
Feast Documentation

Feast Authors

Jun 07, 2021

CONTENTS

1 Feature Store	1
2 Config	7
3 Data Source	9
4 Entity	11
5 Feature View	13
6 Feature	15
Python Module Index	17
Index	19

FEATURE STORE

```
class feast.feature_store.FeatureStore(repo_path: Optional[str] = None, config:
                                        Optional[feast.repo_config.RepoConfig] = None)
```

Bases: `object`

A FeatureStore object is used to define, create, and retrieve features.

Parameters

- **repo_path** – Path to a `feature_store.yaml` used to configure the feature store
- **config** (`RepoConfig`) – Configuration object used to configure the feature store

```
apply(objects: Union[feast.entity.Entity, feast.feature_view.FeatureView,
                    List[Union[feast.feature_view.FeatureView, feast.entity.Entity]]])
```

Register objects to metadata store and update related infrastructure.

The apply method registers one or more definitions (e.g., Entity, FeatureView) and registers or updates these objects in the Feast registry. Once the registry has been updated, the apply method will update related infrastructure (e.g., create tables in an online store) in order to reflect these new definitions. All operations are idempotent, meaning they can safely be rerun.

Parameters objects (`List[Union[FeatureView, Entity]]`) – A list of FeatureView or Entity objects that should be registered

Examples

Register a single Entity and FeatureView.

```
>>> from feast.feature_store import FeatureStore
>>> from feast import Entity, FeatureView, Feature, ValueType, FileSource
>>> from datetime import timedelta
>>>
>>> fs = FeatureStore()
>>> customer_entity = Entity(name="customer", value_type=ValueType.INT64,
↳ description="customer entity")
>>> customer_feature_view = FeatureView(
>>>     name="customer_fv",
>>>     entities=["customer"],
>>>     features=[Feature(name="age", dtype=ValueType.INT64)],
>>>     input=FileSource(path="file.parquet", event_timestamp_column="timestamp
↳"),
>>>     ttl=timedelta(days=1)
>>> )
>>> fs.apply([customer_entity, customer_feature_view])
```

config: `feast.repo_config.RepoConfig`

delete_feature_view(*name: str*)

Deletes a feature view or raises an exception if not found.

Parameters **name** – Name of feature view

get_entity(*name: str*) → `feast.entity.Entity`

Retrieves an entity.

Parameters **name** – Name of entity

Returns Returns either the specified entity, or raises an exception if none is found

get_feature_view(*name: str*) → `feast.feature_view.FeatureView`

Retrieves a feature view.

Parameters **name** – Name of feature view

Returns Returns either the specified feature view, or raises an exception if none is found

get_historical_features(*entity_df: Union[pandas.core.frame.DataFrame, str]*, *feature_refs: List[str]*)

→ `feast.infra.offline_stores.offline_store.RetrievalJob`

Enrich an entity dataframe with historical feature values for either training or batch scoring.

This method joins historical feature data from one or more feature views to an entity dataframe by using a time travel join.

Each feature view is joined to the entity dataframe using all entities configured for the respective feature view. All configured entities must be available in the entity dataframe. Therefore, the entity dataframe must contain all entities found in all feature views, but the individual feature views can have different entities.

Time travel is based on the configured TTL for each feature view. A shorter TTL will limit the amount of scanning that will be done in order to find feature data for a specific entity key. Setting a short TTL may result in null values being returned.

Parameters

- **entity_df** (*Union[pd.DataFrame, str]*) – An entity dataframe is a collection of rows containing all entity columns (e.g., `customer_id`, `driver_id`) on which features need to be joined, as well as a `event_timestamp` column used to ensure point-in-time correctness. Either a Pandas DataFrame can be provided or a string SQL query. The query must be of a format supported by the configured offline store (e.g., BigQuery)
- **feature_refs** – A list of features that should be retrieved from the offline store. Feature references are of the format “`feature_view:feature`”, e.g., “`customer_fv:daily_transactions`”.

Returns RetrievalJob which can be used to materialize the results.

