QCPortal Documentation

Release 5

The QCArchive Development Team

Aug 19, 2019

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:** Explorate 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. There are many types of collections such as:

- `ReactionDataset` - A dataset that allows hold many chemical reactions which can be computed with a large vareity of computational methods.
- `TorsionDriveDataset` - A dataset of molecules where torsion scans can be executed with a variety of computational methods.

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

3.1 Install QCPortal

You can install `qcportal` with `conda`, with `pip`, or by installing from source.

3.1.1 Conda

You can install or update `qcportal` using conda:

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

The `qcportal` package is maintained on the conda-forge channel.

3.1.2 Pip

To install or update `qcportal` with `pip`:

```
pip install -U qcportal
```

3.1.3 Developer Install

The QCPortal package is part of the QCFractal package and is the `qcfractal.interface` folder. To install QCFractal from source, clone the repository from GitHub:

```
git clone https://github.com/MolSSI/QCFractal.git
cd qcfractal
pip install -e .
```

A developer version of `qcportal` can then be imported as:

```
>>> import qcfractal.interface as ptl
```

Collections

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

- *Collections*
- *Dataset*

3.2 Collections

Collections are an organizational objects that keeps track of collections of results results, provides the ability to analyze and visualize these results, and is able to compute new results.

3.2.1 Collections Querying

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

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

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

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

3.2.2 Available Collections

Below is a complete list of collections available from QCPortal:

- *Dataset* - A collection for running a single set of reactions under many methods.
- *TorsionDrive Dataset* - A collection for running a single set of TorsionDrives under different methods.

3.3 Dataset

Datasets are useful for computing many methods for a single set of reactions where a reaction in a combination of molecules such as. There are currently two types of datasets:

- `rxn` for datasets based on canonical chemical reactions $A + B \rightarrow C$
- `ie` for interaction energy datasets $M_{complex} \rightarrow M_{monomer_1} + M_{monomer_2}$

3.3.1 Querying

3.3.2 Visualizing

3.3.3 Creating

Blank dataset can be constructed by choosing a dataset name and a dataset type (`dtype`).

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

New reactions can be added by providing the linear combination of molecules required to compute the desired quantity. When the Dataset is queried these linear combinations area automatically combined for the caller.

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

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

N_atom = ptl.Molecule("""
0 2
N 0.0 0.0 0.0
""")


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

A given reaction can be examined by using the `get_rxn` function. We store the `molecule_hash` followed by the coefficient.

```
json.dumps(ds.get_rxn("Nitrogen Molecule"), indent=2)
{
  "name": "Nitrogen Molecule",
  "stoichiometry": {
    "default": {
      "4d7518cc2c741f2b5f48d7c16e2ad4c660e11890": 1.0,
      "636aa99f49b32dd81d7c8cb3741e16c632835cdf": -2.0
    }
  },
  "attributes": {},
  "reaction_results": {
    "default": {}
  }
}
```

Datasets of dtype `ie` can be automatically contstruct counterpoise-correct (`cp`) and non-counterpoise correct (`default`) n-body expansions. Where the number after the name corresponds to the number of bodies involved in the computation.

```
ie_ds = ptl.collections.Dataset("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": {
      "4cd68e5dde15c19fc2f5101d5fc5f19ac8afbc9c": 1.0,
      "da635a2e012a9ea876ea54422256bd93124e4271": 1.0
    },
    "cp1": {
      "9299ecc50e018f60128845e9f14b803da641f816": 1.0,
      "0f6382da1b658b634a05bc7c7f65ad115328f06f": 1.0
    },
    "default": {
```

(continues on next page)

(continued from previous page)

```
      "358ad4bb4620e35cec79b17ec0f40acae1a548cb": 1.0
    },
    "cp": {
      "358ad4bb4620e35cec79b17ec0f40acae1a548cb": 1.0
    }
  },
  "attributes": {},
  "reaction_results": {
    "default": {}
  }
}
```

3.3.4 Computing

3.3.5 API

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

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

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

Adds a ContributedValues to the database.

