firebolt-python-sdk

Firebolt

Jul 07, 2022

CONTENTS

The Firebolt Python SDK enables connecting to Firebolt, managing Firebolt resources and executing queries using a library of Python classes and functions.

CHAPTER

ONE

PREREQUISITES

- Python version 3.7 or later along with the pip package installer. For more information, see the Python web page.
- A Firebolt account and login credentials.

CHAPTER

TWO

INSTALLATION

Use pip to install the Firebolt Python SDK from the command line as shown in the example below:

```
$ pip install firebolt-sdk
```

2.1 Optional features

By default, the Firebolt Python SDK uses the `datetime` module to parse date and datetime values. For large operations involving date and datetime values, the Python SDK can achieve faster results by using the ciso8601 package, however this can cause installation issues in some cases.

To install firebolt-python-sdk with `ciso8601` support, run `pip install "firebolt-sdk[ciso8601]"`.

2.2 Release notes

For information about changes in the latest version of the Firebolt Python SDK, see the release notes

2.3 Contributing

For procedures and requirements for contributing to this SDK, see the contributing page on Github.

2.4 License

The Firebolt DB API is licensed under the Apache License Version 2.0 software license.

Note: This project is under active development.

2.4.1 Walkthroughs and examples

Connecting and running queries

This topic provides a walkthrough and examples for how to use the Firebolt Python SDK to connect to Firebolt resources to run commands and query data.

Setting up a connection

To connect to a Firebolt database to run queries or command, you must provide your account credentials through a connection request.

To get started, follow the steps below:

1. Import modules

The Firebolt Python SDK requires you to import the following modules before making any command or query requests to your Firebolt database.

```
from firebolt.db import connect
from firebolt.client import DEFAULT_API_URL
```

2. Connect to your database and engine

Your account information can be provided as parameters in a `connection()` function.

A connection requires the following parameters:

`username`	The email address associated with your Firebolt user.
`password`	The password used for connecting to Firebolt.
`database`	The name of the database you would like to connect to.
`engine_name` or `engine_url`	The name or URL of the engine to use for SQL queries. If the engine is not specified, your default engine is used.

This information can be provided in multiple ways.

- **Set credentials manually**

  You can manually include your account information in a connection object in your code for any queries you want to request.

  Replace the values in the example code below with your Firebolt account credentials as appropriate.

```
username = "your_username"
password = "your_password"
engine_name = "your_engine"
database_name = "your_database"

connection = connect(
        engine_name=engine_name,
        database=database_name,
        username=username,
        password=password,
)
```

(continues on next page)

(continued from previous page)

```
cursor = connection.cursor()
```

- **Use an .env file**

  Consolidating all of your Firebolt credentials into a `.env` file can help protect sensitive information from exposure. Create an `.env` file with the following key-value pairs, and replace the values with your information.

  ```
  FIREBOLT_USER="your_username"
  FIREBOLT_PASSWORD="your_password"
  FIREBOLT_ENGINE="your_engine"
  FIREBOLT_DB="your_database"
  ```

  Be sure to place this `.env` file into your root directory.

  Your connection script can load these environmental variables from the `.env` file by using the python-dotenv package. Note that the example below imports the `os` and `dotenv` modules in order to load the environmental variables.

  ```
  import os
  from dotenv import load_dotenv

  load_dotenv()

  connection = connect(
          username=os.getenv('FIREBOLT_USER'),
          password=os.getenv('FIREBOLT_PASSWORD'),
          engine_name=os.getenv('FIREBOLT_ENGINE'),
          database=os.getenv('FIREBOLT_DB')
  )

  cursor = connection.cursor()
  ```

3. Execute commands using the cursor

The `cursor` object can be used to send queries and commands to your Firebolt database and engine. See below for examples of functions using the `cursor` object.

Command and query examples

This section includes Python examples of various SQL commands and queries.

Inserting and selecting data

The example below uses `cursor` to create a new table called `test_table`, insert rows into it, and then select the table's contents.

The engine attached to your specified database must be started before executing any queries. For help, see *Starting an engine*.

```
cursor.execute(
        '''CREATE FACT TABLE IF NOT EXISTS test_table (
                id INT,
                name TEXT
                )
                PRIMARY INDEX id;'''
        )

cursor.execute(
        '''INSERT INTO test_table VALUES
                (1, 'hello'),
                (2, 'world'),
                (3, '!');'''
        )

cursor.execute(
                '''SELECT * FROM test_table;'''
        )

cursor.close()
```

Note: For reference documentation on `cursor` functions, see *Db.cursor*

Fetching query results

After running a query, you can fetch the results using a `cursor` object. The examples below use the data queried from `test_table` created in the *Inserting and selecting data*.

```
print(cursor.fetchone())
```

Returns: `[2, 'world']`

```
print(cursor.fetchmany(2))
```

Returns: `[[1, 'hello'], [3, '!']]`

```
print(cursor.fetchall())
```

Returns: `[[2, 'world'], [1, 'hello'], [3, '!']]`

Executing parameterized queries

Parameterized queries (also known as "prepared statements") format a SQL query with placeholders and then pass values into those placeholders when the query is run. This protects against SQL injection attacks and also helps manage dynamic queries that are likely to change, such as filter UIs or access control.

To run a parameterized query, use the `execute()` cursor method. Add placeholders to your statement using question marks `?`, and in the second argument pass a tuple of parameters equal in length to the number of `?` in the statement.

```
cursor.execute(
        '''CREATE FACT TABLE IF NOT EXISTS test_table2 (
                id INT,
                name TEXT,
                date_value DATE
        )
                PRIMARY INDEX id;'''
        )
```

```
cursor.execute(
        "INSERT INTO test_table2 VALUES (?, ?, ?)",
                (1, "apple", "2018-01-01"),
        )

cursor.close()
```

If you need to run the same statement multiple times with different parameter inputs, you can use the `executemany()` cursor method. This allows multiple tuples to be passed as values in the second argument.

```
cursor.executemany(
        "INSERT INTO test_table2 VALUES (?, ?, ?)",
        (
                (2, "banana", "2019-01-01"),
                (3, "carrot", "2020-01-01"),
                (4, "donut", "2021-01-01")
        )
)

cursor.close()
```

Executing multiple-statement queries

Multiple-statement queries allow you to run a series of SQL statements sequentially with just one method call. Statements are separated using a semicolon `;`, similar to making SQL statements in the Firebolt UI.

```
cursor.execute(
        """
                SELECT * FROM test_table WHERE id < 4;
                SELECT * FROM test_table WHERE id > 2;
        """
        )
print("First query: ", cursor.fetchall())
assert cursor.nextset()
print("Second query: ", cursor.fetchall())
assert cursor.nextset() is None

cursor.close()
```

Returns:

```
First query:  [[2, 'banana', datetime.date(2019, 1, 1)], [3, 'carrot', datetime.
↪date(2020, 1, 1)], [1, 'apple', datetime.date(2018, 1, 1)]]
Second query:  [[3, 'carrot', datetime.date(2020, 1, 1)], [4, 'donut', datetime.
↪date(2021, 1, 1)]]
```

Note: Multiple statement queries are not able to use placeholder values for parameterized queries.

Using DATE and DATETIME values

DATE, DATETIME and TIMESTAMP values used in SQL insertion statements must be provided in a specific format; otherwise they could be read incorrectly.

