Feast Documentation

Feast Authors

Aug 05, 2021

CONTENTS

CHAPTER

ONE

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

apply(*objects: Union[feast.entity.Entity, feast.feature_view.FeatureView, feast.feature_service.FeatureService, List[Union[feast.feature_view.FeatureView, feast.entity.Entity, feast.feature_service.FeatureService]]]*, *commit: bool = True*)

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** – A single object, or a list of objects that should be registered with the Feature Store.
- **commit** – whether to commit changes to the registry

Raises **ValueError** – The 'objects' parameter could not be parsed properly.

Examples

Register an Entity and a FeatureView.

```
>>> from feast import FeatureStore, Entity, FeatureView, Feature, ValueType,␣
↪FileSource, RepoConfig
>>> from datetime import timedelta
>>> fs = FeatureStore(config=RepoConfig(registry="feature_repo/data/registry.db
↪", project="feature_repo", provider="local"))
>>> driver = Entity(name="driver_id", value_type=ValueType.INT64, description=
↪"driver id")
>>> driver_hourly_stats = FileSource(
...     path="feature_repo/data/driver_stats.parquet",
...     event_timestamp_column="event_timestamp",
```

(continues on next page)

(continued from previous page)

```
...     created_timestamp_column="created",
... )
>>> driver_hourly_stats_view = FeatureView(
...     name="driver_hourly_stats",
...     entities=["driver_id"],
...     ttl=timedelta(seconds=86400 * 1),
...     batch_source=driver_hourly_stats,
... )
>>> fs.apply([driver_hourly_stats_view, driver]) # register entity and feature␣
↪view
```

config: feast.repo_config.RepoConfig

delete_feature_service(*name: str*)
Deletes a feature service.

Parameters **name** – Name of feature service.

Raises **FeatureServiceNotFoundException** – The feature view could not be found.

delete_feature_view(*name: str*)
Deletes a feature view.

Parameters **name** – Name of feature view.

Raises **FeatureViewNotFoundException** – The feature view could not be found.

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

Parameters **name** – Name of entity.

Returns The specified entity.

Raises **EntityNotFoundException** – The entity could not be found.

get_feature_service(*name: str*) → *feast.feature_service.FeatureService*
Retrieves a feature service.

Parameters **name** – Name of feature service.

Returns The specified feature service.

Raises **FeatureServiceNotFoundException** – The feature service could not be found.

get_feature_view(*name: str*) → feast.feature_view.FeatureView
Retrieves a feature view.

Parameters **name** – Name of feature view.

Returns The specified feature view.

Raises **FeatureViewNotFoundException** – The feature view could not be found.

get_historical_features(*entity_df: Union[pandas.core.frame.DataFrame, str]*, *features: Optional[Union[List[str]*, feast.feature_service.FeatureService]] = *None*, *feature_refs: Optional[List[str]] = None*, *full_feature_names: bool = False*) → 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)
- **features** – A list of features, that should be retrieved from the offline store. Either a list of string feature references can be provided or a FeatureService object. Feature references are of the format "feature_view:feature", e.g., "customer_fv:daily_transactions".
- **full_feature_names** – A boolean that provides the option to add the feature view prefixes to the feature names, changing them from the format "feature" to "feature_view__feature" (e.g., "daily_transactions" changes to "customer_fv__daily_transactions"). By default, this value is set to False.

Returns RetrievalJob which can be used to materialize the results.

Raises `ValueError` – Both or neither of features and feature_refs are specified.

Examples

Retrieve historical features from a local offline store.

```
>>> from feast import FeatureStore, Entity, FeatureView, Feature, ValueType,␣
↪FileSource, RepoConfig
>>> from datetime import timedelta
>>> import pandas as pd
>>> fs = FeatureStore(config=RepoConfig(registry="feature_repo/data/registry.db
↪", project="feature_repo", provider="local"))
>>> # Before retrieving historical features, we must register the appropriate␣
↪entity and featureview.
>>> driver = Entity(name="driver_id", value_type=ValueType.INT64, description=
↪"driver id")
>>> driver_hourly_stats = FileSource(
...     path="feature_repo/data/driver_stats.parquet",
...     event_timestamp_column="event_timestamp",
...     created_timestamp_column="created",
... )
>>> driver_hourly_stats_view = FeatureView(
...     name="driver_hourly_stats",
...     entities=["driver_id"],
...     ttl=timedelta(seconds=86400 * 1),
...     features=[
...         Feature(name="conv_rate", dtype=ValueType.FLOAT),
...         Feature(name="acc_rate", dtype=ValueType.FLOAT),
...         Feature(name="avg_daily_trips", dtype=ValueType.INT64),
```

(continues on next page)

