QCPortal Documentation

Release 5

The QCArchive Development Team

Sep 28, 2021

GETTING STARTED

QCPortal is a front-end to a QCFractal server which allows the querying, visualization, manipulation of hosted data.

QCPortal emphasizes the following virtues:

- **Organize:** Large sets of computations are organized into Collections for easy reference and manipulation.
- **Reproducibility:** All steps of commonly used pipelines are elucidated in the input without additional human intervention.
- **Exploration:** Explore and query of all data contained within a FractalServer.
- **Visualize:** Plot graphs within Jupyter notebooks or provide 3D graphics of molecules.
- **Accessibility:** Easily share quantum chemistry data with colleagues or the community through accessibility settings.

CHAPTER

ONE

COLLECTIONS

Collections are objects that can reference tens or millions of individual computations and provide handles to access and visualize this data. All collections support the possibility of computing with and comparing multiple methods. There are many types of collections such as:

- *Exploring the Datasets* - A collection for a set of molecules and their computed properties.
- *Reaction Dataset* - A collection for chemical reactions and intermolecular interactions.
- *Optimization Dataset* - A collection for geometry optimization of a set of molecules.
- *TorsionDrive Dataset* - A collection for the TorsionDrive pipeline.

There are many types of collections and more are being added to index and organize computations for every use case.

CHAPTER

TWO

VISUALIZATION

Advanced visualization routines based off Plotly is provided out of the box to allow interactive statistics and rich visual information. In addition, popular molecular visualization tools like 3dMol.js provide interactive molecules within the Jupyter notebook ecosystem.

CHAPTER

THREE

INDEX

Getting Started

- *Install QCPortal*

3.1 Install QCPortal

QCPortal can be installed using `conda` or `pip`.

3.1.1 Conda

The following command installs QCPortal and its dependencies using conda:

```
>>> conda install qcportal -c conda-forge
```

The QCPortal package is maintained on the conda-forge channel.

3.1.2 Pip

QCPortal can also be installed using `pip`:

```
>>> pip install qcportal
```

3.1.3 Testing the Installation

The installation process for QCPortal can be verified using `pytest` through running the tests. The `pytest` package can be installed using `conda`:

```
>>> conda install pytest -c conda-forge
```

or `pip`:

```
>>> pip install pytest
```

After installing `pytest`, the following command collects the tests and runs them individually in order to verify the installation:

```
>>> pytest --pyargs qcportal
```

3.1.4 Developing from Source

The QCPortal is a part of the QCFractal package and resides in the `qcfractal.interface` folder. Developers can make contributions to QCPortal by accessing the source code here.

Collections

Collections are the primary way of viewing and generating new data.

- *Overview*
- *Exploring the Datasets*
- *Reaction Dataset*
- *Optimization Dataset*
- *TorsionDrive Dataset*
- *Common Tasks*

3.2 Overview

Collections are organizational objects that keep track of or compute new results, and provide helper functions for analysis and visualization.

3.2.1 Collections querying

Once a `FractalClient` has been created, it can query a list of all collections currently held on the server.

```
>>> client.list_collections()
{"ReactionDataset": ["S22"]}
```

A collection can then be pulled from the server as follows:

```
>>> client.get_collection("ReactionDataset", "S22")
Dataset(id=`5b7f1fd57b87872d2c5d0a6d`, name=`S22`, client="localhost:7777")
```

3.2.2 Available collections

Below is a complete list of collection types available from QCPortal. All collections support the possibility of computing with and comparing multiple methods.

- *Exploring the Datasets* - A collection of molecular sets and their computed properties.
- *Reaction Dataset* - A collection of chemical reactions and intermolecular interactions.
- *Optimization Dataset* - A collection of geometry optimization results for molecular sets.
- *TorsionDrive Dataset* - A collection of TorsionDrive pipeline data.

3.3 Exploring the Datasets

The *Dataset* collection represents a table whose rows correspond to *Molecules* and whose columns correspond to properties. Columns may either result from QCFractal-based calculations or be contributed from outside sources. For example, the QM9 dataset in QCArchive contains small organic molecules with up to 9 heavy atoms, and includes the original reported PBE0 energies, as well as energies calculated with a variety of other density functionals and basis sets.

The existing *Datasets* can be listed with *FractalClient.list_collections("Dataset")* and obtained using *FractalClient.get_collection("Dataset", name)*.

3.3.1 Querying the Data

Available result specifications (method, basis set, program, keyword, driver combinations) in a *Dataset* may be listed with the *list_values* method. Values are queried with the *get_values* method. For results computed using QCFractal, the underlying *Records* are retrieved with *get_records*.

For further details about how to query *Datasets* see the QCArchive examples.

3.3.2 Statistics and Visualization

Statistical operations on *Datasets* may be performed using *statistics* command and plotted using the *visualize* command.

For examples of visualizing *Datasets*, see the QCArchive examples.

3.3.3 Creating the Datasets

Construct an empty *Dataset*:

```
>>> import qcportal as ptl
>>> client = plt.FractalClient()  # add server and login information as needed
>>> ds = ptl.collections.Dataset("name", client=client)
```

The primary index of a *Dataset* is a list of *Molecules*. *Molecules* can be added to a *Dataset* with *add_entry*:

```
>>> ds.add_entry(name, molecule)
```

Once all *Molecules* are added, the changes can be committed to the server with `save` method. Note that this requires write permissions.

```
>>> ds.save()
```

3.3.4 Computational Tasks

Computations on the molecules within the *Datasets* can be performed using the *compute* command. If the results of the requested computation already exist in the *Dataset*, they will be reused to avoid recomputation. Note that for perfoming computations, compute permissions is required.

```
>>> models = {('b3lyp', 'def2-svp'), ('mp2', 'cc-pVDZ')}

>>> for method, basis in models:
>>>     print(method, basis)
>>>     spec = {"program": "psi4",
>>>         "method": method,
>>>         "basis": basis,
>>>         "keywords": "my_keywords",
>>>         "tag": "mgwtfm"}
>>>     ds.compute(**spec)
```

Note: A default quantum chemical program and a set of computational keywords can be specified for a *Dataset*. These default values will be used in the *compute*, *get_values*, and *get_records* methods.

```
>>> ds.set_default_program("psi4")

>>> keywords = ptl.models.KeywordSet(values={'maxiter': 1000,
>>>                                          'e_convergence': 8,
>>>                                          'guess': 'sad',
>>>                                          'scf_type': 'df'})

>>> ds.add_keywords("my_keywords", "psi4", keywords, default=True)

>>> ds.save()
```

3.3.5 API

class qcportal.collections.**Dataset**(*name: str*, *client: Optional[FractalClient] = None*, ***kwargs: Any*)

The Dataset class for homogeneous computations on many molecules.

Variables

- **client** (*client.FractalClient*) – A FractalClient connected to a server
- **data** (*dict*) – JSON representation of the database backbone
- **df** (*pd.DataFrame*) – The underlying dataframe for the Dataset object

class DataModel(*, *id: str = 'local'*, *name: str*, *collection: str*, *provenance: Dict[str, str] = {}*, *tags: List[str] = []*, *tagline: str = None*, *description: str = None*, *group: str = 'default'*, *visibility: bool = True*, *view_url_hdf5: str = None*, *view_url_plaintext: str = None*, *view_metadata: Dict[str, str] = None*, *view_available: bool = False*, *metadata: Dict[str, Any] = {}*, *default_program: str = None*, *default_keywords: Dict[str, str] = {}*, *default_driver: str = 'energy'*, *default_units: str = 'kcal / mol'*, *default_benchmark: str = None*, *alias_keywords: Dict[str, Dict[str, str]] = {}*, *records: List[qcportal.collections.dataset.MoleculeEntry] = None*, *contributed_values: Dict[str, qcportal.collections.dataset.ContributedValues] = None*, *history: Set[Tuple[str, str, str, Optional[str], Optional[str]]] = {}*, *history_keys: Tuple[str, str, str, str, str] = ('driver', 'program', 'method', 'basis', 'keywords')*)

Parameters

- **id** (*str, Default: local*)
- **name** (*str*)
- **collection** (*str*)
- **provenance** (*Dict[str], Default: {}*)
- **tags** (*List[str], Default: []*)
- **tagline** (*str, Optional*)
- **description** (*str, Optional*)
- **group** (*str, Default: default*)
- **visibility** (*bool, Default: True*)
- **view_url_hdf5** (*str, Optional*)
- **view_url_plaintext** (*str, Optional*)
- **view_metadata** (*Dict[str], Optional*)
- **view_available** (*bool, Default: False*)
- **metadata** (*Dict[Any], Default: {}*)
- **default_program** (*str, Optional*)
- **default_keywords** (*Dict[str], Default: {}*)
- **default_driver** (*str, Default: energy*)
- **default_units** (*str, Default: kcal / mol*)
- **default_benchmark** (*str, Optional*)
- **alias_keywords** (*Dict[Dict[str]], Default: {}*)
- **records** (MoleculeEntry, Optional)
- **contributed_values** (ContributedValues, Optional)
- **history** (*Set[Tuple[str, str, str, str, str]], Default: set()*)
- **history_keys** (*Tuple[str, str, str, str, str], Default: ('driver', 'program', 'method', 'basis', 'keywords')*)

add_contributed_values(*contrib: qcportal.collections.dataset.ContributedValues*, *overwrite: bool = False*) → None

Adds a ContributedValues to the database. Be sure to call save() to commit changes to the server.

Parameters

- **contrib** (*ContributedValues*) – The ContributedValues to add.
- **overwrite** (*bool, optional*) – Overwrites pre-existing values

`add_entry`(*name: str*, *molecule: Molecule*, ***kwargs: Dict[str, Any]*) → None
Adds a new entry to the Dataset

Parameters

- **name** (*str*) – The name of the record
- **molecule** (*Molecule*) – The Molecule associated with this record
- ****kwargs** (*Dict[str, Any]*) – Additional arguments to pass to the record

`add_keywords`(*alias: str*, *program: str*, *keyword: KeywordSet*, *default: bool = False*) → bool
Adds an option alias to the dataset. Not that keywords are not present until a save call has been completed.

Parameters

- **alias** (*str*) – The alias of the option
- **program** (*str*) – The compute program the alias is for
- **keyword** (*KeywordSet*) – The Keywords object to use.
- **default** (*bool, optional*) – Sets this option as the default for the program

`compute`(*method: str*, *basis: Optional[str] = None*, ***, *keywords: Optional[str] = None*, *program: Optional[str] = None*, *subset: Optional[Set[str]] = None*, *tag: Optional[str] = None*, *priority: Optional[str] = None*, *protocols: Optional[Dict[str, Any]] = None*) → qcportal.models.rest_models.ComputeResponse
Executes a computational method for all reactions in the Dataset. Previously completed computations are not repeated.

Parameters

- **method** (*str*) – The computational method to compute (B3LYP)
- **basis** (*Optional[str], optional*) – The computational basis to compute (6-31G)
- **keywords** (*Optional[str], optional*) – The keyword alias for the requested compute
- **program** (*Optional[str], optional*) – The underlying QC program
- **tag** (*Optional[str], optional*) – The queue tag to use when submitting compute requests.
- **priority** (*Optional[str], optional*) – The priority of the jobs low, medium, or high.
- **protocols** (*Optional[Dict[str, Any]], optional*) – Protocols for store more or less data per field. Current valid protocols: {'wavefunction'}
- **subset** (*Set[str], optional*) – Computes only a subset of the dataset.

Returns

An object that contains the submitted ObjectIds of the new compute. This object has the following fields:

- ids: The ObjectId's of the task in the order of input molecules
- submitted: A list of ObjectId's that were submitted to the compute queue
- existing: A list of ObjectId's of tasks already in the database

Return type ComputeResponse

download(*local_path: Optional[Union[str, pathlib.Path]] = None*, *verify: bool = True*, *progress_bar: bool = True*) → None

Download a remote view if available. The dataset will use this view to avoid server queries for calls to: - get_entries - get_molecules - get_values - list_values

Parameters

- **local_path** (*Optional[Union[str, Path]], optional*) – Local path the store downloaded view. If None, the view will be stored in a temporary file and deleted on exit.
- **verify** (*bool, optional*) – Verify download checksum. Default: True.
- **progress_bar** (*bool, optional*) – Display a download progress bar. Default: True

get_entries(*subset: Optional[List[str]] = None*, *force: bool = False*) → pandas.core.frame.DataFrame

Provides a list of entries for the dataset

Parameters

- **subset** (*Optional[List[str]], optional*) – The indices of the desired subset. Return all indices if subset is None.
- **force** (*bool, optional*) – skip cache

Returns A dataframe containing entry names and specifciations. For Dataset, specifications are molecule ids. For ReactionDataset, specifications describe reaction stoichiometry.

Return type pd.DataFrame

get_index(*subset: Optional[List[str]] = None*, *force: bool = False*) → List[str]

Returns the current index of the database.

Returns **ret** – The names of all reactions in the database

Return type List[str]

get_keywords(*alias: str*, *program: str*, *return_id: bool = False*) → Union[KeywordSet, str]

Pulls the keywords alias from the server for inspection.

Parameters

- **alias** (*str*) – The keywords alias.
- **program** (*str*) – The program the keywords correspond to.
- **return_id** (*bool, optional*) – If True, returns the `id` rather than the `KeywordSet` object. Description

Returns The requested `KeywordSet` or `KeywordSet id`.

Return type Union['KeywordSet', str]

get_molecules(*subset: Optional[Union[str, Set[str]]] = None*, *force: bool = False*) → Union[pandas.core.frame.DataFrame, Molecule]

Queries full Molecules from the database.

Parameters

- **subset** (*Optional[Union[str, Set[str]]], optional*) – The index subset to query on
- **force** (*bool, optional*) – Force pull of molecules from server

Returns Either a DataFrame of indexed Molecules or a single Molecule if a single subset string was provided.

Return type Union[pd.DataFrame, 'Molecule']

get_records(*method: str*, *basis: Optional[str] = None*, ***, *keywords: Optional[str] = None*, *program: Optional[str] = None*, *include: Optional[List[str]] = None*, *subset: Optional[Union[str, Set[str]]] = None*, *merge: bool = False*, *status: Optional[List[str]] = None*) → Union[pandas.core.frame.DataFrame, ResultRecord]

Queries full ResultRecord objects from the database.

Parameters

- **method** (*str*) – The computational method to query on (B3LYP)
- **basis** (*Optional[str], optional*) – The computational basis query on (6-31G)
- **keywords** (*Optional[str], optional*) – The option token desired
- **program** (*Optional[str], optional*) – The program to query on
- **include** (*Optional[List[str]], optional*) – The attributes to return. Otherwise returns ResultRecord objects.
- **subset** (*Optional[Union[str, Set[str]]], optional*) – The index subset to query on
- **merge** (*bool*) – Merge multiple results into one (as in the case of DFT-D3). This only works when include=['return_results'], as in get_values.
- **status** (*List[str]*) – Include only records with these statuses. By default, obtain all records

Returns Either a DataFrame of indexed ResultRecords or a single ResultRecord if a single subset string was provided.

Return type Union[pd.DataFrame, 'ResultRecord']

get_values(*method: Optional[Union[List[str], str]] = None*, *basis: Optional[Union[List[str], str]] = None*, *keywords: Optional[str] = None*, *program: Optional[str] = None*, *driver: Optional[str] = None*, *name: Optional[Union[List[str], str]] = None*, *native: Optional[bool] = None*, *subset: Optional[Union[List[str], str]] = None*, *force: bool = False*) → pandas.core.frame.DataFrame

Obtains values matching the search parameters provided for the expected *return_result* values. Defaults to the standard programs and keywords if not provided.

Note that unlike *get_records*, *get_values* will automatically expand searches and return multiple method and basis combinations simultaneously.

None is a wildcard selector. To search for *None*, use *"None"*.

Parameters

- **method** (*Optional[Union[str, List[str]]], optional*) – The computational method (B3LYP)
- **basis** (*Optional[Union[str, List[str]]], optional*) – The computational basis (6-31G)
- **keywords** (*Optional[str], optional*) – The keyword alias
- **program** (*Optional[str], optional*) – The underlying QC program
- **driver** (*Optional[str], optional*) – The type of calculation (e.g. energy, gradient, hessian, dipole…)
- **name** (*Optional[Union[str, List[str]]], optional*) – Canonical name of the record. Overrides the above selectors.
- **native** (*Optional[bool], optional*) – True: only include data computed with QCFractal False: only include data contributed from outside sources None: include both
- **subset** (*Optional[List[str]], optional*) – The indices of the desired subset. Return all indices if subset is None.
- **force** (*bool, optional*) – Data is typically cached, forces a new query if True

Returns A DataFrame of values with columns corresponding to methods and rows corresponding to molecule entries.

Return type DataFrame

`list_keywords()` → pandas.core.frame.DataFrame
Lists keyword aliases for each program in the dataset.

Returns A dataframe containing programs, keyword aliases, KeywordSet ids, and whether those keywords are the default for a program. Indexed on program.

Return type pd.DataFrame

`list_records`(*dftd3: bool = False*, *pretty: bool = True*, ***search: Optional[Union[str, List[str]]]*) → pandas.core.frame.DataFrame
Lists specifications of available records, i.e. method, program, basis set, keyword set, driver combinations *None* is a wildcard selector. To search for *None*, use *"None"*.

Parameters

- **pretty** (*bool*) – Replace NaN with "None" in returned DataFrame
- ****search** (*Dict[str, Optional[str]]*) – Allows searching to narrow down return.

Returns Record specifications matching **search.

Return type DataFrame

`list_values`(*method: Optional[Union[List[str], str]] = None*, *basis: Optional[Union[List[str], str]] = None*, *keywords: Optional[str] = None*, *program: Optional[str] = None*, *driver: Optional[str] = None*, *name: Optional[Union[List[str], str]] = None*, *native: Optional[bool] = None*, *force: bool = False*) → pandas.core.frame.DataFrame
Lists available data that may be queried with get_values. Results may be narrowed by providing search keys. *None* is a wildcard selector. To search for *None*, use *"None"*.

Parameters

- **method** (*Optional[Union[str, List[str]]], optional*) – The computational method (B3LYP)
- **basis** (*Optional[Union[str, List[str]]], optional*) – The computational basis (6-31G)
- **keywords** (*Optional[str], optional*) – The keyword alias
- **program** (*Optional[str], optional*) – The underlying QC program
- **driver** (*Optional[str], optional*) – The type of calculation (e.g. energy, gradient, hessian, dipole…)
- **name** (*Optional[Union[str, List[str]]], optional*) – The canonical name of the data column
- **native** (*Optional[bool], optional*) – True: only include data computed with QCFractal False: only include data contributed from outside sources None: include both
- **force** (*bool, optional*) – Data is typically cached, forces a new query if True

Returns A DataFrame of the matching data specifications

Return type DataFrame

`set_default_benchmark`(*benchmark: str*) → bool
Sets the default benchmark value.

Parameters **benchmark** (*str*) – The benchmark to default to.

`set_default_program`(*program: str*) → bool
Sets the default program.

Parameters **program** (*str*) – The program to default to.

set_view(*path: Union[str, pathlib.Path]*) → None
Set a dataset to use a local view.

Parameters **path** (*Union[str, Path]*) – path to an hdf5 file representing a view for this dataset

statistics(*stype: str*, *value: str*, *bench: Optional[str] = None*, ***kwargs: Dict[str, Any]*) → Union[numpy.ndarray, pandas.core.series.Series, numpy.float64]
Provides statistics for various columns in the underlying dataframe.

Parameters

- **stype** (*str*) – The type of statistic in question
- **value** (*str*) – The method string to compare
- **bench** (*str, optional*) – The benchmark method for the comparison, defaults to *default_benchmark*.
- **kwargs** (*Dict[str, Any]*) – Additional kwargs to pass to the statistics functions

Returns Returns an ndarray, Series, or float with the requested statistics depending on input.

Return type np.ndarray, pd.Series, float

to_file(*path: Union[str, pathlib.Path]*, *encoding: str*) → None
Writes a view of the dataset to a file

Parameters

- **path** (*Union[str, Path]*) – Where to write the file
- **encoding** (*str*) – Options: plaintext, hdf5

visualize(*method: Optional[str] = None*, *basis: Optional[str] = None*, *keywords: Optional[str] = None*, *program: Optional[str] = None*, *groupby: Optional[str] = None*, *metric: str = 'UE'*, *bench: Optional[str] = None*, *kind: str = 'bar'*, *return_figure: Optional[bool] = None*, *show_incomplete: bool = False*) → plotly.Figure

Parameters

- **method** (*Optional[str], optional*) – Methods to query
- **basis** (*Optional[str], optional*) – Bases to query
- **keywords** (*Optional[str], optional*) – Keyword aliases to query
- **program** (*Optional[str], optional*) – Programs aliases to query
- **groupby** (*Optional[str], optional*) – Groups the plot by this index.
- **metric** (*str, optional*) – The metric to use either UE (unsigned error) or URE (unsigned relative error)
- **bench** (*Optional[str], optional*) – The benchmark level of theory to use
- **kind** (*str, optional*) – The kind of chart to produce, either 'bar' or 'violin'
- **return_figure** (*Optional[bool], optional*) – If True, return the raw plotly figure. If False, returns a hosted iPlot. If None, return a iPlot display in Jupyter notebook and a raw plotly figure in all other circumstances.
- **show_incomplete** (*bool, optional*) – Display statistics method/basis set combinations where results are incomplete

Returns The requested figure.

Return type plotly.Figure

3.4 Reaction Dataset

ReactionDatasets are useful for chemical reaction-based computations. Currently, there are two types of *ReactionDatasets*:

- canonical chemical reaction datasets, `rxn`: $A + B \rightarrow C$,
- interaction energy datasets, `ie`: $M_{complex} \rightarrow M_{monomer_1} + M_{monomer_2}$.

3.4.1 Querying the Data

The result specifications in a *ReactionDataset* such as *method*, *basis set*, *program*, *keyword*, and *driver* may be listed with *list_values*. In addition to the aforementioned specifications, *ReactionDataset* provides a `stoich` field to select different strategies, including counterpoise-corrected (`"cp"`) and uncorrected (`"default"`), for the calculation of interaction and reaction energies.

The computed values of the reaction properties such as interaction or reaction energies can be queried using *get_values*. For results calculated with QCFractal, the underlying *Records* can be retrieved with *get_records* and further broken down by *Molecule* within each reaction.

For examples of querying *ReactionDatasets*, see the QCArchive examples.

3.4.2 Statistics and Visualization

Statistical analysis on the *ReactionDatasets* can be performed via *statistics* command which can be complemented by the *visualize* command in order to plot the data. For examples pertinent to data visualization in *ReactionDatasets* see the QCArchive examples.

3.4.3 Creating the Datasets

The *Dataset* constructor can be adopted to create an empty dataset object using a dataset name and type (`dtype`) as arguments.

```
>>> ds = ptl.collections.Dataset("my_dataset", dtype="rxn")
```

New reactions can be added to the *ReactionDatasets* by providing a linear combination of *Molecules* in order to compute the desired quantity. When the *ReactionDataset* is queried, these linear combinations from chemical equations are automatically combined and presented to the caller.

```
>>> ds = ptl.collections.Dataset("Atomization Energies", dtype="ie")

>>> N2 = ptl.Molecule.from_data("""
>>> N 0.0 0.0 1.0975
>>> N 0.0 0.0 0.0
>>> unit angstrom
>>> """)

>>> N_atom = ptl.Molecule.from_data("""
```

(continues on next page)

(continued from previous page)

```
>>> 0 2
>>> N 0.0 0.0 0.0
>>> """)


>>> ds.add_rxn("Nitrogen Molecule", [(N2, 1.0), (N_atom, -2.0)])
```

The details of a given chemical reaction can be obtained through using *get_rxn()* function. The storage of `molecule_hash` in the *ReactionDataset* is followed by that of the reaction coefficients.

```
>>> json.dumps(ds.get_rxn("Nitrogen Molecule"), indent=2)
{
  "name": "Nitrogen Molecule",
  "stoichiometry": {
    "default": {
      "1": 1.0,
      "2": -2.0
    }
  },
  "attributes": {},
  "reaction_results": {
    "default": {}
  }
}
```

Datasets of dtype `ie` can automatically construct counterpoise-corrected (`cp`) or non-counterpoise-corrected (`default`) n-body expansions. The the *key:value* pairs of numbers after the stoichiometry entry correspond to the index of each atomic/molecular species and their corresponding number of moles (or chemical equivalents) involved in the chemical reaction, respectively.

```
>>> ie_ds = ptl.collections.ReactionDataset("my_dataset", dtype="rxn")

>>> water_dimer_stretch = ptl.data.get_molecule("water_dimer_minima.psimol")

>>> ie_ds.add_ie_rxn("water dimer minima", water_dimer_stretch)

>>> json.dumps(ie_ds.get_rxn("water dimer minima"), indent=2)
{
  "name": "water dimer minima",
  "stoichiometry": {
    "default1": {  # Monomers
      "3": 1.0,
      "4": 1.0
    },
    "cp1": {  # Monomers
      "5": 1.0,
      "6": 1.0
    },
    "default": {  # Complex
      "7": 1.0
    },
    "cp": {  # Complex
```

(continues on next page)

(continued from previous page)

```
      "7": 1.0
    }
  },
  "attributes": {},
  "reaction_results": {
    "default": {}
  }
}
```

3.4.4 Computational Tasks

Computations on the `ReactionDatasets` are performed in the same manner as mentioned in *Dataset Documentation* section.