- DATE values should be formatted as **YYYY-MM-DD**
- DATETIME and TIMESTAMP values should be formatted as **YYYY-MM-DD HH:MM:SS.SSSSSS**

The datetime module from the Python standard library contains various classes and methods to format DATE, TIMESTAMP and DATETIME data types.

You can import this module as follows:

```
from datetime import datetime
```

Managing engines and databases

This topic provides a walkthrough and examples for using the Firebolt Python SDK to create and modify Firebolt databases and engines.

Setting up a ResourceManager object

You can perform various functions on Firebolt databases and engines by calling a `ResourceManager` object, which must be configured with its own user credentials through the imported `Settings` class.

To get started, follow the steps below:

1. Import modules

To initialize a `ResourceManager` object, import the modules shown below.

```
from firebolt.service.manager import ResourceManager
from firebolt.common import Settings
```

2. Initialize a Settings object

A Settings object contains the user credentials and other information needed to manage Firebolt databases and engines.

The Settings object uses the following parameters:

`user`	The email address associated with your Firebolt user profile.
`password`	The password used for connecting to Firebolt.
`server`	The API hostname for logging in. Defaults to `api.app.firebolt.io` if not included.
`default_region`	The default region for creating new databases and engines. For more information, see Available AWS Regions.

A `Settings` object can be configured with parameters by multiple methods.

- Add the parameters manually in your command script:

```
settings = Settings(
        user="your_username",
        password="your_password",
        server="api.app.firebolt.io"
        default_region="your_region"
        )
```

- Use a `.env` file located in your root directory containing the following parameters:

```
FIREBOLT_USER="your_username",
FIREBOLT_PASSWORD="your_password",
FIREBOLT_SERVER="api.app.firebolt.io"
FIREBOLT_DEFAULT_REGION="your_region"
```

  In your application file, the `Settings` object can read the values from the `.env` file if it is set to `None` instead of having values, as shown below:

```
settings = None
```

3. Initialize a ResourceManager object

After the `Settings` are configured, create a `ResourceManager` object, which is given the variable name `rm` in the example below.

```
rm = ResourceManager(settings=settings)
```

Note: Subsequent examples on this page use the `rm` object for database and engine functions.

Database function examples

This section includes Python examples of various common functions for creating and managing Firebolt resources.

Listing out databases

List out the names of all databases under your account by using the `get_many` function.

List out all databases and their metadata

This produces an inventory of all databases and their metadata from your account. The Python devtools module used in the example below helps format the metadata to be more readable.

```
from devtools import debug

debug(rm.databases.get_many())
```

Listing out databases by name

This function call lists out the names of your databases, but it can be modified to list out other attributes. This is helpful for tracking down a particular database in your account.

```
all_dbs = rm.databases.get_many()
all_db_names = [d.name for d in all_dbs]
for db in db_names:
        print(db)
```

Note: For a list of all database attributes, see *Model.database*.

Creating a new database

Launch a new database and use it to create a `database` object.

A newly created database uses the default region from your Settings unless you specify a different region as a parameter.

```
database = rm.databases.create(name="database_name", region="us-east-1")
```

Note: For a list of all database parameters, see *Service.database*

Locating a database

Find a specific Firebolt database by using its name or ID. These functions are useful as a starting point to create a `database` object that can be called in other database functions.

In the examples below, replace the values for `database_name` and `database_id` with your database name or ID.

Locating by name

```
database = rm.databases.get_by_name(name="database_name")
```

Locating by ID

```
database = rm.databases.get_by_id(id="database_id")
```

Getting database status

Use the Python devtools module to format metadata from a `database` object. This is a helpful command to run after a database operation to check if its execution was successful.

```
from devtools import debug
debug(database)
```

Dropping a database

Delete a database by calling the `delete` function. The database is deleted along with all of its tables.

```
database.delete()
```

Engine function examples

This section includes Python examples of various common functions for creating and managing Firebolt engines.

Creating an engine

Launch a new Firebolt engine and create an `engine` object. The created engine uses the default region included in your Settings unless you specify a different region as a parameter.

```
engine = rm.engines.create(name="engine_name")
```

Note: For a list of all engine parameters, see *Service.engine*

Listing out engines

List out the names of all engines under your account by using the `get_many` function.

List out all engines and metadata

This produces an inventory of all engines and their metadata from your account. The Python devtools module used in the example below helps format the metadata to be more readable.

```
from devtools import debug

debug(rm.engines.get_many())
```

List out engines by name

This function call lists out the names of your engines, but it can be modified to list out other attributes. This is helpful for tracking down a particular engine in your account.

```
all_engines = rm.engines.get_many()
all_engine_names = [e.name for e in all_engines]
for name in all_engine_names:
        print(name)
```

Note: For a list of all engine attributes, see *Model.engine*

Locating an engine

Find a specific Firebolt engine by using its name or ID. These functions are useful as a starting point to create an `engine` object that can be called in other engine functions.

In the examples below, replace the values for `engine_name` and `engine_id` with your engine name or ID.

Locating by name

```
engine = rm.engines.get_by_name(name="engine_name")
```

Locating by ID

```
engine = rm.engines.get_by_id(name="engine_id")
```

Attaching an engine

Attach an engine to a database. An engine must be attached to a database and started before it can run SQL commands or queries.

```
engine = rm.engines.get_by_name(name="engine_name")
engine.attach_to_database(
        database=rm.databases.get_by_name(name="database_name"))
```

Dropping an engine

Delete an engine by calling the `delete` function. The engine is removed from its attached database and deleted.

```
engine.delete()
```

Starting an engine

Start an engine by calling the `start` function on an `engine` object. An engine must be attached to a database and started before it can run SQL commands or queries.

```
engine.start()
```

Stopping an engine

Stop an engine by calling the `stop` function. When stopped, an engine is not available to run queries and does not accrue additional usage time on your account.

```
engine.stop()
```

Updating an engine

Update an engine to change its specifications, returning an updated version of the engine. The engine must be stopped in order to be updated.

For a list of engine parameters that can be updated, see *update()*

```
engine.update(description = "This is a new description.")
```

Getting engine status

Use the Python devtools module to format metadata from an `engine` object. This is a helpful command to run after an engine operation to check if its execution was successful.

```
from devtools import debug
debug(engine)
```

2.4.2 Reference documentation

Async_db

The async_db package enables connecting to a Firebolt database for asynchronous queries.

Async_db.connection

class firebolt.async_db.connection.**Connection**(*engine_url: str*, *database: str*, *auth: Auth*, *api_endpoint: str = 'api.app.firebolt.io'*, *additional_parameters: Dict[str, Any] = {}*)

Bases: `BaseConnection`

Firebolt asyncronous database connection class. Implements PEP 249.

Parameters

- **engine_url** – Firebolt database engine REST API url
- **database** – Firebolt database name
- **username** – Firebolt account username
- **password** – Firebolt account password
- **api_endpoint** – Optional. Firebolt API endpoint. Used for authentication.
- **connector_versions** – Optional. Tuple of connector name and version or list of tuples of your connector stack. Useful for tracking custom connector usage.

