },
  "error": {
    "$resolver": "exception",
    "field": "message",
    "stringified": true
  },
  "user": {
    "$resolver": "mdc",
    "key": "userId"
  },
  "grpc_server_method": {
    "$resolver": "mdc",
    "key": "grpcServerMethod"
  },
  "thread_id": {
    "$resolver": "thread",
    "field": "name"
  },
  "logger": {
    "$resolver": "logger",
    "field": "name"
  },
  "stacktrace": {
    "$resolver": "exception",
    "field": "stackTrace",
    "stackTrace": {
      "stringified": true
    }
  }
}
```

## Customize log format

It is s possible to provide a custom `log4j2` configuration. To overwrite the `log4j2.properties` file (after installation), edit default log4j config map - `{{ .Release.Name }}-log4j-config`

Also, it is possible to provide a custom config map with log4j2 configuration. Prepare the new `log4j2.properties` file

you want to use. For example:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: log4j-config
data:
  log4j2.properties: |
    monitorInterval=5

    # Root logger option
    rootLogger.level=INFO
    rootLogger.appenderRefs=stdout
    rootLogger.appenderRef.stdout.ref=STDOUT
    appenders=console

    # Direct log messages to stdout
    appender.console.type=Console
    appender.console.name=STDOUT
    appender.console.layout.type = PatternLayout
    appender.console.layout.pattern = %d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n
```

The next step is to configure Helm chart to use that config map. To override log4j configuration for all components, put name of custom config map to `global.log4jConfigMapOverride` value. To provide different config maps for different components put name of custom config maps to following values:

- for core pods - `core.log4jConfigMapOverride`
- for online store pods - `onlinestore.log4jConfigMapOverride`
- for spark operator pod - `sparkoperator.log4jConfigMapOverride`
- for spark driver/executor pods - `sparkoperator.config.spark.log4jConfigMapOverride`
- for telemetry pod - `telemetry.log4jConfigMapOverride`

### Use different file for log4j configuration

By defaults feature store are looking for `log4j2.properties` file in `/opt/h2oai/log4j` directory. If it is required to use different file instead (e.g. `log4j2.xml`), custom config map should contain that file and `JAVA_LOG4J_PROPERTIES` environment variable has to be updated:

- fore core pods:

```
core:
  env:
    JAVA_LOG4J_PROPERTIES: /opt/h2oai/log4j/log4j2.xml
```

- for online store pods:

```
onlinestore:
  env:
    JAVA_LOG4J_PROPERTIES: /opt/h2oai/log4j/log4j2.xml
```

- for sparkoperator store pods:

```
sparkoperator:
  env:
    JAVA_LOG4J_PROPERTIES: /opt/h2oai/log4j/log4j2.xml
```

- for spark driver/executor pods

```
sparkoperator:
  config:
    spark:
      extraOptions:
```

```
      - spark.driver.extraJavaOptions="-Dlog4j2.configurationFile=file:///opt/h2oai/log4j/log4j2.xml"
      - spark.executor.extraJavaOptions="-Dlog4j2.configurationFile=file:///opt/h2oai/log4j/log4j2.xml"
```

- for telemetry store pods:

```
telemetry:
  env: /opt/h2oai/log4j/log4j2.xml
```

# Testing

The subsequent processes are used to ensure the correctness of Feature Store deployments.

1. Deploy Feature Store with Helm
2. Generate User Personal Access Token
3. Create Kubernetes Secret
4. Helm Test

## Deploy Feature Store with Helm

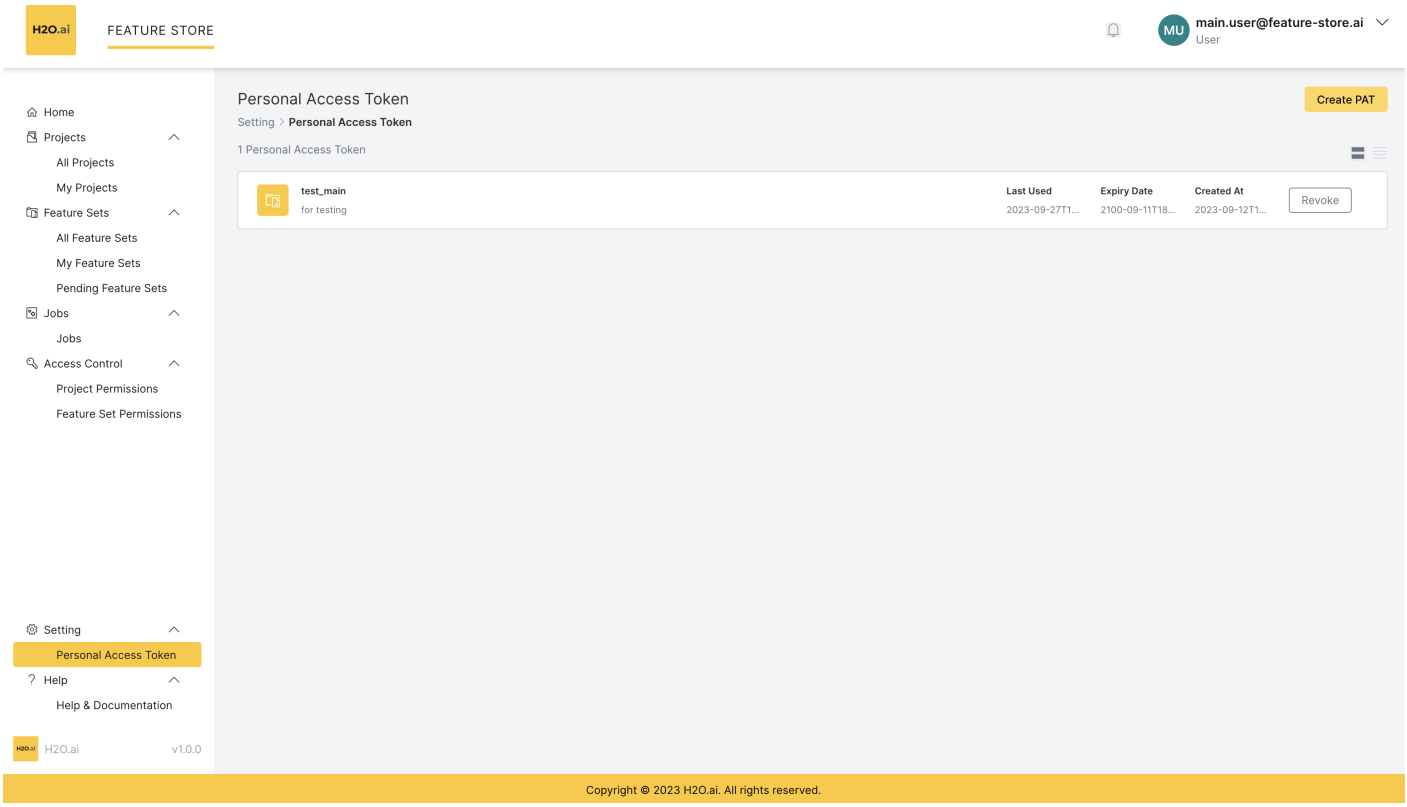
Helm is the official way to install the Feature Store services. See Kubernetes Helm Charts for more details.

## Generate Personal Access Token

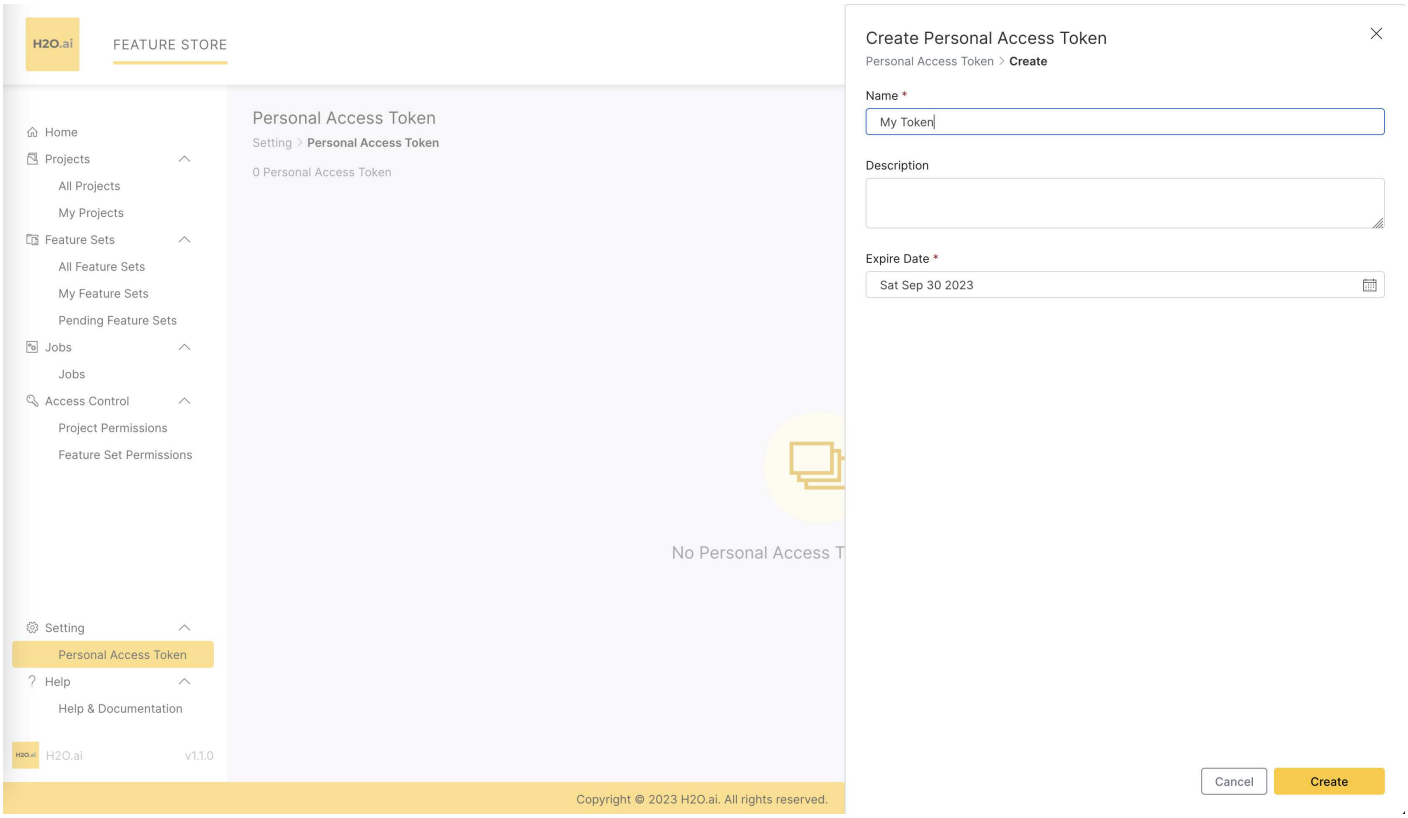
For `helm tests` execution, the **personal access token** is required.

### From UI

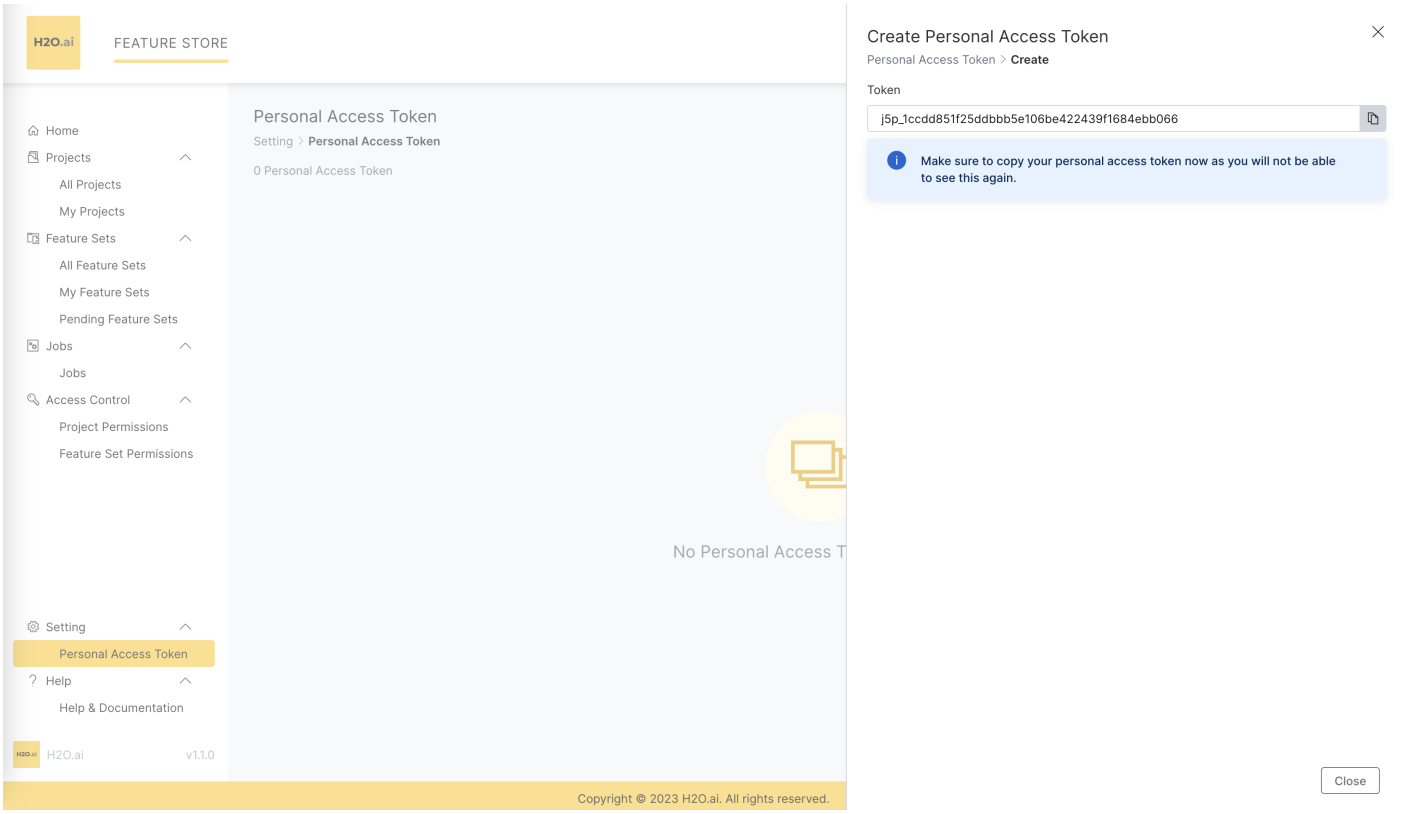
- Click **Create PAT** button (at right top corner)




- Fill up the fields which requires and click **Create** button




- Copy the token string




### From Python client

```
token_str = client.auth.pats.generate(name="background_jobs", description="some description", expiry_date="<d
```

### Create Kubernetes Secret

Prior to running the **Helm Test**, the previously created personal access token should be passed as "Kubernetes Secret" within your cluster.

```
kubectl create secret generic <<secret-name>> --from-literal=<<secret-key-name>>="<<personal-access-token>>"
```

Based on the helm values that you provided, the **secret name** and **key name** should be passed above.

```
## @param test.userAuthTokenSecret  Kubernetes secret name containing Test User's Personal Access Token used
## @param test.userAuthTokenSecretKey Kubernetes secret name key containing Test User's Personal Access Token
```

### Helm Test

`helm test` will start the test pod in your cluster, which conducts simple installation tests.

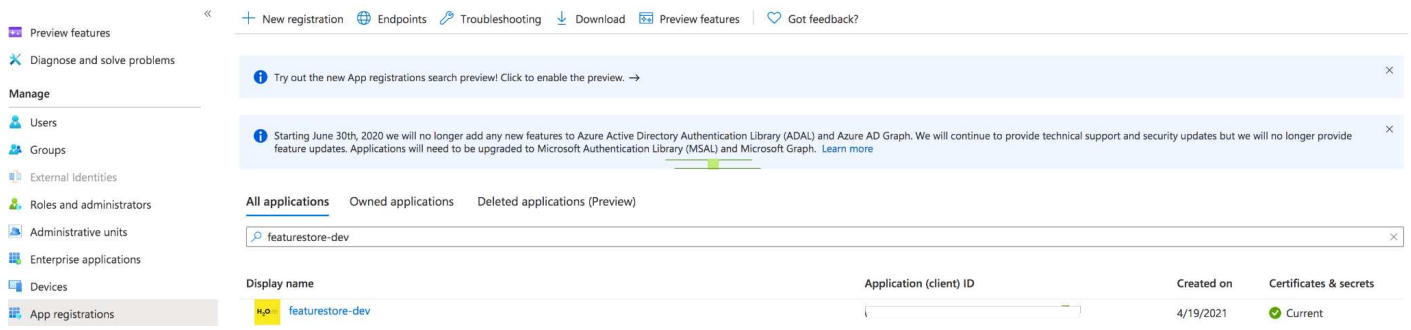
```
helm test <<release-name>> --timeout 15m0s
```

After a successful test run, this **test pod will be deleted** in accordance with the deletion policy.

# Configuration of Azure Active Directory client

## Register your application on portal.azure.com

- Click **App registrations** on the left panel and select + to setup a new registration
- Select **single tenant** as the account for this organization
- Give a name and a redirect uri



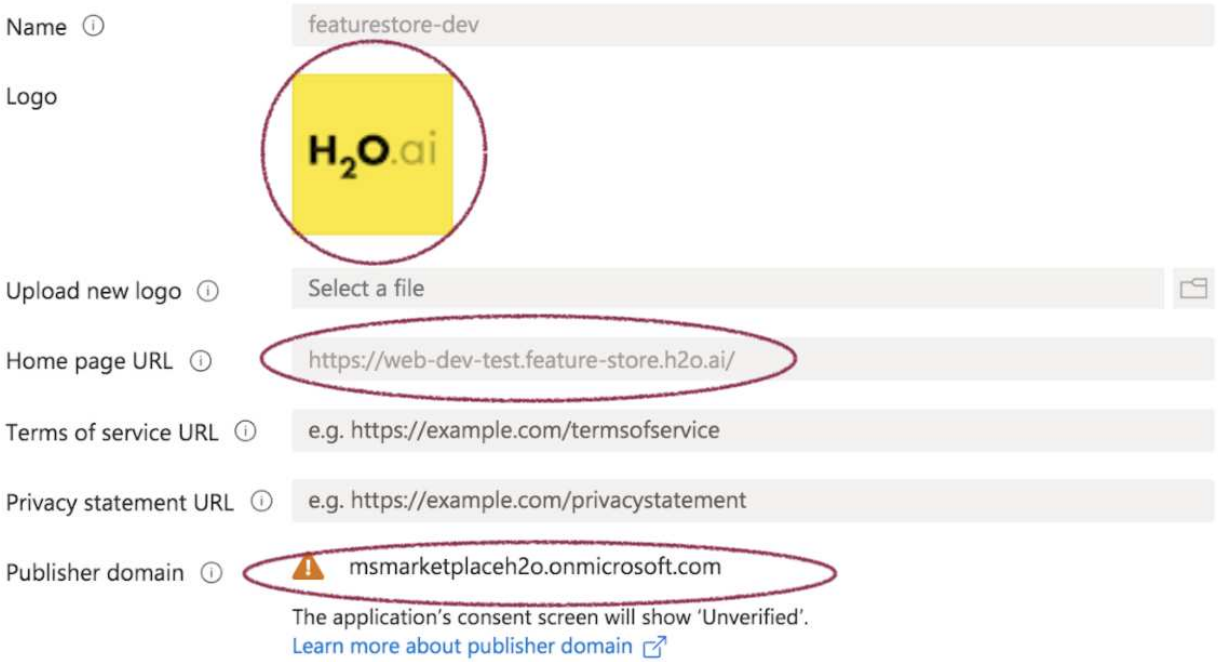

**Redirect uri:**

https://web-dev-test.feature-store.h2o.ai/Callback

## Configuration of application object properties

### Branding

Branding is where the logo, home page URL, and the publisher domain are configured.

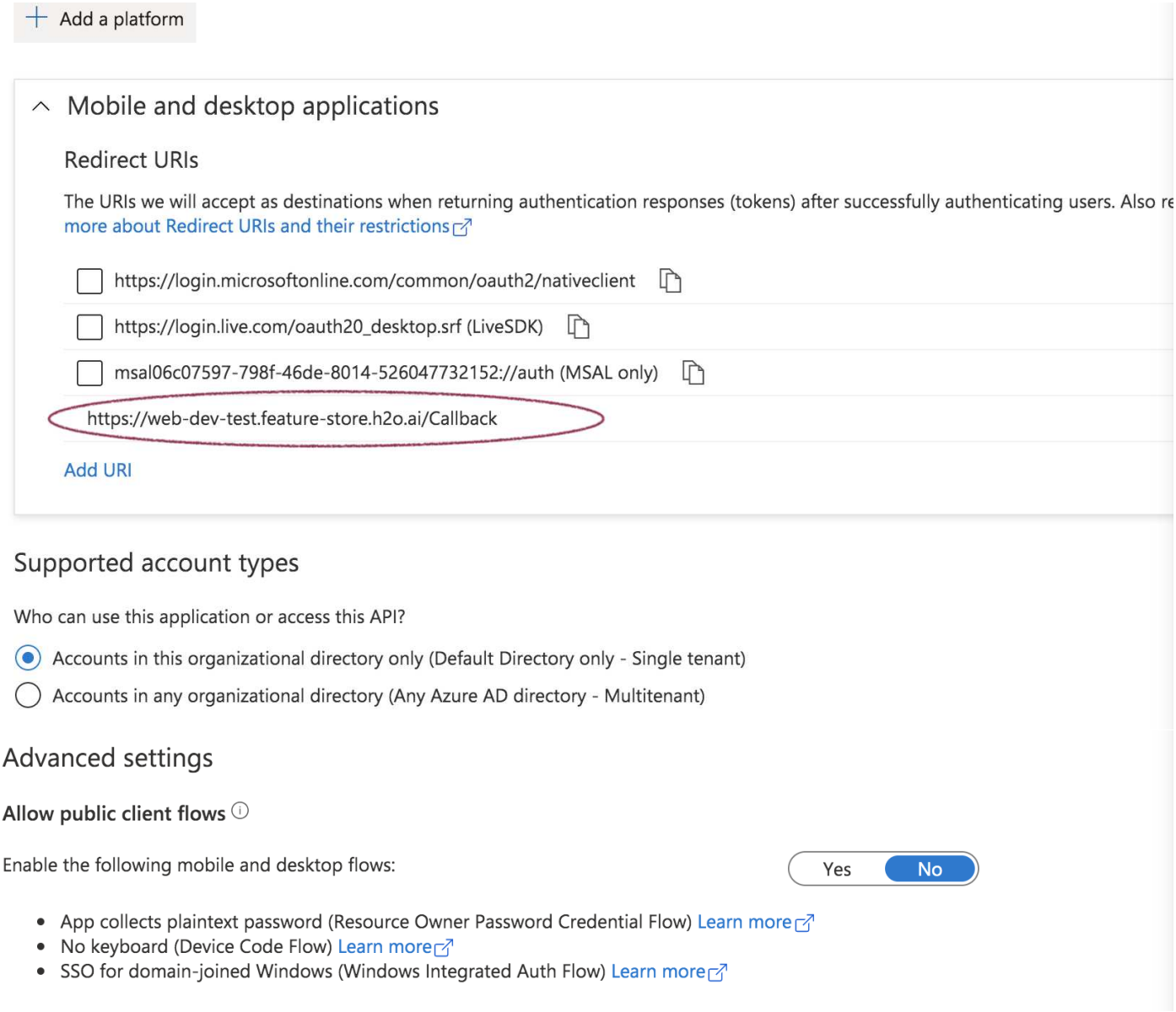



### Authentication

This is where you authenticate users for the application.