3.4.5 API

class qcportal.collections.**ReactionDataset**(*name: str*, *client: Optional[FractalClient] = None*, *ds_type: str = 'rxn'*, ***kwargs*)

The ReactionDataset class for homogeneous computations on many reactions.

Variables

- **client** (*client.FractalClient*) – A FractalClient connected to a server
- **data** (*ReactionDataset.DataModel*) – A Model representation of the database backbone
- **df** (*pd.DataFrame*) – The underlying dataframe for the Dataset object
- **rxn_index** (*pd.Index*) – The unrolled reaction index for all reactions in the Dataset

class DataModel(***, *id: str = 'local'*, *name: str*, *collection: str*, *provenance: Dict[str, str] = {}*, *tags: List[str] = []*, *tagline: str = None*, *description: str = None*, *group: str = 'default'*, *visibility: bool = True*, *view_url_hdf5: str = None*, *view_url_plaintext: str = None*, *view_metadata: Dict[str, str] = None*, *view_available: bool = False*, *metadata: Dict[str, Any] = {}*, *default_program: str = None*, *default_keywords: Dict[str, str] = {}*, *default_driver: str = 'energy'*, *default_units: str = 'kcal / mol'*, *default_benchmark: str = None*, *alias_keywords: Dict[str, Dict[str, str]] = {}*, *records: List[qcportal.collections.reaction_dataset.ReactionEntry] = None*, *contributed_values: Dict[str, qcportal.collections.dataset.ContributedValues] = None*, *history: Set[Tuple[str, str, str, Optional[str], Optional[str], str]] = {}*, *history_keys: Tuple[str, str, str, str, str, str] = ('driver', 'program', 'method', 'basis', 'keywords', 'stoichiometry')*, *ds_type: qcportal.collections.reaction_dataset._ReactionTypeEnum = _ReactionTypeEnum.rxn*)

Parameters

- **id** (*str, Default: local*)
- **name** (*str*)
- **collection** (*str*)
- **provenance** (*Dict[str], Default: {}*)
- **tags** (*List[str], Default: []*)
- **tagline** (*str, Optional*)

- **description** (*str, Optional*)
- **group** (*str, Default: default*)
- **visibility** (*bool, Default: True*)
- **view_url_hdf5** (*str, Optional*)
- **view_url_plaintext** (*str, Optional*)
- **view_metadata** (*Dict[str], Optional*)
- **view_available** (*bool, Default: False*)
- **metadata** (*Dict[Any], Default: {}*)
- **default_program** (*str, Optional*)
- **default_keywords** (*Dict[str], Default: {}*)
- **default_driver** (*str, Default: energy*)
- **default_units** (*str, Default: kcal / mol*)
- **default_benchmark** (*str, Optional*)
- **alias_keywords** (*Dict[Dict[str]], Default: {}*)
- **records** (`ReactionEntry`, Optional)
- **contributed_values** (`ContributedValues`, Optional)
- **history** (*Set[Tuple[str, str, str, str, str, str]], Default: set()*)
- **history_keys** (*Tuple[str, str, str, str, str, str], Default: ('driver', 'program', 'method', 'basis', 'keywords', 'stoichiometry')*)
- **ds_type** (*{rxn,ie}, Default: rxn*)

`compare`(*other: Union[qcelemental.models.basemodels.ProtoModel, pydantic.main.BaseModel]*, ***kwargs*) → bool

Compares the current object to the provided object recursively.

Parameters

- **other** (*Model*) – The model to compare to.
- ****kwargs** – Additional kwargs to pass to `qcelemental.compare_recursive`.

Returns True if the objects match.

Return type bool

`classmethod construct`(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model

Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

`copy`(***, *include: Union[AbstractSetIntStr, MappingIntStrAny] = None*, *exclude: Union[AbstractSetIntStr, MappingIntStrAny] = None*, *update: DictStrAny = None*, *deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns new model instance

dict(***kwargs*) → Dict[str, Any]

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

json(***kwargs*)

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

classmethod parse_file(*path: Union[str, pathlib.Path]*, ***, *encoding: Optional[str] = None*) → qcelemental.models.basemodels.ProtoModel

Parses a file into a Model object.

Parameters

- **path** (*Union[str, Path]*) – The path to the file.
- **encoding** (*str, optional*) – The type of the files, available types are: {'json', 'msgpack', 'pickle'}. Attempts to automatically infer the file type from the file extension if None.

Returns The requested model from a serialized format.

Return type Model

classmethod parse_raw(*data: Union[bytes, str]*, ***, *encoding: Optional[str] = None*) → qcelemental.models.basemodels.ProtoModel

Parses raw string or bytes into a Model object.

Parameters

- **data** (*Union[bytes, str]*) – A serialized data blob to be deserialized into a Model.
- **encoding** (*str, optional*) – The type of the serialized array, available types are: {'json', 'json-ext', 'msgpack-ext', 'pickle'}

Returns The requested model from a serialized format.

Return type Model

serialize(*encoding: str*, ***, *include: Optional[Set[str]] = None*, *exclude: Optional[Set[str]] = None*, *exclude_unset: Optional[bool] = None*, *exclude_defaults: Optional[bool] = None*, *exclude_none: Optional[bool] = None*) → Union[bytes, str]

Generates a serialized representation of the model

Parameters

- **encoding** (*str*) – The serialization type, available types are: {'json', 'json-ext', 'msgpack-ext'}
- **include** (*Optional[Set[str]], optional*) – Fields to be included in the serialization.
- **exclude** (*Optional[Set[str]], optional*) – Fields to be excluded in the serialization.
- **exclude_unset** (*Optional[bool], optional*) – If True, skips fields that have default values provided.
- **exclude_defaults** (*Optional[bool], optional*) – If True, skips fields that have set or defaulted values equal to the default.
- **exclude_none** (*Optional[bool], optional*) – If True, skips fields that have value `None`.

Returns The serialized model.

Return type Union[bytes, str]

classmethod update_forward_refs(***localns: Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

add_contributed_values(*contrib: qcportal.collections.dataset.ContributedValues*, *overwrite: bool = False*) → None

Adds a ContributedValues to the database. Be sure to call save() to commit changes to the server.

Parameters

- **contrib** (*ContributedValues*) – The ContributedValues to add.

- **overwrite** (*bool, optional*) – Overwrites pre-existing values

add_entry(*name: str*, *molecule: Molecule*, ***kwargs: Dict[str, Any]*) → None
Adds a new entry to the Dataset

Parameters

- **name** (*str*) – The name of the record
- **molecule** (*Molecule*) – The Molecule associated with this record
- ****kwargs** (*Dict[str, Any]*) – Additional arguments to pass to the record

add_ie_rxn(*name: str*, *mol:* qcelemental.models.molecule.Molecule, ***kwargs*) →
qcportal.collections.reaction_dataset.ReactionEntry
Add a interaction energy reaction entry to the database. Automatically builds CP and no-CP reactions for the fragmented molecule.

Parameters

- **name** (*str*) – The name of the reaction
- **mol** (*Molecule*) – A molecule with multiple fragments
- ****kwargs** – Additional kwargs to pass into *build_id_fragments*.

Returns A representation of the new reaction.

Return type ReactionEntry

add_keywords(*alias: str*, *program: str*, *keyword: KeywordSet*, *default: bool = False*) → bool
Adds an option alias to the dataset. Not that keywords are not present until a save call has been completed.

Parameters

- **alias** (*str*) – The alias of the option
- **program** (*str*) – The compute program the alias is for
- **keyword** (*KeywordSet*) – The Keywords object to use.
- **default** (*bool, optional*) – Sets this option as the default for the program

add_rxn(*name: str*, *stoichiometry: Dict[str, List[Tuple[*qcelemental.models.molecule.Molecule*, float]]]*,
reaction_results: Optional[Dict[str, str]] = None, *attributes: Optional[Dict[str, Union[int, float, str]]] = None*, *other_fields: Optional[Dict[str, Any]] = None*) →
qcportal.collections.reaction_dataset.ReactionEntry
Adds a reaction to a database object.

Parameters

- **name** (*str*) – Name of the reaction.
- **stoichiometry** (*list or dict*) – Either a list or dictionary of lists
- **reaction_results** (*dict or None, Optional, Default: None*) – A dictionary of the computed total interaction energy results
- **attributes** (*dict or None, Optional, Default: None*) – A dictionary of attributes to assign to the reaction
- **other_fields** (*dict or None, Optional, Default: None*) – A dictionary of additional user defined fields to add to the reaction entry

Returns A complete specification of the reaction

Return type ReactionEntry

static build_ie_fragments(*mol:* qcelemental.models.molecule.Molecule, ***kwargs*) → Dict[str, List[Tuple[*qcelemental.models.molecule.Molecule*, float]]]

Build the stoichiometry for an Interaction Energy.

Parameters

- **mol** (*Molecule class or str*) – Molecule to fragment.
- **do_default** (*bool*) – Create the default (noCP) stoichiometry.
- **do_cp** (*bool*) – Create the counterpoise (CP) corrected stoichiometry.
- **do_vmfc** (*bool*) – Create the Valiron-Mayer Function Counterpoise (VMFC) corrected stoichiometry.
- **max_nbody** (*int*) – The maximum fragment level built, if zero defaults to the maximum number of fragments.

Notes

Returns **ret** – A JSON representation of the fragmented molecule.

Return type dict

compute(*method: str*, *basis: Optional[str] = None*, ***, *keywords: Optional[str] = None*, *program: Optional[str] = None*, *stoich: str = 'default'*, *ignore_ds_type: bool = False*, *tag: Optional[str] = None*, *priority: Optional[str] = None*) → ComputeResponse

Executes a computational method for all reactions in the Dataset. Previously completed computations are not repeated.

Parameters

- **method** (*str*) – The computational method to compute (B3LYP)
- **basis** (*Optional[str], optional*) – The computational basis to compute (6-31G)
- **keywords** (*Optional[str], optional*) – The keyword alias for the requested compute
- **program** (*Optional[str], optional*) – The underlying QC program
- **stoich** (*str, optional*) – The stoichiometry of the requested compute (cp/nocp/etc)
- **ignore_ds_type** (*bool, optional*) – Optionally only compute the "default" geometry
- **tag** (*Optional[str], optional*) – The queue tag to use when submitting compute requests.
- **priority** (*Optional[str], optional*) – The priority of the jobs low, medium, or high.

Returns

An object that contains the submitted ObjectIds of the new compute. This object has the following fields:

- ids: The ObjectId's of the task in the order of input molecules
- submitted: A list of ObjectId's that were submitted to the compute queue
- existing: A list of ObjectId's of tasks already in the database

Return type ComputeResponse

download(*local_path: Optional[Union[str, pathlib.Path]] = None*, *verify: bool = True*, *progress_bar: bool = True*) → None

Download a remote view if available. The dataset will use this view to avoid server queries for calls to: - get_entries - get_molecules - get_values - list_values

Parameters

- **local_path** (*Optional[Union[str, Path]], optional*) – Local path the store downloaded view. If None, the view will be stored in a temporary file and deleted on exit.
- **verify** (*bool, optional*) – Verify download checksum. Default: True.
- **progress_bar** (*bool, optional*) – Display a download progress bar. Default: True

`classmethod from_json`(*data: Dict[str, Any]*, *client: FractalClient = None*) → Collection
Creates a new class from a JSON blob

Parameters

- **data** (*Dict[str, Any]*) – The JSON blob to create a new class from.
- **client** (*FractalClient, optional*) – A FractalClient connected to a server

Returns A constructed collection.

Return type Collection

`classmethod from_server`(*client: FractalClient*, *name: str*) → Collection
Creates a new class from a server

Parameters

- **client** (*FractalClient*) – A FractalClient connected to a server
- **name** (*str*) – The name of the collection to pull from.

Returns A constructed collection.

Return type Collection

`get_entries`(*subset: Optional[List[str]] = None*, *force: bool = False*) → pandas.core.frame.DataFrame
Provides a list of entries for the dataset

Parameters

- **subset** (*Optional[List[str]], optional*) – The indices of the desired subset. Return all indices if subset is None.
- **force** (*bool, optional*) – skip cache

Returns A dataframe containing entry names and specifciations. For Dataset, specifications are molecule ids. For ReactionDataset, specifications describe reaction stoichiometry.

Return type pd.DataFrame

`get_index`(*subset: Optional[List[str]] = None*, *force: bool = False*) → List[str]
Returns the current index of the database.

Returns **ret** – The names of all reactions in the database

Return type List[str]

`get_keywords`(*alias: str*, *program: str*, *return_id: bool = False*) → Union[KeywordSet, str]
Pulls the keywords alias from the server for inspection.

Parameters

- **alias** (*str*) – The keywords alias.
- **program** (*str*) – The program the keywords correspond to.
- **return_id** (*bool, optional*) – If True, returns the `id` rather than the `KeywordSet` object. Description

Returns The requested `KeywordSet` or `KeywordSet id`.

Return type Union['KeywordSet', str]

get_molecules(*subset: Optional[Union[str, Set[str]]] = None*, *stoich: Union[str, List[str]] = 'default'*, *force: bool = False*) → pandas.core.frame.DataFrame

Queries full Molecules from the database.

Parameters

- **subset** (*Optional[Union[str, Set[str]]], optional*) – The index subset to query on
- **stoich** (*Union[str, List[str]], optional*) – The stoichiometries to pull from, either a single or multiple stoichiometries
- **force** (*bool, optional*) – Force pull of molecules from server

Returns Indexed Molecules which match the stoich and subset string.

Return type pd.DataFrame

get_records(*method: str*, *basis: Optional[str] = None*, ***, *keywords: Optional[str] = None*, *program: Optional[str] = None*, *stoich: Union[str, List[str]] = 'default'*, *include: Optional[List[str]] = None*, *subset: Optional[Union[str, Set[str]]] = None*) → Union[pandas.core.frame.DataFrame, ResultRecord]

Queries the local Portal for the requested keys and stoichiometry.

Parameters

- **method** (*str*) – The computational method to query on (B3LYP)
- **basis** (*Optional[str], optional*) – The computational basis to query on (6-31G)
- **keywords** (*Optional[str], optional*) – The option token desired
- **program** (*Optional[str], optional*) – The program to query on
- **stoich** (*Union[str, List[str]], optional*) – The given stoichiometry to compute.
- **include** (*Optional[Dict[str, bool]], optional*) – The attribute project to perform on the query, otherwise returns ResultRecord objects.
- **subset** (*Optional[Union[str, Set[str]]], optional*) – The index subset to query on

Returns The name of the queried column

Return type Union[pd.DataFrame, 'ResultRecord']

get_rxn(*name: str*) → qcportal.collections.reaction_dataset.ReactionEntry

Returns the JSON object of a specific reaction.

Parameters **name** (*str*) – The name of the reaction to query

Returns **ret** – The JSON representation of the reaction

Return type dict

get_values(*method: Optional[Union[List[str], str]] = None*, *basis: Optional[Union[List[str], str]] = None*, *keywords: Optional[str] = None*, *program: Optional[str] = None*, *driver: Optional[str] = None*, *stoich: str = 'default'*, *name: Optional[Union[List[str], str]] = None*, *native: Optional[bool] = None*, *subset: Optional[Union[List[str], str]] = None*, *force: bool = False*) → pandas.core.frame.DataFrame

Obtains values from the known history from the search paramaters provided for the expected *return_result* values. Defaults to the standard programs and keywords if not provided.

Note that unlike *get_records*, *get_values* will automatically expand searches and return multiple method and basis combinations simultaneously.

None is a wildcard selector. To search for *None*, use *"None"*.

Parameters

- **method** (*Optional[Union[str, List[str]]], optional*) – The computational method (B3LYP)
- **basis** (*Optional[Union[str, List[str]]], optional*) – The computational basis (6-31G)
- **keywords** (*Optional[str], optional*) – The keyword alias
- **program** (*Optional[str], optional*) – The underlying QC program
- **driver** (*Optional[str], optional*) – The type of calculation (e.g. energy, gradient, hessian, dipole…)
- **stoich** (*str, optional*) – Stoichiometry of the reaction.
- **name** (*Optional[Union[str, List[str]]], optional*) – Canonical name of the record. Overrides the above selectors.
- **native** (*Optional[bool], optional*) – True: only include data computed with QCFractal False: only include data contributed from outside sources None: include both
- **subset** (*Optional[List[str]], optional*) – The indices of the desired subset. Return all indices if subset is None.
- **force** (*bool, optional*) – Data is typically cached, forces a new query if True

Returns A DataFrame of values with columns corresponding to methods and rows corresponding to reaction entries. Contributed (native=False) columns are marked with "(contributed)" and may include units in square brackets if their units differ in dimensionality from the ReactionDataset's default units.

Return type DataFrame

`list_keywords()` → pandas.core.frame.DataFrame

Lists keyword aliases for each program in the dataset.

Returns A dataframe containing programs, keyword aliases, KeywordSet ids, and whether those keywords are the default for a program. Indexed on program.

Return type pd.DataFrame

`list_records`(*dftd3: bool = False*, *pretty: bool = True*, ***search: Optional[Union[str, List[str]]]*) → pandas.core.frame.DataFrame

Lists specifications of available records, i.e. method, program, basis set, keyword set, driver combinations *None* is a wildcard selector. To search for *None*, use *"None"*.

Parameters

- **pretty** (*bool*) – Replace NaN with "None" in returned DataFrame
- ****search** (*Dict[str, Optional[str]]*) – Allows searching to narrow down return.

Returns Record specifications matching **search.

Return type DataFrame

list_values(*method: Optional[Union[List[str], str]] = None*, *basis: Optional[Union[List[str], str]] = None*, *keywords: Optional[str] = None*, *program: Optional[str] = None*, *driver: Optional[str] = None*, *name: Optional[Union[List[str], str]] = None*, *native: Optional[bool] = None*, *force: bool = False*) → pandas.core.frame.DataFrame

Lists available data that may be queried with get_values. Results may be narrowed by providing search keys. *None* is a wildcard selector. To search for *None*, use *"None"*.

Parameters

- **method** (*Optional[Union[str, List[str]]], optional*) – The computational method (B3LYP)
- **basis** (*Optional[Union[str, List[str]]], optional*) – The computational basis (6-31G)
- **keywords** (*Optional[str], optional*) – The keyword alias
- **program** (*Optional[str], optional*) – The underlying QC program
- **driver** (*Optional[str], optional*) – The type of calculation (e.g. energy, gradient, hessian, dipole…)
- **name** (*Optional[Union[str, List[str]]], optional*) – The canonical name of the data column
- **native** (*Optional[bool], optional*) – True: only include data computed with QCFractal False: only include data contributed from outside sources None: include both
- **force** (*bool, optional*) – Data is typically cached, forces a new query if True

Returns A DataFrame of the matching data specifications

Return type DataFrame

parse_stoichiometry(*stoichiometry: List[Tuple[Union[*qcelemental.models.molecule.Molecule*, str], float]]*) → Dict[str, float]

Parses a stiochiometry list.

Parameters **stoichiometry** (*list*) – A list of tuples describing the stoichiometry.

Returns A dictionary describing the stoichiometry for use in the database. Keys are molecule hashes. Values are stoichiometric coefficients.

Return type Dict[str, float]

Notes

This function attempts to convert the molecule into its corresponding hash. The following will happen depending on t

- Molecule hash - Used directly in the stoichiometry.
- Molecule class - Hash is obtained and the molecule will be added to the database upon saving.
- Molecule string - Molecule will be converted to a Molecule class and the same process as the above will occur.

save(*client: Optional[FractalClient] = None*) → ObjectId

Uploads the overall structure of the Collection (indices, options, new molecules, etc) to the server.

Parameters **client** (*FractalClient, optional*) – A FractalClient connected to a server to upload to

Returns The ObjectId of the saved collection.

Return type ObjectId

`set_default_benchmark`(*benchmark: str*) → bool
Sets the default benchmark value.

Parameters **benchmark** (*str*) – The benchmark to default to.

`set_default_program`(*program: str*) → bool
Sets the default program.

Parameters **program** (*str*) – The program to default to.

`set_view`(*path: Union[str, pathlib.Path]*) → None
Set a dataset to use a local view.

Parameters **path** (*Union[str, Path]*) – path to an hdf5 file representing a view for this dataset

`statistics`(*stype: str*, *value: str*, *bench: Optional[str] = None*, ***kwargs: Dict[str, Any]*) → Union[numpy.ndarray, pandas.core.series.Series, numpy.float64]
Provides statistics for various columns in the underlying dataframe.

Parameters

- **stype** (*str*) – The type of statistic in question
- **value** (*str*) – The method string to compare
- **bench** (*str, optional*) – The benchmark method for the comparison, defaults to *default_benchmark*.
- **kwargs** (*Dict[str, Any]*) – Additional kwargs to pass to the statistics functions

Returns Returns an ndarray, Series, or float with the requested statistics depending on input.

Return type np.ndarray, pd.Series, float

`ternary`(*cvals=None*)
Plots a ternary diagram of the DataBase if available

Parameters **cvals** (*None, optional*) – Description

`to_file`(*path: Union[str, pathlib.Path]*, *encoding: str*) → None
Writes a view of the dataset to a file

Parameters

- **path** (*Union[str, Path]*) – Where to write the file
- **encoding** (*str*) – Options: plaintext, hdf5

`to_json`(*filename: Optional[str] = None*)
If a filename is provided, dumps the file to disk. Otherwise returns a copy of the current data.

Parameters **filename** (*str, Optional, Default: None*) – The filename to drop the data to.

Returns **ret** – A JSON representation of the Collection

Return type dict

`visualize`(*method: Optional[str] = None*, *basis: Optional[str] = None*, *keywords: Optional[str] = None*, *program: Optional[str] = None*, *stoich: str = 'default'*, *groupby: Optional[str] = None*, *metric: str = 'UE'*, *bench: Optional[str] = None*, *kind: str = 'bar'*, *return_figure: Optional[bool] = None*, *show_incomplete: bool = False*) → plotly.Figure

Parameters

- **method** (*Optional[str], optional*) – Methods to query

- **basis** (*Optional[str], optional*) – Bases to query
- **keywords** (*Optional[str], optional*) – Keyword aliases to query
- **program** (*Optional[str], optional*) – Programs aliases to query
- **stoich** (*str, optional*) – Stoichiometry to query
- **groupby** (*Optional[str], optional*) – Groups the plot by this index.
- **metric** (*str, optional*) – The metric to use either UE (unsigned error) or URE (unsigned relative error)
- **bench** (*Optional[str], optional*) – The benchmark level of theory to use
- **kind** (*str, optional*) – The kind of chart to produce, either 'bar' or 'violin'
- **return_figure** (*Optional[bool], optional*) – If True, return the raw plotly figure. If False, returns a hosted iPlot. If None, return a iPlot display in Jupyter notebook and a raw plotly figure in all other circumstances.
- **show_incomplete** (*bool, optional*) – Display statistics method/basis set combinations where results are incomplete

Returns The requested figure.

Return type plotly.Figure

3.5 Optimization Dataset

The *OptimizationDataset* collection represents the results of geometry optimization calculations performed on a series of *Molecules*. The *OptimizationDataset* uses metadata specifications via *Optimization Specification* and *QCSpecification* classes to manage parameters of the geometry optimizer and the underlying gradient calculation, respectively.

The existing *OptimizationDataset* collections can be listed or selectively returned through *FractalClient.list_collections("OptimizationDataset")* and *FractalClient.get_collection("OptimizationDataset", name)*, respectively.

3.5.1 Querying the Data

All available optimization data specifications can be listed via

```
>>> ds.list_specifications()
```

function. In order to show the status of the optimization calculations for a given set of specifications, one can use:

```
>>> ds.status(["default"])
```

For each *Molecule*, the number of steps in a geometry optimization procedure can be queried through calling:

```
>>> ds.counts()
```

function. Individual *OptimizationRecords* can be obtained using:

```
>>> ds.get_record(name="CCO-0", specification="default")
```

3.5.2 Statistics and Visualization

The trajectory of energy change during the course of geometry optimization can be plotted by adopting *qcportal.models.OptimizationRecord.show_history()* function.

3.5.3 Creating the Datasets

A new collection object for *OptimizationDataset* can be created using

```
>>> ds = ptl.collections.OptimizationDataset(name = "QM8-T", client=client)
```

Specific set of parameters for geometry optimization can be defined and added to the dataset as follows:

```
>>> spec = {'name': 'default',
>>>         'description': 'Geometric + Psi4/B3LYP-D3/Def2-SVP.',
>>>         'optimization_spec': {'program': 'geometric', 'keywords': None},
>>>         'qc_spec': {'driver': 'gradient',
>>>         'method': 'b3lyp-d3',
>>>         'basis': 'def2-svp',
>>>         'keywords': None,
>>>         'program': 'psi4'}}

>>>  ds.add_specification(**spec)

>>>  ds.save()
```

Molecules can be added to the *OptimizationDataset* as new entries for optimization via:

```
ds.add_entry(name, molecule)
```

When adding multiple entries of molecules, saving the dataset onto the server should be postponed until after all molecules are added:

```
>>> for name, molecule in new_entries:
>>>     ds.add_entry(name, molecule, save=False)

>>> ds.save()
```

3.5.4 Computational Tasks

In order to run a geometry optimization calculation based on a particular set of parameters (the default set in this case), one can adopt the

```
>>> ds.compute(specification="default", tag="optional_tag")
```

function from *OptimizationDataset* class.