Note: Firebolt currenly doesn't support transactions so commit and rollback methods are not implemented.

async aclose() → None

Close connection and all underlying cursors.

api_endpoint

client_class: type

cursor() → *Cursor*

cursor_class

alias of *Cursor*

database

engine_url

class firebolt.async_db.connection.**OverriddenHttpBackend**

Bases: AutoBackend

This class is a short-term solution for TCP keep-alive issue: https://docs.aws.amazon.com/elasticloadbalancing/latest/network/network-load-balancers.html#connection-idle-timeout Since httpx creates a connection right before executing a request backend has to be overridden in order to set the socket KEEPALIVE and KEEPIDLE settings.

async connect_tcp(*host: str*, *port: int*, *timeout: Optional[float] = None*, *local_address: Optional[str] = None*) → AsyncNetworkStream

firebolt.async_db.connection.**async_connect_factory**(*connection_class: Type*) → Callable

async firebolt.async_db.connection.**connect**(*database: Optional[str] = None*, *username: Optional[str] = None*, *password: Optional[str] = None*, *access_token: Optional[str] = None*, *auth: Optional[Auth] = None*, *engine_name: Optional[str] = None*, *engine_url: Optional[str] = None*, *account_name: Optional[str] = None*, *api_endpoint: str = 'api.app.firebolt.io'*, *use_token_cache: bool = True*, *additional_parameters: Dict[str, Any] = {}*) → *Connection*

Connect to Firebolt database.

Parameters

- **database** (*str*) – Name of the database to connect
- **username** (*Optional[str]*) – User name to use for authentication (Deprecated)
- **password** (*Optional[str]*) – Password to use for authentication (Deprecated)
- **access_token** (*Optional[str]*) – Authentication token to use insead of credentials (Deprecated)
- **auth** (*Auth*) –
- **engine_name** (*Optional[str]*) – The name of the engine to connect to
- **engine_url** (*Optional[str]*) – The engine endpoint to use
- **account_name** (*Optional[str]*) – For customers with multiple accounts; if None uses default.

- **api_endpoint** (*str*) – Firebolt API endpoint. Used for authentication.
- **use_token_cache** (*bool*) – Cached authentication token in filesystem. Default: True
- **additional_parameters** (*Optional[Dict]*) – Dictionary of less widely-used arguments for connection.

Note: Providing both *engine_name* and *engine_url* would result in an error.

Async_db.cursor

class firebolt.async_db.cursor.**Cursor**(**args: Any*, ***kwargs: Any*)

Bases: BaseCursor

Class, responsible for executing asyncio queries to Firebolt Database. Should not be created directly, use *connection.cursor*

Parameters

- **description** – information about a single result row
- **rowcount** – the number of rows produced by last query
- **closed** – True if connection is closed, False otherwise
- **arraysize** – Read/Write, specifies the number of rows to fetch at a time with the *fetchmany()* method

connection

async execute(*query: str*, *parameters: Optional[Sequence[Union[int, float, str, datetime, date, bool, Decimal, Sequence]]] = None*, *set_parameters: Optional[Dict] = None*, *skip_parsing: bool = False*) → int

Prepare and execute a database query.

Supported features:

Parameterized queries: placeholder characters ('?') are substituted
with values provided in *parameters*. Values are formatted to be properly recognized by database and to exclude SQL injection.

Multi-statement queries: multiple statements, provided in a single query
and separated by semicolon are executed separatelly and sequentially. To switch to next statement result, *nextset* method should be used.

SET statements: to provide additional query execution parameters, execute
SET param=value statement before it. All parameters are stored in cursor object until it's closed. They can also be removed with *flush_parameters* method call.

Parameters

- **query** (*str*) – SQL query to execute
- **parameters** (*Optional[Sequence[ParameterType]]*) – A sequence of substitution parameters. Used to replace '?' placeholders inside a query with actual values
- **set_parameters** (*Optional[Dict]*) – List of set parameters to execute a query with. DEPRECATED: Use SET SQL statements instead

- **skip_parsing** (*bool*) – Flag to disable query parsing. This will disable parameterized, multi-statement and SET queries, while improving performance

Returns
Query row count

Return type
int

async executemany(*query: str*, *parameters_seq: Sequence[Sequence[Union[int, float, str, datetime, date, bool, Decimal, Sequence]]]*) → int

Prepare and execute a database query.

Supports providing multiple substitution parameter sets, executing them as multiple statements sequentially.

Supported features:

Parameterized queries: placeholder characters ('?') are substituted
with values provided in *parameters*. Values are formatted to be properly recognized by database and to exclude SQL injection.

Multi-statement queries: multiple statements, provided in a single query
and separated by semicolon are executed separatelly and sequentially. To switch to next statement result, *nextset* method should be used.

SET statements: to provide additional query execution parameters, execute
SET param=value statement before it. All parameters are stored in cursor object until it's closed. They can also be removed with *flush_parameters* method call.

Parameters

- **query** (*str*) – SQL query to execute
- **parameters_seq** (*Sequence[Sequence[ParameterType]]*) – A sequence of substitution parameter sets. Used to replace '?' placeholders inside a query with actual values from each set in a sequence. Resulting queries for each subset are executed sequentially.

Returns
Query row count

Return type
int

async fetchall() → List[List[Optional[Union[int, float, str, datetime, date, bool, list, Decimal]]]]

Fetch all remaining rows of a query result.

async fetchmany(*size: Optional[int] = None*) → List[List[Optional[Union[int, float, str, datetime, date, bool, list, Decimal]]]]

Fetch the next set of rows of a query result, cursor.arraysize is default size.

async fetchone() → Optional[List[Optional[Union[int, float, str, datetime, date, bool, list, Decimal]]]]

Fetch the next row of a query result set.

async nextset() → Optional[bool]

Skip to the next available set, discarding any remaining rows from the current set. Returns True if operation was successful, None if there are no more sets to retrive

class firebolt.async_db.cursor.**CursorState**(*value*)

Bases: Enum

An enumeration.

CLOSED = 4

DONE = 3

ERROR = 2

NONE = 1

class firebolt.async_db.cursor.**Statistics**(*, *elapsed: float*, *rows_read: int*, *bytes_read: int*, *time_before_execution: float*, *time_to_execute: float*, *scanned_bytes_cache: float = None*, *scanned_bytes_storage: float = None*)

Bases: BaseModel

Class for query execution statistics

bytes_read: int

elapsed: float

rows_read: int

scanned_bytes_cache: Optional[float]

scanned_bytes_storage: Optional[float]

time_before_execution: float

time_to_execute: float

firebolt.async_db.cursor.**check_not_closed**(*func: Callable*) → Callable

(Decorator) ensure cursor is not closed before calling method.

firebolt.async_db.cursor.**check_query_executed**(*func: Callable*) → Callable

Async_db.util

async firebolt.async_db.util.**is_db_available**(*connection: Connection*, *database_name: str*) → bool

Verify if the database exists

async firebolt.async_db.util.**is_engine_running**(*connection: Connection*, *engine_url: str*) → bool

Verify if the engine is running

Client

The client package contains functionality for user authentication and logging requests.

Client.auth

Client.client

class firebolt.client.client.**AsyncClient**(**args: Any*, *account_name: Optional[str] = None*, *api_endpoint: str = 'api.app.firebolt.io'*, *auth: Optional[Auth] = None*, ***kwargs: Any*)

Bases: `FireboltClientMixin`, `AsyncClient`

An HTTP client, based on httpx.AsyncClient.

Asyncronously handles authentication for Firebolt database. Authentication can be passed through auth keyword as a tuple or as a FireboltAuth instance

account_id

User account id.

If account_name was provided during AsyncClient construction, returns it's id. Gets default account otherwise

Returns
Account ID

Return type
str

Raises
AccountNotFoundError – No account found with provided name

class firebolt.client.client.**Client**(**args: Any*, *account_name: Optional[str] = None*, *api_endpoint: str = 'api.app.firebolt.io'*, *auth: Optional[Auth] = None*, ***kwargs: Any*)

Bases: `FireboltClientMixin`, `Client`

An HTTP client, based on httpx.Client.