Parameters

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

add_entry(*name: str*, *molecule: qcelemental.models.molecule.Molecule*, ***kwargs*)

Adds a new entry to the Datset

Parameters

- **name** (*str*) – The name of the record
- **molecule** (*Molecule*) – The Molecule associated with this record
- ****kwargs** (*Dict[str, Any]*) – Additional arguements 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*, *tag: Optional[str] = None*, *priority: Optional[str] = None*) → qcfractal.interface.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.

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

get_contributed_values(*key: str*) → qcfractal.interface.collections.dataset.ContributedValues

Returns a copy of the requested ContributedValues object.

Parameters **key** (*str*) – The ContributedValues object key.

Returns The requested ContributedValues object.

Return type ContributedValues

get_contributed_values_column(*key: str*) → Series

Returns a Pandas column with the requested contributed values

Parameters

- **key** (*str*) – The ContributedValues object key.
- **scale** (*None, optional*) – All units are based in Hartree, the default scaling is to kcal/mol.

Returns A pandas Series containing the request values.

Return type Series

get_history(*method: Optional[str] = None*, *basis: Optional[str] = None*, *keywords: Optional[str] = None*, *program: Optional[str] = None*, *force: bool = False*) → DataFrame

Queries known history from the search paramaters provided. Defaults to the standard programs and keywords if not provided.

Parameters

- **method** (*Optional[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

Returns A DataFrame of the queried parameters

Return type DataFrame

get_index() → 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*) → KeywordSet
Pulls the keywords alias from the server for inspection.

Parameters

- **alias** (*str*) – The keywords alias.
- **program** (*str*) – The program the keywords correspond to.

Returns The requested KeywordSet

Return type *KeywordSet*

list_contributed_values() → List[str]
Lists the known keys for all contributed values.

Returns A list of all known contributed values.

Return type List[str]

list_history(*dftd3: bool = False*, *get_base: bool = False*, *pretty: bool = True*, ***search*) → DataFrame
Lists the history of computations completed.

Parameters ****search** (*Dict[str, Optional[str]]*) – Allows searching to narrow down return.

Returns The computed keys.

Return type DataFrame

query(*method: str*, *basis: Optional[str] = None*, ***, *keywords: Optional[str] = None*, *program: Optional[str] = None*, *field: str = None*, *as_array: bool = False*, *force: bool = False*) → str
Queries the local Portal for the requested keys.

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
- **field** (*str, optional*) – The result field to query on
- **as_array** (*bool, optional*) – Converts the returned values to NumPy arrays
- **force** (*bool, optional*) – Forces a requery if data is already present

Returns **success** – The name of the queried column

Return type str

Examples

```
>>> ds.query("B3LYP", "aug-cc-pVDZ", stoich="cp", prefix="cp-")
```

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.

statistics(*stype: str*, *value: str*, *bench: Optional[str] = None*, ***kwargs*)
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 **ret** – Returns a DataFrame, Series, or float with the requested statistics depending on input.

Return type pd.DataFrame, pd.Series, float

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*) → 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.

Returns The requested figure.

Return type plotly.Figure

3.4 TorsionDrive Dataset

TorsionDriveDatasets are sets of TorsionDrive computations where the primary index a set of starting molecules and each column is represented by a new TorsionDrive specification. This Dataset is a procedure-style dataset where a record of the *ObjectId* of each TorsionDrive computation are recorded in the metadata.

3.4.1 Querying

3.4.2 Visualizing

3.4.3 Creating

A empty TorsionDriveDataset can be constructed by choosing a dataset name.

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

3.4.4 Computing

3.4.5 API

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

class DataModel

add_entry(*name: str*, *initial_molecules: List[qcelemental.models.molecule.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: str = None*, *overwrite=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

counts(*entries: Union[str, List[str]], specs: Union[List[str], str, None] = None, count_gradients=False*) → 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

status(*specs: Union[str, List[str]] = None*, *collapse: bool = True*, *status: Optional[str] = None*, *detail: bool = False*) → DataFrame

Returns the status of all current specifications.

Parameters

- **specs** (*Union[str, List[str]], optional*) – Description
- **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

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

Record Documentation

Documentation for compute records.