Examples

Retrieve historical features using a BigQuery SQL entity dataframe

```
>>> from feast.feature_store import FeatureStore
>>>
>>> fs = FeatureStore(config=RepoConfig(provider="gcp"))
>>> retrieval_job = fs.get_historical_features(
>>>     entity_df="SELECT event_timestamp, order_id, customer_id from gcp_
↳project.my_ds.customer_orders",
>>>     feature_refs=["customer:age", "customer:avg_orders_1d", "customer:avg_
↳orders_7d"]
```

(continues on next page)

(continued from previous page)

```

>>> )
>>> feature_data = retrieval_job.to_df()
>>> model.fit(feature_data) # insert your modeling framework here.

```

get_online_features(*feature_refs: List[str], entity_rows: List[Dict[str, Any]]*) → *feast.online_response.OnlineResponse*

Retrieves the latest online feature data.

Note: This method will download the full feature registry the first time it is run. If you are using a remote registry like GCS or S3 then that may take a few seconds. The registry remains cached up to a TTL duration (which can be set to infinity). If the cached registry is stale (more time than the TTL has passed), then a new registry will be downloaded synchronously by this method. This download may introduce latency to online feature retrieval. In order to avoid synchronous downloads, please call `refresh_registry()` prior to the TTL being reached. Remember it is possible to set the cache TTL to infinity (cache forever).

Parameters

- **feature_refs** – List of feature references that will be returned for each entity. Each feature reference should have the following format: “feature_table:feature” where “feature_table” & “feature” refer to the feature and feature table names respectively. Only the feature name is required.
- **entity_rows** – A list of dictionaries where each key-value is an entity-name, entity-value pair.

Returns *OnlineResponse* containing the feature data in records.

Examples

```

>>> from feast import FeatureStore
>>>
>>> store = FeatureStore(repo_path="...")
>>> feature_refs = ["sales:daily_transactions"]
>>> entity_rows = [{"customer_id": 0}, {"customer_id": 1}]
>>>
>>> online_response = store.get_online_features(
>>>     feature_refs, entity_rows)
>>> online_response_dict = online_response.to_dict()
>>> print(online_response_dict)
{'sales:daily_transactions': [1.1, 1.2], 'sales:customer_id': [0, 1]}

```

list_entities(*allow_cache: bool = False*) → *List[feast.entity.Entity]*

Retrieve a list of entities from the registry

Parameters *allow_cache (bool)* – Whether to allow returning entities from a cached registry

Returns List of entities

list_feature_views() → *List[feast.feature_view.FeatureView]*

Retrieve a list of feature views from the registry

Returns List of feature views

materialize(*start_date: datetime.datetime, end_date: datetime.datetime, feature_views: Optional[List[str]] = None*) → *None*

Materialize data from the offline store into the online store.

This method loads feature data in the specified interval from either the specified feature views, or all feature views if none are specified, into the online store where it is available for online serving.

Parameters

- **start_date** (*datetime*) – Start date for time range of data to materialize into the online store
- **end_date** (*datetime*) – End date for time range of data to materialize into the online store
- **feature_views** (*List[str]*) – Optional list of feature view names. If selected, will only run materialization for the specified feature views.

Examples

Materialize all features into the online store over the interval from 3 hours ago to 10 minutes ago.

```
>>> from datetime import datetime, timedelta
>>> from feast.feature_store import FeatureStore
>>>
>>> fs = FeatureStore(config=RepoConfig(provider="gcp"))
>>> fs.materialize(
>>>     start_date=datetime.utcnow() - timedelta(hours=3), end_date=datetime.
↳utcnow() - timedelta(minutes=10)
>>> )
```

materialize_incremental (*end_date: datetime.datetime, feature_views: Optional[List[str]] = None*) → *None*

Materialize incremental new data from the offline store into the online store.

This method loads incremental new feature data up to the specified end time from either the specified feature views, or all feature views if none are specified, into the online store where it is available for online serving. The start time of the interval materialized is either the most recent end time of a prior materialization or (now - ttl) if no such prior materialization exists.

Parameters

- **end_date** (*datetime*) – End date for time range of data to materialize into the online store
- **feature_views** (*List[str]*) – Optional list of feature view names. If selected, will only run materialization for the specified feature views.