Handles the authentication for Firebolt database. Authentication can be passed through auth keyword as a tuple or as a FireboltAuth instance

property account_id: str

User account id.

If account_name was provided during Client construction, returns it's id. Gets default account otherwise

Returns
Account ID

Return type
str

Raises
AccountNotFoundError – No account found with provided name

Client.resource_manager_hooks

firebolt.client.resource_manager_hooks.**log_request**(*request: Request*) → None

Log HTTP requests.

Hook for an HTTP client

Parameters
request (*Request*) – Request to log

firebolt.client.resource_manager_hooks.**log_response**(*response: Response*) → None

Log HTTP response.

Hook for an HTTP client

Parameters
response (*Response*) – Response to log

firebolt.client.resource_manager_hooks.**raise_on_4xx_5xx**(*response: Response*) → None

Raise an error on HTTP response with error return code.

Hook for an HTTP client If an error is message is found raise as an ApiError

Parameters
response (*Response*) – Response to check for error code

Raises

- **RequestError** – Error during performing request
- **RuntimeError** – Error processing request on server
- **HTTPStatusError** – HTTP error

Common

The common package contains settings parameters and error exceptions.

Common.exception

Common.settings

class firebolt.common.settings.**Settings**(*_env_file: Optional[Union[str, PathLike]] = '<object object>'*, *_env_file_encoding: Optional[str] = None*, *_env_nested_delimiter: Optional[str] = None*, *_secrets_dir: Optional[Union[str, PathLike]] = None*, ***, *auth: Auth = None*, *user: str = None*, *password: SecretStr = None*, *access_token: str = None*, *account_name: str = None*, *server: str*, *default_region: str*, *use_token_cache: bool = True*)

Bases: `BaseSettings`

Settings for Firebolt SDK.

user

User name

Type
Optional[str]

password
User password

Type
Optional[str]

access_token
Access token to use for authentication. Mutually exclusive with user and password

Type
Optional[str]

account_name
Account name. Default user account is used if none provided

Type
Optional[str]

server
Environment api endpoint (Advanced). Default api endpoint is used if none provided

Type
Optional[str]

default_region
Default region for provisioning

Type
str

class Config
Bases: object

Internal pydantic config.

env_file = '.env'

access_token: str

account_name: str

auth: Auth

default_region: str

classmethod mutual_exclusive_with_creds(*values: dict*) → dict
Validate that either creds or token is provided.

Parameters
values (*dict*) – settings initial values

Returns
Validated settings values

Return type
dict

Raises
ValueError – Either both or none of credentials and token are provided

password: SecretStr

server: str

use_token_cache: bool

user: str

Common.urls

Common.token_storage

Common.util

Db

The db package enables connecting to a Firebolt database for synchronous queries.

Db.connection

class firebolt.db.connection.**Connection**(**args: Any*, ***kwargs: Any*)

Bases: BaseConnection

Firebolt database connection class. Implements PEP-249.

Parameters

- **engine_url** – Firebolt database engine REST API url
- **database** – Firebolt database name
- **username** – Firebolt account username
- **password** – Firebolt account password
- **api_endpoint** – Optional. Firebolt API endpoint. Used for authentication.

Note: Firebolt currenly doesn't support transactions so commit and rollback methods are not implemented.

---

api_endpoint

client_class: type

close() → None

Close connection and all underlying cursors.

cursor() → *Cursor*

cursor_class

alias of *Cursor*

database

engine_url

Db.cursor

class firebolt.db.cursor.**Cursor**(**args: Any*, ***kwargs: Any*)

Bases: BaseCursor

Class, responsible for executing queries to Firebolt Database. Should not be created directly, use *connection.cursor*

Parameters

- **description** – Information about a single result row
- **rowcount** – The number of rows produced by last query
- **closed** – True if connection is closed, False otherwise
- **arraysize** – Read/Write, specifies the number of rows to fetch at a time with the *fetchmany()* method

connection

execute(*query: str*, *parameters: Optional[Sequence[Union[int, float, str, datetime, date, bool, Decimal, Sequence]]] = None*, *set_parameters: Optional[Dict] = None*, *skip_parsing: bool = False*) → int

Prepare and execute a database query.

Supported features:

Parameterized queries: placeholder characters ('?') are substituted
with values provided in *parameters*. Values are formatted to be properly recognized by database and to exclude SQL injection.

Multi-statement queries: multiple statements, provided in a single query
and separated by semicolon are executed separatelly and sequentially. To switch to next statement result, *nextset* method should be used.

SET statements: to provide additional query execution parameters, execute
SET param=value statement before it. All parameters are stored in cursor object until it's closed. They can also be removed with *flush_parameters* method call.

Parameters

- **query** (*str*) – SQL query to execute
- **parameters** (*Optional[Sequence[ParameterType]]*) – A sequence of substitution parameters. Used to replace '?' placeholders inside a query with actual values
- **set_parameters** (*Optional[Dict]*) – List of set parameters to execute a query with. DEPRECATED: Use SET SQL statements instead
- **skip_parsing** (*bool*) – Flag to disable query parsing. This will disable parameterized, multi-statement and SET queries, while improving performance

Returns
Query row count

Return type
int

executemany(*query: str*, *parameters_seq: Sequence[Sequence[Union[int, float, str, datetime, date, bool, Decimal, Sequence]]]*) → int

Prepare and execute a database query.

Supports providing multiple substitution parameter sets, executing them as multiple statements sequentially.

Supported features:

Parameterized queries: placeholder characters ('?') are substituted
with values provided in *parameters*. Values are formatted to be properly recognized by database and to exclude SQL injection.

Multi-statement queries: multiple statements, provided in a single query
and separated by semicolon are executed separatelly and sequentially. To switch to next statement result, *nextset* method should be used.

SET statements: to provide additional query execution parameters, execute
SET param=value statement before it. All parameters are stored in cursor object until it's closed. They can also be removed with *flush_parameters* method call.

Parameters

- **query** (*str*) – SQL query to execute
- **parameters_seq** (*Sequence[Sequence[ParameterType]]*) – A sequence of substitution parameter sets. Used to replace '?' placeholders inside a query with actual values from each set in a sequence. Resulting queries for each subset are executed sequentially.

Returns
Query row count

Return type
int

fetchall() → List[List[Optional[Union[int, float, str, datetime, date, bool, list, Decimal]]]]

Fetch all remaining rows of a query result.

fetchmany(*size: Optional[int] = None*) → List[List[Optional[Union[int, float, str, datetime, date, bool, list, Decimal]]]]

Fetch the next set of rows of a query result, cursor.arraysize is default size.

fetchone() → Optional[List[Optional[Union[int, float, str, datetime, date, bool, list, Decimal]]]]

Fetch the next row of a query result set.

nextset() → Optional[bool]

Skip to the next available set, discarding any remaining rows from the current set. Returns True if operation was successful, None if there are no more sets to retrive

Model

The model package contains various classes and functions for managing Firebolt engines and databases.

