asedb Documentation

Release 0.0.2a0

Alexander Tygesen

Mar 18, 2024

CONTENTS:

The `asedb` library provides a comprehensive ORM model for storing and querying atomic simulations using SQLAlchemy and ASE (Atomic Simulation Environment). It allows efficient storage and retrieval of atomic structures and their associated calculations.

The source code can be accessed on gitlab.

CHAPTER

ONE

INSTALLATION

The module can be installed via pip:

```
pip install asedb
```

CHAPTER

TWO

MODELS OVERVIEW

The core models in the `asedb` library include:

- `AtomsModel`: Represents atomic structures.
- `Element`: Stores information about elements within an atomic structure.
- `Calculation`: Holds calculation results associated with an atomic structure.

For more information on how to use the `asedb` package, see the *Quickstart*.

2.1 Quickstart

2.1.1 Engine & Session

First, you need to set up an engine & session as with any SQLAlchemy project. A helper function for creating a SQLite engine has been provided:

```
from asedb import make_sqlite_engine, initialize_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy import create_engine

engine = make_sqlite_engine("foo.db")  # Helper function to create a SQL Alchemy engine␣
↪for sqlite
initialize_engine(engine)  # Create necessary schema/tables

Session = sessionmaker(bind=engine)
session = Session()
```

Here, we used the *make_sqlite_engine* helper function to create the SQL Alchemy engine object for a sqlite database. You can also create your own engine, e.g. for postgres:

```
import urllib.parse
from sqlalchemy import create_engine

def make_engine():
    database = os.environ["PG_DATABASE"]
    user = os.environ["PG_USER"]
    password = urllib.parse.quote_plus(os.environ["PG_PASSWORD"])
    host = os.environ["PG_HOST"]
    port = os.environ["PG_PORT"]
    connection_string = (
```

(continues on next page)

(continued from previous page)

```
        f"postgresql+psycopg2://{user}:{password}@{host}:{port}/{database}"
    )
    return create_engine(connection_string)

engine = make_engine()
```

2.1.2 The AtomsModel

Querying Atoms

You can perform complex queries involving atomic structures and their elements. For example, to find all atomic structures containing Hydrogen (H) atoms where the count of H atoms is at least 2, you can use the following query:

```
from asedb import AtomsModel, Element

results = (
    session.query(AtomsModel)
    .join(Element)
    .where((Element.symbol == 'H') & (Element.count >= 2))
    .all()
)

for atom_model in results:
    print(atom_model.to_atoms())
```

This query joins the `AtomsModel` with the `Element` model, filters for structures containing at least two Hydrogen atoms, and retrieves the resulting atomic structures.

Another example, where we query the total number of atoms:

```
(
    session.query(AtomsModel)
    .where(AtomsModel.natoms>2)
    .all()
)
```

Or if you want a particular Atoms object with a previously known ID:

```
my_id = 2  # Example ID
session.query(AtomsModel).filter_by(id=my_id).scalar()
```

Working with AtomsModel

The `AtomsModel` class provides methods to convert between ASE's `Atoms` objects and the database models:

- `to_atoms()`: Converts the database model to an ASE `Atoms` object.
- `from_atoms(atoms, import_calculation=True)`: Converts an ASE `Atoms` object to the database model, optionally importing calculation results.

Here's an example of converting an ASE `Atoms` object to an `AtomsModel` and saving it to the database:

```
from ase import Atoms
from asedb import AtomsModel

# Create an ASE Atoms object
atoms = Atoms('H2O', positions=[[0, 0, 0], [0, 0, 1], [1, 0, 0]])

# Convert to AtomsModel and save to database
atoms_model = AtomsModel.from_atoms(atoms)
session.add(atoms_model)
session.commit()

print("Model ID:", atoms_model.id)  # The ID assigned to the model after the commit.

# Retrieve and convert back to ASE Atoms
retrieved_atoms_model = session.query(AtomsModel).first()
atoms = retrieved_atoms_model.to_atoms()
print(atoms)
```

Updating Atomic Structures

Atomic structures represented by the *AtomsModel* can be updated to reflect changes in their corresponding ASE *Atoms* objects. This is particularly useful when an atomic structure has been modified, such as changing atom positions, adding or removing atoms, or updating simulation parameters, and these changes need to be persisted in the database.

