SQLAthanor Documentation

Release 0.8.0

Insight Industry Inc.

Dec 27, 2021

Contents:

1 Quickstart: Patterns and Best Practices **3**
- 1.1 Installation . . . 3
- 1.2 Meta Configuration Pattern . . . 4
- 1.3 Declarative Configuration Pattern . . . 5
- 1.4 Serializing Model Instances . . . 6
- 1.5 Updating a Model Instance . . . 6
- 1.6 Creating a New Model Instance . . . 7
- 1.7 Error Handling . . . 7
  - 1.7.1 Errors During Serialization . . . 7
  - 1.7.2 Errors During De-serialization . . . 8
- 1.8 Password De-serialization . . . 9
- 1.9 Programmatically Generating Models . . . 11
- 1.10 Using SQLAthanor with SQLAlchemy Reflection . . . 12
- 1.11 Using SQLAthanor with Automap . . . 14
- 1.12 Using SQLAthanor with Flask-SQLAlchemy . . . 16
- 1.13 Generating SQLAlchemy Tables Programmatically . . . 16

2 Using SQLAthanor **19**
- 2.1 Introduction . . . 20
  - 2.1.1 What is Serialization? . . . 20
  - 2.1.2 Why **SQLAthanor**? . . . 21
  - 2.1.3 **SQLAthanor** vs. Alternatives . . . 22
- 2.2 SQLAthanor Features . . . 24
- 2.3 Overview . . . 25
  - 2.3.1 How SQLAthanor Works . . . 25
- 2.4 1. Installing SQLAthanor . . . 26
  - 2.4.1 Dependencies . . . 26
- 2.5 2. Import SQLAthanor . . . 27
- 2.6 3. Define Your Models . . . 28
- 2.7 4. Configure Serialization/De-serialization . . . 30
  - 2.7.1 Declarative Configuration . . . 31
  - 2.7.2 Meta Configuration . . . 34
  - 2.7.3 Meta Configuration vs Declarative Configuration . . . 39
  - 2.7.4 Why Two Configuration Approaches? . . . 41
  - 2.7.5 Configuring at Runtime . . . 41
- 2.8 5. Configuring Pre-processing and Post-processing . . . 42

2.8.1 Serialization Pre-processing . . . 42
2.8.2 De-serialization Post-processing . . . 43
2.9 6. Serializing a Model Instance . . . 44
2.9.1 Nesting Complex Data . . . 44
2.9.2 to_csv() . . . 44
2.9.3 to_json() . . . 45
2.9.4 to_yaml() . . . 46
2.9.5 to_dict() . . . 47
2.9.6 dump_to_csv() . . . 47
2.9.7 dump_to_json() . . . 48
2.9.8 dump_to_yaml() . . . 49
2.9.9 dump_to_dict() . . . 50
2.10 7. Deserializing Data . . . 50
2.10.1 Creating New Instances . . . 51
2.10.2 Updating Instances . . . 54
2.11 Using Declarative Reflection with SQLAthanor . . . 58
2.12 Using Automap with SQLAthanor . . . 59
2.13 Generating SQLAlchemy Tables... . . . 60
2.13.1 ...from Serialized Data . . . 60
2.13.2 ... from Pydantic Models . . . 62

3 SQLAthanor and Pydantic 63
3.1 SQLAthanor, Pydantic, and FastAPI . . . 63
3.2 Generating and Configuring Model Classes Using Pydantic . . . 67
3.3 Generating Tables from Pydantic Models . . . 69
3.4 Configuring Attributes from Pydantic Models . . . 70

4 API Reference 73
4.1 Declarative ORM . . . 74
4.1.1 BaseModel . . . 74
4.1.2 declarative_base() . . . 99
4.1.3 @as_declarative . . . 99
4.1.4 generate_model_from_csv() . . . 100
4.1.5 generate_model_from_json() . . . 101
4.1.6 generate_model_from_yaml() . . . 103
4.1.7 generate_model_from_dict() . . . 105
4.1.8 generate_model_from_pydantic() . . . 106
4.2 Schema . . . 108
4.2.1 Column . . . 108
4.2.2 relationship() . . . 110
4.2.3 Table . . . 111
4.3 Attribute Configuration . . . 118
4.3.1 AttributeConfiguration . . . 118
4.3.2 validate_serialization_config() . . . 126
4.4 Flask-SQLAlchemy / Flask-SQLAthanor . . . 126
4.4.1 initialize_flask_sqlathanor() . . . 126
4.4.2 FlaskBaseModel . . . 126
4.5 Automap . . . 127
4.6 SQLAthanor Internals . . . 127
4.6.1 RelationshipProperty . . . 127

5 Default Serialization Functions 129

6 Default De-serialization Functions 131

7 Error Reference **133**
7.1 Handling Errors . . . 134
7.1.1 Stack Traces . . . 134
7.1.2 Errors During Serialization . . . 134
7.1.3 Errors During De-Serialization . . . 135
7.2 SQLAthanor Errors . . . 136
7.2.1 SQLAthanorError (from `ValueError`) . . . 136
7.2.2 ConfigurationError (from `SQLAthanorError`) . . . 136
7.2.3 SQLAlchemySupportError (from `SQLAthanorError`) . . . 136
7.2.4 InvalidFormatError (from `SQLAthanorError`) . . . 136
7.2.5 SerializationError (from `SQLAthanorError`) . . . 136
7.2.6 ValueSerializationError (from `SerializationError`) . . . 136
7.2.7 SerializableAttributeError (from `SerializationError`) . . . 137
7.2.8 MaximumNestingExceededError (from `SerializationError`) . . . 137
7.2.9 UnsupportedSerializationError (from `SerializationError`) . . . 137
7.2.10 DeserializationError (from `SQLAthanorError`) . . . 137
7.2.11 CSVStructureError (from `DeserializationError`) . . . 137
7.2.12 JSONParseError (from `DeserializationError`) . . . 137
7.2.13 YAMLParseError (from `DeserializationError`) . . . 137
7.2.14 DeserializableAttributeError (from `DeserializationError`) . . . 138
7.2.15 ValueDeserializationError (from `DeserializationError`) . . . 138
7.2.16 UnsupportedValueTypeError (from `DeserializationError`) . . . 138
7.2.17 UnsupportedDeserializationError (from `DeserializationError`) . . . 138
7.2.18 ExtraKeyError (from `DeserializationError`) . . . 138
7.3 SQLAthanor Warnings . . . 138
7.3.1 SQLAthanorWarning (from `UserWarning`) . . . 138
7.3.2 MaximumNestingExceededWarning (from `SQLAthanorWarning`) . . . 138

8 Contributing to SQLAthanor **139**
8.1 Design Philosophy . . . 140
8.2 Style Guide . . . 140
8.2.1 Basic Conventions . . . 140
8.2.2 Naming Conventions . . . 141
8.2.3 Design Conventions . . . 141
8.2.4 Documentation Conventions . . . 142
8.3 Dependencies . . . 143
8.4 Preparing Your Development Environment . . . 143
8.5 Ideas and Feature Requests . . . 143
8.6 Testing . . . 144
8.7 Submitting Pull Requests . . . 144
8.8 Building Documentation . . . 144
8.9 Contributors . . . 144
8.10 References . . . 144

9 Testing SQLAthanor **145**
9.1 Testing Philosophy . . . 145
9.2 Test Organization . . . 146
9.3 Configuring & Running Tests . . . 146
9.3.1 Installing with the Test Suite . . . 146
9.3.2 Command-line Options . . . 146
9.3.3 Configuration File . . . 146
9.3.4 Running Tests . . . 147
9.4 Skipping Tests . . . 147
9.5 Incremental Tests . . . 147

10 Release History **149**
10.1 Release 0.8.0 . . . 150
10.1.1 New Features . . . 150
10.1.2 Other Changes . . . 151
10.2 Release 0.7.0 . . . 151
10.2.1 Bug Fixes . . . 151
10.2.2 Other Changes . . . 151
10.3 Release 0.6.0 . . . 151
10.3.1 Bug Fixes . . . 151
10.3.2 Other Changes . . . 151
10.4 Release 0.5.1 . . . 151
10.4.1 Bug Fixes . . . 152
10.4.2 Other Changes . . . 152
10.5 Release 0.5.0 . . . 152
10.5.1 New Features . . . 152
10.5.2 Bug Fixes . . . 152
10.6 Release 0.4.0 . . . 152
10.6.1 Bug Fixes . . . 153
10.6.2 New Features . . . 153
10.6.3 Other Changes . . . 153
10.7 Release 0.3.1 . . . 153
10.7.1 Bug Fixes . . . 153
10.7.2 Other Changes . . . 153
10.8 Release 0.3.0 . . . 153
10.8.1 New Features . . . 154
10.8.2 Other Changes . . . 154
10.9 Release 0.2.2 . . . 154
10.9.1 Bugs Fixed . . . 154
10.9.2 Other Changes . . . 154
10.10 Release 0.2.1 . . . 154
10.10.1 Bugs Fixed . . . 154
10.11 Release 0.2.0 . . . 155
10.11.1 Features Added . . . 155
10.12 Release 0.1.1 . . . 155
10.13 Release 0.1.0 . . . 155

11 Glossary **157**

12 SQLAthanor License **161**

13 Installation **163**
13.1 Dependencies . . . 163

14 Why SQLAthanor? **165**
14.1 Key SQLAthanor Features . . . 166
14.2 **SQLAthanor** vs Alternatives . . . 166

15 Hello, World and Basic Usage **171**
15.1 1. Import SQLAthanor . . . 171
15.2 2. Declare Your Models . . . 173
15.3 3. Serialize Your Model Instance . . . 177
15.4 4. De-serialize a Model Instance . . . 177

16 Questions and Issues **181**

17 Contributing 183

18 Testing 185

19 License 187

20 Indices and tables 189

Python Module Index 191

Index 193

Serialization/De-serialization Support for the SQLAlchemy Declarative ORM

Version Compatability

SQLAthanor is designed to be compatible with:

- Python 2.7 and Python 3.4 or higher, and
- SQLAlchemy 0.9 or higher

Branch	Unit Tests
latest	
v.0.8	
v.0.7	
v.0.6	
v.0.5	
v.0.4	
v.0.3	
v.0.2	
v.0.1	
develop	

CHAPTER 1

Quickstart: Patterns and Best Practices

- *Installation*
- *Meta Configuration Pattern*
- *Declarative Configuration Pattern*
- *Serializing Model Instances*
- *Updating a Model Instance*
- *Creating a New Model Instance*
- *Error Handling*
  - *Errors During Serialization*
  - *Errors During De-serialization*
- *Password De-serialization*
- *Programmatically Generating Models*
- *Using SQLAthanor with SQLAlchemy Reflection*
- *Using SQLAthanor with Automap*
- *Using SQLAthanor with Flask-SQLAlchemy*
- *Generating SQLAlchemy Tables Programmatically*

1.1 Installation

To install **SQLAthanor**, just execute:

```
$ pip install sqlathanor
```

1.2 Meta Configuration Pattern

See also:

- *Configuring Serialization/De-serialization* > *Meta Configuration*

```
from sqlathanor import declarative_base, Column, relationship, AttributeConfiguration

from sqlalchemy import Integer, String
from sqlalchemy.ext.hybrid import hybrid_property
from sqlalchemy.ext.associationproxy import association_proxy

BaseModel = declarative_base()

class User(BaseModel):
  __tablename__ = 'users'

  __serialization__ = [AttributeConfiguration(name = 'id',
                                              supports_csv = True,
                                              csv_sequence = 1,
                                              supports_json = True,
                                              supports_yaml = True,
                                              supports_dict = True,
                                              on_serialize = None,
                                              on_deserialize = None),
                       AttributeConfiguration(name = 'addresses',
                                              supports_json = True,
                                              supports_yaml = (True, True),
                                              supports_dict = (True, False),
                                              on_serialize = None,
                                              on_deserialize = None),
                       AttributeConfiguration(name = 'hybrid',
                                              supports_csv = True,
                                              csv_sequence = 2,
                                              supports_json = True,
                                              supports_yaml = True,
                                              supports_dict = True,
                                              on_serialize = None,
                                              on_deserialize = None)]
                       AttributeConfiguration(name = 'keywords',
                                              supports_csv = False,
                                              supports_json = True,
                                              supports_yaml = True,
                                              supports_dict = True,
                                              on_serialize = None,
                                              on_deserialize = None)]
                       AttributeConfiguration(name = 'python_property',
                                              supports_csv = (False, True),
                                              csv_sequence = 3,
                                              supports_json = (False, True),
                                              supports_yaml = (False, True),
                                              supports_dict = (False, True),
```

(continues on next page)

(continued from previous page)

```
                                                on_serialize = None,
                                                on_deserialize = None)]

  id = Column('id',
              Integer,
              primary_key = True)

  addresses = relationship('Address',
                           backref = 'user')

  _hybrid = 1

  @hybrid_property
  def hybrid(self):
      return self._hybrid

  @hybrid.setter
  def hybrid(self, value):
      self._hybrid = value

  @hybrid.expression
  def hybrid(cls):
    return False

  keywords = association_proxy('keywords', 'keyword')

  @property
  def python_property(self):
    return self._hybrid * 2
```

1.3 Declarative Configuration Pattern

See also:

- *Configuring Serialization/De-serialization* > *Declarative Configuration*

```
from sqlathanor import declarative_base, Column, relationship

from sqlalchemy import Integer, String

BaseModel = declarative_base()

class User(BaseModel):
  __tablename__ = 'users'

  id = Column('id',
              Integer,
              primary_key = True,
              supports_csv = True,
              csv_sequence = 1,
              supports_json = True,
              supports_yaml = True,
              supports_dict = True,
```

(continues on next page)

(continued from previous page)

```
            on_serialize = None,
            on_deserialize = None)

  addresses = relationship('Address',
                           backref = 'user',
                           supports_json = True,
                           supports_yaml = (True, True),
                           supports_dict = (True, False),
                           on_serialize = None,
                           on_deserialize = None)
```

1.4 Serializing Model Instances

See also:

- *Serializing a Model Instance*
- *to_csv()*
- *to_json()*
- *to_yaml()*
- *to_dict()*

```
# For a SQLAlchemy Model Class named "User" with an instance named "user":

as_csv = user.to_csv()      # CSV
as_json = user.to_json()    # JSON
as_yaml = user.to_yaml()    # YAML
as_dict = user.to_dict()    # dict
```

1.5 Updating a Model Instance

See also:

- *De-serializing Data* > *Updating Instances*
- *update_from_csv()*
- *update_from_json()*
- *update_from_yaml()*
- *update_from_dict()*

```
# For a SQLAlchemy Model Class named "User" with an instance named "user"
# and serialized objects "as_csv" (string), "as_json" (string),
# "as_yaml" (string), and "as_dict" (dict):

user.update_from_csv(as_csv)    # CSV
user.update_from_json(as_json) # JSON
```

(continues on next page)

(continued from previous page)

```
user.update_from_yaml(as_yaml) # YAML
user.update_from_dict(as_dict) # dict
```

1.6 Creating a New Model Instance

See also:

- *De-serializing Data* > *Creating New Instances*
- *new_from_csv()*
- *new_from_json()*
- *new_from_yaml()*
- *new_from_dict()*

```
# For a SQLAlchemy Model Class named "User" and serialized objects "as_csv"
# (string), "as_json" (string), "as_yaml" (string), and "as_dict" (dict):

user = User.new_from_csv(as_csv)    # CSV
user = User.new_from_json(as_json) # JSON
user = User.new_from_yaml(as_yaml) # YAML
user = User.new_from_dict(as_dict) # dict
```

1.7 Error Handling

1.7.1 Errors During Serialization

See also:

- *Error Reference*
- *Serializing a Model Instance*
- *Default Serialization Functions*

```
from sqlathanor.errors import SerializableAttributeError, \
  UnsupportedSerializationError, MaximumNestingExceededError

# For a SQLAlchemy Model Class named "User" and a model instance named "user".

try:
  as_csv = user.to_csv()
  as_json = user.to_json()
  as_yaml = user.to_yaml()
  as_dict = user.to_dict()
except SerializableAttributeError as error:
  # Handle the situation where "User" model class does not have any attributes
  # serializable to JSON.
```

(continues on next page)

(continued from previous page)

```
  pass
except UnsupportedSerializationError as error:
  # Handle the situation where one of the "User" model attributes is of a data
  # type that does not support serialization.
  pass
except MaximumNestingExceededError as error:
  # Handle a situation where "user.to_json()" received max_nesting less than
  # current_nesting.
  #
  # This situation is typically an error on the programmer's part, since
  # SQLAthanor by default avoids this kind of situation.
  #
  # Best practice is simply to let this exception bubble up.
  raise error
```

1.7.2 Errors During De-serialization

See also:

- *Error Reference*
- *De-serializing Data*
- *Configuring Pre-processing and Post-processing*

```
from sqlathanor.errors import DeserializableAttributeError, \
  CSVStructureError, DeserializationError, ValueDeserializationError, \
  ExtraKeysError, UnsupportedDeserializationError

# For a SQLAlchemy Model Class named "User" and a model instance named "user",
# with serialized data in "as_csv", "as_json", "as_yaml", and "as_dict" respectively.

try:
  user.update_from_csv(as_csv)
  user.update_from_json(as_json)
  user.update_from_yaml(as_yaml)
  user.update_from_dict(as_dict)

  new_user = User.new_from_csv(as_csv)
  new_user = User.new_from_json(as_json)
  new_user = User.new_from_yaml(as_yaml)
  new_user = User.new_from_dict(as_dict)
except DeserializableAttributeError as error:
  # Handle the situation where "User" model class does not have any attributes
  # de-serializable from the given format (CSV, JSON, YAML, or dict).
  pass
except DeserializationError as error:
  # Handle the situation where the serialized object ("as_csv", "as_json",
  # "as_yaml", "as_dict") cannot be parsed, for example because it is not
  # valid JSON, YAML, or dict.
  pass
except CSVStructureError as error:
  # Handle the situation where the structure of "as_csv" does not match the
  # expectation configured for the "User" model class.
  raise error
except ExtraKeysError as error:
```

(continues on next page)

(continued from previous page)

```
  # Handle the situation where the serialized object ("as_json",
  # "as_yaml", "as_dict") may have unexpected keys/attributes and
  # the error_on_extra_keys argument is False.
  #
  # Applies to: *_from_json(), *_from_yaml(), and *_from_dict() methods
  pass
except ValueDeserializationError as error:
  # Handle the situation where an input value in the serialized object
  # raises an exception in the deserialization post-processing function.
  pass
except UnsupportedDeserializationError as error:
  # Handle the situation where the de-serialization process attempts to
  # assign a value to an attribute that does not support de-serialization.
  pass
```

1.8 Password De-serialization

See also:

- *Configuring Pre-processing and Post-processing* > *De-serialization Post-processing*
- *Deserialization Functions*
- *Default Deserialization Functions*

Meta Configuration

Declarative Configuration

```
from sqlathanor import declarative_base, Column, AttributeConfiguration

from sqlalchemy import Integer, String

def my_encryption_function(value):
  """Function that accepts an inbound password ``value`` and returns its
  encrypted value."""

  # ENCRYPTION LOGIC GOES HERE

  return encrypted_value

BaseModel = declarative_base()

class User(BaseModel):
  __tablename__ = 'users'

  __serialization__ = [AttributeConfiguration(name = 'id',
                                              supports_csv = True,
                                              csv_sequence = 1,
                                              supports_json = True,
                                              supports_yaml = True,
                                              supports_dict = True,
                                              on_serialize = None,
                                              on_deserialize = None),
```

(continues on next page)

(continued from previous page)

```
                        AttributeConfiguration(name = 'password',
                                               supports_csv = (True, False),
                                               supports_json = (True, False),
                                               supports_yaml = (True, False),
                                               supports_dict = (True, False),
                                               on_serialize = None,
                                               on_deserialize = my_encryption_
↪function)]

  id = Column('id',
              Integer,
              primary_key = True)

  password = Column('password', String(255))
```

```
from sqlathanor import declarative_base, Column

from sqlalchemy import Integer, String

def my_encryption_function(value):
  """Function that accepts an inbound password ``value`` and returns its
  encrypted value."""

  # ENCRYPTION LOGIC GOES HERE

  return encrypted_value

BaseModel = declarative_base()

class User(BaseModel):
  __tablename__ = 'users'

  id = Column('id',
              Integer,
              primary_key = True,
              supports_csv = True,
              csv_sequence = 1,
              supports_json = True,
              supports_yaml = True,
              supports_dict = True,
              on_serialize = None,
              on_deserialize = None)

  password = Column('password',
                    String(255),
                    supports_csv = (True, False),
                    csv_sequence = 2,
                    supports_json = (True, False),
                    supports_yaml = (True, False),
                    supports_dict = (True, False),
                    on_serialize = None,
                    on_deserialize = my_encryption_function)
```

1.9 Programmatically Generating Models

New in version 0.3.0: generation from CSV, JSON, YAML, or `dict`

New in version 0.8.0: generation from Pydantic models

See also:

- *generate_model_from_csv()*
- *generate_model_from_json()*
- *generate_model_from_yaml()*
- *generate_model_from_dict()*
- *generate_model_from_pydantic()*

CSV

JSON

YAML

`dict`

Pydantic

```
# FROM CSV:
from sqlathanor import generate_model_from_csv

# Assuming that "csv_data" contains your CSV data
CSVModel = generate_model_from_csv(csv_data,
                                   tablename = 'my_table_name',
                                   primary_key = 'id')
```

```
from sqlathanor import generate_model_from_json

# Assuming that "json_string" contains your JSON data in a string
JSONModel = generate_model_from_json(json_string,
                                     tablename = 'my_table_name',
                                     primary_key = 'id')
```

```
from sqlathanor import generate_model_from_yaml

# Assuming that "yaml_string" contains your YAML data in a string
YAMLModel = generate_model_from_yaml(yaml_string,
                                     tablename = 'my_table_name',
                                     primary_key = 'id')
```

```
from sqlathanor import generate_model_from_dict

# Assuming that "yaml_string" contains your YAML data in a string
DictModel = generate_model_from_dict(dict_string,
                                     tablename = 'my_table_name',
                                     primary_key = 'id')
```

```
from sqlathanor import generate_model_from_pydantic

# Assumes that "PydanticReadModel" and "PydanticWriteModel" contain the Pydantic
```

(continues on next page)

(continued from previous page)

```
# models for the object/resource you are representing.
PydanticModel = generate_model_from_pydantic([PydanticReadModel, PydanticWriteModel],
                                             tablename = 'my_table_name',
                                             primary_key = 'id')
```

1.10 Using SQLAthanor with SQLAlchemy Reflection

See also:

- *Using Declarative Reflection with SQLAthanor*
- **SQLAlchemy**: Reflecting Database Objects
- **SQLAlchemy**: Using Reflection with Declarative

Meta Approach

Declarative: with Table

Declarative: without Table

```
from sqlathanor import declarative_base, Column, AttributeConfiguration

from sqlalchemy import create_engine, Table

BaseModel = declarative_base()

engine = create_engine('... ENGINE CONFIGURATION GOES HERE ...')
# NOTE: Because reflection relies on a specific SQLAlchemy Engine existing, presumably
# you would know how to configure / instantiate your database engine using SQLAlchemy.
# This is just here for the sake of completeness.

class ReflectedUser(BaseModel):
    __table__ = Table('users',
                      BaseModel.metadata,
                      autoload = True,
                      autoload_with = engine)

    __serialization__ = [AttributeConfiguration(name = 'id',
                                                supports_csv = True,
                                                csv_sequence = 1,
                                                supports_json = True,
                                                supports_yaml = True,
                                                supports_dict = True,
                                                on_serialize = None,
                                                on_deserialize = None),
                         AttributeConfiguration(name = 'password',
                                                supports_csv = (True, False),
                                                supports_json = (True, False),
                                                supports_yaml = (True, False),
                                                supports_dict = (True, False),
                                                on_serialize = None,
                                                on_deserialize = None)]
```

(continues on next page)

(continued from previous page)

```
    # ADDITIONAL RELATIONSHIPS, HYBRID PROPERTIES, OR ASSOCIATION PROXIES
    # GO HERE
```

```
from sqlathanor import declarative_base, Column, AttributeConfiguration

from sqlalchemy import create_engine, Table, Integer, String

BaseModel = declarative_base()

engine = create_engine('... ENGINE CONFIGURATION GOES HERE ...')
# NOTE: Because reflection relies on a specific SQLAlchemy Engine existing, presumably
# you would know how to configure / instantiate your database engine using SQLAlchemy.
# This is just here for the sake of completeness.

UserTable = Table('users',
                  BaseModel.metadata,
                  Column('id',
                         Integer,
                         primary_key = True,
                         supports_csv = True,
                         csv_sequence = 1,
                         supports_json = True,
                         supports_yaml = True,
                         supports_dict = True,
                         on_serialize = None,
                         on_deserialize = None),
                  Column('password',
                         String(255),
                         supports_csv = (True, False),
                         csv_sequence = 2,
                         supports_json = (True, False),
                         supports_yaml = (True, False),
                         supports_dict = (True, False),
                         on_serialize = None,
                         on_deserialize = None))

class ReflectedUser(BaseModel):
    __table__ = Table('users',
                      BaseModel.metadata,
                      autoload = True,
                      autoload_with = engine)

    # ADDITIONAL RELATIONSHIPS, HYBRID PROPERTIES, OR ASSOCIATION PROXIES
    # GO HERE
```

Tip: In practice, this pattern eliminates the time-saving benefits of using reflection in the first place. Instead, I would recommend adopting the *meta configuration* pattern with reflection instead.

```
from sqlathanor import declarative_base, Column, AttributeConfiguration

from sqlalchemy import create_engine, Table, Integer, String

BaseModel = declarative_base()
```

(continues on next page)

(continued from previous page)

```
engine = create_engine('... ENGINE CONFIGURATION GOES HERE ...')
# NOTE: Because reflection relies on a specific SQLAlchemy Engine existing, presumably
# you would know how to configure / instantiate your database engine using SQLAlchemy.
# This is just here for the sake of completeness.

class ReflectedUser(BaseModel):
    __table__ = Table('users',
                      BaseModel.metadata,
                      autoload = True,
                      autoload_with = engine)

    id = Column('id',
                Integer,
                primary_key = True,
                supports_csv = True,
                csv_sequence = 1,
                supports_json = True,
                supports_yaml = True,
                supports_dict = True,
                on_serialize = None,
                on_deserialize = None)

    password = Column('password',
                      String(255),
                      supports_csv = (True, False),
                      csv_sequence = 2,
                      supports_json = (True, False),
                      supports_yaml = (True, False),
                      supports_dict = (True, False),
                      on_serialize = None,
                      on_deserialize = None)

    # ADDITIONAL RELATIONSHIPS, HYBRID PROPERTIES, OR ASSOCIATION PROXIES
    # GO HERE
```

1.11 Using SQLAthanor with Automap

New in version 0.2.0.

See also:

- *Using Automap with SQLAthanor*
- **SQLAlchemy**: Automap Extension

Error: If you try to use *automap_base()* with SQLAlchemy **v.0.9.0**, you will get a *SQLAlchemySupportError*.

Declarative Approach

Meta Approach

```
from sqlathanor.automap import automap_base
from sqlalchemy import create_engine

# Create your Automap Base
Base = automap_base()

engine = create_engine('... DATABASE CONNECTION GOES HERE ...')

# Prepare your automap base. This reads your database and creates your models.
Base.prepare(engine, reflect = True)

# And here you can create a "User" model class and an "Address" model class.
User = Base.classes.users
Address = Base.classes.addresses


User.set_attribute_serialization_config('email_address',
                                        supports_csv = True,
                                        supports_json = True,
                                        supports_yaml = True,
                                        supports_dict = True)
User.set_attribute_serialization_config('password',
                                        supports_csv = (True, False),
                                        supports_json = (True, False),
                                        supports_yaml = (True, False),
                                        supports_dict = (True, False),
                                        on_deserialize = my_encryption_function)
```

```
from sqlathanor.automap import automap_base
from sqlalchemy import create_engine

# Create your Automap Base
Base = automap_base()

engine = create_engine('... DATABASE CONNECTION GOES HERE ...')

# Prepare your automap base. This reads your database and creates your models.
Base.prepare(engine, reflect = True)

# And here you can create a "User" model class and an "Address" model class.
User = Base.classes.users
Address = Base.classes.addresses

User.__serialization__ = [
    {
        'name': 'email_address',
        'supports_csv': True,
        'supports_json': True,
        'supports_yaml': True,
        'supports_dict': True
    },
    {
        'name': 'password',
        'supports_csv': (True, False),
        'supports_json': (True, False),
        'supports_yaml': (True, False),
        'supports_dict': (True, False),
```

(continues on next page)

(continued from previous page)

```
        'on_deserialize': my_encryption_function
    }
]
```

1.12 Using SQLAthanor with Flask-SQLAlchemy

See also:

- *Import SQLAthanor* > Using Flask-SQLAlchemy
- Flask-SQLAlchemy Documentation

```
from sqlathanor import FlaskBaseModel, initialize_flask_sqlathanor
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'

db = SQLAlchemy(app, model_class = FlaskBaseModel)
db = initialize_flask_sqlathanor(db)
```

1.13 Generating SQLAlchemy Tables Programmatically

New in version 0.3.0: CSV, JSON, YAML, and `dict` support

New in version 0.8.0: Pydantic model support

See also:

- *Table.from_csv()*
- *Table.from_json()*
- *Table.from_yaml()*
- *Table.from_dict()*
- *Table.from_pydantic()*

CSV

JSON

YAML

dict

Pydantic

```
from sqlathanor import Table

# Assumes CSV data is in "csv_data" and a MetaData object is in "metadata"
csv_table = Table.from_csv(csv_data,
                           'tablename_goes_here',
                           metadata,
                           'primary_key_column',
                           column_kwargs = None,
                           skip_nested = True,
                           default_to_str = False,
                           type_mapping = None)
```

```
from sqlathanor import Table

# Assumes JSON string is in "json_data" and a MetaData object is in "metadata"
json_table = Table.from_json(json_data,
                             'tablename_goes_here',
                             metadata,
                             'primary_key_column',
                             column_kwargs = None,
                             skip_nested = True,
                             default_to_str = False,
                             type_mapping = None)
```

```
from sqlathanor import Table

# Assumes YAML string is in "yaml_data" and a MetaData object is in "metadata"
yaml_table = Table.from_yaml(yaml_data,
                             'tablename_goes_here',
                             metadata,
                             'primary_key_column',
                             column_kwargs = None,
                             skip_nested = True,
                             default_to_str = False,
                             type_mapping = None)
```

```
from sqlathanor import Table

# Assumes dict object is in "dict_data" and a MetaData object is in "metadata"
dict_table = Table.from_dict(dict_data,
                             'tablename_goes_here',
                             metadata,
                             'primary_key_column',
                             column_kwargs = None,
                             skip_nested = True,
                             default_to_str = False,
                             type_mapping = None)
```

New in version 0.8.0.

```
from pydantic import BaseModel
from sqlathanor import Table

# Define Your Pydantic Models
class UserWriteModel(BaseModel):
    id: int
```

(continues on next page)

(continued from previous page)

```
    username: str
    email: str
    password: str

class UserReadModel(BaseModel):
    id: int
    username: str
    email: str

# Create Your Table
pydantic_table = Table.from_pydantic([UserWriteModel, UserReadModel],
                                     tablename = 'my_tablename_goes_here',
                                     primary_key = 'id')
```

See also:

- *Table*
- *Table.from_pydantic()*
- *SQLAthanor and Pydantic*

CHAPTER 2

Using SQLAthanor

- *Introduction*
  - *What is Serialization?*
  - *Why **SQLAthanor**?*
  - ***SQLAthanor** vs. Alternatives*
- *SQLAthanor Features*
- *Overview*
  - *How SQLAthanor Works*
- *1. Installing SQLAthanor*
  - *Dependencies*
- *2. Import SQLAthanor*
- *3. Define Your Models*
- *4. Configure Serialization/De-serialization*
  - *Declarative Configuration*
  - *Meta Configuration*
  - *Meta Configuration vs Declarative Configuration*
  - *Why Two Configuration Approaches?*
  - *Configuring at Runtime*
- *5. Configuring Pre-processing and Post-processing*
  - *Serialization Pre-processing*
  - *De-serialization Post-processing*

- *6. Serializing a Model Instance*
  - *Nesting Complex Data*
  - *to_csv()*
  - *to_json()*
  - *to_yaml()*
  - *to_dict()*
  - *dump_to_csv()*
  - *dump_to_json()*
  - *dump_to_yaml()*
  - *dump_to_dict()*
- *7. Deserializing Data*
  - *Creating New Instances*
    - *new_from_csv()*
    - *new_from_json()*
    - *new_from_yaml()*
    - *new_from_dict()*
  - *Updating Instances*
    - *update_from_csv()*
    - *update_from_json()*
    - *update_from_yaml()*
    - *update_from_dict()*
- *Using Declarative Reflection with SQLAthanor*
- *Using Automap with SQLAthanor*
- *Generating SQLAlchemy Tables...*
  - *...from Serialized Data*
  - *... from Pydantic Models*

2.1 Introduction

2.1.1 What is Serialization?

"Serialization" is a common short-hand for two important concepts: *serialization* and *de-serialization*.

To over-simplify, **serialization** is the process of taking an object in one programming language (say, Python) that can be understood by one program (say, yours) and converting it into a format that can be understood by a different program, often written in a different programming language (say, JavaScript).

De-serialization is the exact same process - but in reverse. It takes data in another format, often from another program, and converts it into an object that your program can understand.

So why is it important? Well, because modern programming is all about communication. Which in this context means communication between different programs, APIs, microservices, etc. And these various programs and microservices may be written in vastly different programming languages, have (often different) approaches to security, etc. Which means they need to be able to talk to each other appropriately.

Which is where serialization and de-serialization come in, since they're the process that makes that communication possible.

In a very typical example, you can imagine a modern web application. The back-end might be written in Python, maybe using a framework like Flask or Django. The back-end exposes a variety of RESTful APIs that handle the business logic of your web app. But you've got an entirely separate front-end, probably written in JavaScript using React/Redux, AngularJS, or something similar.

In most web applications, at some point your back-end will need to retrieve data from a database (which an *ORM* like SQLAlchemy is great at), and will want to hand it off to your front-end. A typical example might be if you want to list users, or in a social media-style app, list a user's friends. So once your Python back-end has gotten a list of users, how does it communicate that list to your JavaScript front-end? Most likely by exchanging *JSON* objects.

Which means that your Python back-end needs to take the list of users it retrieved, convert their data into JSON format, and transmit it to the front-end. That's **serialization**.

But now let's say a user in your front-end changes their email address. The front-end will need to let the back-end know, and your back-end will need to update the relevant database record with the latest change. So how does the front-end communicate that change to the back-end? Again, by sending a JSON object to the back-end. But your back-end needs to parse that data, validate it, and then reflect the change in the underlying database. The process of parsing that data? That's **de-serialization**.

2.1.2 Why SQLAthanor?

So if serialziation and de-serialization are so important, how does this relate to **SQLAthanor**? Well, serialization and de-serialization can be complicated:

- Different programs may need to serialize and de-serialize into and from multiple formats.
- Some data (like passwords) should only be **de**-serialized, but for security reasons should **never** be serialized.
- Serialization and de-serialization may need various pre-processing steps to validate the data or coerce it to/from different data types... and that validation/coercion logic may be different depending on the data format.
- The (fantastic) SQLAlchemy ORM handles database read/write operations amazingly, but does not include any serialization/de-serialization support.

This leads to a labor-intensive process of writing custom serialization/de-serialization logic for multiple (different) models and repeating that process across multiple applications. Better, we think, to package that functionality into a library.

Which is what **SQLAthanor** is.

It is designed to extend the functionality of the SQLAlchemy ORM with support for serialization and de-serialization into/from:

- *JSON*
- *CSV*
- *YAML*
- Python `dict`

Which should hopefully save some effort when building applications that need to talk to the outside world (and really, don't all apps do these days?).

2.1.3 SQLAthanor vs. Alternatives

Since *serialization* and *de-serialization* are common problems, there are a variety of alternative ways to serialize and de-serialize your SQLAlchemy models. Obviously, I'm biased in favor of **SQLAthanor**. But it might be helpful to compare **SQLAthanor** to some commonly-used alternatives:

Rolling Your Own

Pydantic

Marshmallow

Colander

pandas

Adding your own custom serialization/de-serialization logic to your SQLAlchemy declarative models is a very viable strategy. It's what I did for years, until I got tired of repeating the same patterns over and over again, and decided to build **SQLAthanor** instead.

But of course, implementing custom serialization/de-serialization logic takes a bit of effort.

Tip: When to use it?

In practice, I find that rolling my own solution is great when it's a simple model with very limited business logic. It's a "quick-and-dirty" solution, where I'm trading rapid implementation (yay!) for less flexibility/functionality (boo!).

Considering how easy **SQLAthanor** is to configure / apply, however, I find that I never really roll my own serialization/de-serialization approach when working SQLAlchemy models any more.

Tip: Because Pydantic is growing in popularity, we have decided to integrate Pydantic support within **SQLAthanor**.

Using `generate_model_from_pydantic()`, you can now programmatically generate a **SQLAthanor** `BaseModel` from your *Pydantic models*.

This allows you to only maintain *one* representation of your data model (the Pydantic one), while still being able to use SQLAthanor's rich serialization / de-serialization configuration functionality.

Pydantic is an amazing object parsing library that leverages native Python typing to provide simple syntax and rich functionality. While I have philosophical quibbles about some of its API semantics and architectural choices, I cannot deny that it is elegant, extremely performant, and all around excellent.

Since FastAPI, one of the fastest-growing web application frameworks in the Python ecosystem is tightly coupled with Pydantic, it has gained significant ground within the community.

However, when compared to **SQLAthanor** it has a number of architectural limitations:

While Pydantic has excellent serialization and deserialization functionality to JSON, it is extremely limited with its serialization/deserialization support for other common data formats like CSV or YAML.