3.5 Overview

A *Record* is a

A result is a single quantum chemistry method evaluation, which might be an energy, an analytic gradient or Hessian, or a property evaluation. Collections of evaluations such as finite-difference gradients, complete basis set extrapolation, or geometry optimizations belong under the "Procedures" heading.

3.5.1 Single Results

3.5.2 Procedures

A result can be found based off a unique tuple of `(program, molecule_id, options_set, method, basis)`:

- `program` - A lowercase string representation of the quantum chemistry program used (`gamess`, `nwchem`, `psi4`)
- `molecule_id` - The *ObjectId* of the molecule used in the computation.
- `keywords_set` - The key to the options set stored in the database (`default` -> `{"e_convergence": 1.e-7, "scf_type": "df", ...}`)
- `method` - A lowercase string representation of the method used in the computation (`b3lyp`, `mp2`, `ccsd(t)`).
- `basis` - A lowercase string representation of the basis used in the computation (`6-31g`, `cc-pvdz`, `def2-svp`)

3.6 Results

A result is a single quantum chemistry method evaluation, which might be an energy, an analytic gradient or Hessian, or a property evaluation. Collections of evaluations such as finite-difference gradients, complete basis set extrapolation, or geometry optimizations belong under the "Procedures" heading.

3.6.1 Indices

A result can be found based off a unique tuple of `(program, molecule_id, options_set, method, basis)`

- `program` - A lowercase string representation of the quantum chemistry program used (`gamess`, `nwchem`, `psi4`)
- `molecule_id` - The *ObjectId* of the molecule used in the computation.
- `keywords_set` - The key to the options set stored in the database (`default` -> `{"e_convergence": 1.e-7, "scf_type": "df", ...}`)
- `method` - A lowercase string representation of the method used in the computation (`b3lyp`, `mp2`, `ccsd(t)`).
- `basis` - A lowercase string representation of the basis used in the computation (`6-31g`, `cc-pvdz`, `def2-svp`)

3.6.2 Schema

All results are stored using the QCSchema so that the storage is quantum chemistry program agnostic. An example of the QCSchema input is shown below:

```
{
  "schema_name": "qc_json_input",
  "schema_version": 1,
  "molecule": {
    "geometry": [
      0.0,  0.0,    -0.1294,
      0.0, -1.4941,  1.0274,
      0.0,  1.4941,  1.0274
    ],
    "symbols": ["O", "H", "H"]
  },
  "driver": "energy",
  "model": {
    "method": "MP2",
    "basis": "cc-pVDZ"
  },
  "keywords": {},
}
```

This input would correspond to the following output:

```
{
  "schema_name": "qc_json_output",
  "schema_version": 1,
  "molecule": {
    "geometry": [
      0.0,  0.0,    -0.1294,
```

(continues on next page)

(continued from previous page)


```
      0.0, -1.4941,  1.0274,
      0.0,  1.4941,  1.0274
    ],
    "symbols": ["O", "H", "H"]
  },
  "driver": "energy",
  "model": {
    "method": "MP2",
    "basis": "cc-pVDZ"
  },
  "keywords": {},
  "provenance": {
    "creator": "QM Program",
    "version": "1.1",
    "routine": "module.json.run_json"
  },
  "return_result": -76.22836742810021,
  "success": true,
  "properties": {
    "calcinfo_nbasis": 24,
    "calcinfo_nmo": 24,
    "calcinfo_nalpha": 5,
    "calcinfo_nbeta": 5,
    "calcinfo_natom": 3,
    "return_energy": -76.22836742810021,
    "scf_one_electron_energy": -122.44534536383037,
    "scf_two_electron_energy": 37.62246494040059,
    "nuclear_repulsion_energy": 8.80146205625184,
    "scf_dipole_moment": [0.0, 0.0, 2.0954],
    "scf_iterations": 10,
    "scf_total_energy": -76.02141836717794,
    "mp2_same_spin_correlation_energy": -0.051980792916251864,
    "mp2_opposite_spin_correlation_energy": -0.15496826800602342,
    "mp2_singles_energy": 0.0,
    "mp2_doubles_energy": -0.20694906092226972,
    "mp2_total_correlation_energy": -0.20694906092226972,
    "mp2_total_energy": -76.22836742810021
  }
}
```


3.7 Procedures

3.8 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**(***data*)

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[str, 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=False*, *validate=None*, ***kwargs*)

The fundamental representation of a Molecule inside the QCArchive Ecosystem. This model contains data for atomic elements, coordinates, connectivity, charges, fragmentation, etc. The model also supports orientation and measurement utilities.