(continued from previous page)

```
...     ],
...     batch_source=driver_hourly_stats,
... )
>>> fs.apply([driver_hourly_stats_view, driver]) # register entity and feature␣
↪view
>>> entity_df = pd.DataFrame.from_dict(
...     {
...         "driver_id": [1001, 1002],
...         "event_timestamp": [
...             datetime(2021, 4, 12, 10, 59, 42),
...             datetime(2021, 4, 12, 8, 12, 10),
...         ],
...     }
... )
>>> retrieval_job = fs.get_historical_features(
...     entity_df=entity_df,
...     features=[
...         "driver_hourly_stats:conv_rate",
...         "driver_hourly_stats:acc_rate",
...         "driver_hourly_stats:avg_daily_trips",
...     ],
... )
>>> feature_data = retrieval_job.to_df()
```

get_online_features(*features: Union[List[str], feast.feature_service.FeatureService]*, *entity_rows: List[Dict[str, Any]]*, *feature_refs: Optional[List[str]] = None*, *full_feature_names: bool = False*) → 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

- **features** – 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.

Raises **Exception** – No entity with the specified name exists.

Examples

Materialize all features into the online store over the interval from 3 hours ago to 10 minutes ago, and then retrieve these online features.

```
>>> from feast import FeatureStore, Entity, FeatureView, Feature, ValueType,␣
↪FileSource, RepoConfig
>>> from datetime import timedelta
>>> import pandas as pd
>>> fs = FeatureStore(config=RepoConfig(registry="feature_repo/data/registry.db
↪", project="feature_repo", provider="local"))
>>> # Before getting online features, we must register the appropriate entity␣
↪and featureview and then materialize the features.
>>> driver = Entity(name="driver_id", value_type=ValueType.INT64, description=
↪"driver id",)
>>> driver_hourly_stats = FileSource(
...     path="feature_repo/data/driver_stats.parquet",
...     event_timestamp_column="event_timestamp",
...     created_timestamp_column="created",
... )
>>> driver_hourly_stats_view = FeatureView(
...     name="driver_hourly_stats",
...     entities=["driver_id"],
...     ttl=timedelta(seconds=86400 * 1),
...     features=[
...         Feature(name="conv_rate", dtype=ValueType.FLOAT),
...         Feature(name="acc_rate", dtype=ValueType.FLOAT),
...         Feature(name="avg_daily_trips", dtype=ValueType.INT64),
...     ],
...     batch_source=driver_hourly_stats,
... )
>>> fs.apply([driver_hourly_stats_view, driver]) # register entity and feature␣
↪view
>>> fs.materialize(
...     start_date=datetime.utcnow() - timedelta(hours=3), end_date=datetime.
↪utcnow() - timedelta(minutes=10)
... )
Materializing...

...
>>> online_response = fs.get_online_features(
...     features=[
...         "driver_hourly_stats:conv_rate",
...         "driver_hourly_stats:acc_rate",
...         "driver_hourly_stats:avg_daily_trips",
...     ],
...     entity_rows=[{"driver_id": 1001}, {"driver_id": 1002}, {"driver_id":␣
↪1003}, {"driver_id": 1004}],
... )
>>> online_response_dict = online_response.to_dict()
```

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

Retrieves the list of entities from the registry.

Parameters **allow_cache** – Whether to allow returning entities from a cached registry.

Returns A list of entities.

list_feature_services() → List[*feast.feature_service.FeatureService*]
Retrieves the list of feature services from the registry.

Returns A list of feature services.

list_feature_views() → List[feast.feature_view.FeatureView]
Retrieves the list of feature views from the registry.

Returns A 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 feast import FeatureStore, Entity, FeatureView, Feature, ValueType,␣
↪FileSource, RepoConfig
>>> from datetime import timedelta
>>> fs = FeatureStore(config=RepoConfig(registry="feature_repo/data/registry.db
↪", project="feature_repo", provider="local"))
>>> # Before materializing, we must register the appropriate entity and␣
↪featureview.
>>> driver = Entity(name="driver_id", value_type=ValueType.INT64, description=
↪"driver id",)
>>> driver_hourly_stats = FileSource(
...     path="feature_repo/data/driver_stats.parquet",
...     event_timestamp_column="event_timestamp",
...     created_timestamp_column="created",
... )
>>> driver_hourly_stats_view = FeatureView(
...     name="driver_hourly_stats",
...     entities=["driver_id"],
...     ttl=timedelta(seconds=86400 * 1),
...     features=[
...         Feature(name="conv_rate", dtype=ValueType.FLOAT),
...         Feature(name="acc_rate", dtype=ValueType.FLOAT),
...         Feature(name="avg_daily_trips", dtype=ValueType.INT64),
...     ],
...     batch_source=driver_hourly_stats,
... )
```

(continues on next page)

(continued from previous page)

```
>>> fs.apply([driver_hourly_stats_view, driver]) # register entity and feature␣
↪view
>>> fs.materialize(
...     start_date=datetime.utcnow() - timedelta(hours=3), end_date=datetime.
↪utcnow() - timedelta(minutes=10)
... )
Materializing...