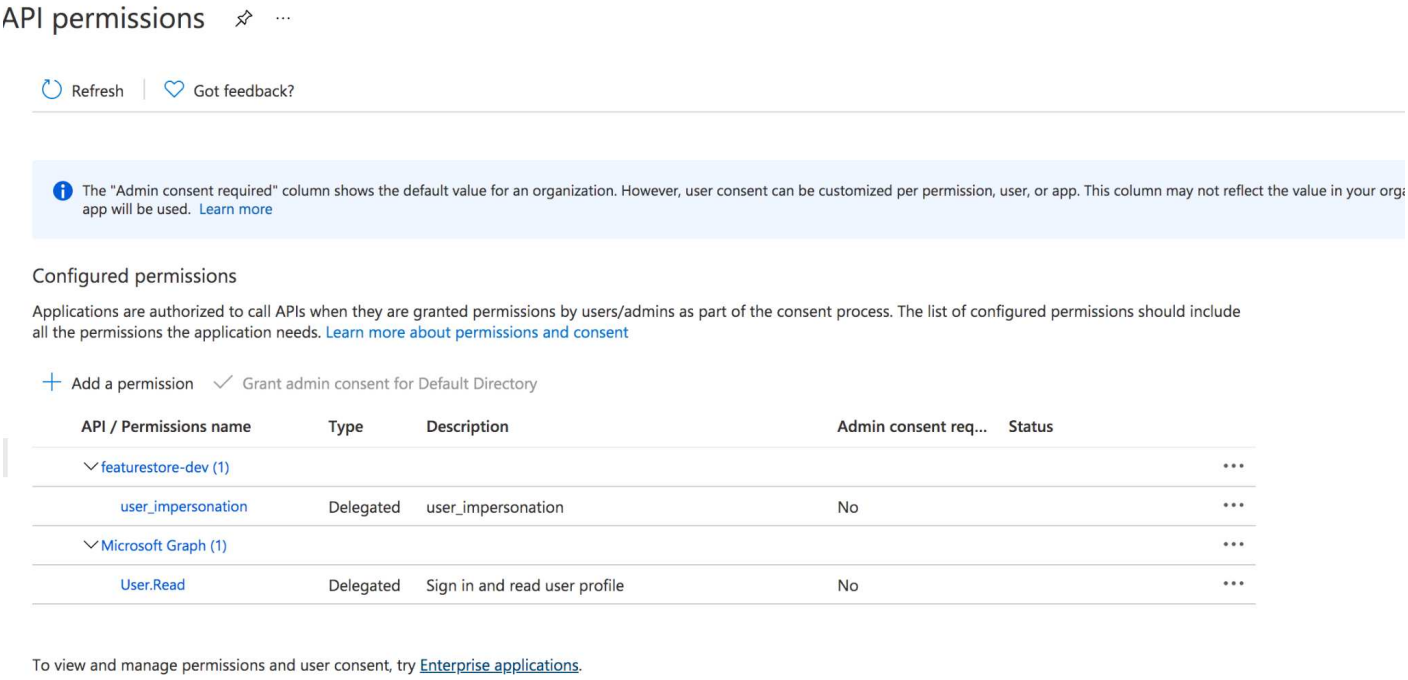
- Click the platform and select **mobile and web applications**
- Add a **redirect uri** (this is where the response token will be redirected to after authentication is complete)
- Select **single tenant**

- Set public clients to **"No"** since public clients cannot keep secrets confidential



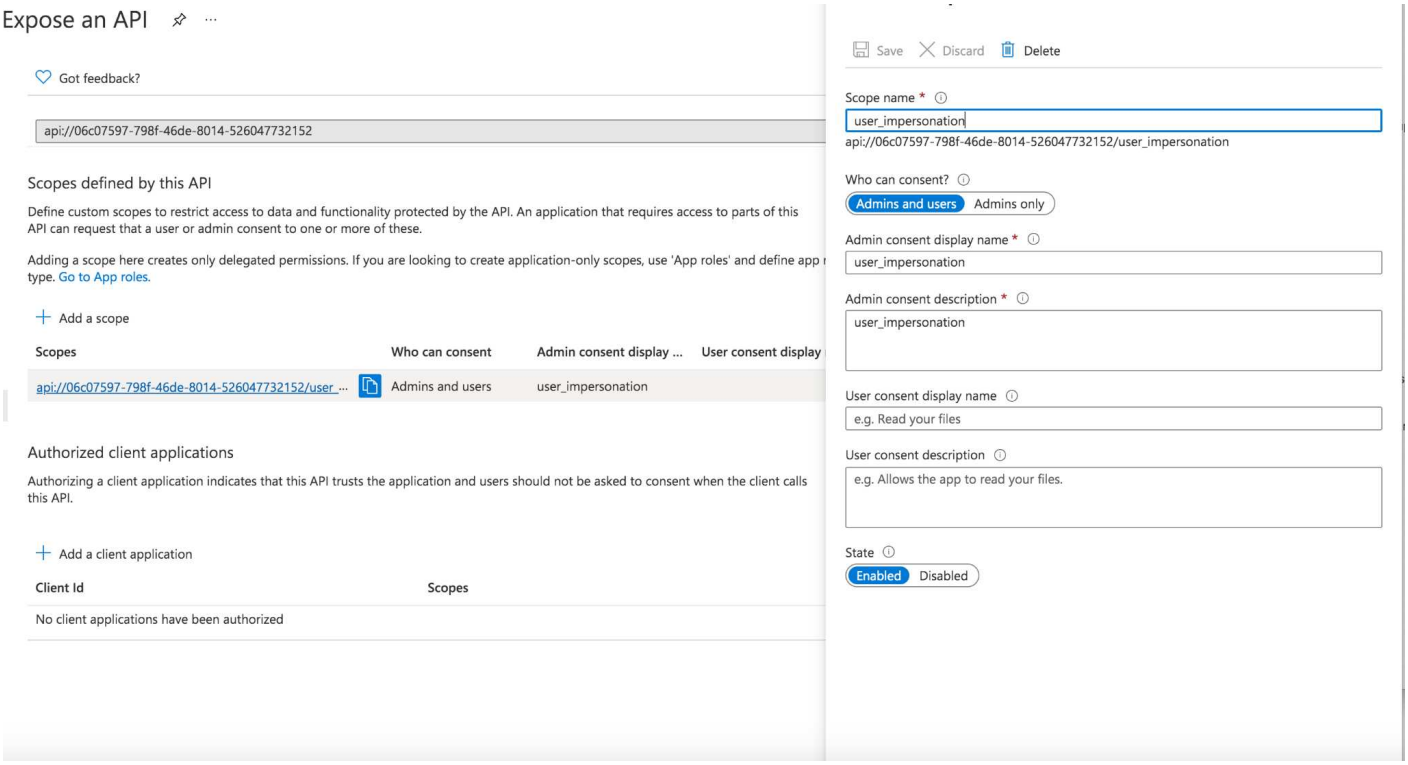

### API permissions

Add permissions to Microsoft Graph and the application (i.e., "featurestore-dev").



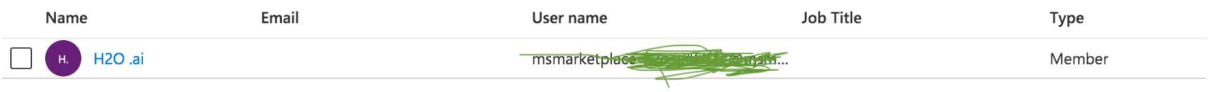

### Expose an API

Add a scope with consent for “admin and users” and name it “user_impersonation”.




### Owners

Add owners who are authorized to manage this registration.




That's it!

# Configuration of Keycloak for PAT exchange

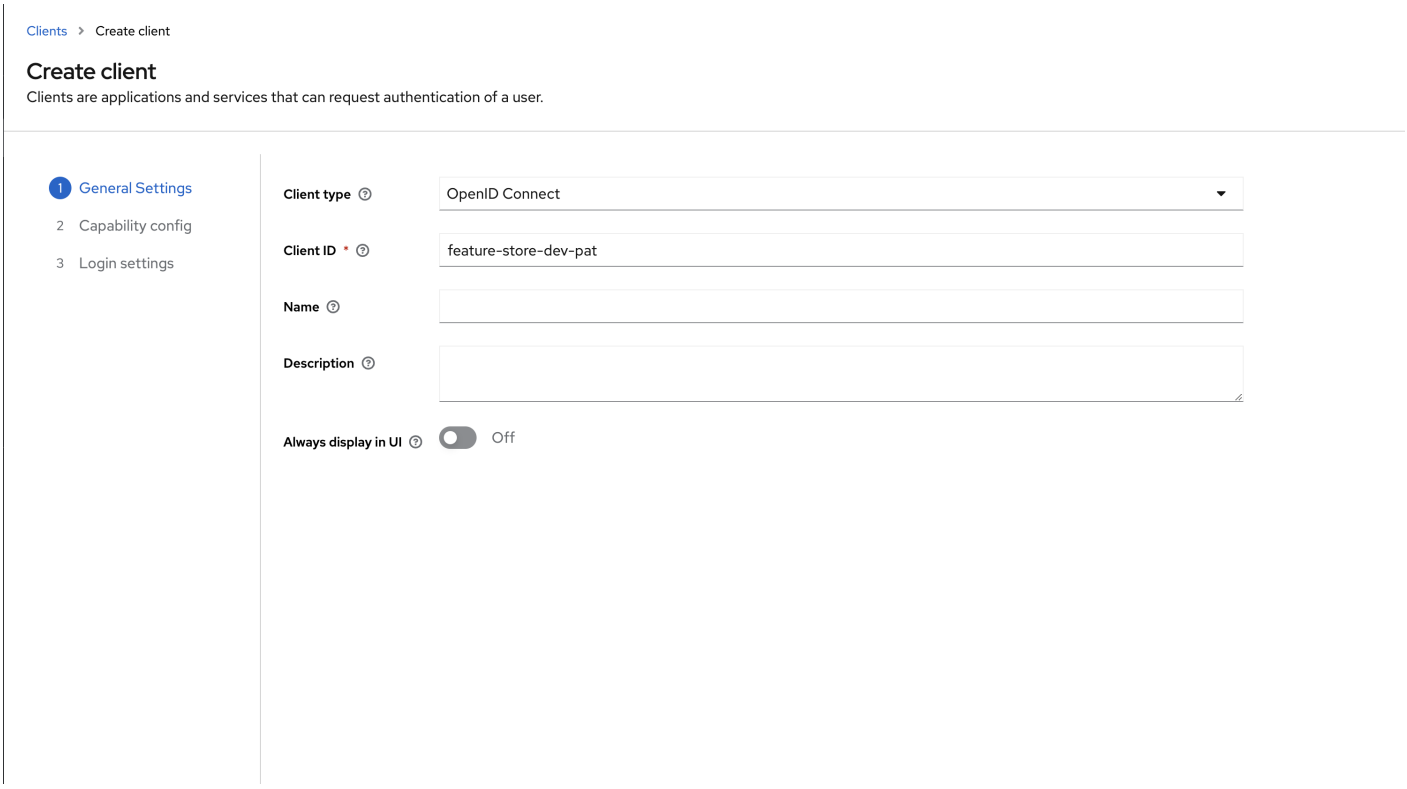
## Introduction

The configuration below is required when using the functionality of exchanging Personal Access Tokens (PAT) for access tokens, for example, when using H2O Drive.

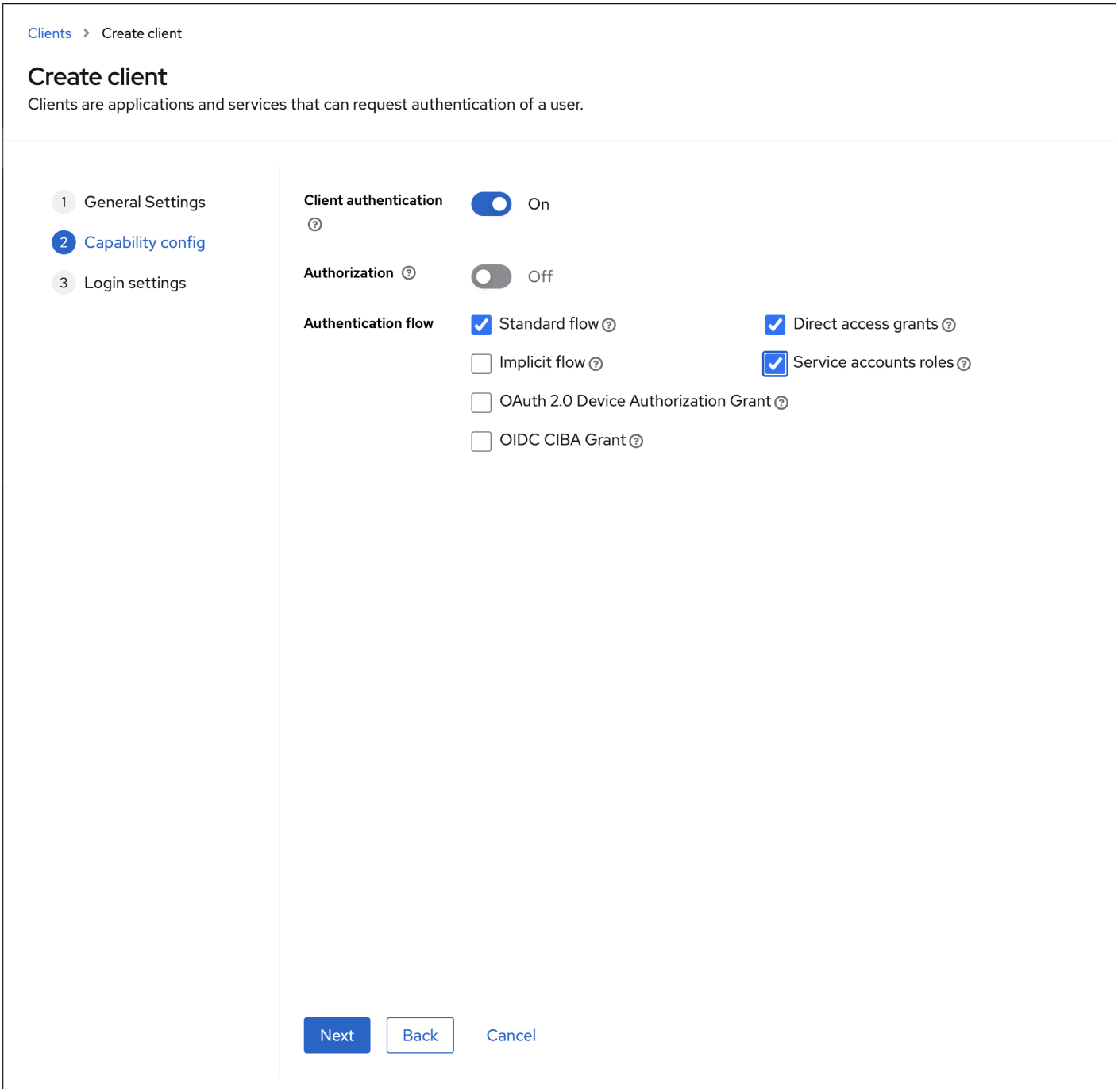
When user is authenticated via Feature Store personal access token, exchange between this PAT and platform access token is required in case interacting with cloud components (Drive/GTPe).

## Register new client in the realm

- Select realm from drop-down
- Click **Clients** on the left panel and click **Create client**
- Select **OpenID Connect** in **Client type**
- Provide **Client ID** for example feature-store-pat
- Provide **Name** for example feature-store-pat
- Click **Next**



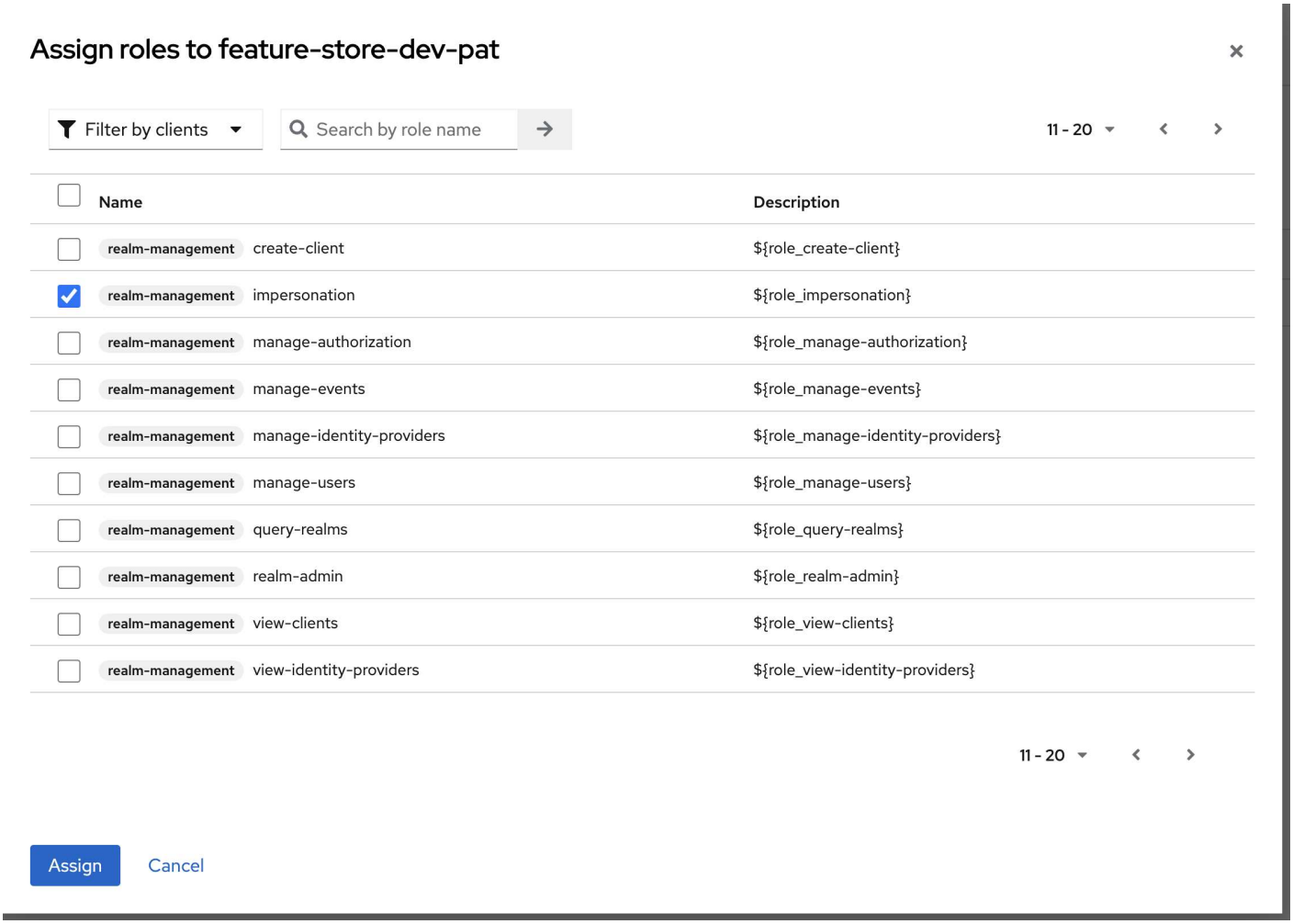

- Select **Client authentication**
- Select **Standard flow** , **Direct access grants**, **Service accounts roles**
- Click **Next**




- Select **Client authentication**
- Select **Standard flow** , **Direct access grants**, **Service accounts roles**
- Click **Next**

After creating the client, click on tab **Service accounts roles** and assing role.

- Select **impersonation** role




## Deployment

Please start keycloak with parameter: `-Dkc.features=token-exchange` or `KC_FEATURES=token-exchange`

## Destroy the stack

```
helm uninstall feature-store
```

# Snowflake prerequisites

## Steps

1. Create new user or use existing one (this user will be used for reading data to spark client)
2. Create new role, for example as: `CREATE ROLE IF NOT EXISTS FS_READ_ONLY_ROLE`
3. Grant usage to database that will be used to store feature sets for example: `GRANT USAGE ON DATABASE <DB_NAME> TO ROLE <ROLE_NAME>`
4. Grant usage to schema that will be created in database `GRANT USAGE ON SCHEMA <SCHEMA_NAME> TO ROLE <ROLE_NAME>`
5. Grant role to database `GRANT SELECT ON FUTURE VIEWS IN DATABASE <DB_NAME> TO ROLE <ROLE_NAME>`
6. Grant role to user `GRANT ROLE <ROLE_NAME> TO USER <USER_NAME>`

Use created user in helm values:

- `global.storage.offline.snowFlake.readOnlyViewSparkUser`
- `global.storage.offline.snowFlake.readOnlyViewSparkPassword`.

# Credentials configuration

To be able to read data from different data sources, you need to pass credentials either as a parameter to specific methods or via environmental variables.

## Specifying using environmental variable

### AWS S3

- S3_ACCESS_KEY
- S3_SECRET_KEY
- S3_REGION

**Note:** An optional parameter, `S3_ROLE_ARN`, can be specified. If specified, an AWS IAM role that delegates access to the bucket will be used.

Without an environmental variable or AWS credential, you are still able to access public S3 data.

### Minio

- S3_ACCESS_KEY
- S3_SECRET_KEY
- S3_REGION
- S3_ENDPOINT

**Note:** An optional parameter, `S3_ROLE_ARN` , can be specified. If specified, a Minio IAM role that delegates access to the bucket will be used.

`S3_ENDPOINT` should be provided so that Feature Store can read from the corresponding Minio server.

### JDBC Postgres

- JDBC_POSTGRES_USER
- JDBC_POSTGRES_PASSWORD

### JDBC Teradata

- JDBC_TERADATA_USER
- JDBC_TERADATA_PASSWORD

### Azure credentials

Feature Store provides three variants for providing Azure Credentials

### Azure name and key credentials

- `AZURE_ACCOUNT_NAME` is the name of the Azure storage account where the data source is stored.
- `AZURE_ACCOUNT_KEY` is the key for the Azure storage account where the data source is stored.

### Azure SAS credentials

- `AZURE_ACCOUNT_NAME` is the name of the Azure storage account where the data source is stored.
- `AZURE_SAS_TOKEN` is the Shared Access Signature (SAS) token for the Azure storage account. It grants restricted access to Azure Storage resources. You can use this form of authentication if provided with a SAS.

### Azure principal credentials

- `AZURE_ACCOUNT_NAME` is the name of the Azure storage account where the data source is stored.
- `AZURE_SP_CLIENT_ID` is the client ID for an Azure Service Principal. It is used to identify and authenticate the Service Principal when it requests access to Azure resources.
- `AZURE_SP_TENANT_ID` is the tenant ID for the Azure Active Directory tenant associated with the Service Principal.
- `AZURE_SP_SECRET` is the client secret for an Azure Service Principal.

#### S3 credentials

- S3_ACCESS_KEY is the Access Key ID for an Amazon S3 (Simple Storage Service) bucket.
- S3_SECRET_KEY is the Secret Access Key for the S3 bucket.
- S3_REGION is the AWS region where the S3 bucket is located.

#### Snowflake credentials

Feature Store provides two variants for providing Snowflake Credentials

#### Snowflake user and password credentials

- SNOWFLAKE_USER is the username for accessing the Snowflake database.
- SNOWFLAKE_PASSWORD is the password for the corresponding user account to authenticate the user when logging in to the Snowflake database.

#### Snowflake key pair credentials

- SNOWFLAKE_USER is the username for accessing the Snowflake database.
- SNOWFLAKE_PRIVATE_KEY_FILE is the location of private key pem file on users local machine from where private key is read and sent to Snowflake where is checked against public key part that's stored in Snowflake.
- PRIVATE_KEY_PASSPHRASE in the case the private key pem file is encrypted a passphrase is needed.

#### Teradata credentials

- USER is the username for accessing the Teradata database.
- PASSWORD is the password for the corresponding user account to authenticate the user when logging in to the Teradata database.

#### Postgres credentials

- USER is the username for accessing the PostgreSQL database.
- PASSWORD is the password for the corresponding user account to authenticate the user when logging in to the PostgreSQL database.

#### GCP credentials

- GOOGLE_APPLICATION_CREDENTIALS environment variable will be evaluated and the corresponding credentials file will be utilized by default
- Alternatively, user can create and use a GcpCredentials instance by specifying his/her own file path pointing to stored GCP credentials file.

#### Python

```
from featurestore import *
credentials = GcpCredentials.from_file(local_file_path)
```

### Passing credentials as a parameters

#### Python

```
from featurestore import *
credentials = AzureKeyCredentials(AZURE_ACCOUNT_NAME, AZURE_ACCOUNT_KEY)
credentials = AzureSasCredentials(AZURE_ACCOUNT_NAME, AZURE_SAS_TOKEN)
credentials = AzurePrincipalCredentials(AZURE_ACCOUNT_NAME, AZURE_SP_CLIENT_ID, AZURE_SP_TENANT_ID, AZURE_SP_
credentials = S3Credentials(S3_ACCESS_KEY, S3_SECRET_KEY, S3_REGION)
credentials = SnowflakeCredentials(SNOWFLAKE_USER, SNOWFLAKE_PASSWORD)
credentails = SnowflakeKeyPairCredentials(SNOWFLAKE_USER, SNOWFLAKE_PRIVATE_KEY_FILE, PRIVATE_KEY_PASSPHRASE)
credentials = TeradataCredentials(USER, PASSWORD)
credentials = PostgresCredentials(USER, PASSWORD)
```

### Passing secrets to environment variables in Databricks Notebook

You can make use of Databricks `dbutils` package to inject secrets into environment variables.

The following example shows passing an Azure Storage Account Key from Databricks Secret Vault into the respective environment variable as required by Feature Store.

```
import os
os.environ["AZURE_ACCOUNT_KEY"] = dbutils.secrets.get("<scope_name>", "<key_name>")
```

# Starting the client

### Python

Once your Python environment is ready, run:

```
from featurestore import Client
client = Client(url, secure=False, root_certificates=None, config=config)
```

or

```
from featurestore import Client
with Client(url, secure=False, root_certificates=None, config=config) as client:
...
```

where:

- `url` - the endpoint address of the Feature Store Server as a string (usually in `ip:port` format).
- `secure` - turn on secure connection for Feature Store API. If you run Feature Store behind nginx-ingress (which requires tls connection) make sure the `secure` flag is set to `True`. Client may also require root certificates.
- `root_certificates` - root certificates file location as a string or None to retrieve them from a default location chosen by gRPC runtime.
- `config` - Additional client configuration. If not specified, defaults are used.

The following API can be used to enable or disable interactive logging. Logging is enabled by default.

```
client.show_progress(False)
```

**Note:** It's good practice to close the connection after all action has proceeded. You should call `client.close()` or use the context manager.

## Client configuration

We can pass a `config` to the client constructor. The following examples show how to create the configuration and explain what options can be specified.

### Python

```
config = ClientConfig(wait_for_backend=True, timeout=, log_level=INFO)
```

- `wait_for_backend` - if `False`, client does not wait for the Feature Store Backend to be ready.
- `timeout` - client-side timeout in seconds to terminate long waiting grpc calls.
- `log_level` - Logging level to be used on the Python client. Supported values are `CRITICAL`, `ERROR`, `WARNING`, `INFO` and `DEBUG`.

**Note:** The client configuration is stored by default in the user's home directory. You can change this location by setting the `FEATURESTORE_USER_CONFIG` environmental variable to the desired location before starting the client.

# Obtaining version

Both client and server version is printed out after client is created to standard output.

Versions can also be obtained by calling the following method:

### Python

```
client.get_version()
```

# Open Web UI

This method opens the Feature Store Web UI.

### Python

```
client.open_website()
```

## Default naming rules

Feature Store is configured to adhere to the following restrictions on setting names for a project or a feature set:

- name must be between 3 and 30 characters long.
- name can only use lowercase letters, numbers, and special character "_" (underscore).
- name must begin with a letter or a number (not underscore).
- each underscore must be preceded and followed by a letter or a number.
- name cannot have spaces.
- project name must be unique across all projects.
- feature set name must be unique across all feature sets within a project.

# Authentication

Feature Store CLI provides 3 forms of authentication:

- Access token from external environment
- Refresh token from identity provider
- Personal Access Tokens (PATs)

All authentication-related methods can be called on the `auth` object on the `client` object (e.g., `client.auth.logout()`).

You can also get the currently logged-in user:

**Python**

```
client.auth.get_active_user()
```

## Access token from external environment

If you are running Feature Store in an environment which already takes care of the client authentication and makes access tokens available, you need to implement a method which returns the access token from the environment and passes it to `client.set_obtain_access_token_method`. This is the same for H2O Wave.

This ensures that during each call, Feature Store obtains a valid access token from the external environment and uses it for authentication.

## Refresh token from identity provider

First, we need to obtain the refresh token. We can achieve this by executing the login method.

**Python**

```
client.auth.login()
```

This method will try to open the returned URL in the browser (if this fails, the user has to do this manually) and wait for the refresh token. Returned refresh tokens will be saved into the client's configuration file. The client configuration file is stored in your home directory under the name `.featurestore.config`. The format of the file is `key=value`. If you wish, you can also set the token in the configuration file by using key `token` directly.

You won't be asked for the authentication again until this token expires.

## Personal access tokens (PATs)

In order to create a personal access token, you first need to be logged-in via one of the previously mentioned methods.