Model.database

class firebolt.model.database.**Database**(*, *name: ConstrainedStrValue*, *compute_region_id: RegionKey*, *id: DatabaseKey = None*, *description: ConstrainedStrValue = None*, *emoji: ConstrainedStrValue = None*, *current_status: str = None*, *health_status: str = None*, *data_size_full: int = None*, *data_size_compressed: int = None*, *is_system_database: bool = None*, *storage_bucket_name: str = None*, *create_time: datetime = None*, *create_actor: str = None*, *last_update_time: datetime = None*, *last_update_actor: str = None*, *desired_status: str = None*)

Bases: FireboltBaseModel

A Firebolt database.

Databases belong to a region and have a description, but otherwise are not configurable.

attach_to_engine(*engine:* Engine, *is_default_engine: bool = False*) → Binding

Attach an engine to this database.

Parameters

- **engine** – The engine to attach.
- **is_default_engine** – Whether this engine should be used as default for this database. Only one engine can be set as default for a single database. This will overwrite any existing default.

compute_region_key: RegionKey

create_actor: Optional[str]

create_time: Optional[datetime]

current_status: Optional[str]

data_size_compressed: Optional[int]

data_size_full: Optional[int]

property database_id: Optional[str]

database_key: Optional[DatabaseKey]

delete() → *Database*

Delete a database from Firebolt.

Raises an error if there are any attached engines.

description: Optional[str]

desired_status: Optional[str]

emoji: Optional[str]

get_attached_engines() → List[*Engine*]

Get a list of engines that are attached to this database.

health_status: Optional[str]

is_system_database: Optional[bool]

last_update_actor: Optional[str]

last_update_time: Optional[datetime]

name: str

classmethod parse_obj_with_service(*obj: Any*, *database_service:* DatabaseService) → *Database*

storage_bucket_name: Optional[str]

update(*description: str*) → *Database*

Updates a database description.

class firebolt.model.database.**FieldMask**(*, *paths: Sequence[str]*)

Bases: FireboltBaseModel

paths: Sequence[str]

Model.engine

class firebolt.model.engine.**Engine**(*, *name: ConstrainedStrValue*, *compute_region_id: RegionKey*, *settings:* EngineSettings, *id: EngineKey = None*, *description: str = None*, *emoji: str = None*, *current_status:* EngineStatus *= None*, *current_status_summary:* EngineStatusSummary *= None*, *latest_revision_id: EngineRevisionKey = None*, *endpoint: str = None*, *endpoint_serving_revision_id: EngineRevisionKey = None*, *create_time: datetime = None*, *create_actor: str = None*, *last_update_time: datetime = None*, *last_update_actor: str = None*, *last_use_time: datetime = None*, *desired_status: str = None*, *health_status: str = None*, *endpoint_desired_revision_id: EngineRevisionKey = None*)

Bases: FireboltBaseModel

A Firebolt engine. Responsible for performing work (queries, data ingestion).

Engines are configured in *Settings* and in *EngineRevisionSpecification*.

attach_to_database(*database:* Database, *is_default_engine: bool = False*) → Binding

Attach this engine to a database.

Parameters

- **database** – Database to which the engine will be attached.
- **is_default_engine** – Whether this engine should be used as default for this database. Only one engine can be set as default for a single database. This will overwrite any existing default.

compute_region_key: RegionKey

create_actor: Optional[str]

create_time: Optional[datetime]

current_status: Optional[*EngineStatus*]

current_status_summary: Optional[*EngineStatusSummary*]

property database: Optional[*Database*]

delete() → *Engine*

Delete an Engine from Firebolt.

description: Optional[str]

desired_status: Optional[str]

emoji: Optional[str]

endpoint: Optional[str]

endpoint_desired_revision_key: Optional[EngineRevisionKey]

endpoint_serving_revision_key: Optional[EngineRevisionKey]

property engine_id: str

get_connection() → *Connection*

Get a connection to the attached database, for running queries.

Returns
engine connection instance

Return type
Connection

get_latest() → *Engine*

Get an up-to-date instance of the Engine from Firebolt.

health_status: Optional[str]

key: Optional[EngineKey]

last_update_actor: Optional[str]

last_update_time: Optional[datetime]

last_use_time: Optional[datetime]

latest_revision_key: Optional[EngineRevisionKey]

name: str

classmethod parse_obj_with_service(*obj: Any*, *engine_service:* EngineService) → *Engine*

restart(*wait_for_startup: bool = True*, *wait_timeout_seconds: int = 3600*) → *Engine*

Restart an engine.

Parameters

- **wait_for_startup** – If True, wait for startup to complete. If false, return immediately after requesting startup.
- **wait_timeout_seconds** – Number of seconds to wait for startup to complete before raising a TimeoutError.

Returns
The updated Engine from Firebolt.

settings: *EngineSettings*

start(*wait_for_startup: bool = True*, *wait_timeout_seconds: int = 3600*, *verbose: bool = False*) → *Engine*

Start an engine. If it's already started, do nothing.

Parameters

- **wait_for_startup** – If True, wait for startup to complete. If false, return immediately after requesting startup.
- **wait_timeout_seconds** – Number of seconds to wait for startup to complete before raising a TimeoutError.
- **verbose** – If True, print dots periodically while waiting for engine startup. If false, do not print any dots.

Returns

The updated Engine from Firebolt.

stop(*wait_for_stop: bool = False*, *wait_timeout_seconds: int = 3600*) → *Engine*

Stop an Engine running on Firebolt.

update(*name: Optional[str] = None*, *engine_type: Optional[EngineType] = None*, *scale: Optional[int] = None*, *spec: Optional[str] = None*, *auto_stop: Optional[int] = None*, *warmup: Optional[WarmupMethod] = None*, *description: Optional[str] = None*, *use_spot: Optional[bool] = None*) → *Engine*

update the engine, and returns an updated version of the engine, all parameters could be set to None, this would keep the old engine parameter value

class firebolt.model.engine.**EngineSettings**(***, *preset: str*, *auto_stop_delay_duration: ConstrainedStrValue*, *minimum_logging_level: str*, *is_read_only: bool*, *warm_up: str*)

Bases: FireboltBaseModel

Engine Settings.

See Also: *EngineRevisionSpecification* which also contains engine configuration.

auto_stop_delay_duration: **str**

classmethod default(*engine_type: EngineType = EngineType.GENERAL_PURPOSE*, *auto_stop_delay_duration: str = '1200s'*, *warm_up: WarmupMethod = WarmupMethod.PRELOAD_INDEXES*, *minimum_logging_level: str = 'ENGINE_SETTINGS_LOGGING_LEVEL_INFO'*) → *EngineSettings*

is_read_only: **bool**

minimum_logging_level: **str**

preset: **str**

warm_up: **str**

class firebolt.model.engine.**FieldMask**(***, *paths: Sequence[str]*)

Bases: FireboltBaseModel

paths: **Sequence[str]**

firebolt.model.engine.**check_attached_to_database**(*func: Callable*) → Callable

(Decorator) Ensure the engine is attached to a database.

firebolt.model.engine.**wait**(*seconds: int*, *timeout_time: float*, *error_message: str*, *verbose: bool*) → None

Model.engine_revision

class firebolt.model.engine_revision.**EngineRevision**(*, *specification:* EngineRevisionSpecification, *id: EngineRevisionKey = None*, *current_status: str = None*, *create_time: datetime = None*, *create_actor: str = None*, *last_update_time: datetime = None*, *last_update_actor: str = None*, *desired_status: str = None*, *health_status: str = None*)

Bases: FireboltBaseModel

A Firebolt Engine revision, which contains a Specification (instance types, counts).