The `set_atoms` Method

The `set_atoms` method of the `AtomsModel` class provides a mechanism to update the model with a new or modified ASE *Atoms* object. This method overwrites the existing atomic structure information in the *AtomsModel* with the data from the provided *Atoms* object, including any changes made to the atomic configuration or associated calculation results.

Example Usage

Consider an existing *AtomsModel* instance that needs to be updated due to changes in the atomic structure or simulation results. The following steps demonstrate how to apply these updates:

1. Retrieve the existing `AtomsModel` from the database.
2. Modify the ASE `Atoms` object as required by your simulation or analysis workflow.
3. Use the `set_atoms` method on the existing `AtomsModel` to apply the updates.
4. Commit the changes to the database.

```
from ase import Atom
from asedb import AtomsModel

# Retrieve an existing AtomsModel from the database
atoms_model = session.query(AtomsModel).first()

# Get the ASE Atoms object from the model
atoms = atoms_model.to_atoms()
```

(continues on next page)

(continued from previous page)

```
# Modify the Atoms object (example: add a new atom)
atoms += Atom('O', position=[1.2, 0, 0])

# Update the AtomsModel with the modified Atoms object
atoms_model.set_atoms(atoms)

# Commit the changes to the database
session.add(atoms_model)
session.commit()
```

This process ensures that the `AtomsModel` in the database accurately reflects the updated atomic structure. The `set_atoms` method allows for flexible and dynamic updates to atomic structures, facilitating iterative workflows and adjustments to simulation parameters or configurations.

class asedb.atoms_model.**AtomsModel**(***kwargs*)

The primary class representing the SQLAlchemy model for an ASE Atoms object.

This class facilitates the storage and retrieval of atomic structures within a relational database, utilizing the SQLAlchemy ORM. It intends to seamlessly serialize and deserialize ASE Atoms objects, including their associated calculator results when available.

The main usage will be something like

```
atoms = ase.Atoms(...)
model = AtomsModel.from_atoms(atoms)
session.add(model)
session.commit()

# Load the model from the database
loaded = session.query(AtomsModel).first().to_atoms()
```

The AtomsModel will also serialize a calculator object into a *Calculation* if a calculator exists.

classmethod from_atoms(*atoms: Atoms*, *import_calculation: bool = True*, *project: None | str = None*) → *AtomsModel*

Helper method to instantiate an AtomsModel instance from an ASE Atoms object.

Parameters

- **atoms** (*ase.Atoms*) – The ASE Atoms instance.
- **import_calculation** (*bool, optional*) – Whether the calculator should be imported, if it exists. Defaults to True.
- **project** (*None | str, optional*) – An optional project name. Defaults to None.

Returns
The newly instantiated AtomsModel.

Return type
AtomsModel

get_element_counts() → dict[str, int]

Get the number of times a particular element occurs in the model.

Returns
A dictionary with element symbols as keys and their counts as values.

Return type
dict[str, int]

property has_calc: bool

Indicates whether the atoms object represented by the model has an associated calculator.

property pbc: ndarray

The full period boundary conditions.

set_atoms(*atoms: Atoms*, *import_calculation: bool = True*) → None

Read the current Atoms configurations, including the calculator, and save the state in the current AtomsModel instance.

If import_calculation is True, then the calculator object will also be serialized into a Calculation object, otherwise the calculator will be ignored.

to_atoms() → Atoms

Export the SQL Alchemy object as an ASE Atoms object. If a corresponding *Calculation* object exists, a SinglePointCalculator will be attached to the constructed Atoms object.

class asedb.atoms_model.**Calculation**(***kwargs*)

Represents the serialization of an ASE Calculator, encapsulating the results of computational chemistry calculations. This class stores various properties such as energy, free energy, magnetic moment, and the maximum force acting on atoms alongside arbitrary arrays of data like forces or stresses that are results of these calculations.

id

The primary key in the database.