Once logged-in, you can create a personal access token:

**Python**

```
token_str = client.auth.pats.generate(name="background_jobs", description="some description", expiry_date="",
```

**Explanation of the parameters:**

**Python**

- `expiry_date` is optional. When provided, it should be in the format *dd/MM/yyyy*. Tokens without expiry date will get an expiry date according to maximal allowed token duration which is a parameter controlled by a Feature Store administrator. To find out its actual value, call *client.auth.pats.maximum_allowed_token_duration*.
- `timezone` is optional. If provided, the provided timezone overrides the system timezone of CLI environment.

This call returns the textual representation of the token. It is not possible to obtain the textual representation of the token again, so save it in a secure location.

You can now **use this token for authentication**:

**Python**

```
client.auth.set_auth_token(token_str)
```

You can **list existing token objects**:

The `query` argument is optional and specifies which access tokens should be returned. By default, no filtering options are specified. To filter tokens by name or description please use `query` parameter.

**Python**

```
client.auth.pats.list(query = None)
```

You can **obtain a particular token object**:

**Python**

```
token = client.auth.pats.get(token_id)
```

**Note:** Token id is different from token name.

You can **revoke the token**:

**Python**

```
token.revoke()
```

**Note:** The Feature Store admin can configure the `max.pat.number.per.user` option to limit the number of personal access tokens one user can have at a single time.

# Permissions

Permissions determine the level of access that a user has to various components of the Feature Store. For example, depending on the level of permission granted, a user may be authorized to edit feature sets, while another user with limited view-only permission can only observe the feature set.

## Levels of permission

Feature Store has five levels of permission:

- Owner
- Editor
- Consumer
- Sensitive consumer
- Viewer

Additionally, Feature Store also has the concept of an *admin* account. An admin is any user with the admin role specified in their identity provider. Admin users can perform additional management tasks.

**Note:** The name of the claim storing the roles and name of admin role is configurable during Feature Store deployment.

### Owner

#### Owner permission for a project

You become the owner by creating a project. As the owner, you can remove the project and assign the owner, editor, consumer, sensitive consumer or viewer permission levels to other users. If you are the **project** owner, you are automatically granted owner permissions to all the **feature sets** within that project.

#### Owner permission for feature sets

You become the owner by creating a feature set. As the owner, you can remove the feature set and assign the owner, editor, consumer, sensitive consumer or viewer permission levels to other users.

**Note:** As the owner, you have all the other permissions.

- Editor
- Sensitive consumer
- Consumer
- Viewer

### Editor

#### Editor permission for a project

If you have editor permission for a project in the Feature Store, you are authorized to update the project's metadata and register new feature sets within the project. As the editor of the project, you are automatically granted owner permission for all feature sets associated with the project.

#### Editor permission for feature sets

If you have editor permission for a feature set, you are authorized to update the feature set's metadata and call `ingest` on the feature set.

**Note:** As an editor, you also have the following permissions,

- Sensitive consumer
- Consumer
- Viewer

### Sensitive consumer

#### Sensitive consumer permission for a project

If you have sensitive consumer permission for a project in the Feature Store, you are authorized to list or obtain a feature set from the project. As the sensitive consumer of the project, you are automatically granted sensitive consumer permission for all feature sets associated with the project.

#### Sensitive consumer permission for feature sets

If you have sensitive consumer permission for a feature set, you are authorized to call `retrieve` on the feature set. The retrieved data contains data in its original, unmasked variant (raw data).

**Note:** As an sensitive consumer, you also have the following permissions,

- Consumer
- Viewer

### Consumer

#### Consumer permission for a project

If you have consumer permission for a project in the Feature Store, you are authorized to list or obtain a feature set from the project. In other words, as a consumer of a project, you can retrieve data from all feature sets. As the consumer of the project, you are automatically granted consumer permission for all feature sets associated with the project.

#### Consumer permission for feature sets

If you have consumer permission for a feature set, you are authorized to call `retrieve` on the feature set. Among retrieved features, only masked features will be displayed as hashed values.

**Note:** As an consumer, you also have the following permission,

- Viewer

### Viewer

#### Viewer permission for a project

If you have viewer permission for a project in the Feature Store, you are authorized to see what feature sets are within the project. This behaviour is also influenced by the Project Access Modifiers.

#### Viewer permission for feature sets

This permission allows you to get feature set and various information about it.

## Project Access Modifiers

Access modifiers on project modifies what users additionally can do. Access modifiers are internally reflected by permissions. Please see Projects for more information.

## Project permission API

#### Add permissions to the project

To **add additional owners** to the project, call:

**Python**

```
project.add_owners(["bob@h2o.ai", "alice@h2o.ai"])
```

To **add additional editors** to the project, call:

**Python**

```
project.add_editors(["bob@h2o.ai", "alice@h2o.ai"])
```

To **add additional consumers** to the project, call:

**Python**

```
project.add_consumers(["bob@h2o.ai", "alice@h2o.ai"])
```

To **add additional sensitive consumers** to the project, call:

**Python**

```
project.add_sensitive_consumers(["bob@h2o.ai", "alice@h2o.ai"])
```

To **add additional viewers** to the project, call:

**Python**

```
project.add_viewers(["bob@h2o.ai", "alice@h2o.ai"])
```

**Remove permissions from the project**

To **remove owners** from the project, call:

**Python**

```
project.remove_owners(["bob@h2o.ai", "alice@h2o.ai"])
```

To **remove editors** from the project, call:

**Python**

```
project.remove_editors(["bob@h2o.ai", "alice@h2o.ai"])
```

To **remove consumers** from the project, call:

**Python**

```
project.remove_consumers(["bob@h2o.ai", "alice@h2o.ai"])
```

To **remove sensitive consumers** from the project, call:

**Python**

```
project.remove_sensitive_consumers(["bob@h2o.ai", "alice@h2o.ai"])
```

To **remove viewers** from the project, call:

**Python**

```
project.remove_viewers(["bob@h2o.ai", "alice@h2o.ai"])
```

**Request permissions to a project**

When cooperating with several users on a project, you may not have a specific permission (i.e., owner/editor/consumer/sensitive consumer) for that project. You can request a specific permission from the project owner.

To begin, **check your current access rights**:

**Python**

```
from featurestore.core.access_type import AccessType
my_access_type = project.current_permission
# returns None in case the user has no access to the project
```

If your level of permission is not sufficient for your needs, **you can request the project owner for access rights**:

**Python**

```
my_request_id = project.request_access(AccessType.CONSUMER, "Preparing the best model")
```

You can **track your pending permission requests** from the clients API:

**Python**

```
my_requests = client.acl.requests.projects.list()
```

When you can no longer see your request, this means it has been processed. To **view the result of your request**, call:

**Python**

```
my_permissions = client.acl.permissions.projects.list(filters)
```

The `filters` argument is optional and specifies which permissions state(s) you are interested in. Its default value is **PermissionState.GRANTED** (which is the most common case). If you do not find your original request granted, it was most likely either rejected or was granted and then revoked.

To **verify the status of your request**, specify using the corresponding filters. For example:

**Python**

```
filters = [PermissionState.REJECTED]
```

In case you changed your mind or situation changed, and you don't need access to the project anymore then you can cancel your request before it was processed by withdrawing it.

**Python**

```
my_requests = client.acl.requests.projects.list()
request = ... # take a request based on your needs
request.withdraw()
```

**Manage permission requests from other users**

As a project owner, a user can request access to that project from you.

To **list the requests pending** for you to handle, call:

**Python**

```
manageable_requests = client.acl.requests.projects.list_manageable()
```

You can then take an item from the returned list and either **approve it**:

**Python**

```
manageable_requests = client.acl.requests.projects.list_manageable()
oldest_request = # select an item from manageable_requests
oldest_request.approve("it will be fun")
```

or **reject it**:

**Python**

```
manageable_requests = client.acl.requests.projects.list_manageable()
oldest_request = # select an item from manageable_requests
oldest_request.reject("it's not ready yet")
```

You can also revoke previously granted access to a project.

First, **list the existing permissions** that you handle:

**Python**

```
manageable_permissions = client.acl.permissions.projects.list_manageable()
```

Then, **select the permission you want to revoke** from the returned list:

**Python**

```
manageable_permission = ... # select an item from manageable_permissions
manageable_permission.revoke("user left the project")
```

The returned request and permission objects from the `list()` and `list_manageable()` method calls contain convenient methods for **accessing the internal state** (the following code is not exhaustive):

**Python**

```
manageable_requests = client.acl.requests.projects.list_manageable()
manageable_request = # select an item from manageable_requests

manageable_request.requestor()  # only on manageable objects
manageable_request.access_type()
manageable_request.status()
manageable_request.reason()
manageable_request.created_on()
manageable_request.resource_type() # requested resource type, PROJECT or FEATURE_SET
manageable_request.get_resource() # returns corresponding feature set/project
```

## Feature set permissions API

### Add permissions to the feature set

In order to add feature set permissions (`owner` / `editor` / `consumer` / `sensitive consumer`), those users should already have the project consumer permission.

For the following examples, "bob@h2o.ai" and "alice@h2o.ai" should already have consumer permissions to the project which consists of the respective feature set.

To **add additional owners** to the feature set, call:

**Python**

```
fs.add_owners(["bob@h2o.ai", "alice@h2o.ai"])
```

To **add additional editors** to the feature set, call:

**Python**

```
fs.add_editors(["bob@h2o.ai", "alice@h2o.ai"])
```

To **add additional consumers** to the feature set, call:

**Python**

```
fs.add_consumers(["bob@h2o.ai", "alice@h2o.ai"])
```

To **add additional sensitive consumers** to the feature set, call:

**Python**

```
fs.add_sensitive_consumers(["bob@h2o.ai", "alice@h2o.ai"])
```

To **add additional viewers** to the feature set, call:

**Python**

```
fs.add_viewers(["bob@h2o.ai", "alice@h2o.ai"])
```

### Remove permissions from the feature set

To **remove owners** from the feature set, call:

**Python**

```
fs.remove_owners(["bob@h2o.ai", "alice@h2o.ai"])
```

To **remove editors** from the feature set, call:

**Python**

```
fs.remove_editors(["bob@h2o.ai", "alice@h2o.ai"])
```

To **remove consumers** from the feature set, call:

**Python**

```
fs.remove_consumers(["bob@h2o.ai", "alice@h2o.ai"])
```

To **remove sensitive consumers** from the feature set, call:

**Python**

```
fs.remove_sensitive_consumers(["bob@h2o.ai", "alice@h2o.ai"])
```

To **remove viewers** from the feature set, call:

**Python**

```
fs.remove_viewers(["bob@h2o.ai", "alice@h2o.ai"])
```

### Request permissions to a feature set

Feature set permissions follow the same structure and reasoning as project permissions. The following is a short list of available methods.

To **list current feature set permissions**, call:

**Python**

```
from featurestore.core.access_type import AccessType
my_access_type = fs.current_permission
# returns None in case the user does not have access to the project
```

To **request feature set permissions**, call:

**Python**

```
my_request_id = fs.request_access(AccessType.CONSUMER, "Preparing the best model")
```

To **list pending requests**, call:

**Python**

```
my_requests = client.acl.requests.feature_sets.list()
```

To **withdraw a pending request**, call:

**Python**

```
my_requests = client.acl.requests.feature_sets.list()
request = ... # take a request based on your needs
request.withdraw()
```

To **list granted** (without passing an argument) **or rejected/revoked** (with provided corresponding `filters` argument) **permissions**, call:

**Python**

```
filters = [PermissionState.REJECTED]
my_permissions = client.acl.permissions.feature_sets.list(filters)
```

### Manage feature set permissions

Feature set permissions follow the same structure and reasoning as project permissions. The following is a short list of available methods.

To **list and approve/reject a pending feature set permission request**, call:

**Python**

```
manageable_requests = client.acl.requests.feature_sets.list_manageable()
manageable_request = # select an item from manageable_requests
manageable_request.approve("it will be fun")
# or
manageable_request.reject("not yet ready")
```

To **list and revoke an existing feature set permission**, call:

**Python**

```
manageable_permissions = client.acl.requests.feature_sets.list_manageable()
manageable_permission = ... # select an item from manageable_permissions
manageable_permission.revoke("user left project")
```

# Projects API

## Listing projects

To **list all projects**, call:

**Python**

```
client.projects.list()
```

This method returns a Python generator which can be used to lazily iterate over all projects.

## Listing feature sets across multiple projects

Each project entity allows you to list projects in it as:

**Python**

```
client.projects.get("..").list()
```

however this only lists feature sets in that specific project. To list feature sets across multiple projects, run:

**Python**

```
client.projects.list_feature_sets(["project_name_A", "project_name_B"])
```

The single argument of this method is always an array containing the names of projects in which to perform the feature set search.

**Note:** The list method does not return feature sets directly, but instead returns an iterator which obtains the feature sets lazily.

## Create a project

To **create a project**, call:

**Python**

```
project = client.projects.create(project_name="project", description="description", access_modifier=AccessMod
```

To see naming conventions for project names, visit Default naming rules.

## Project Access Modifier

`AccessModifier` can be passed when creating a project or during updating a project. Available values are:

- **Public** - means that every user in the system can see this project and feature sets within the project.
- **ProjectOnly** - means that every user can see the project, but only users with viewer permission can see feature sets within this project.
- **Private** - means that only owner and users the owner gave permission to can access this project and feature sets within.

Default value is `AccessModifier.PRIVATE`.

## Get an existing project

To get an **existing project**, call:

**Python**

```
project = client.projects.get("project")
```

## Remove a project

To **remove the project**, call:

**Python**

```
project.delete()
```

This will remove all feature sets and features stored inside this project.

### Update project fields

The following fields are modifiable on the project api:

**Python**

```
- description
- custom_data
```

To update the field, simply call the setter of that field. For example, to **update the description**, call:

**Python**

```
project.description = "Better description"
```

To retrospectively find out who and when updated project, call:

**Python**

```
project.last_updated_by
project.last_updated_date_time
```

To see how to set permissions on projects, visit Authentication.

### Listing project users

From project owner's perspective, it may be needed to understand who is actually allowed to access and modify the given project. Therefore, there are convenience methods to list project users according to their rights. Each of these methods returns list of users that have specified or higher rights, their actual access rights and a resource type specifying, where the access right permission comes from.

**Note:** The list method does not return users directly. Instead, it returns an iterator which obtains the users lazily.

**Python**

```
# listing users by access rights
project = client.projects.get("training_project")
owners = project.list_owners()
editors = project.list_editors()
sensitive_consumers = project.list_sensitive_consumers()
consumers = project.list_consumers()
viewers = project.list_viewers()

# accessing returned element
owner = next(owners)
owner.user
owner.access_type
owner.resource_type
```

## Open project in Web UI

This method opens the given project in Feature Store Web UI.

**Python**

```
project.open_website()
```

# Schema API

A schema is extracted from a data source. The schema represents the features of the feature set.

## Creating the schema

### Python

- `create_from` is available on the `Schema` class and is used to create a schema instance from a string formatted schema
- `create_derived_from` is available on the `Schema` class and is used to create a derived schema instance from a string formatted schema and parent feature set along with transformation
- `to_string` is available on a schema instance and is used to serialise the schema object to string format

## Usage

### Create a schema from a string

A schema can be created from a string format:

### Python

```
from featurestore import Schema
schema = "col1 string, col2 string, col3 integer"
schema = Schema.create_from(schema)
```

### Create a derived schema from a string

### Python

```
from featurestore import Schema
import featurestore.transformations as t
spark_pipeline_transformation = t.SparkPipeline("...")
schema_str = "id INT, text STRING, label DOUBLE, state STRING, date STRING, words ARRAY"
schema = Schema.create_derived_from(schema_str, [parent_feature_set], spark_pipeline_transformation)
```

### Create a schema from a data source

A schema can also be created from a data source. To see all supported data sources, see Supported data sources.

### Python

```
schema = client.extract_schema_from_source(source)
schema = Client.extract_schema_from_source(source, credentials)
```

**Note:** An optional parameter, `credentials` , can be specified. If specified, these credentials are used instead of environmental variables.

### Create a schema from a feature set

### Python

```
feature_set = project.feature_sets.get("example")
schema = Schema.create_from(feature_set)
```

### Create a derived schema from a parent feature set with applied transformation

A derived schema can be created from an existing feature set using selected transformation. To see all supported transformations, see Supported derived transformation.

### Python

```
import featurestore.transformations as t
spark_pipeline_transformation = t.SparkPipeline("...")

schema = client.extract_derived_schema([parent_feature_set], spark_pipeline_transformation)
```

#### Load schema from a feature set

You can also load a schema from an existing feature set:

**Python**

```
schema = feature_set.schema.get()
```

#### Create a new schema by changing the data type of the current schema

**Python**

```
from featurestore.core.data_types import STRING
schema["col"].data_type = STRING
# nested columns
schema["col1"].schema["col2"].data_type = STRING
```

#### Create a new schema by column selection

**Python**

```
schema.select(features)
schema.exclude(features)
```

#### Create a new schema by adding a new feature schema

**Python**

```
from featurestore.core.data_types import STRING
from featurestore import FeatureSchema
new_feature_schema = FeatureSchema("new_name", STRING)
# Append
schema.append(new_feature_schema)  # Append to the end
schema.append(new_feature_schema, schema["old"])  # Append after old
# Prepend
new_schema = schema.prepend(new_feature_schema)  # Prepend to the beginning
new_schema = schema.prepend(new_feature_schema, schema["old"])  # Prepend before old
```

#### Modify special data on a schema

**Python**

```
schema["col1"].special_data.sensitive = True
schema["col2"].special_data.spi = True
# Nested feature modification
schema["col3"].schema["col4"].special_data.pci = True
```

**Note:** Available special data fields on the Schema object are `spi`, `pci`, `rpi`, `demographic` and `sensitive`. These are boolean fields and can be either set with true/false.

#### Modify feature type

**Python**

```
from featurestore.core.entities.feature import *
schema["col1"].feature_type = NUMERICAL
schema["col2"].feature_type = AUTOMATIC_DISCOVERY
# Nested feature modification
schema["col3"].schema["col4"].feature_type = TEXT
```

The `AUTOMATIC_DISCOVERY` means that the feature type will be determined on the backend side based on the feature data type automatically. `AUTOMATIC_DISCOVERY` is the default value for all the schema's feature types.

### Set feature description

It is also possible to provide a description for a feature schema. This description is propagated to the feature.

#### Python

```
schema["col1"].description = "The best feature"
```

### Set feature classifier

Features in a feature set can be tagged by a classifier from a predefined list. The classifier on the feature denotes the type of data stored in the feature.

#### Python

```
client.classifiers.list()  # this returns all configured classifiers on the backend
schema["col1"].classifier = "emailId"
```

### Save schema as string

A schema can be serialized to string format:

#### Python

```
str_schema = schema.to_string()
```

# Feature set API

## Registering a feature set

To register a feature set, you first need to obtain the schema. See Schema API for information on how to create the schema.

### Python

```
project.feature_sets.register(schema, "feature_set_name", description="", primary_key=None, time_travel_colum
MM-dd HH:mm:ss", partition_by=None, time_travel_column_as_partition=False, flow=None)
```

If the `partition_by` argument is not set, the time travel column will be used by Feature Store to partition the layout by each ingestion. If it is defined, `time_travel_column_as_partition` can be set to `True` to use time travel based partitioning additionally.

**Note:** The `feature_sets.register`, and `feature_set.flow` methods use the enum `FeatureSetFlow`. Enum (enumeration) is a fundamental concept in programming languages that allow developers to define a set of named values. They provide a convenient way to group related values and make code more readable and maintainable.

If the `flow` argument is set, it will influence where data is stored. Following values (from enum FeatureSetFlow) are supported:

- **FeatureSetFlow.OFFLINE_ONLY** - data is stored only in offline feature store. Online ingestion and materialization is disabled.
- **FeatureSetFlow.ONLINE_ONLY** - data is stored only in online feature store. Offline ingestion and materialization is disabled.
- **FeatureSetFlow.OFFLINE_ONLINE_MANUAL** - data is stored in both offline and online Feature Store, but automatic materialization to online is disabled. That means propagating data between online to offline is automated, but offline to online is manual and must be triggered by online materialization call.
- **FeatureSetFlow.OFFLINE_ONLINE_AUTOMATIC** - data is stored in both offline and online Feature Store, and automatic materialization to online is enabled. That means this is used to automatically propagate data between offline - online and online - offline. You don't have to call `materialize_online` as it is done automatically.

**Note:** In case primary key or partition by arguments contain same feature multiple times, only distinct values are used.

**Note:** If value in primary key or partition by or time travel column corresponds to two or more features, most nested is selected by default. In other cases, specific feature can be selected by enclosing the feature name in “

For example, feature set contains feature named “test.data” and second feature “test” with nested feature “data”. But default for value “test.data”, nested feature “data” will be selected. If feature with name “test.data” should be selected, value should be changed to “‘.test.data’“

**Note:** Feature Store is using time format used by Spark. Specification is available here.

**Note:** If users wants to create feature sets which are accessible only by the owner and users the owner gave permission to, that feature set should be created in a private project.

To see naming conventions for feature set names, please visit Default naming rules.

To register a derived feature set, you first need to obtain the derived schema. See Schema API for information on how to create the schema.

### Python

```
import featurestore.transformations as t
spark_pipeline_transformation = t.SparkPipeline("...")

derived_schema = client.extract_derived_schema([parent_feature_set], spark_pipeline_transformation)

project.feature_sets.register(derived_schema, "derived_feature_set", description="", primary_key=None, time_t
MM-dd HH:mm:ss", partition_by=None, time_travel_column_as_partition=False)
```

Features can be masked by setting Special Data fields in the schema. For further information, please visit Modify special data on a schema.

Setting any of the following attributes to `true` marks the feature for masking:

- `spi` - Sensitive Personal Information

- `pci` - Payment Card Industry
- `rpi` - Real Property Inventory
- `demographic`
- `sensitive`

Any of the special data tags would allow for the masking functionality to work and separate sensitive consumer output (e.g. unmasked data) from the masked view that the consumer role sees. Which tag is selected is more bookkeeping than leading to different functionality.

**Note:** Feature Store does not support registering feature sets with the following characters in column names:

- `,`
- `;`
- `{` or `}`
- `(` or `)`
- `new line character`
- `tab character`
- `=`

#### Time travel column selection

You can specify a time travel column during the registration call. If the column is specified, Feature Store will use that column to obtain time travel data and will use it for incremental ingest purposes. The explicitly passed time travel column must be present in the schema passed to the registration call.

If the time travel column is not specified, a virtual one is created, so you can still do time travel on static feature sets. Each ingestion to this feature set is treated as a new batch of data with a new timestamp.

Use the following register method argument to **specify the name of the time travel column** explicitly:

**Python**

```
time_travel_column
```

#### Inferring the data type of date-time columns during feature set registration

**File types without schema information**: For file types that have no metadata about column types (e.g., CSV), Feature Store detects date-time columns as regular string.

**File types containing schema information**: For file types that keep information about the data types (e.g., Parquet), Feature Store respects those types. If a date-time column is stored with a type of Timestamp or Date, Feature Store will respect that during the registration.

## Listing feature sets within a project

**Note:** The list method does not return feature sets directly. Instead, it returns an iterator which obtains the feature sets lazily.

**Python**

```
project.feature_sets.list(query=None, advanced_search_options=None)
```

The `query` and `advancedSearchOption` arguments are optional and specify which feature sets should be returned. By default, no filtering options are specified.

To filter feature sets by name, description or tags please use `query` parameter.

**Python**

```
project.feature_sets.list(query="My feature")
```

The `advancedSearchOption` allows to filter feature sets by feature name, description or tags.

To provide the 'advancedSearchOption' in your requests, follow these steps:

**Python**

```
from featurestore.core.search_operator import SearchOperator
from featurestore.core.search_field import SearchField
from featurestore import AdvancedSearchOption
search_options = [AdvancedSearchOption(search_operator=SearchOperator.SEARCH_OPERATOR_LIKE, search_field=Sear

project.feature_sets.list(advanced_search_options=search_options)
```

Both parameters could be used together.

You can also **list all major versions of the feature set**:

**Python**

```
fs.major_versions()
```

This call shows all major versions of the feature set (the current and previous ones).

You can also **list all versions of the feature set**:

**Python**

```
fs.list_versions()
```

This call shows all versions of the feature set (the current and previous ones).

## Obtaining a feature set

**Python**

```
fs = project.feature_sets.get("feature_set_name", version=None)
```

If the version is not specified, the latest version of the feature set is returned.

You can also get the latest minor version of feature set for given major version

**Python**

```
fs = project.feature_sets.get_major_version("feature_set_name", 2)
```

It is also possible to obtain different version of a feature set from some feature set instance as:

**Python**

```
fs = feature_set.get_version("2.1")
```

### Previewing data

You can preview up to a maximum of 100 rows and 50 features.

**Python**

```
fs.get_preview()
```

## Setting feature set permissions

Refer to Permissions for more information.

## Deleting feature sets

**Python**

```
fs = project.feature_sets.get("name")
fs.delete()
```

## Deleting feature set major versions

**Python**

```
fs = project.feature_sets.get("name")
major_versions = fs.major_versions()
major_versions[0].delete()
```

## Updating feature set fields

To update the field, simply call the setter of that field, for example:

**Python**

```
fs = project.feature_sets.get("name")
fs.deprecated = True
fs.time_to_live.offline = 2
fs.special_data.legal.approved = True
fs.special_data.legal.notes = "Legal notes"
fs.features["col"].special_data.legal.approved = True
fs.features["col"].special_data.legal.notes = "Legal notes"
# Add a new tag to the feature set
fs.tags.append("new tag") # This will add the new tag to the list of existing tags
# Add new tags that will overwrite any existing tags
fs.tags = ["new tag 1", "new tag 2"] # This will overwrite the existing tags with the given list of values
# Assigning a string to tags is not supported
fs.tags = "new tag" # This operation is not supported as tags accepts only a list of strings as input
# Add a new value to the data source domains on the feature set
fs.data_source_domains.append("new domain") # This will add the new domain to the list of existing domains
# Add new domains that will overwrite any existing domains
fs.data_source_domains = ["new domain 1", "new domain 2"] # This will overwrite the existing domains with the
# Assigning a string to domain is not supported
fs.data_source_domains = "new domain" # This operation is not supported as domain accepts only a list of stri
```

**Feature type can be changed** by:

**Python**

```
from featurestore.core.entities.feature import CATEGORICAL
fs = project.feature_sets.get("name")
feature = fs.features["feature"]
my_feature.profile.feature_type = CATEGORICAL
```

The following list of fields can be updated on the **feature set object**:

**Python**

```
- tags
- data_source_domains
- feature_set_type
- description
- application_name
- application_id
- deprecated
- process_interval
- process_interval_unit
- flow
- feature_set_state
- time_to_live.ttl_offline
- time_to_live.ttl_offline_interval
- time_to_live.ttl_online
- time_to_live.ttl_online_interval
- special_data.legal.approved
```

```
- special_data.legal.notes
- feature[].status
- feature[].profile.feature_type
- feature[].importance
- feature[].description
- feature[].special
- feature[].monitoring.anomaly_detection
- feature[].classifiers
```

**Note:**

- `feature_set_type` has two values, `RAW` or `ENGINEERED`. It denotes whether the feature set was derived from raw or processed data. This classification exists for information purposes and does not affect Feature Store behavior.
- `time_to_live` is currently respected for data in online feature store only. It indicates the duration for which records remain stored before they are evicted.