As engines are updated with new settings, revisions are created.

create_actor: Optional[str]

create_time: Optional[datetime]

current_status: Optional[str]

desired_status: Optional[str]

health_status: Optional[str]

key: Optional[EngineRevisionKey]

last_update_actor: Optional[str]

last_update_time: Optional[datetime]

specification: *EngineRevisionSpecification*

class firebolt.model.engine_revision.**EngineRevisionSpecification**(*, *db_compute_instances_type_id: InstanceTypeKey*, *db_compute_instances_count: PositiveInt*, *db_compute_instances_use_spot: bool = False*, *db_version: str = ''*, *proxy_instances_type_id: InstanceTypeKey*, *proxy_instances_count: PositiveInt = 1*, *proxy_version: str = ''*)

Bases: FireboltBaseModel

An EngineRevision Specification.

Notably, it determines which instance types and how many of them its Engine gets.

See Also: *Settings*, which also contains engine configuration.

db_compute_instances_count: PositiveInt

db_compute_instances_type_key: InstanceTypeKey

db_compute_instances_use_spot: bool

db_version: str

proxy_instances_count: PositiveInt

proxy_instances_type_key: InstanceTypeKey

proxy_version: str

Model.instance_type

class firebolt.model.instance_type.**InstanceType**(*, *id: InstanceTypeKey*, *name: str*, *is_spot_available: bool = None*, *cpu_virtual_cores_count: int = None*, *memory_size_bytes: int = None*, *storage_size_bytes: int = None*, *price_per_hour_cents: float = None*, *create_time: datetime = None*, *last_update_time: datetime = None*)

Bases: FireboltBaseModel

cpu_virtual_cores_count: Optional[int]

create_time: Optional[datetime]

is_spot_available: Optional[bool]

key: InstanceTypeKey

last_update_time: Optional[datetime]

memory_size_bytes: Optional[int]

name: str

price_per_hour_cents: Optional[float]

storage_size_bytes: Optional[int]

Model.provider

class firebolt.model.provider.**Provider**(*, *id: str*, *name: str*, *create_time: datetime = None*, *display_name: str = None*, *last_update_time: datetime = None*)

Bases: FireboltBaseModel

create_time: Optional[datetime]

display_name: Optional[str]

last_update_time: Optional[datetime]

name: str

provider_id: str

Model.region

class firebolt.model.region.**Region**(*, *id: RegionKey*, *name: str*, *display_name: str = None*, *create_time: datetime = None*, *last_update_time: datetime = None*)

Bases: FireboltBaseModel

create_time: Optional[datetime]

display_name: Optional[str]

key: RegionKey

last_update_time: Optional[datetime]

name: str

Service

The service package enables launching and cataloging Firebolt engines and databases.

Service.database

class firebolt.service.database.**DatabaseService**(*resource_manager:* ResourceManager)

Bases: BaseService

create(*name: str*, *region: Optional[str] = None*, *description: Optional[str] = None*) → *Database*

Create a new Database on Firebolt.

Parameters

- **name** – Name of the database.
- **region** – Region name in which to create the database.

Returns

The newly created Database.

get(*id_: str*) → *Database*

Get a Database from Firebolt by its id.

get_by_name(*name: str*) → *Database*

Get a Database from Firebolt by its name.

get_id_by_name(*name: str*) → str

Get a Database id from Firebolt by its name.

get_many(*name_contains: Optional[str] = None*, *attached_engine_name_eq: Optional[str] = None*, *attached_engine_name_contains: Optional[str] = None*, *order_by: Optional[Union[str,* DatabaseOrder*]] = None*) → List[*Database*]

Get a list of databases on Firebolt.

Parameters

- **name_contains** – Filter for databases with a name containing this substring.
- **attached_engine_name_eq** – Filter for databases by an exact engine name.

- **attached_engine_name_contains** – Filter for databases by engines with a name containing this substring.
- **order_by** – Method by which to order the results.

:param See *firebolt.service.types.DatabaseOrder*.:

Returns
A list of databases matching the filters.

Service.engine

class firebolt.service.engine.**EngineService**(*resource_manager:* ResourceManager)

Bases: BaseService

create(*name: str*, *region: Optional[Union[str,* Region*]] = None*, *engine_type: Union[str,* EngineType*] = EngineType.GENERAL_PURPOSE*, *scale: int = 2*, *spec: Optional[str] = None*, *auto_stop: int = 20*, *warmup: Union[str,* WarmupMethod*] = WarmupMethod.PRELOAD_INDEXES*, *description: str = ''*, *engine_settings_kwargs: Dict[str, Any] = {}*, *revision_spec_kwargs: Dict[str, Any] = {}*) → *Engine*

Create a new Engine.

Parameters

- **name** – An identifier that specifies the name of the engine.
- **region** – The AWS region in which the engine runs.
- **engine_type** – The engine type. GENERAL_PURPOSE or DATA_ANALYTICS
- **scale** – The number of compute instances on the engine. The scale can be any int from 1 to 128.
- **spec** – Firebolt instance type. If not set will default to the cheapest instance.
- **auto_stop** – The amount of time (in minutes)
- **stops.** (*after which the engine automatically*) –
- **warmup** – The warmup method that should be used.

  MINIMAL - On-demand loading (both indexes and tables' data).

  PRELOAD_INDEXES - Load indexes only.

  PRELOAD_ALL_DATA - Full data auto-load (both indexes and table data - full warmup).
- **description** – A short description of the engine's purpose.

Returns
Engine with the specified settings.

get(*id_: str*) → *Engine*

Get an Engine from Firebolt by its id.

get_by_ids(*ids: List[str]*) → List[*Engine*]

Get multiple Engines from Firebolt by their ids.

get_by_name(*name: str*) → *Engine*

Get an Engine from Firebolt by its name.

get_many(*name_contains: Optional[str] = None*, *current_status_eq: Optional[str] = None*, *current_status_not_eq: Optional[str] = None*, *region_eq: Optional[str] = None*, *order_by: Optional[Union[str,* EngineOrder*]] = None*) → List[*Engine*]

Get a list of engines on Firebolt.

Parameters

- **name_contains** – Filter for engines with a name containing this substring.
- **current_status_eq** – Filter for engines with this status.
- **current_status_not_eq** – Filter for engines that do not have this status.
- **region_eq** – Filter for engines by region.
- **order_by** – Method by which to order the results. See [EngineOrder].

Returns

A list of engines matching the filters.

Service.instance_type

class firebolt.service.instance_type.**InstanceTypeService**(*resource_manager:* ResourceManager)

Bases: BaseService

cheapest_instance_in_region(*region:* Region) → Optional[*InstanceType*]

get_by_key(*instance_type_key: InstanceTypeKey*) → *InstanceType*

Get an instance type by key.

get_by_name(*instance_type_name: str*, *region_name: Optional[str] = None*) → *InstanceType*

Get an instance type by name.

Parameters

- **instance_type_name** – Name of the instance (eg. "i3.4xlarge").
- **region_name** – Name of the AWS region from which to get the instance. If not provided, use the default region name from the client.