Examples

Materialize all features into the online store up to 5 minutes ago.

```
>>> from datetime import datetime, timedelta
>>> from feast.feature_store import FeatureStore
>>>
>>> fs = FeatureStore(config=RepoConfig(provider="gcp", registry="gs://my-fs/",
↳project="my_fs_proj"))
>>> fs.materialize_incremental(end_date=datetime.utcnow() -
↳timedelta(minutes=5))
```

property project: str

refresh_registry()

Fetches and caches a copy of the feature registry in memory.

Explicitly calling this method allows for direct control of the state of the registry cache. Every time this method is called the complete registry state will be retrieved from the remote registry store backend (e.g., GCS, S3), and the cache timer will be reset. If `refresh_registry()` is run before `get_online_features()` is called, then `get_online_feature()` will use the cached registry instead of retrieving (and caching) the registry itself.

Additionally, the TTL for the registry cache can be set to infinity (by setting it to 0), which means that `refresh_registry()` will become the only way to update the cached registry. If the TTL is set to a value greater than 0, then once the cache becomes stale (more time than the TTL has passed), a new cache will be downloaded synchronously, which may increase latencies if the triggering method is `get_online_features()`

repo_path: `pathlib.Path`

version() → `str`

Returns the version of the current Feast SDK/CLI

CONFIG

```
class feast.repo_config.BigQueryOfflineStoreConfig(*, type: typing_extensions.Literal[bigquery] =  
                                                    'bigquery', dataset: pydantic.types.StrictStr =  
                                                    'feast')
```

Offline store config for GCP BigQuery

```
dataset: Optional[pydantic.types.StrictStr]  
          (optional) BigQuery Dataset name for temporary tables
```

```
type: typing_extensions.Literal[bigquery]  
       Offline store type selector
```

```
class feast.repo_config.DatastoreOnlineStoreConfig(*, type: typing_extensions.Literal[datastore] =  
                                                    'datastore', project_id: pydantic.types.StrictStr =  
                                                    None, namespace: pydantic.types.StrictStr =  
                                                    None, write_concurrency:  
                                                    pydantic.types.PositiveInt = 40, write_batch_size:  
                                                    pydantic.types.PositiveInt = 50)
```

Online store config for GCP Datastore

```
namespace: Optional[pydantic.types.StrictStr]  
            (optional) Datastore namespace
```

```
project_id: Optional[pydantic.types.StrictStr]  
             (optional) GCP Project Id
```

```
type: typing_extensions.Literal[datastore]  
       Online store type selector
```

```
write_batch_size: Optional[pydantic.types.PositiveInt]  
                   (optional) Amount of feature rows per batch being written into Datastore
```

```
write_concurrency: Optional[pydantic.types.PositiveInt]  
                    (optional) Amount of threads to use when writing batches of feature rows into Datastore
```

```
exception feast.repo_config.FeastConfigError(error_message, config_path)
```

```
class feast.repo_config.FileOfflineStoreConfig(*, type: typing_extensions.Literal[file] = 'file')  
Offline store config for local (file-based) store
```

```
type: typing_extensions.Literal[file]  
       Offline store type selector
```

```
class feast.repo_config.RegistryConfig(*, path: pydantic.types.StrictStr, cache_ttl_seconds:  
                                       pydantic.types.StrictInt = 600)
```

Metadata Store Configuration. Configuration that relates to reading from and writing to the Feast registry.

cache_ttl_seconds: `pydantic.types.StrictInt`

The cache TTL is the amount of time registry state will be cached in memory. If this TTL is exceeded then the registry will be refreshed when any feature store method asks for access to registry state. The TTL can be set to infinity by setting TTL to 0 seconds, which means the cache will only be loaded once and will never expire. Users can manually refresh the cache by calling `feature_store.refresh_registry()`

Type `int`

path: `pydantic.types.StrictStr`

Path to metadata store. Can be a local path, or remote object storage path, e.g. a GCS URI

Type `str`

```
class feast.repo_config.RepoConfig(*, registry: Union[pydantic.types.StrictStr,
    feast.repo_config.RegistryConfig] = 'data/registry.db', project:
    pydantic.types.StrictStr, provider: pydantic.types.StrictStr,
    online_store: Union[feast.repo_config.DatastoreOnlineStoreConfig,
    feast.repo_config.SqliteOnlineStoreConfig] =
    SqliteOnlineStoreConfig(type='sqlite', path='data/online.db'),
    offline_store: Union[feast.repo_config.FileOfflineStoreConfig,
    feast.repo_config.BigQueryOfflineStoreConfig] =
    FileOfflineStoreConfig(type='file'))
```