3.5.5 API

class qcportal.collections.**OptimizationDataset**(*name: str*, *client: FractalClient = None*, ***kwargs*)

class **DataModel**(***, *id: str = 'local'*, *name: str*, *collection: str*, *provenance: Dict[str, str] = {}*, *tags: List[str] = []*, *tagline: str = None*, *description: str = None*, *group: str = 'default'*, *visibility: bool = True*, *view_url_hdf5: str = None*, *view_url_plaintext: str = None*, *view_metadata: Dict[str, str] = None*, *view_available: bool = False*, *metadata: Dict[str, Any] = {}*, *records: Dict[str, qcportal.collections.optimization_dataset.OptEntry] = {}*, *history: Set[str] = {}*, *specs: Dict[str, qcportal.collections.optimization_dataset.OptEntrySpecification] = {}*)

Parameters

- **id** (*str, Default: local*)
- **name** (*str*)
- **collection** (*str*)
- **provenance** (*Dict[str], Default: {}*)
- **tags** (*List[str], Default: []*)
- **tagline** (*str, Optional*)
- **description** (*str, Optional*)
- **group** (*str, Default: default*)
- **visibility** (*bool, Default: True*)
- **view_url_hdf5** (*str, Optional*)
- **view_url_plaintext** (*str, Optional*)
- **view_metadata** (*Dict[str], Optional*)
- **view_available** (*bool, Default: False*)
- **metadata** (*Dict[Any], Default: {}*)
- **records** (`OptEntry`, Default: {})
- **history** (*Set[str], Default: set()*)
- **specs** (`OptEntrySpecification`, Default: {})

compare(*other: Union[qcelemental.models.basemodels.ProtoModel, pydantic.main.BaseModel]*, ***kwargs*) → bool

Compares the current object to the provided object recursively.

Parameters

- **other** (*Model*) – The model to compare to.
- ****kwargs** – Additional kwargs to pass to `qcelemental.compare_recursive`.

Returns True if the objects match.

Return type bool

classmethod **construct**(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model

Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(*, *include: Union[AbstractSetIntStr, MappingIntStrAny] = None*, *exclude: Union[AbstractSetIntStr, MappingIntStrAny] = None*, *update: DictStrAny = None*, *deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns new model instance

dict(***kwargs*) → Dict[str, Any]

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

json(***kwargs*)

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

classmethod parse_file(*path: Union[str, pathlib.Path]*, *, *encoding: Optional[str] = None*) → qcelemental.models.basemodels.ProtoModel

Parses a file into a Model object.

Parameters

- **path** (*Union[str, Path]*) – The path to the file.
- **encoding** (*str, optional*) – The type of the files, available types are: {'json', 'msgpack', 'pickle'}. Attempts to automatically infer the file type from the file extension if None.

Returns The requested model from a serialized format.

Return type Model

classmethod parse_raw(*data: Union[bytes, str]*, *, *encoding: Optional[str] = None*) → qcelemental.models.basemodels.ProtoModel

Parses raw string or bytes into a Model object.

Parameters

- **data** (*Union[bytes, str]*) – A serialized data blob to be deserialized into a Model.
- **encoding** (*str, optional*) – The type of the serialized array, available types are: {'json', 'json-ext', 'msgpack-ext', 'pickle'}

Returns The requested model from a serialized format.

Return type Model

serialize(*encoding: str*, *, *include: Optional[Set[str]] = None*, *exclude: Optional[Set[str]] = None*, *exclude_unset: Optional[bool] = None*, *exclude_defaults: Optional[bool] = None*, *exclude_none: Optional[bool] = None*) → Union[bytes, str]

Generates a serialized representation of the model

Parameters

- **encoding** (*str*) – The serialization type, available types are: {'json', 'json-ext', 'msgpack-ext'}
- **include** (*Optional[Set[str]], optional*) – Fields to be included in the serialization.
- **exclude** (*Optional[Set[str]], optional*) – Fields to be excluded in the serialization.
- **exclude_unset** (*Optional[bool], optional*) – If True, skips fields that have default values provided.
- **exclude_defaults** (*Optional[bool], optional*) – If True, skips fields that have set or defaulted values equal to the default.
- **exclude_none** (*Optional[bool], optional*) – If True, skips fields that have value `None`.

Returns The serialized model.
Return type Union[bytes, str]

classmethod update_forward_refs(***localns: Any*) → None
Try to update ForwardRefs on fields based on this Model, globalns and localns.

add_entry(*name: str*, *initial_molecule: Molecule*, *additional_keywords: Optional[Dict[str, Any]] = None*, *attributes: Optional[Dict[str, Any]] = None*, *save: bool = True*) → None

Parameters

- **name** (*str*) – The name of the entry, will be used for the index
- **initial_molecule** (*Molecule*) – The list of starting Molecules for the Optimization
- **additional_keywords** (*Dict[str, Any], optional*) – Additional keywords to add to the optimization run
- **attributes** (*Dict[str, Any], optional*) – Additional attributes and descriptions for the entry
- **save** (*bool, optional*) – If true, saves the collection after adding the entry. If this is False be careful to call save after all entries are added, otherwise data pointers may be lost.

add_specification(*name: str*, *optimization_spec:* qcportal.models.common_models.OptimizationSpecification, *qc_spec:* qcportal.models.common_models.QCSpecification, *description: Optional[str] = None*, *protocols: Optional[Dict[str, Any]] = None*, *overwrite=False*) → None

Parameters

- **name** (*str*) – The name of the specification
- **optimization_spec** (*OptimizationSpecification*) – A full optimization specification for Optimization
- **qc_spec** (*QCSpecification*) – A full quantum chemistry specification for Optimization
- **description** (*str, optional*) – A short text description of the specification
- **protocols** (*Optional[Dict[str, Any]], optional*) – Protocols for this specification.
- **overwrite** (*bool, optional*) – Overwrite existing specification names

compute(*specification: str*, *subset: Optional[Set[str]] = None*, *tag: Optional[str] = None*, *priority: Optional[str] = None*) → int
Computes a specification for all entries in the dataset.

Parameters

- **specification** (*str*) – The specification name.
- **subset** (*Set[str], optional*) – Computes only a subset of the dataset.
- **tag** (*Optional[str], optional*) – The queue tag to use when submitting compute requests.
- **priority** (*Optional[str], optional*) – The priority of the jobs low, medium, or high.

Returns The number of submitted computations

Return type int

counts(*entries: Optional[Union[List[str], str]] = None*, *specs: Optional[Union[List[str], str]] = None*) → pandas.core.frame.DataFrame
Counts the number of optimization or gradient evaluations associated with the Optimizations.

Parameters

- **entries** (*Union[str, List[str]]*) – The entries to query for
- **specs** (*Optional[Union[str, List[str]]], optional*) – The specifications to query for
- **count_gradients** (*bool, optional*) – If True, counts the total number of gradient calls. Warning! This can be slow for large datasets.

Returns The queried counts.

Return type DataFrame

`classmethod from_json`(*data: Dict[str, Any]*, *client: FractalClient = None*) → Collection
Creates a new class from a JSON blob

Parameters

- **data** (*Dict[str, Any]*) – The JSON blob to create a new class from.
- **client** (*FractalClient, optional*) – A FractalClient connected to a server

Returns A constructed collection.

Return type Collection

`classmethod from_server`(*client: FractalClient*, *name: str*) → Collection
Creates a new class from a server

Parameters

- **client** (*FractalClient*) – A FractalClient connected to a server
- **name** (*str*) – The name of the collection to pull from.

Returns A constructed collection.

Return type Collection

`get_entry`(*name: str*) → Any
Obtains a record from the Dataset

Parameters **name** (*str*) – The record name to pull from.

Returns The requested record

Return type Record

`get_record`(*name: str*, *specification: str*) → Any
Pulls an individual computational record of the requested name and column.

Parameters

- **name** (*str*) – The index name to pull the record of.
- **specification** (*str*) – The name of specification to pull the record of.

Returns The requested Record

Return type Any

`get_specification`(*name: str*) → Any

Parameters **name** (*str*) – The name of the specification

Returns The requested specification.

Return type Specification

`list_specifications`(*description=True*) → Union[List[str], pandas.core.frame.DataFrame]
Lists all available specifications

Parameters **description** (*bool, optional*) – If True returns a DataFrame with Description

Returns A list of known specification names.

Return type Union[List[str], 'DataFrame']

`query`(*specification: str*, *force: bool = False*) → pandas.core.series.Series
Queries a given specification from the server

Parameters

- **specification** (*str*) – The specification name to query
- **force** (*bool, optional*) – Force a fresh query if the specification already exists.

Returns Records collected from the server

Return type pd.Series

`save`(*client: Optional[FractalClient] = None*) → ObjectId
Uploads the overall structure of the Collection (indices, options, new molecules, etc) to the server.

Parameters **client** (*FractalClient, optional*) – A FractalClient connected to a server to upload to

Returns The ObjectId of the saved collection.

Return type ObjectId

`status`(*specs: Optional[Union[str, List[str]]] = None*, *collapse: bool = True*, *status: Optional[str] = None*, *detail: bool = False*) → pandas.core.frame.DataFrame
Returns the status of all current specifications.

Parameters

- **collapse** (*bool, optional*) – Collapse the status into summaries per specification or not.
- **status** (*Optional[str], optional*) – If not None, only returns results that match the provided status.
- **detail** (*bool, optional*) – Shows a detailed description of the current status of incomplete jobs.

Returns A DataFrame of all known statuses

Return type DataFrame

`to_json`(*filename: Optional[str] = None*)
If a filename is provided, dumps the file to disk. Otherwise returns a copy of the current data.

Parameters **filename** (*str, Optional, Default: None*) – The filename to drop the data to.

Returns **ret** – A JSON representation of the Collection

Return type dict

3.6 TorsionDrive Dataset

TorsionDriveDatasets host the results of TorsionDrive computations. Each row of the *TorsionDriveDataset* is comprised of an `Entry` which contains a list of starting molecules for the `TorsionDrive`, a specific set of dihedral angles (zero-indexed), and the angular scan resolution. Each column represents a particular property detail pertinent to the corresponding `TorsionDrive` `Entry`. The *TorsionDriveDataset* is a procedure-style dataset within which, the *ObjectId* for each `TorsionDrive` computation is recorded as metadata.

For additional details about *TorsionDriveDatasets* see here.

3.6.1 Querying the Data

3.6.2 Statistics and Visualization

3.6.3 Creating the Datasets

A empty instance of the *TorsionDriveDataset* object (here, named `My Torsions`) can be constructed as

```
>>> client = ptl.FractalClient("localhost:7777")
>>> ds = ptl.collections.TorsionDriveDataset("My Torsions")
```

3.6.4 Computational Tasks

3.6.5 API

class qcfractal.interface.collections.**TorsionDriveDataset**(*name: str*, *client: FractalClient = None*, ***kwargs*)

class DataModel(***, *id: str = 'local'*, *name: str*, *collection: str*, *provenance: Dict[str, str] = {}*, *tags: List[str] = []*, *tagline: str = None*, *description: str = None*, *group: str = 'default'*, *visibility: bool = True*, *view_url_hdf5: str = None*, *view_url_plaintext: str = None*, *view_metadata: Dict[str, str] = None*, *view_available: bool = False*, *metadata: Dict[str, Any] = {}*, *records: Dict[str, qcfractal.interface.collections.torsiondrive_dataset.TDEntry] = {}*, *history: Set[str] = {}*, *specs: Dict[str, qcfractal.interface.collections.torsiondrive_dataset.TDEntrySpecification] = {}*)

Parameters

- **id** (*str, Default: local*)
- **name** (*str*)
- **collection** (*str*)
- **provenance** (*Dict[str], Default: {}*)
- **tags** (*List[str], Default: []*)
- **tagline** (*str, Optional*)
- **description** (*str, Optional*)
- **group** (*str, Default: default*)

- **visibility** (*bool, Default: True*)
- **view_url_hdf5** (*str, Optional*)
- **view_url_plaintext** (*str, Optional*)
- **view_metadata** (*Dict[str], Optional*)
- **view_available** (*bool, Default: False*)
- **metadata** (*Dict[Any], Default: {}*)
- **records** (`TDEntry`, Default: {})
- **history** (*Set[str], Default: set()*)
- **specs** (`TDEntrySpecification`, Default: {})

compare(*other: Union[qcelemental.models.basemodels.ProtoModel, pydantic.main.BaseModel]*, ***kwargs*) → bool

Compares the current object to the provided object recursively.

Parameters
- **other** (*Model*) – The model to compare to.
- ****kwargs** – Additional kwargs to pass to `qcelemental.compare_recursive`.

Returns True if the objects match.

Return type bool

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model

Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(***, *include: Union[AbstractSetIntStr, MappingIntStrAny] = None*, *exclude: Union[AbstractSetIntStr, MappingIntStrAny] = None*, *update: DictStrAny = None*, *deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters
- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns new model instance

dict(***kwargs*) → Dict[str, Any]

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

json(***kwargs*)

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

classmethod parse_file(*path: Union[str, pathlib.Path]*, ***, *encoding: Optional[str] = None*) → qcelemental.models.basemodels.ProtoModel

Parses a file into a Model object.

Parameters
- **path** (*Union[str, Path]*) – The path to the file.

- **encoding** (*str, optional*) – The type of the files, available types are: {'json', 'msgpack', 'pickle'}. Attempts to automatically infer the file type from the file extension if None.

Returns The requested model from a serialized format.

Return type Model

`classmethod parse_raw`(*data: Union[bytes, str]*, *, *encoding: Optional[str] = None*) → qcelemental.models.basemodels.ProtoModel

Parses raw string or bytes into a Model object.

Parameters

- **data** (*Union[bytes, str]*) – A serialized data blob to be deserialized into a Model.
- **encoding** (*str, optional*) – The type of the serialized array, available types are: {'json', 'json-ext', 'msgpack-ext', 'pickle'}

Returns The requested model from a serialized format.

Return type Model

`serialize`(*encoding: str*, *, *include: Optional[Set[str]] = None*, *exclude: Optional[Set[str]] = None*, *exclude_unset: Optional[bool] = None*, *exclude_defaults: Optional[bool] = None*, *exclude_none: Optional[bool] = None*) → Union[bytes, str]

Generates a serialized representation of the model

Parameters

- **encoding** (*str*) – The serialization type, available types are: {'json', 'json-ext', 'msgpack-ext'}
- **include** (*Optional[Set[str]], optional*) – Fields to be included in the serialization.
- **exclude** (*Optional[Set[str]], optional*) – Fields to be excluded in the serialization.
- **exclude_unset** (*Optional[bool], optional*) – If True, skips fields that have default values provided.
- **exclude_defaults** (*Optional[bool], optional*) – If True, skips fields that have set or defaulted values equal to the default.
- **exclude_none** (*Optional[bool], optional*) – If True, skips fields that have value `None`.

Returns The serialized model.

Return type Union[bytes, str]

`classmethod update_forward_refs`(***localns: Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

`add_entry`(*name: str*, *initial_molecules: List[Molecule]*, *dihedrals: List[Tuple[int, int, int, int]]*, *grid_spacing: List[int]*, *dihedral_ranges: Optional[List[Tuple[int, int]]] = None*, *energy_decrease_thresh: Optional[float] = None*, *energy_upper_limit: Optional[float] = None*, *attributes: Dict[str, Any] = None*, *save: bool = True*) → None

Parameters

- **name** (*str*) – The name of the entry, will be used for the index
- **initial_molecules** (*List[Molecule]*) – The list of starting Molecules for the TorsionDrive
- **dihedrals** (*List[Tuple[int, int, int, int]]*) – A list of dihedrals to scan over
- **grid_spacing** (*List[int]*) – The grid spacing for each dihedrals
- **dihedral_ranges** (*Optional[List[Tuple[int, int]]]*) – The range limit of each dihedrals to scan, within [-180, 360]
- **energy_decrease_thresh** (*Optional[float]*) – The threshold of energy decrease to trigger activating grid points
- **energy_upper_limit** (*Optional[float]*) – The upper limit of energy relative to current global minimum to trigger activating grid points

- **attributes** (*Dict[str, Any], optional*) – Additional attributes and descriptions for the entry
- **save** (*bool, optional*) – If true, saves the collection after adding the entry. If this is False be careful to call save after all entries are added, otherwise data pointers may be lost.

`add_specification`(*name: str*, *optimization_spec: qcfractal.interface.models.common_models.OptimizationSpecification*, *qc_spec: qcfractal.interface.models.common_models.QCSpecification*, *description: Optional[str] = None*, *overwrite: bool = False*) → None

Parameters

- **name** (*str*) – The name of the specification
- **optimization_spec** (*OptimizationSpecification*) – A full optimization specification for TorsionDrive
- **qc_spec** (*QCSpecification*) – A full quantum chemistry specification for TorsionDrive
- **description** (*str, optional*) – A short text description of the specification
- **overwrite** (*bool, optional*) – Overwrite existing specification names

`compute`(*specification: str*, *subset: Optional[Set[str]] = None*, *tag: Optional[str] = None*, *priority: Optional[str] = None*) → int

Computes a specification for all entries in the dataset.

Parameters

- **specification** (*str*) – The specification name.
- **subset** (*Set[str], optional*) – Computes only a subset of the dataset.
- **tag** (*Optional[str], optional*) – The queue tag to use when submitting compute requests.
- **priority** (*Optional[str], optional*) – The priority of the jobs low, medium, or high.

Returns The number of submitted computations

Return type int

`counts`(*entries: Union[str, List[str]]*, *specs: Optional[Union[List[str], str]] = None*, *count_gradients: bool = False*) → pandas.core.frame.DataFrame

Counts the number of optimization or gradient evaluations associated with the TorsionDrives.

Parameters

- **entries** (*Union[str, List[str]]*) – The entries to query for
- **specs** (*Optional[Union[str, List[str]]], optional*) – The specifications to query for
- **count_gradients** (*bool, optional*) – If True, counts the total number of gradient calls. Warning! This can be slow for large datasets.

Returns The queried counts.

Return type DataFrame

`classmethod from_json`(*data: Dict[str, Any]*, *client: FractalClient = None*) → Collection

Creates a new class from a JSON blob

Parameters

- **data** (*Dict[str, Any]*) – The JSON blob to create a new class from.
- **client** (*FractalClient, optional*) – A FractalClient connected to a server

Returns A constructed collection.

Return type Collection

`classmethod from_server`(*client: FractalClient*, *name: str*) → Collection
Creates a new class from a server

Parameters

- **client** (*FractalClient*) – A FractalClient connected to a server
- **name** (*str*) – The name of the collection to pull from.

Returns A constructed collection.

Return type Collection

`get_entry`(*name: str*) → Any
Obtains a record from the Dataset

Parameters **name** (*str*) – The record name to pull from.

Returns The requested record

Return type Record

`get_record`(*name: str*, *specification: str*) → Any
Pulls an individual computational record of the requested name and column.

Parameters

- **name** (*str*) – The index name to pull the record of.
- **specification** (*str*) – The name of specification to pull the record of.

Returns The requested Record

Return type Any

`get_specification`(*name: str*) → Any

Parameters **name** (*str*) – The name of the specification

Returns The requested specification.

Return type Specification

`list_specifications`(*description=True*) → Union[List[str], pandas.core.frame.DataFrame]
Lists all available specifications

Parameters **description** (*bool, optional*) – If True returns a DataFrame with Description

Returns A list of known specification names.

Return type Union[List[str], 'DataFrame']

`query`(*specification: str*, *force: bool = False*) → pandas.core.series.Series
Queries a given specification from the server

Parameters

- **specification** (*str*) – The specification name to query
- **force** (*bool, optional*) – Force a fresh query if the specification already exists.

Returns Records collected from the server

Return type pd.Series

save(*client: Optional[FractalClient] = None*) → ObjectId
Uploads the overall structure of the Collection (indices, options, new molecules, etc) to the server.

Parameters **client** (*FractalClient, optional*) – A FractalClient connected to a server to upload to

Returns The ObjectId of the saved collection.

Return type ObjectId

status(*specs: Optional[Union[str, List[str]]] = None*, *collapse: bool = True*, *status: Optional[str] = None*, *detail: bool = False*) → pandas.core.frame.DataFrame
Returns the status of all current specifications.

Parameters

- **collapse** (*bool, optional*) – Collapse the status into summaries per specification or not.
- **status** (*Optional[str], optional*) – If not None, only returns results that match the provided status.
- **detail** (*bool, optional*) – Shows a detailed description of the current status of incomplete jobs.

Returns A DataFrame of all known statuses

Return type DataFrame

to_json(*filename: Optional[str] = None*)
If a filename is provided, dumps the file to disk. Otherwise returns a copy of the current data.

Parameters **filename** (*str, Optional, Default: None*) – The filename to drop the data to.

Returns **ret** – A JSON representation of the Collection

Return type dict

visualize(*entries: Union[str, List[str]]*, *specs: Union[str, List[str]]*, *relative: bool = True*, *units: str = 'kcal / mol'*, *digits: int = 3*, *use_measured_angle: bool = False*, *return_figure: Optional[bool] = None*) → plotly.Figure

Parameters

- **entries** (*Union[str, List[str]]*) – A single or list of indices to plot.
- **specs** (*Union[str, List[str]]*) – A single or list of specifications to plot.
- **relative** (*bool, optional*) – Shows relative energy, lowest energy per scan is zero.
- **units** (*str, optional*) – The units of the plot.
- **digits** (*int, optional*) – Rounds the energies to n decimal places for display.
- **use_measured_angle** (*bool, optional*) – If True, the measured final angle instead of the constrained optimization angle. Can provide more accurate results if the optimization was ill-behaved, but pulls additional data from the server and may take longer.
- **return_figure** (*Optional[bool], optional*) – If True, return the raw plotly figure. If False, returns a hosted iPlot. If None, return a iPlot display in Jupyter notebook and a raw plotly figure in all other circumstances.

Returns The requested figure.

Return type plotly.Figure

3.7 Common Tasks

3.7.1 Check computation progress

The *FractalClient.query_tasks()* method returns information on active tasks.

```
>>> client = qcportal.FractalClient(...)
>>> client.query_tasks(status='WAITING')  # return tasks in queue
>>> client.query_tasks(status='RUNNING')  # return running tasks
```

The `tag` and `manager` fields can be used to find user-defined tasks.

Finding Errors and Restarting Jobs

During each task's lifetime, errors might occur and the jobs may end up in the `ERROR` state.

```
>>> myerrs = client.query_tasks(status='ERROR')  # return errored tasks
```

Failed jobs may be inspected as follows:

```
>>> record = client.query_results(mye[0].base_result.id)[0]
>>> print(record.stdout)  # Standard output
>>> print(record.stderr)  # Standard error
>>> print(client.query_kvstore(record.error)[0])  # Error message
```

One can restart the failed tasks via the following steps:

```
>>> res = client.modify_tasks("restart", [e.base_result.id for e in myerrs])
>>> print(res.n_updated)
```

3.7.2 Delete a column

Models are stored in `Dataset.data.history`,

```
>>> print(ds.data.history)
    # Example output
    # {('energy', 'psi4', 'b3lyp', 'def2-svp', 'scf_default'),
    # ('energy', 'psi4', 'hf', 'sto-3g', 'scf_default'),
    # ('energy', 'psi4', 'lda', 'sto-3g', 'scf_default')}
```

and can be removed from a *Dataset* or *ReactionDataset* via:

```
>>> ds.data.history.remove(('energy', 'psi4', 'lda', 'sto-3g', 'scf_default'))
>>> ds.save()

>>> ds = client.get_collection(...)
>>> print(ds.data.history)
    # Example output
    # {('energy', 'psi4', 'b3lyp', 'def2-svp', 'scf_default'),
    # ('energy', 'psi4', 'hf', 'sto-3g', 'scf_default')}
```

For further examples on how to interact with dataset collections hosted on QCArchive see the QCArchive examples.

Records

Documentation for compute records.

- *Overview*
- *Results*
- *Optimization*
- *API*

3.8 Overview

Records are the stored values of a completed computation. Each `Record` type corresponds to a specific operation formatted by QCArchive. Several `Record` examples include:

- *Result* - data from a single (often quantum chemical) energy, gradient, Hesssian, or property computation,
- *Optimization* - data resulted from a geometry optimization at a given level of theory,
- `GridOptimization` - results of a chain of geometry optimizations where starting structures at each step depends on previous structures, or
- `TorsionDrive` - the outcome of a special type of `GridOptimization` specifically for torsion scans that is able to find global minimum structures on the potential energy surfaces.

In general, `Records` are indexed by hashes and therefore, can be easily queried within a dataset `Collection`.