Second, by its design Pydantic forces you to maintain **multiple** representations of your data model. On the one hand, you will need your SQLAlchemy ORM representation, but then you will *also* need one or more Pydantic models that will be used to serialize/de-serialize your model instances.

Third, by its design, Pydantic tends to lead to significant amounts of duplicate code, maintaining similar-but-not-quite-identical versions of your data models (with one Pydantic schema for each context in which you might serialize/de-serialize your data model).

Fourth, its API semantics can get extremely complicated when trying to use it as a true serialization/de-serialization library.

While Pydantic has made efforts to integrate ORM support into its API, that functionality is relatively limited in its current form.

Tip: When to use it?

Pydantic is easy to use when building simple web applications due to its reliance on native Python typing. The need to maintain multiple representations of your data models is a trivial burden with small applications or relatively simple data models.

Pydantic is also practically required when building applications using the excellent FastAPI framework.

So given these two things, we recommend using Pydantic *in combination* with **SQLAthanor** to get the best of both words: native Python typing for validation against your Python model (via Pydantic) with rich configurable serialization/de-serialization logic (via **SQLAthanor**), all integrated into the underlying SQLAlchemy ORM.

The Marshmallow library and its Marshmallow-SQLAlchemy extension are both fantastic. However, they have one major architectural difference to **SQLAthanor** and several more minor differences:

The biggest difference is that by design, they force you to maintain *two* representations of your data model. One is your SQLAlchemy *model class*, while the other is your Marshmallow schema (which determines how your model is serialized/de-serialized). Marshmallow-SQLAlchemy specifically tries to simplify this by generating a schema based on your *model class*, but you still need to configure, manage, and maintain both representations - which as your project gets more complex, becomes non-trivial.

SQLAthanor by contrast lets you configure serialization/deserialization **in** your SQLAlchemy *model class* definition. You're only maintaining *one* data model representation in your Python code, which is a massive time/effort/risk-reduction.

Other notable differences relate to the API/syntax used to support non-*JSON* formats. I think Marshmallow uses a non-obvious approach, while with **SQLAthanor** the APIs are clean and simple. Of course, on this point, YMMV.

Tip: When to use it?

Marshmallow has one advantage over **SQLAthanor**: It can serialize/de-serialize *any* Python object, whether it is a SQLAlchemy model class or not. **SQLAthanor** only works with SQLAlchemy.

As a result, it may be worth using Marshmallow instead of **SQLAthanor** if you expect to be serializing / de-serializing a lot of non-SQLAlchemy objects.

The Colander library and the ColanderAlchemy extension are both great, but they have a similar *major* architectural difference to **SQLAthanor** as Marshmallow/Marshmallow-SQLAlchemy:

By design, they force you to maintain *two* representations of your data model. One is your SQLAlchemy *model class*, while the other is your Colander schema (which determines how your model is serialized/de-serialized). Colander-Alchemy tries to simplify this by generating a schema based on your *model class*, but you still need to configure, manage, and maintain both representations - which as your project gets more complex, becomes non-trivial.

SQLAthanor by contrast lets you configure serialization/deserialization **in** your SQLAlchemy *model class* definition. You're only maintaining *one* data model representation in your Python code, which is a massive time/effort/risk-reduction.

A second major difference is that, again by design, Colander is designed to serialize/de-serialize Python objects to a set of Python primitives. Since neither *JSON*, *CSV*, or *YAML* are Python primitives, you'll still need to serialize/de-serialize Colander's input/output to/from its final "transmissable" form. Once you've got a Python primitive, this isn't difficult - but it is an extra step.

Tip: When to use it?

Colander has one advantage over **SQLAthanor**: It can serialize/de-serialize *any* Python object, whether it is a SQLAlchemy model class or not. **SQLAthanor** only works with SQLAlchemy.

As a result, it may be worth using Colander instead of **SQLAthanor** if you expect to be serializing / de-serializing a lot of non-SQLAlchemy objects.

---

pandas is one of my favorite analytical libraries. It has a number of great methods that adopt a simple syntax, like `read_csv()` or `to_csv()` which de-serialize / serialize data to various formats (including SQL, JSON, CSV, etc.).

So at first blush, one might think: Why not just use pandas to handle serialization/de-serialization?

Well, pandas isn't really a serialization alternative to **SQLAthanor**. More properly, it is an ORM alternative to SQLAlchemy itself.

I could write (and have written) a lot on the subject, but the key difference is that pandas is a "lightweight" ORM that focuses on providing a Pythonic interface to work with the output of single SQL queries. It does not support complex relationships between tables, or support the abstracted definition of business logic that applies to an object representation of a "concept" stored in your database.

SQLAlchemy is *specifically* designed to do those things.

So you can think of pandas as being a less-abstract, "closer to bare metal" ORM - which is what you want if you want very efficient computations, on relatively "flat" (non-nested/minimally relational) data. Modification or manipulation of the data can be done by mutating your pandas `DataFrame` without *too much* maintenance burden because those mutations/modifications probably don't rely too much on complex abstract business logic.

SQLAthanor piggybacks on SQLAlchemy's business logic-focused ORM capabilities. It is designed to allow you to configure expected behavior *once* and then re-use that capability across all instances (records) of your data. And it's designed to play well with all of the other complex abstractions that SQLAlchemy supports, like *relationships*, *hybrid properties*, *reflection*, or *association proxies*.

pandas serialization/de-serialization capabilities can only be configured "at use-time" (in the method call), which leads to a higher maintenance burden. **SQLAthanor**'s serialization/de-serialization capabilities are specifically designed to be configurable when defining your data model.

Tip: When to use it?

The decision of whether to use pandas or SQLAlchemy is a complex one, but in my experience a good rule of thumb is to ask yourself whether you're going to need to apply complex business logic to your data.

The more complex the business logic is, the more likely SQLAlchemy will be a better solution. And *if* you are using SQLAlchemy, then **SQLAthanor** provides great and easy-to-use serialization/de-serialization capabilities.

2.2 SQLAthanor Features

- Configure *serialization* and *de-serialization* support when defining your SQLAlchemy *models*.

- Automatically include serialization methods in your SQLAlchemy *model instances*.
- Automatically include de-serialization "creator" methods in your SQLAlchemy *models*.
- Automatically include de-serialization "updater" methods to your SQLAlchemy *model instances*.
- Support serialization and de-serialization across the most-common data exchange formats: *JSON*, *CSV*, *YAML*, and Python `dict`.
- Support pre-processing before serialization/de-serialization for data validation or coercion.
- Support serialization and de-serialization for complex *models* that may include: *relationships*, *hybrid properties*, *association proxies*, or standard Python `@property`.
- Simultaneously support different configurations for different serialization or de-serialization requirements.
- Maintain all of the existing APIs, methods, functions, and functionality of SQLAlchemy Core.
- Maintain all of the existing APIs, methods, functions, and functionality of SQLAlchemy ORM.
- Maintain all of the existing APIs, methods, functions, and functionality of SQLAlchemy Declarative ORM.
- Drive the definition and configuration of your data model from your *Pydantic models*.

2.3 Overview

SQLAthanor is designed to extend the fantastic SQLAlchemy library, to provide it with seamless *serialization* and *de-serialization* support. What do we mean by seamless? Well, in an ideal world we want serialization and de-serialization to work like this:

```
# To create serialized output from a model instance, just use:
as_json = model_instance.to_json()
as_csv = model_instance.to_csv()
as_yaml = model_instance.to_yaml()
as_dict = model_instance.to_dict()

# To create a new model instance from serialized data, just use:
new_as_instance = ModelClass.new_from_json(as_json)
new_as_instance = ModelClass.new_from_csv(as_csv)
new_as_instance = ModelClass.new_from_yaml(as_yaml)
new_as_instance = ModelClass.new_from_dict(as_dict)

# To update an existing model instance from serialized data, just use:
model_instance.update_from_json(as_json)
model_instance.update_from_csv(as_csv)
model_instance.update_from_yaml(as_yaml)
model_instance.update_from_dict(as_dict)
```

Even knowing nothing about SQLAlchemy or **SQLAthanor**, it should be pretty easy to figure out what's happening in that code, right?

Well, that's exactly what **SQLAthanor** does for you. So how? Let's break that down.

2.3.1 How SQLAthanor Works

SQLAthanor is a *drop-in replacement* for SQLAlchemy.

What does this mean? It means that it's designed to seamlessly replace *some* of your SQLAlchemy `import` statements. Then, you can continue to define your *models* just as you would using the SQLAlchemy ORM - but now they'll support *serialization* and *de-serialization*.

In other words, the process of using **SQLAthanor** is very simple:

1. Install **SQLAthanor**. (see *here*)
2. Import the components used to define your *model classes*. (see *here*)
3. Define your *model classes*, just as you would in SQLAlchemy. (see *here*)
4. Configure which *model attributes* to be serialized (output) and de-serialized (input). (see *here*)
5. Configure any pre/post-processing for serialization and de-serialization, respectively. (see *here*)
6. Serialize your *model instances* as needed. (see *here*)
7. Create new *model instances* using de-serialization. (see *here*)
8. Update existing *model instances* using de-serialization. (see *here*)

And that's it! Once you've done the steps above, you can easily serialize data from your models and de-serialize data into your models using simple methods.

Tip: Because **SQLAthanor** inherits from and extends SQLAlchemy, your existing SQLAlchemy *models* will work with no change.

By default, *serialization* and *de-serialization* are **disabled** for any *model attribute* unless they are *explicitly* enabled.

2.4 1. Installing SQLAthanor

To install **SQLAthanor**, just execute:

```
$ pip install sqlathanor
```

2.4.1 Dependencies

Python 3.x

Python 2.x

- SQLAlchemy v.0.9 or higher
- PyYAML v3.10 or higher
- simplejson v3.0 or higher
- Validator-Collection v1.4.0 or higher

- SQLAlchemy v.0.9 or higher
- PyYAML v3.10 or higher
- simplejson v3.0 or higher
- Validator-Collection v1.4.0 or higher

2.5 2. Import SQLAthanor

Since **SQLAthanor** is a *drop-in replacement*, you should import it using the same elements as you would import from SQLAlchemy:

Using SQLAlchemy

Using SQLAthanor

Using Flask-SQLAlchemy

The code below is a pretty standard set of `import` statements when working with SQLAlchemy and its Declarative ORM.

They're provided for reference below, but do **not** make use of **SQLAthanor** and do **not** provide any support for *serialization* or *de-serialization*:

```
from sqlalchemy.ext.declarative import declarative_base, as_declarative
from sqlalchemy import Column, Integer, String          # ... and any other data types

# The following are optional, depending on how your data model is designed:
from sqlalchemy.orm import relationship
from sqlalchemy.ext.hybrid import hybrid_property
from sqlalchemy.ext.associationproxy import association_proxy
```

To import **SQLAthanor**, just replace the relevant SQLAlchemy imports with their **SQLAthanor** counterparts as below:

```
from sqlathanor import declarative_base, as_declarative
from sqlathanor import Column
from sqlathanor import relationship             # This import is optional, depending␣
↪on
                                                # how your data model is designed.

from sqlalchemy import Integer, String          # ... and any other data types

# The following are optional, depending on how your data model is designed:
from sqlalchemy.ext.hybrid import hybrid_property
from sqlalchemy.ext.associationproxy import association_proxy
```

Tip: Because of its many moving parts, SQLAlchemy splits its various pieces into multiple modules and forces you to use many `import` statements.

The example above maintains this strategy to show how **SQLAthanor** is a 1:1 drop-in replacement. But obviously, you can import all of the items you need in just one `import` statement:

```
from sqlathanor import declarative_base, as_declarative, Column, relationship
```

SQLAthanor is designed to work with Flask-SQLAlchemy too! However, you need to:

1. Import the *`FlaskBaseModel`* class, and then supply it as the `model_class` argument when initializing Flask-SQLAlchemy.
2. Initialize **SQLAthanor** on your `db` instance using *`initialize_flask_sqlathanor`*.

```
from sqlathanor import FlaskBaseModel, initialize_flask_sqlathanor
from flask import Flask
```

(continues on next page)

(continued from previous page)

```
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'

db = SQLAlchemy(app, model_class = FlaskBaseModel)
db = initialize_flask_sqlathanor(db)
```

And that's it! Now **SQLAthanor** serialization functionality will be supported by:

- Flask-SQLAlchemy's `db.Model`
- Flask-SQLAlchemy's `db.relationship()`
- Flask-SQLAlchemy's `db.Column`

See also:

For more information about working with Flask-SQLAlchemy, please review their detailed documentation.

As the examples provided above show, importing **SQLAthanor** is very straightforward, and you can include it in an existing codebase quickly and easily. In fact, your code should work **just as before**. Only now it will include new functionality to support serialization and de-serialization.

The table below shows how SQLAlchemy classes and functions map to their **SQLAthanor** replacements:

SQLAlchemy Component	**SQLAthanor** Analog
`declarative_base()` `from sqlalchemy.ext.declarative import declarative_base`	*`declarative_base()`* `from sqlathanor import declarative_base`
`@as_declarative` `from sqlalchemy.ext.declarative import as_declarative`	*`@as_declarative`* `from sqlathanor import as_declarative`
`Column` `from sqlalchemy import Column`	*`Column`* `from sqlathanor import Column`
`relationship()` `from sqlalchemy import relationship`	*`relationship()`* `from sqlathanor import relationship`
`ext.automap.automap_base()` `from sqlalchemy.ext.automap import automap_base`	*`automap.automap_base()`* `from sqlathanor.automap import automap_base`

2.6 3. Define Your Models

Normally

from Pydantic

via Reflection

via Automap

from Serialized Data

Because **SQLAthanor** is a drop-in replacement for SQLAlchemy and its Declarative ORM, you can define your models the *exact same way* you would do so normally:

See also:

- SQLAlchemy Declarative ORM
- SQLAlchemy ORM Tutorial
- Flask-SQLAlchemy: Declaring Models

New in version 0.8.0.

If your application is using Pydantic as a parsing/validation library, then you can programmatically generate a pre-configured SQLAlchemy Declarative *model class* using **SQLAthanor** with the syntax `generate_model_from_pydantic()`.

This function can accept one or more *Pydantic models*, and will consolidate them into a single **SQLAthanor** *model class*, with each underlying *Pydantic model* corresponding to a *configuration set* for easy serialization / deserialization:

```
from sqlathanor import generate_model_from_pydantic

# Assuming that "PydanticBaseModel" is a
PydanticDBModel = generate_model_from_pydantic([PydanticReadModel,␣
↪PydanticWriteModel],
                                               tablename = 'my_table_name',
                                               primary_key = 'id')
```

See also:

- *generate_model_from_pydantic()*
- *SQLAthanor and Pydantic Support*

SQLAlchemy supports the use of reflection with the SQLAlchemy Declarative ORM.

This is a process where SQLAlchemy automatically constructs a Declarative *model class* based on what it reads from the table definition stored in your SQL database *or* a corresponding `Table` instance already defined and registered with a `MetaData` object.

SQLAthanor also supports the same pattern. For details, please see: *Using Declarative Reflection with SQLAthanor*

See also:

- *Using Declarative Reflection with SQLAthanor*
- **SQLAlchemy**: Reflecting Database Objects
- **SQLAlchemy**: Using Reflection with Declarative

New in version 0.2.0.

The Automap Extension is an incredibly useful tool for modeling *existing* databases with minimal effort. What it does is it reads your existing database's metadata and automatically constructs SQLAlchemy Declarative ORM *model classes* populated with your tables' columns.

Neat, right? Saves a ton of effort.

Using **SQLAthanor** you can ensure that your automapped (automatically generated) models support serialization and de-serialization. For more details, please see: *Using Automap with SQLAthanor*.

See also:

- *Using Automap with SQLAthanor*
- **SQLAlchemy**: Automap Extension

New in version 0.3.0.

If you have serialized data in either *CSV*, *JSON*, *YAML*, or a Python `dict`, you can programmatically generate a SQLAlchemy Declarative *model class* using **SQLAthanor** with the syntax `generate_model_from_<format>()` where `<format>` corresponds to `csv`, `json`, `yaml`, or `dict`:

```
##### FROM CSV:
from sqlathanor import generate_model_from_csv

# Assuming that "csv_data" contains your CSV data
CSVModel = generate_model_from_csv(csv_data,
                                   tablename = 'my_table_name',
                                   primary_key = 'id')

##### FROM JSON:
from sqlathanor import generate_model_from_json

# Assuming that "json_string" contains your JSON data in a string
JSONModel = generate_model_from_json(json_string,
                                     tablename = 'my_table_name',
                                     primary_key = 'id')

##### FROM YAML:
from sqlathanor import generate_model_from_yaml

# Assuming that "yaml_string" contains your YAML data in a string
YAMLModel = generate_model_from_yaml(yaml_string,
                                     tablename = 'my_table_name',
                                     primary_key = 'id')

##### FROM DICT:
from sqlathanor import generate_model_from_dict

# Assuming that "yaml_string" contains your YAML data in a string
DictModel = generate_model_from_dict(dict_string,
                                     tablename = 'my_table_name',
                                     primary_key = 'id')
```

See also:

- *generate_model_from_csv()*
- *generate_model_from_json()*
- *generate_model_from_yaml()*
- *generate_model_from_dict()*

2.7 4. Configure Serialization/De-serialization

> explicit is better than implicit
>
> —**PEP 20** - The Zen of Python

By default (for security reasons) *serialization* and *de-serialization* are *disabled* on all of your *model attributes*. However, **SQLAthanor** exists to let you *explicitly* enable *serialization* and/or *de-serialization* for specific *model attributes*.

Here are some important facts to understand for context:

1. You can enable *serialization* separately from *de-serialization*. This allows you to do things like de-serialize a `password` field (expect it as an input), but never include it in the output that you serialize.
2. You can configure *serialization*/*de-serialization* differently for different formats. **SQLAthanor** supports *CSV*, *JSON*, *YAML*, and Python `dict`.
3. You can serialize or de-serialize any *model attribute* that is bound to your *model class*. This includes:
    - Attributes that correspond to columns in the underlying SQL table.
    - Attributes that represent a *relationship* to another *model*.
    - Attributes that are defined as *hybrid properties*.
    - Attributes that are defined as *association proxies*.
    - *Instance attributes* defined using Python's built-in `@property` decorator.

In general, **SQLAthanor** supports two different mechanisms to configure serialization/de-serialization: *Declarative Configuration* and *Meta Configuration*.

2.7.1 Declarative Configuration

The *Declarative Configuration* approach is modeled after the SQLAlchemy Declarative ORM itself. It allows you to configure a *model attribute's serialization* and *de-serialization* *when defining* the *model attribute*.

Here's a super-simplified example of how it works:

```
from sqlathanor import declarative_base, Column, relationship

from sqlalchemy import Integer, String

BaseModel = declarative_base()

class User(BaseModel):
  __tablename__ = 'users'

  id = Column('id',
              Integer,
              primary_key = True,
              supports_csv = True,
              csv_sequence = 1,
              supports_json = True,
              supports_yaml = True,
              supports_dict = True,
              on_serialize = None,
              on_deserialize = None,
              display_name = None)
```

This example defines a *model class* called `User` which corresponds to a SQL database table named `users`. The `User` class defines one *model attribute* named `id` (which corresponds to a database *`Column`* named `id`). The database column is an integer, and it operates as the primary key for the database table.

So far, this is all *exactly* like you would normally see in the SQLAlchemy Declarative ORM.

But, there are some additional arguments supplied to *Column*: supports_csv, csv_sequence, supports_json, supports_yaml, supports_dict, on_serialize, and on_deserialize. As you can probably guess, these arguments are what configure *serialization* and *de-serialization* in **SQLAthanor** when using *declarative configuration*.

Here's what these arguments do:

SQLAthanor Configuration Arguments

Parameters

- **supports_csv** (bool or tuple of form (inbound: bool, outbound: bool)) – Determines whether the column can be *serialized* to or *de-serialized* from *CSV* format.

  If True, can be serialized to CSV and de-serialized from CSV. If False, will not be included when serialized to CSV and will be ignored if present in a de-serialized CSV.

  Can also accept a 2-member tuple (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to False, which means the column will not be serialized to CSV or de-serialized from CSV.

- **csv_sequence** (int or None) – Indicates the numbered position that the column should be in in a valid CSV-version of the object. Defaults to None.

  Note: If not specified, the column will go after any columns that *do* have a csv_sequence assigned, sorted alphabetically.

  If two columns have the same csv_sequence, they will be sorted alphabetically.

- **supports_json** (bool or tuple of form (inbound: bool, outbound: bool)) – Determines whether the column can be *serialized* to or *de-serialized* from *JSON* format.

  If True, can be serialized to JSON and de-serialized from JSON. If False, will not be included when serialized to JSON and will be ignored if present in a de-serialized JSON.

  Can also accept a 2-member tuple (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to False, which means the column will not be serialized to JSON or de-serialized from JSON.

- **supports_yaml** (bool or tuple of form (inbound: bool, outbound: bool)) – Determines whether the column can be *serialized* to or *de-serialized* from *YAML* format.

  If True, can be serialized to YAML and de-serialized from YAML. If False, will not be included when serialized to YAML and will be ignored if present in a de-serialized YAML.

  Can also accept a 2-member tuple (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to False, which means the column will not be serialized to YAML or de-serialized from YAML.

- **supports_dict** (bool or tuple of form (inbound: bool, outbound: bool)) – Determines whether the column can be *serialized* to or *de-serialized* from a Python dict.

  If True, can be serialized to dict and de-serialized from a dict. If False, will not be included when serialized to dict and will be ignored if present in a de-serialized dict.

Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

Defaults to `False`, which means the column will not be serialized to a `dict` or de-serialized from a `dict`.

- **`on_deserialize`** (callable or `dict` with formats as keys and values as callables) – A function that will be called when attempting to assign a de-serialized value to the column. This is intended to either coerce the value being assigned to a form that is acceptable by the column, or raise an exception if it cannot be coerced. If `None`, the data type's default `on_deserialize` function will be called instead.

  Tip: If you need to execute different `on_deserialize` functions for different formats, you can also supply a `dict`:

  ```
  on_deserialize = {
    'csv': csv_on_deserialize_callable,
    'json': json_on_deserialize_callable,
    'yaml': yaml_on_deserialize_callable,
    'dict': dict_on_deserialize_callable
  }
  ```

  Defaults to `None`.

- **`on_serialize`** (callable or `dict` with formats as keys and values as callables) – A function that will be called when attempting to serialize a value from the column. If `None`, the data type's default `on_serialize` function will be called instead.

  Tip: If you need to execute different `on_serialize` functions for different formats, you can also supply a `dict`:

  ```
  on_serialize = {
    'csv': csv_on_serialize_callable,
    'json': json_on_serialize_callable,
    'yaml': yaml_on_serialize_callable,
    'dict': dict_on_serialize_callable
  }
  ```

  Defaults to `None`.

- **`display_name`** (`str` / `None`) – If not `None`, a `str` that will replace your model attribute's name in your serialized object (or where your model attribute's name will be sought when de-serializing). Defaults to `None`

When using Declarative Configuration, the exact same arguments can be applied when defining a *relationship* using *`relationship()`* as shown in the expanded example below. Let's look at a somewhat more complicated example:

```
from sqlathanor import declarative_base, Column, relationship

from sqlalchemy import Integer, String

BaseModel = declarative_base()

class User(BaseModel):
  __tablename__ = 'users'
```

(continues on next page)

(continued from previous page)

```
    id = Column('id',
                Integer,
                primary_key = True,
                supports_csv = True,
                csv_sequence = 1,
                supports_json = True,
                supports_yaml = True,
                supports_dict = True,
                on_serialize = None,
                on_deserialize = None,
                display_name = None)

    addresses = relationship('Address',
                             backref = 'user',
                             supports_json = True,
                             supports_yaml = (True, True),
                             supports_dict = (True, False),
                             on_serialize = None,
                             on_deserialize = None,
                             display_name = None)
```

This example is (obviously) very similar to the previous one. But now we have added a *relationship* defined using the **SQLAthanor** *relationship()* function. This operates exactly as the built-in `sqlalchemy.relationship()` function. But it has the same set of declarative **SQLAthanor** configuration attributes.

So in this example, we define a relationship to a different *model class* called `Address`, and assign that relationship to the *model attribute* `User.addresses`. Given the configuration above, the `User` model will:

- support serializing the `id` attribute *to* CSV, JSON, YAML, and `dict`
- support de-serializing the `id` attribute *from* CSV, JSON, YAML, and `dict`
- support serializing related `addresses` *to* JSON and YAML, but will *not* include the `addresses` attribute when serializing to CSV or `dict`
- support de-serializing related `addresses` *from* JSON, YAML, and `dict`, but *not* from CSV.

2.7.2 Meta Configuration

The *Meta Configuration* approach is more robust than the *Declarative* approach. That's because it supports:

1. More *model attribute* types, including *hybrid properties*, *association proxies*, and Python `@property` *instance attributes*
2. The definition of multiple named configuration sets for different serialization and de-serialization rules.

The Meta Configuration approach relies on a special *model attribute* that you define for your *model class*: `__serialization__`. This attribute contains one or more lists of *AttributeConfiguration* objects, which are used to indicate the explicit *serialization*/*de-serialization* configuration for your *model attributes*.

Here's how you would configure an example `User` model using the Meta Configuration approach:

Standard Configuration

Multiple Configuration Sets

```
from sqlathanor import declarative_base, Column, relationship, AttributeConfiguration

from sqlalchemy import Integer, String

BaseModel = declarative_base()

class User(BaseModel):
  __tablename__ = 'users'

  __serialization__ = [AttributeConfiguration(name = 'id',
                                              supports_csv = True,
                                              csv_sequence = 1,
                                              supports_json = True,
                                              supports_yaml = True,
                                              supports_dict = True,
                                              on_serialize = None,
                                              on_deserialize = None),
                       AttributeConfiguration(name = 'addresses',
                                              supports_json = True,
                                              supports_yaml = (True, True),
                                              supports_dict = (True, False),
                                              on_serialize = None,
                                              on_deserialize = None)]

  id = Column('id',
              Integer,
              primary_key = True)

  addresses = relationship('Address',
                           backref = 'user')
```

The `__serialization__` attribute contains an explicit configuration for both the `id` and `addresses` column. Each *`AttributeConfiguration`* object supports the same configuration arguments as are used by the *declarative* approach, with one addition: It needs a `name` argument that explicitly indicates the name of the *model attribute* that is being configured.

```
from sqlathanor import declarative_base, Column, relationship, AttributeConfiguration

from sqlalchemy import Integer, String

BaseModel = declarative_base()

class User(BaseModel):
  __tablename__ = 'users'

  __serialization__ = {
      'with-addresses': [AttributeConfiguration(name = 'id',
                                                supports_csv = True,
                                                csv_sequence = 1,
                                                supports_json = True,
                                                supports_yaml = True,
                                                supports_dict = True,
                                                on_serialize = None,
                                                on_deserialize = None,
                                                display_name = None),
                         AttributeConfiguration(name = 'addresses',
                                                supports_json = True,
```

(continues on next page)

(continued from previous page)

```
                                                supports_yaml = (True, True),
                                                supports_dict = (True, False),
                                                on_serialize = None,
                                                on_deserialize = None,
                                                display_name = None)],
      'without-addresses': [AttributeConfiguration(name = 'id',
                                                   supports_csv = True,
                                                   csv_sequence = 1,
                                                   supports_json = True,
                                                   supports_yaml = True,
                                                   supports_dict = True,
                                                   on_serialize = None,
                                                   on_deserialize = None,
                                                   display_name = None)],
      'different-display-name': [AttributeConfiguration(name = 'id',
                                                        supports_csv = True,
                                                        csv_sequence = 1,
                                                        supports_json = True,
                                                        supports_yaml = True,
                                                        supports_dict = True,
                                                        on_serialize = None,
                                                        on_deserialize = None,
                                                        display_name = None),
                                 AttributeConfiguration(name = 'addresses',
                                                        supports_json = True,
                                                        supports_yaml = (True, True),
                                                        supports_dict = (True, False),
                                                        on_serialize = None,
                                                        on_deserialize = None,
                                                        display_name = 'custom_
↪address_label')]

  }

  id = Column('id',
              Integer,
              primary_key = True)

  addresses = relationship('Address',
                           backref = 'user')
```

If you will be using different configuration sets, then the `__serialization__` attribute contains a `dict` where each key is the unique name of a given configuration set and its value is a list of *`AttributeConfiguration`* objects.

In the example provided above, we have created three different named configuration sets:

- `'with-addresses'` which serializes both the `id` and `addresses` attributes,
- `'without-addresses'` which serializes just the `id` attribute but *not* the `addresses` attribute
- `'different-display-name'` which serializes both the `id` and `addresses` attribute, but applies a different `display_name` when serializing the `addresses` attribute.

Note: The `__serialization__` attribute and its configuration sets accept both *`AttributeConfiguration`* instances, as well as `dict` representations of those instances.

If you supply `dict` configurations, **SQLAthanor** will automatically convert them to *`AttributeConfiguration`*

instances.

The `__serialization__` below is identical to the one above:

```
__serialization__ = [
    {
        'name': 'id',
        'supports_csv': True,
        'csv_sequence': 1,
        'supports_json': True,
        'supports_yaml': True,
        'supports_dict': True,
        'on_serialize': None,
        'on_deserialize': None
    },
    {
        'name': 'addresses',
        'supports_json': True,
        'supports_yaml': (True, True),
        'supports_dict': (True, False),
        'on_serialize': None,
        'on_deserialize': None
    }
]
```

See also:

- *AttributeConfiguration*
- *SQLAthanor Configuration Arguments*
- *Declarative Configuration*

Unlike the *declarative* approach, you can use the `__serialization__` attribute to configure serialization and de-serialization for more complex types of *model attributes*, including *hybrid properties*, *association proxies*, and Python `@property` attributes.

Using the meta configuration approach, you configure these more complex attributes in exactly the same way:

```
from sqlathanor import declarative_base, Column, relationship, AttributeConfiguration

from sqlalchemy import Integer, String
from sqlalchemy.ext.hybrid import hybrid_property
from sqlalchemy.ext.associationproxy import association_proxy

BaseModel = declarative_base()

class User(BaseModel):
  __tablename__ = 'users'

  __serialization__ = [AttributeConfiguration(name = 'id',
                                              supports_csv = True,
                                              csv_sequence = 1,
                                              supports_json = True,
                                              supports_yaml = True,
                                              supports_dict = True,
                                              on_serialize = None,
                                              on_deserialize = None),
                       AttributeConfiguration(name = 'addresses',
```

(continues on next page)

(continued from previous page)

```
                                        supports_json = True,
                                        supports_yaml = (True, True),
                                        supports_dict = (True, False),
                                        on_serialize = None,
                                        on_deserialize = None),
                        AttributeConfiguration(name = 'hybrid',
                                        supports_csv = True,
                                        csv_sequence = 2,
                                        supports_json = True,
                                        supports_yaml = True,
                                        supports_dict = True,
                                        on_serialize = None,
                                        on_deserialize = None)]
                        AttributeConfiguration(name = 'keywords',
                                        supports_csv = False,
                                        supports_json = True,
                                        supports_yaml = True,
                                        supports_dict = True,
                                        on_serialize = None,
                                        on_deserialize = None)]
                        AttributeConfiguration(name = 'python_property',
                                        supports_csv = (False, True),
                                        csv_sequence = 3,
                                        supports_json = (False, True),
                                        supports_yaml = (False, True),
                                        supports_dict = (False, True),
                                        on_serialize = None,
                                        on_deserialize = None)]

    id = Column('id',
                Integer,
                primary_key = True)

    addresses = relationship('Address',
                             backref = 'user')

    _hybrid = 1

    @hybrid_property
    def hybrid(self):
        return self._hybrid

    @hybrid.setter
    def hybrid(self, value):
        self._hybrid = value

    @hybrid.expression
    def hybrid(cls):
      return False

    keywords = association_proxy('keywords', 'keyword')

    @property
    def python_property(self):
      return self._hybrid * 2
```

This more complicated pattern extends the earlier example with a *hybrid property* nammed `hybrid`, an *association*

proxy named `keywords`, and an *instance attribute* (defined using `@property`) named `python_property`.

The `__serialization__` configuration shown ensures that:

- `hybrid` can be serialized to and de-serialized from CSV, JSON, YAML, and `dict`.
- `keywords` can be serialized to and de-serialized from JSON, YAML, and `dict`, but *cannot* be serialized to or de-serialized from CSV.
- `python_property` can be serialized to all formats, but *cannot* be de-serialized (which makes sense, since it is defined without a `setter`)

> **Warning:** A configuration found in `__serialization__` always takes precedence.
>
> This means that if you mix the *declarative* and *meta* approaches, the configuration in `__serialization__` will be applied.

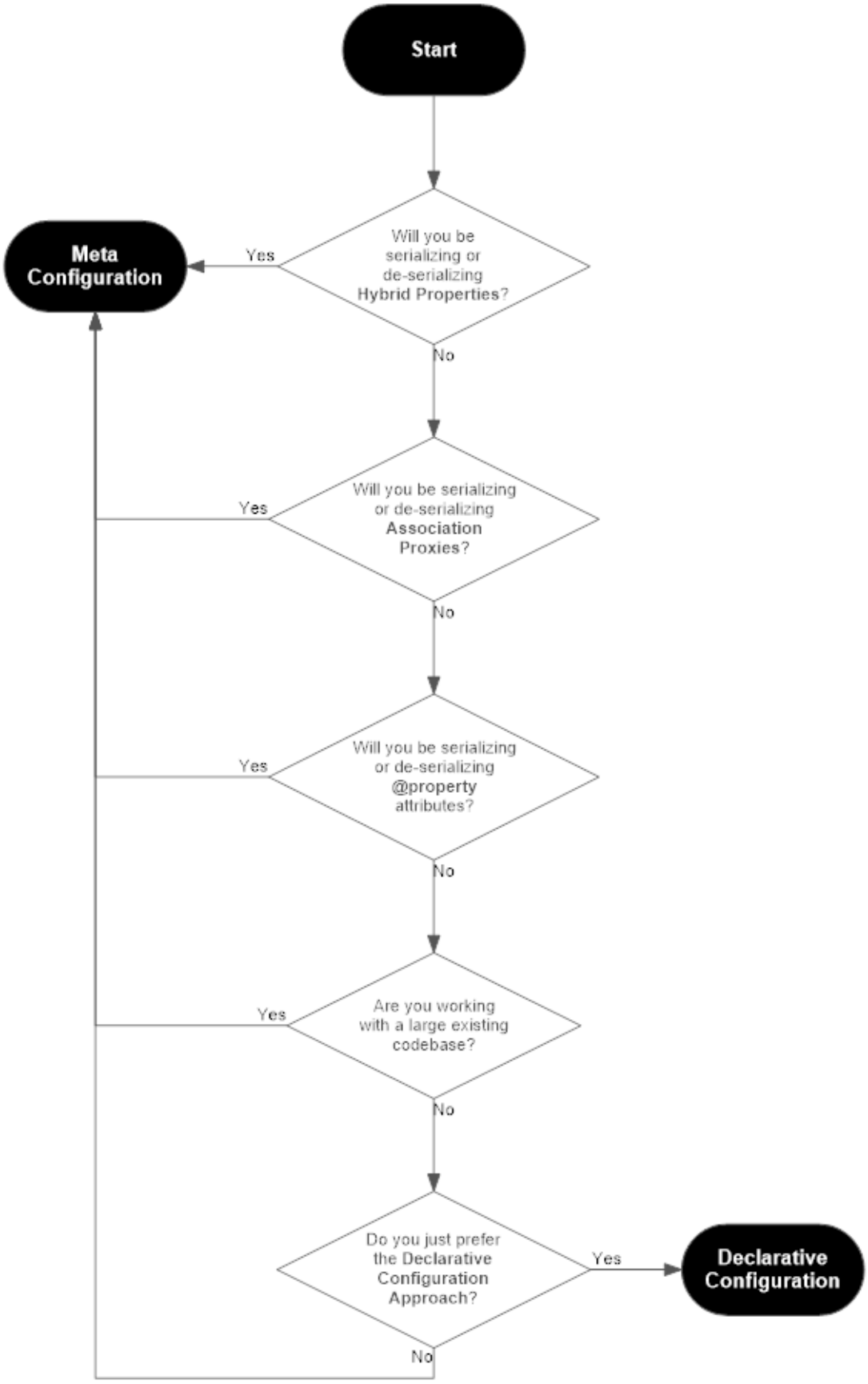
Tip: For security reasons, if you don't explicitly configure serialization/de-serialization for a *model attribute* using the *meta* or *declarative* approach, by default that attribute will not be serialized and will be ignored when de-serializing.

2.7.3 Meta Configuration vs Declarative Configuration

How should you choose between the *meta configuration* and *declarative configuration* approach?

Well, to some extent it's a question of personal preference. For example, I *always* use the *meta* approach because I believe it is cleaner, easier to maintain, and more extensible. But as you can probably tell if you look at my code, I tend to be a bit pedantic about such things. There are plenty of times when the *declarative* approach will make sense and "feel" right.

Here's a handy flowchart to help you figure out which you should use:


Start
Will you be serializing or de-serializing Hybrid Properties?
Yes
Meta Configuration
No
Will you be serializing or de-serializing Association Proxies?
Yes
No
Will you be serializing or de-serializing @property attributes?
Yes
No
Are you working with a large existing codebase?
Yes
No
Do you just prefer the Declarative Configuration Approach?
Yes
Declarative Configuration
No

2.7.4 Why Two Configuration Approaches?

The **Zen of Python** holds that:

> There should be one– and preferably only one –obvious way to do it.