Repo config. Typically loaded from `feature_store.yaml`

offline_store: `Union[feast.repo_config.FileOfflineStoreConfig, feast.repo_config.BigQueryOfflineStoreConfig]`

Offline store configuration (optional depending on provider)

Type `OfflineStoreConfig`

online_store: `Union[feast.repo_config.DatastoreOnlineStoreConfig, feast.repo_config.SqliteOnlineStoreConfig]`

Online store configuration (optional depending on provider)

Type `OnlineStoreConfig`

project: `pydantic.types.StrictStr`

Feast project id. This can be any alphanumeric string up to 16 characters. You can have multiple independent feature repositories deployed to the same cloud provider account, as long as they have different project ids.

Type `str`

provider: `pydantic.types.StrictStr`

local or gcp

Type `str`

registry: `Union[pydantic.types.StrictStr, feast.repo_config.RegistryConfig]`

Path to metadata store. Can be a local path, or remote object storage path, e.g. a GCS URI

Type `str`

```
class feast.repo_config.SqliteOnlineStoreConfig(*, type: typing_extensions.Literal[sqlite] = 'sqlite',
    path: pydantic.types.StrictStr = 'data/online.db')
```

Online store config for local (SQLite-based) store

path: `pydantic.types.StrictStr`

(optional) Path to sqlite db

type: `typing_extensions.Literal[sqlite]`

Online store type selector

DATA SOURCE

```
class feast.data_source.BigQueryOptions(table_ref: Optional[str], query: Optional[str])
    DataSource BigQuery options used to source features from BigQuery query

classmethod from_proto(bigquery_options_proto: feast.core.DataSource_pb2.BigQueryOptions)
    Creates a BigQueryOptions from a protobuf representation of a BigQuery option

    Parameters bigquery_options_proto – A protobuf representation of a DataSource

    Returns Returns a BigQueryOptions object based on the bigquery_options protobuf

property query
    Returns the BigQuery SQL query referenced by this source

property table_ref
    Returns the table ref of this BQ table

to_proto() → feast.core.DataSource_pb2.BigQueryOptions
    Converts an BigQueryOptionsProto object to its protobuf representation.

    Returns BigQueryOptionsProto protobuf

class feast.data_source.BigQuerySource(event_timestamp_column: Optional[str] = None, table_ref:
    Optional[str] = None, created_timestamp_column: Optional[str]
    = "", field_mapping: Optional[Dict[str, str]] = None,
    date_partition_column: Optional[str] = "", query: Optional[str] =
    None)

property bigquery_options
    Returns the bigquery options of this data source

get_table_query_string() → str
    Returns a string that can directly be used to reference this table in SQL

to_proto() → feast.core.DataSource_pb2.DataSource
    Converts an DataSourceProto object to its protobuf representation.

class feast.data_source.DataSource(event_timestamp_column: str, created_timestamp_column:
    Optional[str] = "", field_mapping: Optional[Dict[str, str]] = None,
    date_partition_column: Optional[str] = "")
    DataSource that can be used source features

property created_timestamp_column
    Returns the created timestamp column of this data source

property date_partition_column
    Returns the date partition column of this data source
```

property event_timestamp_column

Returns the event timestamp column of this data source

property field_mapping

Returns the field mapping of this data source

static from_proto(data_source)

Convert data source config in FeatureTable spec to a DataSource class object.

to_proto() → `feast.core.DataSource_pb2.DataSource`

Converts an DataSourceProto object to its protobuf representation.

class `feast.data_source.FileOptions`(*file_format: Optional[feast.data_format.FileFormat]*, *file_url: Optional[str]*)

DataSource File options used to source features from a file

property file_format

Returns the file format of this file

property file_url

Returns the file url of this file

classmethod from_proto(*file_options_proto: feast.core.DataSource_pb2.FileOptions*)

Creates a FileOptions from a protobuf representation of a file option

Parameters `file_options_proto` – a protobuf representation of a datasource

Returns Returns a FileOptions object based on the file_options protobuf

to_proto() → `feast.core.DataSource_pb2.FileOptions`

Converts an FileOptionsProto object to its protobuf representation.