3.9 Results

The `result` of a single (often quantum chemical) calculation which can correspond to an energy, gradient, Hessian, or property computation.

3.9.1 Examples

Query Example

This notebook will cover example usage of `Result` record. As a note we will be using the MolSSI QCArchive server as a data source. Any `id`s used in this example will not be valid for local servers.

[2]:
```
import qcfractal.interface as ptl
client = ptl.FractalClient()
client
```

[2]:
```
FractalClient(server_name='The MolSSI QCArchive Server', address='https://api.qcarchive.
↪molssi.org:443/', username='None')
```

We can query results from the database based off a variety of values, but for this example we will query a known result from the database.

[20]:
```
record = client.query_results(id=1615129)[0]
record
```

[20]:
```
<ResultRecord(id='1615129' status='COMPLETE')>
```

There are a variety of helped functions on this

[21]:
```
record.get_molecule()
```

Data type cannot be displayed: application/3dmoljs_load.v0, text/html

[21]:
```
<Molecule(name='C8H11N3O' formula='C8H11N3O' hash='f0e2dbc')>
```

[37]:
```
print(f"Program: {record.program} Model: {record.method.upper()}/{record.basis} ({record.
↪driver})")
```

```
Program: psi4 Model: B3LYP-D3(BJ)/dzvp (gradient)
```

Since the primary `driver` of this computation a `gradient` is in the `record.return_result` slot as a NumPy array. In addition, all parsed intermediates quantities can be found in the `record.properties` slot for intermediate data.

[38]:
```
record.return_result[:3]
```

[38]:
```
array([[-1.0183802691625713e-05,  2.4983023230544212e-05,
         4.2626711726293591e-05],
       [ 4.8518698310640986e-05,  2.8649537718356986e-06,
        -2.5993909127035813e-05],
       [-1.4760360682320329e-05, -2.1228208508431367e-05,
        -4.6180180609961721e-05]])
```

[28]:
```
record.properties.dict()
```

[28]:
```
{'calcinfo_nbasis': 190,
 'calcinfo_nmo': 190,
 'calcinfo_nalpha': 44,
 'calcinfo_nbeta': 44,
 'calcinfo_natom': 23,
 'nuclear_repulsion_energy': 693.2680312650867,
 'return_energy': -551.0714449809266,
 'scf_one_electron_energy': -2121.681087605063,
 'scf_two_electron_energy': 939.395572266993,
 'scf_xc_energy': -62.01215606794322,
 'scf_dispersion_correction_energy': -0.04180484,
 'scf_dipole_moment': [-0.43157518923938226,
  3.5732964493427315,
  -0.35820710190059735],
 'scf_total_energy': -551.0714449809266,
 'scf_iterations': 14}
```

Additional data such as the output of the quantum chemistry program can be queried as well:

[34]:
```
print(record.get_stdout()[:245])
```

```
  Memory set to  60.800 GiB by Python driver.
gradient() will perform analytic gradient computation.
```

(continues on next page)

(continued from previous page)

```
*** tstart() called on ca167
*** at Fri May 17 00:25:45 2019

   => Loading Basis Set <=

    Name: DZVP
    Role: ORBITAL
    Keyword: BASIS
```

Compute Example

Computation of a result can be accomplished by specifying all parameters to a quantum chemistry computation and a molecule. An example can be seen below using a FractalSnowflake instance.

[39]:
```
from qcfractal import FractalSnowflakeHandler
snowflake = FractalSnowflakeHandler()
client = snowflake.client()
client
```

[39]:
```
FractalClient(server_name='db_0fa4f6ef_db7e_4f2e_9b7e_7c31cd9790d9', address='https://
↪localhost:55881/', username='None')
```

[45]:
```
methane = ptl.Molecule.from_data("pubchem:methane")
methane
```

```
        Searching PubChem database for methane (single best match returned)
        Found 1 result(s)
```

Data type cannot be displayed: application/3dmoljs_load.v0, text/html

[45]:
```
<Molecule(name='IUPAC methane' formula='CH4' hash='b4057dd')>
```

To run a quantum chemistry computation on this methane molecule we need to specify the full input as shown below. It should be noted that this function is also organized in such a way where the computation of many molecules with the same level of theory is most efficient.

[54]:
```
#                          program, method, basis,   driver,  keywords, molecule
compute = client.add_compute('psi4', 'hf', 'sto-3g', 'energy', None, [methane])
compute
```

[54]:
```
<ComputeResponse(nsubmitted=0 nexisting=1)>
```

The ids of the submitted compute can then be queried and examined. As a note the computation is not instantaneous, you may need to wait a moment and requery for this small molecule.

[52]:
```
result = client.query_results(id=compute.ids)[0]
result
```

[52]:
```
<ResultRecord(id='1' status='COMPLETE')>
```

```
[51]: result.return_result
[51]: -39.726728729639106
```

3.9.2 Attributes

class qcportal.models.**ResultRecord**(*, *client: Any = None*, *cache: Dict[str, Any] = {}*, *id: qcportal.models.common_models.ObjectId = None*, *hash_index: str = None*, *procedure: qcportal.models.records.ConstrainedStrValue = 'single'*, *program: str*, *version: int = 1*, *protocols: qcelemental.models.results.ResultProtocols = ResultProtocols(wavefunction=<WavefunctionProtocolEnum.none: 'none'>, stdout=True, error_correction=ErrorCorrectionProtocol(default_policy=True, policies=None))*, *extras: Dict[str, Any] = {}*, *stdout: qcportal.models.common_models.ObjectId = None*, *stderr: qcportal.models.common_models.ObjectId = None*, *error: qcportal.models.common_models.ObjectId = None*, *manager_name: str = None*, *status: qcportal.models.records.RecordStatusEnum = RecordStatusEnum.incomplete*, *modified_on: datetime.datetime = None*, *created_on: datetime.datetime = None*, *provenance: qcelemental.models.common_models.Provenance = None*, *driver: qcportal.models.common_models.DriverEnum*, *method: str*, *molecule: qcportal.models.common_models.ObjectId*, *basis: str = None*, *keywords: qcportal.models.common_models.ObjectId = None*, *return_result: Union[float, qcelemental.models.types.Array, Dict[str, Any]] = None*, *properties: qcelemental.models.results.ResultProperties = None*, *wavefunction: Dict[str, Any] = None*, *wavefunction_data_id: qcportal.models.common_models.ObjectId = None*)

Parameters

- **client** (*Any, Optional*) – The client object which the records are fetched from.
- **cache** (*Dict[Any], Default: {}*) – Object cache from expensive queries. It should be very rare that this needs to be set manually by the user.
- **id** (*ObjectId, Optional*) – Id of the object on the database. This is assigned automatically by the database.
- **hash_index** (*str, Optional*) – Hash of this object used to detect duplication and collisions in the database.
- **procedure** (*ConstrainedStrValue, Default: single*) – Procedure is fixed as "single" because this is single quantum chemistry result.
- **program** (*str*) – The quantum chemistry program which carries out the individual quantum chemistry calculations.
- **version** (*int, Default: 1*) – Version of the ResultRecord Model which this data was created with.
- **protocols** (`ResultProtocols`, Optional)
- **extras** (*Dict[Any], Default: {}*) – Extra information to associate with this record.

- **stdout** (*ObjectId, Optional*) – The Id of the stdout data stored in the database which was used to generate this record from the various programs which were called in the process.
- **stderr** (*ObjectId, Optional*) – The Id of the stderr data stored in the database which was used to generate this record from the various programs which were called in the process.
- **error** (*ObjectId, Optional*) – The Id of the error data stored in the database in the event that an error was generated in the process of carrying out the process this record targets. If no errors were raised, this field will be empty.
- **manager_name** (*str, Optional*) – Name of the Queue Manager which generated this record.
- **status** (*{COMPLETE,INCOMPLETE,RUNNING,ERROR}, Default: INCOMPLETE*) – The state of a record object. The states which are available are a finite set.
- **modified_on** (*datetime, Optional*) – Last time the data this record points to was modified.
- **created_on** (*datetime, Optional*) – Time the data this record points to was first created.
- **provenance** (`Provenance`, Optional) – Provenance information tied to the creation of this record. This includes things such as every program which was involved in generating the data for this record.
- **driver** (*{energy,gradient,hessian,properties}*) – The type of calculation that is being performed (e.g., energy, gradient, Hessian, ...).
- **method** (*str*) – The quantum chemistry method the driver runs with.
- **molecule** (*ObjectId*) – The Id of the molecule in the Database which the result is computed on.
- **basis** (*str, Optional*) – The quantum chemistry basis set to evaluate (e.g., 6-31g, cc-pVDZ, ...). Can be `None` for methods without basis sets.
- **keywords** (*ObjectId, Optional*) – The Id of the *KeywordSet* which was passed into the quantum chemistry program that performed this calculation.
- **return_result** (*Union[float, Array, Dict[Any]], Optional*) – The primary result of the calculation, output is a function of the specified `driver`.
- **properties** (`ResultProperties`, Optional) – Additional data and results computed as part of the `return_result`.
- **wavefunction** (*Dict[Any], Optional*) – Wavefunction data generated by the Result.
- **wavefunction_data_id** (*ObjectId, Optional*) – The id of the wavefunction

3.10 Optimization

The `Record` of a geometry optimization.

3.10.1 Examples

Query Example

This notebook will cover example usage of `Optimization` record. As a note we will be using the MolSSI QCArchive server as a data source. Any `id`s used in this example will not be valid for local servers.

[1]:
```
import qcfractal.interface as ptl
client = ptl.FractalClient()
client
```

[1]:
```
FractalClient(server_name='The MolSSI QCArchive Server', address='https://api.qcarchive.
↪molssi.org:443/', username='None')
```

We can query results from the database based off a variety of values, but for this example we will query a known result from the database.

[2]:
```
record = client.query_procedures(id=1683293)[0]
record
```

[2]:
```
<OptimizationRecord(id='1683293' status='COMPLETE')>
```

There are a variety of helper functions on this object to find quantities related to the computation.

[3]:
```
record.get_final_molecule()
```

Data type cannot be displayed: application/3dmoljs_load.v0, text/html

[3]:
```
<Molecule(name='C14H14O' formula='C14H14O' hash='2594158')>
```

[4]:
```
record.show_history()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

We can also observe the program, method, and basis for which the optimization was executed under.

[5]:
```
record.qc_spec.dict()
```

[5]:
```
{'driver': <DriverEnum.gradient: 'gradient'>,
 'method': 'b3lyp-d3(bj)',
 'basis': 'dzvp',
 'keywords': None,
 'program': 'psi4'}
```

We can also find all keywords passed into the geometry optimization. Here we see that this geometry optimization was evaluated under a dihedral constraint.

[6]:
```
record.keywords
```

[6]:
```
{'coordsys': 'tric',
 'enforce': 0.1,
 'reset': True,
 'qccnv': True,
```

(continues on next page)

(continued from previous page)

```
 'epsilon': 0,
 'constraints': {'set': [{'type': 'dihedral',
    'indices': [9, 10, 13, 14],
    'value': 30}]},
 'program': 'psi4'}
```

Finally, every Result generated in the computational trajectory can be queried and observed. Here we will obtain the very last computed Result.

[7]:
```
record.get_trajectory()[-1]
```

[7]:
```
<ResultRecord(id='1467860' status='COMPLETE')>
```

Compute Example

Computation of a result can be accomplished by specifying all parameters to a quantum chemistry computation and a molecule. An example can be seen below using a FractalSnowflake instance.

[8]:
```
from qcfractal import FractalSnowflakeHandler
snowflake = FractalSnowflakeHandler()
client = snowflake.client()
client
```

[8]:
```
FractalClient(server_name='FractalSnowFlake_db_f1c89', address='https://localhost:57605/
↪', username='None')
```

[9]:
```
methane = ptl.Molecule.from_data("pubchem:methane")
methane
```

```
        Searching PubChem database for methane (single best match returned)
        Found 1 result(s)
```

Data type cannot be displayed: application/3dmoljs_load.v0, text/html

[9]:
```
<Molecule(name='IUPAC methane' formula='CH4' hash='b4057dd')>
```

To run a optimization on this methane molecule we need to specify the full input as shown below. It should be noted that this function is also organized in such a way where the optimization of many molecules with the same level of theory is most efficient.

[11]:
```
options = {
    "keywords": {'coordsys': 'tric'}, # Geometry optimization program options
    "qc_spec": {                      # Quantum chemistry specifications
        "driver": "gradient",
        "method": "HF",
        "basis": "sto-3g",
        "keywords": None,
        "program": "psi4"
    },
}
compute = client.add_procedure("optimization", "geometric", options, [methane])
compute
```

[11]:
```
<ComputeResponse(nsubmitted=1 nexisting=0)>
```

The ids of the submitted optimization can then be queried and examined. As a note the computation is not instantaneous, you may need to wait a moment and requery for this small molecule.

[15]:
```
result = client.query_procedures(id=compute.ids)[0]
result
```

[15]:
```
<OptimizationRecord(id='1' status='COMPLETE')>
```

[20]:
```
ch_bond_original = result.get_initial_molecule().measure([0, 1])
ch_bond_optimized = result.get_final_molecule().measure([0, 1])
print(f"Optimized/Original C-H bond {ch_bond_original}/{ch_bond_optimized} (bohr)")
```

```
Optimized/Original C-H bond 2.0639574742067777/2.0465935246983378 (bohr)
```

[21]:
```
result.show_history()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[ ]:
```

```

3.10.2 Attributes

class qcportal.models.**OptimizationRecord**(*, *client: Any = None*, *cache: Dict[str, Any] = {}*, *id: qcportal.models.common_models.ObjectId = None*, *hash_index: str = None*, *procedure: qcportal.models.records.ConstrainedStrValue = 'optimization'*, *program: str*, *version: int = 1*, *protocols: qcelemental.models.procedures.OptimizationProtocols = OptimizationProtocols(trajectory=<TrajectoryProtocolEnum.all: 'all'>)*, *extras: Dict[str, Any] = {}*, *stdout: qcportal.models.common_models.ObjectId = None*, *stderr: qcportal.models.common_models.ObjectId = None*, *error: qcportal.models.common_models.ObjectId = None*, *manager_name: str = None*, *status: qcportal.models.records.RecordStatusEnum = RecordStatusEnum.incomplete*, *modified_on: datetime.datetime = None*, *created_on: datetime.datetime = None*, *provenance: qcelemental.models.common_models.Provenance = None*, *schema_version: int = 1*, *initial_molecule: qcportal.models.common_models.ObjectId*, *qc_spec: qcportal.models.common_models.QCSpecification*, *keywords: Dict[str, Any] = {}*, *energies: List[float] = None*, *final_molecule: qcportal.models.common_models.ObjectId = None*, *trajectory: List[qcportal.models.common_models.ObjectId] = None*)

A OptimizationRecord for all optimization procedure data.

Parameters

- **client** (*Any, Optional*) – The client object which the records are fetched from.
- **cache** (*Dict[Any], Default: {}*) – Object cache from expensive queries. It should be very rare that this needs to be set manually by the user.
- **id** (*ObjectId, Optional*) – Id of the object on the database. This is assigned automatically by the database.
- **hash_index** (*str, Optional*) – Hash of this object used to detect duplication and collisions in the database.
- **procedure** (*ConstrainedStrValue, Default: optimization*) – A fixed string indication this is a record for an "Optimization".
- **program** (*str*) – The quantum chemistry program which carries out the individual quantum chemistry calculations.
- **version** (*int, Default: 1*) – Version of the OptimizationRecord Model which this data was created with.
- **protocols** (*OptimizationProtocols*, Optional)
- **extras** (*Dict[Any], Default: {}*) – Extra information to associate with this record.
- **stdout** (*ObjectId, Optional*) – The Id of the stdout data stored in the database which was used to generate this record from the various programs which were called in the process.
- **stderr** (*ObjectId, Optional*) – The Id of the stderr data stored in the database which was used to generate this record from the various programs which were called in the process.
- **error** (*ObjectId, Optional*) – The Id of the error data stored in the database in the event that an error was generated in the process of carrying out the process this record targets. If no errors were raised, this field will be empty.
- **manager_name** (*str, Optional*) – Name of the Queue Manager which generated this record.
- **status** (*{COMPLETE,INCOMPLETE,RUNNING,ERROR}, Default: INCOMPLETE*) – The state of a record object. The states which are available are a finite set.
- **modified_on** (*datetime, Optional*) – Last time the data this record points to was modified.
- **created_on** (*datetime, Optional*) – Time the data this record points to was first created.
- **provenance** (Provenance, Optional) – Provenance information tied to the creation of this record. This includes things such as every program which was involved in generating the data for this record.
- **schema_version** (*int, Default: 1*) – The version number of QCSchema under which this record conforms to.
- **initial_molecule** (*ObjectId*) – The Id of the molecule which was passed in as the reference for this Optimization.
- **qc_spec** (*QCSpecification*) – The specification of the quantum chemistry calculation to run at each point.
- **keywords** (*Dict[Any], Default: {}*) – The keyword options which were passed into the Optimization program. Note: These are a dictionary and not a *KeywordSet* object.
- **energies** (*List[float], Optional*) – The ordered list of energies at each step of the Optimization.
- **final_molecule** (*ObjectId, Optional*) – The ObjectId of the final, optimized Molecule the Optimization procedure converged to.

- **trajectory** (*List[ObjectId], Optional*) – The list of Molecule Id's the Optimization procedure generated at each step of the optimization.``initial_molecule`` will be the first index, and `final_molecule` will be the last index.

`get_final_energy`() → float
The final energy of the geometry optimization.

Returns The optimization molecular energy.

Return type float

`get_final_molecule`() → *Molecule*
Returns the optimized molecule

Returns The optimized molecule

Return type *Molecule*

`get_initial_molecule`() → *Molecule*
Returns the initial molecule

Returns The initial molecule

Return type *Molecule*

`get_molecular_trajectory`() → List[*Molecule*]
Returns the Molecule at each gradient evaluation in the trajectory.

Returns A ordered list of Molecules in the trajectory.

Return type List['Molecule']

`get_trajectory`() → List[*qcportal.models.records.ResultRecord*]
Returns the Result records for each gradient evaluation in the trajectory.

Returns A ordered list of Result record gradient computations.

Return type List['ResultRecord']

`show_history`(*units: str = 'kcal/mol'*, *digits: int = 3*, *relative: bool = True*, *return_figure: Optional[bool] = None*) → plotly.Figure
Plots the energy of the trajectory the optimization took.

Parameters

- **units** (*str, optional*) – Units to display the trajectory in.
- **digits** (*int, optional*) – The number of valid digits to show.
- **relative** (*bool, optional*) – If True, all energies are shifted by the lowest energy in the trajectory. Otherwise provides raw energies.
- **return_figure** (*Optional[bool], optional*) – If True, return the raw plotly figure. If False, returns a hosted iPlot. If None, return a iPlot display in Jupyter notebook and a raw plotly figure in all other circumstances.

Returns The requested figure.

Return type plotly.Figure

3.11 API

The complete set of object models and relations implemented by QCPortal. Every class shown here is its own model and the attributes shown are valid kwargs and values which can be fed into the construction.

class qcportal.models.**KeywordSet**(*, *id: qcportal.models.common_models.ObjectId = None*, *hash_index: str*, *values: Dict[str, Optional[Any]]*, *lowercase: bool = True*, *exact_floats: bool = False*, *comments: str = None*)

A key:value storage object for Keywords.

Parameters

- **id** (*ObjectId, Optional*) – The Id of this object, will be automatically assigned when added to the database.
- **hash_index** (*str*) – The hash of this keyword set to store and check for collisions. This string is automatically computed.
- **values** (*Dict[Any]*) – The key-value pairs which make up this KeywordSet. There is no direct relation between this dictionary and applicable program/spec it can be used on.
- **lowercase** (*bool, Default: True*) – String keys are in the `values` dict are normalized to lowercase if this is True. Assists in matching against other `KeywordSet` objects in the database.
- **exact_floats** (*bool, Default: False*) – All floating point numbers are rounded to 1.e-10 if this is False.Assists in matching against other `KeywordSet` objects in the database.
- **comments** (*str, Optional*) – Additional comments for this KeywordSet. Intended for pure human/user consumption and clarity.

class qcportal.models.**Molecule**(*orient: bool = False*, *validate: Optional[bool] = None*, *, *schema_name: qcelemental.models.molecule.ConstrainedStrValue = 'qcschema_molecule'*, *schema_version: int = 2*, *validated: bool = False*, *symbols: qcelemental.models.types.Array*, *geometry: qcelemental.models.types.Array*, *name: str = None*, *identifiers: qcelemental.models.molecule.Identifiers = None*, *comment: str = None*, *molecular_charge: float = 0.0*, *molecular_multiplicity: int = 1*, *masses: qcelemental.models.types.Array = None*, *real: qcelemental.models.types.Array = None*, *atom_labels: qcelemental.models.types.Array = None*, *atomic_numbers: qcelemental.models.types.Array = None*, *mass_numbers: qcelemental.models.types.Array = None*, *connectivity: types.ConstrainedListValue[Tuple[qcelemental.models.molecule.NonnegativeInt, qcelemental.models.molecule.NonnegativeInt, qcelemental.models.molecule.BondOrderFloat]] = None*, *fragments: List[qcelemental.models.types.Array] = None*, *fragment_charges: List[float] = None*, *fragment_multiplicities: List[int] = None*, *fix_com: bool = False*, *fix_orientation: bool = False*, *fix_symmetry: str = None*, *provenance: qcelemental.models.common_models.Provenance = None*, *id: Any = None*, *extras: Dict[str, Any] = None*)

The physical Cartesian representation of the molecular system.

A QCSchema representation of a Molecule. This model contains data for symbols, geometry, connectivity, charges, fragmentation, etc while also supporting a wide array of I/O and manipulation capabilities.

Molecule objects geometry, masses, and charges are truncated to 8, 6, and 4 decimal places respectively to assist with duplicate detection.

Notes

All arrays are stored flat but must be reshapable into the dimensions in attribute `shape`, with abbreviations as follows:

- nat: number of atomic = calcinfo_natom
- nfr: number of fragments
- <varies>: irregular dimension not systematically reshapable

class qcportal.models.**OptimizationRecord**(*, *client: Any = None*, *cache: Dict[str, Any] = {}*, *id: qcportal.models.common_models.ObjectId = None*, *hash_index: str = None*, *procedure: qcportal.models.records.ConstrainedStrValue = 'optimization'*, *program: str*, *version: int = 1*, *protocols: qcelemental.models.procedures.OptimizationProtocols = OptimizationProtocols(trajectory=<TrajectoryProtocolEnum.all: 'all'>)*, *extras: Dict[str, Any] = {}*, *stdout: qcportal.models.common_models.ObjectId = None*, *stderr: qcportal.models.common_models.ObjectId = None*, *error: qcportal.models.common_models.ObjectId = None*, *manager_name: str = None*, *status: qcportal.models.records.RecordStatusEnum = RecordStatusEnum.incomplete*, *modified_on: datetime.datetime = None*, *created_on: datetime.datetime = None*, *provenance: qcelemental.models.common_models.Provenance = None*, *schema_version: int = 1*, *initial_molecule: qcportal.models.common_models.ObjectId*, *qc_spec: qcportal.models.common_models.QCSpecification*, *keywords: Dict[str, Any] = {}*, *energies: List[float] = None*, *final_molecule: qcportal.models.common_models.ObjectId = None*, *trajectory: List[qcportal.models.common_models.ObjectId] = None*)

A OptimizationRecord for all optimization procedure data.

Parameters

- **client** (*Any, Optional*) – The client object which the records are fetched from.
- **cache** (*Dict[Any], Default: {}*) – Object cache from expensive queries. It should be very rare that this needs to be set manually by the user.
- **id** (*ObjectId, Optional*) – Id of the object on the database. This is assigned automatically by the database.
- **hash_index** (*str, Optional*) – Hash of this object used to detect duplication and collisions in the database.
- **procedure** (*ConstrainedStrValue, Default: optimization*) – A fixed string indication this is a record for an "Optimization".
- **program** (*str*) – The quantum chemistry program which carries out the individual quantum chemistry calculations.
- **version** (*int, Default: 1*) – Version of the OptimizationRecord Model which this data was created with.
- **protocols** (*OptimizationProtocols*, Optional)
- **extras** (*Dict[Any], Default: {}*) – Extra information to associate with this record.

