An introduction to task documents, schemas, and emmet

Introduction

If you have been running-workflows, you are now starting to generate data. `atomate2` stores both input and output data for every step of its workflows in Task Documents. Task Documents define a *schema* or structure for organizing information from different types of calculations, which then facilitates automatic processing with tools like `emmet` or `maggma`. This tutorial will familiarize you with these basic concepts.

Objectives

- Understand how `atomate2` stores and organizes calculation data
- Expain the meaning of a "Document Model" or schema
- Inspect a `TaskDoc` generated by `atomate2`

Prerequisites

To complete this tutorial, you need

- A working installation of `atomate2`
- (optional) to complete the running workflows tutorial.

How `atomate2` stores and organizes data.

As explained in Configure calculation output database, `atomate2` stores the results of every `Job` in a database. More specifically, `atomate2` uses a `maggma.Store` to interface with a data storage backend (usually MongoDB). Data is stored in a `JSON`-like or python `dict`-like format, which you can think of as a list of dictionaries, where each dictionary represents one `Job`. Each dictionary in the list is called a "document", so "document" refers to the output data from a single `Job`.

To facilitate automated processing and analysis, it's important that every document follows a consistent format. That's where schemas (also called "Document Models") come in.

Document Models

Schema for `Job`

Schema for `Job`

Document models define a specific format (i.e., structure and data types) for a given `Job` or calculation. `atomate2` uses pydantic to define these schemas. If you'd like to learn more about `pydantic`, we suggest reading this introduction. In brief, every Document Model in `atomate2` is an instance of `pydantic.BaseModel`. The `BaseModel` is then turned into a `dict` (serialized) before being inserted into the store.

To understand how this works, we are going to look at the output data from a structural relaxation for Si. If we examine the `docs` store after running this `Job`, we will see something similar to the following:


```
[
  {"uuid":"c2b5eb7d-838b-4dee-896f-95f21867b62b",
   "index":1,
   "output":{...},
   "completed_at":"2024-05-19T17:13:46.400349",
   "metadata":{},
   "hosts":["dbaebabf-134d-426a-b91c-15abf799da65"],
   "name":"relax",
   "@module":"jobflow.core.schemas",
   "@class":"JobStoreDocument",
   "@version":"0.1.17"
   },
]
```


This document follows a schema (`JobStoreDocument`, defined here) that contains information about the `Job`, such as:

- `uuid`: a unique identifier for the `Job`
- `output`: The actual job output (e.g., calculation results). We'll examine this in the next section.
- `completed_at`: The time the job was completed.
- `name`: The name of the job (in this case "relax" because we did a structure relaxation)
- `@module`, `@class`, `@version`: These keys store the specific origin and version of the document model so that it can be easily re-created from the `dict`.

Because every `atomate2` document is first created as a `JobStoreDocument` before being inserted into the database, you can be assured that every `Job` you run will contain these keys. Document Models have the additional benefit of validating the data types, so for example, `name` is guaranteed to a `str`, whereas `index` is guaranteed to be a `int`.

> Warning
>
> In this tutorial, we show only **excerpts** of the output data to highlight key points. For example, in the box above we have collapsed the `outputs` key. You are encouraged to open the complete output data (`.json` format) in a separate

> tab of your web browser and refer to it as you read through this tutorial.

Schema for `output`

The output data for the calculation itself (the contents of the `Job`) are stored in the `output` key. **The schema of `output` will vary depending on the type of calculation (e.g., VASP relaxation, Q-Chem static, etc.), but will always be consistent for a particular `Job` type**. In the case of a VASP calculation, the schema is called a `TaskDoc`.