Type
int

atoms_id

A foreign key linking to the *AtomsModel* this calculation is associated with.

Type
int

energy

The total energy from the calculation.

Type
float, optional

free_energy

The free energy from the calculation, if available.

Type
float, optional

magmom

The total magnetic moment from the calculation.

Type
float, optional

fmax

The maximum force acting on any atom in the structure, derived from the forces array.

Type
float, optional

arrays

A relationship to a collection of *CalcArray* instances that store arbitrary array results from the calculation.

Type
list[*CalcArray*]

The class provides methods to construct a *Calculation* instance from an ASE Calculator object, manage result arrays, and retrieve calculation results for re-creation of an ASE Calculator object for further analysis.

Example usage:

```
from ase.calculators.emt import EMT
from ase.build import molecule
from asedb import Calculation, AtomsModel

atoms = molecule('H2O')
atoms.calc = EMT()
energy = atoms.get_potential_energy()

model = AtomsModel.from_atoms(atoms)
# The calc object now contains the calculation results and can be associated with␣
↪an AtomsModel
assert energy == model.calculation.energy
```

add_array(*name: str*, *array: ndarray*) → None

Adds an array of calculation results to the *Calculation* object.

This method is used to store additional arrays of results, such as forces or stress tensors, that come from the calculation. If adding forces, the maximum force (*fmax*) is automatically updated.

Parameters

- **name** (*str*) – The name of the array (e.g., "forces", "stress").
- **array** (*np.ndarray*) – The numpy array containing the calculation results.

drop_array(*name: str*) → None

Removes an array of calculation results from the *Calculation* object.

This method allows for the removal of specific arrays of results, useful for correcting or updating calculation data.

Parameters
name (*str*) – The name of the array to remove (e.g., "forces", "stress").

Raises
ValueError – If the specified array name does not exist within the *Calculation* object.

classmethod from_calc(*calc: Calculator*) → *Calculation*

Constructs a *Calculation* instance from an ASE Calculator object, extracting relevant calculation results and arrays.

Parameters
calc (*AseCalculator*) – The ASE Calculator object from which to extract calculation results.

Returns
An instance of *Calculation* populated with results from the ASE Calculator.

Return type
Calculation

This method automatically extracts properties like energy, free energy, and magnetic moment as floats, and results like forces, stress, and magnetic moments as arrays, storing them for later reconstruction.

get_calc_kwargs() → Mapping[str, float | ndarray]

Retrieves calculation results stored in the *Calculation* object, formatted for re-creation of an ASE Calculator object.

This method facilitates the reconstruction of an ASE Calculator object from stored calculation results.

Returns

A dictionary of calculation properties and results,
ready to be passed to an ASE Calculator constructor.

Return type
Mapping[str, float | np.ndarray]

class asedb.atoms_model.**Element**(***kwargs*)

2.1.3 The Trajectory Model

In addition to atomic structures and calculations, the `asedb` library provides a `Trajectory` model for representing sequences of atomic structures, typically used to represent simulation trajectories or a series of states in a calculation.

Model Overview

The `Trajectory` model stores a collection of atomic structures (*AtomsModel* instances) as a sequence. This model facilitates the organization and retrieval of atomic configurations that are related as a time series or a progression of states.

The relationship between `Trajectory` and `AtomsModel` is many-to-many, represented by the *atoms_trajectory_mapping* table.

Note: An `AtomsModel` may be owned by a `Trajectory`, so if an `AtomsModel` is removed from the `atoms_list` property and comitted to the database, this atoms object may be lost.