- **stdout** (*ObjectId, Optional*) – The Id of the stdout data stored in the database which was used to generate this record from the various programs which were called in the process.
- **stderr** (*ObjectId, Optional*) – The Id of the stderr data stored in the database which was used to generate this record from the various programs which were called in the process.
- **error** (*ObjectId, Optional*) – The Id of the error data stored in the database in the event that an error was generated in the process of carrying out the process this record targets. If no errors were raised, this field will be empty.
- **manager_name** (*str, Optional*) – Name of the Queue Manager which generated this record.
- **status** (*{COMPLETE,INCOMPLETE,RUNNING,ERROR}, Default: INCOMPLETE*) – The state of a record object. The states which are available are a finite set.
- **modified_on** (*datetime, Optional*) – Last time the data this record points to was modified.
- **created_on** (*datetime, Optional*) – Time the data this record points to was first created.
- **provenance** (`Provenance`, Optional) – Provenance information tied to the creation of this record. This includes things such as every program which was involved in generating the data for this record.
- **schema_version** (*int, Default: 1*) – The version number of QCSchema under which this record conforms to.
- **initial_molecule** (*ObjectId*) – The Id of the molecule which was passed in as the reference for this Optimization.
- **qc_spec** (*QCSpecification*) – The specification of the quantum chemistry calculation to run at each point.
- **keywords** (*Dict[Any], Default: {}*) – The keyword options which were passed into the Optimization program. Note: These are a dictionary and not a *KeywordSet* object.
- **energies** (*List[float], Optional*) – The ordered list of energies at each step of the Optimization.
- **final_molecule** (*ObjectId, Optional*) – The `ObjectId` of the final, optimized Molecule the Optimization procedure converged to.
- **trajectory** (*List[ObjectId], Optional*) – The list of Molecule Id's the Optimization procedure generated at each step of the optimization.``initial_molecule`` will be the first index, and `final_molecule` will be the last index.

get_final_energy() → float
The final energy of the geometry optimization.

Returns The optimization molecular energy.

Return type float

get_final_molecule() → *Molecule*
Returns the optimized molecule

Returns The optimized molecule

Return type *Molecule*

get_initial_molecule() → *Molecule*
Returns the initial molecule

Returns The initial molecule

Return type *Molecule*

get_molecular_trajectory() → List[*Molecule*]
Returns the Molecule at each gradient evaluation in the trajectory.

Returns A ordered list of Molecules in the trajectory.

Return type List['Molecule']

get_trajectory() → List[*qcportal.models.records.ResultRecord*]
Returns the Result records for each gradient evaluation in the trajectory.

Returns A ordered list of Result record gradient computations.

Return type List['ResultRecord']

show_history(*units: str = 'kcal/mol'*, *digits: int = 3*, *relative: bool = True*, *return_figure: Optional[bool] = None*) → plotly.Figure
Plots the energy of the trajectory the optimization took.

Parameters

- **units** (*str, optional*) – Units to display the trajectory in.
- **digits** (*int, optional*) – The number of valid digits to show.
- **relative** (*bool, optional*) – If True, all energies are shifted by the lowest energy in the trajectory. Otherwise provides raw energies.
- **return_figure** (*Optional[bool], optional*) – If True, return the raw plotly figure. If False, returns a hosted iPlot. If None, return a iPlot display in Jupyter notebook and a raw plotly figure in all other circumstances.

Returns The requested figure.

Return type plotly.Figure

class qcportal.models.**QCSpecification**(*, *driver: qcportal.models.common_models.DriverEnum*, *method: str*, *basis: str = None*, *keywords: qcportal.models.common_models.ObjectId = None*, *protocols: qcelemental.models.results.ResultProtocols = ResultProtocols(wavefunction=<WavefunctionProtocolEnum.none: 'none'>, stdout=True, error_correction=ErrorCorrectionProtocol(default_policy=True, policies=None))*, *program: str*)
The quantum chemistry metadata specification for individual computations such as energy, gradient, and Hessians.

Parameters

- **driver** (*{energy,gradient,hessian,properties}*) – The type of calculation that is being performed (e.g., energy, gradient, Hessian, ...).
- **method** (*str*) – The quantum chemistry method to evaluate (e.g., B3LYP, PBE, ...).
- **basis** (*str, Optional*) – The quantum chemistry basis set to evaluate (e.g., 6-31g, cc-pVDZ, ...). Can be `None` for methods without basis sets.
- **keywords** (*ObjectId, Optional*) – The Id of the `KeywordSet` registered in the database to run this calculation with. This Id must exist in the database.
- **protocols** (`ResultProtocols`, Optional) – None
- **program** (*str*) – The quantum chemistry program to evaluate the computation with. Not all quantum chemistry programs support all combinations of driver/method/basis.

class qcportal.models.**GridOptimizationInput**(*, *program: qcportal.models.gridoptimization.ConstrainedStrValue = 'qcfractal'*, *procedure: qcportal.models.gridoptimization.ConstrainedStrValue = 'gridoptimization'*, *initial_molecule: Union[qcportal.models.common_models.ObjectId, qcelemental.models.molecule.Molecule]*, *keywords: qcportal.models.gridoptimization.GOKeywords*, *optimization_spec: qcportal.models.common_models.OptimizationSpecification*, *qc_spec: qcportal.models.common_models.QCSpecification*)

The input to create a GridOptimization Service with.

Parameters

- **program** (*ConstrainedStrValue, Default: qcfractal*) – The name of the source program which initializes the Grid Optimization. This is a constant and is used for provenance information.
- **procedure** (*ConstrainedStrValue, Default: gridoptimization*) – The name of the procedure being run. This is a constant and is used for provenance information.
- **initial_molecule** (Union[ObjectId, *Molecule*]) – The Molecule to begin the Grid Optimization with. This can either be an existing Molecule in the database (through its `ObjectId`) or a fully specified *Molecule* model.
- **keywords** (`GOKeywords`) – The keyword options to run the Grid Optimization.
- **optimization_spec** (*OptimizationSpecification*) – The specification to run the underlying optimization through at each grid point.
- **qc_spec** (*QCSpecification*) – The specification for each of the quantum chemistry calculations run in each geometry optimization.

class qcportal.models.**GridOptimizationRecord**(*, *client: Any = None*, *cache: Dict[str, Any] = {}*, *id: qcportal.models.common_models.ObjectId = None*, *hash_index: str = None*, *procedure: qcportal.models.gridoptimization.ConstrainedStrValue = 'gridoptimization'*, *program: qcportal.models.gridoptimization.ConstrainedStrValue = 'qcfractal'*, *version: int = 1*, *protocols: Dict[str, Any] = None*, *extras: Dict[str, Any] = {}*, *stdout: qcportal.models.common_models.ObjectId = None*, *stderr: qcportal.models.common_models.ObjectId = None*, *error: qcportal.models.common_models.ObjectId = None*, *manager_name: str = None*, *status: qcportal.models.records.RecordStatusEnum = RecordStatusEnum.incomplete*, *modified_on: datetime.datetime = None*, *created_on: datetime.datetime = None*, *provenance: qcelemental.models.common_models.Provenance = None*, *initial_molecule: qcportal.models.common_models.ObjectId*, *keywords: qcportal.models.gridoptimization.GOKeywords*, *optimization_spec:* qcportal.models.common_models.OptimizationSpecification, *qc_spec:* qcportal.models.common_models.QCSpecification, *starting_molecule: qcportal.models.common_models.ObjectId*, *final_energy_dict: Dict[str, float]*, *grid_optimizations: Dict[str, qcportal.models.common_models.ObjectId]*, *starting_grid: tuple*)

The record of a GridOptimization service result.

A GridOptimization is a type of constrained optimization in which a set of dimension are scanned over. An is to compute the

Parameters

- **client** (*Any, Optional*) – The client object which the records are fetched from.
- **cache** (*Dict[Any], Default: {}*) – Object cache from expensive queries. It should be very rare that this needs to be set manually by the user.
- **id** (*ObjectId, Optional*) – Id of the object on the database. This is assigned automatically by the database.
- **hash_index** (*str, Optional*) – Hash of this object used to detect duplication and collisions in the database.
- **procedure** (*ConstrainedStrValue, Default: gridoptimization*) – The name of the procedure being run, which is Grid Optimization. This is a constant and is used for provenance information.
- **program** (*ConstrainedStrValue, Default: qcfractal*) – The name of the source program which initializes the Grid Optimization. This is a constant and is used for provenance information.
- **version** (*int, Default: 1*) – The version number of the Record.
- **protocols** (*Dict[Any], Optional*) – Protocols that change the data stored in top level fields.

- **extras** (*Dict[Any], Default: {}*) – Extra information to associate with this record.
- **stdout** (*ObjectId, Optional*) – The Id of the stdout data stored in the database which was used to generate this record from the various programs which were called in the process.
- **stderr** (*ObjectId, Optional*) – The Id of the stderr data stored in the database which was used to generate this record from the various programs which were called in the process.
- **error** (*ObjectId, Optional*) – The Id of the error data stored in the database in the event that an error was generated in the process of carrying out the process this record targets. If no errors were raised, this field will be empty.
- **manager_name** (*str, Optional*) – Name of the Queue Manager which generated this record.
- **status** (*{COMPLETE,INCOMPLETE,RUNNING,ERROR}, Default: INCOMPLETE*) – The state of a record object. The states which are available are a finite set.
- **modified_on** (*datetime, Optional*) – Last time the data this record points to was modified.
- **created_on** (*datetime, Optional*) – Time the data this record points to was first created.
- **provenance** (`Provenance`, Optional) – Provenance information tied to the creation of this record. This includes things such as every program which was involved in generating the data for this record.
- **initial_molecule** (*ObjectId*) – Id of the initial molecule in the database.
- **keywords** (`GOKeywords`) – The keywords for this Grid Optimization.
- **optimization_spec** (*OptimizationSpecification*) – The specification of each geometry optimization.
- **qc_spec** (*QCSpecification*) – The specification for each of the quantum chemistry computations used by the geometry optimizations.
- **starting_molecule** (*ObjectId*) – Id of the molecule in the database begins the grid optimization. This will differ from the `initial_molecule` if `preoptimization` is True.
- **final_energy_dict** (*Dict[float]*) – Map of the final energy from the grid optimization at each grid point.
- **grid_optimizations** (*Dict[ObjectId]*) – The Id of each optimization at each grid point.
- **starting_grid** (*tuple*) – Initial grid point from which the Grid Optimization started. This grid point is the closest in structure to the `starting_molecule`.

static deserialize_key(*key: str*) → Tuple[int]
Unpacks a string key to a python object.

Parameters **key** (*str*) – The input key

Returns The unpacked key.

Return type Tuple[int]

get_final_energies(*key: Optional[Union[int, str]] = None*) → Dict[str, float]
Provides the final optimized energies at each grid point.

Parameters **key** (*Union[int, str, None], optional*) – Specifies a single entry to pull from.

Returns **energy** – Returns energies at each grid point in a dictionary or at a single point if a key is specified.

Return type Dict[str, float]

Examples

```
>>> grid_optimization_record.get_final_energies()
{(-90,): -148.7641654446243, (180,): -148.76501336993732, (0,): -148.
↪75056290106735, (90,): -148.7641654446148}
```

```
>>> grid_optimization_record.get_final_energies((-90,))
-148.7641654446243
```

get_final_molecules(*key: Optional[Union[int, str]] = None*) → Dict[str, *qcelemental.models.molecule.Molecule*]

Provides the final optimized molecules at each grid point.

Parameters **key** (*Union[int, str, None], optional*) – Specifies a single entry to pull from.

Returns **final_molecules** – Returns energies at each grid point in a dictionary or at a single point if a key is specified.

Return type Dict[str, 'Molecule']

Examples

```
>>> mols = grid_optimization_record.get_final_molecules()
>>> type(mols[(-90, )])
qcelemental.models.molecule.Molecule
```

```
>>> type(grid_optimization_record.get_final_molecules((-90,)))
qcelemental.models.molecule.Molecule
```

get_final_results(*key: Union[int, Tuple[int, ...], str] = None*) → Dict[str, *ResultRecord*]

Returns the final opt gradient result records at each grid point.

Parameters **key** (*Union[int, Tuple[int, ...], str], optional*) – Specifies a single entry to pull from.

Returns **final_results** – Returns ResultRecord at each grid point in a dictionary or at a single point if a key is specified.

Return type Dict[str, 'ResultRecord']

Examples

```
>>> mols = grid_optimization_record.get_final_results()
>>> type(mols[(-90, )])
qcfractal.interface.models.records.ResultRecord
```

```
>>> type(grid_optimization_record.get_final_results((-90,)))
qcfractal.interface.models.records.ResultRecord
```

get_history(*key: Optional[Union[int, str]] = None*) → Dict[str, Optimization]

Pulls the optimization history of the computation.

Parameters **key** (*Union[int, str, None], optional*) – Specifies a single entry to pull from.

Returns Return the optimizations in the computed history.

Return type Dict[str, 'Optimization']

`get_scan_dimensions()` → Tuple[float, ...]
Returns the overall dimensions of the scan.

Returns The size of each dimension in the scan.

Return type Tuple[float, ...]

`get_scan_value`(*scan_number: Union[str, int, Tuple[int]]*) → Tuple[float, ...]
Obtains the scan parameters at a given grid point.

Parameters **scan_number** (*Union[str, int, Tuple[int]]*) – The key of the scan.

Returns Description

Return type Tuple[float, ...]

`static serialize_key`(*key: Union[int, Tuple[int]]*) → str
Serializes the key to map to the internal keys.

Parameters **key** (*Union[int, Tuple[int]]*) – A integer or list of integers denoting the position in the grid to find.

Returns The internal key value.

Return type str

class qcportal.models.**OptimizationSpecification**(*, *program: str*, *keywords: Dict[str, Any] = None*, *protocols: qcelemental.models.procedures.OptimizationProtocols = OptimizationProtocols(trajectory=<TrajectoryProtocolEnum.all: 'all'>)*)

Metadata describing a geometry optimization.

Parameters

- **program** (*str*) – Optimization program to run the optimization with
- **keywords** (*Dict[Any], Optional*) – Dictionary of keyword arguments to pass into the `program` when the program runs. Note that unlike *QCSpecification* this is a dictionary of keywords, not the Id for a *KeywordSet*.
- **protocols** (*OptimizationProtocols*, Optional) – Protocols regarding the manipulation of a Optimization output data.

class qcportal.models.**OptimizationProtocols**(*, *trajectory: qcelemental.models.procedures.TrajectoryProtocolEnum = TrajectoryProtocolEnum.all*)

Protocols regarding the manipulation of a Optimization output data.

Parameters **trajectory** (*{all,initial_and_final,final,none}, Default: all*) – Which gradient evaluations to keep in an optimization trajectory.

class qcportal.models.**ResultProtocols**(**args*, *wavefunction: qcelemental.models.results.WavefunctionProtocolEnum = WavefunctionProtocolEnum.none*, *stdout: bool = True*, *error_correction: qcelemental.models.results.ErrorCorrectionProtocol = None*)

Parameters

- **wavefunction** (*{all,orbitals_and_eigenvalues,return_results,none}, Default: none*) – Wavefunction to keep from a computation.
- **stdout** (*bool, Default: True*) – Primary output file to keep from the computation
- **error_correction** (`ErrorCorrectionProtocol`, Optional) – Policies for error correction

class qcportal.models.**TorsionDriveInput**(*, *program: qcportal.models.torsiondrive.ConstrainedStrValue = 'torsiondrive'*, *procedure: qcportal.models.torsiondrive.ConstrainedStrValue = 'torsiondrive'*, *initial_molecule: List[Union[qcportal.models.common_models.ObjectId, qcelemental.models.molecule.Molecule]]*, *keywords: qcportal.models.torsiondrive.TDKeywords*, *optimization_spec: qcportal.models.common_models.OptimizationSpecification*, *qc_spec: qcportal.models.common_models.QCSpecification*)

A TorsionDriveRecord Input base class

Parameters

- **program** (*ConstrainedStrValue, Default: torsiondrive*) – The name of the program. Fixed to 'torsiondrive' since this input model is only valid for it.
- **procedure** (*ConstrainedStrValue, Default: torsiondrive*) – The name of the Procedure. Fixed to 'torsiondrive' since this input model is only valid for it.
- **initial_molecule** (List[Union[ObjectId, *Molecule*]]) – The Molecule(s) to begin the TorsionDrive with. This can either be an existing Molecule in the database (through its `ObjectId`) or a fully specified *Molecule* model.
- **keywords** (`TDKeywords`) – TorsionDrive-specific input arguments to pass into the TorsionDrive Procedure
- **optimization_spec** (*OptimizationSpecification*) – The settings which describe how to conduct the energy optimizations at each step of the torsion scan.
- **qc_spec** (*QCSpecification*) – The settings which describe the individual quantum chemistry calculations at each step of the optimization.

class qcportal.models.**TorsionDriveRecord**(*, *client: Any = None*, *cache: Dict[str, Any] = {}*, *id: qcportal.models.common_models.ObjectId = None*, *hash_index: str = None*, *procedure: qcportal.models.torsiondrive.ConstrainedStrValue = 'torsiondrive'*, *program: qcportal.models.torsiondrive.ConstrainedStrValue = 'torsiondrive'*, *version: int = 1*, *protocols: Dict[str, Any] = None*, *extras: Dict[str, Any] = {}*, *stdout: qcportal.models.common_models.ObjectId = None*, *stderr: qcportal.models.common_models.ObjectId = None*, *error: qcportal.models.common_models.ObjectId = None*, *manager_name: str = None*, *status: qcportal.models.records.RecordStatusEnum = RecordStatusEnum.incomplete*, *modified_on: datetime.datetime = None*, *created_on: datetime.datetime = None*, *provenance: qcelemental.models.common_models.Provenance = None*, *initial_molecule: List[qcportal.models.common_models.ObjectId]*, *keywords: qcportal.models.torsiondrive.TDKeywords*, *optimization_spec:* qcportal.models.common_models.OptimizationSpecification, *qc_spec:* qcportal.models.common_models.QCSpecification, *final_energy_dict: Dict[str, float]*, *optimization_history: Dict[str, List[qcportal.models.common_models.ObjectId]]*, *minimum_positions: Dict[str, int]*)

A interface to the raw JSON data of a TorsionDriveRecord torsion scan run.

Parameters

- **client** (*Any, Optional*) – The client object which the records are fetched from.
- **cache** (*Dict[Any], Default: {}*) – Object cache from expensive queries. It should be very rare that this needs to be set manually by the user.
- **id** (*ObjectId, Optional*) – Id of the object on the database. This is assigned automatically by the database.
- **hash_index** (*str, Optional*) – Hash of this object used to detect duplication and collisions in the database.
- **procedure** (*ConstrainedStrValue, Default: torsiondrive*) – The name of the procedure. Fixed to 'torsiondrive' since this is the Record explicit to TorsionDrive.
- **program** (*ConstrainedStrValue, Default: torsiondrive*) – The name of the program. Fixed to 'torsiondrive' since this is the Record explicit to TorsionDrive.
- **version** (*int, Default: 1*) – The version number of the Record.
- **protocols** (*Dict[Any], Optional*) – Protocols that change the data stored in top level fields.
- **extras** (*Dict[Any], Default: {}*) – Extra information to associate with this record.
- **stdout** (*ObjectId, Optional*) – The Id of the stdout data stored in the database which was used to generate this record from the various programs which were called in the process.
- **stderr** (*ObjectId, Optional*) – The Id of the stderr data stored in the database which was used to generate this record from the various programs which were called in the process.
- **error** (*ObjectId, Optional*) – The Id of the error data stored in the database in the event that an error was generated in the process of carrying out the process this record targets. If no errors were raised, this field will be empty.

- **manager_name** (*str, Optional*) – Name of the Queue Manager which generated this record.
- **status** (*{COMPLETE,INCOMPLETE,RUNNING,ERROR}, Default: INCOMPLETE*) – The state of a record object. The states which are available are a finite set.
- **modified_on** (*datetime, Optional*) – Last time the data this record points to was modified.
- **created_on** (*datetime, Optional*) – Time the data this record points to was first created.
- **provenance** (`Provenance`, Optional) – Provenance information tied to the creation of this record. This includes things such as every program which was involved in generating the data for this record.
- **initial_molecule** (*List[ObjectId]*) – Id(s) of the initial molecule(s) in the database.
- **keywords** (`TDKeywords`) – The TorsionDrive-specific input arguments used for this operation.
- **optimization_spec** (*OptimizationSpecification*) – The settings which describe how the energy optimizations at each step of the torsion scan used for this operation.
- **qc_spec** (*QCSpecification*) – The settings which describe how the individual quantum chemistry calculations are handled for this operation.
- **final_energy_dict** (*Dict[float]*) – The final energy at each angle of the TorsionDrive scan.
- **optimization_history** (*Dict[List[ObjectId]]*) – The map of each angle of the TorsionDrive scan to each optimization computations. Each value of the dict maps to a sequence of `ObjectId` strings which each point to a single computation in the Database.
- **minimum_positions** (*Dict[int]*) – A map of each TorsionDrive angle to the integer index of that angle's optimization trajectory which has the minimum-energy of the trajectory.

Fractal Client

A Client is the primary user interface to a Fractal server instance.

3.12 Portal Client

The `FractalClient` is the primary entry point to a `FractalServer` instance which can be initialized by pointing to a server instance:

```
>>> import qcportal as ptl
>>> client = ptl.FractalClient("localhost:8888")
>>> client
FractalClient(server='http://localhost:8888/', username='None')
```

The `FractalClient` handles all communications between `FractalServer` and the Python API layer. The `FractalClient` fascilitates the addition of new molecules to the datasets, performing computations, interacting with collections, and querying the records. A `FractalClient` object instance created by the default constructor (without any arguments) will automatically attempt to connect to the MolSSI QCArchive server.

The `FractalClient` can also be initialized using a YAML configuration file. This approach is useful because server address and username do not have to be retyped everytime the user initializes a server object. It will also reduce the chance for the username and password to be accidentally added to a version control system and exposed to the public. The above strategy adopts the classmethod `FractalClient.from_file()` which by default, searches for the YAML configuration file named `qcportal_config.yaml` under either the current working directory or the canonical `~/.qca` folder.

3.13 Add/Query Objects

Unlike the `Compute` object, the `Molecule`, `KeywordSet`, `Collection`, and `KVStore` objects are always added/queried directly to the server instances as these structures are not acted upon by the server itself.

3.13.1 Adding Objects to the Server

A list of objects can be added to the server via `client.add_*` commands which return the *ObjectId* of the object instance.

```
>>> helium = ptl.Molecule.from_data("He 0 0 0")
>>> data = client.add_molecules([helium])
['5b882c957b87878925ffaf22']
```

Any attempt to add the same molecule again will not proceed and the same *ObjectId* will always be returned:

```
>>> helium = ptl.Molecule.from_data("He 0 0 0")
>>> data = client.add_molecules([helium, helium])
['5b882c957b87878925ffaf22', '5b882c957b87878925ffaf22']
```

Note that the order of *ObjectId*'s are identical to the order of molecules in the input list.

Note: The *ObjectId* can change between object instances but it is unique within a particular database.

3.13.2 Querying Objects from the Server

Each server object has a set of fields that can be queried to obtain the object instances in addition to their *ObjectId*. All queries will return a list of objects.

3.13.3 Interacting with Molecules on the Server

As an example, we can use a molecule that comes with QCPortal and add it to the database. Please note that the Molecule ID (a *ObjectId*) shown below will not be the same as your result and will be unique for each database.

```
>>> hooh = ptl.Molecule.from_data("""
>>>        H          1.8486716127,  1.472346669,  0.644643566
>>>        O          1.3127881568, -0.130419379, -0.211892270
>>>        O         -1.3127927010,  0.133418733, -0.211896415
>>>        H         -1.8386801669, -1.482348324,  0.644636970
>>>        """)
>>> hooh
```

(continues on next page)

(continued from previous page)

```
    Geometry (in Angstrom), charge = 0.0, multiplicity = 1:

       Center              X                 Y                 Z
    ------------   -----------------  -----------------  -----------------
           H          0.977494197627     0.778135098208     0.428565624355
           O          0.694599115267    -0.068915578683    -0.027163830307
           O         -0.694920304666     0.069482110511    -0.026567833892
           H         -0.972396644160    -0.787126321701     0.424194864034


>>> data = client.add_molecules([hooh])
>>> data
['5c82c51895d5923b946989c1']
```

Molecules can either be queried from their Molecule ID or Molecule hash:

```
>>> client.query_molecules(molecule_hash=[hooh.get_hash()])[0].id
'5c82c51895d5923b946989c1'

>>> client.query_molecules(id=data)[0].id
'5c82c51895d5923b946989c1'
```

3.14 Records Querying

Query documents, including projects ideas.

3.15 New Compute Tasks

Add new compute tasks and checking status.

3.16 API

3.16.1 Generics

`__init__`([address, username, password, verify])	Initializes a FractalClient instance from an address and verification information.
from_file([load_path])	Creates a new FractalClient from file.
server_information()	Pull down various data on the connected server.

3.16.2 Add/Query Objects

query_kvstore(id[, full_return])	Queries items from the database's KVStore
query_molecules([id, molecule_hash, ...])	Queries molecules from the database.
add_molecules(mol_list[, full_return])	Adds molecules to the Server.
query_keywords([id, hash_index, limit, ...])	Obtains KeywordSets from the server using keyword ids.
add_keywords(keywords[, full_return])	Adds KeywordSets to the server.
list_collections([collection_type, aslist, ...])	Lists the available collections currently on the server.
get_collection(collection_type, name[, ...])	Acquires a given collection from the server.
add_collection(collection[, overwrite, ...])	Adds a new Collection to the server.

3.16.3 Records Querying

query_results([id, task_id, program, ...])	Queries ResultRecords from the server.
query_procedures([id, task_id, procedure, ...])	Queries Procedures from the server.