And that is a very wise principle. But there are times when it makes sense to diverge from that principle. I made a conscious choice to support two different configuration mechanisms for several practical reasons:

1. SQLAlchemy has been around for a long time, and is very popular. There are many existing codebases that might benefit from integrating with **SQLAthanor**. The *meta configuration* approach lets users make minimal changes to their existing codebases at minimal risk.
2. SQLAlchemy is often used in quick-and-dirty projects where the additional overhead of defining an explicit *meta configuration* in the `__serialization__` class attribute will interrupt a programmer's "flow". The SQLAlchemy Declarative ORM already provides a great API for defining a model with minimal overhead, so piggybacking on that familiar API will enhance the programmer experience.
3. The internal mechanics of how SQLAlchemy implements *hybrid properties* and *association proxies* are very complicated, and can be mutated at various points in a *model class* or *model instance* lifecycle. As a result, supporting them in the *declarative configuration* approach would have been very complicated, with a lot of exposure to potential edge case errors. But the *meta configuration* approach neatly avoids those risks.
4. The only way for the *declarative configuration* approach to support serialization and de-serialization of *model attributes* defined using Python's built-in `@property` decorator would require extending a feature of the standard library... which I consider an anti-pattern that should be done rarely (if ever).

So given those arguments, you might ask: Why not just use the *meta configuration* approach, and call it a day? It clearly has major advantages. And yes, you'd be right to ask the question. It *does* have major advantages, and it is the one I use almost-exclusively in my own code.

But!

I've spent close to twenty years in the world of data science and analytics, and I know what the coding practices in that community look like and how SQLAlchemy often gets used "in the wild". And that experience and knowledge tells me that the *declarative* approach will just "feel more natural" to a large number of data scientists and developers who have a particular workflow, particular coding style, specific coding conventions, and who often don't need SQLAlchemy's more complicated features like *hybrid properties* or *association proxies*.

And since **SQLAthanor** can provide benefits to both "application developers" and "data scientists", I've tried to design an interface that will feel "natural" to both communities.

2.7.5 Configuring at Runtime

See also:

SQLAthanor exposes a number of public methods that allow you to modify the *serialization*/*de-serialization* configuration at run-time. For more information, please see:

- *`BaseModel.configure_serialization()`*
- *`BaseModel.set_attribute_serialization_config()`*

2.8 5. Configuring Pre-processing and Post-processing

When *serializing* and *de-serializing* objects, it is often necessary to either convert data to a more-compatible format, or to validate inbound data to ensure it meets your expectations.

SQLAthanor supports this using serialization pre-processing and de-serialization post-processing, as configured in the `on_serialize` and `on_deserialize` *configuration arguments*.

2.8.1 Serialization Pre-processing

The `on_deserialize` *configuration argument* allows you to assign a *serialization function* to a particular *model attribute*.

If assigned, the serialization function will be called *before* the *model attribute*'s value gets serialized into its target format. This is particularly useful when you need to convert a value from its native (in Python) type/format into a type/format that is supported by the serialized data type.

A typical example of this might be converting a `None` value in Python to an empty string `''` that can be included in a *CSV* record, or ensuring a decimal value is appropriately rounded.

The `on_serialize` *argument* expects to receive either a callable (a Python function), or a `dict` where keys correspond to **SQLAthanor**'s supported formats and values are the callables to use for that format. Thus:

```
...
on_serialize = my_serialization_function,
...
```

will call the `my_serialization_function()` whenever serializing the value, but

```
...
on_serialize = {
    'csv': my_csv_serialization_function,
    'json': my_json_serialization_function,
    'yaml': my_yaml_serialization_function,
    'dict': my_dict_serialization_function
},
...
```

will call a *different* function when serializing the value to each of the four formats given above.

Tip: If `on_serialize` is `None`, or if its value for a particular format is `None`, then **SQLAthanor** will default to a *Default Serialization Function* based on the data type of the *model attribute*.

If defining your own custom serializer function, please bear in mind that a valid serializer function will:

- accept *one* positional argument, which is the value of the *model attribute* to be serialized, and
- return *one* value, which is the value that will be included in the serialized output.

See also:

- *Default Serialization Functions*
- *SQLAthanor Configuration Arguments*
- *`sqlathanor.declarative.BaseModel`*

2.8.2 De-serialization Post-processing

The `on_deserialize` *configuration argument* allows you to assign a *de-serialization function* to a particular *model attribute*.

If assigned, the de-serialization function will be called *after* your serialized object is parsed, but *before* the value is assigned to your Python *model attribute*. This is particularly useful when you need to:

- convert a value from its serialized format (e.g. a string) into the type supported by your *model class* (e.g. an integer),
- validate that a serialized value is "correct" (matches your expectations),
- do something to the value before persisting it (e.g. hash and salt) to the database.

A typical example of this might be:

- converting an empty string `''` in a *CSV* record into `None`
- validating that the serialized value is a proper telephone number
- hashing an inbound password.

The `on_deserialize` *argument* expects to receive either a callable (a Python function), or a `dict` where keys correspond to **SQLAthanor**'s supported formats and values are the callables to use for that format. Thus:

```
...
on_deserialize = my_deserialization_function,
...
```

will call the `my_deserialization_function()` whenever de-serializing the value, but

```
...
on_deserialize = {
    'csv': my_csv_deserialization_function,
    'json': my_json_deserialization_function,
    'yaml': my_yaml_deserialization_function,
    'dict': my_dict_deserialization_function
},
...
```

will call a *different* function when de-serializing the value to each of the four formats given above.

Tip: If `on_deserialize` is `None`, or if its value for a particular format is `None`, then **SQLAthanor** will default to a *Default De-serialization Function* based on the data type of the *model attribute* being de-serialized.

If defining your own custom deserializer function, please bear in mind that a valid deserializer function will:

- accept *one* positional argument, which is the value for the *model attribute* as found in the serialized input, and
- return *one* value, which is the value that will be assigned to the *model attribute* (and thus probably persisted to the underlying database).

See also:

- *Default De-serialization Functions*
- *SQLAthanor Configuration Arguments*
- *sqlathanor.declarative.BaseModel*

2.9 6. Serializing a Model Instance

Once you've *configured* your *model class*, you can now easily serialize it. Your *model instance* will have two serialization methods for each of the formats, named `to_<format>` and `dump_to_<format>` respectively, where `<format>` corresponds to `csv`, `json`, `yaml`, and `dict`:

- *to_csv()*
- *to_json()*
- *to_yaml()*
- *to_dict()*
- *dump_to_csv()*
- *dump_to_json()*
- *dump_to_yaml()*
- *dump_to_dict()*

The `to_<format>` methods adhere to whatever *configuration* you have set up for your *model class*, while the `dump_to_<format>` method will automatically serialize all *model attributes* defined.

> **Caution:** Use the the `dump_to_<format>` methods with caution!
>
> Because they automatically serialize all *model attributes* defined for your *model class*, they potentially expose your application to security vulnerabilities.

2.9.1 Nesting Complex Data

SQLAthanor automatically supports nesting complex structures in JSON, YAML, and `dict`. However, to prevent the risk of infinite recursion, those formats serialization methods all feature a required `max_nesting` argument. By default, it is set to `0` which prevents *model attributes* that resolve to another *model class* from being included in a serialized output.

Unlike other supported formats, *CSV* works best with "flat" structures where each output column contains one and only one simple value, and so `to_csv()` does not include any `max_nesting` or `current_nesting` arguments.

Tip: As a general rule of thumb, we recommend that you avoid enabling CSV serialization on *relationships* unless using a custom *serialization function* to structure nested data.

2.9.2 to_csv()

`BaseModel.`**`to_csv`**(*include_header=False*, *delimiter='|'*, *wrap_all_strings=False*, *null_text='None'*, *wrapper_character="'"*, *double_wrapper_character_when_nested=False*, *escape_character='\\'*, *line_terminator='\r\n'*, *config_set=None*)

Retrieve a CSV string with the object's data.

Parameters

- **`include_header`** (`bool`) – If `True`, will include a header row with column labels. If `False`, will not include a header row. Defaults to `True`.
- **`delimiter`** (`str`) – The delimiter used between columns. Defaults to `|`.

- **wrap_all_strings** (bool) – If True, wraps any string data in the wrapper_character. If None, only wraps string data if it contains the delimiter. Defaults to False.
- **null_text** (str) – The text value to use in place of empty values. Only applies if wrap_empty_values is True. Defaults to 'None'.
- **wrapper_character** (str) – The string used to wrap string values when wrapping is necessary. Defaults to '.
- **double_wrapper_character_when_nested** (bool) – If True, will double the wrapper_character when it is found inside a column value. If False, will precede the wrapper_character by the escape_character when it is found inside a column value. Defaults to False.
- **escape_character** (str) – The character to use when escaping nested wrapper characters. Defaults to \.
- **line_terminator** (str) – The character used to mark the end of a line. Defaults to \r\n.
- **config_set** (str / None) – If not None, the named configuration set to use. Defaults to None.

Returns Data from the object in CSV format ending in a newline (\n).

Return type str

2.9.3 to_json()

BaseModel.**to_json**(*max_nesting=0*, *current_nesting=0*, *serialize_function=None*, *config_set=None*, ***kwargs*)

Return a JSON representation of the object.

Parameters

- **max_nesting** (int) – The maximum number of levels that the resulting JSON object can be nested. If set to 0, will not nest other serializable objects. Defaults to 0.
- **current_nesting** (int) – The current nesting level at which the dict representation will reside. Defaults to 0.
- **serialize_function** (callable / None) – Optionally override the default JSON serializer. Defaults to None, which applies the default simplejson JSON serializer.

  Note: Use the serialize_function parameter to override the default JSON serializer.

  A valid serialize_function is expected to accept a single dict and return a str, similar to simplejson.dumps().

  If you wish to pass additional arguments to your serialize_function pass them as keyword arguments (in kwargs).

- **config_set** (str / None) – If not None, the named configuration set to use. Defaults to None.
- **kwargs** (*keyword arguments*) – Optional keyword parameters that are passed to the JSON serializer function. By default, these are options which are passed to simplejson.dumps().

Returns A `str` with the JSON representation of the object.

Return type `str`

Raises

- ***SerializableAttributeError*** – if attributes is empty
- ***MaximumNestingExceededError*** – if `current_nesting` is greater than `max_nesting`
- ***MaximumNestingExceededWarning*** – if an attribute requires nesting beyond `max_nesting`

2.9.4 to_yaml()

`BaseModel.`**`to_yaml`**(*max_nesting=0*, *current_nesting=0*, *serialize_function=None*, *config_set=None*, ***kwargs*)

Return a YAML representation of the object.

Parameters

- **`max_nesting`** (`int`) – The maximum number of levels that the resulting object can be nested. If set to `0`, will not nest other serializable objects. Defaults to `0`.
- **`current_nesting`** (`int`) – The current nesting level at which the representation will reside. Defaults to `0`.
- **`serialize_function`** (callable / `None`) – Optionally override the default YAML serializer. Defaults to `None`, which calls the default `yaml.dump()` function from the PyYAML library.

  Note: Use the `serialize_function` parameter to override the default YAML serializer.

  A valid `serialize_function` is expected to accept a single `dict` and return a `str`, similar to `yaml.dump()`.

  If you wish to pass additional arguments to your `serialize_function` pass them as keyword arguments (in `kwargs`).

- **`config_set`** (`str` / `None`) – If not `None`, the named configuration set to use. Defaults to `None`.
- **`kwargs`** (*`keyword arguments`*) – Optional keyword parameters that are passed to the YAML serializer function. By default, these are options which are passed to `yaml.dump()`.

Returns A `str` with the JSON representation of the object.

Return type `str`

Raises

- ***SerializableAttributeError*** – if attributes is empty
- ***MaximumNestingExceededError*** – if `current_nesting` is greater than `max_nesting`
- ***MaximumNestingExceededWarning*** – if an attribute requires nesting beyond `max_nesting`

2.9.5 to_dict()

BaseModel.**to_dict**(*max_nesting=0*, *current_nesting=0*, *config_set=None*)

Return a OrderedDict representation of the object.

Parameters

- **max_nesting** (int) – The maximum number of levels that the resulting dict object can be nested. If set to 0, will not nest other serializable objects. Defaults to 0.
- **current_nesting** (int) – The current nesting level at which the dict representation will reside. Defaults to 0.
- **config_set** (str / None) – If not None, the named configuration set to use when processing the input. Defaults to None.

Returns A OrderedDict representation of the object.

Return type OrderedDict

Raises

- ***SerializableAttributeError*** – if attributes is empty
- ***MaximumNestingExceededError*** – if current_nesting is greater than max_nesting
- ***MaximumNestingExceededWarning*** – if an attribute requires nesting beyond max_nesting

2.9.6 dump_to_csv()

BaseModel.**dump_to_csv**(*include_header=False*, *delimiter='|'*, *wrap_all_strings=False*, *null_text='None'*, *wrapper_character="'"*, *double_wrapper_character_when_nested=False*, *escape_character='\\'*, *line_terminator='\r\n'*, *config_set=None*)

Retrieve a *CSV* representation of the object, *with all attributes* serialized regardless of configuration.

> **Caution:** Nested objects (such as *relationships* or *association proxies*) will **not** be serialized.

Note: This method ignores any display_name contributed on the AttributeConfiguration.

Parameters

- **include_header** (bool) – If True, will include a header row with column labels. If False, will not include a header row. Defaults to True.
- **delimiter** (str) – The delimiter used between columns. Defaults to |.
- **wrap_all_strings** (bool) – If True, wraps any string data in the wrapper_character. If None, only wraps string data if it contains the delimiter. Defaults to False.
- **null_text** (str) – The text value to use in place of empty values. Only applies if wrap_empty_values is True. Defaults to 'None'.

- **wrapper_character** (str) – The string used to wrap string values when wrapping is necessary. Defaults to '.
- **double_wrapper_character_when_nested** (bool) – If True, will double the wrapper_character when it is found inside a column value. If False, will precede the wrapper_character by the escape_character when it is found inside a column value. Defaults to False.
- **escape_character** (str) – The character to use when escaping nested wrapper characters. Defaults to \.
- **line_terminator** (str) – The character used to mark the end of a line. Defaults to \r\n.
- **config_set** (str / None) – If not None, the named configuration set to use. Defaults to None.

Returns Data from the object in CSV format ending in a newline (\n).

Return type str

2.9.7 dump_to_json()

BaseModel.**dump_to_json**(*max_nesting=0*, *current_nesting=0*, *serialize_function=None*, *config_set=None*, ***kwargs*)

Return a *JSON* representation of the object, *with all attributes* regardless of configuration.

> **Caution:** Nested objects (such as *relationships* or *association proxies*) will **not** be serialized.

Parameters

- **max_nesting** (int) – The maximum number of levels that the resulting JSON object can be nested. If set to 0, will not nest other serializable objects. Defaults to 0.
- **current_nesting** (int) – The current nesting level at which the dict representation will reside. Defaults to 0.
- **serialize_function** (callable / None) – Optionally override the default JSON serializer. Defaults to None, which applies the default simplejson JSON serializer.

  Note: Use the serialize_function parameter to override the default JSON serializer.

  A valid serialize_function is expected to accept a single dict and return a str, similar to simplejson.dumps().

  If you wish to pass additional arguments to your serialize_function pass them as keyword arguments (in kwargs).

- **config_set** (str / None) – If not None, the named configuration set to use. Defaults to None.
- **kwargs** (*keyword arguments*) – Optional keyword parameters that are passed to the JSON serializer function. By default, these are options which are passed to simplejson.dumps().

Returns A str with the JSON representation of the object.

Return type str

Raises

- ***SerializableAttributeError*** – if attributes is empty
- ***MaximumNestingExceededError*** – if `current_nesting` is greater than `max_nesting`
- ***MaximumNestingExceededWarning*** – if an attribute requires nesting beyond `max_nesting`

2.9.8 dump_to_yaml()

`BaseModel.`**`dump_to_yaml`**(*max_nesting=0*, *current_nesting=0*, *serialize_function=None*, *config_set=None*, ***kwargs*)

Return a *YAML* representation of the object *with all attributes*, regardless of configuration.

> **Caution:** Nested objects (such as *relationships* or *association proxies*) will **not** be serialized.

Parameters

- **`max_nesting`** (`int`) – The maximum number of levels that the resulting object can be nested. If set to `0`, will not nest other serializable objects. Defaults to `0`.
- **`current_nesting`** (`int`) – The current nesting level at which the representation will reside. Defaults to `0`.
- **`serialize_function`** (callable / `None`) – Optionally override the default YAML serializer. Defaults to `None`, which calls the default `yaml.dump()` function from the PyYAML library.

  Note: Use the `serialize_function` parameter to override the default YAML serializer.

  A valid `serialize_function` is expected to accept a single `dict` and return a `str`, similar to `yaml.dump()`.

  If you wish to pass additional arguments to your `serialize_function` pass them as keyword arguments (in `kwargs`).

- **`config_set`** (`str` / `None`) – If not `None`, the named configuration set to use. Defaults to `None`.
- **`kwargs`** (*`keyword arguments`*) – Optional keyword parameters that are passed to the YAML serializer function. By default, these are options which are passed to `yaml.dump()`.

Returns A `str` with the JSON representation of the object.

Return type `str`

Raises

- ***SerializableAttributeError*** – if attributes is empty
- ***MaximumNestingExceededError*** – if `current_nesting` is greater than `max_nesting`

- ***MaximumNestingExceededWarning*** – if an attribute requires nesting beyond `max_nesting`

2.9.9 dump_to_dict()

`BaseModel.`**`dump_to_dict`**(*max_nesting=0*, *current_nesting=0*, *config_set=None*)

Return a `OrderedDict` representation of the object, *with all attributes* regardless of configuration.

> **Caution:** Nested objects (such as *relationships* or *association proxies*) will **not** be serialized.

Parameters

- **`max_nesting`** (`int`) – The maximum number of levels that the resulting `OrderedDict` object can be nested. If set to `0`, will not nest other serializable objects. Defaults to `0`.
- **`current_nesting`** (`int`) – The current nesting level at which the `dict` representation will reside. Defaults to `0`.
- **`config_set`** (`str` / `None`) – If not `None`, the named configuration set to use when processing the input. Defaults to `None`.

Returns A `OrderedDict` representation of the object.

Return type `OrderedDict`

Raises

- ***SerializableAttributeError*** – if attributes is empty
- ***MaximumNestingExceededError*** – if `current_nesting` is greater than `max_nesting`
- ***MaximumNestingExceededWarning*** – if an attribute requires nesting beyond `max_nesting`

2.10 7. Deserializing Data

Once you've *configured* your *model class*, you can now easily *de-serialize* it from the formats you have enabled.

However, unlike *serializing* your data, there are actually two types of de-serialization method to choose from:

- The `new_from_<format>` method operates on your *model class* directly and create a new *model instance* whose properties are set based on the data you are de-serializing.
- The `update_from_<format>` methods operate on a *model instance*, and update that instance's properties based on the data you are de-serializing.

Creating New:

- *new_from_csv()*
- *new_from_json()*
- *new_from_yaml()*
- *new_from_dict()*

Updating:

- *update_from_csv()*
- *update_from_json()*
- *update_from_yaml()*
- *update_from_dict()*

2.10.1 Creating New Instances

new_from_csv()

classmethod BaseModel.**new_from_csv**(*csv_data*, *delimiter='|'*, *wrap_all_strings=False*, *null_text='None'*, *wrapper_character="'"*, *double_wrapper_character_when_nested=False*, *escape_character='\\'*, *line_terminator='\r\n'*, *config_set=None*)

Create a new model instance from a CSV record.

Tip: Unwrapped empty column values are automatically interpreted as null (None).

Parameters

- **csv_data** (str / Path-like object) – The CSV data. If a Path-like object, will read the first record from a file that is assumed to include a header row. If a str and has more than one record (line), will assume the first line is a header row.
- **delimiter** (str) – The delimiter used between columns. Defaults to |.
- **wrapper_character** (str) – The string used to wrap string values when wrapping is applied. Defaults to '.
- **null_text** (str) – The string used to indicate an empty value if empty values are wrapped. Defaults to *None*.
- **config_set** (str / None) – If not None, the named configuration set to use. Defaults to None.

Returns A *model instance* created from the record.

Return type model instance

Raises

- ***DeserializationError*** – if csv_data is not a valid str
- ***CSVStructureError*** – if the columns in csv_data do not match the expected columns returned by get_csv_column_names()
- ***ValueDeserializationError*** – if a value extracted from the CSV failed when executing its *de-serialization function*.

new_from_json()

classmethod BaseModel.**new_from_json**(*input_data*, *deserialize_function=None*, *error_on_extra_keys=True*, *drop_extra_keys=False*, *config_set=None*, ***kwargs*)

Create a new model instance from data in JSON.

Parameters

- **input_data** (str or Path-like object) – The JSON data to de-serialize.

  Note: If input_data points to a file, and the file contains a list of JSON objects, the first JSON object will be considered.

- **deserialize_function** (callable / None) – Optionally override the default JSON deserializer. Defaults to None, which calls the default simplejson.loads() function from the doc:*simplejson* <*simplejson:index*> library.

  Note: Use the deserialize_function parameter to override the default JSON deserializer.

  A valid deserialize_function is expected to accept a single str and return a dict, similar to simplejson.loads().

  If you wish to pass additional arguments to your deserialize_function pass them as keyword arguments (in kwargs).

- **error_on_extra_keys** (bool) – If True, will raise an error if an unrecognized key is found in input_data. If False, will either drop or include the extra key in the result, as configured in the drop_extra_keys parameter. Defaults to True.

  Warning: Be careful setting error_on_extra_keys to False.

  This method's last step passes the keys/values of the processed input data to your model's __init__() method.

  If your instance's __init__() method does not support your extra keys, it will likely raise a TypeError.

- **drop_extra_keys** (bool) – If True, will ignore unrecognized top-level keys in input_data. If False, will include unrecognized keys or raise an error based on the configuration of the error_on_extra_keys parameter. Defaults to False.
- **config_set** (str / None) – If not None, the named configuration set to use. Defaults to None.
- **kwargs** (*keyword arguments*) – Optional keyword parameters that are passed to the JSON deserializer function. By default, these are options which are passed to simplejson.loads().

Raises

- ***ExtraKeyError*** – if error_on_extra_keys is True and input_data contains top-level keys that are not recognized as attributes for the instance model.

- ***DeserializationError*** – if `input_data` is not a `dict` or JSON object serializable to a `dict` or if `input_data` is empty.

new_from_yaml()

classmethod BaseModel.**new_from_yaml**(*input_data*, *deserialize_function=None*, *error_on_extra_keys=True*, *drop_extra_keys=False*, *config_set=None*, ***kwargs*)

Create a new model instance from data in YAML.

Parameters

- **input_data** (`str` / Path-like object) – The YAML data to de-serialize. May be either a `str` or a Path-like object to a YAML file.
- **deserialize_function** (callable / `None`) – Optionally override the default YAML deserializer. Defaults to `None`, which calls the default `yaml.safe_load()` function from the PyYAML library.

  Note: Use the `deserialize_function` parameter to override the default YAML de-serializer.

  A valid `deserialize_function` is expected to accept a single `str` and return a `dict`, similar to `yaml.safe_load()`.

  If you wish to pass additional arguments to your `deserialize_function` pass them as keyword arguments (in `kwargs`).

- **error_on_extra_keys** (`bool`) – If `True`, will raise an error if an unrecognized key is found in `input_data`. If `False`, will either drop or include the extra key in the result, as configured in the `drop_extra_keys` parameter. Defaults to `True`.

  Warning: Be careful setting `error_on_extra_keys` to `False`.

  This method's last step passes the keys/values of the processed input data to your model's `__init__()` method.

  If your instance's `__init__()` method does not support your extra keys, it will likely raise a `TypeError`.

- **drop_extra_keys** (`bool`) – If `True`, will ignore unrecognized top-level keys in `input_data`. If `False`, will include unrecognized keys or raise an error based on the configuration of the `error_on_extra_keys` parameter. Defaults to `False`.
- **config_set** (`str` / `None`) – If not `None`, the named configuration set to use. Defaults to `None`.

Raises

- ***ExtraKeyError*** – if `error_on_extra_keys` is `True` and `input_data` contains top-level keys that are not recognized as attributes for the instance model.
- ***DeserializationError*** – if `input_data` is not a `dict` or JSON object serializable to a `dict` or if `input_data` is empty.

new_from_dict()

classmethod BaseModel.**new_from_dict**(*input_data*, *error_on_extra_keys=True*, *drop_extra_keys=False*, *config_set=None*)

Update the model instance from data in a `dict` object.

Parameters

- **input_data** (`dict`) – The input `dict`
- **error_on_extra_keys** (`bool`) – If `True`, will raise an error if an unrecognized key is found in `input_data`. If `False`, will either drop or include the extra key in the result, as configured in the `drop_extra_keys` parameter. Defaults to `True`.

  Warning: Be careful setting `error_on_extra_keys` to `False`.

  This method's last step passes the keys/values of the processed input data to your model's `__init__()` method.

  If your instance's `__init__()` method does not support your extra keys, it will likely raise a `TypeError`.

- **drop_extra_keys** (`bool`) – If `True`, will omit unrecognized top-level keys from the resulting `dict`. If `False`, will include unrecognized keys or raise an error based on the configuration of the `error_on_extra_keys` parameter. Defaults to `False`.
- **config_set** (`str` / `None`) – If not `None`, the named configuration set to use when processing the input. Defaults to `None`.

Raises

- ***ExtraKeyError*** – if `error_on_extra_keys` is `True` and `input_data` contains top-level keys that are not recognized as attributes for the instance model.
- ***DeserializationError*** – if `input_data` is not a `dict` or JSON object serializable to a `dict` or if `input_data` is empty.

2.10.2 Updating Instances

update_from_csv()

BaseModel.**update_from_csv**(*csv_data*, *delimiter='|'*, *wrap_all_strings=False*, *null_text='None'*, *wrapper_character="'"*, *double_wrapper_character_when_nested=False*, *escape_character='\\'*, *line_terminator='\r\n'*, *config_set=None*)

Update the model instance from a CSV record.

Tip: Unwrapped empty column values are automatically interpreted as null (`None`).

Parameters

- **csv_data** (`str` / Path-like object) – The CSV data. If a Path-like object, will read the first record from a file that is assumed to include a header row. If a `str` and has more than one record (line), will assume the first line is a header row.
- **delimiter** (`str`) – The delimiter used between columns. Defaults to `|`.

- **wrapper_character** (str) – The string used to wrap string values when wrapping is applied. Defaults to '.
- **null_text** (str) – The string used to indicate an empty value if empty values are wrapped. Defaults to *None*.
- **config_set** (str / None) – If not None, the named configuration set to use. Defaults to None.

Raises

- ***DeserializationError*** – if csv_data is not a valid str
- ***CSVStructureError*** – if the columns in csv_data do not match the expected columns returned by get_csv_column_names()
- ***ValueDeserializationError*** – if a value extracted from the CSV failed when executing its *de-serialization function*.

update_from_json()

BaseModel.**update_from_json**(*input_data*, *deserialize_function=None*, *error_on_extra_keys=True*, *drop_extra_keys=False*, *config_set=None*, ***kwargs*)

Update the model instance from data in a JSON string.

Parameters

- **input_data** (str or Path-like object) – The JSON data to de-serialize.

  Note: If input_data points to a file, and the file contains a list of JSON objects, the first JSON object will be considered.

- **deserialize_function** (callable / None) – Optionally override the default JSON deserializer. Defaults to None, which calls the default simplejson.loads() function from the simplejson library.

  Note: Use the deserialize_function parameter to override the default JSON deserializer.

  A valid deserialize_function is expected to accept a single str and return a dict, similar to simplejson.loads().

  If you wish to pass additional arguments to your deserialize_function pass them as keyword arguments (in kwargs).

- **error_on_extra_keys** (bool) – If True, will raise an error if an unrecognized key is found in input_data. If False, will either drop or include the extra key in the result, as configured in the drop_extra_keys parameter. Defaults to True.

  Warning: Be careful setting error_on_extra_keys to False.

  This method's last step attempts to set an attribute on the model instance for every top-level key in the parsed/processed input data.

  If there is an extra key that cannot be set as an attribute on your model instance, it *will* raise AttributeError.

- **drop_extra_keys** (bool) – If True, will ignore unrecognized keys in the input data. If False, will include unrecognized keys or raise an error based on the configuration of the error_on_extra_keys parameter. Defaults to False.
- **config_set** (str / None) – If not None, the named configuration set to use. Defaults to None.
- **kwargs** (*keyword arguments*) – Optional keyword parameters that are passed to the JSON deserializer function.By default, these are options which are passed to simplejson.loads().

Raises

- ***ExtraKeyError*** – if error_on_extra_keys is True and input_data contains top-level keys that are not recognized as attributes for the instance model.
- ***DeserializationError*** – if input_data is not a str JSON de-serializable object to a dict or if input_data is empty.

update_from_yaml()

BaseModel.**update_from_yaml**(*input_data*, *deserialize_function=None*, *error_on_extra_keys=True*, *drop_extra_keys=False*, *config_set=None*, ***kwargs*)

Update the model instance from data in a YAML string.

Parameters

- **input_data** (str / Path-like object) – The YAML data to de-serialize. May be either a str or a Path-like object to a YAML file.
- **deserialize_function** (callable / None) – Optionally override the default YAML deserializer. Defaults to None, which calls the default yaml.safe_load() function from the PyYAML library.

Note: Use the deserialize_function parameter to override the default YAML de-serializer.

A valid deserialize_function is expected to accept a single str and return a dict, similar to yaml.safe_load().

If you wish to pass additional arguments to your deserialize_function pass them as keyword arguments (in kwargs).

- **error_on_extra_keys** (bool) – If True, will raise an error if an unrecognized key is found in input_data. If False, will either drop or include the extra key in the result, as configured in the drop_extra_keys parameter. Defaults to True.

> **Warning:** Be careful setting error_on_extra_keys to False.
>
> This method's last step attempts to set an attribute on the model instance for every top-level key in the parsed/processed input data.
>
> If there is an extra key that cannot be set as an attribute on your model instance, it *will* raise AttributeError.

- **drop_extra_keys** (bool) – If True, will ignore unrecognized keys in the input data. If False, will include unrecognized keys or raise an error based on the configuration of the error_on_extra_keys parameter. Defaults to False.
- **config_set** (str / None) – If not None, the named configuration set to use. Defaults to None.
- **kwargs** (*keyword arguments*) – Optional keyword parameters that are passed to the YAML deserializer function. By default, these are options which are passed to yaml.safe_load().

Raises

- ***ExtraKeyError*** – if error_on_extra_keys is True and input_data contains top-level keys that are not recognized as attributes for the instance model.
- ***DeserializationError*** – if input_data is not a str YAML de-serializable object to a dict or if input_data is empty.

update_from_dict()

BaseModel.**update_from_dict**(*input_data*, *error_on_extra_keys=True*, *drop_extra_keys=False*, *config_set=None*)

Update the model instance from data in a dict object.

Parameters

- **input_data** (dict) – The input dict
- **error_on_extra_keys** (bool) – If True, will raise an error if an unrecognized key is found in input_data. If False, will either drop or include the extra key in the result, as configured in the drop_extra_keys parameter. Defaults to True.

> **Warning:** Be careful setting error_on_extra_keys to False.
>
> This method's last step attempts to set an attribute on the model instance for every top-level key in the parsed/processed input data.
>
> If there is an extra key that cannot be set as an attribute on your model instance, it *will* raise AttributeError.

- **drop_extra_keys** (bool) – If True, will omit unrecognized top-level keys from the resulting dict. If False, will include unrecognized keys or raise an error based on the configuration of the error_on_extra_keys parameter. Defaults to False.
- **config_set** (str / None) – If not None, the named configuration set to use when processing the input. Defaults to None.

Raises

- ***ExtraKeyError*** – if error_on_extra_keys is True and input_data contains top-level keys that are not recognized as attributes for the instance model.
- ***DeserializationError*** – if input_data is not a dict or JSON object serializable to a dict or if input_data is empty.

2.11 Using Declarative Reflection with SQLAthanor

SQLAlchemy supports the use of reflection with the SQLAlchemy Declarative ORM.

This is a process where SQLAlchemy automatically constructs a Declarative *model class* based on what it reads from the table definition stored in your SQL database *or* a corresponding `Table` instance already defined and registered with a `MetaData` object.

See also:

- **SQLAlchemy**: Reflecting Database Objects
- **SQLAlchemy**: Using Reflection with Declarative
- *Using Automap with SQLAthanor*

SQLAthanor is *also* compatible with this pattern. In fact, it works just as you might expect. At a minimum:

```
from sqlathanor import declarative_base, Column, relationship, AttributeConfiguration

from sqlalchemy import create_engine, Integer, String, Table
from sqlalchemy.ext.hybrid import hybrid_property
from sqlalchemy.ext.associationproxy import association_proxy

engine = create_engine('... ENGINE CONFIGURATION GOES HERE ...')
# NOTE: Because reflection relies on a specific SQLAlchemy Engine existing, presumably
# you would know how to configure / instantiate your database engine using SQLAlchemy.
# This is just here for the sake of completeness.

BaseModel = declarative_base()

class ReflectedUser(BaseModel):
  __table__ = Table('users',
                    BaseModel.metadata,
                    autoload = True,
                    autoload_with = engine)
```

will read the structure of your `users` table, and populate your `ReflectedUser` model class with *model attributes* that correspond to the table's columns as defined in the underlying SQL table.

> **Caution:** By design, SQLAlchemy's reflection **ONLY** reflects `Column` definitions. It does **NOT** reflect *relationships* that you may otherwise model using SQLAlchemy.

Because the `ReflectedUser` class inherits from the **SQLAthanor** base model, it establishes the `__serialization__` attribute, and the *`to_csv()`*, *`to_json()`*, *`to_yaml()`*, and *`to_dict()`* methods on the `ReflectedUser` class.

When working with a reflected model class, you can configure serialization/deserialization using either the *declarative* or *meta* approach as you normally would.

> **Warning:** In the example above, if the database table named `users` already has a `Table` associated with it, `ReflectedUser` will inherit the `Column` definitions from the "original" `Table` object.
>
> If those column definitions are defined using *`sqlathanor.schema.Column`* with *declarative*, their serialization/deserialization will *also* be reflected (inherited).

However, the `ReflectedUser` model class will **NOT** inherit any serialization/deserialization configuration defined using the *meta* approach.

Just as with standard SQLAlchemy reflection, you can override your `Column` definitions in your reflecting class (`ReflectedUser`), or add additional `relationship` *model attributes*, *hybrid properties*, or *association proxies* to the reflecting class.

2.12 Using Automap with SQLAthanor

New in version 0.2.0.

Caution: Automap was introduced in SQLAlchemy **v.0.9.1**. If you are using **SQLAthanor** with SQLAlchemy v.0.9.0, then if you attempt to use `automap_base()` you will raise a `SQLAlchemySupportError`.

The Automap Extension is an incredibly useful tool for modeling *existing* databases with minimal effort. What it does is it reasd your existing database's metadata and automatically constructs SQLAlchemy Declarative ORM *model classes* populated with your tables' columns.

Neat, right? Saves a ton of effort.

Using **SQLAthanor** you can ensure that your automapped (automatically generated) models support serialization and de-serialization.

First, you need to create your automapped classes. This works just like in Automap, only you import `automap_base()` from **SQLAthanor** instead as shown below:

```
from sqlathanor.automap import automap_base
from sqlalchemy import create_engine

# Create your Automap Base
Base = automap_base()

engine = create_engine('... DATABASE CONNECTION GOES HERE ...')

# Prepare your automap base. This reads your database and creates your models.
Base.prepare(engine, reflect = True)

# And here you can create a "User" model class and an "Address" model class.
User = Base.classes.users
Address = Base.classes.addresses
```

In the example above, we create `User` and `Address` *model classes* which will be populated with the columns and *relationships* from the `users` and `addresses` tables in the database.

Both `User` and `Address` will have all of the standard **SQLAthanor** methods and functionality. **BUT!** They won't have any serialization/de-serialization configured.

Before you start working with your models, you can configure their serialization/de-serialization using either a *declarative* approach using `.set_attribute_serialization_config()` or a *meta approach* by setting the `__serialization__` attribute directly:

Declarative Approach

Meta Approach

```
User.set_attribute_serialization_config('email_address',
                                        supports_csv = True,
                                        supports_json = True,
                                        supports_yaml = True,
                                        supports_dict = True)
User.set_attribute_serialization_config('password',
                                        supports_csv = (True, False),
                                        supports_json = (True, False),
                                        supports_yaml = (True, False),
                                        supports_dict = (True, False),
                                        on_deserialize = my_encryption_function)
```

```
User.__serialization__ = [
    {
        'name': 'email_address',
        'supports_csv': True,
        'supports_json': True,
        'supports_yaml': True,
        'supports_dict': True
    },
    {
        'name': 'password',
        'supports_csv': (True, False),
        'supports_json': (True, False),
        'supports_yaml': (True, False),
        'supports_dict': (True, False),
        'on_deserialize': my_encryption_function
    }
]
```

Both snippets of code above tell the `User` model to include `users.email_address` in both serialized output, and to expect it in de-serialized input. It also tells the `User` model to *never* serialize the `users.password` column, but to expect it in inbound data to de-serialize.

See also:

- **SQLAlchemy**: Automap Extension
- *Using Declarative Reflection with SQLAthanor*

2.13 Generating SQLAlchemy Tables...

2.13.1 ...from Serialized Data

New in version 0.3.0.