To retrospectively find out who and when updated a feature set, call:

**Python**

```
fs.last_updated_by
fs.last_updated_date_time
```

## Recommendation and classifiers

Refer to the Recommendation API for more information.

## New version API

Refer to the Create new feature set version API for more information.

## Feature set schema API

### Getting schema

To get feature set's schema, run:

**Python**

```
fs = project.feature_sets.get("gdp")
fs.schema.get()
```

### Checking schema compatibility

To compare feature set's schema with the new data source's schema, run:

**Python**

```
fs = project.feature_sets.get("gdp")
new_schema = client.extract_schema_from_source()
fs.schema.is_compatible_with(new_schema, compare_data_types=True)
```

**Parameters explanation:**

**Python**

- `new_schema` new schema to check compatibility with.
- `compare_data_types` accepts True/False, indicates whether data type needs to be compared or not.
- If `compare_data_types` is `True`, then data types for features with same name will be verified.
- If `compare_data_types` is `False`, then data types for features with same name will not be verified.

#### Patching new schema

Patch schema checks for matching features between the 'new schema' and the existing 'fs.schema'. If there is a match, then the meta data such as special_data, description etc are copied into the new_schema

To patch the new schema with feature set's schema, run:

**Python**

```
fs = project.feature_sets.get("gdp")
new_schema = client.extract_schema_from_source()
fs.schema.patch_from(new_schema, compare_data_types=True)
```

**Parameters explanation:**

**Python**

- `new_schema` new schema that needs to be patched.
- `compare_data_types` accepts True/False, indicates whether data type are to be compared while patching.
- If `compare_data_types` is `True`, then data type from feature set schema is retained for features with same name and different types.
- If `compare_data_types` is `False`, then data type from new schema is retained for features with same name and different types.

### Offline to online API

To push existing data from offline Feature store into online, run:

**Blocking approach**:

**Python**

```
feature_set.materialize_online()
```

**Non-Blocking approach**:

**Python**

```
future = feature_set.materialize_online_async()
```

**Note:** Feature set must have a primary key and time travel column defined in order to materialize the offline store into online.

More information about asynchronous methods is available at Asynchronous methods.

Subsequent calls to materialization only push the new records since the last call to online.

### Online to offline API

There is a background process that periodically starts online to offline ingestion, but in case there is a need to push existing data from online Feature store into offline earlier than scheduled, then run:

**Blocking approach**:

**Python**

```
feature_set.start_online_offline_ingestion()
```

**Non-Blocking approach**:

**Python**

```
job = feature_set.start_online_offline_ingestion_async()
```

## Feature set jobs API

You can get the list of jobs that are currently processing for the specific feature set by running:

**Python**

You can also retrieve a specific type of job by specifying the `job_type` parameter.

```
from featurestore.core.job_types import INGEST, RETRIEVE, EXTRACT_SCHEMA
fs.get_active_jobs()
fs.get_active_jobs(job_type=INGEST)
```

## Refreshing feature set

To refresh the feature set to contain the latest information, call:

**Python**

```
fs.refresh()
```

## Getting recommendations

To get recommendations, call:

**Python**

```
fs.get_recommendations()
```

The following conditions must hold for recommendations:

- The feature set must have at least one or more classifiers defined.
- The results will be based on the retrieve permissions of the user.

## Marking feature as target variable

When feature sets are used to train ML models, it can be beneficial to know which feature was used as model's target variable. In order to communicate this knowledge between different feature set users, there is a possibility to mark/discard a feature as a target variable and list those marked features.

**Python**

```
feature_state = fs.features["state"]
feature_state.mark_as_target_variable()

fs.list_features_used_as_target_variable()

feature_state.discard_as_target_variable()
```

## Listing feature set users

From feature set owner's perspective, it may be needed to understand who is actually allowed to access and modify the given feature set. Therefore, there are convenience methods to list feature set users according to their rights. Each of these methods returns list of users that have specified or higher rights, their actual access rights and a resource type (project or feature set) specifying, where the access right permission comes from.

**Note:** The list method does not return users directly. Instead, it returns an iterator which obtains the users lazily.

**Python**

```
# listing users by access rights
fs = project.feature_sets.get("training_fs")
owners = fs.list_owners()
editors = fs.list_editors()
sensitive_consumers = fs.list_sensitive_consumers()
consumers = fs.list_consumers()
viewers = fs.list_viewers()
```

```
# accessing returned element
owner = next(owners)
owner.user
owner.access_type
owner.resource_type
```

### Derived feature sets

As mentioned in the beginning, a (derived) feature set can be defined in terms of other features sets and a transformation. There are several convenience methods that help you find out a lineage of a given feature set.

**Note:**

- In the Feature Store, lineage is preserved by tracking the ingest history. This allows users to identify the data source from which the ingest occurred.
- Users can create derived feature sets which are transformations of existing feature sets. This relationship is also preserved within the Feature Store.

Is the feature set a derived one or not?

**Python**

```
fs.is_derived()
```

Which feature sets were used to define this derived feature set?

**Python**

```
parent_feature_sets = fs.get_parent_feature_sets()
```

To get a list of derived feature set(s) that were build upon this feature set.

**Python**

```
derived_feature_sets = fs.get_derived_feature_sets()
```

## Open feature set in Web UI

This method opens the given feature set in Feature Store Web UI.

**Python**

```
fs.open_website()
```

## Optimizing feature set storage (Delta lake backend only)

In special cases, there can be a performance benefit when a feature set's data gets optimized. In order to manually enforce a storage optimization use following call. By default, feature set storage gets optimized by Z-order optimization for primary key(s). In case an optimization for different feature's list is needed, you can specify the optimization explicitly when making the call.

The optimization call returns optimization metrics provided by storage. Furthermore, a new minor feature set version gets created. The updated feature set version contains optimization input as one of its attributes.

**Python**

```
# z-order optimization for primary key(s) by default
response = fs.optimize_storage()

# show response details
response.optimization_metrics

# z-order optimization for specific columns
fs.optimize_storage(ZOrderByOptimization(["name", "age"]))
```

```
# refresh version and show optimization input
fs.refresh()
fs.storage_optimization
```

# Feature API

## Feature statistics

Feature can have several feature types:

- TEXT
- CATEGORICAL
- NUMERICAL
- TEMPORAL
- COMPOSITE

For feature of type `CATEGORICAL`, categorical statistics are computed. For feature of type `NUMERICAL`, numerical statistics are computed.

# Ingest API

Feature store ensures that data for each specific feature set does not contain duplicates. That means that only data which are unique to the feature set cache are ingested as part of the ingest operation. The rows that would lead to duplicates are skipped.

Ingest can be run on instance of feature set representing any minor version. The data are always ingested on top of latest storage stage.

## Offline ingestion

To ingest data into the Feature Store, run:

**Blocking approach**:

**Python**

```
fs = project.feature_sets.get("gdp")
fs.ingest(source)
fs.ingest(source, credentials=credentials)
```

**Note:** This method is not allowed for derived feature sets.

**Non-Blocking approach**:

**Python**

```
fs = project.feature_sets.get("gdp")
future = fs.ingest_async(source)
fs.ingest_async(source, credentials=credentials)
```

**Note:** This method is not allowed for derived feature sets.

More information about asynchronous methods is available at Asynchronous methods.

**Parameters explanation:**

**Python**

- `source` is the data source where Feature Store will ingest from.
- `credentials` are credentials for the data source. If not provided, the client tries to read them from environmental variables. For more information about passing credentials as a parameter or via environmental variables, see Credentials configuration.

To ingest data into feature store from **sources that gets changed periodically**, run:

**Python**

```
fs = project.feature_sets.get("gdp")
new_schema = client.extract_from_source(ingest_source)
if not fs.schema.is_compatible_with(new_schema, compare_data_types=True):
patched_schema = fs.schema.patch_from(new_schema, compare_data_types=True)
new_feature_set = fs.create_new_version(schema=patched_schema, reason="schema changed before ingest")
new_feature_set.ingest(ingest_source)
else:
fs.ingest(ingest_source)
```

This call materializes the data and stores it in the Feature Store storage.

**Note:** Feature Store does not allow specification of a feature with the same name but different case. However, during ingestion we treat feature names case-insensitive. For example, when ingesting into feature set with single feature named `city`, the data are ingested correctly regardless of the case of the column name in the provided data source. We correctly match and ingest into `city` feature if column in the data source is named for example as `CITY`, `CiTy` or `city`.

## Online ingestion

To ingest data into the online Feature Store, run:

**Python**

```
feature_set.ingest_online(row/s)
```

The `row/s` is either a single JSON row or an array of JSON strings used to ingest into the online.

**Note:** Feature set must have a primary key defined in order to ingest and retrieve from the online Feature Store.

This method is not allowed for derived feature sets.

## Lazy ingestion

Lazy ingestion is a method which when be used to migrate feature sets from different systems without the need of ingesting feature sets immediately. In lazy ingestion, the ingestion process starts the first time data from feature sets are retrieved.

Corresponding scheduled task is created when lazy ingest is defined. For more information, please see Obtaining a lazy ingest task. Only one lazy ingest task can be defined per feature set major version.

When you ingest feature sets directly using `feature_set.ingest(source)`, the ingest task associated with the feature set will be deleted from the feature store.

To ingest data lazy, run:

### Python

```
fs.ingest_lazy(source)
fs.ingest(source, credentials=credentials)
```

**Note:** This method will run ingest on feature set *retrieve*. See Feature set schedule API for information on how to delete or edit scheduled task.

# Ingest history API

## Getting the ingestion history

The following call returns the ingestion history for the feature set:

**Python**

To **create an ingest history** containing all ingestions:

```
history = my_feature_set.ingest_history()
history.list()
```

To **obtain the size** of the history:

```
history.size
```

To **refresh the history** to contain the latest ingestions:

```
history.refresh()
```

To **obtain the first or last ingestion**:

```
first_ingest = history.first
last_ingest = history.last
```

To **obtain a specific ingestion** using an ingest id:

```
specific_ingest = history.get(ingest_id)
```

To **retrieve data for a specific ingestion**:

```
ingest.retrieve()
```

To **get information about the ingestion** like who and when did it, call:

```
first_ingest.ingested_at
first_ingest.ingested_records_count
first_ingest.scope
first_ingest.source
first_ingest.started_by
```

**Note:** Data ingested before system version 0.0.36 is not supported for retrieval via this API.

## Reverting ingestion

Any specific ingest can be reverted. Reverting creates a new minor version with the data corresponding to the specific ingest removed.

**Python**

```
ingest.revert()
```

**Note:** This method is not allowed for derived feature sets.

There are also **asynchronous variants** of these methods:

**Python**

```
ingest.revert_async()
```

**Note:** The above method is not allowed for derived feature sets.

# Retrieve API

To retrieve the data, first run:

**Python**

```
ref = fs.retrieve(start_date_time=None, end_date_time=None)
```

**Parameters explanation:**

**Python**

If `start_date_time` and `end_date_time` are empty, all ingested data are fetched. Otherwise, these parameters are used to retrieve only a specific range of ingested data. For example, when ingested data are in a time range between 'T1

This call returns immediately with a retrieve holder allowing you to use multiple approaches on how to retrieve the data. Based on the input parameters, the specific call for data retrieval searches the cache and tries to find the ingested data.

**Note:** When utilizing Snowflake as the backend storage for your data, it's important to understand how nested features are stored and retrieved. This note provides insights into the storage and retrieval process, differentiating between Snowflake and Delta Lake storage.

Nested features are stored as VARIANT data type. For example, in a column named 'Person,' a nested feature might be stored as follows:

```
{ "Age": 5, "Name": "John"}
```

**Retrieval in Spark or Parquet File**

- When a user retrieves data stored in Snowflake as a backend using Spark or as a Parquet file, the structure is retained.
- In the retrieved data, the nested feature appears as a JSON string within the designated column and row.

Understanding the nuances of how Snowflake and Delta Lake handle nested features is crucial for seamless data storage, retrieval, and compatibility with different data processing tools. Whether it's the JSON format in Snowflake or the hierarchical column structure in Delta Lake, this information ensures efficient utilization of your chosen backend storage solution.

## Downloading the files from Feature Store

You can download the data to your local machine by:

**Blocking approach:**

**Python**

```
dir = ref.download()
```

**Non-Blocking approach:**

**Python**

```
future = ref.download_async()
```

**Note:** More information about asynchronous methods is available at Asynchronous methods.

This will download all produced data files (parquet) into a newly created directory.

## Obtaining data as a Spark Frame

You can also read the data from the retrieve call directly as a Spark frame:

**Python**

```
ref = my_feature_set.retrieve()
data_frame = ref.as_spark_frame(spark_session)
```

Read more about Spark Dependencies in the Spark dependencies section.

## Retrieving from online

To retrieve data from the online Feature Store, run:

### Python

```
json = feature_set.retrieve_online(key)
```

The `key` represents a specific primary key value for which the entry is obtained.

# Jobs API

## Listing jobs

The `List Jobs` API returns your currently initiated jobs. By default, only active jobs, which means the jobs that are currently executing, are returned.

### Python

You can provide an additional argument `active=False` to return all jobs. You can also retrieve a specific type of job by specifying the `job_type` parameter.

```
from featurestore.core.job_types import INGEST, RETRIEVE, EXTRACT_SCHEMA, REVERT_INGEST, MATERIALIZATION_ONLI
client.jobs.list()
client.jobs.list(active=False, job_type=INGEST)
```

**Note:** The `active` parameter indicates that the job is currently executing.

## Getting a job

### Python

```
job = client.jobs.get("job_id")
```

## Cancelling a job

To request cancel without waiting for cancellation to complete you need to call

### Python

```
job.cancel()
```

To request cancel and wait for cancellation to complete you need to call

### Python

```
job.cancel(wait_for_completion=True)
```

## Checking job status

### Python

```
job.is_done()
```

## Checking if job is cancelled

### Python

```
job.is_cancelled()
```

## Getting job results

### Python

```
job.get_result()
```

## Checking job metrics

### Python

```
job.get_metrics()
```

## How to download using RetrieveJob

### Python

Unlike other job types, `RetrieveJob` also has a `download` method which gives you the option to download retrieved data created by the backend job.

You can also make use of the `download_async` method that downloads the files asynchronously. More information about asynchronous methods is at Asynchronous methods.

```
retrieve_job = client.jobs.get("job_id")
data_path = retrieve_job.download()
```

## Job metadata

| Field Name | User Modifiable | Values |
|---|---|---|
| id | No | - |
| jobType | No | Ingest, Retrieve, ExtractSchema, Revert, MaterializationOnline, ComputeStatistics, ComputeRecommendationClassifiers, CreateMLDataset, Backfill |
| done | No | true, false |
| cancelled | No | true, false |
| childJobIds | No | Child job ids |

**Note:** The `done` parameter indicates that the job has completed its execution and the results are available.

# Create new feature set version API

A feature set is a collection of features. Users can create a new version of an existing feature set for various reasons.

## When to create a new version of a feature set

A new major version of a feature set can be created for various reasons, for example:

- If the schema of a feature set has changed, such as, changing the data type of one or more features, adding one or more features, deleting one or more features or modifying a special data field in one or more features
- A new version of a feature set may need to be derived from another feature set.
- If there is a change in how a feature is calculated by an external tool, which refers to an affected feature in the Feature Store The API is capable of specifying a list of affected features, which will lead to an increment in the version number of those affected features.
- Changing partition columns, primary keys or whether time travel columns is used as partition column
- User wants to create a new version of feature set by back-filling with data from other feature set version

## What happens after creating a new version

- The feature set's major version number is incremented.
- For all the affected features, the version number is incremented.
- The version number is incremented for all features whose type has been changed because the schema has been provided.
- Appropriate messaging is updated on the feature set and features describing the new version.
- If a new version of the feature set is derived, an automatic ingestion job will be triggered.

## How to create a new version

The following command is used to create a new version of a feature set.

### Python

```
feature_set.create_new_version(...)
```

The following examples show how new version can be created:

- Create a new version on a schema change
- Create a new version by specifying affected features
- Create a new version by specifying affected features and schema
- Create a new version with backfilling

### Create a new version on a schema change

### Python

```
fs = project.feature_sets.get("abc")

# Get current schema
schema = fs.schema.get()

# Change datatype
from featurestore.core.data_types import STRING
schema["xyz"].data_type = STRING
# Change special flag
schema["xyz"].sensitive = True

# Create new version
new_fs = fs.create_new_version(schema=schema, reason="some message", primary_key=[])
```

- `schema` is the new schema of the feature set. Refer to Schema API for information on how to create the schema.
- `reason (optional)` is your provided message describing the changes to the feature set. This message will be applied to the feature set and the affected features. By default, an auto-generated message will be populated describing the features added/removed/modified.

- `primary_key (optional)` if not empty, new primary key is set on the feature set
- `partition_by (optional)` if not empty new partition columns are set on the feature set
- `time_travel_column_as_partition (optional)` if true, time travel column is used as partition in the new feature set version
- `backfill_options (optional)` If specified, feature store will back-fill data from older feature set version based on the configuration passed in this object
- `time_travel_column (optional)` if not empty new time travel column is set on the feature set
- `time_travel_column_format (optional)` if not empty new time travel column format is set on the feature set

**Note:** In case primary key or partition by arguments contain same feature multiple times, only distinct values are used.

#### Create a new version by specifying affected features

**Python**

```
fs = project.feature_sets.get("abc")

# Create new data source
new_source = CSVFile("new file source")

# Create new version
new_fs = fs.create_new_version(affected_features=["xyz"], reason="Computation of feature XYZ changed")
```

- `affected_features` is a list of feature names for which you are explicitly asking to increment the version number.
- `reason (optional)` is your provided message describing the changes to the feature set. This message will be applied to the feature set and the affected features. By default, an auto-generated message will be populated describing the features added/removed/modified.

#### Create a new version by specifying affected features and schema

A new schema will define a new feature set version. For features marked as affected and included in the old feature set version and in the new version, the version number will be incremented as in Option 2: Create a new version by specifying affected features.

#### Create a new version with backfilling

In H2O Feature Store, backfilling involves creating a new version of a feature set that includes data from a previous version, along with any necessary transformations such as feature mapping or filtering based on a time range.

**User scenario:**

You have a previous version (version 1.5) of a feature set that contains data from the past 5 years, and you want to create a new version (version 2.0) that only includes data from the past 2 years. To accomplish this, you need to use backfilling. You must specify the version (version 1.5) from which you want to use the data. Then you apply a time range filter on a "time travel" column in the feature set to select the data from the past 2 years. Once the filter is applied, the H2O Feature Store will create a new version of the feature set (version 2.0) that includes only the selected data.

**Python**

```
fs = project.feature_sets.get("abc")

# Get current schema
schema = fs.schema.get()

# Change datatype
from featurestore.core.data_types import STRING
schema["xyz"].data_type = STRING
# Change special flag
schema["xyz"].sensitive = True
```

```
# Create new version with backfilling
backfill = BackfillOption(from_version="", from_date = None, to_date = None, spark_pipeline=None, feature_map
new_fs = fs.create_new_version(schema=schema, reason="some message", backfill_options=backfill)
```

- `from_version` is the version from which backfill will be executed. If the argument refers to just major version, e.g. "1", then the corresponding latest minor version will be used.
- `from_date` is date from which data will be filter out
- `to_date` is date to which data will be filter out
- `spark_pipeline` is transformation that will be applied to data. Refer to Supported derived transformation for information on how to use transformation
- `feature_mapping` is default value mapping for feature

**Example:**

**Python**

```
import datetime
import featurestore.core.transformations as t
spark_pipeline_transformation = t.SparkPipeline("/path_to_pipeline/spark_pipeline.zip")
backfill = BackfillOption(from_version="1.1", from_date = datetime.datetime(2021, 2, 24, 00, 00), to_date = d
new_fs = fs.create_new_version(schema=schema, reason="some message", backfill_options=backfill)
```

**Note:** `Spark pipeline transformation` is triggered after applying all options: `from_date`, `to_date`, `feature_mapping`.

# Asynchronous methods

### Python

Several methods in the Feature Store Client API have asynchronous variants (methods ending with `_async`).

For example, calling `retrieve` in an asynchronous way:

```
job = fs.retrieve_async(start_date_time=None, end_date_time=None)
```

This method returns a job. The job has 2 methods:

- `is_done`
- `get_result`

The method `is_done` returns true if the job has finished, false otherwise. The method `get_result` obtains the results of the job. If the method is called before the job has finished, an exception is thrown.

# Spark dependencies

If you want to interact with Feature Store from a Spark session, several dependencies need to be added on the Spark Classpath. Supported Spark versions are 3.5.x.

## Using S3 as the Feature Store storage:

- `io.delta:delta-spark_2.12:3.0.0`
- `org.apache.hadoop:hadoop-aws:${HADOOP_VERSION}`

**Note:** `HADOOP_VERSION` is the hadoop version your Spark is built for.

Version of delta-spark library needs to match your Spark version. Version `3.0.0` can be used by Spark 3.5.

## Using Azure Gen2 as the Feature Store storage:

- `io.delta:delta-spark_2.12:3.0.0`
- `featurestore-spark-dependencies.jar`
- `org.apache.hadoop:hadoop-azure:${HADOOP_VERSION}`

**Note:** `HADOOP_VERSION` is the hadoop version your Spark is built for.

Version of delta-spark library needs to match your Spark version. Version `3.0.0` can be used by Spark 3.5.

The Spark dependencies jar can be downloaded from the Downloads page.

## Using Snowflake as the Feature Store storage:

- `net.snowflake:spark-snowflake_${SCALA_VERSION}:2.12.0-spark_3.4`

**Note:** `SCALA_VERSION` is the scala version used.

Version of spark-snowflake library needs to match your Spark version. Version `2.12.0-spark_3.4` can be used by Spark 3.4.

## General configuration

Spark needs to be started with the following configuration to ensure that the time travel queries are correct:

- `spark.sql.session.timeZone=UTC`
- `spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension`
- `spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog`

In case of running Databricks 11.3 and higher, following options need to be set as well:

- `databricks.loki.fileSystemCache.enabled=false`

If you do not have Apache Spark started, please start it first.

# Recommendation API

A Recommendation API can be used to suggest personalized recommendations based on the data stored in the feature sets. If you have two different feature sets, you can use a Recommendation API to find similarities between the features in those sets and recommend features that are similar in nature or data type.

A classifier can be considered as pattern recognition. Classifiers are used for recommending features based on pattern matching amongst different feature sets. For example, assume you specify a pattern for one feature set. If the same pattern appears in another feature set, the feature store will automatically recognize the pattern in the second feature set and recommend it to the user.

Feature store supports three types of classifiers:

- **Empty classifier** - this classifier can only be assigned to the feature manually
- **Regex classifier** - this classifier will be assigned to the feature after ingestion if the feature values match the configured regex. Regex classifier is typically used for numerical features.
- **Sample classifier** - this classifier will be assigned to the feature after ingestion if the feature values match the configured sample data. Sample classifier is used for text-based features.

**Note:** Classifiers can be defined only by the admins, and it applies to the entire feature store.

When multiple feature sets contain the same classifier, the Recommendation API generates a list of these feature sets. This list then can be used for joining feature sets that have common classifiers.

## Creating a regex classifier

Regex classifiers are used to check if the value of the feature matches the regular expression provided by the classifier specification.

**Python**

```
from featurestore import RegexClassifier

# Create a regex classifier for a feature "zipcode" if 90% of incoming data match a pattern of 5 digits.
client.classifiers.create(RegexClassifier("zipcode", "^\d{5}$", percentage_match=90))
```

- `zipcode` is the name of the classifier
- `^\d{5}$` is the classifier pattern that begins and ends with 5 digits
- `percentage_match=90` indicates at least 90% of the numbers should be 5 digits. `percentage_match` defines the minimum percentage of data that should match the pattern.

To check the output, run the following code, which will list all the classifiers you have created.

```
client.classifiers.list()
```

**Output:**

```
[
    RegexClassifier(name=zipcode, regex=^\d{5}$, percentage_match=90)
]
```

## Creating a sample classifier

Sample classifiers partition an existing dataset to obtain a sample and find the closest pattern match on the new dataset.

**Python**

```
from featurestore import SampleClassifier

# Parameters included: Sampling fraction, Fuzzy distance and the minimum percentage that the data must match
client.classifiers.create(SampleClassifier.from_feature_set(feature_set = fs, name = "countyname_classifier",
```

- `feature_set` is the feature set that you want to apply
- `name` is the name of the classifier
- `column_name` is the name of the column on which you create the classifier. You have to specify which text column you want to match.

- `sample_fraction` specifies the fraction percentage of sample data that should be taken from the above column as opposed to taking the whole set of data. For example, the value specified above (0.50) indicates that only 50% of the sample data should be used from the above column.
- `fuzzy_distance` means if you change one character, it should still match the pattern. For example, let's say you have $AZ$ for Arizona, and if there's $TZ$ somewhere, it will be treated as $AZ$ because only one character is changed
- `percentage_match` indicates that you want to match about 85% of the sample fraction

## Creating an empty classifier

An empty classifier is used to manually apply a pattern to a feature.

**Python**

```
# create an empty classifier
client.classifiers.create("classifierName")
```

To check the output, run the following code, which will list all the classifiers you have created.

```
client.classifiers.list()
```

**Output:**

```
[
    EmptyClassifier(name=classifierName)
]
```

## Changing a classifier manually

By using this method, you can annotate a feature with a specific classifier directly. The main advantage of classifiers is that they are assigned automatically, but users can also do this manually.

**Python**

```
fs = project.feature_sets.get("name")
feature = fs.features["feature"]
client.classifiers.list()  # lists all classifiers
feature.classifiers = {"ssn"}
```

## Updating an existing classifier

An administrator of the Feature Store can update the classifiers:

**Python**

```
from featurestore import RegexClassifier, SampleClassifier

# create an empty classifier
client.classifiers.create("classifierName")

# update empty classifiers to regex classifier which will be applied if 10% of data match "test\d+" regex
client.classifiers.update(RegexClassifier("classifierName", "test\d+", 10))
```

**Note:** No update will be executed on the features. All automatically applied classifiers won't be changed until a new ingestion.