Using the Trajectory Model

Creating a new trajectory involves adding `AtomsModel` instances to the `atoms_list` attribute of a `Trajectory` instance. Here's an example of how to create a trajectory, add atomic structures to it, and retrieve the structures as ASE `Atoms` objects:

```
from ase import Atoms
from asedb import Trajectory
from sqlalchemy.orm import Session

session = Session()

# Create a new Trajectory instance
trajectory = Trajectory(project='My Simulation Project')

# Add atomic structures to the trajectory
atoms1 = Atoms('H2', positions=[[0, 0, 0], [0, 0, 1]])
```

(continues on next page)

(continued from previous page)

```
atoms2 = Atoms('H2O', positions=[[0, 0, 0], [0, 0, 1], [1, 0, 0]])
trajectory.add_atoms(atoms1)
trajectory.add_atoms(atoms2)

session.add(trajectory)
session.commit()

# Retrieve the trajectory and its atomic structures
retrieved_trajectory = session.query(Trajectory).first()
atoms_list = retrieved_trajectory.to_atoms_list()
for atoms in atoms_list:
    print(atoms)
```

The `add_atoms` method accepts an ASE `Atoms` object, converts it to an `AtomsModel`, and adds it to the trajectory. The `to_atoms_list` method converts the stored `AtomsModel` instances back into a list of ASE `Atoms` objects.

2.1.4 Projects

Managing Projects with the *project* Attribute

The *asedb* library supports the management of atomic structures and trajectories across multiple projects using the *project* attribute. This feature allows users to categorize and isolate data within specific projects, simplifying data organization and retrieval. The *project* attribute is available in both the *AtomsModel* and *Trajectory* models, enabling consistent project-based filtering across different types of data.

Using the *project* Attribute

The *project* attribute can be set when creating new instances of *AtomsModel* or *Trajectory*, and can be used as a criterion for querying the database to retrieve only the data associated with a specific project.

When creating a new *AtomsModel* or *Trajectory*, specify the *project* attribute to categorize the data:

```
from ase import Atoms
from asedb import AtomsModel, Trajectory

# Create a Trajectory within a specific project
trajectory = Trajectory(project='Molecular Dynamics')

# Create an AtomsModel within a specific project
atoms = Atoms('H2O', positions=[[0, 0, 0], [0, 0, 1], [1, 0, 0]])

model = trajectory.add_atoms(atoms)  # Creates an returns a new AtomsModel instance
model.project = "Water Simulation"

session.add(trajectory)
session.commit()
```

Querying by Project

To retrieve data from a specific project, use the *project* attribute as a filter in your queries:

```
# Retrieve all AtomsModel instances from the 'Water Simulation' project
water_models = session.query(AtomsModel).filter(AtomsModel.project == 'Water Simulation
↪').all()

# Retrieve all Trajectory instances from the 'Molecular Dynamics' project
md_trajectories = session.query(Trajectory).filter(Trajectory.project == 'Molecular␣
↪Dynamics').all()
```

This approach allows for efficient organization and retrieval of project-specific data, ensuring that queries are streamlined and manageable.

2.2 Package API

2.2.1 asedb

asedb package

Submodules

asedb.abstract module

class asedb.abstract.**ArrayType**(**args: Any*, ***kwargs: Any*)

Bases: TypeDecorator

Custom type for saving/loading NumPy arrays.

compare_values(*x: Any*, *y: Any*) → bool

Given two values, compare them for equality.

By default this calls upon TypeEngine.compare_values() of the underlying "impl", which in turn usually uses the Python equals operator ==.