...
```

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.

Raises **Exception** – A feature view being materialized does not have a TTL set.

Examples

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

```
>>> from feast import FeatureStore, Entity, FeatureView, Feature, ValueType,␣
↪FileSource, RepoConfig
>>> from datetime import timedelta
>>> fs = FeatureStore(config=RepoConfig(registry="feature_repo/data/registry.db
↪", project="feature_repo", provider="local"))
>>> # Before materializing, we must register the appropriate entity and␣
↪featureview.
>>> driver = Entity(name="driver_id", value_type=ValueType.INT64, description=
↪"driver id",)
>>> driver_hourly_stats = FileSource(
...     path="feature_repo/data/driver_stats.parquet",
...     event_timestamp_column="event_timestamp",
...     created_timestamp_column="created",
... )
>>> driver_hourly_stats_view = FeatureView(
...     name="driver_hourly_stats",
...     entities=["driver_id"],
...     ttl=timedelta(seconds=86400 * 1),
...     features=[
...         Feature(name="conv_rate", dtype=ValueType.FLOAT),
...         Feature(name="acc_rate", dtype=ValueType.FLOAT),
...         Feature(name="avg_daily_trips", dtype=ValueType.INT64),
```

(continues on next page)

(continued from previous page)

```
...     ],
...     batch_source=driver_hourly_stats,
... )
>>> fs.apply([driver_hourly_stats_view, driver]) # register entity and feature␣
↪view
>>> fs.materialize_incremental(end_date=datetime.utcnow() -␣
↪timedelta(minutes=5))
Materializing...