Returns FileOptionsProto protobuf

class `feast.data_source.FileSource`(*event_timestamp_column: Optional[str] = None*, *file_url: Optional[str] = None*, *path: Optional[str] = None*, *file_format: Optional[feast.data_format.FileFormat] = None*, *created_timestamp_column: Optional[str] = ""*, *field_mapping: Optional[Dict[str, str]] = None*, *date_partition_column: Optional[str] = ""*)

property file_options

Returns the file options of this data source

property path

Returns the file path of this feature data source

to_proto() → `feast.core.DataSource_pb2.DataSource`

Converts an DataSourceProto object to its protobuf representation.

class `feast.data_source.SourceType`(*value*)

DataSource value type. Used to define source types in DataSource.

ENTITY

```
class feast.entity.Entity(name: str, value_type: feast.value_type.ValueType = <ValueType.UNKNOWN: 0>,
                           description: str = "", join_key: Optional[str] = None, labels:
                           Optional[MutableMapping[str, str]] = None)
```

Represents a collection of entities and associated metadata.

property created_timestamp

Returns the created_timestamp of this entity

property description

Returns the description of this entity

classmethod from_dict(entity_dict)

Creates an entity from a dict

Parameters **entity_dict** – A dict representation of an entity

Returns Returns a EntityV2 object based on the entity dict

classmethod from_proto(entity_proto: feast.core.Entity_pb2.Entity)

Creates an entity from a protobuf representation of an entity

Parameters **entity_proto** – A protobuf representation of an entity

Returns Returns a EntityV2 object based on the entity protobuf

classmethod from_yaml(yml: str)

Creates an entity from a YAML string body or a file path

Parameters **yml** – Either a file path containing a yaml file or a YAML string

Returns Returns a EntityV2 object based on the YAML file

is_valid()

Validates the state of a entity locally. Raises an exception if entity is invalid.

property join_key

Returns the join key of this entity

property labels

Returns the labels of this entity. This is the user defined metadata defined as a dictionary.

property last_updated_timestamp

Returns the last_updated_timestamp of this entity

property name

Returns the name of this entity

to_dict() → Dict

Converts entity to dict

Returns Dictionary object representation of entity

to_proto() → `feast.core.Entity_pb2.Entity`

Converts an entity object to its protobuf representation

Returns `EntityV2Proto` protobuf

to_spec_proto() → `feast.core.Entity_pb2.EntitySpecV2`

Converts an `EntityV2` object to its protobuf representation. Used when passing `EntitySpecV2` object to Feast request.

Returns `EntitySpecV2` protobuf

to_yaml()

Converts an entity to a YAML string.

Returns Entity string returned in YAML format

property value_type: `feast.value_type.ValueType`

Returns the type of this entity

FEATURE VIEW

```
class feast.feature_view.FeatureView(name: str, entities: List[str], ttl:
    Optional[Union[google.protobuf.duration_pb2.Duration,
    datetime.timedelta]], input:
    Union[feast.data_source.BigQuerySource,
    feast.data_source.FileSource], features: List[feast.feature.Feature] =
    [], tags: Optional[Dict[str, str]] = None, online: bool = True)
```

A FeatureView defines a logical grouping of serveable features.

```
classmethod from_proto(feature_view_proto: feast.core.FeatureView_pb2.FeatureView)
```

Creates a feature view from a protobuf representation of a feature view

Parameters **feature_view_proto** – A protobuf representation of a feature view

Returns Returns a FeatureViewProto object based on the feature view protobuf

```
is_valid()
```

Validates the state of a feature view locally. Raises an exception if feature view is invalid.

```
to_proto() → feast.core.FeatureView_pb2.FeatureView
```

Converts an feature view object to its protobuf representation.