This function is used by the ORM to compare an original-loaded value with an intercepted "changed" value, to determine if a net change has occurred.

impl

alias of LargeBinary

process_bind_param(*value*, *dialect*)

Convert the array going into the database.

process_result_value(*value*, *dialect*)

Convert the result coming from the database.

class asedb.abstract.**NamedArray**(***kwargs: Any*)

Bases: Base

Base table object for storing a NumPy array along with a name and some metadata.

property array_meta: None | dict[str, Any]

array_meta_json: Mapped[str] = <sqlalchemy.orm.properties.MappedColumn object>

array_obj: Mapped[ndarray] = <sqlalchemy.orm.properties.MappedColumn object>

classmethod from_np_array(*name: str*, *array: ndarray*) → T_Arr

Construct an instance of the NamedArray model from a NumPy array.

get_array() → ndarray

Access the NumPy array object from the model.

id: Mapped[int] = <sqlalchemy.orm.properties.MappedColumn object>

last_update_time: Mapped[float] = <sqlalchemy.orm.properties.MappedColumn object>

name: Mapped[str] = <sqlalchemy.orm.properties.MappedColumn object>

set_array(*array: ndarray | Cell*) → None

Update the blob representing a NumPy array. The array will be serialized to a binary blob using the NumPy save function.

Note: Arbitrary Python objects are not allowed, as Pickle serialization is disabled by default for security purposes. Change the asedb.abstract.ALLOW_PICKLE variable to True to allow pickle serialization.

asedb.atoms_model module

class asedb.atoms_model.**AtomsArray**(***kwargs*)

Bases: *NamedArray*

array_meta_json: Mapped[str]

array_obj: Mapped[ndarray]

atoms_id: Mapped[int]

id: Mapped[int]

last_update_time: Mapped[float]

name: Mapped[str]

class asedb.atoms_model.**AtomsModel**(***kwargs*)

Bases: Base

The primary class representing the SQLAlchemy model for an ASE Atoms object.

This class facilitates the storage and retrieval of atomic structures within a relational database, utilizing the SQLAlchemy ORM. It intends to seamlessly serialize and deserialize ASE Atoms objects, including their associated calculator results when available.

The main usage will be something like

```
atoms = ase.Atoms(...)
model = AtomsModel.from_atoms(atoms)
session.add(model)
session.commit()

# Load the model from the database
loaded = session.query(AtomsModel).first().to_atoms()
```

The AtomsModel will also serialize a calculator object into a *Calculation* if a calculator exists.

arrays: Mapped[list[*AtomsArray*]]

calculation: Mapped[*Calculation*]

creation_time: Mapped[float]

elements: Mapped[list[*Element*]]

classmethod from_atoms(*atoms: Atoms*, *import_calculation: bool = True*, *project: None | str = None*) → *AtomsModel*

Helper method to instantiate an AtomsModel instance from an ASE Atoms object.

Parameters

- **atoms** (*ase.Atoms*) – The ASE Atoms instance.
- **import_calculation** (*bool, optional*) – Whether the calculator should be imported, if it exists. Defaults to True.
- **project** (*None | str, optional*) – An optional project name. Defaults to None.

Returns
The newly instantiated AtomsModel.

Return type
AtomsModel

get_element_counts() → dict[str, int]

Get the number of times a particular element occurs in the model.

Returns
A dictionary with element symbols as keys and their counts as values.

Return type
dict[str, int]

property has_calc: bool

Indicates whether the atoms object represented by the model has an associated calculator.

id: Mapped[int]

last_updated: Mapped[float]

natoms: Mapped[int]

property pbc: ndarray

The full period boundary conditions.

pbc_int: Mapped[int]

project: Mapped[str]

set_atoms(*atoms: Atoms*, *import_calculation: bool = True*) → None

Read the current Atoms configurations, including the calculator, and save the state in the current AtomsModel instance.

If import_calculation is True, then the calculator object will also be serialized into a Calculation object, otherwise the calculator will be ignored.

to_atoms() → Atoms

Export the SQL Alchemy object as an ASE Atoms object. If a corresponding *Calculation* object exists, a SinglePointCalculator will be attached to the constructed Atoms object.

class asedb.atoms_model.CalcArray(**kwargs)

Bases: *NamedArray*

array_meta_json: Mapped[str]

array_obj: Mapped[ndarray]

calc_id: Mapped[int]

id: Mapped[int]

last_update_time: Mapped[float]

name: Mapped[str]

class asedb.atoms_model.Calculation(**kwargs)

Bases: Base

Represents the serialization of an ASE Calculator, encapsulating the results of computational chemistry calculations. This class stores various properties such as energy, free energy, magnetic moment, and the maximum force acting on atoms alongside arbitrary arrays of data like forces or stresses that are results of these calculations.

id

The primary key in the database.