That being said, most `Job` types have a few features in common, which we will highlight in our example. If we look at the top-level keys of `output` from the `JobStoreDocument` in the previous section, we see:


```
{
  "builder_meta": {...}
  "nsites": 2,
  "elements": ["Si"],
  "nelements": 1,
  "composition": {"Si": 2},
  "composition_reduced": {"Si": 1},
  "formula_pretty": "Si",
  "formula_anonymous": "A",
  "chemsys": "Si",
  "volume": 40.163300666862035,
  "density": 2.3223723738160613,
  "density_atomic": 20.081650333431018,
  "symmetry": {...},
  "tags": null,
  "dir_name": "/scratch/gpfs/.../job_2024-05-19-21-13-15-058677-64911",
  "state": "successful",
  "calcs_reversed": [...],
  "structure": {...},
  "task_type": "Structure Optimization",
  "task_id": null,
  "orig_inputs": {...},
  "input": {...},
  "output": {...},
  "@module": "emmet.core.tasks",
  "@class": "TaskDoc",
  "@version": null
}
```


Even though we are looking at an example for a VASP calculations, `atomate2` **uses hierarchichal or modular Document Models wherever possible**. Therefore, the Task Documents generated for other calculation types have the same general structure (e.g., `inputs`, `outputs`, structure metadata, `custodian`, `orig_inputs`, `calcs_reversed`, etc.)

We describe many of these top-level keys in more detail in the following subsections.

> Note
>
> You can also generate `TaskDoc` from VASP calculations that you have run manually. To do so, use the `from_directory` class method:

```
from emmet.core.tasks import TaskDoc
doc = TaskDoc.from_directory("<path/to/your/calculation/>")
```

Structure Metadata

The root level of the `TaskDoc` has keys containing basic structural information including:

- `nsites`: The number of sites
- `composition`: Full composition for the material.
- `elements`: List of elements in the material.
- `formula_pretty`: Cleaned representation of the formula.
- `chemsys`: dash-delimited string of elements in the material.

And more. These keys illustrate another principle of Document Models – they are **hierarchical**. Specifically, the structure metadata keys are populated by *another* `pydantic` schema called StructureMetadata defined in `emmet`. So the `TaskDoc` schema comprises several subsidiary models that organize different types of information, as discussed further below.

`structure`

The `structure` key contains the **final output structure** of the calculation as a serialized `pymatgen.Structure` object.


```
"structure": {
                "@module": "pymatgen.core.structure",
                "@class": "Structure",
                "charge": 0,
                "lattice": {...},
                "properties": {},
                "sites": [...]
            }
```


`builder_meta`

The `builder_meta` key contains information about the software used to generate the data in the `TaskDoc`. Here is the example from our structure relaxation:


```
"builder_meta": {
```

```
"builder_meta": {
                "emmet_version": "0.83.0",
                "pymatgen_version": "2024.4.13",
                "pull_request": null,
                "database_version": null,
                "build_date": "2024-05-19T21:13:45.541000",
                "license": null
            }
```


Calculation metadata: `dir_name`, `run_stats`, `task_label`, and `task_type`

- `task_label`: A user-definable label for the specific calculation
- `task_type`: A standardized label specifying the specific type of calculation being performed.
- `dir_name`: The path of the directory in which input/output files were written
- `run_stats`: Information about the walltime, cpu time, and computational resources utilized.


```
"task_type": "Structure Optimization",
"task_label": "relax",
"dir_name": "della-r3c1n3:/scratch/gpfs/ab6989/MPScanRelaxSet/atomate2/Ca_Mg_runs/job_2024-05-1
"run_stats": {
              "average_memory": 0,
              "max_memory": 241584,
              "elapsed_time": 18.833,
              "system_time": 1.114,
              "user_time": 16.166,
              "total_time": 17.28,
              "cores": 40
          }
```


Calculation Inputs

`atomate2` stores a record of not just the outputs of a calculation, but also the inputs, and any modifications that were made to those inputs.

The `input` key contains the **complete, final input data for the calculation**. It's schema is defined by InputDoc and includes everything one needs to specify a VASP calculation: a `Structure` object, INCAR settings, Pseudopotential specifications, etc. Let's just look at the top-level keys of our `input` section:


```
"input": {
          "structure": {...},
          "parameters": {...},
          "pseudo_potentials": {...},
          "potcar_spec": [ ... ],
          "xc_override": "PS",
          "is_lasph": true,
```

```
            "is_hubbard": false,
            "hubbards": {},
            "magnetic_moments": [
                0.6,
                0.6
            ]
        },
```