If you are **not** using SQLAlchemy's Declarative ORM but would like to generate SQLAlchemy `Table` objects programmatically based on serialized data, you can do so by importing the **SQLAthanor** *`Table`* object and calling a `from_<format>()` class method:

CSV

JSON

YAML

dict

```
from sqlathanor import Table

# Assumes CSV data is in "csv_data" and a MetaData object is in "metadata"
csv_table = Table.from_csv(csv_data,
                           'tablename_goes_here',
                           metadata,
                           'primary_key_column',
                           column_kwargs = None,
                           skip_nested = True,
                           default_to_str = False,
                           type_mapping = None)
```

```
from sqlathanor import Table

# Assumes JSON string is in "json_data" and a MetaData object is in "metadata"
json_table = Table.from_json(json_data,
                             'tablename_goes_here',
                             metadata,
                             'primary_key_column',
                             column_kwargs = None,
                             skip_nested = True,
                             default_to_str = False,
                             type_mapping = None)
```

```
from sqlathanor import Table

# Assumes YAML string is in "yaml_data" and a MetaData object is in "metadata"
yaml_table = Table.from_yaml(yaml_data,
                             'tablename_goes_here',
                             metadata,
                             'primary_key_column',
                             column_kwargs = None,
                             skip_nested = True,
                             default_to_str = False,
                             type_mapping = None)
```

```
from sqlathanor import Table

# Assumes dict object is in "dict_data" and a MetaData object is in "metadata"
dict_table = Table.from_dict(dict_data,
                             'tablename_goes_here',
                             metadata,
                             'primary_key_column',
                             column_kwargs = None,
                             skip_nested = True,
                             default_to_str = False,
                             type_mapping = None)
```

See also:

- *Table*
- *Table.from_csv()*
- *Table.from_json()*

- *Table.from_yaml()*
- *Table.from_dict()*

2.13.2 ... from Pydantic Models

If you are **not** using SQLAlchemy's Declarative ORM but would like to generate SQLAlchemy `Table` objects programmatically based on *Pydantic models*, you can do so by importing the **SQLAthanor** *Table* class and calling the *from_pydantic()* class method:

```
from pydantic import BaseModel
from sqlathanor import Table

# Define Your Pydantic Models
class PydanticWriteModel(BaseModel):
    id: int
    username: str
    email: str
    password: str

class PydanticReadModel(BaseModel):
    id: int
    username: str
    email: str

# Create Your Table
pydantic_table = Table.from_pydantic([PydanticWriteModel, PydanticReadModel],
                                     tablename = 'my_tablename_goes_here',
                                     primary_key = 'id')
```

See also:

- *Table*
- *Table.from_pydantic()*
- *SQLAthanor and Pydantic*

Some Caveats

I'm a big fan of Pydantic. Its authors have made different choices, but that doesn't diminish my respect for their work, and that's one of the main reasons why I have decided to extend **SQLAthanor** with built-in support for Pydantic.

My interpretation of the priorities below is my subjective evaluation of the library and its API semantics, and from my experience using the library in a number of web application projects of varying complexity. These observations are not meant to be a criticism: They are merely my thoughts on where the libraries have taken different philosophical and stylistic paths.

CHAPTER 3

SQLAthanor and Pydantic

- *SQLAthanor, Pydantic, and FastAPI*
- *Generating and Configuring Model Classes Using Pydantic*
- *Generating Tables from Pydantic Models*
- *Configuring Attributes from Pydantic Models*

New in version 0.8.0.

3.1 SQLAthanor, Pydantic, and FastAPI

SQLAthanor and Pydantic are both concerned with the serialization and deserialization of data. However, they approach the topic from different angles and have made a variety of (very different) architectural and stylistic choices.

To be clear, neither set of choices is "better" or "worse", but they do reflect the authors' different priorities and stylistic preferences:

SQLAthanor Priorities	Pydantic Priorities
Database/ORM compatibility with SQLAlchemy	Database/ORM agnosticism
The maintenance of a single representation of your data model, tied to its database implementation	Multiple representations of your data model, each of which is tied to its usage in your code
Explicit reference and conceptual documentation	Documentation by example / in code
Explicit APIs for the data model's lifecycle	Implicit APIs relying on the Python standard library

Both libraries have their place: in general terms, if I were working on a simple web application, on a microservice, or on a relatively simple data model I would consider Pydantic as a perfectly viable "quick-and-dirty" option. Its use of Python's native typing hints/annotation is a beautifully elegant solution.

However, if I need to build a robust API with complex data model representations, tables with multiple relationships, or complicated business logic? Then I would prefer the robust and extensible capabilities afforded by the SQLAlchemy Delarative ORM and the **SQLAthanor** library.

If that were it, I would consider Pydantic to be equivalent to Marshmallow and Colander: an interesting tool for serialization/deserialization, and one that has its place, but not one that **SQLAthanor** need be concerned with.

But there's one major difference: FastAPI.

FastAPI is an amazing microframework, and is rapidly rising in popularity across the Python ecosystem. That's for very good reason: It is blisteringly fast, its API is relatively simple, and it has the ability to automatically generate OpenAPI/Swagger schemas of your API endpoints. What's not to love?

Well, its tight coupling with Pydantic, for one thing. When building an application using the FastAPI framework, I am practically forced to use *Pydantic models* as my API inputs, outputs, and validators. If I choose not to use Pydantic models, then I lose many of the valuable features (besides performance) which make FastAPI so attractive for writing API applications.

But using FastAPI and Pydantic in a complex API application may require a lot of "extra" code: the repetition of object models, the replication of business logic, the duplication of context, etc. All of these are concerns that **SQLAthanor** was explicitly designed to minimize.

So what to do? Most patterns, documentation, and best practices found on the internet for authoring FastAPI applications explicitly suggest that you (manually, in your code):

- Create a SQLAlchemy *model class* for the database interface for each data model
- Create one Pydantic *model class* for *each* "version" of your data model's output/input. So if you need one read version and a different write version? You need two *Pydantic models*.
- Use your *Pydantic models* as the validators for your API endpoints, as needed.

This is all fine and dandy, but now what happens if you need to add an attribute to your data model? You have to make a change to your SQLAlchemy model class, and to one or more Pydantic models, and possibly to your API endpoints. And let's not get started on changes to your data model's underlying business logic!

There has to be a better way.

Which is why I added Pydantic support to **SQLAthanor**. With this added support, you can effectively use your *Pydantic models* as the "canonical definition" of your data model. Think of the lifecycle this way:

- You define your data model in one or more *Pydantic models*.
- You programmatically create a SQLAlchemy *model class* whose columns are *automatically* derived from the underlying *Pydantic models* and for whom each *Pydantic Model* serves as a serialization/deserialization *configuration set*.

Thus, you remove one of the (more complicated) steps in the process of writing your FastAPI application. Now all you have to do is create your Pydantic models, and then generate your **SQLAthanor** *model classes*. Your FastAPI can still validate based on your Pydantic models, even if you choose to drive serialization/deserialization from your SQLAlchemy *model classes*.

In other words: It saves you code! And maintenance!

Just look at the example below. Not only does it save you a couple of lines, but most importantly when in the future you need to modify your data model (and let's face it, that is one of the most common modifications in real applications) you make your changes in one place rather than two:

FastAPI with Pydantic only

FastAPI with SQLAthanor/Pydantic

```
# THIS CODE SNIPPET HAS BEEN ADAPTED FROM THE OFFICIAL FASTAPI DOCUMENTATION:
# https://fastapi.tiangolo.com/tutorial/sql-databases/

# Assumes that there is a "database" module that defines your SQLAlchemy BaseModel.
from typing import List, Optional
from pydantic import BaseModel

from .database import Base

class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True)
    email = Column(String, unique=True, index=True)
    hashed_password = Column(String)
    is_active = Column(Boolean, default=True)

    items = relationship("Item", back_populates="owner")

class Item(Base):
    __tablename__ = "items"

    id = Column(Integer, primary_key=True, index=True)
    title = Column(String, index=True)
    description = Column(String, index=True)
    owner_id = Column(Integer, ForeignKey("users.id"))

    owner = relationship("User", back_populates="items")

class ItemBase(BaseModel):
    title: str
    description: Optional[str] = None

class ItemCreate(ItemBase):
    pass

```

(continues on next page)

(continued from previous page)

```
class Item(ItemBase):
    id: int
    owner_id: int

    class Config:
        orm_mode = True

class UserBase(BaseModel):
    email: str

class UserCreate(UserBase):
    password: str

class User(UserBase):
    id: int
    is_active: bool
    items: List[Item] = []

    class Config:
        orm_mode = True
```

```
from typing import List, Optional
from pydantic import BaseModel as PydanticBase

from sqlathanor import BaseModel, Column, generate_model_from_pydantic
from sqlalchemy.types import String

class ItemBase(PydanticBase):
    title: str
    description: Optional[str] = None

class ItemCreate(ItemBase):
    pass

class ItemRead(ItemBase):
    id: int
    owner_id: int

    class Config:
        orm_mode = True

class UserBase(PydanticBase):
    email: str

class UserCreate(UserBase):
    password: str

class UserRead(UserBase):
    id: int
    is_active: bool
    items: List[Item] = []

    class Config:
        orm_mode = True

User = generate_model_from_pydantic({ 'create': UserCreate
```

(continues on next page)

(continued from previous page)

```
                                        'read': UserRead },
                                      tablename = 'users',
                                      primary_key = 'id')


Item = generate_model_from_pydantic({ 'create': ItemCreate,
                                        'read': ItemRead },
                                      tablename = 'items',
                                      primary_key = 'id')

Item.owner = relationship("User", back_populates="items")
User.items = relationship("Item", back_populates="owner")
User.hashed_password = Column(String,
                              supports_csv = False,
                              supports_json = False,
                              supports_yaml = False,
                              supports_dict = False)
```

3.2 Generating and Configuring Model Classes Using Pydantic

As **SQLAthanor** relies on the creation of *model classes* which both define your database representation and provide serialization/deserialization configuration instructions, the first step to using Pydantic with **SQLAthanor** is to generate your *model classes* based on your *Pydantic models*.

You can do this in **SQLAthanor** using the *generate_model_from_pydantic()* function. This function takes your *Pydantic models* as an input, and creates a **SQLAthanor** *model class* (which is a subclass of *sqlathanor.declarative.BaseModel*).

When generating your model classes from *Pydantic models*, you can supply multiple models which will then get consolidated into a single **SQLAthanor** *BaseModel*. For example:

1 Model

2 Models (shared config set)

2 Models (independent config sets)

This example shows how you would generate a single *sqlathanor.BaseModel* from a single `pydantic.BaseModel`. Since it only has one model, it would have only one serialization/deserialization *configuration set* by default:

```
from pydantic import BaseModel as PydanticBaseModel
from sqlathanor import generate_model_from_pydantic

class SinglePydanticModel(PydanticBaseModel):
    id: int
    username: str
    email: str

SingleSQLAthanorModel = generate_model_from_pydantic(SinglePydanticModel,
                                                     tablename = 'my_tablename',
                                                     primary_key = 'id')
```

This code will generate a single **SQLAthanor** *model class* named `SingleSQLAthanorModel`, which will contain three columns: `id`, `username`, and `email`. The column types will be set to correspond to the data types annotated in the `SinglePydanticModel` class definition.

This example shows how you would combine multiple *Pydantic models* into a single *`sqlathanor.BaseModel`*. A typical use case would be if one *Pydantic model* represents the output when you are retrieving/viewing a user's data (which does not have a `password` field for security reasons) and hte other *Pydantic model* represents the input when you are writing/creating a new user (which does need the password field).

Note: Because both *Pydantic models* are passed to the function in a single `list`, they will receive a single **SQLAthanor** *configuration set*.

```
from pydantic import BaseModel as PydanticBaseModel
from sqlathanor import generate_model_from_pydantic

class ReadUserModel(PydanticBaseModel):
    id: int
    username: str
    email: str

class WriteUserModel(ReadUserModel):
    password: str

SingleSQLAthanorModel = generate_model_from_pydantic([ReadUserModel,
                                                      WriteUserModel],
                                                     tablename = 'my_tablename',
                                                     primary_key = 'id')
```

This code will generate a single **SQLAthanor** *model class* named `SingleSQLAthanorModel` with four columns (`id`, `username`, `email`, and `password`). However, because all models were passed in as a single list, the columns will be consolidated with only *one* *configuration set*.

Caution: In my experience, it is very rare that you would want to consolidate multiple *Pydantic models* with only one *configuration set*. Most of the type, each *Pydantic model* will actually represent its own *configuration set* as documented in the next example.

This example shows how you would combine multiple *Pydantic models* into a single *`sqlathanor.BaseModel`*, but configure multiple serialization/deserialization *configuration sets* based on those *Pydantic models*.

This is the most-common use case, and is fairly practical. To define multiple *configuration sets*, simply pass the *Pydantic models* as key/value pairs in the first argument:

```
from pydantic import BaseModel as PydanticBaseModel
from sqlathanor import generate_model_from_pydantic

class ReadUserModel(PydanticBaseModel):
    id: int
    username: str
    email: str

class WriteUserModel(ReadUserModel):
    password: str

SQLAthanorModel = generate_model_from_pydantic({ 'read': ReadUserModel,
```

(continues on next page)

(continued from previous page)

```
                                                    'write': WriteUserModel
                                                },
                                                tablename = 'my_tablename',
                                                primary_key = 'id')
```

This code will generate a single **SQLAthanor** *model class* (`SQLAthanorModel`, with four columns - `id`, `username`, `email`, and `password`), but that model class will have two configuration sets: `read` which will serialize/de-serialize only three columns (`id`, `username`, and `email`) and `write` which will serialize/de-serialize four columns (`id`, `username`, `email`, and `password`).

This `SQLAthanorModel` then becomes useful when serializing your *model instances* to `dict` or de-serializing them from `dict` using the context-appropriate *configuration set*:

```
# Assumes that "as_dict" contains a string JSON representation with attributes as
# defined in your "WriteUserModel" Pydantic model.
model_instance = SQLAthanorModel.new_from_json(as_json, config_set = 'write')

# Produces a dict representation of the object with three attributes, corresponding
# to your "ReadUserModel" Pydantic model.
readable_as_dict = model_instance.to_dict(config_set = 'read')
```

Tip: When generating your **SQLAthanor** *model classes* from your *Pydantic models*, it is important to remember that serialization and de-serialization is disabled by default for security reasons. Therefore a best practice is to *enable/disable your serialization and de-serialization at runtime*.

See also:

- *`BaseModel.configure_serialization()`*
- *`BaseModel.set_attribute_serialization_config()`*

Caution: This functionality *does not* support more complex table structures, including relationships, hybrid properties, or association proxies.

3.3 Generating Tables from Pydantic Models

Just as you can *generate SQLAthanor model classes from Pydantic models*, you can also create *`Table`* objects from *Pydantic models*, consolidating their attributes into standard SQL *`Column`* definitions.

```
from pydantic import BaseModel
from sqlathanor import Table

# Define Your Pydantic Models
class UserWriteModel(BaseModel):
    id: int
    username: str
    email: str
    password: str
```

(continues on next page)

(continued from previous page)

```
class UserReadModel(BaseModel):
    id: int
    username: str
    email: str

# Create Your Table
pydantic_table = Table.from_pydantic([UserWriteModel, UserReadModel],
                                     tablename = 'my_tablename_goes_here',
                                     primary_key = 'id')
```

This code will generate a single *Table* instance (`pydantic_table`) which will have four columns: `id`, `username`, `email`, and `password`. Their column types will correspond to the type hints defined in the Pydantic models.

See also:

- *Table*
- *Table.from_pydantic()*

3.4 Configuring Attributes from Pydantic Models

There may be times when you wish to configure the serialization / de-serialization of *model class* attributes based on a related *Pydantic model*. You can programmatically create a new *AttributeConfiguration* instance from a *Pydantic model* by calling the *AttributeConfiguration.from_pydantic_model()* class method:

```
from pydantic import BaseModel
from sqlathanor import Table

# Define Your Pydantic Models
class UserWriteModel(BaseModel):
    id: int
    username: str
    email: str
    password: str

class UserReadModel(BaseModel):
    id: int
    username: str
    email: str

password_config = AttributeConfiguration.from_pydantic_model(UserWriteModel,
                                                             field = 'password',
                                                             supports_csv = (True,␣
↪False),
                                                             supports_json = (True,␣
↪False),
                                                             supports_yaml = (True,␣
↪False),
                                                             supports_dict = (True,␣
↪False),
                                                             on_deserialize = my_
↪encryption_function)
```

This code will produce a single *AttributeConfiguration* instance named `password_config`. It will support the de-serialization of data, but will never be serialized (a typical pattern for password fields!). Furthermore, it will execute the `my_encryption_function` during the de-serialization process.

A very common use case is to configure the serialization/de-serialization profile for attributes that were programmatically derived from *Pydantic models*.

See also:

- *AttributeConfiguration.from_pydantic_model()*

CHAPTER 4

API Reference

- *Declarative ORM*
  - *BaseModel*
  - *declarative_base()*
  - *@as_declarative*
  - *generate_model_from_csv()*
  - *generate_model_from_json()*
  - *generate_model_from_yaml()*
  - *generate_model_from_dict()*
  - *generate_model_from_pydantic()*
- *Schema*
  - *Column*
  - *relationship()*
  - *Table*
- *Attribute Configuration*
  - *AttributeConfiguration*
  - *validate_serialization_config()*
- *Flask-SQLAlchemy / Flask-SQLAthanor*
  - *initialize_flask_sqlathanor()*
  - *FlaskBaseModel*
- *Automap*

- *SQLAthanor Internals*
  - *RelationshipProperty*

4.1 Declarative ORM

SQLAthanor provides a drop-in replacement for SQLAlchemy's `declarative_base()` function and decorators, which can either be used as a base for your SQLAlchemy models directly or can be mixed into your SQLAlchemy models.

See also:

It is *`BaseModel`* which exposes the methods and properties that enable *serialization* and *de-serialization* support.

For more information, please see:

- *Using SQLAthanor*.

4.1.1 BaseModel

class BaseModel(**args*, ***kwargs*)

Base class that establishes shared methods, attributes, and properties.

When constructing your ORM models, inherit from (or mixin) this class to add support for *serialization* and *de-serialization*.

Note: It is this class which adds **SQLAthanor**'s methods and properties to your SQLAlchemy model.

If your SQLAlchemy models do not inherit from this class, then they will not actually support *serialization* or *de-serialization*.

You can construct your declarative models using three approaches:

Using BaseModel Directly

Using `declarative_base()`

Using `@as_declarative`

By inheriting or mixing in *`BaseModel`* directly as shown in the examples below.

```
from sqlathanor import BaseModel

# EXAMPLE 1: As a direct parent class for your model.
class MyModel(BaseModel):

    # Standard SQLAlchemy declarative model definition goes here.

# EXAMPLE 2: As a mixin parent for your model.
class MyBaseModel(object):

    # An existing base model that you have developed.

class MyModel(MyBaseModel, BaseModel):
```

(continues on next page)

(continued from previous page)

```
    # Standard SQLAlchemy declarative model definition goes here.
```

By calling the *declarative_base()* function from **SQLAthanor**:

```
from sqlathanor import declarative_base

MyBaseModel = declarative_base()
```

By decorating your base model class with the *@as_declarative* decorator:

```
from sqlathanor import as_declarative

@as_declarative
class MyBaseModel(object):

    # Standard SQLAlchemy declarative model definition goes here.
```

classmethod clear_serialization_cache()
Clears any cached serialization/de-serialization configurations.

Note: Does not affect the configuration itself - merely clears the heap caches if present.

---

classmethod configure_serialization(*configs=None*, *attributes=None*, *supports_csv=False*, *supports_json=False*, *supports_yaml=False*, *supports_dict=False*, *on_serialize=None*, *on_deserialize=None*, *config_set=None*)
Apply configuration settings to the *model class*.

Caution: This method either overwrites the entire configuration (if no `config_set` is supplied) set for the *model class*, or overwrites the `config_set` indicated.

Tip: For this method, the configuration settings applied in `configs` receive priority over a configuration specified in keyword arguments.

This means that if an attribute is configured in both `configs` and `attributes`, the configuration in `configs` will apply.

Parameters

- **configs** (iterable of *AttributeConfiguration* / `pydantic.main.ModelMetaclass` / `None`) – Collection of *AttributeConfiguration* objects to apply to the class. Defaults to `None`.
- **attributes** (iterable of `str` / `None`) – Collection of *model attribute* names to which a configuration will be applied. Defaults to `None`.
- **supports_csv** (`bool` / `tuple` of form (inbound: `bool`, outbound: `bool`)) – Determines whether the column can be serialized to or de-serialized from CSV format.

If `True`, can be serialized to CSV and de-serialized from CSV. If `False`, will not be included when serialized to CSV and will be ignored if present in a de-serialized CSV.

Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

Defaults to `False`.

- **`supports_json`** (`bool` / `tuple` of form (inbound: `bool`, outbound: `bool`)) – Determines whether the column can be serialized to or de-serialized from JSON format.

  If `True`, can be serialized to JSON and de-serialized from JSON. If `False`, will not be included when serialized to JSON and will be ignored if present in a de-serialized JSON.

  Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to `False`.

- **`supports_yaml`** (`bool` / `tuple` of form (inbound: `bool`, outbound: `bool`)) – Determines whether the column can be serialized to or de-serialized from YAML format.

  If `True`, can be serialized to YAML and de-serialized from YAML. If `False`, will not be included when serialized to YAML and will be ignored if present in a de-serialized YAML.

  Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to `False`.

- **`supports_dict`** (`bool` / `tuple` of form (inbound: `bool`, outbound: `bool`)) – Determines whether the column can be serialized to or de-serialized to a Python `dict`.

  If `True`, can be serialized to `dict` and de-serialized from a `dict`. If `False`, will not be included when serialized to `dict` and will be ignored if present in a de-serialized `dict`.

  Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to `False`.

- **`on_deserialize`** (callable / `dict` with formats as keys and values as callables / `None`) – A function that will be called when attempting to assign a de-serialized value to the column. This is intended to either coerce the value being assigned to a form that is acceptable by the column, or raise an exception if it cannot be coerced.

  Tip: If you need to execute different `on_deserialize` functions for different formats, you can also supply a `dict`:

  ```
  on_deserialize = {
    'csv': csv_on_deserialize_callable,
    'json': json_on_deserialize_callable,
    'yaml': yaml_on_deserialize_callable,
    'dict': dict_on_deserialize_callable
  }
  ```

  If `False`, will clear the current configuration to apply the default.

  If `None`, will retain whatever configuration is currently applied. Defaults to `None`

Tip: To clear the `on_deserialize` function, you can either supply a value of `False` or pass a `dict` with particular formats set to `None`:

```
on_deserialize = {
  'csv': None,
  'json': None,
  'yaml': None,
  'dict': None
}

# is equivalent to:

on_deserialize = False
```

This will revert the *on_deserialize* function to the attribute's default *on_deserialize* function based on its data type.

- **on_serialize** (callable / `dict` with formats as keys and values as callables / `None` / `False`) – A function that will be called when attempting to serialize a value from the column.

Tip: If you need to execute different `on_serialize` functions for different formats, you can also supply a `dict`:

```
on_serialize = {
  'csv': csv_on_serialize_callable,
  'json': json_on_serialize_callable,
  'yaml': yaml_on_serialize_callable,
  'dict': dict_on_serialize_callable
}
```

If `False`, will clear the current configuration to apply the default configuration.

If `None`, will retain whatever configuration is currently applied. Defaults to `None`

Tip: To clear the `on_serialize` function, you need to pass `False` or a `dict` with particular formats set to `None`:

```
on_serialize = {
  'csv': None,
  'json': None,
  'yaml': None,
  'dict': None
}

# is equivalent to

on_serialize = False
```

This will revert the *on_serialize* function to the attribute's default *on_serialize* function based on its data type.

- **config_set** (`str` / `None`) – If not `None`, will apply `configs` to the *configuration*

set named. If the class does not use pre-existing configuration sets, will switch the class' meta configuration to use configuration sets, with any pre-existing configuration assigned to a set named `_original`. Defaults to `None`.

classmethod does_support_serialization(*attribute*, *from_csv=None*, *to_csv=None*, *from_json=None*, *to_json=None*, *from_yaml=None*, *to_yaml=None*, *from_dict=None*, *to_dict=None*, *config_set=None*)

Indicate whether `attribute` supports serialization/deserializtion.

Parameters

- **attribute** (`str`) – The name of the attribute whose serialization support should be confirmed.
- **from_csv** (`bool` / `None`) – If `True`, includes attribute names that **can** be de-serialized from CSV strings. If `False`, includes attribute names that **cannot** be de-serialized from CSV strings. If `None`, will not include attributes based on CSV de-serialization support (but may include them based on other parameters). Defaults to `None`.
- **to_csv** (`bool` / `None`) – If `True`, includes attribute names that **can** be serialized to CSV strings. If `False`, includes attribute names that **cannot** be serialized to CSV strings. If `None`, will not include attributes based on CSV serialization support (but may include them based on other parameters). Defaults to `None`.
- **from_json** (`bool` / `None`) – If `True`, includes attribute names that **can** be de-serialized from JSON strings. If `False`, includes attribute names that **cannot** be de-serialized from JSON strings. If `None`, will not include attributes based on JSON de-serialization support (but may include them based on other parameters). Defaults to `None`.
- **to_json** (`bool` / `None`) – If `True`, includes attribute names that **can** be serialized to JSON strings. If `False`, includes attribute names that **cannot** be serialized to JSON strings. If `None`, will not include attributes based on JSON serialization support (but may include them based on other parameters). Defaults to `None`.
- **from_yaml** (`bool` / `None`) – If `True`, includes attribute names that **can** be de-serialized from YAML strings. If `False`, includes attribute names that **cannot** be de-serialized from YAML strings. If `None`, will not include attributes based on YAML de-serialization support (but may include them based on other parameters). Defaults to `None`.
- **to_yaml** (`bool` / `None`) – If `True`, includes attribute names that **can** be serialized to YAML strings. If `False`, includes attribute names that **cannot** be serialized to YAML strings. If `None`, will not include attributes based on YAML serialization support (but may include them based on other parameters). Defaults to `None`.
- **from_dict** (`bool` / `None`) – If `True`, includes attribute names that **can** be de-serialized from `dict` objects. If `False`, includes attribute names that **cannot** be de-serialized from `dict` objects. If `None`, will not include attributes based on `dict` de-serialization support (but may include them based on other parameters). Defaults to `None`.
- **to_dict** – If `True`, includes attribute names that **can** be serialized to `dict` objects. If `False`, includes attribute names that **cannot** be serialized to `dict` objects. If `None`, will not include attributes based on `dict` serialization support (but may include them based on other parameters). Defaults to `None`.
- **config_set** (`str` / `None`) – If not `None`, will determine serialization support within the indicated *configuration set*. Defaults to `None`.

Returns `True` if the attribute's serialization support matches, `False` if not, and `None` if no serialization support was specified.

Return type `bool` / `None`

Raises

- ***UnsupportedSerializationError*** – if `attribute` is not present on the object
- **ValueError** – if `config_set` is not `None` and its value does not match a named configuration set
- ***ConfigurationError*** – if `config_set` is `None` and the object uses named configuration sets
- ***ConfigurationError*** – if the object does not use configuration sets but `config_set` is not None

dump_to_csv(*include_header=False*, *delimiter='|'*, *wrap_all_strings=False*, *null_text='None'*, *wrapper_character="'"*, *double_wrapper_character_when_nested=False*, *escape_character='\\'*, *line_terminator='\r\n'*, *config_set=None*)

Retrieve a *CSV* representation of the object, *with all attributes* serialized regardless of configuration.

> **Caution:** Nested objects (such as *relationships* or *association proxies*) will **not** be serialized.

> **Note:** This method ignores any `display_name` contributed on the `AttributeConfiguration`.

Parameters

- **include_header** (`bool`) – If `True`, will include a header row with column labels. If `False`, will not include a header row. Defaults to `True`.
- **delimiter** (`str`) – The delimiter used between columns. Defaults to `|`.
- **wrap_all_strings** (`bool`) – If `True`, wraps any string data in the `wrapper_character`. If `None`, only wraps string data if it contains the `delimiter`. Defaults to `False`.
- **null_text** (`str`) – The text value to use in place of empty values. Only applies if `wrap_empty_values` is `True`. Defaults to `'None'`.
- **wrapper_character** (`str`) – The string used to wrap string values when wrapping is necessary. Defaults to `'`.
- **double_wrapper_character_when_nested** (`bool`) – If `True`, will double the `wrapper_character` when it is found inside a column value. If `False`, will precede the `wrapper_character` by the `escape_character` when it is found inside a column value. Defaults to `False`.
- **escape_character** (`str`) – The character to use when escaping nested wrapper characters. Defaults to `\`.
- **line_terminator** (`str`) – The character used to mark the end of a line. Defaults to `\r\n`.
- **config_set** (`str` / `None`) – If not `None`, the named configuration set to use. Defaults to `None`.

Returns Data from the object in CSV format ending in a newline (`\n`).

Return type str

dump_to_dict(*max_nesting=0*, *current_nesting=0*, *config_set=None*)

Return a OrderedDict representation of the object, *with all attributes* regardless of configuration.

> **Caution:** Nested objects (such as *relationships* or *association proxies*) will **not** be serialized.

Parameters

- **max_nesting** (int) – The maximum number of levels that the resulting OrderedDict object can be nested. If set to 0, will not nest other serializable objects. Defaults to 0.
- **current_nesting** (int) – The current nesting level at which the dict representation will reside. Defaults to 0.
- **config_set** (str / None) – If not None, the named configuration set to use when processing the input. Defaults to None.

Returns A OrderedDict representation of the object.

Return type OrderedDict

Raises

- ***SerializableAttributeError*** – if attributes is empty
- ***MaximumNestingExceededError*** – if current_nesting is greater than max_nesting
- ***MaximumNestingExceededWarning*** – if an attribute requires nesting beyond max_nesting

dump_to_json(*max_nesting=0*, *current_nesting=0*, *serialize_function=None*, *config_set=None*, ***kwargs*)

Return a *JSON* representation of the object, *with all attributes* regardless of configuration.

> **Caution:** Nested objects (such as *relationships* or *association proxies*) will **not** be serialized.

Parameters

- **max_nesting** (int) – The maximum number of levels that the resulting JSON object can be nested. If set to 0, will not nest other serializable objects. Defaults to 0.
- **current_nesting** (int) – The current nesting level at which the dict representation will reside. Defaults to 0.
- **serialize_function** (callable / None) – Optionally override the default JSON serializer. Defaults to None, which applies the default simplejson JSON serializer.

  Note: Use the serialize_function parameter to override the default JSON serializer.

  A valid serialize_function is expected to accept a single dict and return a str, similar to simplejson.dumps().

If you wish to pass additional arguments to your `serialize_function` pass them as keyword arguments (in `kwargs`).

- **config_set** (`str` / `None`) – If not `None`, the named configuration set to use. Defaults to `None`.
- **kwargs** (*keyword arguments*) – Optional keyword parameters that are passed to the JSON serializer function. By default, these are options which are passed to `simplejson.dumps()`.

Returns A `str` with the JSON representation of the object.

Return type `str`

Raises

- ***SerializableAttributeError*** – if attributes is empty
- ***MaximumNestingExceededError*** – if `current_nesting` is greater than `max_nesting`
- ***MaximumNestingExceededWarning*** – if an attribute requires nesting beyond `max_nesting`

dump_to_yaml(*max_nesting=0*, *current_nesting=0*, *serialize_function=None*, *config_set=None*, ***kwargs*)

Return a *YAML* representation of the object *with all attributes*, regardless of configuration.

> **Caution:** Nested objects (such as *relationships* or *association proxies*) will **not** be serialized.

Parameters

- **max_nesting** (`int`) – The maximum number of levels that the resulting object can be nested. If set to `0`, will not nest other serializable objects. Defaults to `0`.
- **current_nesting** (`int`) – The current nesting level at which the representation will reside. Defaults to `0`.
- **serialize_function** (callable / `None`) – Optionally override the default YAML serializer. Defaults to `None`, which calls the default `yaml.dump()` function from the PyYAML library.

  Note: Use the `serialize_function` parameter to override the default YAML serializer.

  A valid `serialize_function` is expected to accept a single `dict` and return a `str`, similar to `yaml.dump()`.

  If you wish to pass additional arguments to your `serialize_function` pass them as keyword arguments (in `kwargs`).

- **config_set** (`str` / `None`) – If not `None`, the named configuration set to use. Defaults to `None`.
- **kwargs** (*keyword arguments*) – Optional keyword parameters that are passed to the YAML serializer function. By default, these are options which are passed to `yaml.dump()`.

Returns A `str` with the JSON representation of the object.

Return type `str`

Raises

- ***SerializableAttributeError*** – if attributes is empty
- ***MaximumNestingExceededError*** – if `current_nesting` is greater than `max_nesting`
- ***MaximumNestingExceededWarning*** – if an attribute requires nesting beyond `max_nesting`

classmethod get_attribute_serialization_config(*attribute*, *config_set=None*)

Retrieve the *AttributeConfiguration* for `attribute`.

Parameters

- **attribute** (`str`) – The attribute/column name whose serialization configuration should be returned.
- **config_set** (`str` / `None`) – If not `None`, will return the *AttributeConfiguration* object for `attribute` that is contained within the named *configuration set*. Defaults to `None`.

Returns The *AttributeConfiguration* for `attribute`.

Return type *AttributeConfiguration*

Raises

- ***ConfigurationError*** – if `config_set` is not empty and there are no configuration sets defined on `cls` or if there are configuration sets defined but no `config_set` is specified
- **ValueError** – if `config_set` is not defined within `__serialization__`

classmethod get_csv_column_names(*deserialize=True*, *serialize=True*, *config_set=None*)

Retrieve a list of CSV column names.

Parameters

- **deserialize** (`bool`) – If `True`, returns columns that support *de-serialization*. If `False`, returns columns that do *not* support deserialization. If `None`, does not take deserialization into account. Defaults to `True`.
- **serialize** (`bool`) – If `True`, returns columns that support *serialization*. If `False`, returns columns that do *not* support serialization. If `None`, does not take serialization into account. Defaults to `True`.
- **config_set** (`str` / `None`) – If not `None`, the named configuration set to use. Defaults to `None`.

Returns List of CSV column names, sorted according to their configuration.

Return type `list` of `str`

get_csv_data(*delimiter='|'*, *wrap_all_strings=False*, *null_text='None'*, *wrapper_character="'"*, *double_wrapper_character_when_nested=False*, *escape_character='\\'*, *line_terminator='\r\n'*, *config_set=None*)

Return the CSV representation of the model instance (record).

Parameters

- **delimiter** (`str`) – The delimiter used between columns. Defaults to `|`.

- **wrap_all_strings** (bool) – If True, wraps any string data in the wrapper_character. If None, only wraps string data if it contains the delimiter. Defaults to False.
- **null_text** (str) – The text value to use in place of empty values. Only applies if wrap_empty_values is True. Defaults to 'None'.
- **wrapper_character** (str) – The string used to wrap string values when wrapping is necessary. Defaults to '.
- **double_wrapper_character_when_nested** (bool) – If True, will double the wrapper_character when it is found inside a column value. If False, will precede the wrapper_character by the escape_character when it is found inside a column value. Defaults to False.
- **escape_character** (str) – The character to use when escaping nested wrapper characters. Defaults to \.
- **line_terminator** (str) – The character used to mark the end of a line. Defaults to \r\n.
- **config_set** (str / None) – If not None, the named configuration set to use. Defaults to None.

Returns Data from the object in CSV format ending in line_terminator.

Return type str

classmethod get_csv_header(*deserialize=None*, *serialize=True*, *delimiter='|'*, *wrap_all_strings=False*, *wrapper_character="'"*, *double_wrapper_character_when_nested=False*, *escape_character='\\'*, *line_terminator='\r\n'*, *config_set=None*)

Retrieve a header string for a CSV representation of the model.

Parameters

- **attributes** (list of str) – List of *model attributes* to include.
- **delimiter** (str) – The character(s) to utilize between columns. Defaults to a pipe (|).
- **wrap_all_strings** (bool) – If True, wraps any string data in the wrapper_character. If None, only wraps string data if it contains the delimiter. Defaults to False.
- **null_text** (str) – The text value to use in place of empty values. Only applies if wrap_empty_values is True. Defaults to 'None'.
- **null_text** – The text value to use in place of empty values. Only applies if wrap_empty_values is True. Defaults to 'None'.
- **wrapper_character** (str) – The string used to wrap string values when wrapping is necessary. Defaults to '.
- **double_wrapper_character_when_nested** (bool) – If True, will double the wrapper_character when it is found inside a column value. If False, will precede the wrapper_character by the escape_character when it is found inside a column value. Defaults to False.
- **escape_character** (str) – The character to use when escaping nested wrapper characters. Defaults to \.
- **line_terminator** (str) – The character used to mark the end of a line. Defaults to \r\n.

- **config_set** (str / None) – If not None, the named configuration set to use. Defaults to None.

Returns A string ending in line_terminator with the model's CSV column names listed, separated by the delimiter.

Return type str

classmethod get_csv_serialization_config(*deserialize=True*, *serialize=True*, *config_set=None*)

Retrieve the CSV serialization configurations that apply for this object.

Parameters

- **deserialize** (bool / None) – If True, returns configurations for attributes that **can** be de-serialized from CSV strings. If False, returns configurations for attributes that **cannot** be de-serialized from CSV strings. If None, ignores de-serialization configuration when determining which attribute configurations to return. Defaults to None.
- **serialize** (bool / None) – If True, returns configurations for attributes that **can** be serialized to CSV strings. If False, returns configurations for attributes that **cannot** be serialized to CSV strings. If None, ignores serialization configuration when determining which attribute configurations to return. Defaults to None.
- **config_set** (str / None) – If not None, the named *configuration set* whose CSV serialization configuration should be returned. Defaults to None.

Returns Set of attribute serialization configurations that match the arguments supplied.

Return type list of *AttributeConfiguration*

Raises

- ***ConfigurationError*** – if cls does not use named configuration sets but config_set is not None
- ***ConfigurationError*** – if cls uses named configuration sets but config_set is empty
- **ValueError** – if config_set is not defined within __serialization__

classmethod get_dict_serialization_config(*deserialize=True*, *serialize=True*, *config_set=None*)

Retrieve the dict serialization configurations that apply for this object.