All Molecule objects are oriented via inertia tensor with the center-of-mass at (0,0,0) unless explicitly blocked.

All Molecule objects have coordinates truncated to 8 (GEOMETRY_NOISE) decimal places and mass and charge truncated to 6 (MASS_NOISE) and 4 (CHARGE_NOISE), respectively, unless explicitly blocked.

Parameters

- **schema_name** (*ConstrainedStrValue, Default: qcschema_molecule*) – The QCSchema specification this model conforms to. Explicitly fixed as qcschema_molecule.
- **schema_version** (*int, Default: 2*) – The version number of `schema_name` that this Molecule model conforms to.
- **validated** (*bool, Default: False*) – A boolean indicator (for speed purposes) that the input Molecule data has been previously checked for schema (data layout and type) and physics (e.g., non-overlapping atoms, feasible multiplicity) compliance. This should be False in most cases. A `True` setting should only ever be set by the constructor for this class itself or other trusted sources such as a Fractal Server or previously serialized Molecules.
- **symbols** (*Array*) – An ordered (nat,) array-like object of atomic elemental symbols of shape (nat,). The index of this attribute sets atomic order for all other per-atom setting like `real` and the first dimension of `geometry`. Ghost/Virtual atoms must have an entry in this array-like and are indicated by the matching the 0-indexed indices in `real` field.
- **geometry** (*Array*) – An ordered (nat,3) array-like for XYZ atomic coordinates [a0]. Atom ordering is fixed; that is, a consumer who shuffles atoms must not reattach the input (pre-shuffling) molecule schema instance to any output (post-shuffling) per-atom results (e.g., gradient). Index of the first dimension matches the 0-indexed indices of all other per-atom settings like `symbols` and `real`. Can also accept array-likes which can be mapped to (nat,3) such as a 1-D list of length 3*nat, or the serialized version of the array in (3*nat,) shape; all forms will be reshaped to (nat,3) for this attribute.
- **name** (*str, Optional*) – A common or human-readable name to assign to this molecule. Can be arbitrary.
- **identifiers** (`Identifiers`, Optional) – An optional dictionary of additional identifiers by which this Molecule can be referenced, such as INCHI, canonical SMILES, etc. See the :class:`Identifiers` model for more details.
- **comment** (*str, Optional*) – Additional comments for this Molecule. Intended for pure human/user consumption and clarity.