3.16.4 New Compute Tasks

add_compute([program, method, basis, ...])	Adds a "single" compute to the server.
add_procedure(procedure, program, ...[, ...])	Adds a "single" Procedure to the server.
add_service(service[, tag, priority, ...])	Adds a new service to the service queue.
query_tasks([id, hash_index, program, ...])	Checks the status of Tasks in the Fractal queue.
query_services([id, procedure_id, ...])	Checks the status of services in the Fractal queue.

3.16.5 Function Definitions

qcportal.FractalClient.**from_file**(*load_path: Optional[str] = None*) → qcfractal.interface.client.FractalClient

Creates a new FractalClient from file. If no path is passed in, the current working directory and ~.qca/ are searched for "qcportal_config.yaml"

Parameters **load_path** (*Optional[str], optional*) – Path to find "qcportal_config.yaml", the filename, or a dictionary containing keys {"address", "username", "password", "verify"}

Returns A new FractalClient from file.

Return type FractalClient

qcportal.FractalClient.**server_information**(*self*) → Dict[str, str]

Pull down various data on the connected server.

Returns Server information.

Return type Dict[str, str]

qcportal.FractalClient.**query_kvstore**(*self*, *id: QueryObjectId*, *full_return: bool = False*) → Dict[str, Any]

Queries items from the database's KVStore

Parameters

- **id** (*QueryObjectId*) – A list of KVStore id's

- **full_return** (*bool, optional*) – Returns the full server response if True that contains additional metadata.

Returns A list of found KVStore objects in {"id": "value"} format

Return type Dict[str, Any]

qcportal.FractalClient.**query_molecules**(*self*, *id: Optional[QueryObjectId] = None*, *molecule_hash: Optional[QueryStr] = None*, *molecular_formula: Optional[QueryStr] = None*, *limit: Optional[int] = None*, *skip: int = 0*, *full_return: bool = False*) → Union[MoleculeGETResponse, List[Molecule]]

Queries molecules from the database.

Parameters

- **id** (*QueryObjectId, optional*) – Queries the Molecule `id` field.
- **molecule_hash** (*QueryStr, optional*) – Queries the Molecule `molecule_hash` field.
- **molecular_formula** (*QueryStr, optional*) – Queries the Molecule `molecular_formula` field. Molecular formulas are case-sensitive. Molecular formulas are not order-sensitive (e.g. "H2O == OH2 != Oh2").
- **limit** (*Optional[int], optional*) – The maximum number of Molecules to query
- **skip** (*int, optional*) – The number of Molecules to skip in the query, used during pagination
- **full_return** (*bool, optional*) – Returns the full server response if True that contains additional metadata.

Returns A list of found molecules.

Return type List[*Molecule*]

qcportal.FractalClient.**add_molecules**(*self*, *mol_list: List[Molecule]*, *full_return: bool = False*) → List[str]

Adds molecules to the Server.

Parameters

- **mol_list** (*List[Molecule]*) – A list of Molecules to add to the server.
- **full_return** (*bool, optional*) – Returns the full server response if True that contains additional metadata.

Returns A list of Molecule id's in the sent order, can be None where issues occured.

Return type List[str]

qcportal.FractalClient.**query_keywords**(*self*, *id: Optional[QueryObjectId] = None*, ***, *hash_index: Optional[QueryStr] = None*, *limit: Optional[int] = None*, *skip: int = 0*, *full_return: bool = False*) → Union[KeywordGETResponse, List[KeywordSet]]

Obtains KeywordSets from the server using keyword ids.

Parameters

- **id** (*QueryObjectId, optional*) – A list of ids to query.
- **hash_index** (*QueryStr, optional*) – The hash index to look up
- **limit** (*Optional[int], optional*) – The maximum number of keywords to query
- **skip** (*int, optional*) – The number of keywords to skip in the query, used during pagination

- **full_return** (*bool, optional*) – Returns the full server response if True that contains additional metadata.

Returns The requested KeywordSet objects.

Return type List[*KeywordSet*]

qcportal.FractalClient.**add_keywords**(*self*, *keywords: List[KeywordSet]*, *full_return: bool = False*) → List[str]

Adds KeywordSets to the server.

Parameters

- **keywords** (*List[KeywordSet]*) – A list of KeywordSets to add.
- **full_return** (*bool, optional*) – Returns the full server response if True that contains additional metadata.

Returns A list of KeywordSet id's in the sent order, can be None where issues occured.

Return type List[str]

qcportal.FractalClient.**list_collections**(*self*, *collection_type: Optional[str] = None*, *aslist: bool = False*, *group: Optional[str] = 'default'*, *show_hidden: bool = False*, *tag: Optional[Union[List[str], str]] = None*) → pandas.core.frame.DataFrame

Lists the available collections currently on the server.

Parameters

- **collection_type** (*Optional[str], optional*) – If *None* all collection types will be returned, otherwise only the specified collection type will be returned
- **aslist** (*bool, optional*) – Returns a canonical list rather than a dataframe.
- **group** (*Optional[str], optional*) – Show only collections belonging to a specified group. To explicitly return all collections, set group=None
- **show_hidden** (*bool, optional*) – Show collections whose visibility flag is set to False. Default: False.
- **tag** (*Optional[Union[str, List[str]]], optional*) – Show collections whose tags match one of the passed tags. By default, collections are not filtered on tag.

Returns A dataframe containing the collection, name, and tagline.

Return type DataFrame

qcportal.FractalClient.**get_collection**(*self*, *collection_type: str*, *name: str*, *full_return: bool = False*, *include: QueryListStr = None*, *exclude: QueryListStr = None*) → Collection

Acquires a given collection from the server.

Parameters

- **collection_type** (*str*) – The collection type to be accessed
- **name** (*str*) – The name of the collection to be accessed
- **full_return** (*bool, optional*) – Returns the full server response if True that contains additional metadata.
- **include** (*QueryListStr, optional*) – Return only these columns.
- **exclude** (*QueryListStr, optional*) – Return all but these columns.

Returns A Collection object if the given collection was found otherwise returns *None*.

Return type Collection

qcportal.FractalClient.**add_collection**(*self*, *collection: Dict[str, Any]*, *overwrite: bool = False*, *full_return: bool = False*) → Union[CollectionGETResponse, List[ObjectId]]

Adds a new Collection to the server.

Parameters

- **collection** (*Dict[str, Any]*) – The full collection data representation.
- **overwrite** (*bool, optional*) – Overwrites the collection if it already exists in the database, used for updating collection.
- **full_return** (*bool, optional*) – Returns the full server response if True that contains additional metadata.

Returns The ObjectId's of the added collection.

Return type List[ObjectId]

qcportal.FractalClient.**query_results**(*self*, *id: Optional[QueryObjectId] = None*, *task_id: Optional[QueryObjectId] = None*, *program: Optional[QueryStr] = None*, *molecule: Optional[QueryObjectId] = None*, *driver: Optional[QueryStr] = None*, *method: Optional[QueryStr] = None*, *basis: Optional[QueryStr] = None*, *keywords: Optional[QueryObjectId] = None*, *status: QueryStr = 'COMPLETE'*, *limit: Optional[int] = None*, *skip: int = 0*, *include: Optional[QueryListStr] = None*, *full_return: bool = False*) → Union[ResultGETResponse, List[ResultRecord], Dict[str, Any]]

Queries ResultRecords from the server.

Parameters

- **id** (*QueryObjectId, optional*) – Queries the Result `id` field.
- **task_id** (*QueryObjectId, optional*) – Queries the Result `task_id` field.
- **program** (*QueryStr, optional*) – Queries the Result `program` field.
- **molecule** (*QueryObjectId, optional*) – Queries the Result `molecule` field.
- **driver** (*QueryStr, optional*) – Queries the Result `driver` field.
- **method** (*QueryStr, optional*) – Queries the Result `method` field.
- **basis** (*QueryStr, optional*) – Queries the Result `basis` field.
- **keywords** (*QueryObjectId, optional*) – Queries the Result `keywords` field.
- **status** (*QueryStr, optional*) – Queries the Result `status` field.
- **limit** (*Optional[int], optional*) – The maximum number of Results to query
- **skip** (*int, optional*) – The number of Results to skip in the query, used during pagination
- **include** (*QueryListStr, optional*) – Filters the returned fields, will return a dictionary rather than an object.
- **full_return** (*bool, optional*) – Returns the full server response if True that contains additional metadata.

Returns Returns a List of found RecordResult's without include, or a dictionary of results with include.

Return type Union[List[RecordResult], Dict[str, Any]]

qcportal.FractalClient.**query_procedures**(*self*, *id: Optional[QueryObjectId] = None*, *task_id: Optional[QueryObjectId] = None*, *procedure: Optional[QueryStr] = None*, *program: Optional[QueryStr] = None*, *hash_index: Optional[QueryStr] = None*, *status: QueryStr = 'COMPLETE'*, *limit: Optional[int] = None*, *skip: int = 0*, *include: Optional[QueryListStr] = None*, *full_return: bool = False*) → Union[ProcedureGETResponse, List[Dict[str, Any]]]

Queries Procedures from the server.

Parameters

- **id** (*QueryObjectId, optional*) – Queries the Procedure `id` field.
- **task_id** (*QueryObjectId, optional*) – Queries the Procedure `task_id` field.
- **procedure** (*QueryStr, optional*) – Queries the Procedure `procedure` field.
- **program** (*QueryStr, optional*) – Queries the Procedure `program` field.
- **hash_index** (*QueryStr, optional*) – Queries the Procedure `hash_index` field.
- **status** (*QueryStr, optional*) – Queries the Procedure `status` field.
- **limit** (*Optional[int], optional*) – The maximum number of Procedures to query
- **skip** (*int, optional*) – The number of Procedures to skip in the query, used during pagination
- **include** (*QueryListStr, optional*) – Filters the returned fields, will return a dictionary rather than an object.
- **full_return** (*bool, optional*) – Returns the full server response if True that contains additional metadata.

Returns Returns a List of found RecordResult's without include, or a dictionary of results with include.

Return type Union[List['RecordBase'], Dict[str, Any]]

qcportal.FractalClient.**add_compute**(*self*, *program: str = None*, *method: str = None*, *basis: Optional[str] = None*, *driver: str = None*, *keywords: Optional[ObjectId] = None*, *molecule: Union[ObjectId, Molecule, List[Union[ObjectId, Molecule]]] = None*, ***, *priority: Optional[str] = None*, *protocols: Optional[Dict[str, Any]] = None*, *tag: Optional[str] = None*, *full_return: bool = False*) → ComputeResponse

Adds a "single" compute to the server.

Parameters

- **program** (*str, optional*) – The computational program to execute the result with (e.g., "rdkit", "psi4").
- **method** (*str, optional*) – The computational method to use (e.g., "B3LYP", "PBE")
- **basis** (*Optional[str], optional*) – The basis to apply to the computation (e.g., "cc-pVDZ", "6-31G")
- **driver** (*str, optional*) – The primary result that the compute will aquire {"energy", "gradient", "hessian", "properties"}
- **keywords** (*Optional['ObjectId'], optional*) – The KeywordSet ObjectId to use with the given compute

- **molecule** (*Union['ObjectId', 'Molecule', List[Union['ObjectId', 'Molecule']]], optional*) – The Molecules or Molecule ObjectId's to compute with the above methods
- **priority** (*Optional[str], optional*) – The priority of the job {"HIGH", "MEDIUM", "LOW"}. Default is "MEDIUM".
- **protocols** (*Optional[Dict[str, Any]], optional*) – Protocols for store more or less data per field. Current valid protocols: {'wavefunction'}
- **tag** (*Optional[str], optional*) – The computational tag to add to your compute, managers can optionally only pull based off the string tags. These tags are arbitrary, but several examples are to use "large", "medium", "small" to denote the size of the job or "project1", "project2" to denote different projects.
- **full_return** (*bool, optional*) – Returns the full server response if True that contains additional metadata.

Returns

An object that contains the submitted ObjectIds of the new compute. This object has the following fields:

- ids: The ObjectId's of the task in the order of input molecules
- submitted: A list of ObjectId's that were submitted to the compute queue
- existing: A list of ObjectId's of tasks already in the database

Return type ComputeResponse

Raises `ValueError` – Description

qcportal.FractalClient.**add_procedure**(*self*, *procedure: str*, *program: str*, *program_options: Dict[str, Any]*, *molecule: Union[ObjectId, Molecule, List[Union[str, Molecule]]]*, *priority: Optional[str] = None*, *tag: Optional[str] = None*, *full_return: bool = False*) → ComputeResponse

Adds a "single" Procedure to the server.

Parameters

- **procedure** (*str*) – The computational procedure to spawn {"optimization"}
- **program** (*str*) – The program to use for the given procedure (e.g., "geomeTRIC")
- **program_options** (*Dict[str, Any]*) – Additional options and specifications for the given procedure.
- **molecule** (*Union[ObjectId, Molecule, List[Union[str, Molecule]]]*) – The Molecules or Molecule ObjectId's to use with the above procedure
- **priority** (*str, optional*) – The priority of the job {"HIGH", "MEDIUM", "LOW"}. Default is "MEDIUM".
- **tag** (*str, optional*) – The computational tag to add to your procedure, managers can optionally only pull based off the string tags. These tags are arbitrary, but several examples are to use "large", "medium", "small" to denote the size of the job or "project1", "project2" to denote different projects.
- **full_return** (*bool, optional*) – Returns the full server response if True that contains additional metadata.

Returns

An object that contains the submitted ObjectIds of the new procedure. This object has the following fields:

- ids: The ObjectId's of the task in the order of input molecules
- submitted: A list of ObjectId's that were submitted to the compute queue
- existing: A list of ObjectId's of tasks already in the database

Return type ComputeResponse

qcportal.FractalClient.**query_tasks**(*self*, *id: Optional[QueryObjectId] = None*, *hash_index: Optional[QueryStr] = None*, *program: Optional[QueryStr] = None*, *status: Optional[QueryStr] = None*, *base_result: Optional[QueryStr] = None*, *tag: Optional[QueryStr] = None*, *manager: Optional[QueryStr] = None*, *limit: Optional[int] = None*, *skip: int = 0*, *include: Optional[QueryListStr] = None*, *full_return: bool = False*) → Union[TaskQueueGETResponse, List[TaskRecord], List[Dict[str, Any]]]

Checks the status of Tasks in the Fractal queue.

Parameters

- **id** (*QueryObjectId, optional*) – Queries the Tasks `id` field.
- **hash_index** (*QueryStr, optional*) – Queries the Tasks `hash_index` field.
- **program** (*QueryStr, optional*) – Queries the Tasks `program` field.
- **status** (*QueryStr, optional*) – Queries the Tasks `status` field.
- **base_result** (*QueryStr, optional*) – Queries the Tasks `base_result` field.
- **tag** (*QueryStr, optional*) – Queries the Tasks `tag` field.
- **manager** (*QueryStr, optional*) – Queries the Tasks `manager` field.
- **limit** (*Optional[int], optional*) – The maximum number of Tasks to query
- **skip** (*int, optional*) – The number of Tasks to skip in the query, used during pagination
- **include** (*QueryListStr, optional*) – Filters the returned fields, will return a dictionary rather than an object.
- **full_return** (*bool, optional*) – Returns the full server response if True that contains additional metadata.

Returns A dictionary of each match that contains the current status and, if an error has occurred, the error message.

Return type List[Dict[str, Any]]

Examples

```
>>> client.query_tasks(id="5bd35af47b878715165f8225",include=["status"])
[{"status": "WAITING"}]
```

qcportal.FractalClient.**add_service**(*self*, *service: Union[List[GridOptimizationInput], List[TorsionDriveInput]]*, *tag: Optional[str] = None*, *priority: Optional[str] = None*, *full_return: bool = False*) → ComputeResponse

Adds a new service to the service queue.

Parameters

- **service** (*Union[GridOptimizationInput, TorsionDriveInput]*) – An available service input

- **tag** (*Optional[str], optional*) – The compute tag to add the service under.
- **priority** (*Optional[str], optional*) – The priority of the job within the compute queue.
- **full_return** (*bool, optional*) – Returns the full server response if True that contains additional metadata.

Returns

An object that contains the submitted ObjectIds of the new service. This object has the following fields:

- ids: The ObjectId's of the task in the order of input molecules
- submitted: A list of ObjectId's that were submitted to the compute queue
- existing: A list of ObjectId's of tasks already in the database

Return type ComputeResponse

qcportal.FractalClient.**query_services**(*self*, *id: Optional[QueryObjectId] = None*, *procedure_id: Optional[QueryObjectId] = None*, *hash_index: Optional[QueryStr] = None*, *status: Optional[QueryStr] = None*, *limit: Optional[int] = None*, *skip: int = 0*, *full_return: bool = False*) → Union[ServiceQueueGETResponse, List[Dict[str, Any]]]

Checks the status of services in the Fractal queue.

Parameters

- **id** (*QueryObjectId, optional*) – Queries the Services `id` field.
- **procedure_id** (*QueryObjectId, optional*) – Queries the Services `procedure_id` field, or the ObjectId of the procedure associated with the service.
- **hash_index** (*QueryStr, optional*) – Queries the Services `procedure_id` field.
- **status** (*QueryStr, optional*) – Queries the Services `status` field.
- **limit** (*Optional[int], optional*) – The maximum number of Services to query
- **skip** (*int, optional*) – The number of Services to skip in the query, used during pagination
- **full_return** (*bool, optional*) – Returns the full server response if True that contains additional metadata.

Returns A dictionary of each match that contains the current status and, if an error has occurred, the error message.

Return type List[Dict[str, Any]]

Developer Documentation

Contains in-depth developer documentation.

3.17 Glossary

DB Index A DB Index (or Database Index) is a commonly queried field used to speed up searches in a *DB Table*.

DB Socket A DB Socket (or Database Socket) is the interface layer between standard Python queries and raw SQL or MongoDB query language.

DB Table A set of data inside the Database which has a common *ObjectId*. The `table` name follows SQL conventions which is also known as a `collection` in MongoDB.

Hash Index A index that hashes the information contained in the object in a reproducible manner. This hash index is only used to find duplicates and should not be relied upon as it may change in the future.

Molecule A unique 3D representation of a molecule. Any changes to the protonation state, multiplicity, charge, fragments, coordinates, connectivity, isotope, or ghost atoms represent a change in the molecule.

ObjectId A ObjectId (or Database ID) is a unique ID for a given row (a document or entry) in the database that uniquely defines that particular row in a *DB Table*. These rows are automatically generated and will be different for every database, but outlines ways to reference other rows in the database quickly. A ObjectId is unique to a DB Table.

Procedures On-node computations, these can either be a single computation (energy, gradient, property, etc.) or a series of calculations such as a geometry optimization.

Queue Adapter The interface between QCFractal's internal queue representation and other queueing systems such as Dask or Fireworks.

Record A document that contains all results (or links) of a given computation.

Services Iterative workflows where the required computations are distributed via the queue and then are processed on the server to acquire the next iteration of calculations.

3.18 REST API

The items in this list document the REST API calls which can be made against the server, this includes both the Body and the Responses for the various GET, POST, and PUT calls.

The entries are organized such that the API is presented first, separated by objects. The last group of entries are common models which are *parts* of the API Bodies and Responses (like Metadata), but occur many times in the normal calls.

3.18.1 KV Store

class qcportal.models.rest_models.**KVStoreGETBody**(*, *meta:* qcportal.models.rest_models.EmptyMeta = *{}*, *data: qcportal.models.rest_models.KVStoreGETBody.Data*)

Parameters

- **meta** (*EmptyMeta*, Default: {}) – There is no metadata accepted, so an empty metadata is sent for completion.
- **data** (Data) – Data of the KV Get field: consists of Id of the Key/Value object to fetch.

class qcportal.models.rest_models.**KVStoreGETResponse**(*, *meta:* qcportal.models.rest_models.ResponseGETMeta, *data: Dict[str, qcportal.models.common_models.KVStore]*)

Parameters

- **meta** (*ResponseGETMeta*) – Standard Fractal Server response metadata for GET/fetch type requests.
- **data** (KVStore) – The entries of Key/Value object requested.

3.18.2 Molecule

class qcportal.models.rest_models.**MoleculeGETBody**(*, *meta:* qcportal.models.rest_models.QueryMeta = *QueryMeta(limit=None, skip=0)*, *data: qcportal.models.rest_models.MoleculeGETBody.Data*)

Parameters

- **meta** (*QueryMeta*, Optional) – Standard Fractal Server metadata for Database queries containing pagination information
- **data** (Data) – Data fields for a Molecule query.

class qcportal.models.rest_models.**MoleculeGETResponse**(*, *meta:* qcportal.models.rest_models.ResponseGETMeta, *data: List[*qcelemental.models.molecule.Molecule*]*)

Parameters

- **meta** (*ResponseGETMeta*) – Standard Fractal Server response metadata for GET/fetch type requests.
- **data** (Molecule) – The List of Molecule objects found by the query.

class qcportal.models.rest_models.**MoleculePOSTBody**(*, *meta:* qcportal.models.rest_models.EmptyMeta = *{}*, *data: List[*qcelemental.models.molecule.Molecule*]*)

Parameters

- **meta** (*EmptyMeta*, Default: {}) – There is no metadata accepted, so an empty metadata is sent for completion.
- **data** (Molecule) – A list of Molecule objects to add to the Database.

class qcportal.models.rest_models.**MoleculePOSTResponse**(*, *meta:* qcportal.models.rest_models.ResponsePOSTMeta, *data: List[qcportal.models.common_models.ObjectId]*)

Parameters

- **meta** (*ResponsePOSTMeta*) – Standard Fractal Server response metadata for POST/add type requests.

- **data** (*List[ObjectId]*) – A list of Id's assigned to the Molecule objects passed in which serves as a unique identifier in the database. If the Molecule was already in the database, then the Id returned is its existing Id (entries are not duplicated).

3.18.3 Keywords

class qcportal.models.rest_models.**KeywordGETBody**(*, *meta:* qcportal.models.rest_models.QueryMeta = *QueryMeta(limit=None, skip=0)*, *data: qcportal.models.rest_models.KeywordGETBody.Data*)

Parameters

- **meta** (*QueryMeta*, Optional) – Standard Fractal Server metadata for Database queries containing pagination information
- **data** (Data) – The formal query for a Keyword fetch, contains `id` or `hash_index` for the object to fetch.

class qcportal.models.rest_models.**KeywordGETResponse**(*, *meta:* qcportal.models.rest_models.ResponseGETMeta, *data:* *List[*qcportal.models.common_models.KeywordSet*]*)

Parameters

- **meta** (*ResponseGETMeta*) – Standard Fractal Server response metadata for GET/fetch type requests.
- **data** (KeywordSet) – The `KeywordSet` found from in the database based on the query.

class qcportal.models.rest_models.**KeywordPOSTBody**(*, *meta:* qcportal.models.rest_models.EmptyMeta = *{}*, *data:* *List[*qcportal.models.common_models.KeywordSet*]*)

Parameters

- **meta** (*EmptyMeta*, Default: {}) – There is no metadata with this, so an empty metadata is sent for completion.
- **data** (KeywordSet) – The list of `KeywordSet` objects to add to the database.

class qcportal.models.rest_models.**KeywordPOSTResponse**(*, *data:* *List[Optional[qcportal.models.common_models.ObjectId]]*, *meta:* qcportal.models.rest_models.ResponsePOSTMeta)

Parameters

- **data** (*List[ObjectId]*) – The Ids assigned to the added `KeywordSet` objects. In the event of duplicates, the Id will be the one already found in the database.
- **meta** (*ResponsePOSTMeta*) – Standard Fractal Server response metadata for POST/add type requests.

3.18.4 Collections

class qcportal.models.rest_models.**CollectionGETBody**(*, *meta: qcportal.models.rest_models.QueryFilter = None*, *data: qcportal.models.rest_models.CollectionGETBody.Data*)

Parameters

- **meta** (`QueryFilter`, Optional) – Additional metadata to make with the query. Collections can only have an `include/exclude` key in its meta and therefore does not follow the standard GET metadata model.
- **data** (`Data`) – Information about the Collection to search the database with.

class qcportal.models.rest_models.**CollectionGETResponse**(*, *meta:* qcportal.models.rest_models.ResponseGETMeta, *data: List[Dict[str, Optional[Any]]]*)

Parameters

- **meta** (*ResponseGETMeta*) – Standard Fractal Server response metadata for GET/fetch type requests.
- **data** (*List[Dict[Any]]*) – The Collection objects returned by the server based on the query.

class qcportal.models.rest_models.**CollectionPOSTBody**(*, *meta: qcportal.models.rest_models.CollectionPOSTBody.Meta = Meta(overwrite=False)*, *data: qcportal.models.rest_models.CollectionPOSTBody.Data*)

Parameters

- **meta** (`Meta`, Optional) – Metadata to specify how the Database should handle adding this Collection if it already exists. Metadata model for adding Collections can only accept `overwrite` as a key to choose to update existing Collections or not.
- **data** (`Data`) – The data associated with this Collection to add to the database.

class qcportal.models.rest_models.**CollectionPOSTResponse**(*, *data: str*, *meta:* qcportal.models.rest_models.ResponsePOSTMeta)

Parameters

- **data** (*str*) – The Id of the Collection uniquely pointing to it in the Database. If the Collection was not added (e.g. `overwrite=False` for existing Collection), then a None is returned.
- **meta** (*ResponsePOSTMeta*) – Standard Fractal Server response metadata for POST/add type requests.