Parameters

- **deserialize** (bool / None) – If True, returns configurations for attributes that **can** be de-serialized from dict objects. If False, returns configurations for attributes that **cannot** be de-serialized from dict objects. If None, ignores de-serialization configuration when determining which attribute configurations to return. Defaults to None.
- **serialize** (bool / None) – If True, returns configurations for attributes that **can** be serialized to dict objects. If False, returns configurations for attributes that **cannot** be serialized to dict objects. If None, ignores serialization configuration when determining which attribute configurations to return. Defaults to None.
- **config_set** (str / None) – If not None, the named *configuration set* whose dict serialization configuration should be returned. Defaults to None.

Returns Set of attribute serialization configurations that match the arguments supplied.

Return type list of *AttributeConfiguration*

classmethod get_json_serialization_config(*deserialize=True*, *serialize=True*, *config_set=None*)

Retrieve the JSON serialization configurations that apply for this object.

Parameters

- **deserialize** (bool / None) – If True, returns configurations for attributes that **can** be de-serialized from JSON strings. If False, returns configurations for attributes that **cannot** be de-serialized from JSON strings. If None, ignores de-serialization configuration when determining which attribute configurations to return. Defaults to None.
- **serialize** (bool / None) – If True, returns configurations for attributes that **can** be serialized to JSON strings. If False, returns configurations for attributes that **cannot** be serialized to JSON strings. If None, ignores serialization configuration when determining which attribute configurations to return. Defaults to None.
- **config_set** (str / None) – If not None, the named *configuration set* whose JSON serialization configuration should be returned. Defaults to None.

Returns Set of attribute serialization configurations that match the arguments supplied.

Return type list of *AttributeConfiguration*

classmethod get_primary_key_column_names()

Retrieve the column names for the model's primary key columns.

Return type list of str

classmethod get_primary_key_columns()

Retrieve the model's primary key columns.

Returns list of Column objects corresponding to the table's primary key(s).

Return type list of Column

classmethod get_serialization_config(*from_csv=None*, *to_csv=None*, *from_json=None*, *to_json=None*, *from_yaml=None*, *to_yaml=None*, *from_dict=None*, *to_dict=None*, *exclude_private=True*, *config_set=None*)

Retrieve a list of *AttributeConfiguration* objects corresponding to attributes whose values can be serialized from/to CSV, JSON, YAML, etc.

Parameters

- **from_csv** (bool / None) – If True, includes attribute names that **can** be de-serialized from CSV strings. If False, includes attribute names that **cannot** be de-serialized from CSV strings. If None, will not include attributes based on CSV de-serialization support (but may include them based on other parameters). Defaults to None.
- **to_csv** (bool / None) – If True, includes attribute names that **can** be serialized to CSV strings. If False, includes attribute names that **cannot** be serialized to CSV strings. If None, will not include attributes based on CSV serialization support (but may include them based on other parameters). Defaults to None.
- **from_json** (bool / None) – If True, includes attribute names that **can** be de-serialized from JSON strings. If False, includes attribute names that **cannot** be de-serialized from JSON strings. If None, will not include attributes based on JSON de-serialization support (but may include them based on other parameters). Defaults to None.
- **to_json** (bool / None) – If True, includes attribute names that **can** be serialized to JSON strings. If False, includes attribute names that **cannot** be serialized to JSON strings. If None, will not include attributes based on JSON serialization support (but may include them based on other parameters). Defaults to None.

- **from_yaml** (bool / None) – If True, includes attribute names that **can** be de-serialized from YAML strings. If False, includes attribute names that **cannot** be de-serialized from YAML strings. If None, will not include attributes based on YAML de-serialization support (but may include them based on other parameters). Defaults to None.
- **to_yaml** (bool / None) – If True, includes attribute names that **can** be serialized to YAML strings. If False, includes attribute names that **cannot** be serialized to YAML strings. If None, will not include attributes based on YAML serialization support (but may include them based on other parameters). Defaults to None.
- **from_dict** (bool / None) – If True, includes attribute names that **can** be de-serialized from dict objects. If False, includes attribute names that **cannot** be de-serialized from dict objects. If None, will not include attributes based on dict de-serialization support (but may include them based on other parameters). Defaults to None.
- **to_dict** – If True, includes attribute names that **can** be serialized to dict objects. If False, includes attribute names that **cannot** be serialized to dict objects. If None, will not include attributes based on dict serialization support (but may include them based on other parameters). Defaults to None.
- **exclude_private** (bool) – If True, will exclude private attributes whose names begin with a single underscore. Defaults to True.
- **config_set** (str / None) – If not None, will return those *AttributeConfiguration* objects that are contained within the named *configuration set*. Defaults to None.

Returns List of attribute configurations.

Return type list of *AttributeConfiguration*

Raises

- ***ConfigurationError*** – if cls does not use named configuration sets but config_set is not None
- ***ConfigurationError*** – if cls uses named configuration sets but config_set is empty
- **ValueError** – if config_set is not defined within __serialization__

classmethod get_yaml_serialization_config(*deserialize=True*, *serialize=True*, *config_set=None*)

Retrieve the YAML serialization configurations that apply for this object.

Parameters

- **deserialize** (bool / None) – If True, returns configurations for attributes that **can** be de-serialized from YAML strings. If False, returns configurations for attributes that **cannot** be de-serialized from YAML strings. If None, ignores de-serialization configuration when determining which attribute configurations to return. Defaults to None.
- **serialize** (bool / None) – If True, returns configurations for attributes that **can** be serialized to YAML strings. If False, returns configurations for attributes that **cannot** be serialized to YAML strings. If None, ignores serialization configuration when determining which attribute configurations to return. Defaults to None.
- **config_set** (str / None) – If not None, the named *configuration set* whose YAML serialization configuration should be returned. Defaults to None.

Returns Set of attribute serialization configurations that match the arguments supplied.

Return type list of *AttributeConfiguration*

classmethod new_from_csv(*csv_data*, *delimiter='|'*, *wrap_all_strings=False*, *null_text='None'*, *wrapper_character="'"*, *double_wrapper_character_when_nested=False*, *escape_character='\\'*, *line_terminator='\r\n'*, *config_set=None*)

Create a new model instance from a CSV record.

Tip: Unwrapped empty column values are automatically interpreted as null (`None`).

Parameters

- **csv_data** (`str` / Path-like object) – The CSV data. If a Path-like object, will read the first record from a file that is assumed to include a header row. If a `str` and has more than one record (line), will assume the first line is a header row.
- **delimiter** (`str`) – The delimiter used between columns. Defaults to `|`.
- **wrapper_character** (`str`) – The string used to wrap string values when wrapping is applied. Defaults to `'`.
- **null_text** (`str`) – The string used to indicate an empty value if empty values are wrapped. Defaults to *None*.
- **config_set** (`str` / `None`) – If not `None`, the named configuration set to use. Defaults to `None`.

Returns A *model instance* created from the record.

Return type model instance

Raises

- ***DeserializationError*** – if `csv_data` is not a valid `str`
- ***CSVStructureError*** – if the columns in `csv_data` do not match the expected columns returned by *`get_csv_column_names()`*
- ***ValueDeserializationError*** – if a value extracted from the CSV failed when executing its *de-serialization function*.

classmethod new_from_dict(*input_data*, *error_on_extra_keys=True*, *drop_extra_keys=False*, *config_set=None*)

Update the model instance from data in a `dict` object.

Parameters

- **input_data** (`dict`) – The input `dict`
- **error_on_extra_keys** (`bool`) – If `True`, will raise an error if an unrecognized key is found in `input_data`. If `False`, will either drop or include the extra key in the result, as configured in the `drop_extra_keys` parameter. Defaults to `True`.

Warning: Be careful setting `error_on_extra_keys` to `False`.

This method's last step passes the keys/values of the processed input data to your model's `__init__()` method.

If your instance's `__init__()` method does not support your extra keys, it will likely raise a `TypeError`.

- **drop_extra_keys** (bool) – If True, will omit unrecognized top-level keys from the resulting dict. If False, will include unrecognized keys or raise an error based on the configuration of the error_on_extra_keys parameter. Defaults to False.
- **config_set** (str / None) – If not None, the named configuration set to use when processing the input. Defaults to None.

Raises

- ***ExtraKeyError*** – if error_on_extra_keys is True and input_data contains top-level keys that are not recognized as attributes for the instance model.
- ***DeserializationError*** – if input_data is not a dict or JSON object serializable to a dict or if input_data is empty.

classmethod new_from_json(*input_data*, *deserialize_function=None*, *error_on_extra_keys=True*, *drop_extra_keys=False*, *config_set=None*, ***kwargs*)

Create a new model instance from data in JSON.

Parameters

- **input_data** (str or Path-like object) – The JSON data to de-serialize.

  Note: If input_data points to a file, and the file contains a list of JSON objects, the first JSON object will be considered.

- **deserialize_function** (callable / None) – Optionally override the default JSON deserializer. Defaults to None, which calls the default simplejson.loads() function from the doc:*simplejson <simplejson:index>* library.

  Note: Use the deserialize_function parameter to override the default JSON deserializer.

  A valid deserialize_function is expected to accept a single str and return a dict, similar to simplejson.loads().

  If you wish to pass additional arguments to your deserialize_function pass them as keyword arguments (in kwargs).

- **error_on_extra_keys** (bool) – If True, will raise an error if an unrecognized key is found in input_data. If False, will either drop or include the extra key in the result, as configured in the drop_extra_keys parameter. Defaults to True.

  Warning: Be careful setting error_on_extra_keys to False.

  This method's last step passes the keys/values of the processed input data to your model's __init__() method.

  If your instance's __init__() method does not support your extra keys, it will likely raise a TypeError.

- **drop_extra_keys** (bool) – If True, will ignore unrecognized top-level keys in input_data. If False, will include unrecognized keys or raise an error based on the configuration of the error_on_extra_keys parameter. Defaults to False.

- **config_set** (str / None) – If not None, the named configuration set to use. Defaults to None.
- **kwargs** (*keyword arguments*) – Optional keyword parameters that are passed to the JSON deserializer function. By default, these are options which are passed to simplejson.loads().

Raises

- ***ExtraKeyError*** – if error_on_extra_keys is True and input_data contains top-level keys that are not recognized as attributes for the instance model.
- ***DeserializationError*** – if input_data is not a dict or JSON object serializable to a dict or if input_data is empty.

classmethod new_from_yaml(*input_data*, *deserialize_function=None*, *error_on_extra_keys=True*, *drop_extra_keys=False*, *config_set=None*, ***kwargs*)

Create a new model instance from data in YAML.

Parameters

- **input_data** (str / Path-like object) – The YAML data to de-serialize. May be either a str or a Path-like object to a YAML file.
- **deserialize_function** (callable / None) – Optionally override the default YAML deserializer. Defaults to None, which calls the default yaml.safe_load() function from the PyYAML library.

Note: Use the deserialize_function parameter to override the default YAML deserializer.

A valid deserialize_function is expected to accept a single str and return a dict, similar to yaml.safe_load().

If you wish to pass additional arguments to your deserialize_function pass them as keyword arguments (in kwargs).

- **error_on_extra_keys** (bool) – If True, will raise an error if an unrecognized key is found in input_data. If False, will either drop or include the extra key in the result, as configured in the drop_extra_keys parameter. Defaults to True.

> **Warning:** Be careful setting error_on_extra_keys to False.
>
> This method's last step passes the keys/values of the processed input data to your model's __init__() method.
>
> If your instance's __init__() method does not support your extra keys, it will likely raise a TypeError.

- **drop_extra_keys** (bool) – If True, will ignore unrecognized top-level keys in input_data. If False, will include unrecognized keys or raise an error based on the configuration of the error_on_extra_keys parameter. Defaults to False.
- **config_set** (str / None) – If not None, the named configuration set to use. Defaults to None.

Raises

- ***ExtraKeyError*** – if `error_on_extra_keys` is `True` and `input_data` contains top-level keys that are not recognized as attributes for the instance model.
- ***DeserializationError*** – if `input_data` is not a `dict` or JSON object serializable to a `dict` or if `input_data` is empty.

classmethod set_attribute_serialization_config(*attribute*, *config=None*, *supports_csv=None*, *csv_sequence=None*, *supports_json=None*, *supports_yaml=None*, *supports_dict=None*, *on_deserialize=None*, *on_serialize=None*, *config_set=None*)

Set the serialization/de-serialization configuration for `attribute`.

Note: Supplying keyword arguments like `supports_csv` or `supports_json` will override any configuration set in `config`.

Parameters

- **attribute** (`str`) – The name of the *model attribute* whose serialization/de-serialization configuration is to be configured.
- **config** (*AttributeConfiguration* / / Pydantic `ModelField` object / / `None`) – The *AttributeConfiguration* to apply. If `None`, will set particular values based on their corresponding keyword arguments.
- **supports_csv** (`bool` / `tuple` of form (inbound: `bool`, outbound: `bool`) / `None`) – Determines whether the column can be serialized to or de-serialized from CSV format.

  If `True`, can be serialized to CSV and de-serialized from CSV. If `False`, will not be included when serialized to CSV and will be ignored if present in a de-serialized CSV.

  Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

  If `None`, will retain whatever configuration is currently applied. Defaults to `None`
- **supports_json** (`bool` / `tuple` of form (inbound: `bool`, outbound: `bool`) / `None`) – Determines whether the column can be serialized to or de-serialized from JSON format.

  If `True`, can be serialized to JSON and de-serialized from JSON. If `False`, will not be included when serialized to JSON and will be ignored if present in a de-serialized JSON.

  Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

  If `None`, will retain whatever configuration is currently applied. Defaults to `None`
- **supports_yaml** (`bool` / `tuple` of form (inbound: `bool`, outbound: `bool`) / `None`) – Determines whether the column can be serialized to or de-serialized from YAML format.

  If `True`, can be serialized to YAML and de-serialized from YAML. If `False`, will not be included when serialized to YAML and will be ignored if present in a de-serialized YAML.

  Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

  If `None`, will retain whatever configuration is currently applied. Defaults to `None`

- **supports_dict** (bool / tuple of form (inbound: bool, outbound: bool) / None) – Determines whether the column can be serialized to or de-serialized to a Python dict.

  If True, can be serialized to dict and de-serialized from a dict. If False, will not be included when serialized to dict and will be ignored if present in a de-serialized dict.

  Can also accept a 2-member tuple (inbound / outbound) which determines de-serialization and serialization support respectively.

  If None, will retain whatever configuration is currently applied. Defaults to None

- **on_deserialize** (callable / dict with formats as keys and values as callables / None) – A function that will be called when attempting to assign a de-serialized value to the column. This is intended to either coerce the value being assigned to a form that is acceptable by the column, or raise an exception if it cannot be coerced.

  Tip: If you need to execute different on_deserialize functions for different formats, you can also supply a dict:

  ```
  on_deserialize = {
    'csv': csv_on_deserialize_callable,
    'json': json_on_deserialize_callable,
    'yaml': yaml_on_deserialize_callable,
    'dict': dict_on_deserialize_callable
  }
  ```

  If False, will clear the current configuration to apply the default.

  If None, will retain whatever configuration is currently applied. Defaults to None

  ---

  Tip: To clear the on_deserialize function, you can either supply a value of False or pass a dict with particular formats set to None:

  ```
  on_deserialize = {
    'csv': None,
    'json': None,
    'yaml': None,
    'dict': None
  }

  # is equivalent to:

  on_deserialize = False
  ```

  This will revert the *on_deserialize* function to the attribute's default *on_deserialize* function based on its data type.

- **on_serialize** (callable / dict with formats as keys and values as callables / None / False) – A function that will be called when attempting to serialize a value from the column.

  Tip: If you need to execute different on_serialize functions for different formats, you can also supply a dict:

```
on_serialize = {
  'csv': csv_on_serialize_callable,
  'json': json_on_serialize_callable,
  'yaml': yaml_on_serialize_callable,
  'dict': dict_on_serialize_callable
}
```

If `False`, will clear the current configuration to apply the default configuration.

If `None`, will retain whatever configuration is currently applied. Defaults to `None`

Tip: To clear the `on_serialize` function, you need to pass `False` or a `dict` with particular formats set to `None`:

```
on_serialize = {
  'csv': None,
  'json': None,
  'yaml': None,
  'dict': None
}

# is equivalent to

on_serialize = False
```

This will revert the *on_serialize* function to the attribute's default *on_serialize* function based on its data type.

- **csv_sequence** (`int` / `None` / `False`) – Indicates the numbered position that the column should be in in a valid CSV-version of the object.

Note: If not specified, the column will go after any columns that *do* have a `csv_sequence` assigned, sorted alphabetically.

If two columns have the same `csv_sequence`, they will be sorted alphabetically.

If `False`, will set the position to `None` <python:None>‘ which will position the column *after* any explicitly positioned columns in alphabetical order.

If `None`, will retain whatever configuration is currently applied. Defaults to `None`

- **config_set** (`str` / `None`) – The name of the *configuration set* where the serialization/de-serialization configuration for `attribute` should be updated. Defaults to `None`

Warning: If the `config_set` is not defined on the model, then a *`ConfigurationError`* will be raised.

Raises

- ***ConfigurationError*** – if `config_set` is not empty and there are no configuration sets defined on `cls` or if there are configuration sets defined but no `config_set` is specified

- **ValueError** – if `config_set` is not defined within `__serialization__`
- **ValueError** – if `attribute` does not match `config.name` if `config` is not `None`

to_csv(*include_header=False*, *delimiter='|'*, *wrap_all_strings=False*, *null_text='None'*, *wrapper_character="'"*, *double_wrapper_character_when_nested=False*, *escape_character='\\'*, *line_terminator='\r\n'*, *config_set=None*)

Retrieve a CSV string with the object's data.

Parameters

- **include_header** (`bool`) – If `True`, will include a header row with column labels. If `False`, will not include a header row. Defaults to `True`.
- **delimiter** (`str`) – The delimiter used between columns. Defaults to `|`.
- **wrap_all_strings** (`bool`) – If `True`, wraps any string data in the `wrapper_character`. If `None`, only wraps string data if it contains the `delimiter`. Defaults to `False`.
- **null_text** (`str`) – The text value to use in place of empty values. Only applies if `wrap_empty_values` is `True`. Defaults to `'None'`.
- **wrapper_character** (`str`) – The string used to wrap string values when wrapping is necessary. Defaults to `'`.
- **double_wrapper_character_when_nested** (`bool`) – If `True`, will double the `wrapper_character` when it is found inside a column value. If `False`, will precede the `wrapper_character` by the `escape_character` when it is found inside a column value. Defaults to `False`.
- **escape_character** (`str`) – The character to use when escaping nested wrapper characters. Defaults to `\`.
- **line_terminator** (`str`) – The character used to mark the end of a line. Defaults to `\r\n`.
- **config_set** (`str` / `None`) – If not `None`, the named configuration set to use. Defaults to `None`.

Returns Data from the object in CSV format ending in a newline (`\n`).

Return type `str`

to_dict(*max_nesting=0*, *current_nesting=0*, *config_set=None*)

Return a `OrderedDict` representation of the object.

Parameters

- **max_nesting** (`int`) – The maximum number of levels that the resulting `dict` object can be nested. If set to `0`, will not nest other serializable objects. Defaults to `0`.
- **current_nesting** (`int`) – The current nesting level at which the `dict` representation will reside. Defaults to `0`.
- **config_set** (`str` / `None`) – If not `None`, the named configuration set to use when processing the input. Defaults to `None`.

Returns A `OrderedDict` representation of the object.

Return type `OrderedDict`

Raises

- ***SerializableAttributeError*** – if attributes is empty

- ***MaximumNestingExceededError*** – if `current_nesting` is greater than `max_nesting`
- ***MaximumNestingExceededWarning*** – if an attribute requires nesting beyond `max_nesting`

to_json(*max_nesting=0*, *current_nesting=0*, *serialize_function=None*, *config_set=None*, ***kwargs*)
Return a JSON representation of the object.

Parameters

- **max_nesting** (`int`) – The maximum number of levels that the resulting JSON object can be nested. If set to `0`, will not nest other serializable objects. Defaults to `0`.
- **current_nesting** (`int`) – The current nesting level at which the `dict` representation will reside. Defaults to `0`.
- **serialize_function** (callable / `None`) – Optionally override the default JSON serializer. Defaults to `None`, which applies the default simplejson JSON serializer.

Note: Use the `serialize_function` parameter to override the default JSON serializer.

A valid `serialize_function` is expected to accept a single `dict` and return a `str`, similar to `simplejson.dumps()`.

If you wish to pass additional arguments to your `serialize_function` pass them as keyword arguments (in `kwargs`).

- **config_set** (`str` / `None`) – If not `None`, the named configuration set to use. Defaults to `None`.
- **kwargs** (*keyword arguments*) – Optional keyword parameters that are passed to the JSON serializer function. By default, these are options which are passed to `simplejson.dumps()`.

Returns A `str` with the JSON representation of the object.

Return type `str`

Raises

- ***SerializableAttributeError*** – if attributes is empty
- ***MaximumNestingExceededError*** – if `current_nesting` is greater than `max_nesting`
- ***MaximumNestingExceededWarning*** – if an attribute requires nesting beyond `max_nesting`

to_yaml(*max_nesting=0*, *current_nesting=0*, *serialize_function=None*, *config_set=None*, ***kwargs*)
Return a YAML representation of the object.

Parameters

- **max_nesting** (`int`) – The maximum number of levels that the resulting object can be nested. If set to `0`, will not nest other serializable objects. Defaults to `0`.
- **current_nesting** (`int`) – The current nesting level at which the representation will reside. Defaults to `0`.

- **serialize_function** (callable / None) – Optionally override the default YAML serializer. Defaults to None, which calls the default yaml.dump() function from the PyYAML library.

  Note: Use the serialize_function parameter to override the default YAML serializer.

  A valid serialize_function is expected to accept a single dict and return a str, similar to yaml.dump().

  If you wish to pass additional arguments to your serialize_function pass them as keyword arguments (in kwargs).

- **config_set** (str / None) – If not None, the named configuration set to use. Defaults to None.
- **kwargs** (*keyword arguments*) – Optional keyword parameters that are passed to the YAML serializer function. By default, these are options which are passed to yaml.dump().

Returns A str with the JSON representation of the object.

Return type str

Raises

- ***SerializableAttributeError*** – if attributes is empty
- ***MaximumNestingExceededError*** – if current_nesting is greater than max_nesting
- ***MaximumNestingExceededWarning*** – if an attribute requires nesting beyond max_nesting

update_from_csv(*csv_data*, *delimiter='|'*, *wrap_all_strings=False*, *null_text='None'*, *wrapper_character="'"*, *double_wrapper_character_when_nested=False*, *escape_character='\\'*, *line_terminator='\r\n'*, *config_set=None*)

Update the model instance from a CSV record.

Tip: Unwrapped empty column values are automatically interpreted as null (None).

Parameters

- **csv_data** (str / Path-like object) – The CSV data. If a Path-like object, will read the first record from a file that is assumed to include a header row. If a str and has more than one record (line), will assume the first line is a header row.
- **delimiter** (str) – The delimiter used between columns. Defaults to |.
- **wrapper_character** (str) – The string used to wrap string values when wrapping is applied. Defaults to '.
- **null_text** (str) – The string used to indicate an empty value if empty values are wrapped. Defaults to *None*.
- **config_set** (str / None) – If not None, the named configuration set to use. Defaults to None.

Raises

- ***DeserializationError*** – if `csv_data` is not a valid `str`
- ***CSVStructureError*** – if the columns in `csv_data` do not match the expected columns returned by *`get_csv_column_names()`*
- ***ValueDeserializationError*** – if a value extracted from the CSV failed when executing its *de-serialization function*.

update_from_dict(*input_data*, *error_on_extra_keys=True*, *drop_extra_keys=False*, *config_set=None*)

Update the model instance from data in a `dict` object.

Parameters

- **input_data** (`dict`) – The input `dict`
- **error_on_extra_keys** (`bool`) – If `True`, will raise an error if an unrecognized key is found in `input_data`. If `False`, will either drop or include the extra key in the result, as configured in the `drop_extra_keys` parameter. Defaults to `True`.

  > **Warning:** Be careful setting `error_on_extra_keys` to `False`.
  >
  > This method's last step attempts to set an attribute on the model instance for every top-level key in the parsed/processed input data.
  >
  > If there is an extra key that cannot be set as an attribute on your model instance, it *will* raise `AttributeError`.

- **drop_extra_keys** (`bool`) – If `True`, will omit unrecognized top-level keys from the resulting `dict`. If `False`, will include unrecognized keys or raise an error based on the configuration of the `error_on_extra_keys` parameter. Defaults to `False`.
- **config_set** (`str` / `None`) – If not `None`, the named configuration set to use when processing the input. Defaults to `None`.

Raises

- ***ExtraKeyError*** – if `error_on_extra_keys` is `True` and `input_data` contains top-level keys that are not recognized as attributes for the instance model.
- ***DeserializationError*** – if `input_data` is not a `dict` or JSON object serializable to a `dict` or if `input_data` is empty.

update_from_json(*input_data*, *deserialize_function=None*, *error_on_extra_keys=True*, *drop_extra_keys=False*, *config_set=None*, ***kwargs*)

Update the model instance from data in a JSON string.

Parameters

- **input_data** (`str` or Path-like object) – The JSON data to de-serialize.

  > **Note:** If `input_data` points to a file, and the file contains a list of JSON objects, the first JSON object will be considered.

- **deserialize_function** (callable / `None`) – Optionally override the default JSON deserializer. Defaults to `None`, which calls the default `simplejson.loads()` function from the simplejson library.

Note: Use the `deserialize_function` parameter to override the default JSON deserializer.

A valid `deserialize_function` is expected to accept a single `str` and return a `dict`, similar to `simplejson.loads()`.

If you wish to pass additional arguments to your `deserialize_function` pass them as keyword arguments (in `kwargs`).

- **`error_on_extra_keys`** (`bool`) – If `True`, will raise an error if an unrecognized key is found in `input_data`. If `False`, will either drop or include the extra key in the result, as configured in the `drop_extra_keys` parameter. Defaults to `True`.

> **Warning:** Be careful setting `error_on_extra_keys` to `False`.
>
> This method's last step attempts to set an attribute on the model instance for every top-level key in the parsed/processed input data.
>
> If there is an extra key that cannot be set as an attribute on your model instance, it *will* raise `AttributeError`.

- **`drop_extra_keys`** (`bool`) – If `True`, will ignore unrecognized keys in the input data. If `False`, will include unrecognized keys or raise an error based on the configuration of the `error_on_extra_keys` parameter. Defaults to `False`.
- **`config_set`** (`str` / `None`) – If not `None`, the named configuration set to use. Defaults to `None`.
- **`kwargs`** (*`keyword arguments`*) – Optional keyword parameters that are passed to the JSON deserializer function.By default, these are options which are passed to `simplejson.loads()`.

Raises

- ***`ExtraKeyError`*** – if `error_on_extra_keys` is `True` and `input_data` contains top-level keys that are not recognized as attributes for the instance model.
- ***`DeserializationError`*** – if `input_data` is not a `str` JSON de-serializable object to a `dict` or if `input_data` is empty.

`update_from_yaml`(*input_data*, *deserialize_function=None*, *error_on_extra_keys=True*, *drop_extra_keys=False*, *config_set=None*, ***kwargs*)

Update the model instance from data in a YAML string.

Parameters

- **`input_data`** (`str` / Path-like object) – The YAML data to de-serialize. May be either a `str` or a Path-like object to a YAML file.
- **`deserialize_function`** (callable / `None`) – Optionally override the default YAML deserializer. Defaults to `None`, which calls the default `yaml.safe_load()` function from the PyYAML library.

Note: Use the `deserialize_function` parameter to override the default YAML deserializer.

A valid `deserialize_function` is expected to accept a single `str` and return a `dict`, similar to `yaml.safe_load()`.

If you wish to pass additional arguments to your `deserialize_function` pass them as keyword arguments (in `kwargs`).

- **`error_on_extra_keys`** (`bool`) – If `True`, will raise an error if an unrecognized key is found in `input_data`. If `False`, will either drop or include the extra key in the result, as configured in the `drop_extra_keys` parameter. Defaults to `True`.

  Warning: Be careful setting `error_on_extra_keys` to `False`.

  This method's last step attempts to set an attribute on the model instance for every top-level key in the parsed/processed input data.

  If there is an extra key that cannot be set as an attribute on your model instance, it *will* raise `AttributeError`.

- **`drop_extra_keys`** (`bool`) – If `True`, will ignore unrecognized keys in the input data. If `False`, will include unrecognized keys or raise an error based on the configuration of the `error_on_extra_keys` parameter. Defaults to `False`.
- **`config_set`** (`str` / `None`) – If not `None`, the named configuration set to use. Defaults to `None`.
- **`kwargs`** (*`keyword arguments`*) – Optional keyword parameters that are passed to the YAML deserializer function. By default, these are options which are passed to `yaml.safe_load()`.

Raises

- ***`ExtraKeyError`*** – if `error_on_extra_keys` is `True` and `input_data` contains top-level keys that are not recognized as attributes for the instance model.
- ***`DeserializationError`*** – if `input_data` is not a `str` YAML de-serializable object to a `dict` or if `input_data` is empty.

`primary_key_value`

The instance's primary key.

Note: If not `None`, this value can always be passed to `Query.get()`.

Warning: Returns `None` if the instance is pending, in a transient state, or does not have a primary key.

Returns scalar or `tuple` value representing the primary key. For a composite primary key, the order of identifiers corresponds to the order with which the model's primary keys were defined. If no primary keys are available, will return `None`.

Return type scalar / `tuple` / `None`

4.1.2 declarative_base()

declarative_base(*cls=<class 'sqlathanor.declarative.base_model.BaseModel'>*, ***kwargs*)

Construct a base class for declarative class definitions.

The new base class will be given a metaclass that produces appropriate `Table` objects and makes the appropriate `mapper` calls based on the information provided declaratively in the class and any subclasses of the class.

Parameters

- **cls** (`None` / `tuple` of classes / class object) – Defaults to *`BaseModel`* to provide serialization/de-serialization support.

  If a `tuple` of classes, will include *`BaseModel`* in that list of classes to mixin serialization/de-serialization support.

  If not `None` and not a `tuple`, will mixin *`BaseModel`* with the value passed to provide serialization/de-serialization support.

- **kwargs** (*`keyword arguments`*) – Additional keyword arguments supported by the original `sqlalchemy.ext.declarative.declarative_base()` function

Returns Base class for declarative class definitions with support for serialization and de-serialization.

4.1.3 @as_declarative

as_declarative(***kw*)

Class decorator for *`declarative_base()`*.

Provides a syntactical shortcut to the `cls` argument sent to *`declarative_base()`*, allowing the base class to be converted in-place to a "declarative" base:

```
from sqlathanor import as_declarative

@as_declarative()
class Base(object):
    @declared_attr
    def __tablename__(cls):
        return cls.__name__.lower()

    id = Column(Integer,
                primary_key = True,
                supports_csv = True)

class MyMappedClass(Base):
    # ...
```

Tip: All keyword arguments passed to *`as_declarative()`* are passed along to *`declarative_base()`*.

See also:

- *`declarative_base()`*

4.1.4 generate_model_from_csv()

generate_model_from_csv(*serialized*, *tablename*, *primary_key*, *cls=<class 'sqlathanor.declarative.base_model.BaseModel'>*, *serialization_config=None*, *skip_nested=True*, *default_to_str=False*, *type_mapping=None*, *base_model_attrs=None*, *delimiter='|'*, *wrap_all_strings=False*, *null_text='None'*, *wrapper_character="'"*, *double_wrapper_character_when_nested=False*, *escape_character='\\'*, *line_terminator='\r\n'*, ***kwargs*)

Generate a *model class* from a serialized *CSV* string.

Note: This function *cannot* programmatically create *relationships*, *hybrid properties*, or *association proxies*.

Parameters

- **serialized** (`str` / Path-like object / `list`) – The CSV data whose column headers will be treated as column names, while value data types will determine *model attribute* data types.

  Note: If a Path-like object, will read the file contents from a file that is assumed to include a header row. If a `str` and has more than one record (line), will assume the first line is a header row. If a `list`, will assume the first item is the header row.

- **tablename** (`str`) – The name of the SQL table to which the model corresponds.
- **primary_key** (`str`) – The name of the column/key that should be used as the table's primary key.
- **cls** (`None` / `tuple` of classes / class object) – The base class to use when generating a new *model class*. Defaults to *`BaseModel`* to provide serialization/de-serialization support.

  If a `tuple` of classes, will include *`BaseModel`* in that list of classes to mixin serialization/de-serialization support.

  If not `None` and not a `tuple`, will mixin *`BaseModel`* with the value passed to provide serialization/de-serialization support.

- **serialization_config** (Iterable of *`AttributeConfiguration`* or coercable `dict` objects / `None`) – Collection of *`AttributeConfiguration`* that determine the generated model's *serialization*/*de-serialization configuration*. If `None`, will support serialization and de-serialization across all keys in `serialized_dict`. Defaults to `None`.
- **skip_nested** (`bool`) – If `True` then any keys in `serialized_json` that feature nested items (e.g. iterables, JSON objects, etc.) will be ignored. If `False`, will treat serialized items as `str`. Defaults to `True`.
- **default_to_str** (`bool`) – If `True`, will automatically set a key/column whose value type cannot be determined to `str` (`Text`). If `False`, will use the value type's `__name__` attribute and attempt to find a mapping. Defaults to `False`.
- **type_mapping** (`dict` with type names as keys and column data types as values.) – Determines how value types in `serialized` map to SQL column data types. To add a new mapping or override a default, set a key to the name of the value type in Python, and set the value to a SQLAlchemy Data Type. The following are the default mappings applied:

Python Literal	SQL Column Type
bool	Boolean
str	Text
int	Integer
float	Float
date	Date
datetime	DateTime
time	Time

- **base_model_attrs** (dict / None) – Optional dict of special attributes that will be applied to the generated *BaseModel* (e.g. __table_args__). Keys will correspond to the attribute name, while the value is the value that will be applied. Defaults to None.
- **delimiter** (str) – The delimiter used between columns. Defaults to |.
- **wrapper_character** (str) – The string used to wrap string values when wrapping is applied. Defaults to '.
- **null_text** (str) – The string used to indicate an empty value if empty values are wrapped. Defaults to *None*.
- **kwargs** – Any additional keyword arguments will be passed to *declarative_base()* when generating the programmatic *BaseModel*.

Returns *Model class* whose structure matches serialized.

Return type *BaseModel*

Raises

- ***UnsupportedValueTypeError*** – when a value in serialized does not have a corresponding key in type_mapping
- **ValueError** – if tablename is empty
- ***DeserializationError*** – if serialized is not a valid str
- ***CSVStructureError*** – if there are less than 2 (two) rows in serialized or if column headers are not valid Python variable names

---

4.1.5 generate_model_from_json()

generate_model_from_json(*serialized*, *tablename*, *primary_key*, *deserialize_function=None*, *cls=<class 'sqlathanor.declarative.base_model.BaseModel'>*, *serialization_config=None*, *skip_nested=True*, *default_to_str=False*, *type_mapping=None*, *base_model_attrs=None*, *deserialize_kwargs=None*, ***kwargs*)

Generate a *model class* from a serialized *JSON* string.

Note: This function *cannot* programmatically create *relationships*, *hybrid properties*, or *association proxies*.

Parameters

- **serialized** (str / Path-like object) – The JSON data whose keys will be treated as column names, while value data types will determine *model attribute* data types, or the path to a file whose contents will be the JSON object in question.

  Note: If providing a path to a file, and if the file contains more than one JSON object, will only use the first JSON object listed.

- **tablename** (str) – The name of the SQL table to which the model corresponds.
- **primary_key** (str) – The name of the column/key that should be used as the table's primary key.
- **deserialize_function** (callable / None) – Optionally override the default JSON deserializer. Defaults to None, which calls the default simplejson.loads() function from the doc:*simplejson <simplejson:index>* library.

  Note: Use the deserialize_function parameter to override the default JSON deserializer.

  A valid deserialize_function is expected to accept a single str and return a dict, similar to simplejson.loads().

  If you wish to pass additional arguments to your deserialize_function pass them as keyword arguments (in deserialize_kwargs).

- **cls** (None / tuple of classes / class object) – The base class to use when generating a new *model class*. Defaults to *BaseModel* to provide serialization/de-serialization support.

  If a tuple of classes, will include *BaseModel* in that list of classes to mixin serialization/de-serialization support.

  If not None and not a tuple, will mixin *BaseModel* with the value passed to provide serialization/de-serialization support.

- **serialization_config** (Iterable of *AttributeConfiguration* or coercable dict objects / None) – Collection of *AttributeConfiguration* that determine the generated model's *serialization/de-serialization configuration*. If None, will support serialization and de-serialization across all keys in serialized_dict. Defaults to None.
- **skip_nested** (bool) – If True then any keys in serialized_json that feature nested items (e.g. iterables, JSON objects, etc.) will be ignored. If False, will treat serialized items as str. Defaults to True.
- **default_to_str** (bool) – If True, will automatically set a key/column whose value type cannot be determined to str (Text). If False, will use the value type's __name__ attribute and attempt to find a mapping. Defaults to False.
- **type_mapping** (dict with type names as keys and column data types as values.) – Determines how value types in serialized map to SQL column data types. To add a new mapping or override a default, set a key to the name of the value type in Python, and set the value to a SQLAlchemy Data Type. The following are the default mappings applied:

Python Literal	SQL Column Type
bool	Boolean
str	Text
int	Integer
float	Float
date	Date
datetime	DateTime
time	Time

- **base_model_attrs** (dict / None) – Optional dict of special attributes that will be applied to the generated *BaseModel* (e.g. __table_args__). Keys will correspond to the attribute name, while the value is the value that will be applied. Defaults to None.
- **deserialize_kwargs** (dict / None) – Optional additional keyword arguments that will be passed to the deserialize function. Defaults to None.
- **kwargs** – Any additional keyword arguments will be passed to *declarative_base()* when generating the programmatic *BaseModel*.