Type
int

atoms_id

A foreign key linking to the *AtomsModel* this calculation is associated with.

Type
int

energy

The total energy from the calculation.

Type
float, optional

free_energy

The free energy from the calculation, if available.

Type
float, optional

magmom

The total magnetic moment from the calculation.

Type
float, optional

fmax

The maximum force acting on any atom in the structure, derived from the forces array.

Type
float, optional

arrays

A relationship to a collection of *CalcArray* instances that store arbitrary array results from the calculation.

Type
list[*CalcArray*]

The class provides methods to construct a *Calculation* instance from an ASE Calculator object, manage result arrays, and retrieve calculation results for re-creation of an ASE Calculator object for further analysis.

Example usage:

```
from ase.calculators.emt import EMT
from ase.build import molecule
from asedb import Calculation, AtomsModel

atoms = molecule('H2O')
atoms.calc = EMT()
energy = atoms.get_potential_energy()

model = AtomsModel.from_atoms(atoms)
# The calc object now contains the calculation results and can be associated with␣
↪an AtomsModel
assert energy == model.calculation.energy
```

add_array(*name: str*, *array: ndarray*) → None

Adds an array of calculation results to the *Calculation* object.

This method is used to store additional arrays of results, such as forces or stress tensors, that come from the calculation. If adding forces, the maximum force (*fmax*) is automatically updated.

Parameters

- **name** (*str*) – The name of the array (e.g., “forces”, “stress”).
- **array** (*np.ndarray*) – The numpy array containing the calculation results.

arrays: Mapped[list[*CalcArray*]]

atoms_id: Mapped[int]

drop_array(*name: str*) → None

Removes an array of calculation results from the *Calculation* object.

This method allows for the removal of specific arrays of results, useful for correcting or updating calculation data.

Parameters
name (*str*) – The name of the array to remove (e.g., “forces”, “stress”).

Raises
ValueError – If the specified array name does not exist within the *Calculation* object.

energy: Mapped[float]

fmax: Mapped[float]

free_energy: Mapped[float]

classmethod from_calc(*calc: Calculator*) → *Calculation*

Constructs a *Calculation* instance from an ASE Calculator object, extracting relevant calculation results and arrays.

Parameters
calc (*AseCalculator*) – The ASE Calculator object from which to extract calculation results.

Returns
An instance of *Calculation* populated with results from the ASE Calculator.

Return type
Calculation

This method automatically extracts properties like energy, free energy, and magnetic moment as floats, and results like forces, stress, and magnetic moments as arrays, storing them for later reconstruction.

get_calc_kwargs() → Mapping[str, float | ndarray]

Retrieves calculation results stored in the *Calculation* object, formatted for re-creation of an ASE Calculator object.

This method facilitates the reconstruction of an ASE Calculator object from stored calculation results.

Returns

A dictionary of calculation properties and results,
ready to be passed to an ASE Calculator constructor.

Return type
Mapping[str, float | np.ndarray]

id: Mapped[int]

magmom: Mapped[float]

class asedb.atoms_model.**Element**(***kwargs*)

Bases: Base

atoms_id: Mapped[int]

count: Mapped[int]

id: Mapped[int]

symbol: Mapped[str]

asedb.initialization module

asedb.initialization.**create_schema**(*engine: Engine*)

asedb.initialization.**initialize_engine**(*engine: Engine*)

asedb.initialization.**make_sqlite_engine**(*filename: str*, *initialize: bool = False*) → Engine

Helper function to create a sqlite Engine, that disables the usage of the default schema.

asedb.properties module

class asedb.properties.**ArrayProperties**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: _PropertyBase