- **molecular_charge** (*float, Default: 0.0*) – The net electrostatic charge of this Molecule.
- **molecular_multiplicity** (*int, Default: 1*) – The total multiplicity of this Molecule.
- **masses** (*Array, Optional*) – An ordered 1-D array-like object of atomic masses [u] of shape (nat,). Index order matches the 0-indexed indices of all other per-atom settings like `symbols` and `real`. If this is not provided, the mass of each atom is inferred from their most common isotope. If this is provided, it must be the same length as `symbols` but can accept `None` entries for standard masses to infer from the same index in the `symbols` field.
- **real** (*Array, Optional*) – An ordered 1-D array-like object of shape (nat,) indicating if each atom is real (`True`) or ghost/virtual (`False`). Index matches the 0-indexed indices of all other per-atom settings like `symbols` and the first dimension of `geometry`. If this is not provided, all atoms are assumed to be real (`True`).If this is provided, the reality or ghostality of every atom must be specified.
- **atom_labels** (*Array, Optional*) – Additional per-atom labels as a 1-D array-like of of strings of shape (nat,). Typical use is in model conversions, such as Elemental <-> Molpro and not typically something which should be user assigned. See the `comments` field for general human-consumable text to affix to the Molecule.
- **atomic_numbers** (*Array, Optional*) – An optional ordered 1-D array-like object of atomic numbers of shape (nat,). Index matches the 0-indexed indices of all other per-atom settings like `symbols` and `real`. Values are inferred from the `symbols` list if not explicitly set.
- **mass_numbers** (*Array, Optional*) – An optional ordered 1-D array-like object of atomic *mass* numbers of shape (nat). Index matches the 0-indexed indices of all other per-atom settings like `symbols` and `real`. Values are inferred from the most common isotopes of the `symbols` list if not explicitly set.
- **connectivity** (*List[Tuple[int, int, float]], Optional*) – The connectivity information between each atom in the `symbols` array. Each entry in this list is a Tuple of `(atom_index_A, atom_index_B, bond_order)` where the `atom_index` matches the 0-indexed indices of all other per-atom settings like `symbols` and `real`.
- **fragments** (*List[Array], Optional*) – An indication of which sets of atoms are fragments within the Molecule. This is a list of shape (nfr) of 1-D array-like objects of arbitrary length. Each entry in the list indicates a new fragment. The index of the list matches the 0-indexed indices of `fragment_charges` and `fragment_multiplicities`. The 1-D array-like objects are sets of atom indices indicating the atoms which compose the fragment. The atom indices match the 0-indexed indices of all other per-atom settings like `symbols` and `real`.
- **fragment_charges** (*List[float], Optional*) – The total charge of each fragment in the `fragments` list of shape (nfr,). The index of this list matches the 0-index indices of `fragment` list. Will be filled in based on a set of rules if not provided (and `fragments` are specified).
- **fragment_multiplicities** (*List[int], Optional*) – The multiplicity of each fragment in the `fragments` list of shape (nfr,). The index of this list matches the 0-index indices of `fragment` list. Will be filled in based on a set of rules if not provided (and `fragments` are specified).
- **fix_com** (*bool, Default: False*) – An indicator which prevents pre-processing the Molecule object to translate the Center-of-Mass to (0,0,0) in euclidean coordinate space. Will result in a different `geometry` than the one provided if False.
- **fix_orientation** (*bool, Default: False*) – An indicator which prevents pre-processes the Molecule object to orient via the inertia tensor.Will result in a different `geometry` than the one provided if False.

- **fix_symmetry** (*str, Optional*) – Maximal point group symmetry which `geometry` should be treated. Lowercase.
- **provenance** (`Provenance`, Default: {'creator': 'QCElemental', 'version': 'v0.6.0', 'routine': 'qcelemental.models.molecule'}) – The provenance information about how this Molecule (and its attributes) were generated, provided, and manipulated.
- **id** (*Any, Optional*) – A unique identifier for this Molecule object. This field exists primarily for Databases (e.g. Fractal's Server) to track and lookup this specific object and should virtually never need to be manually set.
- **extras** (*Dict[str, Any], Optional*) – Extra information to associate with this Molecule.

class qcportal.models.**ResultRecord**(***data*)

Parameters

- **client** (*Any, Optional*) – The client object which the records are fetched from.
- **cache** (*Dict[str, 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.
- **extras** (*Dict[str, 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.
- **task_id** (*ObjectId, Optional*) – Id of the compute task tracked by Fractal in its TaskTable.
- **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[str, 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`.

class qcportal.models.**OptimizationRecord**(***data*)

A OptimizationRecord for all optimization procedure data.

Parameters

- **client** (*Any, Optional*) – The client object which the records are fetched from.
- **cache** (*Dict[str, 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.
- **extras** (*Dict[str, 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.
- **task_id** (*ObjectId, Optional*) – Id of the compute task tracked by Fractal in its TaskTable.
- **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[str, Any], Default: {}*) – The keyword options which were passed into the Optimization program. Note: These are a Dict, not a *KeywordSet*.
- **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.

class qcportal.models.**QCSpecification**

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.
- **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**

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**(***data*)

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[str, 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.
- **extras** (*Dict[str, 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.
- **task_id** (*ObjectId, Optional*) – Id of the compute task tracked by Fractal in its TaskTable.
- **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** (*final_energy_dict type=float required*) – Map of the final energy from the grid optimization at each grid point.
- **grid_optimizations** (*grid_optimizations type=ObjectId required*) – 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.

class qcportal.models.**OptimizationRecord**(***data*)

A OptimizationRecord for all optimization procedure data.