## Deleting an existing classifier

An administrator of the Feature Store can delete the classifiers:

**Python**

```
from featurestore import RegexClassifier, SampleClassifier

# create empty classifier
client.classifiers.create("classifierName")
```

```
# delete classifier
client.classifiers.delete("classifierName")
```

**Note:** No classifiers will be deleted from the features. To delete a classifier from a feature, you need to do so manually.

# Feature set schedule API

You can schedule an ingestion job from Feature Store by using API available on the feature sets.

## Schedule a new task

To create new scheduled task, you first need to obtain the feature set.

### Python

```
fs.schedule_ingest("task_name", source, schedule = "0 2 * * *", description = "", credentials = None, allowed
```

`schedule` argument is in cron format (e.g., `0 2 * * *` will execute task every day at 2 am).

`allowed_failures` argument determines how many times the task can fail till a next failure will deschedule the task in order to save resources. A negative number has the meaning that any number of failures is allowed. Default value is 2.

**Note:** Scheduling ingestion task is allowed from all data sources except Spark dataframe. Data source used for scheduling must be stored in permanent accessible locations, which is not true for Spark Dataframes as they live in memory of some Spark session.

## To list scheduled tasks

List methods do not return tasks directly. Instead, it returns an iterator which obtains the tasks sets lazily.

### Python

```
fs.schedule.tasks()
```

## Obtaining a task

### Python

```
task = fs.schedule.get("task_id")
```

## Examining task executions

Basic information about the task executions can be obtained by asking for executions history. It will provide the start/end times of scheduled task runs and a final (job) status. A special status 'Created' is delivered in case the scheduled task started, but not yet finished. An accompanied job id information can be utilized to get access to a job that fulfilled the execution in the past.

### Python

```
for execution_record in task.execution_history():
print(execution_record)
```

## Obtaining a lazy ingest task

The lazy ingest task allows you to schedule the ingestion of the data for a feature set to a later time, rather than ingesting the data immediately. Each major version of a feature set can contain only one lazy ingest task. To obtain it, run:

### Python

```
task = fs.schedule.get_lazy_ingest_task()
```

## Deleting task

### Python

```
task = fs.schedule.get("task_id")
task.delete()
```

## Updating task fields

To update the field, simply call the setter of that field, for example:

**Python**

```
task = fs.schedule.get("task_id")
task.description = "new description"
task.schedule = "0 6 * * *"
```

## Controlling task liveness

In case a task was scheduled with some defined limit on failures and the failures actually occurred then the task gets automatically paused by Feature Store in order to save resources. To check whether the task was paused or not use following call:

**Python**

```
task = fs.schedule.get("task_id")
task.is_paused()
```

A task can be paused even manually if a user decides so.

**Python**

```
task.pause()
```

A paused task can be rescheduled again by calling a `resume()` method. The resume method can take an optional argument that enables to set a new limit on allowed failures. If the value isn't provided then existing limit stays without change.

**Python**

```
task.resume(allowed_failures = None)
```

To check a current limit on allowed failures see

**Python**

```
task.allowed_failures
```

## Starting lazy ingest task

If lazy ingest task exist on feature set it will be run automatically on first retrieve. The user has the option to run it:

**Python**

```
fs.schedule.start_lazy_ingest_task()
```

**Note:** In case some ingest was executed on feature set version, lazy ingest task will not run.

## Timezone configuration for task

By default, Feature Store clients pick the system timezone. It is possible to change the timezone such as:

**Python**

```
import os, time
os.environ['TZ'] = 'UTC-05:00'
```

**Note:** Supported timezone format is `UTC-XX:XX`, `UTC+XX:XX` or timezones supported by Python.

# Feature set review API

The feature set review process involves the reviewer's acceptance. Depending on the system configuration, all feature sets or only sensitive ones may be subject to review.

## Manage review requests from other users

Reviewer is a user who can approve or reject feature sets.

### List of all pending feature set reviews requests from users

**Python**

```
reviews = client.feature_set_reviews.manageable_requests(filters)
```

The `filters` argument is optional and specifies which review status(es) you are interested in. By default, it is empty.

To provide filter to your requests, please create it as:

**Python**

```
from featurestore.core.review_statuses IN_PROGRESS, APPROVED, REJECTED
filters = [IN_PROGRESS, REJECTED]
```

### List of pending feature set reviews requests related to project

Similarly, you can list the pending feature set reviews on a project basis.

**Python**

```
project = client.projects.get("project_name")
reviews = project.feature_set_reviews.manageable_requests(filters)
```

### Approve a feature set review request from the user

**Python**

```
review_request.approve("it will be fun")
```

### Reject a feature set review request from the user

**Python**

```
review_request.reject("it's not ready yet")
```

### Get a feature set to review

To get feature set in review, please call:

**Python**

```
review_request.get_feature_set()
```

### Preview the data of feature set to review

To preview data ingested to feature set related to review, simply call method:

**Python**

```
review_request.get_preview()
```

## Manage own feature sets in review

User can see own review requests.

#### List all feature sets review requests in review

**Python**

```
reviews = client.feature_set_reviews.my_requests(filters)
```

The `filters` argument is optional and specifies which review status(es) you are interested in. By default, it is empty.

To **verify the status of your request**, specify using the corresponding filters. For example:

**Python**

```
from featurestore.core.review_statuses IN_PROGRESS, APPROVED, REJECTED
filters = [IN_PROGRESS, REJECTED]
```

#### List feature sets review requests in review related to project

Similarly, you can list your own review requests that are related to a project.

**Python**

```
project = client.projects.get("project_name")
reviews = project.feature_set_reviews.my_requests(filters)
```

#### Get a feature set in review

To get feature set with features related to the review, simply call method:

**Python**

```
review.get_feature_set()
```

#### Preview the data of feature set in review

To preview data on feature set in review, please call:

**Python**

```
review.get_preview()
```

#### Delete feature set version in in review

To delete feature set major version which is in review and is in status IN_PROGRESS or REJECTED, please call:

**Python**

```
review.delete()
```

# Dashboard API

Dashboard provides a short summary about the usage of Feature store.

## Recently used projects

To get overview about recently used projects, to list their names, descriptions, access times and optionally to get access to a project itself, use following methods:

**Python**

```
recently_used_projects = client.dashboard.get_recently_used_projects()
recently_used_project = recently_used_projects[0]
recently_used_project.name
recently_used_project.description
recently_used_project.updated_at
recently_used_project.last_access_at
project = recently_used_project.get_project()
```

## Recently used feature sets

Similarly, to get overview about recently used feature sets, to list their names, descriptions, access times and to get access to a feature set itself, use following methods:

**Python**

```
recently_used_feature_sets = client.dashboard.get_recently_used_feature_sets()
recently_used_feature_set = recently_used_feature_sets[0]
recently_used_feature_set.name
recently_used_feature_set.description
recently_used_feature_set.updated_at
recently_used_feature_set.last_access_at
feature_set = recently_used_feature_set.get_feature_set()
```

## Feature sets popularity

To provide hints about feature sets usage, Feature store tracks how often are individual feature sets retrieved (among all users) and provides a sorted list of those feature sets. A user can find out a feature set name, description, how many times was retrieved and its own access rights to that feature set.

**Python**

```
feature_sets_popularity = client.dashboard.get_feature_sets_popularity()
popular_feature_set = feature_sets_popularity[0]
popular_feature_set.name
popular_feature_set.description
popular_feature_set.current_permission
popular_feature_set.number_of_retrievals
feature_set = popular_feature_set.get_feature_set()
```

## Making list of favorite feature sets

To simplify navigation across different feature sets, a user can mark a feature set to include it into a list of personal favorite feature sets. Whenever a feature set gets *pinned* then its reference is put onto the top of the list (it applies for a feature set pinned in the past as well). When a feature set is no more of interest it can be *unpinned* to remove it from the list. To get the favorite feature sets list, use a method from dashboard API.

**Note:** The list method does not return feature sets directly. Instead, it returns an iterator which obtains the feature sets lazily.

**Python**

```
fs = project.feature_sets.get("training_fs")
```

```
# adding a feature set to favorite list
fs.pin()

# getting the list of favorite feature sets
favorites = client.dashboard.list_pinned_feature_sets()

# accessing returned element
favorite = next(favorites)
favorite.name
favorite.description
favorite.updated_at
favorite.pinned_at
favorite_fs = favorite.get_feature_set()

# removing a feature set from favorite list
fs.unpin()
```

## CSV example

### Python

```
from featurestore import Client, CSVFile

# Initialise feature store client
client = Client("url")
client.auth.login()

# Set project specifics
project = client.projects.create("demo")

# Create the csv source
csv = CSVFile("s3a://h2o-datasets/taxi_small.csv")
csv_schema = client.extract_schema_from_source(csv)

# Register the feature set
my_feature_set = project.feature_sets.register(csv_schema, "feature_set_name",
primary_key=["key_name"])

# Ingest to cache
my_feature_set.ingest(csv)

# Retrieve feature set
ref = my_feature_set.retrieve()
ref.download()
```

# CSV folder example

## Example 1: directory structure

- bucket_root
  - nested_folder/
    - 2021-05-03/ -> Date Partition
      - training_data/ -> Sub Folder
        - data.csv
    - 2021-05-04/ -> Date Partition
      - training_sample/ -> Sub Folder
        - data.csv

**Python**

```
from featurestore import Client, CSVFolder

# Initialise feature store client
client = Client("ip:port")
client.auth.login()

# Set project specifics
project = client.projects.create("demo")

# Create the csv folder source
csv_folder = CSVFolder(
root_folder="s3a://feature-store-test-data/nested_folder",
filter_pattern=".*/training.*"
)
csv_folder_schema = client.extract_schema_from_source(csv_folder)

# Register the feature set
my_feature_set = project.feature_sets.register(csv_folder_schema, "feature_set_name",
primary_key=["key_name"])

# Ingest to cache
my_feature_set.ingest(csv_folder)

# Retrieve feature set
ref = my_feature_set.retrieve()
ref.download()
```

## Example 2: directory structure

- bucket_root
  - nested_folder/
    - California
      - 2021-05-03/ -> Date Partition
        - training_data/ -> Sub Folder
          - date.csv
    - Arizona
      - 2021-05-04/ -> Date Partition
        - training_sample/ -> Sub Folder
          - data.csv
    - Texas
      - 2021-05-04/ -> Date Partition
        - training_sample/ -> Sub Folder
          - data.csv

**Python**

```
from featurestore import Client, CSVFolder

# Initialise feature store client
client = Client("ip:port")
client.auth.login()

# Set project specifics
project = client.projects.create("demo")

# Create the csv folder source
csv_folder_source = CSVFolder(
root_folder="s3a://feature-store-test-data/nested_folder",
filter_pattern=".*/.*/training.*"  # To ingest from all states
)
csv_folder_schema = client.extract_schema_from_source(csv_folder_source)
# Note
# To ingest only from California, then filter_pattern = "California/.*/training.*"
# To ingest only from California & Arizona, then filter_pattern = "(Arizona|California)/.*/training.*"

# Register the feature set
my_feature_set = project.feature_sets.register(csv_folder_schema, "feature_set_name",
primary_key=["key_name"])

# Ingest to cache
my_feature_set.ingest()

# Retrieve feature set
ref = my_feature_set.retrieve()
ref.download()
```

### Example 3: directory structure (no date folder)

- bucket_root
  - nested_folder/
    - California
      - training_data/ -> Sub Folder
        - data.csv
    - Arizona
      - training_sample/ -> Sub Folder
        - data.csv
    - Texas
      - training_sample/ -> Sub Folder
        - data.csv

**Python**

```
from featurestore import Client, CSVFolder

# Initialise feature store client
client = Client("ip:port")
client.auth.login()

# Set project specifics
project = client.projects.create("demo")

# Create the csv folder source
csv_folder_source = CSVFolder(
root_folder="s3a://feature-store-test-data/nested_folder",
filter_pattern=".*/training.*"  # To ingest from all states
)
```

```
csv_folder_schema = client.extract_schema_from_source(csv_folder_source)
# Note
# To ingest only from California, then filter_pattern = "California/training.*"
# To ingest only from California & Arizona, then filter_pattern = "(Arizona|California)/training.*"

# Register the feature set
my_feature_set = project.feature_sets.register(csv_folder_schema, "feature_set_name",
primary_key=["key_name"])

# Ingest to cache
my_feature_set.ingest()

# Retrieve feature set
ref = my_feature_set.retrieve()
ref.download()
```

# Driverless AI MOJO example

### Python

```
from featurestore import Client, CSVFile, DriverlessAIMOJO
from featurestore.core.job_types import INGEST

# Initialise feature store client
client = Client("ip:port")
client.auth.login()

# Set project specifics
project = client.projects.create("demo")

# Create a DAI mojo pipeline source
csv = CSVFile("")
csv_schema = client.extract_schema_from_source(csv)
input_fs = project.feature_sets.register(csv_schema, "input")
input_fs.ingest(csv)

mojo_pipeline = DriverlessAIMOJO("")
mojo_pipeline_schema = client.extract_derived_schema([input_fs], mojo_pipeline)

# Register the feature set
my_feature_set = project.feature_sets.register(mojo_pipeline_schema, "feature_set_name",
primary_key=["key_name"])

# Get ingest job
auto_ingest_job = my_feature_set.get_active_jobs(INGEST)[0]
auto_ingest_job.wait_for_result()

# Retrieve feature set
ref = my_feature_set.retrieve()
ref.download()
```

## Delta table example

### Python

```
from featurestore import Client, DeltaTable

# Initialise feature store client
client = Client("ip:port")
fclient.auth.login()

# Set project specifics
project = client.projects.create("demo")

# Create the delta table source
delta = DeltaTable("")
delta_schema = client.extract_schema_from_source(delta)

# Register the feature set
my_feature_set = project.feature_sets.register(delta_schema, "feature_set_name", primary_key=["key_name"])

# Ingest to cache
my_feature_set.ingest(delta)

# Retrieve feature set
ref = my_feature_set.retrieve()
ref.download()
```

## How to apply a filter on Delta table

### Python

```
from featurestore import DeltaTable, DeltaTableFilter
delta_table_filter = DeltaTableFilter(column=..., operator=..., value=...)
delta_source = DeltaTable(path=..., filter=delta_table_filter)
```

# JDBC example

### Python

```
from featurestore import Client, JdbcTable

# Initialise feature store client
client = Client("ip:port")
client.auth.login()

# Set project specifics
project = client.projects.create("demo")

# Create the jdbc source
jdbc_source = JdbcTable("", "")
jdbc_source_schema = client.extract_schema_from_source(jdbc_source)

# Register the feature set
my_feature_set = project.feature_sets.register(jdbc_source_schema, "feature_set_name",
primary_key=["key_name"])

# Ingest to cache
my_feature_set.ingest(jdbc_source)

# Retrieve feature set
ref = my_feature_set.retrieve()
ref.download()
```

## Joined feature sets example

### Python

```
from featurestore import *
from featurestore.core.job_types import INGEST
import featurestore.core.transformations as t

# Initialise feature store client
client = Client("ip:port")
client.auth.login()

# Set project specifics
project = client.projects.create("demo")

# Create first feature set
csv = CSVFile("")
csv_schema = client.extract_schema_from_source(csv)
fs_1 = project.feature_sets.register(csv_schema, "feature_set_1", primary_key=["key"])
fs_1.ingest(csv)

# Create second feature set
snowflake_table = SnowflakeTable("", "warehouse name", "database name", "schema name", "table name")
snowflake_table_schema = client.extract_schema_from_source(snowflake_table)
fs_2 = project.feature_sets.register(snowflake_table_schema, "feature_set_2", primary_key=["key"])
fs_2.ingest(snowflake_table)

# Create joined feature set transformation
join_transformation = JoinFeatureSets(left_key = "key", right_key = "key")
input_schema = client.extract_derived_schema([fs_1, fs_2], join_transformation)
joined_fs = project.feature_sets.register(input_schema, "joined_feature_set")

# Get ingest job
val auto_ingest_job = joined_fs.get_active_jobs(INGEST)[0]
auto_ingest_job.wait_for_result()

# Retrieve feature set
ref = joined_fs.retrieve()
ref.download()
```

# JSON example

### Python

```
from featurestore import Client, JSONFile

# Initialise feature store client
client = Client("url")
client.auth.login()

# Set project specifics
project = client.projects.create("demo")

# Create the json source
json =
JSONFile("wasbs://data@featurestoretesting.blob.core.windows.net/weather.json", multiline=True)
json_schema = client.extract_schema_from_source(json)

# Register the feature set
my_feature_set = project.feature_sets.register(json_schema,
"feature_set_name", primary_key=["key_name"])

# Ingest to cache
my_feature_set.ingest(json)

# Retrieve feature set
ref = my_feature_set.retrieve()
ref.download()
```

# JSON folder example

## Example directory structure

- bucket_root
  - nested_folder/
    - 2021-05-03/ -> Date Partition
      - training_data/ -> Sub Folder
        - data.json
    - 2021-05-04/ -> Date Partition
      - training_sample/ -> Sub Folder
        - data.json

### Python

```
from featurestore import Client, JSONFolder

# Initialise feature store client
client = Client("ip:port")
client.auth.login()

# Set project specifics
project = client.projects.create("demo")

# Create the JSON folder source
json_folder = JSONFolder(
root_folder="s3a://feature-store-test-data/nested_folder",
filter_pattern=".*/training.*"
)
json_folder_schema = client.extract_schema_from_source(json_folder)

# Register the feature set
my_feature_set = project.feature_sets.register(json_folder_schema,
"feature_set_name", primary_key=["key_name"])

# Ingest to cache
my_feature_set.ingest(json_folder)

# Retrieve feature set
ref = my_feature_set.retrieve()
ref.download()
```

## MongoDb example

### Python

```
from featurestore import Client, MongoDbCollection

# Initialise feature store client
client = Client("ip:port")
client.auth.login()

# Set project specifics
project = client.projects.create("demo")

# Create the jdbc source
mongo_db_source = MongoDbCollection("mongodb+srv://some_cluster.mongodb.net/test",
database="sample_guides", collection="planets")
schema = client.extract_schema_from_source(mongo_db_source)

# Register the feature set
my_feature_set = project.feature_sets.register(schema, "feature_set_name",
primary_key="_id")

# Quick look on the data
my_feature_set.get_preview()
```

## Parquet example

### Python

```
from featurestore import Client, ParquetFile

# Initialise feature store client
client = Client("ip:port")
client.auth.login()

# Set project specifics
project = client.projects.create("demo")

# Create the parquet source
parquet = ParquetFile("")
parquet_schema = client.extract_schema_from_source(parquet)

# Register the feature set
my_feature_set = project.feature_sets.register(parquet_schema, "feature_set_name",
primary_key=["key_name"])

# Ingest to cache
my_feature_set.ingest(parquet)

# Retrieve feature set
ref = my_feature_set.retrieve()
ref.download()
```

# Parquet folder example

## Example directory structure

- bucket_root
  - nested_folder/
    - 2021-05-03/ -> Date Partition
      - training_data/ -> Sub Folder
        - data.csv
    - 2021-05-04/ -> Date Partition
      - training_sample/ -> Sub Folder
        - data.csv

### Python

```
from featurestore import Client, ParquetFolder

# Initialise feature store client
client = Client("ip:port")
client.auth.login()

# Set project specifics
project = client.projects.create("demo")

# Create the parquet folder source
parquet_folder = ParquetFolder(
root_folder="s3a://feature-store-test-data/nested_folder",
filter_pattern=".*/training.*"
)
parquet_folder_schema = client.extract_schema_from_source(parquet_folder)

# Register the feature set
my_feature_set = project.feature_sets.register(parquet_folder_schema, "feature_set_name",
primary_key=["key_name"])

# Ingest to cache
my_feature_set.ingest(parquet_folder)

# Retrieve feature set
ref = my_feature_set.retrieve()
ref.download()
```

## Snowflake example

### Python

```
from featurestore import Client, SnowflakeTable

# Initialise feature store client
client = Client("ip:port")
client.auth.login()

# Set project specifics
project = client.projects.create("demo")

# Create a snowflake table source
proxy = Proxy("url", "port", "username", "password")
snowflake_table = SnowflakeTable("", "warehouse name", "database name", "schema name",
"table name", insecure=False, proxy=proxy, role="role", account="account name")
snowflake_table_schema = client.extract_schema_from_source(snowflake_table)

# Create a snowflake query source
snowflake_query = SnowflakeTable("", "warehouse name", "database name", "schema name",
query="")
snowflake_query_schema = client.extract_schema_from_source(snowflake_query)

# Register the feature set
my_feature_set_1 = project.feature_sets.register(snowflake_table_schema, "feature_set_1",
primary_key=["key_name"])
my_feature_set_2 = project.feature_sets.register(snowflake_query_schema, "feature_set_2",
primary_key=["key_name"])

# Ingest to cache
my_feature_set_1.ingest(snowflake_table)
my_feature_set_2.ingest(snowflake_query)

# Retrieve feature set
ref = my_feature_set_1.retrieve()
ref.download()
ref = my_feature_set_2.retrieve()
ref.download()
```

## Spark pipeline example

### Python

```
from featurestore import Client, CSVFile, SparkPipeline
from featurestore.core.job_types import INGEST

# Initialise feature store client
client = Client("ip:port")
client.auth.login()

# Set project specifics
project = client.projects.create("demo")

# Create source for input feature set
csv = CSVFile("wasbs://featurestore@featurestorekuba.blob.core.windows.net/training.csv")
# Extract schema
schema = client.extract_schema_from_source(csv)
# Register input feature set
input_fs = project.feature_sets.register(schema, "input")
# Ingest the input feature set
input_fs.ingest(csv)

# Define Spark pipeline transformation
spark_pipeline = SparkPipeline("pipeline_path")
# Extract schema
schema = client.extract_derived_schema([input_fs], spark_pipeline)
# Register the feature set
my_feature_set = project.feature_sets.register(schema, "feature_set_name",
primary_key=["state"])

# Get ingest job
val auto_ingest_job = my_feature_set.get_active_jobs(INGEST)[0]
auto_ingest_job.wait_for_result()

# Retrieve feature set
ref = my_feature_set.retrieve()
ref.download()
```

# Admin Transfer Ownership Example

### Python

```
from featurestore import Client
from featurestore.core.access_type import AccessType
from grpc import RpcError

# Initialise feature store client using a user that has admin access
client = Client("ip:port")
client.auth.login()

# Old and new email to transfer the user's projects
old_email = "old@example.com"
new_email = "new@example.com"

# Find all the projects the user is the owner of.
owner_projects = list(client.projects.admin.list(user_email=old_email,
required_permission=AccessType.OWNER))

# Remove the old owner and add the new owner.
for project in owner_projects:
try:
project.add_owners([new_email])
except RpcError as error:
if "is already 'OWNER'" in error.details():
print(f"{new_email} is already OWNER")
project.remove_owners([old_email])
print(f"Transferred ownership from {old_email} to {new_email} for project {project.name}")
```

# Supported data sources

Data must first be ingested into Feature Store before it can be used. Ingesting is the act of uploading data into Feature Store.

Feature Store supports reading data from the following protocols:

- s3 (internally reusing s3a client)
- s3a
- wasbs (encrypted) and wasb (legacy)
- abfss (encrypted) and abfs (legacy)
- http/https (data gets uploaded to internal storage)
- drive (to read files from H2O Drive)
- gs to read files from Google Cloud Storage

**Note:** Due to technical limitations of underlying libraries, reading from Google Cloud Storage isn't supported when FeatureStore is configured to utilize Google Cloud offline storage.

## CSV

CSV file format. Supported path locations are S3 bucket, Azure Blob Storage, HTTP/HTTPS URL and H20 Drive.

**User API:**

**Python**

**Parameters**:

- `path: String` - path to csv file
- `delimiter: String` - values delimiter

```
source = CSVFile(path=..., delimiter=...)
```

## CSV folder

CSV Folder source. Supported path locations are S3 bucket and Azure Blob Storage.

**User API:**

**Python**

**Parameters**:

- `root_folder: String` - path to the root folder
- `delimiter: String` - values delimiter
- `filter_pattern: String` - Pattern to locate the files. To match the files at depth "N", the filter pattern must contain N expressions separated by "/" where each string is either an exact string or a regex pattern.
- For example: `filter_pattern="data/.*/.*/.*comp/.*"` will match this file "data/1996-03-03/1/1679-comp/hello.json".

```
source = CSVFolder(root_folder=..., delimiter=..., filter_pattern=...)
```

## Parquet

Parquet file format. Supported path locations are S3 bucket, Azure Blob Storage, HTTP/HTTPS URL and H20 Drive.

**User API:**

**Python**

**Parameters**:

- `path: String` - path to parquet file

```
source = ParquetFile(path=...)
```

## Parquet folder

Parquet folder source. Supported path locations are S3 bucket and Azure Blob Storage.

**User API:**

**Python**

**Parameters**:

- `root_folder: String` - path to the root folder
- `filter_pattern: String` - Pattern to locate the files. To match the files at depth "N", the filter pattern must contain N expressions separated by "/" where each string is either an exact string or a regex pattern.
- For example: `filter_pattern="data/.*/.*/.*comp/.*"` will match this file "data/1996-03-03/1/1679-comp/hello.json".

```
source = ParquetFolder(root_folder=..., filter_pattern=...)
```

## JSON

JSON file format. Supported path locations are S3 bucket, Azure Blob Storage, HTTP/HTTPS URL and H20 Drive. Different types of JSON formats are supported. Read more here to learn what types of JSON files are supported. By default `multiline` is set to `False`.

**User API:**

**Python**

**Parameters**:

- `path: String` - path to JSON file
- `multiline: Boolean` - `True` whether the input is JSON where one entry is on multiple lines, otherwise `False`.

```
source = JSONFile(path=..., multiline=...)
```

**Note:** Please keep in mind that a JSON object is an unordered set of name/value pairs. This means that using JSON files for extracting schema can produce a schema with a different order of features than that used in the file.

## JSON folder

JSON folder source. Supported path locations are S3 bucket and Azure Blob Storage.

**User API:**

**Python**

**Parameters**:

- `root_folder: String` - path to the root folder
- `multiline: Boolean` - `True` whether the input is JSON where one entry is on multiple lines, otherwise `False`.
- `filter_pattern: String` - Pattern to locate the files. To match the files at depth "N", the filter pattern must contain N expressions separated by "/" where each string is either an exact string or a regex pattern.
- For example: `filter_pattern="data/.*/.*/.*comp/.*"` will match this file "data/1996-03-03/1/1679-comp/hello.json".

```
source = JSONFolder(root_folder=..., multiline=..., filter_pattern=...)
```

**Note:** Please keep in mind that a JSON object is an unordered set of name/value pairs. This means that using JSON files for extracting schema can produce a schema with a different order of features than that used in the file.