Returns FeatureViewProto protobuf

FEATURE

```
class feast.feature.Feature(name: str, dtype: feast.value_type.ValueType, labels:
                           Optional[MutableMapping[str, str]] = None)
```

Feature field type

```
property dtype: feast.value_type.ValueType
```

Getter for data type of this field

```
classmethod from_proto(feature_proto: feast.core.Feature_pb2.FeatureSpecV2)
```

Parameters `feature_proto` – FeatureSpecV2 protobuf object

Returns Feature object

```
property labels: MutableMapping[str, str]
```

Getter for labels of this field

```
property name
```

Getter for name of this field

```
to_proto() → feast.core.Feature_pb2.FeatureSpecV2
```

Converts Feature object to its Protocol Buffer representation

```
class feast.feature.FeatureRef(name: str, feature_table: str)
```

Feature Reference represents a reference to a specific feature.

```
classmethod from_proto(proto: feast.serving.ServingService_pb2.FeatureReferenceV2)
```

Construct a feature reference from the given FeatureReference proto

Parameters `proto` – Protobuf FeatureReference to construct from

Returns FeatureRef that refers to the given feature

```
classmethod from_str(feature_ref_str: str)
```

Parse the given string feature reference into FeatureRef model String feature reference should be in the format `feature_table:feature`. Where “feature_table” and “name” are the feature_table name and feature name respectively.

Parameters `feature_ref_str` – String representation of the feature reference

Returns FeatureRef that refers to the given feature

```
to_proto() → feast.serving.ServingService_pb2.FeatureReferenceV2
```

Convert and return this feature table reference to protobuf.

Returns Protobuf representation of this feature table reference.

PYTHON MODULE INDEX

f

`feast.data_source`, 9

`feast.entity`, 11

`feast.feature`, 15

`feast.feature_store`, 1

`feast.feature_view`, 13

`feast.repo_config`, 7

A

`apply()` (*feast.feature_store.FeatureStore* method), 1

B

`bigquery_options` (*feast.data_source.BigQuerySource* property), 9

`BigQueryOfflineStoreConfig` (class in *feast.repo_config*), 7

`BigQueryOptions` (class in *feast.data_source*), 9

`BigQuerySource` (class in *feast.data_source*), 9

C

`cache_ttl_seconds` (*feast.repo_config.RegistryConfig* attribute), 7

`config` (*feast.feature_store.FeatureStore* attribute), 1

`created_timestamp` (*feast.entity.Entity* property), 11

`created_timestamp_column` (*feast.data_source.DataSource* property), 9

D

`dataset` (*feast.repo_config.BigQueryOfflineStoreConfig* attribute), 7

`DataSource` (class in *feast.data_source*), 9

`DatastoreOnlineStoreConfig` (class in *feast.repo_config*), 7

`date_partition_column` (*feast.data_source.DataSource* property), 9

`delete_feature_view()` (*feast.feature_store.FeatureStore* method), 2

`description` (*feast.entity.Entity* property), 11

`dtype` (*feast.feature.Feature* property), 15

E

`Entity` (class in *feast.entity*), 11

`event_timestamp_column` (*feast.data_source.DataSource* property), 9

F

`feast.data_source` module, 9

`feast.entity` module, 11

`feast.feature` module, 15

`feast.feature_store` module, 1

`feast.feature_view` module, 13

`feast.repo_config` module, 7

`FeastConfigError`, 7

`Feature` (class in *feast.feature*), 15

`FeatureRef` (class in *feast.feature*), 15

`FeatureStore` (class in *feast.feature_store*), 1

`FeatureView` (class in *feast.feature_view*), 13

`field_mapping` (*feast.data_source.DataSource* property), 10

`file_format` (*feast.data_source.FileOptions* property), 10

`file_options` (*feast.data_source.FileSource* property), 10

`file_url` (*feast.data_source.FileOptions* property), 10

`FileOfflineStoreConfig` (class in *feast.repo_config*), 7

`FileOptions` (class in *feast.data_source*), 10

`FileSource` (class in *feast.data_source*), 10

`from_dict()` (*feast.entity.Entity* class method), 11

`from_proto()` (*feast.data_source.BigQueryOptions* class method), 9

`from_proto()` (*feast.data_source.DataSource* static method), 10

`from_proto()` (*feast.data_source.FileOptions* class method), 10

`from_proto()` (*feast.entity.Entity* class method), 11

`from_proto()` (*feast.feature.Feature* class method), 15

`from_proto()` (*feast.feature.FeatureRef* class method), 15

`from_proto()` (*feast.feature_view.FeatureView* class method), 13

`from_str()` (*feast.feature.FeatureRef* class method), 15
`from_yaml()` (*feast.entity.Entity* class method), 11