CHARGES = 'charges'

FORCES = 'forces'

MAGMOMS = 'magmoms'

STRESS = 'stress'

STRESSES = 'stresses'

class asedb.properties.**ValueProperties**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: _PropertyBase

ENERGY = 'energy'

FREE_ENERGY = 'free_energy'

MAGMOM = 'magmom'

asedb.time_utils module

asedb.time_utils.**datetime_from_timestamp**(*timestamp: float*) → DateTime

Convert a float POSIX time to a datetime object in UTC.

asedb.time_utils.**get_posix_timestamp**() → float

The current POSIX time as a float.

asedb.trajectory_model module

class asedb.trajectory_model.**Trajectory**(***kwargs*)

Bases: Base

add_atoms(*atoms: Atoms*) → *AtomsModel*

Add an Atoms object to the Trajectory. Returns the newly created AtomsModel instance, which belongs to the trajectory.

atoms_list: Mapped[list[*AtomsModel*]]

id: Mapped[int]

project: Mapped[str]

to_atoms_list() → list[Atoms]

asedb.utils module

asedb.utils.**float_or_none**(*value: Any*) → float | None

asedb.version module

asedb.version.**get_version**() → Version

Get the packaging Version object for the package.

Module contents

class asedb.**AtomsModel**(***kwargs*)

Bases: Base

The primary class representing the SQLAlchemy model for an ASE Atoms object.

This class facilitates the storage and retrieval of atomic structures within a relational database, utilizing the SQLAlchemy ORM. It intends to seamlessly serialize and deserialize ASE Atoms objects, including their associated calculator results when available.

The main usage will be something like

```
atoms = ase.Atoms(...)
model = AtomsModel.from_atoms(atoms)
session.add(model)
session.commit()

# Load the model from the database
loaded = session.query(AtomsModel).first().to_atoms()
```

The AtomsModel will also serialize a calculator object into a *Calculation* if a calculator exists.

arrays: Mapped[list[*AtomsArray*]]

calculation: Mapped[*Calculation*]

creation_time: Mapped[float]

elements: Mapped[list[*Element*]]

classmethod from_atoms(*atoms: Atoms*, *import_calculation: bool = True*, *project: None | str = None*) → *AtomsModel*

Helper method to instantiate an AtomsModel instance from an ASE Atoms object.

Parameters

- **atoms** (*ase.Atoms*) – The ASE Atoms instance.
- **import_calculation** (*bool, optional*) – Whether the calculator should be imported, if it exists. Defaults to True.
- **project** (*None | str, optional*) – An optional project name. Defaults to None.

Returns

The newly instantiated AtomsModel.

Return type
AtomsModel

`get_element_counts()` → dict[str, int]

Get the number of times a particular element occurs in the model.

Returns
A dictionary with element symbols as keys and their counts as values.

Return type
dict[str, int]

`property has_calc: bool`

Indicates whether the atoms object represented by the model has an associated calculator.

`id: Mapped[int]`

`last_updated: Mapped[float]`

`natoms: Mapped[int]`

`property pbc: ndarray`

The full period boundary conditions.

`pbc_int: Mapped[int]`

`project: Mapped[str]`

`set_atoms(atoms: Atoms, import_calculation: bool = True)` → None

Read the current Atoms configurations, including the calculator, and save the state in the current AtomsModel instance.

If import_calculation is True, then the calculator object will also be serialized into a Calculation object, otherwise the calculator will be ignored.

`to_atoms()` → Atoms

Export the SQL Alchemy object as an ASE Atoms object. If a corresponding *Calculation* object exists, a SinglePointCalculator will be attached to the constructed Atoms object.

`class asedb.Calculation(**kwargs)`

Bases: Base

Represents the serialization of an ASE Calculator, encapsulating the results of computational chemistry calculations. This class stores various properties such as energy, free energy, magnetic moment, and the maximum force acting on atoms alongside arbitrary arrays of data like forces or stresses that are results of these calculations.