Returns *Model class* whose structure matches serialized.

Return type *BaseModel*

Raises

- ***UnsupportedValueTypeError*** – when a value in serialized does not have a corresponding key in type_mapping
- **ValueError** – if tablename is empty

4.1.6 generate_model_from_yaml()

generate_model_from_yaml(*serialized*, *tablename*, *primary_key*, *deserialize_function=None*, *cls=<class 'sqlathanor.declarative.base_model.BaseModel'>*, *serialization_config=None*, *skip_nested=True*, *default_to_str=False*, *type_mapping=None*, *base_model_attrs=None*, *deserialize_kwargs=None*, ***kwargs*)

Generate a *model class* from a serialized *YAML* string.

Note: This function *cannot* programmatically create *relationships*, *hybrid properties*, or *association proxies*.

Parameters

- **serialized** (str / Path-like object) – The YAML data whose keys will be treated as column names, while value data types will determine *model attribute* data types, or the path to a file whose contents will be the YAML object in question.
- **tablename** (str) – The name of the SQL table to which the model corresponds.
- **primary_key** (str) – The name of the column/key that should be used as the table's primary key.

- **deserialize_function** (callable / None) – Optionally override the default YAML deserializer. Defaults to None, which calls the default yaml.safe_load() function from the PyYAML library.

  Note: Use the deserialize_function parameter to override the default YAML deserializer.

  A valid deserialize_function is expected to accept a single str and return a dict, similar to yaml.safe_load().

  If you wish to pass additional arguments to your deserialize_function pass them as keyword arguments (in kwargs).

- **cls** (None / tuple of classes / class object) – The base class to use when generating a new *model class*. Defaults to *BaseModel* to provide serialization/de-serialization support.

  If a tuple of classes, will include *BaseModel* in that list of classes to mixin serialization/de-serialization support.

  If not None and not a tuple, will mixin *BaseModel* with the value passed to provide serialization/de-serialization support.

- **serialization_config** (Iterable of *AttributeConfiguration* or coercable dict objects / None) – Collection of *AttributeConfiguration* that determine the generated model's *serialization/de-serialization configuration*. If None, will support serialization and de-serialization across all keys in serialized_dict. Defaults to None.

- **skip_nested** (bool) – If True then any keys in serialized_json that feature nested items (e.g. iterables, JSON objects, etc.) will be ignored. If False, will treat serialized items as str. Defaults to True.

- **default_to_str** (bool) – If True, will automatically set a key/column whose value type cannot be determined to str (Text). If False, will use the value type's __name__ attribute and attempt to find a mapping. Defaults to False.

- **type_mapping** (dict with type names as keys and column data types as values.) – Determines how value types in serialized map to SQL column data types. To add a new mapping or override a default, set a key to the name of the value type in Python, and set the value to a SQLAlchemy Data Type. The following are the default mappings applied:

Python Literal	SQL Column Type
bool	Boolean
str	Text
int	Integer
float	Float
date	Date
datetime	DateTime
time	Time

- **base_model_attrs** (dict / None) – Optional dict of special attributes that will be applied to the generated *BaseModel* (e.g. __table_args__). Keys will correspond to the attribute name, while the value is the value that will be applied. Defaults to None.

- **deserialize_kwargs** (dict / None) – Optional additional keyword arguments that will be passed to the deserialize function. Defaults to None.

- **kwargs** – Any additional keyword arguments will be passed to *declarative_base()* when generating the programmatic *BaseModel*.

Returns *Model class* whose structure matches `serialized`.

Return type *BaseModel*

Raises

- ***UnsupportedValueTypeError*** – when a value in `serialized` does not have a corresponding key in `type_mapping`
- **ValueError** – if `tablename` is empty

4.1.7 generate_model_from_dict()

generate_model_from_dict(*serialized_dict*, *tablename*, *primary_key*, *cls=<class 'sqlathanor.declarative.base_model.BaseModel'>*, *serialization_config=None*, *skip_nested=True*, *default_to_str=False*, *type_mapping=None*, *base_model_attrs=None*, ***kwargs*)

Generate a *model class* from a serialized `dict`.

Note: This function *cannot* programmatically create *relationships*, *hybrid properties*, or *association proxies*.

Parameters

- **serialized_dict** (`dict`) – The `dict` that has been de-serialized from a given string. Keys will be treated as column names, while value data types will determine *model attribute* data types.
- **tablename** (`str`) – The name of the SQL table to which the model corresponds.
- **primary_key** (`str`) – The name of the column/key that should be used as the table's primary key.
- **cls** (`None` / `tuple` of classes / class object) – The base class to use when generating a new *model class*. Defaults to *BaseModel* to provide serialization/de-serialization support.

  If a `tuple` of classes, will include *BaseModel* in that list of classes to mixin serialization/de-serialization support.

  If not `None` and not a `tuple`, will mixin *BaseModel* with the value passed to provide serialization/de-serialization support.
- **serialization_config** (Iterable of *AttributeConfiguration* or coercable `dict` objects / `None`) – Collection of *AttributeConfiguration* that determine the generated model's *serialization*/*de-serialization configuration*. If `None`, will support serialization and de-serialization across all keys in `serialized_dict`. Defaults to `None`.
- **skip_nested** (`bool`) – If `True` then any keys in `serialized_dict` that feature nested items (e.g. iterables, `dict` objects, etc.) will be ignored. If `False`, will treat serialized items as `str`. Defaults to `True`.
- **default_to_str** (`bool`) – If `True`, will automatically set a key/column whose value type cannot be determined to `str` (`Text`). If `False`, will use the value type's `__name__` attribute and attempt to find a mapping. Defaults to `False`.

- **type_mapping** (dict with type names as keys and column data types as values.) – Determines how value types in serialized_dict map to SQL column data types. To add a new mapping or override a default, set a key to the name of the value type in Python, and set the value to a SQLAlchemy Data Type. The following are the default mappings applied:

Python Literal	SQL Column Type
bool	Boolean
str	Text
int	Integer
float	Float
date	Date
datetime	DateTime
time	Time

- **base_model_attrs** (dict / None) – Optional dict of special attributes that will be applied to the generated *BaseModel* (e.g. __table_args__). Keys will correspond to the attribute name, while the value is the value that will be applied. Defaults to None.
- **kwargs** – Any additional keyword arguments will be passed to *declarative_base()* when generating the programmatic *BaseModel*.

Returns *Model class* whose structure matches serialized_dict.

Return type *BaseModel*

Raises

- ***UnsupportedValueTypeError*** – when a value in serialized_dict does not have a corresponding key in type_mapping
- **ValueError** – if serialized_dict is not a dict or is empty
- **ValueError** – if tablename is empty
- **ValueError** – if primary_key is empty

4.1.8 generate_model_from_pydantic()

generate_model_from_pydantic(*pydantic_models*, *tablename*, *primary_key*, *cls=<class 'sqlathanor.declarative.base_model.BaseModel'>*, *skip_nested=True*, *default_to_str=False*, *type_mapping=None*, *base_model_attrs=None*, ***kwargs*)

Generate a *model class* from one or more *Pydantic models*.

Note: This function *cannot* programmatically create *relationships*, *hybrid properties*, or *association proxies*.

Parameters

- **pydantic_models** (pydantic.BaseModel / iterable of pydantic.BaseModel / dict whose keys correspond to *configuration set* names and whose value must be a pydantic.BaseModel) – The *Pydantic Model(s)* which will determine the ORM columns/attributes that will be present within the generated *model class*.

Hint: If supplying an iterable of *Pydantic models*, then all models will be coalesced into a single *model class*. If supplying a `dict` whose values are separate *Pydantic models*, then each `dict` key will correspond to a single serialization/de-serialization *configuration set* in the resulting *model class*.

- **tablename** (`str`) – The name of the SQL table to which the model corresponds.
- **primary_key** (`str`) – The name of the column/key that should be used as the table's primary key.
- **cls** (`None` / `tuple` of classes / class object) – The base class to use when generating a new *model class*. Defaults to *`BaseModel`* to provide serialization/de-serialization support.

  If a `tuple` of classes, will include *`BaseModel`* in that list of classes to mixin serialization/de-serialization support.

  If not `None` and not a `tuple`, will mixin *`BaseModel`* with the value passed to provide serialization/de-serialization support.
- **skip_nested** (`bool`) – If `True` then any keys in `serialized_dict` that feature nested items (e.g. iterables, `dict` objects, etc.) will be ignored. If `False`, will treat serialized items as `str`. Defaults to `True`.
- **default_to_str** (`bool`) – If `True`, will automatically set a key/column whose value type cannot be determined to `str` (`Text`). If `False`, will use the value type's `__name__` attribute and attempt to find a mapping. Defaults to `False`.
- **type_mapping** (`dict` with type names as keys and column data types as values.) – Determines how value types in `pydantic_models` map to SQL column data types. To add a new mapping or override a default, set a key to the name of the value type in Python, and set the value to a SQLAlchemy Data Type. The following are the default mappings applied:

Python Literal	SQL Column Type
`bool`	`Boolean`
`str`	`Text`
`int`	`Integer`
`float`	`Float`
`date`	`Date`
`datetime`	`DateTime`
`time`	`Time`

- **base_model_attrs** (`dict` / `None`) – Optional `dict` of special attributes that will be applied to the generated *`BaseModel`* (e.g. `__table_args__`). Keys will correspond to the attribute name, while the value is the value that will be applied. Defaults to `None`.
- **kwargs** – Any additional keyword arguments will be passed to *`declarative_base()`* when generating the programmatic *`BaseModel`*.

Returns *Model class* whose structure matches `pydantic_models` and whose serialization/de-serialization configuration corresponds to the pattern implied by `pydantic_models` structure

Return type *`BaseModel`*

Raises

- ***`UnsupportedValueTypeError`*** – when a value in `pydantic_models` does not have a corresponding key in `type_mapping`

- **ValueError** – if `pydantic_models` is not a supported type or is empty
- **ValueError** – if `tablename` is empty
- **ValueError** – if `primary_key` is empty

4.2 Schema

The classes defined in `sqlathanor.schema` are drop-in replacements for SQLAlchemy's Core Schema elements.

They add *serialization* and *de-serialization* support to your model's columns and relationsips, and so should replace your imports of their SQLAlchemy analogs.

See also:

- *Using SQLAthanor*

4.2.1 Column

class Column(**args*, ***kwargs*)

Represents a column in a database table. Inherits from `sqlalchemy.schema.Column`

__init__(**args*, ***kwargs*)

Construct a new `Column` object.

> **Warning:** This method is analogous to the original SQLAlchemy `Column.__init__()` from which it inherits. The only difference is that it supports additional keyword arguments which are not supported in the original, and which are documented below.
>
> **For the original SQLAlchemy version, see:**
>
> - SQLAlchemy: `Column.__init__()`

Parameters

- **supports_csv** (`bool` / `tuple` of form (inbound: `bool`, outbound: `bool`)) – Determines whether the column can be serialized to or de-serialized from CSV format.

  If `True`, can be serialized to CSV and de-serialized from CSV. If `False`, will not be included when serialized to CSV and will be ignored if present in a de-serialized CSV.

  Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to `False`, which means the column will not be serialized to CSV or de-serialized from CSV.

- **csv_sequence** (`int` / `None`) – Indicates the numbered position that the column should be in in a valid CSV-version of the object. Defaults to `None`.

  Note: If not specified, the column will go after any columns that *do* have a `csv_sequence` assigned, sorted alphabetically.

  If two columns have the same `csv_sequence`, they will be sorted alphabetically.

- **supports_json** (bool / tuple of form (inbound: bool, outbound: bool)) – Determines whether the column can be serialized to or de-serialized from JSON format.

  If True, can be serialized to JSON and de-serialized from JSON. If False, will not be included when serialized to JSON and will be ignored if present in a de-serialized JSON.

  Can also accept a 2-member tuple (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to False, which means the column will not be serialized to JSON or de-serialized from JSON.

- **supports_yaml** (bool / tuple of form (inbound: bool, outbound: bool)) – Determines whether the column can be serialized to or de-serialized from YAML format.

  If True, can be serialized to YAML and de-serialized from YAML. If False, will not be included when serialized to YAML and will be ignored if present in a de-serialized YAML.

  Can also accept a 2-member tuple (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to False, which means the column will not be serialized to YAML or de-serialized from YAML.

- **supports_dict** (bool / tuple of form (inbound: bool, outbound: bool)) – Determines whether the column can be serialized to or de-serialized to a Python dict.

  If True, can be serialized to dict and de-serialized from a dict. If False, will not be included when serialized to dict and will be ignored if present in a de-serialized dict.

  Can also accept a 2-member tuple (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to False, which means the column will not be serialized to a dict or de-serialized from a dict.

- **on_deserialize** (callable / dict with formats as keys and values as callables) – A function that will be called when attempting to assign a de-serialized value to the column. This is intended to either coerce the value being assigned to a form that is acceptable by the column, or raise an exception if it cannot be coerced. If None, the data type's default on_deserialize function will be called instead.

  Tip: If you need to execute different on_deserialize functions for different formats, you can also supply a dict:

  ```
  on_deserialize = {
    'csv': csv_on_deserialize_callable,
    'json': json_on_deserialize_callable,
    'yaml': yaml_on_deserialize_callable,
    'dict': dict_on_deserialize_callable
  }
  ```

  Defaults to None.

- **on_serialize** (callable / dict with formats as keys and values as callables) – A function that will be called when attempting to serialize a value from the column. If None, the data type's default on_serialize function will be called instead.

Tip: If you need to execute different `on_serialize` functions for different formats, you can also supply a `dict`:

```
on_serialize = {
  'csv': csv_on_serialize_callable,
  'json': json_on_serialize_callable,
  'yaml': yaml_on_serialize_callable,
  'dict': dict_on_serialize_callable
}
```

Defaults to `None`.

4.2.2 relationship()

relationship(*argument*, *supports_json = False*, *supports_yaml = False*, *supports_dict = False*, ***kwargs*)

Provide a relationship between two mapped classes.

See also:

- *RelationshipProperty*
- `sqlalchemy.orm.relationship()`

Warning: This constructor is analogous to the original SQLAlchemy `relationship()` from which it inherits. The only difference is that it supports additional keyword arguments which are not supported in the original, and which are documented below.

For the original SQLAlchemy version, see: `sqlalchemy.orm.relationship()`

Parameters

- **argument** – see `sqlalchemy.orm.relationship()`
- **supports_json** (`bool` / `tuple` of form (inbound: `bool`, outbound: `bool`)) – Determines whether the column can be serialized to or de-serialized from JSON format.

  If `True`, can be serialized to JSON and de-serialized from JSON. If `False`, will not be included when serialized to JSON and will be ignored if present in a de-serialized JSON.

  Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to `False`, which means the column will not be serialized to JSON or de-serialized from JSON.

- **supports_yaml** (`bool` / `tuple` of form (inbound: `bool`, outbound: `bool`)) – Determines whether the column can be serialized to or de-serialized from YAML format.

  If `True`, can be serialized to YAML and de-serialized from YAML. If `False`, will not be included when serialized to YAML and will be ignored if present in a de-serialized YAML.

  Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

Defaults to `False`, which means the column will not be serialized to YAML or de-serialized from YAML.

- **supports_dict** (`bool` / `tuple` of form (inbound: `bool`, outbound: `bool`)) – Determines whether the column can be serialized to or de-serialized to a Python `dict`.

  If `True`, can be serialized to `dict` and de-serialized from a `dict`. If `False`, will not be included when serialized to `dict` and will be ignored if present in a de-serialized `dict`.

  Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to `False`, which means the column will not be serialized to a `dict` or de-serialized from a `dict`.

4.2.3 Table

class Table(**args*, ***kwargs*)

Represents a table in a database. Inherits from `sqlalchemy.schema.Table`

__init__(**args*, ***kwargs*)

Construct a new `Table` object.

> **Warning:** This method is analogous to the original SQLAlchemy `Table.__init__()` from which it inherits. The only difference is that it supports additional keyword arguments which are not supported in the original, and which are documented below.
>
> **For the original SQLAlchemy version, see:**
>
> - **SQLAlchemy:** `sqlalchemy.schema.Table`

classmethod from_csv(*serialized*, *tablename*, *metadata*, *primary_key*, *column_kwargs=None*, *skip_nested=True*, *default_to_str=False*, *type_mapping=None*, *delimiter='|'*, *wrap_all_strings=False*, *null_text='None'*, *wrapper_character="'"*, *double_wrapper_character_when_nested=False*, *escape_character='\\'*, *line_terminator='\r\n'*, ***kwargs*)

Generate a *`Table`* object from a *CSV* string.

Parameters

- **serialized** (`str` / Path-like object / `list`) – The CSV data whose column headers will be treated as column names, while value data types will determine *model attribute* data types.

  > **Note:** If a Path-like object, will read the file contents from a file that is assumed to include a header row. If a `str` and has more than one record (line), will assume the first line is a header row. If a `list`, will assume the first item is the header row.

- **tablename** (`str`) – The name of the SQL table to which the model corresponds.
- **metadata** (`MetaData`) – a `MetaData` object which will contain this table. The metadata is used as a point of association of this table with other tables which are referenced via foreign key. It also may be used to associate this table with a particular `Connectable`.

- **primary_key** (str) – The name of the column/key that should be used as the table's primary key.
- **column_kwargs** (dict / None) – An optional dictionary whose keys correspond to column/key, and whose values are themselves dictionaries with keyword arguments that will be passed ot the applicable *Column* constructor. Defaults to None.
- **skip_nested** (bool) – If True then any keys in serialized that feature nested items (e.g. iterables, dict objects, etc.) will be ignored. If False, will treat nested items as str. Defaults to True.
- **default_to_str** (bool) – If True, will automatically set a key/column whose value type cannot be determined to str (Text). If False, will use the value type's __name__ attribute and attempt to find a mapping. Defaults to False.
- **type_mapping** (dict with type names as keys and column data types as values.) – Determines how value types in serialized map to SQL column data types. To add a new mapping or override a default, set a key to the name of the value type in Python, and set the value to a SQLAlchemy Data Type. The following are the default mappings applied:

Python Literal	SQL Column Type
bool	Boolean
str	Text
int	Integer
float	Float
date	Date
datetime	DateTime
time	Time

- **delimiter** (str) – The delimiter used between columns. Defaults to |.
- **wrapper_character** (str) – The string used to wrap string values when wrapping is applied. Defaults to '.
- **null_text** (str) – The string used to indicate an empty value if empty values are wrapped. Defaults to *None*.
- **kwargs** – Any additional keyword arguments will be passed to the *Table* constructor. For a full list of options, please see sqlalchemy.schema.Table.

Returns A *Table* object.

Return type *Table*

Raises

- ***DeserializationError*** – if serialized is not a valid str
- ***UnsupportedValueTypeError*** – when a value in serialized does not have a corresponding key in type_mapping
- **ValueError** – if tablename is empty
- **ValueError** – if primary_key is empty
- ***CSVStructureError*** – if there are less than 2 (two) rows in serialized or if column headers are not valid Python variable names

classmethod from_dict(*serialized*, *tablename*, *metadata*, *primary_key*, *column_kwargs=None*, *skip_nested=True*, *default_to_str=False*, *type_mapping=None*, ***kwargs*)

Generate a *Table* object from a dict.

Parameters

- **serialized** (dict) – The dict that has been de-serialized from a given string. Keys will be treated as column names, while value data types will determine *Column* data types.
- **tablename** (str) – The name of the SQL table to which the model corresponds.
- **metadata** (MetaData) – a MetaData object which will contain this table. The metadata is used as a point of association of this table with other tables which are referenced via foreign key. It also may be used to associate this table with a particular Connectable.
- **primary_key** (str) – The name of the column/key that should be used as the table's primary key.
- **column_kwargs** (dict / None) – An optional dictionary whose keys correspond to column/key, and whose values are themselves dictionaries with keyword arguments that will be passed ot the applicable *Column* constructor. Defaults to None.
- **skip_nested** (bool) – If True then any keys in serialized that feature nested items (e.g. iterables, dict objects, etc.) will be ignored. If False, will treat nested items as str. Defaults to True.
- **default_to_str** (bool) – If True, will automatically set a key/column whose value type cannot be determined to str (Text). If False, will use the value type's __name__ attribute and attempt to find a mapping. Defaults to False.
- **type_mapping** (dict with type names as keys and column data types as values.) – Determines how value types in serialized map to SQL column data types. To add a new mapping or override a default, set a key to the name of the value type in Python, and set the value to a SQLAlchemy Data Type. The following are the default mappings applied:

Python Literal	SQL Column Type
bool	Boolean
str	Text
int	Integer
float	Float
date	Date
datetime	DateTime
time	Time

- **kwargs** – Any additional keyword arguments will be passed to the *Table* constructor. For a full list of options, please see sqlalchemy.schema.Table.

Returns A *Table* object.

Return type *Table*

Raises

- ***UnsupportedValueTypeError*** – when a value in serialized does not have a corresponding key in type_mapping
- **ValueError** – if serialized is not a dict or is empty

- **ValueError** – if `tablename` is empty
- **ValueError** – if `primary_key` is empty

classmethod from_json(*serialized*, *tablename*, *metadata*, *primary_key*, *column_kwargs=None*, *skip_nested=True*, *default_to_str=False*, *type_mapping=None*, *deserialize_function=None*, *deserialize_kwargs=None*, ***kwargs*)

Generate a *Table* object from a *JSON* string.

Parameters

- **serialized** (`str` / Path-like object) – The *JSON* data to use. Keys will be treated as column names, while value data types will determine *Column* data types.

  Note: If providing a path to a file, and if the file contains more than one JSON object, will only use the first JSON object listed.

- **tablename** (`str`) – The name of the SQL table to which the model corresponds.
- **metadata** (`MetaData`) – a `MetaData` object which will contain this table. The metadata is used as a point of association of this table with other tables which are referenced via foreign key. It also may be used to associate this table with a particular `Connectable`.
- **primary_key** (`str`) – The name of the column/key that should be used as the table's primary key.
- **column_kwargs** (`dict` / `None`) – An optional dictionary whose keys correspond to column/key, and whose values are themselves dictionaries with keyword arguments that will be passed ot the applicable *Column* constructor. Defaults to `None`.
- **skip_nested** (`bool`) – If `True` then any keys in `serialized` that feature nested items (e.g. iterables, `dict` objects, etc.) will be ignored. If `False`, will treat nested items as `str`. Defaults to `True`.
- **default_to_str** (`bool`) – If `True`, will automatically set a key/column whose value type cannot be determined to `str` (`Text`). If `False`, will use the value type's `__name__` attribute and attempt to find a mapping. Defaults to `False`.
- **type_mapping** (`dict` with type names as keys and column data types as values.) – Determines how value types in `serialized` map to SQL column data types. To add a new mapping or override a default, set a key to the name of the value type in Python, and set the value to a SQLAlchemy Data Type. The following are the default mappings applied:

Python Literal	SQL Column Type
`bool`	`Boolean`
`str`	`Text`
`int`	`Integer`
`float`	`Float`
`date`	`Date`
`datetime`	`DateTime`
`time`	`Time`

- **deserialize_function** (callable / `None`) – Optionally override the default JSON deserializer. Defaults to `None`, which calls the default `simplejson.loads()` function from the doc:*simplejson <simplejson:index>* library.

Note: Use the `deserialize_function` parameter to override the default JSON deserializer.

A valid `deserialize_function` is expected to accept a single `str` and return a `dict`, similar to `simplejson.loads()`.

If you wish to pass additional arguments to your `deserialize_function` pass them as keyword arguments (in `deserialize_kwargs`).

- **deserialize_kwargs** (`dict` / `None`) – Optional additional keyword arguments that will be passed to the deserialize function. Defaults to `None`.
- **kwargs** – Any additional keyword arguments will be passed to the *`Table`* constructor. For a full list of options, please see `sqlalchemy.schema.Table`.

Returns A *`Table`* object.

Return type *`Table`*

Raises

- ***UnsupportedValueTypeError*** – when a value in `serialized` does not have a corresponding key in `type_mapping`
- ***DeserializationError*** – if `serialized` is not a valid `str`
- **ValueError** – if `tablename` is empty
- **ValueError** – if `primary_key` is empty

classmethod from_yaml(*serialized*, *tablename*, *metadata*, *primary_key*, *column_kwargs=None*, *skip_nested=True*, *default_to_str=False*, *type_mapping=None*, *deserialize_function=None*, *deserialize_kwargs=None*, ***kwargs*)

Generate a *`Table`* object from a *YAML* string.

Parameters

- **serialized** (`str` / Path-like object) – The *YAML* string to use. Keys will be treated as column names, while value data types will determine *`Column`* data types.

Note: If providing a path to a file, and if the file contains more than one object, will only use the first object listed.

- **tablename** (`str`) – The name of the SQL table to which the model corresponds.
- **metadata** (`MetaData`) – a `MetaData` object which will contain this table. The metadata is used as a point of association of this table with other tables which are referenced via foreign key. It also may be used to associate this table with a particular `Connectable`.
- **primary_key** (`str`) – The name of the column/key that should be used as the table's primary key.
- **column_kwargs** (`dict` / `None`) – An optional dictionary whose keys correspond to column/key, and whose values are themselves dictionaries with keyword arguments that will be passed ot the applicable *`Column`* constructor. Defaults to `None`.
- **skip_nested** (`bool`) – If `True` then any keys in `serialized` that feature nested items (e.g. iterables, `dict` objects, etc.) will be ignored. If `False`, will treat nested items as `str`. Defaults to `True`.

- **default_to_str** (bool) – If True, will automatically set a key/column whose value type cannot be determined to str (Text). If False, will use the value type's __name__ attribute and attempt to find a mapping. Defaults to False.
- **type_mapping** (dict with type names as keys and column data types as values.) – Determines how value types in serialized map to SQL column data types. To add a new mapping or override a default, set a key to the name of the value type in Python, and set the value to a SQLAlchemy Data Type. The following are the default mappings applied:

Python Literal	SQL Column Type
bool	Boolean
str	Text
int	Integer
float	Float
date	Date
datetime	DateTime
time	Time

- **deserialize_function** (callable / None) – Optionally override the default YAML deserializer. Defaults to None, which calls the default yaml.safe_load() function from the PyYAML library.

Note: Use the deserialize_function parameter to override the default YAML deserializer.

A valid deserialize_function is expected to accept a single str and return a dict, similar to yaml.safe_load().

If you wish to pass additional arguments to your deserialize_function pass them as keyword arguments (in kwargs).

- **deserialize_kwargs** (dict / None) – Optional additional keyword arguments that will be passed to the deserialize function. Defaults to None.
- **kwargs** – Any additional keyword arguments will be passed to the *Table* constructor. For a full list of options, please see sqlalchemy.schema.Table.

Returns A *Table* object.

Return type *Table*

Raises

- ***UnsupportedValueTypeError*** – when a value in serialized does not have a corresponding key in type_mapping
- ***DeserializationError*** – if serialized is not a valid str
- **ValueError** – if tablename is empty
- **ValueError** – if primary_key is empty

classmethod from_pydantic(*models*, *tablename*, *metadata*, *primary_key*, *skip_nested=True*, *default_to_str=False*, *type_mapping=None*, ***kwargs*)

Generate a *Table* object from one or more *Pydantic models*.

Parameters

- **models** (pydantic.BaseModel / iterable of pydantic.BaseModel / dict whose keys correspond to *configuration set* names and whose value must be a pydantic.BaseModel) – The *Pydantic model(s)* which will determine the columns/attributes that will be present within the generated table.

  Hint: If supplying an iterable of *Pydantic models*, then all models will be coalesced into a single *Table*. If supplying a dict whose values are separate *Pydantic models*, then each dict key will correspond to a single serialization/de-serialization *configuration set* in the resulting *model class*.

- **tablename** (str) – The name of the SQL table to which the model corresponds.
- **metadata** (MetaData) – a MetaData object which will contain this table. The metadata is used as a point of association of this table with other tables which are referenced via foreign key. It also may be used to associate this table with a particular Connectable.
- **primary_key** (str) – The name of the column/key that should be used as the table's primary key.
- **skip_nested** (bool) – If True then any keys in serialized that feature nested items (e.g. iterables, dict objects, etc.) will be ignored. If False, will treat nested items as str. Defaults to True.
- **default_to_str** (bool) – If True, will automatically set a key/column whose value type cannot be determined to str (Text). If False, will use the value type's __name__ attribute and attempt to find a mapping. Defaults to False.
- **type_mapping** (dict with type names as keys and column data types as values.) – Determines how value types in serialized map to SQL column data types. To add a new mapping or override a default, set a key to the name of the value type in Python, and set the value to a SQLAlchemy Data Type. The following are the default mappings applied:

Python Literal	SQL Column Type
bool	Boolean
str	Text
int	Integer
float	Float
date	Date
datetime	DateTime
time	Time

- **kwargs** – Any additional keyword arguments will be passed to the *Table* constructor. For a full list of options, please see sqlalchemy.schema.Table.

Returns A *Table* object.

Return type *Table*

Raises

- ***UnsupportedValueTypeError*** – when a value in serialized does not have a corresponding key in type_mapping
- **ValueError** – if models is empty or is not an acceptable type
- **ValueError** – if tablename is empty

- **ValueError** – if `primary_key` is empty

4.3 Attribute Configuration

The following classes and functions are used to apply *meta configuration* to your SQLAlchemy model.

See also:

- *Using SQLAthanor*

4.3.1 AttributeConfiguration

class AttributeConfiguration(**args*, ***kwargs*)

Serialization/de-serialization configuration of a *model attribute*.

__init__(**args*, ***kwargs*)

Construct an *`AttributeConfiguration`* object.

Parameters

- **name** (`str`) – The name of the attribute. Defaults to `None`.
- **supports_csv** (`bool` / `tuple` of form (inbound: `bool`, outbound: `bool`)) – Determines whether the column can be serialized to or de-serialized from CSV format.

  If `True`, can be serialized to CSV and de-serialized from CSV. If `False`, will not be included when serialized to CSV and will be ignored if present in a de-serialized CSV.

  Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to `False`, which means the column will not be serialized to CSV or de-serialized from CSV.
- **supports_json** (`bool` / `tuple` of form (inbound: `bool`, outbound: `bool`)) – Determines whether the column can be serialized to or de-serialized from JSON format.

  If `True`, can be serialized to JSON and de-serialized from JSON. If `False`, will not be included when serialized to JSON and will be ignored if present in a de-serialized JSON.

  Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to `False`, which means the column will not be serialized to JSON or de-serialized from JSON.
- **supports_yaml** (`bool` / `tuple` of form (inbound: `bool`, outbound: `bool`)) – Determines whether the column can be serialized to or de-serialized from YAML format.

  If `True`, can be serialized to YAML and de-serialized from YAML. If `False`, will not be included when serialized to YAML and will be ignored if present in a de-serialized YAML.

  Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to `False`, which means the column will not be serialized to YAML or de-serialized from YAML.

- **supports_dict** (bool / tuple of form (inbound: bool, outbound: bool)) – Determines whether the column can be serialized to or de-serialized to a Python dict.

  If True, can be serialized to dict and de-serialized from a dict. If False, will not be included when serialized to dict and will be ignored if present in a de-serialized dict.

  Can also accept a 2-member tuple (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to False, which means the column will not be serialized to a dict or de-serialized from a dict.

- **on_deserialize** (callable / dict with formats as keys and values as callables) – A function that will be called when attempting to assign a de-serialized value to the column. This is intended to either coerce the value being assigned to a form that is acceptable by the column, or raise an exception if it cannot be coerced. If None, the data type's default on_deserialize function will be called instead.

  Tip: If you need to execute different on_deserialize functions for different formats, you can also supply a dict:

  ```
  on_deserialize = {
    'csv': csv_on_deserialize_callable,
    'json': json_on_deserialize_callable,
    'yaml': yaml_on_deserialize_callable,
    'dict': dict_on_deserialize_callable
  }
  ```

  Defaults to None.

- **on_serialize** (callable / dict with formats as keys and values as callables) – A function that will be called when attempting to serialize a value from the column. If None, the data type's default on_serialize function will be called instead.

  Tip: If you need to execute different on_serialize functions for different formats, you can also supply a dict:

  ```
  on_serialize = {
    'csv': csv_on_serialize_callable,
    'json': json_on_serialize_callable,
    'yaml': yaml_on_serialize_callable,
    'dict': dict_on_serialize_callable
  }
  ```

  Defaults to None.

- **csv_sequence** (int / None) – Indicates the numbered position that the column should be in in a valid CSV-version of the object. Defaults to None.

  Note: If not specified, the column will go after any columns that *do* have a csv_sequence assigned, sorted alphabetically.

  If two columns have the same csv_sequence, they will be sorted alphabetically.

- **attribute** (*class attribute*) – The object representation of an attribute. Supplying this value overrides any other configuration options supplied. Defaults to `None`.
- **display_name** (`str` / `None`) – An optional name to use or expect in place of *name* when serializing or de-serializing the attribute. If `None`, the attribute will default to the same name as in its Python model class and as provided in `name`. Defaults to `None`
- **pydantic_field** (`pydantic.fields.ModelField` / `pydantic.fields.FieldInfo` / `None`) – An optional Pydantic `ModelField` which can be used to validate the attribute on serialization. Defaults to `None`.

Note: If present, values will be validated against the Pydantic field *after* any `on_deserialize` function is executed.

---

classmethod from_pydantic_model(*model*, *name*, ***kwargs*)

Return a new *`AttributeConfiguration`* instance produced from a Pydantic model's field definition for `name`.

Parameters

- **model** (Pydantic `ModelMetaclass`) – The *Pydantic model* whose field should be used.
- **name** (`str`) – The name of the field to convert to convert to an *`AttributeConfiguration`*
- **supports_csv** (`bool` / `tuple` of form (inbound: `bool`, outbound: `bool`)) – Determines whether the column can be serialized to or de-serialized from CSV format.

  If `True`, can be serialized to CSV and de-serialized from CSV. If `False`, will not be included when serialized to CSV and will be ignored if present in a de-serialized CSV.

  Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to `False`, which means the column will not be serialized to CSV or de-serialized from CSV.

- **supports_json** (`bool` / `tuple` of form (inbound: `bool`, outbound: `bool`)) – Determines whether the column can be serialized to or de-serialized from JSON format.

  If `True`, can be serialized to JSON and de-serialized from JSON. If `False`, will not be included when serialized to JSON and will be ignored if present in a de-serialized JSON.

  Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to `False`, which means the column will not be serialized to JSON or de-serialized from JSON.

- **supports_yaml** (`bool` / `tuple` of form (inbound: `bool`, outbound: `bool`)) – Determines whether the column can be serialized to or de-serialized from YAML format.

  If `True`, can be serialized to YAML and de-serialized from YAML. If `False`, will not be included when serialized to YAML and will be ignored if present in a de-serialized YAML.

  Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to `False`, which means the column will not be serialized to YAML or de-serialized from YAML.

- **supports_dict** (bool / tuple of form (inbound: bool, outbound: bool)) – Determines whether the column can be serialized to or de-serialized to a Python dict.

  If True, can be serialized to dict and de-serialized from a dict. If False, will not be included when serialized to dict and will be ignored if present in a de-serialized dict.

  Can also accept a 2-member tuple (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to False, which means the column will not be serialized to a dict or de-serialized from a dict.

- **on_deserialize** (callable / dict with formats as keys and values as callables) – A function that will be called when attempting to assign a de-serialized value to the column. This is intended to either coerce the value being assigned to a form that is acceptable by the column, or raise an exception if it cannot be coerced. If None, the data type's default on_deserialize function will be called instead.

  Tip: If you need to execute different on_deserialize functions for different formats, you can also supply a dict:

  ```
  on_deserialize = {
    'csv': csv_on_deserialize_callable,
    'json': json_on_deserialize_callable,
    'yaml': yaml_on_deserialize_callable,
    'dict': dict_on_deserialize_callable
  }
  ```

  Defaults to None.

- **on_serialize** (callable / dict with formats as keys and values as callables) – A function that will be called when attempting to serialize a value from the column. If None, the data type's default on_serialize function will be called instead.

  Tip: If you need to execute different on_serialize functions for different formats, you can also supply a dict:

  ```
  on_serialize = {
    'csv': csv_on_serialize_callable,
    'json': json_on_serialize_callable,
    'yaml': yaml_on_serialize_callable,
    'dict': dict_on_serialize_callable
  }
  ```

  Defaults to None.

- **csv_sequence** (int / None) – Indicates the numbered position that the column should be in in a valid CSV-version of the object. Defaults to None.

  Note: If not specified, the column will go after any columns that *do* have a csv_sequence assigned, sorted alphabetically.

  If two columns have the same csv_sequence, they will be sorted alphabetically.

Returns An *AttributeConfiguration* for attribute `name` derived from the Pydantic model.

Return type *AttributeConfiguration*

Raises

- **ValueError** – if `model` is not a Pydantic `ModelMetaclass`
- **validator_collection.errors.InvalidVariableName** – if `name` is not a valid variable name
- **FieldNotFoundError** – if a field with `name` is not found within `model`

classmethod from_attribute(*attribute*)

Return an instance of *AttributeConfiguration* configured for a given attribute.

classmethod from_pydantic_model(*model*, *name*, ***kwargs*)

Return a new *AttributeConfiguration* instance produced from a Pydantic model's field definition for `name`.

Parameters

- **model** (Pydantic `ModelMetaclass`) – The *Pydantic model* whose field should be used.
- **name** (`str`) – The name of the field to convert to convert to an *AttributeConfiguration*
- **supports_csv** (`bool` / `tuple` of form (inbound: `bool`, outbound: `bool`)) – Determines whether the column can be serialized to or de-serialized from CSV format.

  If `True`, can be serialized to CSV and de-serialized from CSV. If `False`, will not be included when serialized to CSV and will be ignored if present in a de-serialized CSV.

  Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to `False`, which means the column will not be serialized to CSV or de-serialized from CSV.