Returns

The requested instance type.

get_instance_types_per_region(*region:* Region) → List[*InstanceType*]

List of instance types available on Firebolt in specified region.

property instance_types: List[*InstanceType*]

List of instance types available on Firebolt.

property instance_types_by_key: Dict[InstanceTypeKey, *InstanceType*]

Dict of {InstanceTypeKey to InstanceType}

property instance_types_by_name: Dict[InstanceTypeLookup, *InstanceType*]

Dict of {InstanceTypeLookup to InstanceType}

Service.manager

class firebolt.service.manager.**ResourceManager**(*settings: Optional[Settings] = None*)

Bases: `object`

ResourceManager to access various Firebolt resources:

- databases
- engines
- bindings (the bindings between an engine and a database)
- engine revisions (versions of an engine)

Also provides listings of:

- regions (AWS regions in which engines can run)
- instance types (AWS instance types which engines can use)

Service.provider

firebolt.service.provider.**get_provider_id**(*client: Client*) → str

Get the AWS provider_id.

Service.region

class firebolt.service.region.**RegionService**(*resource_manager: ResourceManager*)

Bases: `BaseService`

property default_region: ***Region***

Default AWS Region, could be provided from environment.

get_by_id(*id_: str*) → *Region*

Get an AWS Region by region_id.

get_by_key(*key: RegionKey*) → *Region*

Get an AWS Region by its key.

get_by_name(*name: str*) → *Region*

Get an AWS region by its name (eg. us-east-1).

property regions: **List[*Region*]**

List of available AWS Regions on Firebolt.

property regions_by_key: **Dict[RegionKey, *Region*]**

Dict of {RegionKey to Region}

property regions_by_name: **Dict[str, *Region*]**

Dict of {RegionLookup to Region}

Service.types

class firebolt.service.types.**DatabaseOrder**(*value*)

Bases: Enum

An enumeration.

DATABASE_ORDER_COMPUTE_REGION_ID_ASC = 'DATABASE_ORDER_COMPUTE_REGION_ID_ASC'

DATABASE_ORDER_COMPUTE_REGION_ID_DESC = 'DATABASE_ORDER_COMPUTE_REGION_ID_DESC'

DATABASE_ORDER_CREATE_ACTOR_ASC = 'DATABASE_ORDER_CREATE_ACTOR_ASC'

DATABASE_ORDER_CREATE_ACTOR_DESC = 'DATABASE_ORDER_CREATE_ACTOR_DESC'

DATABASE_ORDER_CREATE_TIME_ASC = 'DATABASE_ORDER_CREATE_TIME_ASC'

DATABASE_ORDER_CREATE_TIME_DESC = 'DATABASE_ORDER_CREATE_TIME_DESC'

DATABASE_ORDER_DATA_SIZE_COMPRESSED_ASC = 'DATABASE_ORDER_DATA_SIZE_COMPRESSED_ASC'

DATABASE_ORDER_DATA_SIZE_COMPRESSED_DESC =
'DATABASE_ORDER_DATA_SIZE_COMPRESSED_DESC'

DATABASE_ORDER_DATA_SIZE_FULL_ASC = 'DATABASE_ORDER_DATA_SIZE_FULL_ASC'

DATABASE_ORDER_DATA_SIZE_FULL_DESC = 'DATABASE_ORDER_DATA_SIZE_FULL_DESC'

DATABASE_ORDER_LAST_UPDATE_ACTOR_ASC = 'DATABASE_ORDER_LAST_UPDATE_ACTOR_ASC'

DATABASE_ORDER_LAST_UPDATE_ACTOR_DESC = 'DATABASE_ORDER_LAST_UPDATE_ACTOR_DESC'

DATABASE_ORDER_LAST_UPDATE_TIME_ASC = 'DATABASE_ORDER_LAST_UPDATE_TIME_ASC'

DATABASE_ORDER_LAST_UPDATE_TIME_DESC = 'DATABASE_ORDER_LAST_UPDATE_TIME_DESC'

DATABASE_ORDER_NAME_ASC = 'DATABASE_ORDER_NAME_ASC'

DATABASE_ORDER_NAME_DESC = 'DATABASE_ORDER_NAME_DESC'

DATABASE_ORDER_UNSPECIFIED = 'DATABASE_ORDER_UNSPECIFIED'

class firebolt.service.types.**EngineOrder**(*value*)

Bases: Enum

An enumeration.

ENGINE_ORDER_COMPUTE_REGION_ID_ASC = 'ENGINE_ORDER_COMPUTE_REGION_ID_ASC'

ENGINE_ORDER_COMPUTE_REGION_ID_DESC = 'ENGINE_ORDER_COMPUTE_REGION_ID_DESC'

ENGINE_ORDER_CREATE_ACTOR_ASC = 'ENGINE_ORDER_CREATE_ACTOR_ASC'

ENGINE_ORDER_CREATE_ACTOR_DESC = 'ENGINE_ORDER_CREATE_ACTOR_DESC'

ENGINE_ORDER_CREATE_TIME_ASC = 'ENGINE_ORDER_CREATE_TIME_ASC'

ENGINE_ORDER_CREATE_TIME_DESC = 'ENGINE_ORDER_CREATE_TIME_DESC'

ENGINE_ORDER_CURRENT_STATUS_ASC = 'ENGINE_ORDER_CURRENT_STATUS_ASC'

ENGINE_ORDER_CURRENT_STATUS_DESC = 'ENGINE_ORDER_CURRENT_STATUS_DESC'

ENGINE_ORDER_LAST_UPDATE_ACTOR_ASC = 'ENGINE_ORDER_LAST_UPDATE_ACTOR_ASC'

ENGINE_ORDER_LAST_UPDATE_ACTOR_DESC = 'ENGINE_ORDER_LAST_UPDATE_ACTOR_DESC'

ENGINE_ORDER_LAST_UPDATE_TIME_ASC = 'ENGINE_ORDER_LAST_UPDATE_TIME_ASC'

ENGINE_ORDER_LAST_UPDATE_TIME_DESC = 'ENGINE_ORDER_LAST_UPDATE_TIME_DESC'

ENGINE_ORDER_LATEST_REVISION_CURRENT_STATUS_ASC =
'ENGINE_ORDER_LATEST_REVISION_CURRENT_STATUS_ASC'

ENGINE_ORDER_LATEST_REVISION_CURRENT_STATUS_DESC =
'ENGINE_ORDER_LATEST_REVISION_CURRENT_STATUS_DESC'

ENGINE_ORDER_LATEST_REVISION_SPECIFICATION_DB_COMPUTE_INSTANCES_COUNT_ASC =
'ENGINE_ORDER_LATEST_REVISION_SPECIFICATION_DB_COMPUTE_INSTANCES_COUNT_ASC'

ENGINE_ORDER_LATEST_REVISION_SPECIFICATION_DB_COMPUTE_INSTANCES_COUNT_DESC =
'ENGINE_ORDER_LATEST_REVISION_SPECIFICATION_DB_COMPUTE_INSTANCES_COUNT_DESC'

ENGINE_ORDER_LATEST_REVISION_SPECIFICATION_DB_COMPUTE_INSTANCES_TYPE_ID_ASC =
'ENGINE_ORDER_LATEST_REVISION_SPECIFICATION_DB_COMPUTE_INSTANCES_TYPE_ID_ASC'

ENGINE_ORDER_LATEST_REVISION_SPECIFICATION_DB_COMPUTE_INSTANCES_TYPE_ID_DESC =
'ENGINE_ORDER_LATEST_REVISION_SPECIFICATION_DB_COMPUTE_INSTANCES_TYPE_ID_DESC'

ENGINE_ORDER_NAME_ASC = 'ENGINE_ORDER_NAME_ASC'

ENGINE_ORDER_NAME_DESC = 'ENGINE_ORDER_NAME_DESC'

ENGINE_ORDER_UNSPECIFIED = 'ENGINE_ORDER_UNSPECIFIED'

class firebolt.service.types.**EngineStatus**(*value*)

Bases: Enum

Detailed engine status.