`id`

The primary key in the database.

Type
int

`atoms_id`

A foreign key linking to the *AtomsModel* this calculation is associated with.

Type
int

energy

The total energy from the calculation.

Type
float, optional

free_energy

The free energy from the calculation, if available.

Type
float, optional

magmom

The total magnetic moment from the calculation.

Type
float, optional

fmax

The maximum force acting on any atom in the structure, derived from the forces array.

Type
float, optional

arrays

A relationship to a collection of *CalcArray* instances that store arbitrary array results from the calculation.

Type
list[*CalcArray*]

The class provides methods to construct a *Calculation* instance from an ASE Calculator object, manage result arrays, and retrieve calculation results for re-creation of an ASE Calculator object for further analysis.

Example usage:

```
from ase.calculators.emt import EMT
from ase.build import molecule
from asedb import Calculation, AtomsModel

atoms = molecule('H2O')
atoms.calc = EMT()
energy = atoms.get_potential_energy()

model = AtomsModel.from_atoms(atoms)
# The calc object now contains the calculation results and can be associated with␣
↪an AtomsModel
assert energy == model.calculation.energy
```

add_array(*name: str*, *array: ndarray*) → None

Adds an array of calculation results to the *Calculation* object.

This method is used to store additional arrays of results, such as forces or stress tensors, that come from the calculation. If adding forces, the maximum force (*fmax*) is automatically updated.

Parameters

- **name** (*str*) – The name of the array (e.g., “forces”, “stress”).
- **array** (*np.ndarray*) – The numpy array containing the calculation results.

arrays: Mapped[list[*CalcArray*]]

atoms_id: Mapped[int]

drop_array(*name: str*) → None

Removes an array of calculation results from the *Calculation* object.

This method allows for the removal of specific arrays of results, useful for correcting or updating calculation data.

Parameters
name (*str*) – The name of the array to remove (e.g., "forces", "stress").

Raises
ValueError – If the specified array name does not exist within the *Calculation* object.

energy: Mapped[float]

fmax: Mapped[float]

free_energy: Mapped[float]

classmethod from_calc(*calc: Calculator*) → *Calculation*

Constructs a *Calculation* instance from an ASE Calculator object, extracting relevant calculation results and arrays.

Parameters
calc (*AseCalculator*) – The ASE Calculator object from which to extract calculation results.

Returns
An instance of *Calculation* populated with results from the ASE Calculator.

Return type
Calculation

This method automatically extracts properties like energy, free energy, and magnetic moment as floats, and results like forces, stress, and magnetic moments as arrays, storing them for later reconstruction.

get_calc_kwargs() → Mapping[str, float | ndarray]

Retrieves calculation results stored in the *Calculation* object, formatted for re-creation of an ASE Calculator object.

This method facilitates the reconstruction of an ASE Calculator object from stored calculation results.

Returns

A dictionary of calculation properties and results,
ready to be passed to an ASE Calculator constructor.

Return type
Mapping[str, float | np.ndarray]

id: Mapped[int]

magmom: Mapped[float]

class asedb.**Element**(***kwargs*)

Bases: Base

atoms_id: Mapped[int]

count: Mapped[int]

id: Mapped[int]

symbol: Mapped[str]

class asedb.**Trajectory**(***kwargs*)

Bases: Base

add_atoms(*atoms: Atoms*) → *AtomsModel*

Add an Atoms object to the Trajectory. Returns the newly created AtomsModel instance, which belongs to the trajectory.

atoms_list: Mapped[list[*AtomsModel*]]

id: Mapped[int]

project: Mapped[str]

to_atoms_list() → list[Atoms]

asedb.**initialize_engine**(*engine: Engine*)

asedb.**make_sqlite_engine**(*filename: str*, *initialize: bool = False*) → Engine

Helper function to create a sqlite Engine, that disables the usage of the default schema.

2.3 License

PYTHON MODULE INDEX

a

INDEX

F

G

H

I

L

M

N

P

S

T

V