- **supports_json** (`bool` / `tuple` of form (inbound: `bool`, outbound: `bool`)) – Determines whether the column can be serialized to or de-serialized from JSON format.

  If `True`, can be serialized to JSON and de-serialized from JSON. If `False`, will not be included when serialized to JSON and will be ignored if present in a de-serialized JSON.

  Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to `False`, which means the column will not be serialized to JSON or de-serialized from JSON.

- **supports_yaml** (`bool` / `tuple` of form (inbound: `bool`, outbound: `bool`)) – Determines whether the column can be serialized to or de-serialized from YAML format.

  If `True`, can be serialized to YAML and de-serialized from YAML. If `False`, will not be included when serialized to YAML and will be ignored if present in a de-serialized YAML.

  Can also accept a 2-member `tuple` (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to `False`, which means the column will not be serialized to YAML or de-serialized from YAML.

- **supports_dict** (bool / tuple of form (inbound: bool, outbound: bool)) – Determines whether the column can be serialized to or de-serialized to a Python dict.

  If True, can be serialized to dict and de-serialized from a dict. If False, will not be included when serialized to dict and will be ignored if present in a de-serialized dict.

  Can also accept a 2-member tuple (inbound / outbound) which determines de-serialization and serialization support respectively.

  Defaults to False, which means the column will not be serialized to a dict or de-serialized from a dict.

- **on_deserialize** (callable / dict with formats as keys and values as callables) – A function that will be called when attempting to assign a de-serialized value to the column. This is intended to either coerce the value being assigned to a form that is acceptable by the column, or raise an exception if it cannot be coerced. If None, the data type's default on_deserialize function will be called instead.

  Tip: If you need to execute different on_deserialize functions for different formats, you can also supply a dict:

  ```
  on_deserialize = {
    'csv': csv_on_deserialize_callable,
    'json': json_on_deserialize_callable,
    'yaml': yaml_on_deserialize_callable,
    'dict': dict_on_deserialize_callable
  }
  ```

  Defaults to None.

- **on_serialize** (callable / dict with formats as keys and values as callables) – A function that will be called when attempting to serialize a value from the column. If None, the data type's default on_serialize function will be called instead.

  Tip: If you need to execute different on_serialize functions for different formats, you can also supply a dict:

  ```
  on_serialize = {
    'csv': csv_on_serialize_callable,
    'json': json_on_serialize_callable,
    'yaml': yaml_on_serialize_callable,
    'dict': dict_on_serialize_callable
  }
  ```

  Defaults to None.

- **csv_sequence** (int / None) – Indicates the numbered position that the column should be in in a valid CSV-version of the object. Defaults to None.

  Note: If not specified, the column will go after any columns that *do* have a csv_sequence assigned, sorted alphabetically.

  If two columns have the same csv_sequence, they will be sorted alphabetically.

Returns An *AttributeConfiguration* for attribute `name` derived from the Pydantic model.

Return type *AttributeConfiguration*

Raises

- **ValueError** – if `model` is not a Pydantic `ModelMetaclass`
- **validator_collection.errors.InvalidVariableName** – if `name` is not a valid variable name
- **FieldNotFoundError** – if a field with `name` is not found within `model`

classmethod fromkeys(*seq*, *value=None*)

Create a new *AttributeConfiguration* with keys in `seq` and values in `value`.

Parameters

- **seq** (*iterable*) – Iterable of keys
- **value** (*iterable*) – Iterable of values

Return type *AttributeConfiguration*

csv_sequence

Column position when serialized to or de-serialized from CSV.

Note: If `None`, will default to alphabetical sorting *after* any attributes that have an explicit `csv_sequence` provided.

Return type `int` / `None`.

display_name

The property name to apply when serializing the attribute or to expect when de-serializing.

Note: If `None`, will default to the attribute name in the Python model class.

Return type `str` / `None`.

name

The name of the attribute.

Return type `str` / `None`

on_deserialize

A function that will be called when attempting to assign a de-serialized value to the attribute.

This is intended to either coerce the value being assigned to a form that is acceptable by the attribute, or raise an exception if it cannot be coerced.

If `None`, the data type's default `on_deserialize` function will be called instead.

Tip: If you need to execute different `on_deserialize` functions for different formats, you can also supply a `dict`:

```
on_deserialize = {
    'csv': csv_on_deserialize_callable,
    'json': json_on_deserialize_callable,
    'yaml': yaml_on_deserialize_callable,
    'dict': dict_on_deserialize_callable
}
```

Defaults to `None`.

Return type callable / `dict` with formats as keys and values as callables

`on_serialize`
A function that will be called when attempting to serialize a value from the attribute.

If `None`, the data type's default `on_serialize` function will be called instead.

Tip: If you need to execute different `on_serialize` functions for different formats, you can also supply a `dict`:

```
on_serialize = {
    'csv': csv_on_serialize_callable,
    'json': json_on_serialize_callable,
    'yaml': yaml_on_serialize_callable,
    'dict': dict_on_serialize_callable
}
```

Defaults to `None`.

Return type callable / `dict` with formats as keys and values as callables

`pydantic_field`
A Pydantic `ModelField` object that can be used to validate the attribute. Defaults to `None`.

Return type `pydantic.fields.ModelField` / `pydantic.fields.FieldInfo` / `None`

`supports_csv`
Indicates whether the attribute can be serialized / de-serialized to CSV.

Returns 2-member `tuple` (inbound de-serialization / outbound serialization)

Return type `tuple` of form (`bool`, `bool`)

`supports_dict`
Indicates whether the attribute can be serialized / de-serialized to `dict`.

Returns 2-member `tuple` (inbound de-serialization / outbound serialization)

Return type `tuple` of form (`bool`, `bool`)

`supports_json`
Indicates whether the attribute can be serialized / de-serialized to JSON.

Returns 2-member `tuple` (inbound de-serialization / outbound serialization)

Return type `tuple` of form (`bool`, `bool`)

`supports_yaml`
Indicates whether the attribute can be serialized / de-serialized to YAML.

Returns 2-member `tuple` (inbound de-serialization / outbound serialization)

Return type `tuple` of form (`bool`, `bool`)

4.3.2 validate_serialization_config()

validate_serialization_config(*config*)

Validate that `config` contains or can be converted to *AttributeConfiguration* objects.

Parameters **config** (iterable of *AttributeConfiguration* objects / iterable of `dict` objects corresponding to a *AttributeConfiguration* / *AttributeConfiguration* / `dict` object corresponding to a *AttributeConfiguration* / `pydantic.main.ModelMetaclass` object whose fields correspond to *AttributeConfiguration* objects) – Object or iterable of objects that represent *AttributeConfigurations*

Return type `list` of *AttributeConfiguration* objects

4.4 Flask-SQLAlchemy / Flask-SQLAthanor

4.4.1 initialize_flask_sqlathanor()

initialize_flask_sqlathanor(*db*)

Initialize **SQLAthanor** contents on a Flask-SQLAlchemy instance.

Parameters **db** (`flask_sqlalchemy.SQLAlchemy`) – The `flask_sqlalchemy.SQLAlchemy` instance.

Returns A mutated instance of `db` that replaces SQLAlchemy components and their Flask-SQLAlchemy flavors with **SQLAthanor** analogs while maintaining Flask-SQLAlchemy and SQLAlchemy functionality and interfaces.

Return type `flask_sqlalchemy.SQLAlchemy`

Raises

- **ImportError** – if called when Flask-SQLAlchemy is not installed
- **ValueError** – if `db` is not an instance of `flask_sqlalchemy.SQLAlchemy`

4.4.2 FlaskBaseModel

class FlaskBaseModel

Base class that establishes shared methods, attributes, and properties.

Designed to be supplied as a `model_class` when initializing Flask-SQLAlchemy.

See also:

For detailed explanation of functionality, please see *BaseModel*.

4.5 Automap

automap_base(*declarative_base=None*, ***kwargs*)

Produce a declarative automap base.

This function produces a new base class that is a product of the `AutomapBase` class as well as a declarative base that you supply.

If no declarative base is supplied, then the **SQLAthanor** default *`BaseModel`* will be used, to provide serialization/de-serialization support to the resulting automapped base class.

Parameters

- **declarative_base** (The declarative base model to combine with the automapped *model class* produced.) – The declarative base class that is to be combined with the `AutomapBase` class to construct the resulting automapped *model class*. To ensure that **SQLAthanor** *serialization*/*de-serialization* functionality is provided to your automapped model class, make sure that the value provided is produced by *`sqlathanor.declarative_base()`* or otherwise inherits from *`sqlathanor.declarative.BaseModel`*. If `None`, will default to *`sqlathanor.declarative.BaseModel`*.
- **kwargs** (*`keyword arguments`*) – Passed to `declarative_base()`

Returns A `AutomapBase` that can reflect your database schema structure with auto-generated declarative models that support **SQLAthanor** serialization/de-serialization.

Return type `AutomapBase`

Raises ***`SQLAlchemySupportError`*** – if called in an environment where SQLAlchemy is installed with a version less than 0.9.1 (which introduces automap support).

4.6 SQLAthanor Internals

4.6.1 RelationshipProperty

class RelationshipProperty(*argument*, *supports_json=False*, *supports_yaml=False*, *supports_dict=False*, *on_serialize=None*, *on_deserialize=None*, *display_name=None*, ***kwargs*)

Describes an object property that holds a single item or list of items that correspond to a related database table.

Public constructor is the *`sqlathanor.schema.relationship()`* function.

CHAPTER 5

Default Serialization Functions

See also:

- *Configuring Pre-processing and Post-processing*
- *Serialization Functions*
- *Default Deserialization Functions*

SQLAthanor applies default *serialization functions* to pre-process your *model attributes* before serializing them. These default functions primarily exist to convert a SQLAlchemy Data Type into a type appropriate for the format to which you are serializing your *model instance*.

The table below explains how a given *model attribute* data type is serialized using the default *serialization function* for each data type. If no information is provided, that means the value is serialized "as-is".

Data Types	*CSV*	*JSON*	*YAML*	dict
None	Empty string ''			
int Integer INTEGER INT long oracle.LONG mssql.TINYINT mysql.TINYINT mysql.MEDIUMINT SmallInteger SMALLINT BigInteger BIGINT mssql.UNIQUEIDENTIFIER				
bool Boolean BOOLEAN				
str String Text TEXT mssql.NTEXT mysql.LONGTEXT VARCHAR oracle.VARCHAR2 NVARCHAR oracle.NVARCHAR2 CHAR NCHAR Unicode UnicodeText CLOB oracle.NCLOB				
float Float FLOAT decimal.Decimal DECIMAL complex REAL Numeric NUMERIC mysql.DOUBLE oracle.DOUBLE_PRECISION postgresql.DOUBLE_PRECISION				

CHAPTER 6

Default De-serialization Functions

See also:

- *Configuring Pre-processing and Post-processing*
- *De-serialization Functions*
- *Default Serialization Functions*

SQLAthanor applies default *de-serialization functions* to process your *model attributes* before assigning their de-serialized value to your *model attribute*. These default functions primarily exist to validate that a value assigned to a given attribute is valid given that attribute's SQLAlchemy Data Type.

The table below shows the data type that is assigned to a *model attribute* by the default *de-serialization function* based on the attribute's data type.

Note: If the default de-serializer function cannot coerce the value extracted from your serialized data to either `None` or the expected data type, **SQLAthanor** will raise `ValueDeserializationError`.

Data Types	*CSV*	*JSON*	*YAML*	dict
None				
int Integer INTEGER INT mssql.TINYINT mysql.TINYINT mysql.MEDIUMINT SmallInteger SMALLINT BigInteger BIGINT	int	int	int	int
bool Boolean BOOLEAN	bool	bool	bool	bool
str String Text TEXT mssql.NTEXT mysql.LONGTEXT VARCHAR oracle.VARCHAR2 NVARCHAR oracle.NVARCHAR2 CHAR NCHAR Unicode UnicodeText CLOB oracle.NCLOB Enum ENUM SET	str	str	str	str
float Float FLOAT oracle.BINARY_FLOAT	float	float	float	float
long oracle.LONG complex REAL Numeric NUMERIC mysql.DOUBLE oracle.DOUBLE_PRECISION postgresql.DOUBLE_PRECISION mssql.ROWVERSION	validator_collection.validators.numeric()	validator_collection.validators.numeric()	validator_collection.validators.numeric()	validator_collection.validators.numeric()

CHAPTER 7

Error Reference

- *Handling Errors*
  - *Stack Traces*
  - *Errors During Serialization*
  - *Errors During De-Serialization*
- *SQLAthanor Errors*
  - *SQLAthanorError (from* `ValueError`*)*
  - *ConfigurationError (from* `SQLAthanorError`*)*
  - *SQLAlchemySupportError (from* `SQLAthanorError`*)*
  - *InvalidFormatError (from* `SQLAthanorError`*)*
  - *SerializationError (from* `SQLAthanorError`*)*
  - *ValueSerializationError (from* `SerializationError`*)*
  - *SerializableAttributeError (from* `SerializationError`*)*
  - *MaximumNestingExceededError (from* `SerializationError`*)*
  - *UnsupportedSerializationError (from* `SerializationError`*)*
  - *DeserializationError (from* `SQLAthanorError`*)*
  - *CSVStructureError (from* `DeserializationError`*)*
  - *JSONParseError (from* `DeserializationError`*)*
  - *YAMLParseError (from* `DeserializationError`*)*
  - *DeserializableAttributeError (from* `DeserializationError`*)*
  - *ValueDeserializationError (from* `DeserializationError`*)*

        - *UnsupportedValueTypeError (from* `DeserializationError`*)*
        - *UnsupportedDeserializationError (from* `DeserializationError`*)*
        - *ExtraKeyError (from* `DeserializationError`*)*
    - *SQLAthanor Warnings*
        - *SQLAthanorWarning (from* `UserWarning`*)*
        - *MaximumNestingExceededWarning (from* `SQLAthanorWarning`*)*

7.1 Handling Errors

7.1.1 Stack Traces

Because **SQLAthanor** produces exceptions which inherit from the standard library, it leverages the same API for handling stack trace information. This means that it will be handled just like a normal exception in unit test frameworks, logging solutions, and other tools that might need that information.

7.1.2 Errors During Serialization

See also:

- *Error Reference*
- *Serializing a Model Instance*
- *Default Serialization Functions*

```
from sqlathanor.errors import SerializableAttributeError, \
  UnsupportedSerializationError, MaximumNestingExceededError

# For a SQLAlchemy Model Class named "User" and a model instance named "user".

try:
  as_csv = user.to_csv()
  as_json = user.to_json()
  as_yaml = user.to_yaml()
  as_dict = user.to_dict()
except SerializableAttributeError as error:
  # Handle the situation where "User" model class does not have any attributes
  # serializable to JSON.
  pass
except UnsupportedSerializationError as error:
  # Handle the situation where one of the "User" model attributes is of a data
  # type that does not support serialization.
  pass
except MaximumNestingExceededError as error:
  # Handle a situation where "user.to_json()" received max_nesting less than
  # current_nesting.
  #
  # This situation is typically an error on the programmer's part, since
  # SQLAthanor by default avoids this kind of situation.
```

(continues on next page)

(continued from previous page)

```
  #
  # Best practice is simply to let this exception bubble up.
  raise error
```

7.1.3 Errors During De-Serialization

See also:

- *Error Reference*
- *De-serializing Data*
- *Configuring Pre-processing and Post-processing*

```
from sqlathanor.errors import DeserializableAttributeError, \
  CSVStructureError, DeserializationError, ValueDeserializationError, \
  ExtraKeysError, UnsupportedDeserializationError

# For a SQLAlchemy Model Class named "User" and a model instance named "user",
# with serialized data in "as_csv", "as_json", "as_yaml", and "as_dict" respectively.

try:
  user.update_from_csv(as_csv)
  user.update_from_json(as_json)
  user.update_from_yaml(as_yaml)
  user.update_from_dict(as_dict)

  new_user = User.new_from_csv(as_csv)
  new_user = User.new_from_json(as_json)
  new_user = User.new_from_yaml(as_yaml)
  new_user = User.new_from_dict(as_dict)
except DeserializableAttributeError as error:
  # Handle the situation where "User" model class does not have any attributes
  # de-serializable from the given format (CSV, JSON, YAML, or dict).
  pass
except DeserializationError as error:
  # Handle the situation where the serialized object ("as_csv", "as_json",
  # "as_yaml", "as_dict") cannot be parsed, for example because it is not
  # valid JSON, YAML, or dict.
  pass
except CSVStructureError as error:
  # Handle the situation where the structure of "as_csv" does not match the
  # expectation configured for the "User" model class.
  raise error
except ExtraKeysError as error:
  # Handle the situation where the serialized object ("as_json",
  # "as_yaml", "as_dict") may have unexpected keys/attributes and
  # the error_on_extra_keys argument is False.
  #
  # Applies to: *_from_json(), *_from_yaml(), and *_from_dict() methods
  pass
except ValueDeserializationError as error:
  # Handle the situation where an input value in the serialized object
  # raises an exception in the deserialization post-processing function.
  pass
except UnsupportedDeserializationError as error:
```

(continues on next page)

(continued from previous page)

```
    # Handle the situation where the de-serialization process attempts to
    # assign a value to an attribute that does not support de-serialization.
    pass
```

7.2 SQLAthanor Errors

7.2.1 SQLAthanorError (from `ValueError`)

class SQLAthanorError
Base error raised by **SQLAthanor**. Inherits from `ValueError`.

7.2.2 ConfigurationError (from `SQLAthanorError`)

class ConfigurationError
Error raised when the serialization confirmation does not match expectations.

7.2.3 SQLAlchemySupportError (from `SQLAthanorError`)

class SQLAlchemySupportError
Error raised when attempting to leverage a SQLAlchemy functionality that is not supported in your environment's version of SQLAlchemy.

7.2.4 InvalidFormatError (from `SQLAthanorError`)

class InvalidFormatError
Error raised when supplying a format that is not recognized by **SQLAthanor**.

7.2.5 SerializationError (from `SQLAthanorError`)

class SerializationError
Error raised when something went wrong during serialization.

7.2.6 ValueSerializationError (from `SerializationError`)

class ValueSerializationError
Error raised when an attribute value fails the serialization process.

7.2.7 SerializableAttributeError (from `SerializationError`)

class SerializableAttributeError
Error raised when there are no serializable attributes on a model instance.

7.2.8 MaximumNestingExceededError (from `SerializationError`)

class MaximumNestingExceededError
Error raised when the maximum permitted nesting is exceeded during serialization.

7.2.9 UnsupportedSerializationError (from `SerializationError`)

class UnsupportedSerializationError
Error raised when attempting to serialize an attribute that does not support serialization.

7.2.10 DeserializationError (from `SQLAthanorError`)

class DeserializationError
Error raised when something went wrong during de-serialization.

7.2.11 CSVStructureError (from `DeserializationError`)

class CSVStructureError
Error raised when there is a mismatch between expected columns and found columns in CSV data.

7.2.12 JSONParseError (from `DeserializationError`)

class JSONParseError
Error raised when something went wrong parsing input JSON data.

7.2.13 YAMLParseError (from `DeserializationError`)

class YAMLParseError
Error raised when something went wrong parsing input YAML data.

7.2.14 DeserializableAttributeError (from `DeserializationError`)

class DeserializableAttributeError
Error raised when there are no de-serializable attributes on a model instance or an inbound data source is empty.

7.2.15 ValueDeserializationError (from `DeserializationError`)

class ValueDeserializationError
Error raised when an attribute value fails the de-serialization process.

7.2.16 UnsupportedValueTypeError (from `DeserializationError`)

class UnsupportedValueTypeError
Error raised when a value type found in a serialized string is not supported.

7.2.17 UnsupportedDeserializationError (from `DeserializationError`)

class UnsupportedDeserializationError
Error raised when attempting to de-serialize an attribute that does not support de-serialization.

7.2.18 ExtraKeyError (from `DeserializationError`)

class ExtraKeyError
Error raised when an inbound object being de-serialized has extra (unrecognized) keys.

7.3 SQLAthanor Warnings

7.3.1 SQLAthanorWarning (from `UserWarning`)

class SQLAthanorWarning
Base warning raised by **SQLAthanor**. Inherits from `UserWarning`.

7.3.2 MaximumNestingExceededWarning (from `SQLAthanorWarning`)

class MaximumNestingExceededWarning
Warning raised when the maximum permitted nesting is exceeded during serialization.

CHAPTER 8

Contributing to SQLAthanor

Note: As a general rule of thumb, **SQLAthanor** applies **PEP 8** styling, with some important differences.

Branch	Unit Tests
latest	
v.0.8	
v.0.7	
v.0.6	
v.0.5	
v.0.4	
v.0.3	
v.0.2	
v.0.1	
develop	

What makes an API idiomatic?

One of my favorite ways of thinking about idiomatic design comes from a talk given by Luciano Ramalho at Pycon 2016[5] where he listed traits of a Pythonic API as being:

- don't force [the user] to write boilerplate code
- provide ready to use functions and objects
- don't force [the user] to subclass unless there's a *very good* reason
- include the batteries: make easy tasks easy
- are simple to use but not simplistic: make hard tasks possible
- leverage the Python data model to:
  - provide objects that behave as you expect

- avoid boilerplate through introspection (reflection) and metaprogramming.

Contents:

- *Design Philosophy*
- *Style Guide*
  - *Basic Conventions*
  - *Naming Conventions*
  - *Design Conventions*
  - *Documentation Conventions*
    - *Sphinx*
    - *Docstrings*
- *Dependencies*
- *Preparing Your Development Environment*
- *Ideas and Feature Requests*
- *Testing*
- *Submitting Pull Requests*
- *Building Documentation*
- *Contributors*
- *References*

8.1 Design Philosophy

SQLAthanor is meant to be a "beautiful" and "usable" library. That means that it should offer an idiomatic API that:

- works out of the box as intended,
- minimizes "bootstrapping" to produce meaningful output, and
- does not force users to understand how it does what it does.

In other words:

> Users should simply be able to drive the car without looking at the engine.

8.2 Style Guide

8.2.1 Basic Conventions

- Do not terminate lines with semicolons.

[5] https://www.youtube.com/watch?v=k55d3ZUF3ZQ

- Line length should have a maximum of *approximately* 90 characters. If in doubt, make a longer line or break the line between clear concepts.
- Each class should be contained in its own file.
- If a file runs longer than 2,000 lines... it should probably be refactored and split.
- All imports should occur at the top of the file.
- Do not use single-line conditions:

```
# GOOD
if x:
  do_something()

# BAD
if x: do_something()
```

- When testing if an object has a value, be sure to use `if x is None:` or `if x is not None`. Do **not** confuse this with `if x:` and `if not x:`.
- Use the `if x:` construction for testing truthiness, and `if not x:` for testing falsiness. This is **different** from testing:
  - `if x is True:`
  - `if x is False:`
  - `if x is None:`
- As of right now, because we feel that it negatively impacts readability and is less-widely used in the community, we are **not** using type annotations.

8.2.2 Naming Conventions

- `variable_name` and not `variableName` or `VariableName`. Should be a noun that describes what information is contained in the variable. If a `bool`, preface with `is_` or `has_` or similar question-word that can be answered with a yes-or-no.
- `function_name` and not `function_name` or `functionName`. Should be an imperative that describes what the function does (e.g. `get_next_page`).
- `CONSTANT_NAME` and not `constant_name` or `ConstantName`.
- `ClassName` and not `class_name` or `Class_Name`.

8.2.3 Design Conventions

- Functions at the module level can only be aware of objects either at a higher scope or singletons (which effectively have a higher scope).
- Functions and methods can use **one** positional argument (other than `self` or `cls`) without a default value. Any other arguments must be keyword arguments with default value given.

```
def do_some_function(argument):
  # rest of function...

def do_some_function(first_arg,
                     second_arg = None,
```

(continues on next page)

(continued from previous page)

```
                    third_arg = True):
  # rest of function ...
```

- Functions and methods that accept values should start by validating their input, throwing exceptions as appropriate.
- When defining a class, define all attributes in `__init__`.
- When defining a class, start by defining its attributes and methods as private using a single-underscore prefix. Then, only once they're implemented, decide if they should be public.
- Don't be afraid of the private attribute/public property/public setter pattern:

```
class SomeClass(object):
  def __init__(*args, **kwargs):
    self._private_attribute = None

  @property
  def private_attribute(self):
    # custom logic which  may override the default return

    return self._private_attribute

  @setter.private_attribute
  def private_attribute(self, value):
    # custom logic that creates modified_value

    self._private_attribute = modified_value
```

- Separate a function or method's final (or default) `return` from the rest of the code with a blank line (except for single-line functions/methods).

8.2.4 Documentation Conventions

We are very big believers in documentation (maybe you can tell). To document **SQLAthanor** we rely on several tools:

Sphinx[1]

Sphinx[1] is used to organize the library's documentation into this lovely readable format (which is also published to ReadTheDocs[2]). This documentation is written in reStructuredText[3] files which are stored in `<project>/docs`.

Tip: As a general rule of thumb, we try to apply the ReadTheDocs[2] own Documentation Style Guide[4] to our RST documentation.

Hint: To build the HTML documentation locally:

1. In a terminal, navigate to `<project>/docs`.
2. Execute `make html`.

---

[1] http://sphinx-doc.org
[2] https://readthedocs.org
[3] http://www.sphinx-doc.org/en/stable/rest.html
[4] http://documentation-style-guide-sphinx.readthedocs.io/en/latest/style-guide.html

When built locally, the HTML output of the documentation will be available at `./docs/_build/index.html`.

Docstrings

- Docstrings are used to document the actual source code itself. When writing docstrings we adhere to the conventions outlined in **PEP 257**.

8.3 Dependencies

Python 3.x

Python 2.x

- SQLAlchemy v.0.9 or higher
- PyYAML v3.10 or higher
- simplejson v3.0 or higher
- Validator-Collection v1.4.0 or higher
- SQLAlchemy v.0.9 or higher
- PyYAML v3.10 or higher
- simplejson v3.0 or higher
- Validator-Collection v1.4.0 or higher

8.4 Preparing Your Development Environment

In order to prepare your local development environment, you should:

1. Fork the Git repository.
2. Clone your forked repository.
3. Set up a virtual environment (optional).
4. Install dependencies:

```
sqlathanor/ $ pip install -r requirements.txt
```

And you should be good to go!

8.5 Ideas and Feature Requests

Check for open issues or create a new issue to start a discussion around a bug or feature idea.

8.6 Testing

If you've added a new feature, we recommend you:

- create local unit tests to verify that your feature works as expected, and
- run local unit tests before you submit the pull request to make sure nothing else got broken by accident.

See also:

For more information about the **SQLAthanor** testing approach please see: *Testing SQLAthanor*

8.7 Submitting Pull Requests

After you have made changes that you think are ready to be included in the main library, submit a pull request on Github and one of our developers will review your changes. If they're ready (meaning they're well documented, pass unit tests, etc.) then they'll be merged back into the main repository and slated for inclusion in the next release.

8.8 Building Documentation

In order to build documentation locally, you can do so from the command line using:

```
sqlathanor/ $ cd docs
sqlathanor/docs $ make html
```

When the build process has finished, the HTML documentation will be locally available at:

```
sqlathanor/docs/_build/html/index.html
```

Note: Built documentation (the HTML) is **not** included in the project's Git repository. If you need local documentation, you'll need to build it.

8.9 Contributors

Thanks to everyone who helps make **SQLAthanor** useful:

- Chris Modzelewski (@insightindustry)
- Robert Schönthal (@digitalkaoz)
- Timmy (@timmy224)

8.10 References

CHAPTER 9

Testing SQLAthanor

Contents

- *Testing SQLAthanor*
  - *Testing Philosophy*
  - *Test Organization*
  - *Configuring & Running Tests*
    - *Installing with the Test Suite*
    - *Command-line Options*
    - *Configuration File*
    - *Running Tests*
  - *Skipping Tests*
  - *Incremental Tests*

9.1 Testing Philosophy

Note: Unit tests for **SQLAthanor** are written using pytest[1] and a comprehensive set of test automation are provided by tox[2].

There are many schools of thought when it comes to test design. When building **SQLAthanor**, we decided to focus on practicality. That means:

[1] https://docs.pytest.org/en/latest/
[2] https://tox.readthedocs.io

- **DRY is good, KISS is better.** To avoid repetition, our test suite makes extensive use of fixtures, parametrization, and decorator-driven behavior. This minimizes the number of test functions that are nearly-identical. However, there are certain elements of code that are repeated in almost all test functions, as doing so will make future readability and maintenance of the test suite easier.
- **Coverage matters...kind of.** We have documented the primary intended behavior of every function in the **SQLAthanor** library, and the most-likely failure modes that can be expected. At the time of writing, we have about 85% code coverage. Yes, yes: We know that is less than 100%. But there are edge cases which are almost impossible to bring about, based on confluences of factors in the wide world. Our goal is to test the key functionality, and as bugs are uncovered to add to the test functions as necessary.

9.2 Test Organization

Each individual test module (e.g. `test_validators.py`) corresponds to a conceptual grouping of functionality. For example:

- `test_validators.py` tests validator functions found in `sqlathanor/_validators.py`

Certain test modules are tightly coupled, as the behavior in one test module may have implications on the execution of tests in another. These test modules use a numbering convention to ensure that they are executed in their required order, so that `test_1_NAME.py` is always executed before `test_2_NAME.py`.

9.3 Configuring & Running Tests

9.3.1 Installing with the Test Suite

Installing via pip

From Local Development Environment

```
$ pip install sqlathanor[tests]
```

See also:

When you *create a local development environment*, all dependencies for running and extending the test suite are installed.

9.3.2 Command-line Options

SQLAthanor does not use any custom command-line options in its test suite.

Tip: For a full list of the CLI options, including the defaults available, try:

```
sqlathanor $ cd tests/
sqlathanor/tests/ $ pytest --help
```

9.3.3 Configuration File

Because **SQLAthanor** has a very simple test suite, we have not prepared a `pytest.ini` configuration file.

9.3.4 Running Tests

Entire Test Suite

Test Module

Test Function

```
tests/ $ pytest
```

```
tests/ $ pytest tests/test_module.py
```

```
tests/ $ pytest tests/test_module.py -k 'test_my_test_function'
```

9.4 Skipping Tests

Note: Because of the simplicity of **SQLAthanor**, the test suite does not currently support any test skipping.

9.5 Incremental Tests

Note: The **SQLAthanor** test suite does support incremental testing using, however at the moment none of the tests designed rely on this functionality.

A variety of test functions are designed to test related functionality. As a result, they are designed to execute incrementally. In order to execute tests incrementally, they need to be defined as methods within a class that you decorate with the `@pytest.mark.incremental` decorator as shown below:

```
@pytest.mark.incremental
class TestIncremental(object):
    def test_function1(self):
        pass
    def test_modification(self):
        assert 0
    def test_modification2(self):
        pass
```

This class will execute the `TestIncremental.test_function1()` test, execute and fail on the `TestIncremental.test_modification()` test, and automatically fail `TestIncremental.test_modification2()` because of the `.test_modification()` failure.

To pass state between incremental tests, add a `state` argument to their method definitions. For example:

```
@pytest.mark.incremental
class TestIncremental(object):
    def test_function(self, state):
        state.is_logged_in = True
        assert state.is_logged_in = True
    def test_modification1(self, state):
        assert state.is_logged_in is True
```

(continues on next page)

(continued from previous page)

```
        state.is_logged_in = False
        assert state.is_logged_in is False
    def test_modification2(self, state):
        assert state.is_logged_in is True
```

Given the example above, the third test (`test_modification2`) will fail because `test_modification` updated the value of `state.is_logged_in`.

Note: `state` is instantiated at the level of the entire test session (one run of the test suite). As a result, it can be affected by tests in other test modules.

CHAPTER 10

Release History

Contributors

- Chris Modzelewski (@insightindustry)
- Robert Schönthal (@digitalkaoz)
- Timmy (@timmy224)

Contents

- *Release History*
  - *Release 0.8.0*
    - *New Features*
    - *Other Changes*
  - *Release 0.7.0*
    - *Bug Fixes*
    - *Other Changes*
  - *Release 0.6.0*
    - *Bug Fixes*
    - *Other Changes*
  - *Release 0.5.1*
    - *Bug Fixes*
    - *Other Changes*

- *Release 0.5.0*
  - *New Features*
  - *Bug Fixes*
- *Release 0.4.0*
  - *Bug Fixes*
  - *New Features*
  - *Other Changes*
- *Release 0.3.1*
  - *Bug Fixes*
  - *Other Changes*
- *Release 0.3.0*
  - *New Features*
  - *Other Changes*
- *Release 0.2.2*
  - *Bugs Fixed*
  - *Other Changes*
- *Release 0.2.1*
  - *Bugs Fixed*
- *Release 0.2.0*
  - *Features Added*
- *Release 0.1.1*
- *Release 0.1.0*

10.1 Release 0.8.0

10.1.1 New Features

- #99: Added Pydantic support. This includes:
  - the ability to generate **SQLAthanor** model classes from Pydantic models (`generate_model_from_pydantic()`)
  - the ability to generate `Table` objects from Pydantic models (`Table.from_pydantic()`)
  - the ability to create `AttributeConfiguration` instances from fields in a Pydantic model (`AttributeConfiguration.from_pydantic_model()`)
  - updates to documentation to reflect new functionality

10.1.2 Other Changes

- #100: Fixed missing documentation.

10.2 Release 0.7.0

10.2.1 Bug Fixes

- Fixed a typo in the documentation (thanks, @timmy224!)

10.2.2 Other Changes

- Significant performance improvement through the introduction of a heap cache (thanks, @digitalkaoz!)
- Added contributors list.

10.3 Release 0.6.0

10.3.1 Bug Fixes

- #87: Fixed a longstanding bug that prevented one-to-one relationships from being deserialized properly.

10.3.2 Other Changes

- #89: Implemented support for `display_name` on relationships.
- Updated test matrix to maintain test compability in the Python 3.4 environment.

10.4 Release 0.5.1

10.4.1 Bug Fixes

- #80: Revised how the default deserializer functions for `datetime.timedelta` and `datetime.datetime` objects functions. When deserializing either, the default deserializer now starts by attempting to coerce the value to a `datetime.timedelta` using the Validator Collection `timedelta()` validator function, which supports the expression of amounts of time as both integers (e.g. `23 seconds`) and strings (e.g. `00:00:23`). If the object cannot be converted to a `timedelta` (if it is a complete / proper datetime with both a time and date value), the default deserializer will then revert to returning a `datetime.datetime` object.
- #82: Revised the minimum version needed for the Validator Collection library to resolve a bug in iterable serialization/deserialization.

10.4.2 Other Changes

- #84: Added Python 3.8 to test matrix.

10.5 Release 0.5.0

10.5.1 New Features

- #68: Replaced serialization to `dict` with serialization to `OrderedDict` to preserve key ordering when serializing to JSON and YAML. The interface and interaction with `OrderedDict` should be 100% consistent with the behavior of past `dict` objects - just now their order will be preserved when on Python versions before 3.7.

10.5.2 Bug Fixes

- #71: Modified default `to_str()` serializer function to coerce values to strings.
- #73: Corrected a variety of mismatches in the default serializer/deserializer functions relating to `datetime.timedelta` objects and SQLAlchemy `Interval` and `DATETIME` type objects.
- #75: Corrected a bug that may have introduced errors in applications using Python 3.7, SQLAlchemy 1.3+, and relying on `AssociationProxy` constructions in their models.
- Updated the `requirements.txt` (which does not actually indicate utilization dependencies, and instead indicates development dependencies) to upgrade a number of libraries that had recently had security vulnerabilities discovered.

10.6 Release 0.4.0

10.6.1 Bug Fixes

- #63: Fixed error handling for when SQLAlchemy returns `UnsupportedCompilationError` on certain data types.

10.6.2 New Features

- #61: Added `display_name` attribute configuration option to re-write attribute names on serialization / de-serialization.
- #62: Added support for multiple named configuration sets when using the meta configuration pattern.

10.6.3 Other Changes

- Upgraded PyYAML version in `requirements.txt`.

10.7 Release 0.3.1

10.7.1 Bug Fixes

- #58: Fixed problem where `None` values are mistakenly serialized to empty lists.
- #57: Fixed problem where `on_serialize` functions were ignored for relationships.
- #56: Fixed problem where relationships were not properly deserialized.

10.7.2 Other Changes

- #26: Added Python 3.7 to test matrix.
- Removed some unnecessary print statements.

10.8 Release 0.3.0

10.8.1 New Features

- #35: Added `BaseModel.dump_to_csv()`
- #35: Added `BaseModel.dump_to_json()`
- #35: Added `BaseModel.dump_to_yaml()`
- #35: Added `BaseModel.dump_to_dict()`
- #34: Added `BaseModel.configure_serialization()`
- #42: Added support for the programmatic generation of declarative model classes.
- #41: Added support for the programmatic generation of `Table` objects.
- #51: All `*from_<format>()` methods and functions now accept Path-like objects as inputs to load serialized data from a file.

10.8.2 Other Changes

- #43: Refactored declarative classes and functions.
- #50: Updated Validator-Collection dependency.

10.9 Release 0.2.2

10.9.1 Bugs Fixed

- #36: Fixed error in documentation (`flask_sqlathanor.initialize_flask_sqlathanor()` initially documented as `flask_sqlathanor.initialize_sqlathanor()`).

10.9.2 Other Changes

- #32: Added Code of Conduct.

10.10 Release 0.2.1

10.10.1 Bugs Fixed

- #30: Tweaked function signature for `declarative_base()` to make `cls` a keyword argument.

10.11 Release 0.2.0

10.11.1 Features Added

- #21: Added support for SQLAlchemy Automap Extension.
- #27: Added support for programmatically modifying serialization/de-serialization configuration after model definition.

10.12 Release 0.1.1

- #22: Added unit tests testing support for SQLAlchemy Declarative Reflection.
- #23: Added documentation for **SQLAthanor** usage with SQLAlchemy Declarative Reflection.
- #24: Added documentation comparing/contrasting to alternative serialization/deserialization libraries.
- Fixed project URLs in `setup.py` for display on PyPi.