Parameters

- **client** (*Any, Optional*) – The client object which the records are fetched from.
- **cache** (*Dict[str, 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.
- **extras** (*Dict[str, 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.
- **task_id** (*ObjectId, Optional*) – Id of the compute task tracked by Fractal in its TaskTable.
- **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[str, Any], Default: {}*) – The keyword options which were passed into the Optimization program. Note: These are a Dict, not a *KeywordSet*.
- **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.

class qcportal.models.**TorsionDriveInput**

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**(***data*)

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[str, 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.
- **extras** (*Dict[str, 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.
- **task_id** (*ObjectId, Optional*) – Id of the compute task tracked by Fractal in its TaskTable.
- **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** (*final_energy_dict type=float required*) – The final energy at each angle of the TorsionDrive scan.
- **optimization_history** (*Dict[str, List[qcportal.models.common_models.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** (*minimum_positions type=int required*) – 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.

3.9 Portal Client

The `FractalClient` is the primary entry point to a `FractalServer` instance.

We can initialize a `FractalClient` by pointing it to a server instance. If you would like to start your own server see the setting up a server (NYI) section.

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

The `FractalClient` handles all communication to the `FractalServer` from the Python API layer. This includes adding new molecules, computations, collections, and querying for records. A `FractalClient` constructed without arguments will automatically connect to the MolSSI QCArchive server.

The `FractalClient` can also be initialized from a file which is useful so that addresses and username do not have to be retyped for everytime and reduces the chance that a username and password could accidentally be added to a version control system. Creation from file uses the classmethod `FractalClient.from_file()`, by default the client searches for a `qcportal_config.yaml` file in either the current working directory or from the canonical `~/.qca` folder.

3.10 Add/Query Objects

`Molecule`, `KeywordSet`, `Collection`, and `KVStore` objects are always added/queried directly to the server unlike compute objects as this particular set of structures are not acted upon by the server itself.

3.10.1 Adding Objects

Adding objects to the server uses the `client.add_*` commands and takes in a list of objects to add and returns the *ObjectId* of the object.

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

Adding the same molecule again will not add a new molecule and will always return the same *ObjectId*:

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

The order of *ObjectId* returned is identical to the order of molecules added.

Note: The *ObjectId* changes and is unique to a particular database.

3.10.2 Querying Objects

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

3.10.3 Molecules

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

3.11 Records Querying

Query documents, including projects ideas.

3.12 New Compute Tasks

Add new compute tasks and checking status.

3.13 API

3.13.1 Generics

from_file([load_path])	Creates a new FractalClient from file.
server_information()	Pull down various data on the connected server.

3.13.2 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.

3.13.3 Records Querying

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

3.13.4 New Compute Tasks

add_compute(program, method, basis, driver, ...)	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.

3.13.5 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: QueryObjectId = None*, *molecule_hash: QueryStr = None*, *molecular_formula: QueryStr = None*, *limit: Optional[int] = None*, *skip: int = 0*, *full_return: bool = False*) → List[qcelemental.models.molecule.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.
- **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[qcelemental.models.molecule.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: QueryObjectId = None*, ***, *hash_index: QueryStr = None*, *limit: Optional[int] = None*, *skip: int = 0*, *full_return: bool = False*) → 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*) → 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.

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*) → 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.

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*) → List[qcfractal.interface.models.common_models.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: QueryObjectId = None*, *task_id: QueryObjectId = None*, *program: QueryStr = None*, *molecule: QueryObjectId = None*, *driver: QueryStr = None*, *method: QueryStr = None*, *basis: QueryStr = None*, *keywords: QueryObjectId = None*, *status: QueryStr = 'COMPLETE'*, *limit: Optional[int] = None*, *skip: int = 0*, *projection: QueryProjection = None*, *full_return: bool = False*) → Union[List[RecordResult], 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
- **projection** (*QueryProjection, 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 projection, or a dictionary of results with projection.