Calculation Outputs

Much like `input`, the `output` key is populated by a nested schema called `OutputDoc`. `OutputDoc` captures key summary information about the final result of a VASP calculation, including the `Structure`, final energy, `energy_per_atom`, and `bandgap`. In our example:


```
"output": {
            "structure": {...},
            "density": 2.3223723738160613,
            "energy": -11.48288783,
            "forces": [
                [
                    0,
                    0,
                    0
                ],
                [
                    0,
                    0,
                    0
                ]
            ],
            "stress": [
                [
                    0.04088458,
                    0,
                    0
                ],
                [
                    0,
                    0.04088458,
                    0
                ],
                [
                    0,
                    0,
                    0.04088458
                ]
            ],
            "energy_per_atom": -5.741443915,
```

```
            "bandgap": 0.45999999999999996
            },
```


`custodian` and `orig_inputs`

There is also a key called `orig_inputs` that contains the **original inputs given by the user** when the calculation was launched. It is possible for `input` and `orig_inputs` to differ if `custodian` is invoked to apply some adjustment to the calculation settings. `orig_inputs` is retained to provide 100% transparent provenance in such cases.

In addition, there is a `custodian` key that will capture and list any corrections or changes made by `custodian` during the calculation, as well as additional metadata. In our case, the `custodian.corrections` list is empty, which means that no modifications or restarts were made.


```
"custodian": [
            {
                "corrections": [],
                "job": {
                    "@module": "custodian.vasp.jobs",
                    "@class": "VaspJob",
                    "@version": "2024.4.18",
                    "vasp_cmd": [
                        "srun",
                        "/scratch/gpfs/ab6989/MPScanRelaxSet/atomate2/vasp_std"
                    ],
                    "output_file": "vasp.out",
                    "stderr_file": "std_err.txt",
                    "suffix": "",
                    "final": true,
                    "backup": true,
                    "auto_npar": false,
                    "auto_gamma": true,
                    "settings_override": null,
                    "gamma_vasp_cmd": [
                        "vasp_gam"
                    ],
                    "copy_magmom": false,
                    "auto_continue": false
                }
            }
        ],
```


`calcs_reversed`

Most Task Documents also contain a key called `calcs_reversed` which, as the name implies, contains calculation inputs and outputs **in reverse order**. These are stored as a `list`, so index `[0]` corresponds to the last (most recent) calculation, whereas index `[-1]` is the first calculation.

[0] corresponds to the last (most recent) calculation, whereas index [-1] is the first calculation.

Each element in the list contains `input`, `output`, `dir_name`, and other keys that give a complete specification of that calculation step.

In this example, there is only one element in `calcs_reversed`, because we just did a one-step `Job`. However, more complex workflows that contain multiple individual calculations would have an entry for each step.


```
"calcs_reversed": [
                    { "dir_name": "/scratch/gpfs/.../job_2024-05-19-21-13-15-058677-64911",
                      "vasp_version": "6.4.2",
                      "has_vasp_completed": "successful",
                      "input": {
                          "incar": {...},
                          "kpoints": {...},
                          "nkpoints": 20,
                          "potcar": ["PAW_PBE"],
                          "potcar_spec": [ ... ],
                          "potcar_type": ["PAW_PBE"],
                          "parameters": {...},
                          "lattice_rec": {...},
                          "structure": {...},
                          "is_hubbard": false,
                          "hubbards": {}
                      },
                      "output": {
                          "energy": -11.48288783,
                          "energy_per_atom": -5.741443915,
                          "structure": { .... },
                          "efermi": 5.96853235,
                          "is_metal": false,
                          "bandgap": 0.45999999999999996,
                          "cbm": 6.2225,
                          "vbm": 5.7625,
                          "is_gap_direct": false,
                          "direct_gap": 2.5146000000000006,
                          "transition": "(0.000,0.000,0.000)-(0.429,0.429,-0.000)",
                          "mag_density": -1.2698159551931228e-7,
                          "epsilon_static": null,
                          "epsilon_static_wolfe": null,
                          "epsilon_ionic": null,
                          "frequency_dependent_dielectric": {
                              "real": null,
                              "imaginary": null,
                              "energy": null
                          },
                          "ionic_steps": [ ... ],
                          "force_constants": null,
                          "normalmode_frequencies": null,
```

```
                    "normalmode_eigenvals": null,
                    "normalmode_eigenvecs": null,
                    "elph_displaced_structures": {
                        "temperatures": null,
                        "structures": null
                    },
                    "dos_properties": {...},
                    "run_stats": {
                        "average_memory": 0,
                        "max_memory": 241584,
                        "elapsed_time": 18.833,
                        "system_time": 1.114,
                        "user_time": 16.166,
                        "total_time": 17.28,
                        "cores": 40
                    }
                },
                "completed_at": "2024-05-19 17:13:34.897366",
                "task_name": "standard",
                "output_file_paths": {
                    "chgcar": "CHGCAR",
                    "aeccar0": "AECCAR0",
                    "aeccar1": "AECCAR1",
                    "aeccar2": "AECCAR2"
                },
                "bader": null,
                "ddec6": null,
                "run_type": "PBESol",
                "task_type": "Structure Optimization",
                "calc_type": "PBESol Structure Optimization"
            },
        ]
```