## MongoDB

Data stored in a MongoDb can be accessed by Feature Store as well. For a MongoDb authentication, environment variables

- MONGODB_USER
- MONGODB_PASSWORD will be used to provide user information.
- **User API:**

**Python**

**Parameters**:

- `connection_uri: String` - a MongoDb server URI
- E.g. connection_uri="mongodb+srv://my_cluster.mongodb.net/test"
- `database: String` - Name of a database on the server
- E.g. database="sample_guides"
- `collection: String` - Name of a collection to read the data from
- E.g. collection="planets"

```
source = MongoDbCollection(connection_uri=..., database= ..., collection = ...)
```

## Delta table

Delta table format. Table can be stored in either S3 or Azure Blob Storage.

**User API:**

**Python**

**Parameters**:

- `path: String` - path to delta table
- `version: Int` - (Optional) - version of the delta table
- `timestamp: String` - (Optional) - timestamp of the data in the table
- `filter: DeltaTableFilter` - (Optional) - Filter on the delta table

```
source = DeltaTable(path=..., version=..., timestamp=..., filter=...)
```

**DeltaTableFilter API:**

**Python**

**Parameters**:

- `column: String` - name of the column
- `operator: String` - operator to be applied
- `value: String|Double|Boolean` - value to be applied on the filter

```
delta_filter = DeltaTableFilter(column=..., operator=..., value=...)
```

### Supported operators

The following are the supported operators : `==`, `<`, `>`, , and .

### Valid parameter combinations

1. `Path`
2. `Path`, `Version`
3. `Path`, `Version`, `Filter`
4. `Path`, `Timestamp`
5. `Path`, `Timestamp`, `Filter`
6. `Path`, `Filter`

## JDBC

JDBC table format. Currently, we support the following JDBC connections:

- PostgreSQL
- Teradata

**User API:**

**Python**

**Parameters**:

- `connection_url: String` - connection string including the database name
- `table: String` - table to fetch data from
- `query: String` - query to fetch data from
- `partition_options: PartitionOptions` - (Optional) parameters to enable parallel execution. These are applicable only when `table` is specified
- `PartitionOptions` constitutes : `num_partitions`, `partition_column`, `lower_bound`, `upper_bound`, `fetch_size`

```
source = JdbcTable(connection_url=..., table=..., partition_options=PartitionOptions(num_partitions = ..., pa
source = JdbcTable(connection_url=..., query=...)
```

The format of the connection URL is a standard JDBC connection string, such as:

- For Teradata, `jdbc:teradata://host:port/database`
- For PostgreSQL, `jdbc:postgresql://host:port/database`

The database is a mandatory part of the connection string in the case of Feature Store. Note that only one of table or query is supported at the same time. Additionally, `PartitionOptions` can only be specified with table. These options must all be specified if any of them is specified. They describe how to partition the table when reading in parallel from multiple workers. `partitionColumn` must be a numeric, date, or timestamp column from the table in question. Notice that `lowerBound` and `upperBound` are just used to decide the partition stride, not for filtering the rows in table. All rows in the table will be partitioned and returned. This option applies only to reading.

## Snowflake table

Extract data from Snowflake tables or queries.

**User API:**

**Python**

**Parameters**:

- `table: String` - table to fetch data from
- `database: String` - Snowflake database
- `url: String` - url to Snowflake instance
- `query: String` - query to fetch data from
- `warehouse: String` - Snowflake warehouse
- `schema: String` - Snowflake schema
- `insecure: Boolean` - if `True`, Snowflake will not perform SSL verification
- `proxy: Proxy object` - proxy specification
- `role: String` - Snowflake role
- `account: String` - Snowflake account name

**Note:** `table` and `query` parameters cannot be configured simultaneously.

```
from featurestore import *
proxy = Proxy(host..., port=..., user=..., password=...)
source = SnowflakeTable(table=..., database=..., url=..., query=..., warehouse=..., schema=..., insecure=...,
proxy=..., role=..., account=...)
```

**Note:** A proxy is an optional argument in the Snowflake data source API. If a proxy is not being used, the proxy configuration can simply be set to `None`.

The use of a proxy is possible for users only if the proxy feature is enabled by the administrator of the Snowflake account. Therefore, it is important to confirm whether proxy support is enabled before attempting to configure a proxy in the Snowflake data source API.

## Snowflake Cursor object

Extract data from Snowflake tables or queries.

**User API:**

The Snowflake Cursor object is currently only supported in the Python client.

**Parameters:**

- `database: String` - Snowflake database
- `url: String` - url to Snowflake instance
- `warehouse: String` - Snowflake warehouse
- `schema: String` - Snowflake schema
- `snowflake_cursor: Object` - Snowflake cursor
- `insecure: Boolean` - if `True`, Snowflake will not perform SSL verification
- `proxy: Proxy object` - proxy specification
- `role: String` - Snowflake role
- `account: String` - Snowflake account name

```
source = SnowflakeCursor(database=..., url=..., warehouse=..., schema=..., snowflake_cursor=..., insecure=...
                         proxy=..., role=..., account=...)
```

**Database snippet**:

Internally, the Snowflake Cursor is converted to `SnowflakeTable` with `query` and is therefore saved in the same format in the database.

## Spark Data Frame

When using Spark Data Frame as the source, several conditions must be met first. Read about the Spark dependencies to understand these requirements.

**User API:**

**Python**

**Parameters**:

- `dataframe: DataFrame` - Spark Data Frame instance

```
source = SparkDataFrame(dataframe...)
```

## Accessing H2O Drive Data

When H2O Drive application is running in the same cloud environment as Feature Store, then user is able to access files that he/she uploaded into H2O Drive. To refer to those files, let's specify the scheme as **drive**. However, due to technical limitations access to H2O Drive files is currently not possible when user is authenticated to Feature Store via PAT token.

**Examples**

**Python**

```
source_1 = CSVFile("drive://example-file-1.csv")
source_2 = CSVFile("drive://my-subdirectory/example-file-2.csv")
```

## BigQuery (Google Cloud)

FeatureStore can extract data from BigQuery tables or queries.

**User API:**

**Python**

**Parameters**:

- `table: String` - table to fetch data from
- `parent_project: String` - (Optional) The Google Cloud Project ID of the table to bill for the export
- `query: String` - query to perform and read its result from
- `materialization_dataset: String` - When a query parameter was specified, then a dataset where the materialized view is going to be created. This dataset should be in same location as the view or the queried tables.

**Note:** `table` and `query` parameters cannot be configured simultaneously.

```
source = BigQueryTable(table=your_dataset.your_table, parent_project=your_project)

sql="SELECT label, count(1)  FROM `your_project.your_dataset.your_table` group by label"
source = BigQueryTable(parent_project=your_project, query=sql, materialization_dataset="your_temporal_dataset
```

An instance of GcpCredentials will be utilized to authenticate to Google BigQuery. See info on GcpCredentials when a specific instance is required.

# Supported derived transformation

Transformation changes the raw data and makes it usable by a model.

## Spark pipeline

Creating a feature set via Spark pipeline. The Spark pipeline generates the data from an existing feature set that you pass in as an input to the pipeline. Feature Store then uploads the Spark pipeline to the Feature Store artifacts cache and stores only the location of the pipeline in the database.

**User API:**

**Python**

**Parameters**:

- `pipeline_local_location: String or Pipeline Object` - you pass the local path to the pipeline or the pipeline object itself. Once the feature set is registered, this parameter contains the path to the uploaded Spark pipeline in the Feature Store artifacts storage.

```
import featurestore.core.transformations as t
spark_pipeline_transformation = t.SparkPipeline("...")
```

## Driverless AI MOJO

Creating a feature set via Driverless AI MOJO. The MOJO pipeline generates the data from an existing feature set that you pass in as an input to the pipeline. Feature Store then uploads the MOJO pipeline to the Feature Store artifacts cache and stores only the location of the pipeline in the database.

**Note:** Only features created from Driverless AI with the `make_mojo_scoring_pipeline_for_features_only` setting are supported in Feature Store.

**User API:**

**Python**

**Parameters:**

- `mojo_local_location: String` - you pass the local path to the pipeline. Once the feature set is registered, this parameter contains the path to the uploaded MOJO pipeline in the Feature Store artifacts cache

```
import featurestore.core.transformations as t

transformation = t.DriverlessAIMOJO(...)
```

## JoinFeatureSets

Creating a new feature set by joining together two different feature sets.

**User API:**

**Python**

**Parameters**:

- `left_key: String` - joining key which must be present in left feature set
- `right_key: String` - joining key which must be present in right feature set
- `join_type: JoinFeatureSetsType` - join type (default: JoinFeatureSetsType.INNER)

**JoinFeatureSetsType**

- JoinFeatureSetsType.INNER - The inner join is the default join in Spark SQL. It selects rows that have matching values in both relations.
- JoinFeatureSetsType.LEFT - A left join returns all values from the left relation and the matched values from the right relation, or appends NULL if there is no match.
- JoinFeatureSetsType.RIGHT - A right join returns all values from the right relation and the matched values from the left relation, or appends NULL if there is no match.

- JoinFeatureSetsType.FULL - A full join returns all values from both relations, appending NULL values on the side that does not have a match.
- JoinFeatureSetsType.CROSS - A cross join returns the Cartesian product of two relations.

```
import featurestore.core.transformations as t

transformation = t.JoinFeatureSets(left_key=..., right_key=..., join_type=...)
```

**Note:** During join transformations, Feature Store perform inner joins

### Python Sparkling Water

1. In a Python environment, pip install the featurestore client.
2. Download spark and pysparkling by following the instructions from the Sparkling Water documentation.
3. Start the pysparkling session with the Spark dependencies.

```
./bin/pysparkling --jars <spark dependency jar file>
```

Example:

```
from featurestore import Client
ref = fs.retrieve()
data_frame = ref.as_spark_frame(spark)

# sparklingwater
from pysparkling import *
hc = H2OContext.getOrCreate()
from pysparkling.ml import H2OGLM
estimator = H2OGLM(labelCol = "RainTomorrow")
model = estimator.fit(data_frame)
```

The Feature Store can be integrated with h2oGPTe and deployed with h2oGPTe enabled. Enterprise h2oGPTe is an AI-powered search assistant powered by H2O LLM that helps you find answers to questions about your documents, websites, and workplace content.

For detailed information on how to integrate h2oGPTe with the Feature Store, see H2O GPTE Integration.

When you create or update projects and feature sets within the Feature Store, and you have this integration active, it organizes your data within h2oGPTe. It creates a collection for a project or document of a feature set within h2oGPTe. It allows you to interact with your data from the Feature Store through h2oGPTe.

# Key terms

This page houses the keys terms used throughout this documentation.

## Classifier

Classifiers are used for recommending features based on pattern matching amongst different feature sets. For example, if you provide the pattern to Feature Store on feature set A that a column with 5 digits is a zip code, then Feature Store will be able to identify any single column in feature set B that has 5 digits as a zip code (provided that there are not multiple columns with 5 digits).

## Consumer

This is a user with view-only rights.

## Core

The Feature Store Core is an application within Feature Store and has multiple duties. We use the Core to create the features for the database. It is also used to trigger the start of data manipulation tasks on the Spark cluster. It also performs authentication and queries for authorization permissions.

## Data source

A data source is the file you ingest into Feature Store.

## Derived feature set

When you apply transformations to a feature set, it will create a derived (new) feature set.

## Editor

This is a user that has been given permission by the owner allowing them to view and update a project and its contents.

## Extraction

Extraction is the act of retrieving the schema from a data source.

## Feature

Features are highly curated data. They are used to enhance the performance of ML models for training models and model prediction.

## Feature set

A feature set is a collection of features.

## Ingesting

Ingesting is the term used to describe the act of loading a data source into Feature Store.

## Joining

This is the act of combining two different feature sets.

## Keys

Keys are used to search for a specific item in your data. Primary keys use in Feature Store have to be a unique value (e.g., a social security number).

### Offline Feature Store

Offline Feature Store is responsible for storing features based on big data. It stores all the metadata about feature set schema, features, etc.

### Online Feature Store

Online Feature Store is responsible for working with feature sets with which needs to be stored and obtained very quickly.

### Owner

This is the person who created the project. They can view, edit, and update a project and its contents without any extra permissions. Owners can give permission.

### Permission

Permission dictates on which level you can interact with entities in Feature Store. Users with different permissions are allowed to do more or less privileged actions.

### Project

A project is used to store feature sets. It is the highest level of the organizational hierarchy. Projects are the first thing that must be created when using Feature Store because they house all of the information the data sources, schemas, feature sets, etc.

### Registration

Registration is the act of registering feature sets into Feature Store. It is the command that creates a new feature set.

### Retrieving

Retrieving is the action of re-acquiring your ingested data. You can filter data by `start_date_time` and `end_date_time`.

### Reverting

Reverting is the removal of ingested data. The act of reversion creates a new version of the feature set with that data removed.

### Schema

A schema represents the features of the feature set. It is extracted from a data source.

### Serialization and deserialization

Serialization is the process of converting data into a series of bytes that can be stored and transmitted between objects. Deserialization is the reverse process where you create objects from a sequence of bytes.

### Transformation

A transformation is a change to the raw data that makes it usable by a model. There are different types of transformations, like changing the data format.

## Version 2.1.0 (16-07-2025)

### New features

- Administrators can now limit which user roles are allowed to create projects.
- Administrators can now add or remove project access for other users.
- Added `client.projects.admin.list()` API, which allows administrators to retrieve a list of all projects, regardless of ownership.
- Added support for transferring project ownership. For details, see Admin Transfer Ownership Example.
- Feature Set downloads can now be disabled from the UI using a configuration setting. This option is available only to the Feature Store administrator.
- Upgraded Spark to version 3.5.6.

### Fixes

- Removed the Feature Views and Machine Learning Datasets capability.
  - The Feature View API has also been removed.

## Version 2.0.2 (09-06-2025)

### Fixes

- Fixed an UI issue preventing special fields to be used during schema registration.
- Fixed an issue preventing derived transformations from being created in the UI.

## Version 2.0.0 (22-05-2025)

### New features

- Upgraded Spark to version 3.5.5.
- Added support for Workload Identity Authentication for Azure Data Lake Gen 2 Storage and Azure PostgresQL.
- Added possibility to use path-style access for S3 in shared storage.
- Added support for `SASL_PLAINTEXT` for Kafka message broker.

### Fixes

- Switched base images to Chainguard to reduce CVEs.

## Version 1.2.0 (25-01-2024)

### Fixes

- Preview now correctly show entries per selected feature set version
- Entry on `My Access` tab on a feature set did not show option to request permission in case user does not have any permission
- Fix beamer location bug in UI
- Fix error swallowing on `Create project` and `Create feature set` pages
- Expose missing preview configuration in Helm values
- Fix issue leading to wrong schema in case feature type was modified in schema on Python CLI
- Redis connection in online store is now correctly refreshed
- Fix logging configuration for spark driver and executor pods
- When maximum session length is reached, logout and redirect user to login interface
- Cleanup orphaned records on Redis database
- Switch installer test image to be Alpine based to reduce vulnerabilities
- Fix several online store memory leakage issues
- Fix issue with job error not being propagated in case Linkerd was enabled
- User should not be able to select day before today when creating personal access tokens

### New features

- New method used to print schema in SQL format in CLIs created
- GPTe integration. When project or feature set is created or updated, information are send as a collection/document to GPTe

- Expose button in UI to manually trigger online to offline sync
- Introduce H2O Drive as a data source
- Ability to use H2O Drive and GPTe integration even when use is authenticated using personal access token
- Ability to use GCP as offline and supporting storage
- Ability to remove major feature set version
- Automatically roll helm deployments in case config map or secrets change
- Show pending permissions on Dashboard in UI
- Publish Conda packages for Python CLI

## Version 1.1.2 (30-11-2023)

### Fixes

- Fix regression which prevented using private certificate authorities
- Fix `CVE-2022-1471` by upgrading to Spring 3.2 and Spark 3.5

## Version 1.1.1 (15-11-2023)

### Fixes

- Fix several issues when Snowflake is used as storage backend
- Improve parallelism of processing message in online store
- Fix installer tests

## Version 1.1.0 (09-11-2023)

### Fixes

- Skip leading and trailing whitespaces in column names when parsing CSV
- Fix error when Azure Gen2 was used as supporting storage and S3 as offline storage
- Fix storage backend naming on helm level
- Fix wrong paths in several endpoints used in the UI
- Fix issue that some feature set is created twice whilst requesting higher permissions
- Update lazy ingest documentation and explain the motivation better
- Improve classifier and recommendation api documentation
- User is now able to ingest in the UI without selecting the cloud provider
- Fix issue leading to no owner being displayed in the UI on feature set tab
- Fix bug leading to `Unexpected end of JSON input` when inspecting preview of feature set in the UI
- Fix all fixable CVEs to the date of the release

### New features

- Ability to use Snowflake as the offline feature set storage
- Introduce access tab on the UI where owner can see all users with access to the feature set or project
- Add support for requesting higher permission in UI if user already has lower permissions
- Show list of derived and parent feature sets on feature set page in the UI
- Ability to use AWS SSO credentials for S3 data sources
- Ability to revert ingest in the UI
- Mark feature set as derived if it is derived in the UI
- Introduce method and API to perform Z-ordering on a feature set
- Feature Store deployment no longer requires any cluster roles
- Add support for Azure MSI authentication for offline store
- Feature Store now respects the limitations specified by the tier
- Add ability to withdraw pending permission request
- If user requested more than one permission and the higher one is approved, the lower one are also automatically approved
- Ability to ingest and retrieve from online feature store in UI
- Ability to use key-pair authentication for Snowflake offline store and Snowflake data source
- Expose method in CLI on feature set level to get specific version of that feature set
- Add ability to register derived feature sets in the UI

- Run database migration script as part of init script instead of directly in the pods. This way database migrations are not affected by k8s startup probes and can finish successfully even if they take longer time.

## Version 1.0.0 (27-09-2023)

### New features

### CLI

- Ability to get latest minor version for specific major version of feature set in
- Ability to see all feature set pending and manageable reviews in scope of a specific project
- Expose method to open website with specific feature set or project from client
- Ability to change time travel column during creating new major version
- Ability to use key-pair authentication for Snowflake data source
- Ability to extract schema and ingest data source from http/https locations
- Secured connection is not used by default

### UI

- Feature Store version is showed on the UI
- Show number of pending or manageable reviews next to the title in the left bar
- Ability to list and see versions of feature sets
- Ability to create new major version of feature set
- Ability to schedule data ingests
- Ability to download list of features in CSV format
- Share link to Feature Store documentation in UI left bar
- Ability to extract schema and ingest data source from http/https locations
- User is redirected to feature set in case clicking on `View` button on feature set in review in states `Created` and `Approved`

### Backend and others

- Ability to restrict access to Feature Store based on presents of specific JWT roles
- Improve performance of online Feature Store
- Ability to read from public S3 buckets even though credentials(correct or invalid) are provided
- Introduce API to display parent and child derived feature sets
- Introduce ability to test basic functionalities of feature Store directly using `helm test`
- Alpha support for using Snowflake as storage for offline Feature Sets
- Simplify changing logging configuration for whole Feature Store stack
- Spark jobs no longer requires cluster-role
- Expand API for getting user permissions to return permissions only for specific resource
- Document better time to live and meaning of fields marking feature sets as sensitive

### Fixes

- Resolve all fixable vulnerabilities to the date of the release
- Fix bug in Spark Operator caused by parallel updates of spark resources
- During schema creation in UI, data type and feature type enum was not populated in certain browsers
- Fix UI bug that user was able to request same permission more than once
- Ability to use both protocols for `s3` and `s3a` when working with data sources in S3
- About item was not visible in the UI
- Other users were able to see not approved feature sets before ingestions, this is now fixed. Only owner cam see feature sets prior their approval
- Select all files to download by default when retrieving from UI
- Fix but that user was able to schedule ingestions on feature sets without access to that feature set
- Do not show button to delete artifact in UI if user does not have editor permissions
- Scope is optional field when retrieving from UI now
- Fix bug when UI was not showing the error in case job failed
- Use fonts in UI from ui-kit to allow air-gaped UI usage
- Expiration for working with feature sets using user spark sessions was hard-coded to 1 hour. Now it is configurable
- Fix bug when UI was not respecting links with specific paths

- Fix bug in UI where page was not persisted after refreshing
- Internal column (ingest_id) was leaking to retrieved files which is now fixed
- In the retrieve example notebook, the dependencies are now pointing to their maven locations
- Allow unlimited expiration for personal access tokens
- Fix bug that UI did not refresh after updating feature or feature set
- Fix bug in UI caused when opening list of versions or list of ingestions on not yet reviewed feature set
- Toggle `Use Time Travel Column as Partition` was incorrectly places in UI

This version introduced breaking changes and is not compatible with older CLIs.

## Version 0.19.3 (21-08-2023)

**Fixes**

- After the helm changes in 0.19.2, the IAM connection was not recreated correctly which is now fixed

## Version 0.19.2 (17-08-2023)

**New features**

- Ability to pass affinity specification to Feature Store pods via Helm
- Ability to obtain JDBC connection string for core PostgreSQL database from existing secret
- Ability for namespace override in Helm
- Add `telemetry.cloud.h2o.ai/include: true` annotation to Spark driver and executors
- Add ownership attribution labels/annotations to feature store resources

**Fixes**

- All fixable vulnerabilities at the time of the release have been addressed
- Fix incorrect mapping of feature set flow field in Python CLI from its internal representation

## Version 0.19.1 (24-07-2023)

**Fixes**

- Read OAuth token from correct field after upgrade to latest Fabric8 Kubernetes library
- Fix issue with removing artifacts when using Azure as storage backend

## Version 0.19.0 (20-07-2023)

**Fixes**

- All fixable vulnerabilities at the time of the release have been addressed
- Better handling of feature containing dot in their name
- Fix bug where record was never stored to online store in case Postgresql was used as backend
- Fix several UX issues when displaying UI on small screens
- Fix non-deterministic output of versionChange flag on feature set and feature entities during updates
- Fix auth problems when using folder data sources
- Fix issue when user could not create personal access token with same name different user used
- Fix navigation bar to show all available cloud components
- Fix handling public data sources in UI
- Fix issue where files on Gen2 azure store were not accessible using SAS token
- Improve error message handling for out of memory issues
- Prevent generating pre-signed urls to Spark temporary files
- Fix issue with displaying job id in UI which contained the `x` character
- Fix issue where `Canceled` stated wasn't properly displayed in jobs list on UI
- Fix several spelling issues in the UI
- Add missing time travel column to the feature set page on the UI
- Fixing issue where backend tried to delete project first before deleting the feature sets inside the project
- Fix issue with ingest history not displaying correctly in UI for derived feature sets
- Ensure consistency between data in the storage and the information in the database
- Ensure documentation for log configuration is up-to-date

- Fix problem where spark properties passed as extra spark options to operator contained space characters

**New features**

- Implement Notifications in the UI
- Ability to create, list and revoke personal access tokens in the UI
- Ability to download pre-generated retrieve notebook via CLI and UI
- Implement review process in the UI
- Ability to ingest and retrieve from UI
- Expose ingest history in the UI
- Ability for Feature Store administrator to specify maximum duration of a personal access token
- Ability to filter jobs based on their types in the UI
- Use stable API for HPA in Feature Store Helm charts
- Introduce expiration date on a feature set drafts

## Version 0.18.1 (14-06-2023)

**Fixes**

- Fix telemetry error causing pod restart after successfully sent message
- Fix failure when user credentials already exists during a job
- Share more logs in case sending message to telemetry service is not successful
- Fix job scheduling in case of multiple parallel ingest jobs
- Fix migration related to uploaded artifacts

## Version 0.18.0 (01-06-2023)

**Fixes**

- Fix scheduling of ingest and revert jobs in case there is more then 1 job on the queue
- Fix bug leading to error during extract schema in UI
- Change spark app status to cancelled directly when there is no pod for that job
- Use string instead of UUID for project history
- Fix SQL constraint violation when deleting job related to feature set draft
- Strip extra spaces in URL in Python and Scala CLI
- Fix position of search bar in UI on feature set pages
- Housekeeping of uploaded artifacts

**New features**

- Ability to List jobs on UI
- Ability to see progress of jobs on UI
- Expose updated by field on project and feature set CLI entities and APIs
- Expose number of retrievals on popular feature sets in UI

## Version 0.17.0 (25-05-2023)

**Fixes**

- Improved health check for Redis
- Several improved validations to register feature set UI flow
- Handle case where spark driver is deleted by something else then operator
- Fix feature set permission promotion when higher or equal project permission is created
- Fix issue with jobs failing due to having large inputs
- Generate `GetFeatureSet` even when obtaining a listable feature set
- Fix issue with UI global search being extremely slow on high number of feature sets
- Fix dashboard computation being slow when high number of feature sets exists
- Fix feature view deletion bug
- Fix issue with incorrect pooling of PostgreSQL connections in online store
- Fix issue where incremental statistics were not computed for features containing dot in their names
- Fix trace id propagation on internal exception

- Do not compute Spark telemetry details on a closed Spark session
- Prevent storing internal columns into the feature set preview
- Fix SQL constraint violation during deleting derived feature sets
- Fix SQL constraint violation when deleting parent job

**New features**

- Azure Gen2 Jar is now published to maven central
- Introduce feature set flow configuration - user can configure synchronization between online and offline stores
- Implement recently visited projects and feature sets
- Implement popular feature sets
- Integrate with H2O AI Cloud logging service
- Introduce PostgreSQL and remove Mongo as online backend database
- IAM support for Redis
- Helm charts provide more granular control whether IAM should be used or not
- Expose method in CLI to open feature store web
- Implement pinned feature sets
- Implement UI home page
- Expose ingested records count in the ingest history api
- Support for passing security context for containers
- Expose button to trigger online materialization on UI
- Allow specifying join type in derived feature sets
- Allow to select join type in feature views
- Expose filter on feature sets to be reviewed
- Expose data source, time of ingestion, scope and user who performed the ingestion on the ingest history api

## Version 0.16.0 (26-04-2023)

**Fixes**

- Do not create a new version of a feature set or feature in case nothing has changed during an update call
- Share warning message if join hasn't joined any data during derived feature set transformation
- Improve credentials and permission sections of documentation to be more explicit
- Improve cleaning of ill k8s resources
- Implement transitive deletion of derived feature sets
- Remove left-overs from documentation regarding MongoDB
- Improve lazy ingest message to be more explicit
- Improve telemetry health-checks
- Improve Kafka health-checks
- Fix bug in Python CLI schema extraction logic regarding nested data types
- Remove transitive dependencies from Azure Gen2 dependencies jar
- Update the dependencies section of the documentation to contain valid versions
- Project in UI should not be locked and secret by default
- Fix typo in helm charts affecting notifications configuration
- Fix handling dates prior year 1900
- Fix bug in the online store in case the data type of feature is Timestamp, and that feature is also a time travel column
- Improve error handling in UI

**New features**

- Ability to create feature sets in UI
- Ability to order project, feature sets or features based on specific fields in UI
- Introduce API to cancel a job and improve handing of cancelled jobs
- Introduce API to download a pre-generated notebook demonstrating retrieve flow
- Introduce API to upload and download artifacts to a specific feature set
- Support for deleting of major feature set versions
- Introduce approval process in CLIs and backend
- Introduce support for LinkerD
- Expose API to mark/unmark feature as target variable

- Display number of ingested records on CLI entities and in the UI
- Introduce API for popular feature sets
- Introduce API for recent projects and recent feature sets
- Introduce configuration for dear letter in Kafka
- Improve schema representation on Python and Scala CLI
- Expose monitoring and custom data on feature schema

## Version 0.15.0 (21-03-2023)

### Fixes

- Throw user-friendly exception if CLIs are trying to call non-existent API
- Dashboard API returning wrong number of features
- Documentation now clearly states what type of join is used in Feature Store
- Follow Spark logic to parsing timestamps to have more generic inputs for online ingestion
- Provide stronger validation for DeltaTable data source filters
- Schedule interval is now human-readable on CLIs
- Fix redirection message in browser after login
- Fix data back-fill in case the original data had not explicit time travel column
- Feature Stores allows auth flow for users without name and e-mail now
- Fix deletion of historical feature view when feature view was deleted
- Fix deletion of jobs related to project ids
- Provide user-friendly error in case connection to API service failed from Python and Scala CLI
- Handle internal failure during online-offline sync when feature set was deleted in the meanwhile

### New features

- Internal database used to store meta-data was changed from Mongo to Postgres
- Introduction of project history
- Integration with H2O AI cloud discovery service
- MongoDB collection data source introduced
- Add possibility to change partition columns when creation a new feature set version
- Expose number of ingested records on Feature Set entity in CLIs
- Introduce `Viewer` permission. See Permissions for more details.
- Send notification after PAT login
- Docusaurus is used as documentation tooling
- Introduce API to pause and resume scheduled ingest task
- Scheduled ingest tasks is paused automatically if it fails subsequently based on user defined boundary

## Version 0.14.4 (28-02-2023)

### Fixes

- Migration fixes to ensure compatibility with Driverless AI
- Time travel column, partition columns and primary keys are case-insensitive during their specification

### New features

- Lookup for features in CLI is now case-insensitive

## Version 0.14.3 (28-02-2023)

### Fixes

- JWT token no longer requires expiration date to ensure consistent experience in H2O AI cloud

## Version 0.14.2 (27-02-2023)

### Fixes

- Sensitive consumer permission is not being granted if user is regular consumer

## Version 0.14.1 (20-02-2023)

### Fixes

- Fix online materialization on feature sets with features containing dot in their names

## Version 0.14.0 (30-01-2023)

### Fixes

- Provide error if timezone is incorrect in scheduler API
- Fix “None.get” bug during subsequent update of a feature
- Fix online materialization on timestamp column with data representing date only
- Fix online retrieval where primary key is of type timestamp with data representing date only
- Replace prints by logger in python CLI
- Fix and re-introduce `disable-api.deletion` under new more generic API

### New features

- Add tooltips to `secret` and `locked` in UI
- Add docstrings to all Python CLI methods
- Show values of auto generated time travel column in human readable format

Please see Migration guide for changes and deprecations.