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

qcportal.FractalClient.**query_procedures**(*self*, *id: QueryObjectId = None*, *task_id: QueryObjectId = None*, *procedure: QueryStr = None*, *program: QueryStr = None*, *hash_index: QueryStr = None*, *status: QueryStr = 'COMPLETE'*, *limit: Optional[int] = None*, *skip: int = 0*, *projection: QueryProjection = None*, *full_return: bool = False*) → Union[List[RecordBase], 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
- **projection** (*QueryProjection, 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 projection, or a dictionary of results with projection.

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

qcportal.FractalClient.**add_compute**(*self, program: str, method: str, basis: str, driver: str, keywords: Optional[qcfractal.interface.models.common_models.ObjectId], molecule: Union[qcfractal.interface.models.common_models.ObjectId, qcelemental.models.molecule.Molecule, List[Union[str, qcelemental.models.molecule.Molecule]]], priority: str = None, tag: str = None, full_return: bool = False*) → qcfractal.interface.models.rest_models.ComputeResponse

Adds a "single" compute to the server.

Parameters

- **program** (*str*) – The computational program to execute the result with (e.g., "rdkit", "psi4").
- **method** (*str*) – The computational method to use (e.g., "B3LYP", "PBE")
- **basis** (*str*) – The basis to apply to the computation (e.g., "cc-pVDZ", "6-31G")
- **driver** (*str*) – The primary result that the compute will aquire {"energy", "gradient", "hessian", "properties"}
- **keywords** (*Union[str, None]*) – The KeywordSet ObjectId to use with the given compute
- **molecule** (*Union[str, Molecule, List[Union[str, Molecule]]]*) – The Molecules or Molecule ObjectId's to compute with the above methods
- **priority** (*str, optional*) – The priority of the job {"HIGH", "MEDIUM", "LOW"}. Default is "MEDIUM".
- **tag** (*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

qcportal.FractalClient.**add_procedure**(*self, procedure: str, program: str, program_options: Dict[str, Any], molecule: Union[qcfractal.interface.models.common_models.ObjectId, qcelemental.models.molecule.Molecule, List[Union[str, qcelemental.models.molecule.Molecule]]], priority: str = None, tag: str = None, full_return: bool = False*) → qcfractal.interface.models.rest_models.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: QueryObjectId = None*, *hash_index: QueryStr = None*, *program: QueryStr = None*, *status: QueryStr = None*, *base_result: QueryStr = None*, *limit: Optional[int] = None*, *skip: int = 0*, *projection: QueryProjection = None*, *full_return: bool = False*) → 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.
- **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
- **projection** (*QueryProjection, 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 occured, the error message.

Return type List[Dict[str, Any]]

Examples

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

qcportal.FractalClient.**add_service**(*self*, *service: Union[qcfractal.interface.models.gridoptimization.GridOptimizationI qcfractal.interface.models.torsiondrive.TorsionDriveInput]*, *tag: Optional[str] = None*, *priority: Optional[str] = None*, *full_return: bool = False*) → qcfractal.interface.models.rest_models.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: QueryObjectId = None*, *procedure_id: QueryObjectId = None*, *hash_index: QueryStr = None*, *status: QueryStr = None*, *limit: Optional[int] = None*, *skip: int = 0*, *full_return: bool = False*) → 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]]

Examples

```
>>> client.query_services(id="5bd35af47b878715165f8225", projection={"status":␣
↪True})
[{"status": "RUNNING"}]
```

Developer Documentation

Contains in-depth developer documentation.

3.14 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.

3.15 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.

3.15.1 KV Store

class qcportal.models.rest_models.**KVStoreGETBody**

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 a dict for Id of the Key/Value object to fetch.

class qcportal.models.rest_models.**KVStoreGETResponse**

Parameters

- **meta** (*ResponseGETMeta*) – Standard Fractal Server response metadata
- **data** (*Dict[str, Any]*) – The entries of Key/Value object requested.

3.15.2 Molecule

class qcportal.models.rest_models.**MoleculeGETBody**

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**

Parameters

- **meta** (*ResponseGETMeta*) – Standard Fractal Server response metadata
- **data** (Molecule) – The List of Molecule objects found by the query.

class qcportal.models.rest_models.**MoleculePOSTBody**

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**

Parameters

- **meta** (*ResponsePOSTMeta*) – Standard Fractal Server response metadata
- **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).