See: https://api.dev.firebolt.io/devDocs#operation/coreV1GetEngine

ENGINE_STATUS_CREATED = 'ENGINE_STATUS_CREATED'

Engine status was created.

ENGINE_STATUS_DELETED = 'ENGINE_STATUS_DELETED'

Engine is soft-deleted.

ENGINE_STATUS_PROVISIONING_FAILED = 'ENGINE_STATUS_PROVISIONING_FAILED'

Engine initialization failed due to error.

ENGINE_STATUS_PROVISIONING_FINISHED = 'ENGINE_STATUS_PROVISIONING_FINISHED'

Engine initialization was finished successfully.

ENGINE_STATUS_PROVISIONING_PENDING = 'ENGINE_STATUS_PROVISIONING_PENDING'

Engine initialization request was sent.

ENGINE_STATUS_PROVISIONING_STARTED = 'ENGINE_STATUS_PROVISIONING_STARTED'

Engine initialization request was received and initialization process started.

ENGINE_STATUS_RUNNING_IDLE = 'ENGINE_STATUS_RUNNING_IDLE'

Engine is initialized, but there are no running or starting engine revisions.

ENGINE_STATUS_RUNNING_REVISIONS_TERMINATING = 'ENGINE_STATUS_RUNNING_REVISIONS_TERMINATING'

Engine is initialized, all child revisions are being terminated.

ENGINE_STATUS_RUNNING_REVISION_CHANGE_FAILED = 'ENGINE_STATUS_RUNNING_REVISION_CHANGE_FAILED'

Engine is ready (serves an engine revision), replacement revision failed to provision or start.

ENGINE_STATUS_RUNNING_REVISION_CHANGING = 'ENGINE_STATUS_RUNNING_REVISION_CHANGING'

Engine is ready (serves an engine revision), zero-downtime replacement revision is starting.

ENGINE_STATUS_RUNNING_REVISION_RESTARTING = 'ENGINE_STATUS_RUNNING_REVISION_RESTARTING'

Engine is initialized, replacement of the revision with a downtime is in progress.

ENGINE_STATUS_RUNNING_REVISION_RESTART_FAILED = 'ENGINE_STATUS_RUNNING_REVISION_RESTART_FAILED'

Engine is initialized, replacement revision failed to provision or start.

ENGINE_STATUS_RUNNING_REVISION_SERVING = 'ENGINE_STATUS_RUNNING_REVISION_SERVING'

Engine is ready (serves an engine revision).

ENGINE_STATUS_RUNNING_REVISION_STARTING = 'ENGINE_STATUS_RUNNING_REVISION_STARTING'

Engine is initialized, there are no running engine revision but it's starting.

ENGINE_STATUS_RUNNING_REVISION_STARTUP_FAILED = 'ENGINE_STATUS_RUNNING_REVISION_STARTUP_FAILED'

Engine is initialized, initial revision is failed to provision or start.

ENGINE_STATUS_TERMINATION_F = 'ENGINE_STATUS_TERMINATION_FAILED'

Engine termination failed.

ENGINE_STATUS_TERMINATION_FIN = 'ENGINE_STATUS_TERMINATION_FINISHED'

Engine termination finished.

ENGINE_STATUS_TERMINATION_PENDING = 'ENGINE_STATUS_TERMINATION_PENDING'

Engine termination request was sent.

ENGINE_STATUS_TERMINATION_ST = 'ENGINE_STATUS_TERMINATION_STARTED'

Engine termination started.

ENGINE_STATUS_UNSPECIFIED = 'ENGINE_STATUS_UNSPECIFIED'

Logical record is created, however underlying infrastructure is not initialized. In other words this means that engine is stopped.

class firebolt.service.types.**EngineStatusSummary**(*value*)

Bases: Enum

Engine summary status.

See: https://api.dev.firebolt.io/devDocs#operation/coreV1GetEngine

ENGINE_STATUS_SUMMARY_DELETED = 'ENGINE_STATUS_SUMMARY_DELETED'

Infrastructure is terminated, engine data is deleted.

ENGINE_STATUS_SUMMARY_DELETING = 'ENGINE_STATUS_SUMMARY_DELETING'

Termination is in progress. All infrastructure that belongs to this engine will be completely destroyed.

ENGINE_STATUS_SUMMARY_FAILED = 'ENGINE_STATUS_SUMMARY_FAILED'

Failed to start or stop. This status only indicates that there were issues during provisioning operations. If engine enters this status, all infrastructure should be stopped/terminated already.

ENGINE_STATUS_SUMMARY_REPAIRING = 'ENGINE_STATUS_SUMMARY_REPAIRING'

Underlying infrastructure has issues and is being repaired. Engine is still running, but it's not fully healthy and some queries may fail.

ENGINE_STATUS_SUMMARY_RESTARTING = 'ENGINE_STATUS_SUMMARY_RESTARTING'

Hard restart (full stop/start cycle) is in progress. Underlying infrastructure is being recreated.

ENGINE_STATUS_SUMMARY_RESTARTING_INITIALIZING = 'ENGINE_STATUS_SUMMARY_RESTARTING_INITIALIZING'

Hard restart (full stop/start cycle) is in progress. Underlying infrastructure is ready, waiting for PackDB cluster to initialize and start. This status is logically the same as ENGINE_STATUS_SUMMARY_STARTING_INITIALIZING, but used during restart cycle.

ENGINE_STATUS_SUMMARY_RUNNING = 'ENGINE_STATUS_SUMMARY_RUNNING'

Fully started. Engine is ready to serve requests.

ENGINE_STATUS_SUMMARY_STARTING = 'ENGINE_STATUS_SUMMARY_STARTING'

Provisioning process is in progress. We are creating cloud infra for this engine.

ENGINE_STATUS_SUMMARY_STARTING_INITIALIZING = 'ENGINE_STATUS_SUMMARY_STARTING_INITIALIZING'

Provisioning process is complete. We are now waiting for PackDB cluster to initialize and start.

ENGINE_STATUS_SUMMARY_STOPPED = 'ENGINE_STATUS_SUMMARY_STOPPED'

Fully stopped.

ENGINE_STATUS_SUMMARY_STOPPING = 'ENGINE_STATUS_SUMMARY_STOPPING'

Stop is in progress.

ENGINE_STATUS_SUMMARY_UNSPECIFIED = 'ENGINE_STATUS_SUMMARY_UNSPECIFIED'

Status unspecified

ENGINE_STATUS_SUMMARY_UPGRADING = 'ENGINE_STATUS_SUMMARY_UPGRADING'

Version of the PackDB is changing. This is zero downtime operation that does not affect engine work. This status is reserved for future use (not used fow now).

class firebolt.service.types.**EngineType**(*value*)

Bases: Enum

An enumeration.

DATA_ANALYTICS = 'DATA_ANALYTICS'

GENERAL_PURPOSE = 'GENERAL_PURPOSE'

api_settings_preset_name

class firebolt.service.types.**WarmupMethod**(*value*)

Bases: Enum

An enumeration.

```
MINIMAL = 'MINIMAL'

PRELOAD_ALL_DATA = 'PRELOAD_ALL_DATA'

PRELOAD_INDEXES = 'PRELOAD_INDEXES'

api_name
```

2.4.3 Indices and tables

- genindex

PYTHON MODULE INDEX

f

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

W