## Version 0.13.0 (05-01-2023)

### Fixes

- Avoid page reload every time access token expires
- Fix OOM error in core service while deleting feature sets with 1mln+ files

## Version 0.12.2 (14-12-2022)

### Fixes

- Disable Locked projects in the Feature Store website

### New features

- Add Google Tag Manager (GTM) support into Feature Store website
- Add custom string representation for all entities used in CLI

## Version 0.12.1 (06-12-2022)

### New features

- Feature Store UI as integral part of Cloud design

## Version 0.12.0 (25-11-2022)

### Fixes

- Unable to read data from S3 folder data source with path ending with slash
- Publish Java GRPC API with Java 11 instead of Java 17
- Fix rare bug in operator caused by its restart/redeployment leading to hanging jobs
- Fix bug caused by improper handling of trailing slash in S3 data source path
- Handle expired logging session more gracefully in CLIs
- Properly handle different schema exception in case of spark data frame ingestion

**New features**

- Expose access control in documentation and Python & Scala clients
- Ability to create a new major version of feature set with data back-filled from older version
- Display Navigation bar in UI
- Support for custom certificate authorities in all Feature store components

## Version 0.11.0 (09-11-2022)

**Fixes**

- Handle missing region in AWS credentials
- Retrieve correct version of feature set after lazy ingest
- Fix sample classifier documentation
- Improve documentation for statistics computation
- Start respecting consumer and sensitive consumer permission from projects on feature sets
- Wait for MLDataset materialization
- Display feature set and project owners in UI
- Allow reverting ingests only created after derived feature set creations
- Fix 404 error when clicking of feature from Search All List
- Correctly display empty statistics on feature set in UI
- Fix statistics re-computation after revert
- Fix preview to return the preview instead of printing
- Rename TrainingDataset to MLDataset
- Enforce order of parent feature sets information
- Fix permission check while getting feature from get feature endpoint
- Start respecting minor versions of feature set

**New features**

- Ability to specify reason during approval/rejection/revocation of permission on UI
- Ability to edit project, feature set and feature meta-data in UI
- Introduce online MLDatasets
- New endpoints for updating project, feature set and features
- Automatically detect categorical variables during statistics computation
- Add transformations functions to feature view and MLDatasets
- Expose API to get current permission of the project or feature set
- Ability to lazy ingest into a feature set

Please see Migration guide for changes and deprecations.

## Version 0.10.0 (06-10-2022)

**Fixes**

- Fix bad computation of time travel scope
- Better message during create new version in case version already exists

- Remove default partitioning based on time travel column (the parameter time_travel_column_as_partition is still respected)
- Run ingest job on all available executors
- Fix issue when nested schema elements are not updated
- Document what formats are valid for time travel column format
- Fix running a MOJO derived feature set in case the MOJO results in same output column as is the input
- Sanitize user emails to support emails with special characters
- Return empty response in case no classifiers are defined on a feature set
- Fix problem of CLI failing in case empty AWS region is provided
- Fix converting SampleClassifiers to internal proto representation
- Fix ingest scope computation in case previous feature set time travel scope is overlapping
- Fix empty last update on fields on projects and feature sets after creation
- Preserve order of columns in joined dataframe to fix joined derived feature set random ingestion errors

**New features**

- Alpha release of UI
- Capability to schedule ingests
- Feature view and training dataframe capabilities
- GRPC api exposing permissions and approval process
- Re-implement feature set preview and make sure it is available immediately without running a job
- Expand notifications to more methods (see Events for more information)
- Add md5 checks to validate integrity of uploaded pipelines to Feature store

## Version 0.9.0 (07-09-2022)

**Fixes**

- Fix ingest of data from encrypted S3 buckets
- Ensure that ingest on non-latest major version does not update latest feature sets collection
- HPA support for feature store services
- Add TLS and IAM support to telemetry kafka stream
- Fix python retrieve holder to support calling preview and download in the same retrieve instance
- Add validation for specification of recommendation percentage specification
- Preview does not respect `start_date_time` and `end_date_time`

**New features**

- Ingest API now ingest only unique rows. Please check migration guide for more details.
- Expose custom data on feature level
- Add support for compound primary key
- Search API for projects/feature sets and features for UI

## Version 0.8.0 (05-08-2022)

**Fixes**

- Fix creation of join derived feature sets with space in name
- Transaction in job handler commit instead rollback when some exception is thrown
- Rollback transaction when error occurs during updating job output
- Use file instead of env variable for job input to handle big inputs
- Fix revert on derived feature sets created using aggregation pipelines
- Fix bug preventing ingestion using specific spark pipelines
- Raise error during registration if feature set contains invalid characters
- Fix mojo derived feature set in case column contains a dot
- Fix bug where schema parser behaves differently on CLI and backend
- Support online materialization also on static data (without explicit time travel column)
- Fix retrieval of parent feature set during derived(join) feature set ingestion
- Fix join key validation in join feature sets to be case-insensitive
- Fail extract schema job in case `_corrupt_record` is computed

**New features**

- Pagination on projects and feature sets
- Improve notification API to provide more details
- Telemetry implementation
- Expose Dashboard endpoints in GRPC API
- API to delete and update recommendation classifiers

## Version 0.7.1 (02-08-2022)

**Fixes**

- Support feature sets with high number of features
- Fix patch schema method to correctly work on nested structs

## Version 0.7.0 (07-07-2022)

**New features**

- Recommendation engine
- Multi project search
- Validating regex as part of folder data sources before run job
- Rename (deprecate) the partitionPattern field in CSVFolder/ParquetFolder/JsonFolder/OnlineSource to filterPattern
- Ingestion validation to derived feature set operations

**Fixes**

- Ingesting History when a major version happens
- Creating spark pipeline file in databricks environment
- Migration for historical feature set

## Version 0.6.0 (15-06-2022)

**New features**

- Removal of deprecated derived data sources
- Timezone independent personal access tokens expiration
- GRPC API is now versioned
- Allow read the folder data sources with empty filter

**Fixes**

- Use projection for feature set and project during deletion to avoid obtaining full object from database

## Version 0.5.0 (07-06-2022)

**New features**

- Introduce derived feature sets, please refer to documentation and migration guide for more information
- Introduce concept of admin to be able to manage Feature Store via admin API
- Support for Minio as source of data

**Fixes**

- Fix bug in statistics job quantiles computation on empty data
- Fix problem with incorrect detection of bad data in time travel column
- Disable version checks
- Fix fullyQualifiedFeatureName migration
- Fix problem with data source having spaces in their names
- Fix hanging of jobs submitted at the same time
- Fix idempotency during deleting online feature set

## Version 0.4.0 (24-05-2022)

**New features**

- Ability to use Mongo as Online Feature Store backend
- Give possibility to define custom log4j property files to Feature Store services
- Support for IAM roles when reading data from S3 data sources
- Support for reading data from public S3 data sources
- Document usage of feature store notifications
- Hide feature set statistics for non-sensitive consumers
- Update CRD automatically during Helm release

**Fixes**

- Feature set scope wasn't emptied when new major version was created
- Online to offline sync fails because of the schema mismatch
- Fix problem where job finished with state 1
- Prevent executing update on already finished job
- Don't get job output from Mongo if not required
- Prevent retrying job in case schema is different
- Optimise Kafka health checks

- Do not throw error when CLI version is not provided during GRPC call
- Fix missing import in recommendation API on Python CLI
- Fix searching feature sets based on nested names
- Fix bug when operator crushes when online messaging properties are missing
- Fix bug with missing `featureClassifiers` field
- Better error reporting when data in time travel column is in invalid format during ingest

## Version 0.3.0 (12-05-2022)

### New features

- Replace the capability of creating new version during ingest by explicit api, please see migration guide
- Add possibility to remotely debug Feature Store application
- Add project id and feature set id as spark job pod labels
- Introduce feature recommender
- Compute stddev and mean incrementally
- Expose TTL on register feature set GRPC api

### Fixes

- Improve health checks for Feature Store services
- Fix error where auth pages leads to 404 error
- Fix online feature store to work with both root and separate buckets
- Correctly fail in case feature set contains features with same name (case insensitive)
- Improve online feature store idempotency
- Correctly fail in case array is being passed as primary/secondary or time travel column
- Fix unsupported BinaryType error
- Do not put user secrets to Spark config map
- Fix incremental stats assignment in the database
- Create more user friendly error in case user is not logged in Scala and Python CLI
- Queue ingest job in case there are more jobs submitted at the same time and process them one by one on the backend

Please see Migration guide for more information on breaking changes introduced in this version.

## Version 0.2.0 (21-04-2022)

### New features

- Introduce incremental statistics computation for specific feature statistics
- Provide timing information about specific parts of jobs on job API
- Store child job ids on job itself in CLIs and GRPC API
- Publish events from Spark operator to Kubernetes, making them visible using `kubectl describe`
- Introduce time to live configuration for entries in jobs collection
- Use JSON format for logging across all Feature Store components
- Significantly lower the size of the operator image by removing spark distribution from it
- Expose description on the schema API
- Introduce validation which prevents modification of time travel column once feature set has been created

- Feature type can now be specified on the schema API during registering feature set or creating a new feature set version.
- Expose metrics endpoint
- Add time to live to Spark application and remove the need for spark jobs cron job
- Spark operator is now resilient towards restarts

**Fixes**

- Fix intermittent Mongo errors by updating Mongo client library to latest version
- Disable retry for Out Of Memory errors
- Use asynchronous call in job persister
- Fix client retry in Scala client
- Fix progress reporting in Scala client
- Fix wrong bucket name error when using root bucket on AWS deployments
- Fix bug where preview only works after downloading data

Please see Migration guide for more information on breaking changes introduced in this version.

## Version 0.1.3 (08-04-2022)

**Fixes**

- Calling update request subsequently fails when we reach version x.10

## Version 0.1.2 (31-03-2022)

**New features**

- Send notifications about various major events in feature store to notifications topic
- Native support for nested data types on Schema API
- Expose special data information on feature level and automatically propagate to feature set level
- Support for creating a new feature set version by changing a special data information on feature
- Expose auto project owner configuration
- Expose online and custom data fields on GRPC api
- Java GRPC api is now downloadable from feature store documentation

**Fixes**

- Avoid duplicate unique count computation in statistics job
- Run all job output handlers in transaction to avoid bad database state in case core restarts during job handling
- Handle case where spark driver pod is killed by K8 before the container within pod is initialized
- Prevent running multiple ingest and revert jobs on the same feature set major version
- Ensure Feature Store Core can be restarted at any stage without introducing a bad state
- Fix time to live migration on historical feature sets
- Avoid multiple notifications from online to core about data ready to be ingested
- Fix Online2Offline to work with Redis cluster deployment
- Fix statistics computation
- Fix project delete by stabilizing core during restarts + by introducing migration to remove stale jobs
- Propagate error to client in case job does not exist

- Fix cases that could lead to writing feature set to historical if it already existed
- Ensure jobs on folder resources can work when root folder ends with slash
- Fix various rare database bad states during handling revert and ingest jobs
- Fix problem during fetching user id in online ingestion
- Change stats computation to true by default
- GRPC retry now correctly works on Python client

Please see Migration guide for more information on breaking changes introduced in this version.

## Version 0.1.1 (17-03-2022)

**New features**

- Properly refresh properties on project and feature set after updating on CLI
- Expose option for specifying min and max number of Spark executors. For more information refer to deployment section of the documentation.
- Expose configuration which enables/disables notifications logging. For more information refer to deployment section of the documentation.
- Introduce Offline to Online component in online feature store, including automatic sync of offline and online stores.
- More robust project and feature set update api. See the `Breaking Changes` section bellow.

**Fixes**

- Fixed time to live migration to enum, it was not executed in version 0.1.0
- Mark job as pending after it has been created
- Refresh functionality now correctly loads only latest minor version for current major feature set version
- Fix validation of online ingestion -> accept only valid json strings
- Fix and test retry mechanism. Intermittent problems within jobs are now being correctly retried.
- Remove incoming request from notification message as it can contain secure information
- Store confidential data in Kubernetes secrets instead of as in regular configuration on custom resource
- Fix regression bug causing authentication failure when using Azure service principal
- Provide proper error message if job does not exist when using job api

Please see Migration guide for more information on breaking changes introduced in this version.

## Version 0.1.0 (10-03-2022)

**New features**

- Improve Spark operator to use K8s informers instead of regular polling of resources
- Add owner reference to spark driver pod to its parent custom resource
- Implement Online Feature Store Ingestion and Retrieve
- Implement Online 2 offline feature service
- Integrate Online Feature Store with deployment templates
- Introduce automatic notifications for each observer request from API
- Add possibility to read AWS credentials from ~/.aws/credentials
- Authentication callback endpoint now properly propagates errors

**Fixes**

- Fixed problem when new Version GRPC API is switching to default values of properties such as marked for masking
- Validate spaces in feature names during registration
- Remove groups and roles from user collection as feature store is not using those
- Fix permission problem when project editor is not getting access to feature sets
- Preserve capitals in the project and feature set names
- Handle failed status from operator when the driver pod gets terminated abruptly
- Fix problem which could cause job with long input to fail

## Version 0.0.39 (17-02-2022)

**New features**

- Deployment Helm charts are available for download from Feature Store documentation

**Fixes**

- Support Mongo 4.2 (Create collections during core startup)
- Fix preview functionality when running on specific ingest
- Fix None.get error in job output handler
- Fix problem with duplicated data ingestion when time travel column is explicitly provided
- Fix retry functionality - store only result of lastly retried job
- Fix spark frame retrieval of specific ingest
- Fix wrong ingest id column name in Scala client

## Version 0.0.38 (10-02-2022)

**New features**

- Introduce Spark Operator -> ensures Spark Jobs subsystem is scalable and asynchronous
- Use enum for process interval field on grpc feature set registration API
- Expose custom data on grpc feature set registration api
- Support for scheduling spark executors and drivers based on matching taints
- Expose configuration to change Spark log level
- Ensure we have only the most permissive policy on the policies collection
- Introduce historical collection for policies
- Document Sparkling Water & Feature Store integration
- Support for masking primitive and nested types ( struct and arrays) and any nested and combination level
- Introduce logging in the spark jobs

**Fixes**

- Fix bug caused by adding permission to an user which does not exist
- Statistics computation is now correctly started when triggered by asynchronous job
- Fix missing tls messaging documentation
- Ensure the error message from spark job can always fit into the grpc header
- Avoid reading full container for meta-data in case of using folder resource

- Ingest job now generates warning in case there is a schema difference only in type/s
- Use mounted secrets in spark jobs instead of transferring those in plain text
- Handle job outputs asynchronously
- Project consumer now does not add feature set consumer permission.
- Ensure `from featurestore import -` imports all data sources
- Ensure ingest history gives correct results
- Ensure that unlocked project still requires feature set consumer permissions to retrieve from feature sets
- Fix partitionBy migration
- Fix cache migrations
- Remove extra timestamp column when retrieving data as Spark
- Ensure large access token (up to 16Kb) can be consumed by feature store
- User default partitioning when user does not specify partition by argument in register feature set API.

## Version 0.0.37 (19-01-2022)

**New features**

- Offline Feature Store helm charts are up-to-date with latest S3 & Gen2 changes
- Support for explicitly specifying credentials during schema extraction and ingest
- Improved login functionality for CLI
- Introduce support for partitioning
- Introduce support for reverting any ingest. This change also migrates revert functionality to be based on ingest ids. This also means that the reverted data are actually getting deleted now
- Use Kafka for communication between Spark job and core. Preparing the ground for Spark operator
- Expose marked for masking on feature level in CLI
- Remove ingest number from ingest history as we use ingest ids now

**Fixes**

- Introduce migration to remove temporary collections created during migrations
- Fix problem with credentials for retrieving and writing spark data frames on S3 and Gen2
- Fix incorrect behaviour in folder ingest capability in case feature set did not have time travel column defined
- Fix bug when registering a feature set on project currently being deleted
- Fix ingest using spark frame when cache is configured to use single root bucket
- Gen2 & S3 support as feature store cache
- Fix problem during delete - file not found
- Asynchronous ingest job now correctly starts statistics computation job
- Fix bug where we treated DecimalType as categorical instead numerical during statistics computation

# Migration guide

## From 2.0.0 to 2.1.0

### Feature View API removal

As of version 2.1.0, the Feature View API is no longer available.

## From 1.2.0 to 2.0.0

Release 2.x has some breaking API changes and is not backwards compatible with 1.x releases.

### Scala client removal

Only Python client is supported from now on.

### Permission Model Improvement

`Secret` and `locked` flags has been removed from project in version 2.0.0 and were replaced a single `access modifier` flag. The `access modifier` flag has the following values:

- **Public** - means that every user can see this project and feature sets within the project. This option represents situation where project was neither `locked` or `secret`
- **ProjectOnly** - means that every user can see the project, but only users with viewer permission can see feature sets within this project. This value replaces `locked` flag.
- **Private** - means that only owner and users the owner gave permission to can access this project and feature sets within. This value replaces `secret` flag. Previously `secret` meant that only owners can see and access the project. Newly, the `private` flag means that owner or the users the owner gave permission to have access to the project. Hence, this way the system correctly respects the permission system. Therefore, when a project or feature set is created as secret, only owner can see it. If the owner decides to grant permission to someone else, that someone will also have the provided access to the project or feature set. By default, projects are created as private.

`Secret` flag has been removed from feature set in version 2.0.0. If users do not want to create publicly accessible feature set, they should create feature sets in a private project. Migration script from 1.2 to 2.0 automatically marks project as private in case the feature set was secret to avoid accidental exposure.

All feature set drafts for secret feature sets are removed during the migration. If you have any drafts like this, and you don't want to lose those, please create feature sets from those drafts before you start the migration process.

### Changes

- In CLI, methods `select`, `exclude`, `append`, `prepend` on Schema are now immutable.
- API method `GetIngestWriteCredentials` has been removed. `GenerateTemporaryUpload` should be used instead
- Internal Raw Data source `TempParquetFile` has been removed as it is no longer needed.
- Ingestion from local sources (SparkDataSource among them) is disabled by default. If ingestion from local files and SparkDataSource should be enabled, new helm value `global.storage.commonConfig.temporaryStorageUploadEnabled` should be set to true
- `secret` argument in register feature set method has been removed in all clients.
- `secret` field has been removed on feature set entity in all clients.
- `secret` field has been removed from the following Proto messages: `FeatureSet`, `RegisterFeatureSetRequest`, `ListableFeatureSet`, `UpdateFeatureSetRequest`, `FeatureSetDraftRequest`, `DraftToFeatureSetRequest`, `FeatureSetDraft`
- Enum value `FEATURE_SET_SECRET` has been removed from the Proto message `UpdatableFeatureSetField`
- Helm property `core.config.isProjectLockedByDefault` is removed without replacement. When project is being created, user can select the appropriate access modifier. Grpc API `GetProjectsDefault` has been removed as well without replacement. Projects are created as `private` by default.

### From 1.1.2 to 1.2.0

- The Helm parameter `telemetry.window.timeDuration` has been changed from integer representing minutes, to ISO 8601-1 duration format
- The helm parameter `global.storage.supporting.authMethod` has been moved to `global.storage.supporting.s3.authMethod` or `global.storage.supporting.datalakegen2.authMethod` (depends on selected backend)
- The helm parameter `global.storage.offline.authMethod` has been moved to `global.storage.supporting.offline.s3.authMethod` or `global.storage.offline.datalakegen2.authMethod` or `global.storage.offline.snowflake.authMethod` (depends on selected backend)
- The helm parameter `global.storage.offline.privateKey` has been moved to `global.storage.offline.snowflake.privateKey`
- The helm parameter `global.storage.offline.passphrase` has been moved to `global.storage.offline.snowflake.passphrase`
- The helm parameter `global.storage.offline.deltalakegen2.authMethod` has been moved to `global.storage.offline.datalakegen2.authMethod`
- The helm parameter `global.storage.offline.deltalakegen2.ami.tenantId` has been moved to `global.storage.offline.datalakegen2.ami.tenantIdd`
- The helm parameter `global.storage.offline.deltalakegen2.ami.endpoint` has been moved to `global.storage.offline.datalakegen2.ami.endpoint`
- The helm parameter `global.storage.offline.deltalakegen2.ami.clientId` has been moved to `global.storage.offline.datalakegen2.ami.clientId`
- The helm parameter `global.storage.offline.deltalakegen2.ami.accountName` has been moved to `global.storage.offline.datalakegen2.ami.accountName`
- The helm parameter `global.storage.offline.deltalakegen2.ami.aadpodidbindingValue` has been moved to `global.storage.offline.datalakegen2.ami.aadpodidbindingValue`
- The helm parameter `global.storage.supporting.deltalakegen2.authMethod` has been moved to `global.storage.supporting.datalakegen2.authMethod`
- The helm parameter `global.storage.supporting.deltalakegen2.ami.tenantId` has been moved to `global.storage.supporting.datalakegen2.ami.tenantId`
- The helm parameter `global.storage.supporting.deltalakegen2.ami.endpoint` has been moved to `global.storage.supporting.datalakegen2.ami.endpoint`
- The helm parameter `global.storage.supporting.deltalakegen2.ami.clientId` has been moved to `global.storage.supporting.datalakegen2.ami.clientId`
- The helm parameter `global.storage.supporting.deltalakegen2.ami.accountName` has been moved to `global.storage.supporting.datalakegen2.ami.accountName`
- The helm parameter `global.storage.supporting.deltalakegen2.ami.aadpodidbindingValue` has been moved to `global.storage.supporting.datalakegen2.ami.aadpodidbindingValue`

### From 1.0.0 to 1.1.0

All jobs must be finished before upgrading to Feature Store 1.1 and higher from previous versions.

The values for the option `global.storage.offline.backend` has been changed to `s3`, `datalakegen2` and `snowflake`

- The following GRPC APIs has been changed:
  - `RetrieveMLDatasetAsLinks`, please use `StartRetrieveMLDatasetAsLinksJob` and `GetRetrieveMLDatasetAsLinksJobOutput` after job finished to get links

### From 0.19.3 to 1.0.0

Before upgrading to newer versions after 1.0.0, Feature Store must be first upgraded to 1.0.0.

- Version 1.0.0 is no longer backwards compatible with older clients due to removal of several deprecated APIs. Releases within 1.x.y will again be backwards compatible from CLI's perspective.
- Bearer token prefix is now required in Authorization header.
- Starting with 1.0.0, secure connection is used by default in both Scala and Python CLI
- Spark no longer requires cluster role but only a role. Before upgrading to 1.0.0, please remove the `{helm-release-name}-spark-binding`. This step is required to avoid error as helm can not update cluster-role to a role.