3.18.5 Result

class qcportal.models.rest_models.**ResultGETBody**(*, *meta: qcportal.models.rest_models.QueryMetaFilter = QueryMetaFilter(include=None, exclude=None, limit=None, skip=0)*, *data: qcportal.models.rest_models.ResultGETBody.Data*)

Parameters

- **meta** (QueryMetaFilter, Optional) – Fractal Server metadata for Database queries allowing for filtering and pagination
- **data** (Data) – The keys with data to search the database on for individual quantum chemistry computations.

class qcportal.models.rest_models.**ResultGETResponse**(*, *meta:* qcportal.models.rest_models.ResponseGETMeta, *data: Union[List[*qcportal.models.records.ResultRecord*], List[Dict[str, Any]]]*)

Parameters

- **meta** (*ResponseGETMeta*) – Standard Fractal Server response metadata for GET/fetch type requests.
- **data** (Union[ResultRecord, List[Dict[Any]]]) – Results found from the query. This is a list of ResultRecord in most cases, however, if a projection was specified in the GET request, then a dict is returned with mappings based on the projection.

3.18.6 Procedures

class qcportal.models.rest_models.**ProcedureGETBody**(*, *meta: qcportal.models.rest_models.QueryMetaFilter = QueryMetaFilter(include=None, exclude=None, limit=None, skip=0)*, *data:* qcportal.models.rest_models.ProcedureGETBody.Data)

Parameters

- **meta** (QueryMetaFilter, Optional) – Fractal Server metadata for Database queries allowing for filtering and pagination
- **data** (Data) – The keys with data to search the database on for Procedures.

class qcportal.models.rest_models.**ProcedureGETResponse**(*, *meta:* qcportal.models.rest_models.ResponseGETMeta, *data: List[Dict[str, Optional[Any]]]*)

Parameters

- **meta** (*ResponseGETMeta*) – Standard Fractal Server response metadata for GET/fetch type requests.
- **data** (*List[Dict[Any]]*) – The list of Procedure specs found based on the query.

3.18.7 Task Queue

class qcportal.models.rest_models.**TaskQueueGETBody**(*, *meta: qcportal.models.rest_models.QueryMetaFilter = QueryMetaFilter(include=None, exclude=None, limit=None, skip=0)*, *data: qcportal.models.rest_models.TaskQueueGETBody.Data*)

Parameters

- **meta** (QueryMetaFilter, Optional) – Fractal Server metadata for Database queries allowing for filtering and pagination
- **data** (Data) – The keys with data to search the database on for Tasks.

class qcportal.models.rest_models.**TaskQueueGETResponse**(*, *meta:* qcportal.models.rest_models.ResponseGETMeta, *data: Union[List[qcportal.models.task_models.TaskRecord], List[Dict[str, Any]]]*)

Parameters

- **meta** (*ResponseGETMeta*) – Standard Fractal Server response metadata for GET/fetch type requests.
- **data** (Union[TaskRecord, List[Dict[Any]]]) – Tasks found from the query. This is a list of TaskRecord in most cases, however, if a projection was specified in the GET request, then a dict is returned with mappings based on the projection.

class qcportal.models.rest_models.**TaskQueuePOSTBody**(*, *meta: qcportal.models.rest_models.TaskQueuePOSTBody.Meta*, *data: List[Union[qcportal.models.common_models.ObjectId,* qcelemental.models.molecule.Molecule*]]*)

Parameters

- **meta** (Meta) – The additional specification information for the Task to add to the Database.
- **data** (List[Union[ObjectId, Molecule]]) – The list of either Molecule objects or Molecule Id's (those already in the database) to submit as part of this Task.

class qcportal.models.rest_models.**TaskQueuePOSTResponse**(*, *meta:* qcportal.models.rest_models.ResponsePOSTMeta, *data: qcportal.models.rest_models.ComputeResponse*)

Parameters

- **meta** (*ResponsePOSTMeta*) – Standard Fractal Server response metadata for POST/add type requests.
- **data** (ComputeResponse) – Data returned from the server from adding a Task.

class qcportal.models.rest_models.**TaskQueuePUTBody**(*, *meta: qcportal.models.rest_models.TaskQueuePUTBody.Meta*, *data: qcportal.models.rest_models.TaskQueuePUTBody.Data*)

Parameters

- **meta** (Meta) – The instructions to pass to the target Task from data.
- **data** (Data) – The information which contains the Task target in the database.

class qcportal.models.rest_models.**TaskQueuePUTResponse**(*, *meta:* qcportal.models.rest_models.ResponseMeta, *data: qcportal.models.rest_models.TaskQueuePUTResponse.Data*)

Parameters

- **meta** (*ResponseMeta*) – Standard Fractal Server response metadata
- **data** (Data) – Information returned from attempting updates of Tasks.

3.18.8 Service Queue

class qcportal.models.rest_models.**ServiceQueueGETBody**(*, *meta:* qcportal.models.rest_models.QueryMeta = *QueryMeta(limit=None, skip=0)*, *data: qcportal.models.rest_models.ServiceQueueGETBody.Data*)

Parameters

- **meta** (*QueryMeta*, Optional) – Standard Fractal Server metadata for Database queries containing pagination information
- **data** (Data) – The keys with data to search the database on for Services.

class qcportal.models.rest_models.**ServiceQueueGETResponse**(*, *meta:* qcportal.models.rest_models.ResponseGETMeta, *data: List[Dict[str, Optional[Any]]]*)

Parameters

- **meta** (*ResponseGETMeta*) – Standard Fractal Server response metadata for GET/fetch type requests.
- **data** (*List[Dict[Any]]*) – The return of Services found in the database mapping their Ids to the Service spec.

class qcportal.models.rest_models.**ServiceQueuePOSTBody**(*, *meta: qcportal.models.rest_models.ServiceQueuePOSTBody.Meta*, *data: List[Union[*qcportal.models.torsiondrive.TorsionDriveInput, qcportal.models.gridoptimization.GridOptimizationInput*]]*)

Parameters

- **meta** (Meta) – Metadata information for the Service for the Tag and Priority of Tasks this Service will create.
- **data** (List[Union[TorsionDriveInput, GridOptimizationInput]]) – A list the specification for Procedures this Service will manage and generate Tasks for.

class qcportal.models.rest_models.**ServiceQueuePOSTResponse**(*, *meta:* qcportal.models.rest_models.ResponsePOSTMeta, *data: qcportal.models.rest_models.ComputeResponse*)

Parameters

- **meta** (*ResponsePOSTMeta*) – Standard Fractal Server response metadata for POST/add type requests.
- **data** (ComputeResponse) – Data returned from the server from adding a Service.

class qcportal.models.rest_models.**ServiceQueuePUTBody**(*, *meta: qcportal.models.rest_models.ServiceQueuePUTBody.Meta*, *data: qcportal.models.rest_models.ServiceQueuePUTBody.Data*)

Parameters

- **meta** (Meta) – The instructions to pass to the targeted Service.
- **data** (Data) – The information which contains the Service target in the database.

class qcportal.models.rest_models.**ServiceQueuePUTResponse**(*, *meta:* qcportal.models.rest_models.ResponseMeta, *data: qcportal.models.rest_models.ServiceQueuePUTResponse.Data*)

Parameters

- **meta** (*ResponseMeta*) – Standard Fractal Server response metadata
- **data** (Data) – Information returned from attempting updates of Services.

3.18.9 Queue Manager

class qcportal.models.rest_models.**QueueManagerGETBody**(*, *meta:* qcportal.models.rest_models.QueueManagerMeta, *data: qcportal.models.rest_models.QueueManagerGETBody.Data*)

Parameters

- **meta** (*QueueManagerMeta*) – Validation and identification Meta information for the Queue Manager's communication with the Fractal Server.
- **data** (Data) – A model of Task request data for the Queue Manager to fetch. Accepts limit as the maximum number of tasks to pull.

class qcportal.models.rest_models.**QueueManagerGETResponse**(*, *meta:* qcportal.models.rest_models.ResponseGETMeta, *data: List[Dict[str, Optional[Any]]]*)

Parameters

- **meta** (*ResponseGETMeta*) – Standard Fractal Server response metadata for GET/fetch type requests.
- **data** (*List[Dict[Any]]*) – A list of tasks retrieved from the server to compute.

class qcportal.models.rest_models.**QueueManagerPOSTBody**(*, *meta:* qcportal.models.rest_models.QueueManagerMeta, *data: Dict[qcportal.models.common_models.ObjectId, Any]*)

Parameters

- **meta** (*QueueManagerMeta*) – Validation and identification Meta information for the Queue Manager's communication with the Fractal Server.
- **data** (*Dict[Any]*) – A Dictionary of tasks to return to the server.

class qcportal.models.rest_models.**QueueManagerPOSTResponse**(*, *meta:* qcportal.models.rest_models.ResponsePOSTMeta, *data: bool*)

Parameters

- **meta** (*ResponsePOSTMeta*) – Standard Fractal Server response metadata for POST/add type requests.
- **data** (*bool*) – A True/False return on if the server accepted the returned tasks.

class qcportal.models.rest_models.**QueueManagerPUTBody**(*, *meta:* qcportal.models.rest_models.QueueManagerMeta, *data: qcportal.models.rest_models.QueueManagerPUTBody.Data*)

Parameters

- **meta** (*QueueManagerMeta*) – Validation and identification Meta information for the Queue Manager's communication with the Fractal Server.
- **data** (`Data`) – The update action which the Queue Manager requests the Server take with respect to how the Queue Manager is tracked.

class qcportal.models.rest_models.**QueueManagerPUTResponse**(*, *meta: Dict[str, Any] = {}*, *data: Union[Dict[str, int], bool]*)

Parameters

- **meta** (*Dict[Any], Default: {}*) – There is no metadata accepted, so an empty metadata is sent for completion.

- **data** (*Union[Dict[int], bool]*) – The response from the Server attempting to update the Queue Manager's server-side status. Response type is a function of the operation made from the PUT request.

3.18.10 Common REST Components

These are NOT complete Body or Responses to the REST API, but common fragments which make up things like the Metadata or the Data fields.

class qcportal.models.rest_models.**EmptyMeta**

There is no metadata accepted, so an empty metadata is sent for completion.

class qcportal.models.rest_models.**ResponseMeta**(*, *errors: List[Tuple[str, str]]*, *success: bool*, *error_description: Union[str, bool]*)

Standard Fractal Server response metadata

Parameters

- **errors** (*List[Tuple[str, str]]*) – A list of error pairs in the form of [(error type, error message), ...]
- **success** (*bool*) – Indicates if the passed information was successful in its duties. This is contextual to the data being passed in.
- **error_description** (*Union[str, bool]*) – Details about the error if `success` is `False`, otherwise this is `False` in the event of no errors.

class qcportal.models.rest_models.**ResponseGETMeta**(*, *errors: List[Tuple[str, str]]*, *success: bool*, *error_description: Union[str, bool]*, *missing: List[str]*, *n_found: int*)

Standard Fractal Server response metadata for GET/fetch type requests.

Parameters

- **errors** (*List[Tuple[str, str]]*) – A list of error pairs in the form of [(error type, error message), ...]
- **success** (*bool*) – Indicates if the passed information was successful in its duties. This is contextual to the data being passed in.
- **error_description** (*Union[str, bool]*) – Details about the error if `success` is `False`, otherwise this is `False` in the event of no errors.
- **missing** (*List[str]*) – The Id's of the objects which were not found in the database.
- **n_found** (*int*) – The number of entries which were already found in the database from the set which was provided.

class qcportal.models.rest_models.**ResponsePOSTMeta**(*, *errors: List[Tuple[str, str]]*, *success: bool*, *error_description: Union[str, bool]*, *n_inserted: int*, *duplicates: Union[List[str], List[Tuple[str, str]]]*, *validation_errors: List[str]*)

Standard Fractal Server response metadata for POST/add type requests.

Parameters

- **errors** (*List[Tuple[str, str]]*) – A list of error pairs in the form of [(error type, error message), ...]
- **success** (*bool*) – Indicates if the passed information was successful in its duties. This is contextual to the data being passed in.

- **error_description** (*Union[str, bool]*) – Details about the error if `success` is `False`, otherwise this is `False` in the event of no errors.
- **n_inserted** (*int*) – The number of new objects amongst the inputs which did not exist already, and are now in the database.
- **duplicates** (*Union[List[str], List[Tuple[str, str]]]*) – The Ids of the objects which already exist in the database amongst the set which were passed in.
- **validation_errors** (*List[str]*) – All errors with validating submitted objects will be documented here.

class qcportal.models.rest_models.**QueryMeta**(*, *limit: int = None*, *skip: int = 0*)

Standard Fractal Server metadata for Database queries containing pagination information

Parameters

- **limit** (*int, Optional*) – Limit to the number of objects which can be returned with this query.
- **skip** (*int, Default: 0*) – The number of records to skip on the query.

class qcportal.models.rest_models.**QueueManagerMeta**(*, *cluster: str*, *hostname: str*, *uuid: str*, *username: str = None*, *qcengine_version: str*, *manager_version: str*, *programs: List[str]*, *procedures: List[str]*, *tag: Optional[Union[str, List[str]]] = None*, *total_worker_walltime: float = None*, *total_task_walltime: float = None*, *active_tasks: int = None*, *active_cores: int = None*, *active_memory: float = None*)

Validation and identification Meta information for the Queue Manager's communication with the Fractal Server.

Parameters

- **cluster** (*str*) – The Name of the Cluster the Queue Manager is running on.
- **hostname** (*str*) – Hostname of the machine the Queue Manager is running on.
- **uuid** (*str*) – A UUID assigned to the QueueManager to uniquely identify it.
- **username** (*str, Optional*) – Fractal Username the Manager is being executed under.
- **qcengine_version** (*str*) – Version of QCEngine which the Manager has access to.
- **manager_version** (*str*) – Version of the QueueManager (Fractal) which is getting and returning Jobs.
- **programs** (*List[str]*) – A list of programs which the QueueManager, and thus QCEngine, has access to. Affects which Tasks the Manager can pull.
- **procedures** (*List[str]*) – A list of procedures which the QueueManager has access to. Affects which Tasks the Manager can pull.
- **tag** (*Union[str, List[str]], Optional*) – Optional queue tag to pull Tasks from. If None, tasks are pulled from all tags. If a list of tags is provided, tasks are pulled in order of tags. (This does not guarantee tasks will be executed in that order, however.)
- **total_worker_walltime** (*float, Optional*) – The total worker walltime in core-hours.
- **total_task_walltime** (*float, Optional*) – The total task walltime in core-hours.
- **active_tasks** (*int, Optional*) – The total number of active running tasks.
- **active_cores** (*int, Optional*) – The total number of active cores.
- **active_memory** (*float, Optional*) – The total amount of active memory in GB.

3.19 Changelog

3.19.1 0.15.6 / 2021-06-06

Some minor fixes and additions to the user interface

- (GH#672) Adds ability to add compute specs to only a subset of entries in a Dataset
- (GH#673) Allow for selecting by status in dataset get_records()
- (GH#678) Fixes errors related to str vs. bytes in collection views
- (GH#679) Fix incorrect status reporting in collections

3.19.2 0.15.3 / 2021-03-15

Similar to the previous release, this this release focused on core QCFractal issues. The previous release should be compatible with newer QCFractal instances.

3.19.3 0.15.0 / 2020-11-11

Similar to the previous release, this this release focused on core QCFractal issues. The previous release should be compatible with newer QCFractal instances.

Bug Fixes

- (GH#626) Fix printing of client version during version check failure
- (GH#638) Fix incorrect error in datasets

3.19.4 0.14.0 / 2020-09-30

This release focused on core QCFractal issues and therefore QCPortal was not modified much, other than a few minor bugfixes.

New Features

- (GH#597) Add ability to query managers
- (GH#612) Adds KVStore compression, which is reflected in a change to `get_kvstore`

Bug Fixes

- (GH#586) Fixed an issue with status() with collections

3.19.5 0.13.1 / 2020-02-18

New Features

- (GH#566) A `list_keywords` function was added to `Dataset`.

Enhancements

- (GH#547, GH#553) Miscellaneous documentation edits and improvements.
- (GH#556) Molecule queries filtered on molecular formula no longer depend on the order of elements.
- (GH#565) `query` method for `Datasets` now returns collected records.

Bug Fixes

- (GH#559) Fixed an issue where Docker images did not have qcfractal in their PATH.
- (GH#561) Fixed a bug that caused errors with pandas v1.0.
- (GH#564) Fixes a bug where optimization protocols were not respected during torsiondrives and grid optimizations.

3.19.6 0.13.0 / 2020-01-15

New Features

Enhancements

- (GH#507) Automatically adds collection molecules in chunks if more than the current limit needs to be submitted.
- (GH#515) Conda environments now correspond to docker images in all deployed cases.
- (GH#524) The `delete_collection` function was added to `qcportal.FractalClient`.
- (GH#535) Allows dftd3 to be computed for all stoichiometries rather than just defaults.

Bug Fixes

- (GH#506) Fixes repeated visualize calls where previously the visualize call would corrupt local state.
- (GH#522) Fixes a bug where `ProcedureDataset.status()` failed for specifications where only a subset was computed.
- (GH#525) This PR fixes ENTRYPOINT of the qcarchive_worker_openff worker. (Conda and Docker are not friends.)
- (GH#543) Fixes a bug where `qcfractal-server` "start" before an "upgrade" prevented the "upgrade" command from correctly running.
- (GH#545) Fixed an issue in Dataset.get_records() that could occur when the optional arguments keywords and basis were not provided.

3.19.7 0.12.2 / 2019-12-07

Enhancements

- (GH#477) Removes 0.12.x xfails when connecting to the server.
- (GH#481) Expands Parsl Manager Adapter to include ALCF requirements.
- (GH#483) Dataset Views are now much faster to load in HDF5.
- (GH#488) Allows gzipped dataset views.
- (GH#490) Computes checksums on gzipped dataset views.

Bug Fixes

- (GH#486) Fixes pydantic `__repr__` issues after update.
- (GH#492) Fixes error where `ReactionDataset` didn't allow a minimum number of n-body expansion to be added.
- (GH#493) Fixes an issue with `ReactionDataset.get_molecules` when a subset is present.
- (GH#494) Fixes an issue where queries with `limit=0` erroneously returned all results.
- (GH#496) TorsionDrive tests now avoid 90 degree angles with RDKit to avoid some linear issues in the forcefield and make them more stable.
- (GH#497) `TorsionDrive.get_history` now works for extremely large (1000+) optimizations in the procedure.

3.19.8 0.12.1 / 2019-11-08

Enhancements

- (GH#472) Update to GitHub ISSUE templates.
- (GH#473) Server `/information` endpoint now contains the number of records for molecules, results, procedures, and collections.
- (GH#474) Dataset Views can now be of arbitrary shape.
- (GH#475) Changes the default formatting of the codebase to Black.

Bug Fixes

- (GH#470) Dataset fix for non-energy units.

3.19.9 0.12.0 / 2019-10-01

New Features

- (GH#433) `Dataset` and `ReactionDataset` (`interface.collections`) now have a `download` method which downloads a frozen view of the dataset. This view is used to speed up calls to `get_values`, `get_molecules`, `get_entries`, and `list_values`.
- (GH#440) Wavefunctions can now be stored in the database using Result `protocols`.

Enhancements

- (GH#429) Enables protocols for `OptimizationDataset` collections.
- (GH#430) Adds additional QCPortal type hints.
- (GH#433, GH#443) `Dataset` and `ReactionDataset` (`interface.collections`) are now faster for calls to calls to `get_values`, `get_molecules`, `get_entries`, and `list_values` for large datasets if the server is configured to use frozen views. See "Server-side Dataset Views" documentation. Subsets may be passed to `get_values`, `get_molecules`, and `get_entries`
- (GH#447) Enables the creation of plaintext (xyz and csv) output from Dataset Collections.
- (GH#458) Collections now have a metadata field.
- (GH#462) Dataset downloads now have a TQDM progress bar.
- (GH#463) `FractalClient.list_collections` by default only returns collections whose visibility flag is set to true, and whose group is "default". This change was made to filter out in-progress, intermediate, and specialized collections.

Bug Fixes

- (GH#424) Fixes a `ReactionDataset.visualize` bug with `groupby='D3'`.
- (GH#456, GH#452) Queries that project hybrid properties should now work as expected.

Deprecated Features

- (GH#426) In `Dataset` and `ReactionDataset` (`interface.collections`), the previously deprecated functions `query`, `get_history`, and `list_history` have been removed.

3.19.10 0.11.0 / 2019-10-01

New Features

- (GH#420) Pre-storage data handling through Elemental's `Protocols` feature are now present in Fractal. Although only optimization protocols are implemented functionally, the database side has been upgraded to store protocol settings.

Enhancements

- (GH#385, GH#404, GH#411) `Dataset` and `ReactionDataset` have five new functions for accessing data. `get_values` returns the canonical headline value for a dataset (e.g. the interaction energy for S22) in data columns with caching, both for result-backed values and contributed values. This function replaces the now-deprecated `get_history` and `get_contributed_values`. `list_values` returns the list of data columns available from `get_values`. This function replaces the now-deprecated `list_history` and `list_contributed_values`. `get_records` either returns `ResultRecord` or a projection. For the case of `ReactionDataset`, the results are broken down into component calculations. The function replaces the now-deprecated `query`. `list_records` returns the list of data columns available from `get_records`. `get_molecules` returns the `Molecule` associated with a dataset.
- (GH#393) A new feature added to `Client` to be able to have more custom and fast queries, the `custom_query` method. Those fast queries are now used in `torsiondrive.get_final_molecules` and `torsiondrive.get_final_results`. More Advanced queries will be added.
- (GH#394) Adds `tag` and `manager` selector fields to `client.query_tasks`. This is helpful for managing jobs in the queue and detecting failures.
- (GH#400, GH#401, GH#410) Adds Dockerfiles corresponding to builds on Docker Hub.
- (GH#406) The `Dataset` collection's primary indices (database level) have been updated to reflect its new understanding.

Bug Fixes

- (GH#396) Fixed a bug in internal `Dataset` function which caused `ComputeResponse` to be truncated when the number of calculations is larger than the query_limit.
- (GH#403) Fixed `Dataset.get_values` for any method which involved DFTD3.
- (GH#409) Fixed a compatibility bug in specific version of Intel-OpenMP by skipping version 2019.5-281.

Documentation Improvements

- (GH#399) A Kubernetes quickstart guide has been added.

3.19.11 0.10.0 / 2019-08-26

Note: Stable Beta Release

This release marks Fractal's official Stable Beta Release. This means that future, non-backwards compatible changes to the API will result in depreciation warnings.

Enhancements

- (GH#356) Collections' database representations have been improved to better support future upgrade paths.
- (GH#375) Dataset Records are now copied alongside the Collections.
- (GH#377) The `testing` suite from Fractal now exposes as a PyTest entry-point when Fractal is installed so that tests can be run from anywhere with the `--pyargs qcfractal` flag of `pytest`.
- (GH#384) "Dataset Records" and "Reaction Dataset Records" have been renamed to "Dataset Entry" and "Reaction Dataset Entry" respectively.
- (GH#387) The auto-documentation tech introduced in GH#321 has been replaced by the improved implementation in Elemental.

Bug Fixes

- (GH#388) Queue Manager shutdowns will now signal to reset any running tasks they own.

Documentation Improvements

- (GH#372, GH#376) Installation instructions have been updated and typo-corrected such that they are accurate now for both Conda and PyPi.

3.19.12 0.9.0 / 2019-08-16

New Features

- (GH#354) Fractal now takes advantage of Elemental's new Msgpack serialization option for Models. Serialization defaults to msgpack when available (`conda install msgpack-python [-c conda-forge]`), falling back to JSON otherwise. This results in substantial speedups for both serialization and deserialization actions and should be a transparent replacement for users within Fractal, Engine, and Elemental themselves.
- (GH#358) Fractal Server now exposes a CLI for user/permissions management through the `qcfractal-server user` command. See the full documentation for details.
- (GH#358) Fractal Server's CLI now supports user manipulations through the `qcfractal-server user` subcommand. This allows server administrators to control users and their access without directly interacting with the storage socket.

Enhancements

- (GH#330, GH#340, GH#348, GH#349) Many Pydantic based Models attributes are now documented and in an on-the-fly manner derived from the Pydantic Schema of those attributes.
- (GH#338) The Queue Manager which generated a `Result` is now stored in the `Result` records themselves.
- (GH#341) Skeletal Queue Manager YAML files can now be generated through the `--skel` or `--skeleton` CLI flag on `qcfractal-manager`
- (GH#361) Staged DB's in Fractal copy Alembic alongside them.
- (GH#363) A new REST API hook for services has been added so Clients can manage Services.

Bug Fixes

- (GH#359) A *FutureWarning* from Pandas has been addressed before it becomes an error.

3.19.13 0.8.1 / 2019-07-30

Bug Fixes

- (GH#335) Dataset's `get_history` function is fixed by allowing the ability to force a new query even if one has already been cached.