...
```

property project: str
Gets the project of this feature store.

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()

property registry: feast.registry.Registry
Gets the registry of this feature store.

repo_path: pathlib.Path

teardown()
Tears down all local and cloud resources for the feature store.

version() → str
Returns the version of the current Feast SDK/CLI.

CHAPTER

TWO

CONFIG

class feast.repo_config.**FeastConfigBaseModel**
Feast Pydantic Configuration Class

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

class feast.repo_config.**RegistryConfig**(*, *path: pydantic.types.StrictStr*, *cache_ttl_seconds: pydantic.types.StrictInt = 600*, ***extra_data: Any*)
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: Any = None*, *offline_store: Any = None*, *repo_path: pathlib.Path = None*, ***data: Any*)
Repo config. Typically loaded from *feature_store.yaml*

offline_store: Any
Offline store configuration (optional depending on provider)

Type OfflineStoreConfig

online_store: Any
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 or aws

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

CHAPTER

THREE

DATA SOURCE

class feast.data_source.**DataSource**(*event_timestamp_column: Optional[str] = None*, *created_timestamp_column: Optional[str] = None*, *field_mapping: Optional[Dict[str, str]] = None*, *date_partition_column: Optional[str] = None*)

DataSource that can be used to source features.

Parameters

- **event_timestamp_column** (*optional*) – Event timestamp column used for point in time joins of feature values.
- **created_timestamp_column** (*optional*) – Timestamp column indicating when the row was created, used for deduplicating rows.
- **field_mapping** (*optional*) – A dictionary mapping of column names in this data source to feature names in a feature table or view. Only used for feature columns, not entity or timestamp columns.
- **date_partition_column** (*optional*) – Timestamp column used for partitioning.

property created_timestamp_column: str

Returns the created timestamp column of this data source.

property date_partition_column: str

Returns the date partition column of this data source.

property event_timestamp_column: str

Returns the event timestamp column of this data source.

property field_mapping: Dict[str, str]

Returns the field mapping of this data source.

abstract static from_proto(*data_source: feast.core.DataSource_pb2.DataSource*) → Any

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

Parameters **data_source** – A protobuf representation of a DataSource.

Returns A DataSource class object.

Raises **ValueError** – The type of DataSource could not be identified.

get_table_column_names_and_types(*config: feast.repo_config.RepoConfig*) → Iterable[Tuple[str, str]]

Returns the list of column names and raw column types.

Parameters **config** – Configuration object used to configure a feature store.

get_table_query_string() → str

Returns a string that can directly be used to reference this table in SQL.

abstract static source_datatype_to_feast_value_type() → Callable[[str], feast.value_type.ValueType]
Returns the callable method that returns Feast type given the raw column type.

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

validate(*config: feast.repo_config.RepoConfig*)
Validates the underlying data source.

Parameters **config** – Configuration object used to configure a feature store.

class feast.data_source.**SourceType**(*value*)
DataSource value type. Used to define source types in DataSource.

CHAPTER

FOUR

ENTITY

class feast.entity.**Entity**(*name: str*, *value_type: feast.value_type.ValueType = ValueType.UNKNOWN*, *description: str = ''*, *join_key: Optional[str] = None*, *labels: Optional[Dict[str, str]] = None*)

Represents a collection of entities and associated metadata.

Parameters

- **name** – Name of the entity.
- **value_type** (*optional*) – The type of the entity, such as string or float.
- **description** (*optional*) – Additional information to describe the entity.
- **join_key** (*optional*) – A property that uniquely identifies different entities within the collection. Used as a key for joining entities with their associated features. If not specified, defaults to the name of the entity.
- **labels** (*optional*) – User-defined metadata in dictionary form.

property created_timestamp: Optional[datetime.datetime]

Gets the created_timestamp of this entity.

property description: str

Gets 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 An 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 An 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 An EntityV2 object based on the YAML file.

is_valid()

Validates the state of this entity locally.

Raises **ValueError** – The entity does not have a name or does not have a type.

property join_key: str
Gets the join key of this entity.

property labels: Dict[str, str]
Gets the labels of this entity.

property last_updated_timestamp: Optional[datetime.datetime]
Gets the last_updated_timestamp of this entity.

property name: str
Gets 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 An 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 An EntitySpecV2 protobuf.

to_yaml()
Converts a entity to a YAML string.

Returns An entity string returned in YAML format.

property value_type: feast.value_type.ValueType
Gets the type of this entity.

CHAPTER

FIVE

FEATURE VIEW

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

A FeatureView defines a logical grouping of serveable features.

Parameters

- **name** – Name of the group of features.
- **entities** – The entities to which this group of features is associated.
- **ttl** – The amount of time this group of features lives. A ttl of 0 indicates that this group of features lives forever. Note that large ttl's or a ttl of 0 can result in extremely computationally intensive queries.
- **input** – The source of data where this group of features is stored.
- **batch_source** (*optional*) – The batch source of data where this group of features is stored.
- **stream_source** (*optional*) – The stream source of data where this group of features is stored.
- **features** (*optional*) – The set of features defined as part of this FeatureView.
- **tags** (*optional*) – A dictionary of key-value pairs used for organizing FeatureViews.

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 A FeatureViewProto object based on the feature view protobuf.

infer_features_from_batch_source(*config: feast.repo_config.RepoConfig*)

Infers the set of features associated to this feature view from the input source.

Parameters **config** – Configuration object used to configure the feature store.

Raises **RegistryInferenceFailure** – The set of features could not be inferred.

is_valid()

Validates the state of this feature view locally.

Raises **ValueError** – The feature view does not have a name or does not have entities.

property most_recent_end_time: Optional[datetime.datetime]
Retrieves the latest time up to which the feature view has been materialized.

Returns The latest time, or None if the feature view has not been materialized.

to_proto() → feast.core.FeatureView_pb2.FeatureView
Converts a feature view object to its protobuf representation.

Returns A FeatureViewProto protobuf.

CHAPTER

SIX

FEATURE

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

A Feature represents a class of serveable feature.

Parameters

- **name** – Name of the feature.
- **dtype** – The type of the feature, such as string or float.
- **labels** (*optional*) – User-defined metadata in dictionary form.

property dtype: feast.value_type.ValueType

Gets the data type of this feature.

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

Parameters **feature_proto** – FeatureSpecV2 protobuf object

Returns Feature object

property labels: Dict[str, str]

Gets the labels of this feature.

property name

Gets the name of this feature.

to_proto() → feast.core.Feature_pb2.FeatureSpecV2

Converts Feature object to its Protocol Buffer representation.

Returns A FeatureSpecProto protobuf.

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 respresentation of this feature table reference.

CHAPTER

SEVEN

FEATURE SERVICE

class feast.feature_service.**FeatureService**(*name: str*, *features: List[Union[feast.feature_table.FeatureTable, feast.feature_view.FeatureView, feast.feature_view_projection.FeatureViewProjection]]*, *tags: Optional[Dict[str, str]] = None*)

A feature service is a logical grouping of features for retrieval (training or serving). The features grouped by a feature service may come from any number of feature views.

Parameters

- **name** – Unique name of the feature service.
- **features** – A list of Features that are grouped as part of this FeatureService. The list may contain Feature Views, Feature Tables, or a subset of either.
- **tags** (*optional*) – A dictionary of key-value pairs used for organizing Feature Services.

static from_proto(*feature_service_proto: feast.core.FeatureService_pb2.FeatureService*)

Converts a FeatureServiceProto to a FeatureService object.

Parameters **feature_service_proto** – A protobuf representation of a FeatureService.

to_proto() → feast.core.FeatureService_pb2.FeatureService

Converts a FeatureService to its protobuf representation.

Returns A FeatureServiceProto protobuf.

PYTHON MODULE INDEX

f

INDEX