- Starting from 1.0.0 configuration for file storage (supporting storage) is separated from offline storage. The following Helm parameters were added:
  - `global.storage.offline.username`
  - `global.storage.offline.password`
  - `global.storage.offline.authMethod`
  - `global.storage.offline.backend`
  - `global.storage.offline.s3.endpoint`
  - `global.storage.offline.s3.region`
  - `global.storage.offline.s3.sessionRoleArn`
  - `global.storage.offline.s3.kmsEnabled`
  - `global.storage.offline.containers.root`
  - `global.storage.offline.containers.data`
- The following Helm parameters were renamed:
  - `global.config.dataBucket` to `global.storage.offline.containers.data`
  - `global.config.retrieveBucket` to `global.storage.supporting.containers.retrieve`
  - `core.config.commonStorageConfiguration.tempCredentialsExpiration` to `global.storage.commonConfig.tempCredentialsExpiration`
  - `core.config.offlineStorage.storageIdConversionEnabled` to `global.storage.offline.storageIdConversionEna`
  - `global.storage.username` to `global.storage.supporting.username`
  - `global.storage.password` to `global.storage.supporting.password`
  - `global.storage.authMethod` to `global.storage.supporting.authMethod`
  - `global.config.storageBackend` to `global.storage.supporting.backend`
  - `global.config.s3` to `global.storage.supporting.s3`
  - `global.config.s3.endpoint` to `global.storage.supporting.s3.endpoint`
  - `global.config.s3.region` to `global.storage.supporting.s3.region`
  - `global.config.s3.sessionRoleArn` to `global.storage.supporting.s3.sessionRoleArn`
  - `global.config.s3.kmsEnabled` to `global.storage.supporting.s3.kmsEnabled`
  - `global.config.rootBucket` to `global.storage.supporting.containers.root`
  - `global.config.artifactsBucket` to `global.storage.supporting.containers.artifacts`
  - `global.config.tempBucket` to `global.storage.supporting.containers.temp`
  - `global.config.previewBucket` to `global.storage.supporting.containers.preview`
  - `global.config.onlineStoreDataBucket` to `global.storage.supporting.containers.online`
- To use the same storage for files and offline data, please use the same values for the new parameters as those used for storage. For example:
  - `global.storage.supporting.username: <STORAGE_USERNAME>` for offline `global.storage.offline.username: <STORAGE_USERNAME>`
  - `global.storage.supporting.password: <STORAGE_PASSWORD>` for offline `global.storage.offline.password: <STORAGE_PASSWORD>`
  - `global.storage.supporting.backend: s3` for offline `global.storage.offline.backend: offline_storage_s3`
  - `global.storage.supporting.s3.endpoint: <STORAGE_ENDPOINT>` for offline `global.storage.offline.s3.endpoint: <STORAGE_ENDPOINT>`
  - `global.storage.supporting.containers.root: <STORAGE_ROOT_BUCKET>` for offline `global.storage.offline.containers.root: <STORAGE_ROOT_BUCKET>`
  - `global.storage.supporting.containers.root: <STORAGE_ROOT_BUCKET>` for offline `global.storage.offline.containers.root: <STORAGE_ROOT_BUCKET>`
  - `global.storage.supporting.containers.preview` to use previous generated files for preview `global.storage.offline.containers.data: <STORAGE_DATA_BUCKET>`
- Helm argument `global.online.postgres.connectionString` has been renamed to `global.online.postgres.dsn`. This parameter expects PostgreSQL JDBC connection string
- Helm arguments `global.online.postgres.username` and `global.online.postgres.password` have been removed. Username and password (if applicable) must be passed to `global.online.postgres.dsn`. Please inspect PostgreSQL connection string format for more details.
- Helm argument `core.config.idpMetadataUri` has been renamed to `global.config.idpMetadataUri`

- Helm argument `global.cloud.discovery.restApiPublicUri` has been renamed to `global.cloud.discovery.restApiCorePublicUri`
- Helm argument `global.cloud.apiUrl` has been removed. The value should be put to existing argument `global.cloud.discovery.grpcApiPublicUri`
- The deprecated `feature_set` field has been removed in the following GRPC messages:
  - `GetIngestHistoryRequest`
  - `StartRevertIngestJobRequest`
  - `ListJobsRequest`
  - `GetRecommendationRequest`
  - `FeatureSetPermissionRequest`
  - `ListFeatureSetsVersionRequest`
  - `DeleteFeatureSetRequest`
  - `StartIngestJobRequest`
  - `IngestResponse`
  - `RetrieveRequest`
  - `StartMaterializationOnlineRequest`
  - `CreateNewFeatureSetVersionRequest`
- The deprecated `project` field has been removed in the following GRPC messages:
  - `ProjectPermissionRequest`
  - `DeleteProjectRequest`
  - `GetFeatureSetRequest`
- The following GRPC APIs has been removed:
  - `GetProjectOwners`, please use `GetUserProjectPermissions` instead
  - `GetFeatureSetOwners`, please use `GetUserFeatureSetPermissions` instead
  - `ListTokens`, please use `ListPersonalAccessTokens` instead
  - `OnlineIngest`, please use POST call `/online/api/v1/ingestion/feature-sets/{featureSetId}/{featureSetMajorVersion}` instead
  - `OnlineRetrieve`, please use GET call `/online/api/v1/retrieve/{featureSetId}/{featureSetMajorVersion}` instead
  - `RetrieveMLDatasetOnline`, please use GET call `/online/api/v1/retrieve/ml-datasets/{mlDataSetId}` instead
- The `owners` getter on project and feature set CLI entities has been removed in Python CLI. Please user `list_owners` instead.
- The `owners` getter on project and feature set CLI entities has been removed in Scala CLI. Please user `listOwners` instead.

### From 0.19.1 to 0.19.2

- Helm argument `core.config.databaseName` has been removed without replacement. Database must be included in the PostgreSQL JDBC connection string
- Helm argument `core.config.dbConnectionString` has been renamed to `core.database.dsn`. This parameter expects PostgreSQL JDBC connection string
- Helm arguments `core.database.username` and `core.database.password` have been removed. Username and password (if applicable) must be passed to `core.database.dsn`. Please inspect PostgreSQL connection string format for more details.
- The following Helm parameters were removed:
  - `core.config.spark.userNameAttribute` - from now on there is no possibility to select which attribute will be set to label
- The following k8 labels on spark jobs are renamed:
  - `job-id` to `featurestore.h2o.ai/job-id`
  - `job-name` to `featurestore.h2o.ai/job-name`

- `project` to `cloud.h2o.ai/workspace`
- `feature-set` to `featurestore.h2o.ai/feature-set-id` and `featurestore.h2o.ai/feature-set-version`
- `feature-view` to `featurestore.h2o.ai/feature-view-id` and `featurestore.h2o.ai/feature-view-version`
- `mldataset` to `featurestore.h2o.ai/mldataset-id`
- `user-name` to `cloud.h2o.ai/creator`

### From 0.18.0 to 0.19.0

- Starting from 0.19.0 feature name cannot contain ‘
- Starting from 0.19.0 feature in partition_by cannot be nested or have complex type (struct, array)
- Starting from 0.19.0 api `GetUserByMail` is deleted
- In Helm, extra Spark options in property `sparkoperator.config.spark.extraOptions` should be passed as array elements instead as single value

### From 0.16.0 to 0.17.0

- Starting from 0.17.0 methods feature_sets.register, feature_set.flow use enum FeatureSetFlow instead of string
- To enable the `pg_trgm` extension, which is required by the Azure platform, you can follow the steps outlined in the Azure extensions documentation

### From 0.15.0 to 0.16.0

- Starting from 0.16.0 Azure Gen2 Dependencies jar doesn't contain the transitive dependencies. Please refer to Spark dependencies to see which dependencies must be present on your local Spark cluster to support retrieval of data using Spark frames.
- The following Helm parameters were renamed:
  - `global.cache.username` to `global.storage.username`
  - `global.cache.password` to `global.storage.password`
  - `global.config.cacheBackend` to `global.config.storageBackend`

### From 0.14.0 to 0.15.0

- Kafka related Helm properties `global.config.messaging.kafka.topicsConfig.[topic-name].retentionMs`, `global.config.messaging.kafka.topicsConfig.[topic-name].retentionMinutes` and `global.config.messaging.kafka.topicsConfig.[topic-name].retentionHours` are replaced by single `global.config.messaging.kafka.topicsConfig.[topic-name].retentionPolicy`. Policy is specified by duration format defined in ISO 8601-1 standard.
- Added new fields `feature_set_id` and `feature_set_version` and marked `feature_set` as deprecated in proto message `IngestResponse`. `feature_set` field will be deleted in next major version 1.0.0.
- Added new field `project_id` and marked `project` field as deprecated proto messages: `ProjectPermissionRequest`, `DeleteProjectRequest`, `GetFeatureSetRequest`. `project` field will be deleted in next major version 1.0.0.
- Added new fields `feature_set_id` and marked `feature_set` as deprecated in proto messages: `ListJobsRequest`, `GetRecommendationRequest`, `FeatureSetPermissionRequest`, `ListFeatureSetsVersionRequest`, `DeleteFeatureSetRequest`. `feature_set` field will be deleted in next major version 1.0.0.
- Added new fields `feature_set_id` and `feature_set_version` and marked `feature_set` as deprecated in proto messages: `GetIngestHistoryRequest`, `StartRevertIngestJobRequest`, `StartIngestJobRequest`, `RetrieveRequest`, `StartMaterializationOnlineRequest`, `CreateNewFeatureSetVersionRequest`. `feature_set` field will be deleted in release 1.0.0
- GRPC method `ListTokens` has been deprecated and replaced by `ListPersonalAccessTokens` which uses pagination. The former method will be removed in release 1.0.0
- In Scala and Python client, `client.auth.pats.list()` now returns iterator instead of list.

## From 0.13.0 to 0.14.0

- Deprecated behaviour starting preview job has been removed.
- Bearer token prefix is now required in Authorization header.
- All deprecated arguments in release 0.12.0 are now removed.
- All deprecated updated API methods are removed.
- Deprecated owner field has been removed.
- On GRPC level, FeatureSetHeader has been replaced by `featureSetId` and `featureSetVersion` fields in the following messages: `OnlineRetrieveRequest`, `OnlineIngestRequest` and `GetFeatureSetsLastMinorForCurrentMajorRequest`.
- Helm properties `disable-api.deletion` and `disable-api.role-assigment` have been removed. New Helm property `prohibited.cli.methods` has been introduced. This property allows the admin to specify list of methods to be disabled from CLI, such as:`ai.h2o.featurestore.api.v1.CoreService/DeleteFeatureSet,ai.h2o.featurestore.api.v1.CoreService/DeleteProject`.
- GRPC method `listFeature` has been renamed to `listFeatures`.
- Event method `GetFeatureSetLastMinor` has been renamed to `GetFeatureSet`.

## From 0.12.0 to 0.12.1

In Scala CLI, the arguments `tags`, `filterBuilder` and `jsonQuery` in `featureSets.list` and argument `filterBuilder` in `projects.listFeatureSets` method are deprecated and will be removed in 0.14.0. If you need to filter the listed feature sets, please use Scala filtering capabilities on received `FeatureSet` iterator.

In Python CLI, the arguments `tags`, `filters` in `feature_sets.list` and argument `filters` in `projects.list_feature_sets` method are deprecated and will be removed in 0.14.0. If you need to filter the listed feature sets, please use Python filtering capabilities on received `FeatureSet` iterator (such as list comprehensions).

On GRPC API, the argument `query` in `ListFeatureSetsPageRequest` is deprecated and will be removed in 0.14.0. If you need to filter the feature sets, please filter those on the received end of your application.

## From 0.11.0 to 0.12.0

From version 0.12.0, it is recommended to add prefix "Bearer" to Authorization header. Handling Authorization header without that prefix will be removed in 0.14.0

Java GRPC API methods were previously generated into a single class. With version 0.12.0 the API is split into multiple classes. If you are using Java GRPC api, you will need to update the imports on your application.

## From 0.10.0 to 0.11.0

Deprecated GRPC methods:

- `ListFeatureSets` and `ListFeatureSetsAcrossProjects` have been removed. Please use `ListFeatureSetsPage` instead.
- `ListProjects` has been removed. Please use `ListProjectsPage` instead.
- `UpdateFeatureSetPrimaryKey` will be removed in 0.14.0 without replacement. Changing the primary key is now only possible during the creation of a new feature set version.
- The following will be removed in 0.14.0, so please use `UpdateProject` instead:
  - `UpdateProjectCustomData`
  - `UpdateProjectDescription`
  - `UpdateProjectSecret`
  - `UpdateProjectLocked`
- The following will be removed in 0.14.0, so please use `UpdateFeatureSet` instead:
  - `UpdateFeatureSetTags`
  - `UpdateFeatureSetDataSourceDomains`

  - `UpdateFeatureSetDescription`
  - `UpdateFeatureSetType`
  - `UpdateFeatureSetApplicationName`
  - `UpdateFeatureSetApplicationId`
  - `UpdateFeatureSetDeprecated`
  - `UpdateFeatureSetProcessInterval`
  - `UpdateFeatureSetProcessIntervalUnit`
  - `UpdateFeatureSetFlow`
  - `UpdateFeatureSetState`
  - `UpdateFeatureSetSecret`
  - `UpdateFeatureSetCustomData`
  - `UpdateTimeToLiveOfflineInterval`
  - `UpdateTimeToLiveOfflineIntervalUnit`
  - `UpdateTimeToLiveOnlineInterval`
  - `UpdateTimeToLiveOnlineIntervalUnit`
  - `UpdateFeatureSetOnlineNamespace`
  - `UpdateFeatureSetOnlineTopic`
  - `UpdateFeatureSetOnlineConnectionType`
  - `UpdateFeatureSetLegalApproved`
  - `UpdateFeatureSetLegalApprovedNotes`
- The following will be removed in 0.14.0, so please use `UpdateFeature` instead:
  - `UpdateFeatureStatus`
  - `UpdateFeatureType`
  - `UpdateFeatureImportance`
  - `UpdateFeatureDescription`
  - `UpdateFeatureSpecial`
  - `UpdateFeatureAnomalyDetection`
  - `UpdateFeatureCustomData`
  - `UpdateFeatureClassifiers`
- `UpdateProjectOwner` will be removed in 0.14.0. Please use `AddProjectPermission` or `RemoveProjectPermission` instead.
- `UpdateFeatureSetOwner` will be removed in 0.14.0. Please use `AddFeatureSetPermission` or `RemoveFeatureSetPermission` instead.

In both the Scala and Python CLI, the setter for the primary key is deprecated and will be removed in 0.14.0. Changing the primary key is now only possible using a new argument exposed on the create new version API call.

Deprecated `classifierName` field has been removed from `CreateRecommendationClassifierRequest` GRPC API.

Deprecated `preview` on the retrieve holder has been removed. Please use `fs.get_preview()` instead.

Deprecated `secondary_key` field has been removed from feature set. All secondary keys are pushed into `primary_key` field.

Deprecated `owner` field will be removed from project and feature set in 0.14.0 on API and also on proto entities. Please use `owners` instead.

Starting with release 0.11.0, the `retrieve` method starts respecting minor versions of feature sets. That means that running `retrieve` on version `1.3` retrieves the data up to version `1.3`. This ensures proper consistency for external tools depending on a specific feature set version. The data are also immutable in case of reverts. Meaning that previously, when you reverted an ingest, the data retrieved for that feature set were different after that retrieve operation.

**Note:** The consistent retrieval works as explained above for all ingestions and reverts called starting with version 0.11.0. Retrieving feature set prior version 0.10.0 leads to the original behaviour.

## From 0.9.0 to 0.10.0

The collection of historical policies has been removed and migrated to a new `permissions` collection. This collection contains information about previous permission updates. If a permission has been replaced by a new higher permission, its state is `PROMOTED`. If the permission has been removed, its state is `REVOKED`.

`ref.preview()` has been deprecated and replaced with the new API command `fs.get_preview()`. This preview is computed and stored during the first ingestion. Until the upcoming Feature Store release of **0.14.0**, the `get_preview()` method will compute the missing preview and store it on the backend. We highly recommend that you run this method on prior existing feature sets before **0.14.0** to make sure that the preview is populated.

## From 0.8.0 to 0.9.0

The optional arguments `start_date_time` and `end_date_time` for the Python CLI have been removed from the `ingest` / `ingest_async` methods as they are no longer needed.

The optional arguments `startDateTime` and `endDateTime` for the Scala CLI have been removed from the `ingest` / `ingestAsync` methods as they are no longer needed.

## From 0.6.0 to 0.8.0

GRPC method `listProjects` is now deprecated. Please switch to the `listProjectsPage` API which uses pagination. While we don't plan to remove the original methods to preserve backwards compatibility, we strongly suggest using the paginated variant.

GRPC methods `ListFeatureSets` and `ListFeatureSetsAcrossProjects` are now deprecated. Please switch to the `ListFeatureSetsPage` API which uses pagination and replaces both of the former methods. While we don't plan to remove the original methods to preserve backwards compatibility, we strongly suggest using the paginated variant.

The list methods in Python and Scala for projects and feature sets now return iterators instead of full collections starting from version 0.7.0.

`partitionPattern` is now deprecated and has been removed on folder data sources.

## From 0.5.0 to 0.6.0

All Proto and GRPC classed have been moved from package `ai.h2o.featurestore.core` to package `ai.h2o.featurestore.api.v1`. Please update your code using our GRPC API by updating your imports.

The GenerateToken RPC call now accepts a Proto timestamp as an expiration date instead of string representation.

## From 0.4.0 to 0.5.0

The environment variables required to pass AWS credentials have been changed to a more generic name to support AWS S3 and S3 compatible sources like Minio, Google Cloud, etc.

Previously, you needed to set the following environment variables to read data from AWS:

```
export AWS_ACCESS_KEY=my aws key
export AWS_SECRET_KEY=my secret
export AWS_REGION=my region
export AWS_ROLE_ARN=my role
```

Now, to achieve the same, you set the following variables:

```
export S3_ACCESS_KEY=my aws key
export S3_SECRET_KEY=my secret
export S3_REGION=my region
export S3_ROLE_ARN=my role
```

We have also renamed the AWS credentials pass on the clients from `AWSCredentials` to `S3Credentials`.

### Derived feature sets

In 0.5.0, we introduced derived feature sets. Derived data sources (e.g., SparkPipeline, DriverlessAIMOJO, JoinFeatureSets) are now deprecated and will be removed 0.6.0. As such, if you want to ingest new data to your feature sets that are using those derived data sources, you must migrate to derived feature sets instead.

To migrate to a derived feature set, a new version needs to be created using that derived schema with the selected transformation. Once this new version is created, ingestion is automatically triggered. This action will write all data from

the parent feature set(s) with the applied transformation. The following example shows how to do this using the Python client:

```
import featurestore.transformations as t

spark_pipeline_transformation = t.SparkPipeline("...")
spark_pipeline_schema = client.extract_derived_schema([parent_feature_set],
spark_pipeline_transformation)
derived_feature_set = feature_set_to_be_derived.create_new_version(schema=spark_pipeline_schema)
```

**Note:** To allow automatic ingestion on the derived feature set that uses DriverlessAIMOJO, the new parameter `sparkoperator.driverlessAiLicenseKey` needs to be added to the Helm values. It should contain your license to Driverless AI (which is kept in k8 secrets).

## From 0.2.0 to 0.3.0

Prior to version 0.3.0, the partition pattern accepted `date{}` syntax in the folder's data sources. This has been removed as it is now obsolete due to several optimizations of the internal code.

Please update all your existing partition patterns and update the `date{..}` by `.*`.

### Feature set ingest API changes

#### Python

Previously, when ingesting data from data sources that periodically change using the Python CLI, you would use the following API call:

```
fs.ingest(ingest_source, new_version_on_schema_change=True)
```

Now, to achieve the same, you use the following API:

```
new_schema = client.extract_from_source(ingest_source)
if not fs.schema.is_compatible_with(new_schema, compare_data_types=False):
patched_schema = fs.schema.patch_from(new_schema, compare_data_types=False)
new_feature_set = fs.create_new_version(schema=patched_schema,
reason="schema changed before ingest")
new_feature_set.ingest(ingest_source)
else:
fs.ingest(ingest_source)
```

#### Scala

Previously, when ingesting data from data sources that periodically change using the Scala CLI, you would use the following API call:

```
fs.ingest(ingestSource, newVersionOnSchemaChange=true)
```

Now, to achieve the same, you use the following API:

```
val newSchema = client.extractSchemaFromSource(ingestSource)
if (!fs.schema().isCompatibleWith(newSchema, compareDataTypes=false) {
val patchedSchema = fs.schema().patchFrom(newSchema, compareDataTypes=false)
val newFeatureSet = fs.createNewVersion(schema=patchedSchema,
reason="schema changed before ingest")
newFeatureSet.ingest(ingestSource)
} else {
fs.ingest(ingestSource)
}
```

#### GRPC

Previously, when ingesting data from data sources that periodically change using the GRPC API, you would use the following API call:

```
val startIngestJobRequest = StartIngestJobRequest(featureSet = Some(featureSet),
newVersionOnSchemaChange=true)
blockingStub.startIngestJob(startIngestJobRequest)
```

Now, to achieve the same, you use the following API:

```
val request = FeatureSetSchemaCompatibilityRequest(featureSet = Some(featureSet),
newSchema = newSchema, compareDataTypes = false)
val response = blockingStub.isFeatureSetSchemaCompatible(request)
if (!response.isCompatible) {
val schemaPatchRequest = FeatureSetSchemaPatchRequest(featureSet = Some(featureSet),
newSchema = newSchema, compareDataTypes = false)
val schemaPatchResponse = blockingStub.patchFeatureSetSchema(schemaPatchRequest)
val patchedSchema = schemaPatchResponse.schema
val createNewVersionRequest = CreateNewFeatureSetVersionRequest(featureSet = Some(featureSet),
schema = patchedSchema, reason = "")
val createNewVersionResponse = blockingStub.createNewFeatureSetVersion(createNewVersionRequest)
val newFeatureSet = createNewVersionResponse.getFeatureSet
val startIngestJobRequest = StartIngestJobRequest(featureSet = Some(newFeatureSet), ...)
blockingStub.startIngestJob(startIngestJobRequest)
}
```

#### Feature set schema API changes

#### Python

Previously, when loading a schema from a feature set using the Python CLI, you would use the following API call:

```
schema = feature_set.get_schema()
```

Now, to achieve the same, you use the following API:

```
schema = feature_set.schema.get()
```

#### Scala

Previously, when loading a schema from a feature set using the Scala CLI, you would use the following API call:

```
val schema = feature_set.getSchema()
```

Now, to achieve the same, you use the following API:

```
val schema = feature_set.schema().get()
```

### From 0.1.3 to 0.2.0

Prior to version 0.2.0, the feature type was determined as part of the statistics computation. Now, in version version 0.2.0, you can specify the feature type using the schema API. The feature type can be specified explicitly or can be left empty (i.e. the backend will automatically discover it).

We have removed the `Undefined` feature type because each feature now is correctly assigned its feature type after being registered or creating a new version. We have also introduced the `Composite` feature type; it is used for features containing nested features.

We have stopped the backend from automatically marking specific textual features with the `Categorical` feature type since the logic behind it was not solid. Now, if you want to mark the feature type as `Categorical`, please specify that during registration explicitly using the schema API.

For more information, please see the Schema API.

### From 0.1.1 to 0.1.2

The Custom Resource Definition (CRD) has been changed.

The Python CLI method `from_string` has been renamed to `create_from` in the schema.

The Scala CLI argument `maskedFeatures` from the register feature set call has been removed. Please use the schema API to describe which features should be masked. For example:

**Python**

```
schema["my_feature_name"].special_data.pci = True
project.register_feature_set(schema, "feature_set_name")
```

**Scala**

```
schema("my_feature_name").specialSata.pci = true
project.registerFeatureSet(schema, "feature_set_name")
```

The feature set type and feature type on the GRPC API has been migrated from strings to enums. This allows for better validation.

A new parameter, `jobsCredentialsKey`, was added to Helm values. Please make sure to provide this. Supported sizes for this variable are 16, 24, and 32 (in bytes)

```
core:
  ...

  core:
    salt: Yy7c8pzqSJXw6LHbUnhQ1234
    jobsCredentialsKey: Yy7c8pzqSJXw6LHbUnhQ1234
```

## From 0.1.0 to 0.1.1

### Version change setter for feature removed

The Python CLI setter `version_change` and the Scala CLI setter `versionChange` on the feature has been removed. This setter was initially exposed by accident. It is not possible to update the version change directly. It is updated automatically on the backend.

### Update metadata method removed on project and feature set

The Python CLI method `update_metadata` and the Scala CLI setter `update_metadata` have been removed from both the project and feature set. To update the metadata simply call the setter:

Previously, this was the call to update the feature set description:

**Python**

```
feature_set.description = "new_description"
feature_set.update_metadata()
```

**Scala**

```
featureSet.description = "new_description"
featureSet.updateMetadata()
```

Now, this is the call to achieve the same and to update the metadata on the client and backend:

**Python**

```
feature_set.description = "new_description"
```

**Scala**

```
featureSet.description = "new_description"
```

### GRPC project API changes

We have removed the `UpdateProjectMetadata` call for the GRPC API and exposed specific API calls for each field which can be modified on the project.

Previously, to update the project description and locked using the Scala API:

```
project.description = "new_description"
project.locked = true
val request = UpdateProjectMetadataRequest(project = Some(project))
blockingStub.updateProjectMetadata(request)
```

Now, to achieve the same, you use the following code for each field you need to update:

```
blockingStub.updateProjectDescription(ProjectStringFieldUpdateRequest(project.id,
"new_description"))
blockingStub.updateProjectLocked(ProjectBooleanFieldUpdateRequest(project.id,  true))
```

Previously, it was possible to accidentally modify fields which were not exposed for modification because the API transferred the full project object, but that is no longer possible with the new API.

### GRPC feature set API changes

We have removed the `UpdateFeatureSetMetadata` call for the GRPC API and exposed specific API calls for each field which can be modified on the project.

Previously, to update a feature set description and feature status using the Scala API, you would use the following API:

```
featureSet.description = "new_description"
featureSet.features.find(_.name == "feature_name").get.status = "new_status"
val request = UpdateFeatureSetMetadataRequest(featureSet = Some(featureSet))
blockingStub.updateFeatureSetMetadata(request)
```

Now, to achieve the same, you use the following code for each field you need to update:

```
val featureSetHeader = FeatureSetHeader(projectId, internalFeatureSetId,
internalFeatureSetVersion)
val descriptionUpdateRequest = FeatureSetStringFieldUpdateRequest(Some(featureSetHeader),
"new_feature_set_description")
blockingStub.updateFeatureSetDescription(descriptionUpdateRequest)

val statusUpdateRequest = FeatureStringFieldUpdateRequest(Some(featureSetHeader),
featureName, "new_status")
blockingStub.updateFeatureStatus(statusUpdateRequest)
```

Previously, it was possible to accidentally modify fields which were not exposed for modification because the API transferred the full feature set object, but that is no longer possible with the new API.