3.15.3 Keywords

class qcportal.models.rest_models.**KeywordGETBody**

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**

Parameters

- **meta** (*ResponseGETMeta*) – Standard Fractal Server response metadata
- **data** (KeywordSet) – The KeywordSet found from in the database based on the query.

class qcportal.models.rest_models.**KeywordPOSTBody**

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**

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

3.15.4 Collections

class qcportal.models.rest_models.**CollectionGETBody**

Parameters

- **meta** (Meta, Optional) – Additional metadata to make with the query. Collections can only have a projection 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**

Parameters

- **meta** (*ResponseGETMeta*) – Standard Fractal Server response metadata
- **data** (*List[Dict[str, Any]]*) – The Collection objects returned by the server based on the query.

class qcportal.models.rest_models.**CollectionPOSTBody**

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**

Parameters

- **data** (*str, Optional*) – 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

3.15.5 Result

class qcportal.models.rest_models.**ResultGETBody**

Parameters

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

class qcportal.models.rest_models.**ResultGETResponse**

Parameters

- **meta** (*ResponseGETMeta*) – Standard Fractal Server response metadata
- **data** (Union[ResultRecord, List[Dict[str, 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.

3.15.6 Procedures

class qcportal.models.rest_models.**ProcedureGETBody**

Parameters

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

class qcportal.models.rest_models.**ProcedureGETResponse**

Parameters

- **meta** (*ResponseGETMeta*) – Standard Fractal Server response metadata
- **data** (*List[Dict[str, Any]]*) – The list of Procedure specs found based on the query.

3.15.7 Task Queue

class qcportal.models.rest_models.**TaskQueueGETBody**

Parameters

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

class qcportal.models.rest_models.**TaskQueueGETResponse**

Parameters

- **meta** (*ResponseGETMeta*) – Standard Fractal Server response metadata
- **data** (Union[TaskRecord, List[Dict[str, 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**

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**

Parameters

- **meta** (*ResponsePOSTMeta*) – Standard Fractal Server response metadata
- **data** (ComputeResponse) – Data returned from the server from adding a Task.

class qcportal.models.rest_models.**TaskQueuePUTBody**

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**

Parameters

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

3.15.8 Service Queue

class qcportal.models.rest_models.**ServiceQueueGETBody**

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**

Parameters

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

class qcportal.models.rest_models.**ServiceQueuePOSTBody**

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**

Parameters

- **meta** (*ResponsePOSTMeta*) – Standard Fractal Server response metadata
- **data** (ComputeResponse) – Data returned from the server from adding a Service.

class qcportal.models.rest_models.**ServiceQueuePUTBody**

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**

Parameters

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

3.15.9 Queue Manager

class qcportal.models.rest_models.**QueueManagerGETBody**

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**

Parameters

- **meta** (*ResponseGETMeta*) – Standard Fractal Server response metadata
- **data** (*List[Dict[str, Any]]*) – A list of tasks retrieved from the server to compute.

class qcportal.models.rest_models.**QueueManagerPOSTBody**

Parameters

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

class qcportal.models.rest_models.**QueueManagerPOSTResponse**

Parameters

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

class qcportal.models.rest_models.**QueueManagerPUTBody**

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**

Parameters

- **meta** (*Dict[str, Any], Default: {}*) – There is no metadata accepted, so an empty metadata is sent for completion.
- **data** (*Union[data_typing.Dict[str, int] type=int required, 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.

3.15.10 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**
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**
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.**ResponsePOSTMeta**
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.**QueryMeta**
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.**QueryMetaProjection**
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**
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** (*str, Optional*) – Optional queue tag to pull Tasks from.

3.16 Changelog

3.16.1 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.

3.16.2 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.

3.16.3 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.

3.16.4 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.

3.16.5 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.

3.16.6 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.

3.16.7 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.

3.16.8 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.

3.16.9 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.

3.16.10 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

3.16.11 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`).

3.16.12 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

3.16.13 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.

3.16.14 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”.

3.16.15 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.

3.16.16 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.

3.16.17 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.

3.16.18 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

L

M

O

P

Q

R

S

T

V