3.19.14 0.8.0 / 2019-07-25

Breaking Changes

Warning: PostgreSQL is now the only supported database backend.

Fractal has officially dropped support for MongoDB in favor of PostgreSQL as our database backend. Although MongoDB served the start of Fractal well, our database design as evolved since then and will be better served by PostgreSQL.

New Features

- (GH#307, GH#319 GH#321) Fractal's Server CLI has been overhauled to more intuitively and intelligently control Server creation, startup, configuration, and upgrade paths. This is mainly reflected in a Fractal Server config file, a config folder (default location `~/.qca`, and sub-commands `init`, `start`, `config`, and `upgrade` of the `qcfractal-server (command)` CLI. See the full documentation for details
- (GH#323) First implementation of the `GridOptimizationDataset` for collecting Grid Optimization calculations. Not yet fully featured, but operational for users to start working with.

Enhancements

- (GH#291) Tests have been formally added for the Queue Manager to reduce bugs in the future. They cannot test on actual Schedulers yet, but its a step in the right direction.
- (GH#295) Quality of life improvement for Mangers which by default will be less noisy about heartbeats and trigger a heartbeat less frequently. Both options can still be controlled through verbosity and a config setting.
- (GH#296) Services are now prioritized by the date they are created to properly order the compute queue.
- (GH#301) `TorsionDriveDataset` status can now be checked through the `.status()` method which shows the current progress of the computed data.
- (GH#310) The Client can now modify tasks and restart them if need be in the event of random failures.
- (GH#313) Queue Managers now have more detailed statistics about failure rates, and core-hours consumed (estimated)
- (GH#314) The `PostgresHarness` has been improved to include better error handling if Postgress is not found, and will not try to stop/start if the target data directory is already configured and running.
- (GH#318) Large collections are now automatically paginated to improve Server/Client response time and reduce query sizes. See also GH#322 for the Client-side requested pagination.

- (GH#322) Client's can request paginated queries for quicker responses. See also GH#318 for the Server-side auto-pagination.
- (GH#322) `Record` models and their derivatives now have a `get_molecule()` method for fetching the molecule directly.
- (GH#324) Optimization queries for its trajectory pull the entire trajectory in one go and keep the correct order. `get_trajectory` also pulls the correct order.
- (GH#325) Collections' have been improved to be more efficient. Previous queries are cached locally and the `compute` call is now a single function, removing the need to make a separate call to the submission formation.
- (GH#326) `ReactionDataset` now explicitly groups the fragments to future-proof this method from upstream changes to `Molecule` fragmentation.
- (GH#329) All API requests are now logged server side anonymously.
- (GH#331) Queue Manager jobs can now auto-retry failed jobs a finite number of times through QCEngine's retry capabilities. This will only catch RandomErrors and all other errors are raised normally.
- (GH#332) SQLAlchemy layer on the PostgreSQL database has received significant polish

Bug Fixes

- (GH#291) Queue Manager documentation generation works on Pydantic 0.28+. A number as-of-yet uncaught/unseen bugs were revealed in tests and have been fixed as well.
- (GH#300) Errors thrown in the level between Managers and their Adapters now correctly return a `FailedOperation` instead of `dict` to be consistent with all other errors and not crash the Manager.
- (GH#301) Invalid passwords present a helpful error message now instead of raising an Internal Server Error to the user.
- (GH#306) The Manager CLI option `tasks-per-worker` is correctly hyphens instead of underscores to be consistent with all other flags.
- (GH#316) Queue Manager workarounds for older versions of Dask-Jobqueue and Parsl have been removed and implicit dependency on the newer versions of those Adapters is enforced on CLI usage of `qcfractal-manager`. These packages are *not required* for Fractal, so their versions are only checked when specifically used in the Managers.
- (GH#320) Duplicated `initial_molecules` in the `TorsionDriveDataset` will no longer cause a failure in adding them to the database while still preserving de-duplication.
- (GH#327) Jupyter Notebook syntax highlighting has been fixed on Fractal's documentation pages.
- (GH#331) The BaseModel/Settings auto-documentation function can no longer throw an error which prevents using the code.

Deprecated Features

- (GH#291) Queue Manager Template Generator CLI has been removed as its functionality is superseded by the *qcfractal-manager* CLI.

3.19.15 0.7.2 / 2019-06-06

New Features

- (GH#279) Tasks will be deleted from the `TaskQueue` once they are completed successfully.
- (GH#271) A new set of scripts have been created to facilitate migration between MongoDB and PostgreSQL.

Enhancements

- (GH#275) Documentation has been further updated to be more contiguous between pages.
- (GH#276) Imports and type hints in Database objects have been improved to remove ambiguity and make imports easier to follow.
- (GH#280) Optimizations queried in the database are done with a more efficient lazy `selectin`. This should make queries much faster.
- (GH#281) Database Migration tech has been moved to their own folder to keep them isolated from normal production code. This PR also called the testing database `test_qcarchivedb` to avoid clashes with production DBs. Finally, a new keyword for testing geometry optimizations has been added.

Bug Fixes

- (GH#280) Fixed a SQL query where `join` was set instead of `noload` in the lazy reference.
- (GH#283) The monkey-patch for Dask + LSF had a typo in the keyword for its invoke. This has been fixed for the monkey-patch, as the upstream change was already fixed.

3.19.16 0.7.1 / 2019-05-28

Bug Fixes

- (GH#277) A more informative error is thrown when Mongo is not found by `FractalSnowflake`.
- (GH#277) ID's are no longer presented when listing Collections in Portal to minimize extra data.
- (GH#278) Fixed a bug in Portal where the Server was not reporting the correct unit.

3.19.17 0.7.0 / 2019-05-27

New Features

- (GH#206, GH#249, GH#264, GH#267) SQL Database is now feature complete and implemented. As final testing in production is continued, MongoDB will be phased out in the future.
- (GH#242) Parsl can now be used as an `Adapter` in the Queue Managers.
- (GH#247) The new `OptimizationDataset` collection has been added! This collection returns a set of optimized molecular structures given an initial input.
- (GH#254) The QCFractal Server Dashboard is now available through a Dash interface. Although not fully featured yet, future updates will improve this as features are requested.

- (GH#260) Its now even easier to install Fractal/Portal through conda with pre-built environments on the `qcarchive` conda channel. This channel only provides environment files, no packages (and there are not plans to do so.)
- (GH#269) The Fractal Snowflake project has been extended to work in Jupyter Notebooks. A Fractal Snowflake can be created with the `FractalSnowflakeHandler` inside of a Jupyter Session.

Database Compatibility Updates

- (GH#256) API calls to Elemental 0.4 have been updated. This changes the hashing system and so upgrading your Fractal Server instance to this (or higher) will require an upgrade path to the indices.

Enhancements

- (GH#238) `GridOptimizationRecord` supports the helper function `get_final_molecules` which returns the set of molecules at each final, optimized grid point.
- (GH#259) Both `GridOptimizationRecord` and `TorsionDriveRecord` support the helper function `get_final_results`, which is like `get_final_molecules`, but for x
- (GH#241) The visualization suite with Plotly has been made more general so it can be invoked in different classes. This particular PR updates the TorsionDriveDataSet objects.
- (:pr:`243) TorsionDrives in Fractal now support the updated Torsion Drive API from the underlying package. This includes both the new arguments and the "extra constraints" features.
- (GH#244) Tasks which fail are now more verbose in the log as to why they failed. This is additional information on top of the number of pass/fail.
- (GH#246) Queue Manager `verbosity` level is now passed down into the adapter programs as well and the log file (if set) will continue to print to the terminal as well as the physical file.
- (GH#247) Procedure classes now all derive from a common base class to be more consistent with one another and for any new Procedures going forward.
- (GH#248) Jobs which fail, or cannot be returned correctly, from Queue Managers are now better handled in the Manager and don't sit in the Manager's internal buffer. They will attempt to be returned to the Server on later updates. If too many jobs become stale, the Manager will shut itself down for safety.
- (GH#258 and GH#268) Fractal Queue Managers are now fully documented, both from the CLI and through the doc pages themselves. There have also been a few variables renamed and moved to be more clear the nature of what they do. See the PR for the renamed variables.
- (GH#251) The Fractal Server now reports valid minimum/maximum allowed client versions. The Portal Client will try check these numbers against itself and fail to connect if it is not within the Server's allowed ranges. Clients started from Fractal's `interface` do not make this check.

Bug Fixes

- (GH#248) Fixed a bug in Queue Managers where the extra worker startup commands for the Dask Adapter were not being parsed correctly.
- (GH#250) Record objects now correctly set their provenance time on object creation, not module import.
- (GH#253) A spelling bug was fixed in GridOptimization which caused hashing to not be processed correctly.
- (GH#270) LSF clusters not in `MB` for the units on memory by config are now auto-detected (or manually set) without large workarounds in the YAML file and the CLI file itself. Supports documented settings of LSF 9.1.3.

3.19.18 0.6.0 / 2019-03-30

Enhancements

- (GH#236 and GH#237) A large number of docstrings have been improved to be both more uniform, complete, and correct.
- (GH#239) DFT-D3 can now be queried through the `Dataset` and `ReactionDataset`.
- (GH#239) `list_collections` now returns Pandas Dataframes.

3.19.19 0.5.5 / 2019-03-26

New Features

- (GH#228) ReactionDatasets visualization statistics plots can now be generated through Plotly! This feature includes bar plots and violin plots and is designed for interactive use through websites, Jupyter notebooks, and more.
- (GH#233) TorsionDrive Datasets have custom visualization statistics through Plotly! This allows plotting 1-D torsion scans against other ones.

Enhancements

- (GH#226) LSF can now be specified for the Queue Managers for Dask managers.
- (GH#228) Plotly is an optional dependency overall, it is not required to run QCFractal or QCPortal but will be downloaded in some situations. If you don't have Plotly installed, more graceful errors beyond just raw `ImportErrors` are given.
- (GH#234) Queue Managers now report the number of passed and failed jobs they return to the server and can also have verbose (debug level) outputs to the log.
- (GH#234) Dask-driven Queue Managers can now be set to simply scale up to a fixed number of workers instead of trying to adapt the number of workers on the fly.

Bug Fixes

- (GH#227) SGE Clusters specified in Queue Manager under Dask correctly process `job_extra` for additional scheduler headers. This is implemented in a stable way such that if the upstream Dask Jobqueue implements a fix, the Manager will keep working without needing to get a new release.
- (GH#234) Fireworks Managers now return the same pydantic models as every other Manager instead of raw dictionaries.

3.19.20 0.5.4 / 2019-03-21

New Features

- (GH#216) Jobs submitted to the queue can now be assigned a priority to be served out to the Managers.
- (GH#219) Temporary, pop-up, local instances of `FractalServer` can now be created through the `FractalSnowflake`. This creates an instance of `FractalServer`, with its database structure, which is entirely held in temporary storage and memory, all of which is deleted upon exit/stop. This feature is designed for those who want to tinker with Fractal without needed to create their own database or connect to a production `FractalServer`.
- (GH#220) Queue Managers can now set the `scratch_directory` variable that is passed to QCEngine and its workers.

Enhancements

- (GH#216) Queue Managers now report what programs and procedures they have access to and will only pull jobs they think they can execute.
- (GH#222) All of `FractalClient`'s methods now have full docstrings and type annotations for clairy
- (GH#222) Massive overhaul to the REST interface to simplify internal calls from the client and server side.
- (GH#223) `TorsionDriveDataset` objects are modeled through pydantic objects to allow easier interface with the database back end and data validation.

Bug Fixes

- (GH#215) Dask Jobqueue for the `qcfractal-manager` is now tested and working. This resolve the outstanding issue introduced in GH#211 and pushed in v0.5.3.
- (GH#216) Tasks are now stored as `TaskRecord` pydantic objects which now preempts a bug introduced from providing the wrong schema.
- (GH#217) Standalone QCPortal installs now report the correct version
- (GH#221) Fixed a bug in `ReactionDataset.query` where passing in `None` was treated as a string.

3.19.21 0.5.3 / 2019-03-13

New Features

- (GH#207) All compute operations can now be augmented with a `tag` which can be later consumed by different ``QueueManager``s to only carry out computations with specified tags.
- (GH#210) Passwords in the database can now be generated for new users and user information can be updated (server-side only)
- (GH#210) `Collections` can now be updated automatically from the defaults
- (GH#211) The `qcfractal-manager` CLI command now accepts a config file for more complex Managers through Dask JobQueue. As such, many of the command line flags have been altered and can be used to either spin up a PoolExecutor, or overwrite the config file on-the-fly. As of this PR, the Dask Jobqueue component has been untested. Future updates will indicate when this has been tested.

Enhancements

- (GH#203) `FractalClient`'s `get_X` methods have been renamed to `query_X` to better reflect what they actually do. An exception to this is the `get_collections` method which is still a true `get`.
- (GH#207) `FractalClient.list_collections` now respects show case sensitive results and queries are case insensitive
- (GH#207) `FractalServer` can now compress responses to reduce the amount of data transmitted over the serialization. The main benefactor here is the `OpenFFWorkflow` collection which has significant transfer speed improvements due to compression.
- (GH#207) The `OpenFFWorkflow` collection now has better validation on input and output data.
- (GH#210) The `OpenFFWorkflow` collection only stores database `id` to reduce duplication and data transfer quantities. This results in about a 50x duplication reduction.
- (GH#211) The `qcfractal-template` command now has fields for Fractal username and password.
- (GH#212) The docs for QCFractal and QCPortal have been split into separate structures. They will be hosted on separate (although linked) pages, but their content will all be kept in the QCFractal source code. QCPortal's docs are for most users whereas QCFractal docs will be for those creating their own Managers, Fractal instances, and developers.

Bug Fixes

- (GH#207) `FractalClient.get_collections` is now correctly case insensitive.
- (GH#210) Fixed a bug in the `iterate` method of services which returned the wrong status if everything completed right away.
- (GH#210) The `repr` of the MongoEngine Socket now displays correctly instead of crashing the socket due to missing attribute

3.19.22 0.5.2 / 2019-03-08

New Features

- (GH#197) New `FractalClient` instances will automatically connect to the central MolSSI Fractal Server

Enhancements

- (GH#195) Read-only access has been granted to many objects separate from their write access. This is in contrast to the previous model where either there was no access security, or everything was access secure.
- (GH#197) Unknown stoichiometry are no longer allowed in the `ReactionDataset`
- (GH#197) CLI for FractalServer uses Executor only to encourage using the Template Generator introduced in GH#177.
- (GH#197) `Dataset` objects can now query keywords from aliases as well.

Bug Fixes

- (GH#195) Managers cannot pull too many tasks and potentially loose data due to query limits.
- (GH#195) `Records` now correctly adds Provenance information
- (GH#196) `compute_torsion` example update to reflect API changes
- (GH#197) Fixed an issue where CLI input flags were not correctly overwriting default values
- (GH#197) Fixed an issue where `Collections` were not correctly updating when the `save` function was called on existing objects in the database.
- (GH#197) `_qcfractal_tags` are no longer carried through the `Records` objects in errant.
- (GH#197) Stoichiometry information is no longer accepted in the `Dataset` object since this is not used in this class of object anymore (see `ReactionDataset`).

3.19.23 0.5.1 / 2019-03-04

New Features

- (GH#177) Adds a new `qcfractal-template` command to generate `qcfractal-manager` scripts.
- (GH#181) Pagination is added to queries, defaults to 1000 matches.
- (GH#185) Begins setup documentation.
- (GH#186) Begins database design documentation.
- (GH#187) Results add/update is now simplified to always store entire objects rather than update partials.
- (GH#189) All database compute records now go through a single `BaseRecord` class that validates and hashes the objects.

Enhancements

- (GH#175) Refactors query massaging logic to a single function, ensures all program queries are lowercase, etc.
- (GH#175) Keywords are now lazy reference fields.
- (GH#182) Reworks models to have strict fields, and centralizes object hashing with many tests.
- (GH#183) Centralizes duplicate checking so that accidental mixed case duplicate results could go through.
- (GH#190) Adds QCArchive sphinx theme to the documentation.

Bug Fixes

- (GH#176) Benchmarks folder no longer shipped with package

3.19.24 0.5.0 / 2019-02-20

New Features

- (GH#165) Separates datasets into a Dataset, ReactionDataset, and OptimizationDataset for future flexability.
- (GH#168) Services now save their Procedure stubs automatically, the same as normal Procedures.
- (GH#169) `setup.py` now uses the README.md and conveys Markdown to PyPI.
- (GH#171) Molecule addition now takes in a flat list and returns a flat list of IDs rather than using a dictionary.
- (GH#173) Services now return their correspond Procedure ID fields.

Enhancements

- (GH#163) Ignores pre-existing IDs during storage add operations.
- (GH#167) Allows empty queries to successfully return all results rather than all data in a collection.
- (GH#172) Bumps pydantic version to 0.20 and updates API.

Bug Fixes

- (GH#170) Switches Parsl from IPPExecutor to ThreadExecutor to prevent some bad semaphore conflicts with PyTest.

3.19.25 0.5.0rc1 / 2019-02-15

New Features

- (GH#114) A new Collection: `Generic`, has been added to allow semi-structured user defined data to be built without relying only on implemented collections.
- (GH#125) QCElemental common pydantic models have been integrated throughout the QCFractal code base, making a common model repository for the prevalent `Molecule` object (and others) come from a single source. Also converted QCFractal to pass serialized pydantic objects between QCFractal and QCEngine to allow validation and (de)serialization of objects automatically.
- (GH#130, GH#142, and GH#145) Pydantic serialization has been added to all REST calls leaving and entering both QCFractal Servers and QCFractal Portals. This allows automatic REST call validation and formatting on both server and client sides.
- (GH#141 and GH#152) A new GridOptimizationRecord service has been added to QCFractal. This feature supports relative starting positions from the input molecule.

Enhancements

General note: `Options` objects have been renamed to `KeywordSet` to better match their goal (See GH#155.)

- (GH#110) QCFractal now depends on QCElemental and QCEngine to improve consistent imports.
- (GH#116) Queue Manger Adapters are now more generalized and inherit more from the base classes.
- (GH#118) Single and Optimization procedures have been streamlined to have simpler submission specifications and less redundancy.
- (GH#133) Fractal Server and Queue Manager startups are much more verbose and include version information.
- (GH#135) The TorsionDriveService has a much more regular structure based on pydantic models and a new TorsionDrive model has been created to enforce both validation and regularity.
- (GH#143) `Task``s in the Mongo database can now be referenced by multiple ``Results` and `Procedures` (i.e. a single `Result` or `Procedure` does not have ownership of a `Task`.)
- (GH#147) Service submission has been overhauled such that all services submit to a single source. Right now, only one service can be submitted at a time (to be expanded in a future feature.) TorsionDrive can now have multiple molecule inputs.
- (GH#149) Package import logic has been reworked to reduce the boot-up time of QCFractal from 3000ms at the worst to about 600ms.
- (GH#150) ``KeywordSet``s are now modeled much more consistently through pydantic models and are consistently hashed to survive round trip serialization.
- (GH#153) Datasets now support option aliases which map to the consistent `KeywordSet` models from GH#150.
- (GH#155) Adding multiple `Molecule` or `Result` objects to the database at the same time now always return their Database ID's if added, and order of returned list of ID's matches input order. This PR also renamed `Options` to `KeywordSet` to properly reflect the goal of the object.
- (GH#156) Memory and Number of Cores per Task can be specified when spinning up a Queue Manager and/or Queue Adapter objects. These settings are passed on to QCEngine. These must be hard-set by users and no environment inspection is done. Users may continue to choose not to set these and QCEngine will consume everything it can when it lands on a compute.
- (GH#162) Services can now be saved and fetched from the database through MongoEngine with document validation on both actions.

Bug Fixes

- (GH#132) Fixed MongoEngine Socket bug where calling some functions before others resulted in an error due to lack of initialized variables.
- (GH#133) `Molecule` objects cannot be oriented once they enter the QCFractal ecosystem (after optional initial orientation.) ``Molecule``s also cannot be oriented by programs invoked by the QCFractal ecosystem so orientation is preserved post-calculation.
- (GH#146) CI environments have been simplified to make maintaining them easier, improve test coverage, and find more bugs.
- (GH#158) Database addition documents in general will strip IDs from the input dictionary which caused issues from MongoEngine having a special treatment for the dictionary key "id".

3.19.26 0.4.0a / 2019-01-15

This is the fourth alpha release of QCFractal focusing on the database backend and compute manager enhancements.

New Features

- (GH#78) Migrates Mongo backend to MongoEngine.
- (GH#78) Overhauls tasks so that results or procedures own a task and ID.
- (GH#78) Results and procedures are now inserted upon creation, not just completion. Added a status field to results and procedures.
- (GH#78) Overhauls storage API to no longer accept arbitrary JSON queries, but now pinned kwargs.
- (GH#106) Compute managers now have heartbeats and tasks are recycled after a manager has not been heard from after a preset interval.
- (GH#106) Managers now also quietly shutdown on SIGTERM as well as SIGINT.

Bug Fixes

- (GH#102) Py37 fix for pydantic and better None defaults for `options`.
- (GH#107) `FractalClient.get_collections` now raises an exception when no collection is found.

3.19.27 0.3.0a / 2018-11-02

This is the third alpha release of QCFractal focusing on a command line interface and the ability to have multiple queues interacting with a central server.

New Features

- (GH#72) Queues are no longer required of FractalServer instances, now separate QueueManager instances can be created that push and pull tasks to the server.
- (GH#80) A Parsl Queue Manager was written.
- (GH#75) CLI's have been added for the *qcfractal-server* and *qcfractal-manager* instances.
- (GH#83) The status of server tasks and services can now be queried from a FractalClient.
- (GH#82) OpenFF Workflows can now add single optimizations for fragments.

Enhancements

- (GH#74) The documentation now has flowcharts showing task and service pathways through the code.
- (GH#73) Collection *.data* attributes are now typed and validated with pydantic.
- (GH#85) The CLI has been enhanced to cover additional features such as *queue-manager* ping time.
- (GH#84) QCEngine 0.4.0 and geomeTRIC 0.9.1 versions are now compatible with QCFractal.

Bug Fixes

- (GH#92) Fixes an error with query OpenFFWorkflows.

3.19.28 0.2.0a / 2018-10-02

This is the second alpha release of QCFractal containing architectural changes to the relational pieces of the database. Base functionality has been expanded to generalize the collection idea with BioFragment and OpenFFWorkflow collections.

Documentation

- (GH#58) A overview of the QCArchive project was added to demonstrate how all modules connect together.

New Features

- (GH#57) OpenFFWorkflow and BioFragment collections to support OpenFF uses cases.
- (GH#57) Requested compute will now return the id of the new submissions or the id of the completed results if duplicates are submitted.
- (GH#67) The OpenFFWorkflow collection now supports querying of individual geometry optimization trajectories and associated data for each torsiondrive.

Enhancements

- (GH#43) Services and Procedures now exist in the same unified table when complete as a single procedure can be completed in either capacity.
- (GH#44) The backend database was renamed to storage to prevent misunderstanding of the Database collection.
- (GH#47) Tests can that require an activate Mongo instance are now correctly skipped.
- (GH#51) The queue now uses a fast hash index to determine uniqueness and prevent duplicate tasks.
- (GH#52) QCFractal examples are now tested via CI.
- (GH#53) The MongoSocket *get_generic_by_id* was deprecated in favor of *get_generic* where an ID can be a search field.
- (GH#61, GH#64) TorsionDrive now tracks tasks via ID rather than hash to ensure integrity.
- (GH#63) The Database collection was renamed Dataset to more correctly illuminate its purpose.
- (GH#65) Collection can now be aquired directly from a client via the *client.get_collection* function.

Bug Fixes

- (GH#52) The molecular comparison technology would occasionally incorrectly orientate molecules.

3.19.29 0.1.0a / 2018-09-04

This is the first alpha release of QCFractal containing the primary structure of the project and base functionality.

New Features

- (GH#41) Molecules can now be queried by molecule formula
- (GH#39) The server can now use SSL protection and auto-generates SSL certificates if no certificates are provided.
- (GH#31) Adds authentication to the FractalServer instance.
- (GH#26) Adds TorsionDrive (formally Crank) as the first service.
- (GH#26) Adds a "services" feature which can create large-scale iterative workflows.
- (GH#21) QCFractal now maintains its own internal queue and uses queuing services such as Fireworks or Dask only for the currently running tasks

Enhancements

- (GH#40) Examples can now be testing through PyTest.
- (GH#38) First major documentation pass.
- (GH#37) Canonicalizes string formatting to the `"{}".format` usage.
- (GH#36) Fireworks workflows are now cleared once complete to keep the active entries small.
- (GH#35) The "database" table can now be updated so that database entries can now evolve over time.
- (GH#32) TorsionDrive services now track all computations that are completed rather than just the last iteration.
- (GH#30) Creates a Slack Community and auto-invite badge on the main readme.
- (GH#24) Remove conda-forge from conda-envs so that more base libraries can be used.

Bug Fixes

- Innumerable bug fixes and improvements in this alpha release.

INDEX

E

F

G

H

J

K

L

M

O

P

Q

R

S

T

U

V