There is some redundancy in the information stored in `input`, `output`, and `calcs_reversed`, but this is by design. `input` and `output` capture summary information about the first and last steps of the `Job`, whereas `calcs_reversed` records practically every detail of all the intermediate steps.

> Note
>
> The `TaskDoc` `calcs_reversed` section is designed to capture **all the information that can be obtained from a VASP OUTCAR** (or `vasprun.xml`). Therefore, if you query your output data from the `atomate2` database, you should not need to manually look up anything from the OUTCAR. Chances are very good that the information is available somewhere in the `TaskDoc`. For example, you can get the electronic energy of the last SCF iteration (index `[-1]`) of the first ionic step (index `[0]`) in `calcs_reversed[0].output.ionic_steps[0].electronic_steps[0].e_fr_energy`.

emmet

Materials Project and Community document models

Most document models used by `atomate2` "live" in a separate package called `emmet` (or more specifically, `emmet-core`), which is installed by default as a dependency of `atomate2`. In general, mature document models that are used in the Materials Project website or database are developed in `emmet`, whereas some document models that are more niche or are in earlier stages of development may exist in `atomate2` itself.

Here is a partial listing of the codes and calculation types currently supported in `emmet-core`:

- VASP Structure optimization, static calculation,

Code-agnostic document models for analysis

So far, we have introduced Document Models as a way of parsing input and output data from a specific calculation software (VASP). However, document models are also useful for capturing data from "downstream" analysis that is not dependent on the specific code used to generate the data. Hence, **many document models in `emmet-core` are agnostic or independent of the specific software used in the initial calculation**.

To take a simple example, `emmet-core` contains a schema called `ElectronicStructureSummaryData` that stores the `band_gap`, conduction band minimum (`cbm`), valence band maximum (`vbm`), and Fermi level (`e_fermi`):

```
class ElectronicStructureBaseData(BaseModel):
    task_id: MPID = Field(
        ...,
        description="The source calculation (task) ID for the electronic structure data. "
        "This has the same form as a Materials Project ID.",
    )

    band_gap: float = Field(..., description="Band gap energy in eV.")

    cbm: Optional[Union[float, Dict]] = Field(
        None, description="Conduction band minimum data."
    )

    vbm: Optional[Union[float, Dict]] = Field(
        None, description="Valence band maximum data."
    )

    efermi: Optional[float] = Field(None, description="Fermi energy in eV.")
```

Clearly, this simple document model could be used to store output from any periodic DFT code.

Builders

Builders

`emmet-core` also defines `Builder` classes, which take raw calculation results (e.g., the `TaskDoc`) from our example, perform some analysis or transformation, and then create new document models in additional `Store`. This paradigm makes it possible to construct automated data processing pipelines, and is the basis for how the Materials Project database. For more about how builders and stores work together, see the `maggma` documentation.

Conclusion

In this tutorial, you learned

To see what workflows can be run, see the List of VASP workflows. They can be set up and run in the same way as in this tutorial.

At this point, you might:

- Learn how to chain workflows together: Chaining workflows.
- Learn how to customise VASP input settings: Modifying input sets.
- Configure atomate2 with FireWorks to manage and execute many workflows at once: Using atomate2 with FireWorks.