G

`get_entity()` (*feast.feature_store.FeatureStore* method), 2
`get_feature_view()` (*feast.feature_store.FeatureStore* method), 2
`get_historical_features()` (*feast.feature_store.FeatureStore* method), 2
`get_online_features()` (*feast.feature_store.FeatureStore* method), 3
`get_table_query_string()` (*feast.data_source.BigQuerySource* method), 9

I

`is_valid()` (*feast.entity.Entity* method), 11
`is_valid()` (*feast.feature_view.FeatureView* method), 13

J

`join_key` (*feast.entity.Entity* property), 11

L

`labels` (*feast.entity.Entity* property), 11
`labels` (*feast.feature.Feature* property), 15
`last_updated_timestamp` (*feast.entity.Entity* property), 11
`list_entities()` (*feast.feature_store.FeatureStore* method), 3
`list_feature_views()` (*feast.feature_store.FeatureStore* method), 3

M

`materialize()` (*feast.feature_store.FeatureStore* method), 3
`materialize_incremental()` (*feast.feature_store.FeatureStore* method), 4

module

- `feast.data_source`, 9
- `feast.entity`, 11
- `feast.feature`, 15
- `feast.feature_store`, 1
- `feast.feature_view`, 13
- `feast.repo_config`, 7

N

`name` (*feast.entity.Entity* property), 11
`name` (*feast.feature.Feature* property), 15

`namespace` (*feast.repo_config.DatastoreOnlineStoreConfig* attribute), 7

O

`offline_store` (*feast.repo_config.RepoConfig* attribute), 8
`online_store` (*feast.repo_config.RepoConfig* attribute), 8

P

`path` (*feast.data_source.FileSource* property), 10
`path` (*feast.repo_config.RegistryConfig* attribute), 8
`path` (*feast.repo_config.SQLiteOnlineStoreConfig* attribute), 8
`project` (*feast.feature_store.FeatureStore* property), 4
`project` (*feast.repo_config.RepoConfig* attribute), 8
`project_id` (*feast.repo_config.DatastoreOnlineStoreConfig* attribute), 7
`provider` (*feast.repo_config.RepoConfig* attribute), 8

Q

`query` (*feast.data_source.BigQueryOptions* property), 9

R

`refresh_registry()` (*feast.feature_store.FeatureStore* method), 4
`registry` (*feast.repo_config.RepoConfig* attribute), 8
`RegistryConfig` (class in *feast.repo_config*), 7
`repo_path` (*feast.feature_store.FeatureStore* attribute), 5
`RepoConfig` (class in *feast.repo_config*), 8

S

`SourceType` (class in *feast.data_source*), 10
`SQLiteOnlineStoreConfig` (class in *feast.repo_config*), 8

T

`table_ref` (*feast.data_source.BigQueryOptions* property), 9
`to_dict()` (*feast.entity.Entity* method), 11
`to_proto()` (*feast.data_source.BigQueryOptions* method), 9
`to_proto()` (*feast.data_source.BigQuerySource* method), 9
`to_proto()` (*feast.data_source.DataSource* method), 10
`to_proto()` (*feast.data_source.FileOptions* method), 10
`to_proto()` (*feast.data_source.FileSource* method), 10
`to_proto()` (*feast.entity.Entity* method), 12
`to_proto()` (*feast.feature.Feature* method), 15
`to_proto()` (*feast.feature.FeatureRef* method), 15
`to_proto()` (*feast.feature_view.FeatureView* method), 13
`to_spec_proto()` (*feast.entity.Entity* method), 12

`to_yaml()` (*feast.entity.Entity* method), 12
type (*feast.repo_config.BigQueryOfflineStoreConfig* attribute), 7
type (*feast.repo_config.DatastoreOnlineStoreConfig* attribute), 7
type (*feast.repo_config.FileOfflineStoreConfig* attribute), 7
type (*feast.repo_config.SQLiteOnlineStoreConfig* attribute), 8

V

`value_type` (*feast.entity.Entity* property), 12
`version()` (*feast.feature_store.FeatureStore* method), 5

W

`write_batch_size` (*feast.repo_config.DatastoreOnlineStoreConfig* attribute), 7
`write_concurrency` (*feast.repo_config.DatastoreOnlineStoreConfig* attribute), 7