10.13 Release 0.1.0

- First public release

CHAPTER 11

Glossary

Association Proxy A concept in SQLAlchemy that is used to define a *model attribute* within one *model class* that acts as a proxy (mapping to) a model attribute on a different *model class*, related by way of a *relationship*.

See also:

- **SQLAlchemy**: `association_proxy`

Athanor An alchemical furnace, sometimes called a *piger henricus* (slow henry), philosophical furnace, Furnace of Arcana, or Tower furnace. They were used by alchemists to apply uniform heat over an extended (weeks, even!) period of time and require limited maintenance / management.

Basically, these were alchemical slow cookers.

The term "athanor" is believed to derive from the Arabic *al-tannoor* ("bread oven") which Califate-era alchemical texts describe as being used for slow, uniform alchemical disgestion in talismanic alchemy.

Comma-Separated Value (CSV) A text-based data exchange format where data is represented in one row of text, with fields (columns) separated by a delimiter character (typically a comma `,` or pipe `|`).

Configuration Set A named set of attribute configurations which determine which *model class* attributes get *serialized* / *de-serialized* under given circumstances (when indicating the configuration set during the serialization / de-serialization operation).

Tip: Think of a configuration set as a set of "rules" that determine what gets processed when serializing or de-serializing a *model class*.

Note: It is possible for a single *model class* to have many different configuration sets, so long as each set has a unique name.

Declarative Configuration A way of configuring *serialization* and *de-serialization* for particular *model attributes* when defining those attributes on the *model class* using the SQLAlchemy Declarative ORM.

Tip: The Declarative Configuration approach does **not** support serialization or de-serialization for *model attributes* that are not `Column` or `relationship()`.

If you want to support serialization on *hybrid properties*, *association proxies*, or *instance attributes* please use *Meta Configuration*.

See also:

- *Configure Serialization/De-serialization* > *Declarative Configuration*
- *Quickstart* > *Declarative Configuration Pattern*

De-serialization De-Serialization - as you can probably guess - is the reverse of *serialization*. It's the process whereby data is received in one format (say a JSON string) and is converted into a Python object (a *model instance*) that you can more easily work with in your Python code.

Think of it this way: A web app written in JavaScript needs to ask your Python code to register a user. Your Python code will need to know that user's details to register the user. So how does the web app deliver that information to your Python code? It'll most typically send JSON - but your Python code will need to then de-serialize (translate) it from JSON into an object representation (your `User` object) that it can work with.

De-serialization Function A function that is called when *de-serializing* a specific value. The function accepts a single positional argument (the value to be de-serialized), does whatever it needs to do to the value, and then returns the value that will be assigned to the appropriate *model attribute*.

Typical usages include value validation and hashing/salting/encryption. **SQLAthanor** applies a set of default de-serialization functions that apply for the data types supported by SQLAlchemy and its dialects.

See also:

- *De-serialization Post-processing*

Drop-in Replacement A Python library that extends the functionality of an existing library by inheriting from (and extending or modifying) its original classes or replacing its original functions.

Hybrid Property A concept in SQLAlchemy that is used to define a *model attribute* that is not directly represented in the *model class*'s underlying database table (i.e. the hybrid property is calculated/determined on-the-fly in your Python code when referenced).

See also:

- **SQLAlchemy**: `hybrid_property`

Instance Attribute A *model attribute* that is only present within a *model instance* that is defined using Python's built-in `@property` decorator.

JavaScript Object Notation (JSON) A lightweight data-interchange format that has become the *de facto* standard for communication across internet-enabled APIs.

For a formal definition, please see the ECMA-404 Standard: JSON Data Interchange Syntax

Meta Configuration A way of configuring *serialization* and *de-serialization* using a private *model attribute* labeled `__serialization__`.

Tip: Meta configuration is used to configure serialization/de-serialization for *hybrid properties*, *association proxies*, and regular (non-hybrid) Python properties.

See also:

- *Configure Serialization/De-serialization* > *Meta Configuration*
- *Quickstart* > *Meta Configuration Pattern*

Model Attribute A property or attribute that belongs to a *model class* or *model instance*. It will typically correspond to an underlying database column, relationship (foreign key constraint), *hybrid property*, or *association proxy*.

Serialization and *De-serialization* both operate on model attributes.

Model Class A model class is a Python class that is used to instantiate *model instances*. It typically is constructed using the SQLAlchemy ORM.

A model class is composed of one or more *model attributes* which correspond to columns in an underlying SQL table. The SQLAlchemy *ORM* maps the model class to a corresponding `Table` object, which in turn describes the structure of the underlying SQL table.

Note: Throughout **SQLAthanor** we use the terms "model class" and "model" interchangably.

Model Instance A model instance is an object representation of a database record in your Python code. Technically, it is an instance of a *model class*.

It stores and exposes the record's data and (if you're using a robust *ORM* like SQLAlchemy) exposes methods to modify that data.

Note: Throughout **SQLAthanor** we use the terms "model instance" and "record" interchangably.

Object Relational Mapper (ORM) An **Object Relational Mapper** (ORM) is a software tool that makes it easier to write code that reads data from or writes data to a relational database.

Fundamentally, it maps a class in your code to the tables and columns in the underlying database so that you can work with that class, rather than worrying about how to construct multiple (often related!) records directly in SQL.

The SQLAlchemy ORM is one of the most powerful Python ORMs available, and also provides a great Declarative system that makes their super-powerful ORM incredibly easy to use.

Pickling A process of *serializing* a Python object to a binary representation. Typically performed using the pickle module from the standard Python library, or an outside pickling library like dill.

Pydantic Model A Pydantic Model is a representation of a class which contains data and some attributes that inherits from the `pydantic.BaseModel` class. This model is used by the Pydantic library to either validate the typing of data upon deserialization or to serialize data to appropriate types when needed.

By definition, each Pydantic model is self-contained (though they may inherit across models). Pydantic models are inherently reliant on Python's native typing support, relying on type hints and annotations to provide the canonical instructions against which to validate or based on which to serialize.

See also:

- *SQLAthanor and Pydantic*
- *`generate_model_from_pydantic()`*
- *`Table.from_pydantic()`*
- *`AttributeConfiguration.from_pydantic_model()`*

Relationship A connection between two database tables or their corresponding *model classes* defined using a foreign key constraint.

Serialization Serialization is a process where a Python object (like a *model instance*) is converted into a different format, typically more suited to transmission to or interpretation by some other program.

Think of it this way: You've got a virtual representation of some information in your Python code. It's an object that you can work with in your Python code. But how do you give that information to some other application (like a web app) written in JavaScript? You serialize (translate) it into a format that other language can understand.

Serialization Function A function that is called when *serializing* a specific value. The function accepts a single positional argument (the *model attribute* value to serialize), does whatever it needs to do to the value, and then returns the value that will be included in the serialized output.

Typical usages include value format conversion. **SQLAthanor** applies a set of default serialization functions that apply for the data types supported by SQLAlchemy and its dialects.

See also:

- *Serialization Pre-processing*

YAML Ain't a Markup Language (YAML) YAML is a text-based data serialization format similar in some respects to *JSON*. For more information, please see the YAML 1.2 (3rd Edition) Specification.

Note: If we're being absolutely formal, JSON is actually a subset of YAML's syntax. But that's being needlessly formal.

CHAPTER 12

SQLAthanor License

MIT License

SQLAthanor is a Python library that extends SQLAlchemy's fantastic Declarative ORM to provide easy-to-use record *serialization*/*de-serialization* with support for:

- *JSON*
- *CSV*
- *YAML*
- `dict`

The library works as a *drop-in extension* - change one line of existing code, and it should just work. Furthermore, it has been extensively tested on Python 2.7, 3.4, 3.5, 3.6, 3.7, and 3.8 using SQLAlchemy 0.9 and higher.

Contents

- *SQLAthanor*

- *Installation*
  - *Dependencies*
- *Why SQLAthanor?*
  - *Key SQLAthanor Features*
  - ***SQLAthanor** vs Alternatives*
- *Hello, World and Basic Usage*
  - *1. Import SQLAthanor*
  - *2. Declare Your Models*
  - *3. Serialize Your Model Instance*
  - *4. De-serialize a Model Instance*
- *Questions and Issues*
- *Contributing*
- *Testing*
- *License*
- *Indices and tables*

CHAPTER 13

Installation

To install **SQLAthanor**, just execute:

```
$ pip install sqlathanor
```

13.1 Dependencies

Python 3.x

Python 2.x

- SQLAlchemy v.0.9 or higher
- PyYAML v3.10 or higher
- simplejson v3.0 or higher
- Validator-Collection v1.4.0 or higher

- SQLAlchemy v.0.9 or higher
- PyYAML v3.10 or higher
- simplejson v3.0 or higher
- Validator-Collection v1.4.0 or higher

CHAPTER 14

Why SQLAthanor?

Odds are you've used SQLAlchemy before. And if you haven't, why on earth not? It is hands down the best relational database toolkit and *ORM* available for Python, and has helped me quickly write code for many APIs, software platforms, and data science projects. Just look at some of these great features.

As its name suggests, SQLAlchemy focuses on the problem of connecting your Python code to an underlying relational (SQL) database. That's a super hard problem, especially when you consider the complexity of abstraction, different SQL databases, different SQL dialects, performance optimization, etc. It ain't easy, and the SQLAlchemy team has spent years building one of the most elegant solutions out there.

What's in a name?

Who can resist a good (for certain values of good) pun?

In the time-honored "science" of alchemy, an *athanor* is a furnace that provides uniform heat over an extended period of time.

Since **SQLAthanor** extends the great SQLAlchemy library, the idea was to keep the alchemical theme going.

Bottom line: I - for one - clearly cannot resist a pun, whether good or not.

But as hard as Pythonically communicating with a database is, in the real world with microservices, serverless architectures, RESTful APIs and the like we often need to do more with the data than read or write from/to our database. In almost all of the projects I've worked on over the last two decades, I've had to:

- hand data off in some fashion (*serialize*) for another program (possibly written by someone else in another programming language) to work with, or
- accept and interpret data (*de-serialize*) received from some other program (possibly written by someone else in another programming language).

Python objects (*pickled* or not) are great, but they're rarely the best way of transmitting data over the wire, or communicating data between independent applications. Which is where formats like *JSON*, *CSV*, and *YAML* come in.

So when writing many Python APIs, I found myself writing methods to convert my SQLAlchemy records (technically, *model instances*) into JSON or creating new SQLAlchemy records based on data I received in JSON. So after writing

similar methods many times over, I figured a better approach would be to write the serialization/de-serialization code just once, and then re-use it across all of my various projects.

Which is how **SQLAthanor** came about.

It adds simple methods like *to_json()*, *new_from_csv()*, and *update_from_csv()* to your SQLAlchemy declarative models and provides powerful configuration options that give you tons of flexibility.

14.1 Key SQLAthanor Features

- **Easy to adopt**: Just tweak your existing SQLAlchemy `import` statements and you're good to go.
- With one method call, convert SQLAlchemy model instances to:
  - CSV records
  - JSON objects
  - YAML objects
  - `dict` objects (technically, `OrderedDict` objects, but they behave just like regular `dict` objects)
- With one method call, create or update SQLAlchemy model instances from:
  - `dict` or `OrderedDict` objects
  - CSV records
  - JSON objects
  - YAML objects
- Decide which serialization formats you want to support for which models.
- Decide which columns/attributes you want to include in their serialized form (and pick different columns for different formats, too).
- Default validation for de-serialized data for every SQLAlchemy data type.
- Customize the validation used when de-serializing particular columns to match your needs.
- Works with *Declarative Reflection* and the *Automap Extension*.
- Programmatically *generate Declarative Base Models from serialized data or Pydantic models*.
- Programmatically create *SQLAlchemy Table objects* from serialized data or *Pydantic models*.

14.2 SQLAthanor vs Alternatives

Since *serialization* and *de-serialization* are common problems, there are a variety of alternative ways to serialize and de-serialize your SQLAlchemy models. Obviously, I'm biased in favor of **SQLAthanor**. But it might be helpful to compare **SQLAthanor** to some commonly-used alternatives:

Rolling Your Own

Pydantic

Marshmallow

Colander

pandas

Adding your own custom serialization/de-serialization logic to your SQLAlchemy declarative models is a very viable strategy. It's what I did for years, until I got tired of repeating the same patterns over and over again, and decided to build **SQLAthanor** instead.

But of course, implementing custom serialization/de-serialization logic takes a bit of effort.

Tip: When to use it?

In practice, I find that rolling my own solution is great when it's a simple model with very limited business logic. It's a "quick-and-dirty" solution, where I'm trading rapid implementation (yay!) for less flexibility/functionality (boo!).

Considering how easy **SQLAthanor** is to configure / apply, however, I find that I never really roll my own serialization/de-serialization approach when working SQLAlchemy models any more.

Tip: Because Pydantic is growing in popularity, we have decided to integrate Pydantic support within **SQLAthanor**.

Using `generate_model_from_pydantic()`, you can now programmatically generate a **SQLAthanor** `BaseModel` from your *Pydantic models*.

This allows you to only maintain *one* representation of your data model (the Pydantic one), while still being able to use SQLAthanor's rich serialization / de-serialization configuration functionality.

Pydantic is an amazing object parsing library that leverages native Python typing to provide simple syntax and rich functionality. While I have philosophical quibbles about some of its API semantics and architectural choices, I cannot deny that it is elegant, extremely performant, and all around excellent.

Since FastAPI, one of the fastest-growing web application frameworks in the Python ecosystem is tightly coupled with Pydantic, it has gained significant ground within the community.

However, when compared to **SQLAthanor** it has a number of architectural limitations:

While Pydantic has excellent serialization and deserialization functionality to JSON, it is extremely limited with its serialization/deserialization support for other common data formats like CSV or YAML.

Second, by its design Pydantic forces you to maintain **multiple** representations of your data model. On the one hand, you will need your SQLAlchemy ORM representation, but then you will *also* need one or more Pydantic models that will be used to serialize/de-serialize your model instances.

Third, by its design, Pydantic tends to lead to significant amounts of duplicate code, maintaining similar-but-not-quite-identical versions of your data models (with one Pydantic schema for each context in which you might serialize/de-serialize your data model).

Fourth, its API semantics can get extremely complicated when trying to use it as a true serialization/de-serialization library.

While Pydantic has made efforts to integrate ORM support into its API, that functionality is relatively limited in its current form.

Tip: When to use it?

Pydantic is easy to use when building simple web applications due to its reliance on native Python typing. The need to maintain multiple representations of your data models is a trivial burden with small applications or relatively simple data models.

Pydantic is also practically required when building applications using the excellent FastAPI framework.

So given these two things, we recommend using Pydantic *in combination* with **SQLAthanor** to get the best of both words: native Python typing for validation against your Python model (via Pydantic) with rich configurable serialization/de-serialization logic (via **SQLAthanor**), all integrated into the underlying SQLAlchemy ORM.

The Marshmallow library and its Marshmallow-SQLAlchemy extension are both fantastic. However, they have one major architectural difference to **SQLAthanor** and several more minor differences:

The biggest difference is that by design, they force you to maintain *two* representations of your data model. One is your SQLAlchemy *model class*, while the other is your Marshmallow schema (which determines how your model is serialized/de-serialized). Marshmallow-SQLAlchemy specifically tries to simplify this by generating a schema based on your *model class*, but you still need to configure, manage, and maintain both representations - which as your project gets more complex, becomes non-trivial.

SQLAthanor by contrast lets you configure serialization/deserialization **in** your SQLAlchemy *model class* definition. You're only maintaining *one* data model representation in your Python code, which is a massive time/effort/risk-reduction.

Other notable differences relate to the API/syntax used to support non-*JSON* formats. I think Marshmallow uses a non-obvious approach, while with **SQLAthanor** the APIs are clean and simple. Of course, on this point, YMMV.

Tip: When to use it?

Marshmallow has one advantage over **SQLAthanor**: It can serialize/de-serialize *any* Python object, whether it is a SQLAlchemy model class or not. **SQLAthanor** only works with SQLAlchemy.

As a result, it may be worth using Marshmallow instead of **SQLAthanor** if you expect to be serializing / de-serializing a lot of non-SQLAlchemy objects.

The Colander library and the ColanderAlchemy extension are both great, but they have a similar *major* architectural difference to **SQLAthanor** as Marshmallow/Marshmallow-SQLAlchemy:

By design, they force you to maintain *two* representations of your data model. One is your SQLAlchemy *model class*, while the other is your Colander schema (which determines how your model is serialized/de-serialized). Colander-Alchemy tries to simplify this by generating a schema based on your *model class*, but you still need to configure, manage, and maintain both representations - which as your project gets more complex, becomes non-trivial.

SQLAthanor by contrast lets you configure serialization/deserialization **in** your SQLAlchemy *model class* definition. You're only maintaining *one* data model representation in your Python code, which is a massive time/effort/risk-reduction.

A second major difference is that, again by design, Colander is designed to serialize/de-serialize Python objects to a set of Python primitives. Since neither *JSON*, *CSV*, or *YAML* are Python primitives, you'll still need to serialize/de-serialize Colander's input/output to/from its final "transmissable" form. Once you've got a Python primitive, this isn't difficult - but it is an extra step.

Tip: When to use it?

Colander has one advantage over **SQLAthanor**: It can serialize/de-serialize *any* Python object, whether it is a SQLAlchemy model class or not. **SQLAthanor** only works with SQLAlchemy.

As a result, it may be worth using Colander instead of **SQLAthanor** if you expect to be serializing / de-serializing a lot of non-SQLAlchemy objects.

---

pandas is one of my favorite analytical libraries. It has a number of great methods that adopt a simple syntax, like `read_csv()` or `to_csv()` which de-serialize / serialize data to various formats (including SQL, JSON, CSV, etc.).

So at first blush, one might think: Why not just use pandas to handle serialization/de-serialization?

Well, pandas isn't really a serialization alternative to **SQLAthanor**. More properly, it is an ORM alternative to SQLAlchemy itself.

I could write (and have written) a lot on the subject, but the key difference is that pandas is a "lightweight" ORM that focuses on providing a Pythonic interface to work with the output of single SQL queries. It does not support complex relationships between tables, or support the abstracted definition of business logic that applies to an object representation of a "concept" stored in your database.

SQLAlchemy is *specifically* designed to do those things.

So you can think of pandas as being a less-abstract, "closer to bare metal" ORM - which is what you want if you want very efficient computations, on relatively "flat" (non-nested/minimally relational) data. Modification or manipulation of the data can be done by mutating your pandas `DataFrame` without *too much* maintenance burden because those mutations/modifications probably don't rely too much on complex abstract business logic.

SQLAthanor piggybacks on SQLAlchemy's business logic-focused ORM capabilities. It is designed to allow you to configure expected behavior *once* and then re-use that capability across all instances (records) of your data. And it's designed to play well with all of the other complex abstractions that SQLAlchemy supports, like *relationships*, *hybrid properties*, *reflection*, or *association proxies*.

pandas serialization/de-serialization capabilities can only be configured "at use-time" (in the method call), which leads to a higher maintenance burden. **SQLAthanor**'s serialization/de-serialization capabilities are specifically designed to be configurable when defining your data model.

Tip: When to use it?

The decision of whether to use pandas or SQLAlchemy is a complex one, but in my experience a good rule of thumb is to ask yourself whether you're going to need to apply complex business logic to your data.

The more complex the business logic is, the more likely SQLAlchemy will be a better solution. And *if* you are using SQLAlchemy, then **SQLAthanor** provides great and easy-to-use serialization/de-serialization capabilities.

CHAPTER 15

Hello, World and Basic Usage

SQLAthanor is a *drop-in replacement* for the SQLAlchemy Declarative ORM and parts of the SQLAlchemy Core.

15.1 1. Import SQLAthanor

Since **SQLAthanor** is a *drop-in replacement*, you should import it using the same elements as you would import from SQLAlchemy:

Using SQLAlchemy

Using SQLAthanor

Using Flask-SQLAlchemy

The code below is a pretty standard set of `import` statements when working with SQLAlchemy and its Declarative ORM.

They're provided for reference below, but do **not** make use of **SQLAthanor** and do **not** provide any support for *serialization* or *de-serialization*:

```
from sqlalchemy.ext.declarative import declarative_base, as_declarative
from sqlalchemy import Column, Integer, String          # ... and any other data types

# The following are optional, depending on how your data model is designed:
from sqlalchemy.orm import relationship
from sqlalchemy.ext.hybrid import hybrid_property
from sqlalchemy.ext.associationproxy import association_proxy
```

To import **SQLAthanor**, just replace the relevant SQLAlchemy imports with their **SQLAthanor** counterparts as below:

```
from sqlathanor import declarative_base, as_declarative
from sqlathanor import Column
from sqlathanor import relationship             # This import is optional, depending␣
↪on
```

(continues on next page)

(continued from previous page)

```
                                                # how your data model is designed.

from sqlalchemy import Integer, String          # ... and any other data types

# The following are optional, depending on how your data model is designed:
from sqlalchemy.ext.hybrid import hybrid_property
from sqlalchemy.ext.associationproxy import association_proxy
```

Tip: Because of its many moving parts, SQLAlchemy splits its various pieces into multiple modules and forces you to use many `import` statements.

The example above maintains this strategy to show how **SQLAthanor** is a 1:1 drop-in replacement. But obviously, you can import all of the items you need in just one `import` statement:

```
from sqlathanor import declarative_base, as_declarative, Column, relationship
```

SQLAthanor is designed to work with Flask-SQLAlchemy too! However, you need to:

1. Import the *`FlaskBaseModel`* class, and then supply it as the `model_class` argument when initializing Flask-SQLAlchemy.
2. Initialize **SQLAthanor** on your `db` instance using *`initialize_flask_sqlathanor`*.

```
from sqlathanor import FlaskBaseModel, initialize_flask_sqlathanor
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'

db = SQLAlchemy(app, model_class = FlaskBaseModel)
db = initialize_flask_sqlathanor(db)
```

And that's it! Now **SQLAthanor** serialization functionality will be supported by:

- Flask-SQLAlchemy's `db.Model`
- Flask-SQLAlchemy's `db.relationship()`
- Flask-SQLAlchemy's `db.Column`

See also:

For more information about working with Flask-SQLAlchemy, please review their detailed documentation.

As the examples provided above show, importing **SQLAthanor** is very straightforward, and you can include it in an existing codebase quickly and easily. In fact, your code should work **just as before**. Only now it will include new functionality to support serialization and de-serialization.

The table below shows how SQLAlchemy classes and functions map to their **SQLAthanor** replacements:

SQLAlchemy Component	**SQLAthanor** Analog
`declarative_base()` `from sqlalchemy.ext.declarative import declarative_base`	*`declarative_base()`* `from sqlathanor import declarative_base`
`@as_declarative` `from sqlalchemy.ext.declarative import as_declarative`	*`@as_declarative`* `from sqlathanor import as_declarative`
`Column` `from sqlalchemy import Column`	*`Column`* `from sqlathanor import Column`
`relationship()` `from sqlalchemy import relationship`	*`relationship()`* `from sqlathanor import relationship`
`ext.automap.automap_base()` `from sqlalchemy.ext.automap import automap_base`	*`automap.automap_base()`* `from sqlathanor.automap import automap_base`

15.2 2. Declare Your Models

Now that you have imported **SQLAthanor**, you can just declare your models the way you normally would, even using the exact same syntax.

But now when you define your model, you can also configure serialization and de-serialization for each attribute using two approaches:

- The *Meta Configuration approach* lets you define a single `__serialization__` attribute on your model that configures serialization/de-serialization for all of your model's columns, hybrid properties, association proxies, and properties.
- The *Declarative Configuration approach* lets you supply additional arguments to your attribute definitions that control whether and how they are serialized, de-serialized, or validated.

See also:

- *Configuring Serialization and De-serialization*
  - *Meta Configuration*
  - *Declarative Configuration*
- *Quickstart*
  - *Meta Configuration Pattern*
  - *Declarative Configuration Pattern*

Note:

> explicit is better than implicit
>
> —**PEP 20** - The Zen of Python

By default, all columns, relationships, association proxies, and hybrid properties will **not** be serialized. In order for a column, relationship, proxy, or hybrid property to be serializable to a given format or de-serializable from a given format, you will need to **explicitly** enable serialization/deserialization.

Meta Approach

Declarative Approach

```
from sqlathanor import declarative_base, Column, relationship, AttributeConfiguration

from sqlalchemy import Integer, String
from sqlalchemy.ext.hybrid import hybrid_property
from sqlalchemy.ext.associationproxy import association_proxy

BaseModel = declarative_base()

class User(BaseModel):
  __tablename__ = 'users'

  __serialization__ = [AttributeConfiguration(name = 'id',
                                              supports_csv = True,
                                              csv_sequence = 1,
                                              supports_json = True,
                                              supports_yaml = True,
                                              supports_dict = True,
                                              on_serialize = None,
                                              on_deserialize = None),
                       AttributeConfiguration(name = 'addresses',
                                              supports_json = True,
                                              supports_yaml = (True, True),
                                              supports_dict = (True, False),
                                              on_serialize = None,
                                              on_deserialize = None),
                       AttributeConfiguration(name = 'hybrid',
                                              supports_csv = True,
                                              csv_sequence = 2,
                                              supports_json = True,
                                              supports_yaml = True,
                                              supports_dict = True,
                                              on_serialize = None,
                                              on_deserialize = None)]
                       AttributeConfiguration(name = 'keywords',
                                              supports_csv = False,
                                              supports_json = True,
                                              supports_yaml = True,
                                              supports_dict = True,
                                              on_serialize = None,
                                              on_deserialize = None)]
                       AttributeConfiguration(name = 'python_property',
                                              supports_csv = (False, True),
                                              csv_sequence = 3,
                                              supports_json = (False, True),
                                              supports_yaml = (False, True),
                                              supports_dict = (False, True),
                                              on_serialize = None,
                                              on_deserialize = None)]
```

(continues on next page)

(continued from previous page)

```
  id = Column('id',
              Integer,
              primary_key = True)

  addresses = relationship('Address',
                           backref = 'user')

  _hybrid = 1

  @hybrid_property
  def hybrid(self):
      return self._hybrid

  @hybrid.setter
  def hybrid(self, value):
      self._hybrid = value

  @hybrid.expression
  def hybrid(cls):
    return False

  keywords = association_proxy('keywords', 'keyword')

  @property
  def python_property(self):
    return self._hybrid * 2
```

As you can see, we've added a `__serialization__` attribute to your standard model. The `__serialization__` attribute takes a list of *`AttributeConfiguration`* instances, where each configures the serialization and de-serialization of a *model attribute*.

```
from sqlathanor import declarative_base

BaseModel = declarative_base()

class User(BaseModel):
  __tablename__ = 'users'

  id = Column("id",
              Integer,
              primary_key = True,
              autoincrement = True,
              supports_csv = True,
              csv_sequence = 1,
              supports_json = True,
              supports_yaml = True,
              supports_dict = True,
              on_serialize = None,
              on_deserialize = None)

  name = Column("name",
                Text,
                supports_csv = True,
                csv_sequence = 2,
                supports_json = True,
                supports_yaml = True,
```

(continues on next page)

(continued from previous page)

```
                supports_dict = True,
                on_serialize = None,
                on_deserialize = None)

email = Column("email",
               Text,
               supports_csv = True,
               csv_sequence = 3,
               supports_json = True,
               supports_yaml = True,
               supports_dict = True,
               on_serialize = None,
               on_deserialize = validators.email)

 password = Column("password",
                   Text,
                   supports_csv = (True, False),
                   csv_sequence = 4,
                   supports_json = (True, False),
                   supports_yaml = (True, False),
                   supports_dict = (True, False),
                   on_serialize = None,
                   on_deserialize = my_custom_password_hash_function)
```

As you can see, we've just added some (optional) arguments to the `Column` constructor. Hopefully these configuration arguments are self-explanatory.

Both the Meta and the Declarative configuration approaches use the same API for configuring serialization and de-serialization. While there are a lot of details, in general, the configuration arguments are:

- `supports_<format>` determines whether that attribute is included when *serializing* or *de-serializing* the object to the `<format>` indicated.

  Tip: If you give these options one value, it will either enable (`True`) or disable (`False`) both serialization and de-serialization, respectively.

  But you can also supply a `tuple` with two values, where the first value controls whether the attribute supports the format when inbound (de-serialization) or whether it supports the format when outbound (serialization).

  In the example above, the `password` attribute will **not** be included when serializing the object (outbound). But it *will* be expected / supported when de-serializing the object (inbound).

- `on_serialize` indicates the function or functions that are used to prepare an attribute for serialization. This can either be a single function (that applies to all serialization formats) or a `dict` where each key corresponds to a format and its value is the function to use when serializing to that format.

  Tip: If `on_serialize` is left as `None`, then **SQLAthanor** will apply a default `on_serialize` function based on the attribute's data type.

- `on_deserialize` indicates the function or functions that are used to validate or pre-process an attribute when de-serializing. This can either be a single function (that applies to all formats) or a `dict` where each key corresponds to a format and its value is the function to use when de-serializing from that format.

  Tip: If `on_deserialize` is left as `None`, then **SQLAthanor** will apply a default `on_deserialize`

function based on the attribute's data type.

15.3 3. Serialize Your Model Instance

See also:

- *Serialization Reference*:
- *to_csv()*
- *to_json()*
- to_yaml()
- *to_dict()*

So now let's say you have a *model instance* and want to serialize it. It's super easy:

JSON

CSV

YAML

dict

```
# Get user with id == 123 from the database
user = User.query.get(123)

# Serialize the user record to a JSON string.
serialized_version = user.to_json()
```

```
# Get user with id == 123 from the database
user = User.query.get(123)

# Serialize the user record to a CSV string.
serialized_version = user.to_csv()
```

```
# Get user with id == 123 from the database
user = User.query.get(123)

# Serialize the user record to a YAML string.
serialized_version = user.to_yaml()
```

```
# Get user with id == 123 from the database
user = User.query.get(123)

# Serialize the user record to a Python dict.
serialized_version = user.to_dict()
```

That's it! Of course, the serialization methods all support a variety of other (*optional!*) options to fine-tune their behavior (CSV formatting, relationship nesting, etc.).

15.4 4. De-serialize a Model Instance

See also:

- *De-serialization Reference*:
- Create a new *model instance*:
  - *new_from_csv()*
  - *new_from_json()*
  - *new_from_yaml()*
  - *new_from_dict()*
- Update an existing model instance:
  - *update_from_csv()*
  - *update_from_json()*
  - *update_from_yaml()*
  - *update_from_dict()*

Now let's say you receive a `User` object in serialized form and want to create a proper Python `User` object. That's easy, too:

JSON

CSV

YAML

dict

```
# EXAMPLE 1: Create a new User from a JSON string called "deserialized_object".
user = User.new_from_json(deserialized_object)

# EXAMPLE 2: Update an existing "user" instance from a JSON
# string called "deserialized_object".
user.update_from_json(updated_object)
```

```
# EXAMPLE 1: Create a new User from a CSV string called "deserialized_object".
user = User.new_from_csv(deserialized_object)

# EXAMPLE 2: Update an existing "user" instance from a CSV
# string called "deserialized_object".
user.update_from_csv(updated_object)
```

```
# EXAMPLE 1: Create a new User from a YAML string called "deserialized_object".
user = User.new_from_json(deserialized_object)

# EXAMPLE 2: Update an existing "user" instance from a YAML
# string called "deserialized_object".
user.update_from_yaml(updated_object)
```

```
# EXAMPLE 1: Create a new User from a dict called "deserialized_object".
user = User.new_from_dict(deserialized_object)

# EXAMPLE 2: Update an existing "user" instance from a dict called
# "deserialized_object".
user.update_from_dict(updated_object)
```

That's it! Of course, all the de-serialization functions have additional options to fine-tune their behavior as needed. But that's it.

CHAPTER 16

Questions and Issues

You can ask questions and report issues on the project's Github Issues Page

CHAPTER 17

Contributing

We welcome contributions and pull requests! For more information, please see the *Contributor Guide*. And thanks to all those who've already contributed:

- Chris Modzelewski (@insightindustry)
- Robert Schönthal (@digitalkaoz)
- Timmy (@timmy224)

CHAPTER 18

Testing

We use TravisCI for our build automation and ReadTheDocs for our documentation.

Detailed information about our test suite and how to run tests locally can be found in our *Testing Reference*.

CHAPTER 19

License

SQLAthanor is made available under an *MIT License*.

CHAPTER 20

Indices and tables

- genindex
- modindex
- search

Python Module Index

s

sqlathanor.attributes, 118
sqlathanor.automap, 126
sqlathanor.declarative, 74
sqlathanor.errors, 133
sqlathanor.flask_sqlathanor, 126
sqlathanor.schema, 108

t

tests, 145

Index

Symbols

__init__() (AttributeConfiguration method), 118
__init__() (Column method), 108
__init__() (Table method), 111

A

as_declarative() (in module sqlathanor.declarative), 99
Association Proxy, **157**
Athanor, **157**
AttributeConfiguration (class in sqlathanor.attributes), 118
automap_base() (in module sqlathanor.automap), 127

B

BaseModel (class in sqlathanor.declarative), 74

C

clear_serialization_cache() (sqlathanor.declarative.BaseModel class method), 75
Column (class in sqlathanor.schema), 108
Comma-Separated Value (CSV), **157**
Configuration Set, **157**
ConfigurationError (class in sqlathanor.errors), 136
configure_serialization() (sqlathanor.declarative.BaseModel class method), 75
csv_sequence (AttributeConfiguration attribute), 124
CSVStructureError (class in sqlathanor.errors), 137

D

De-serialization, **158**
De-serialization Function, **158**
Declarative Configuration, **157**
declarative_base() (in module sqlathanor.declarative), 99
DeserializableAttributeError (class in sqlathanor.errors), 138
DeserializationError (class in sqlathanor.errors), 137
display_name (AttributeConfiguration attribute), 124
does_support_serialization() (sqlathanor.declarative.BaseModel class method), 78
Drop-in Replacement, **158**
dump_to_csv() (BaseModel method), 47, 79
dump_to_dict() (BaseModel method), 50, 80
dump_to_json() (BaseModel method), 48, 80
dump_to_yaml() (BaseModel method), 49, 81

E

ExtraKeyError (class in sqlathanor.errors), 138

F

FlaskBaseModel (class in sqlathanor.flask_sqlathanor), 126
from_attribute() (sqlathanor.attributes.AttributeConfiguration class method), 122
from_csv() (sqlathanor.schema.Table class method), 111
from_dict() (sqlathanor.schema.Table class method), 112
from_json() (sqlathanor.schema.Table class method), 114
from_pydantic() (sqlathanor.schema.Table class method), 116
from_pydantic_model() (sqlathanor.attributes.AttributeConfiguration class method), 120, 122
from_yaml() (sqlathanor.schema.Table class method), 115
fromkeys() (sqlathanor.attributes.AttributeConfiguration class method), 124

G

generate_model_from_csv() (in module sqlathanor.declarative), 100
generate_model_from_dict() (in module sqlathanor.declarative), 105
generate_model_from_json() (in module sqlathanor.declarative), 101
generate_model_from_pydantic() (in module sqlathanor.declarative), 106
generate_model_from_yaml() (in module sqlathanor.declarative), 103

get_attribute_serialization_config() (sqlathanor.declarative.BaseModel class method), 82
get_csv_column_names() (sqlathanor.declarative.BaseModel class method), 82
get_csv_data() (BaseModel method), 82
get_csv_header() (sqlathanor.declarative.BaseModel class method), 83
get_csv_serialization_config() (sqlathanor.declarative.BaseModel class method), 84
get_dict_serialization_config() (sqlathanor.declarative.BaseModel class method), 84
get_json_serialization_config() (sqlathanor.declarative.BaseModel class method), 84
get_primary_key_column_names() (sqlathanor.declarative.BaseModel class method), 85
get_primary_key_columns() (sqlathanor.declarative.BaseModel class method), 85
get_serialization_config() (sqlathanor.declarative.BaseModel class method), 85
get_yaml_serialization_config() (sqlathanor.declarative.BaseModel class method), 86

H

Hybrid Property, **158**

I

initialize_flask_sqlathanor() (in module sqlathanor.flask_sqlathanor), 126
Instance Attribute, **158**
InvalidFormatError (class in sqlathanor.errors), 136

J

JavaScript Object Notation (JSON), **158**
JSONParseError (class in sqlathanor.errors), 137

M

MaximumNestingExceededError (class in sqlathanor.errors), 137
MaximumNestingExceededWarning (class in sqlathanor.errors), 138
Meta Configuration, **158**
Model Attribute, **159**
Model Class, **159**
Model Instance, **159**

N

name (AttributeConfiguration attribute), 124
new_from_csv() (sqlathanor.BaseModel class method), 51
new_from_csv() (sqlathanor.declarative.BaseModel class method), 86
new_from_dict() (sqlathanor.BaseModel class method), 54
new_from_dict() (sqlathanor.declarative.BaseModel class method), 87
new_from_json() (sqlathanor.BaseModel class method), 52
new_from_json() (sqlathanor.declarative.BaseModel class method), 88
new_from_yaml() (sqlathanor.BaseModel class method), 53
new_from_yaml() (sqlathanor.declarative.BaseModel class method), 89

O

Object Relational Mapper (ORM), **159**
on_deserialize (AttributeConfiguration attribute), 124
on_serialize (AttributeConfiguration attribute), 125

P

Pickling, **159**
primary_key_value (BaseModel attribute), 98
Pydantic Model, **159**
pydantic_field (AttributeConfiguration attribute), 125
Python Enhancement Proposals
    PEP 20, 30, 41, 173
    PEP 257, 143
    PEP 8, 139

R

Relationship, **159**
relationship() (in module sqlathanor.schema), 110
RelationshipProperty (class in sqlathanor.schema), 127

S

SerializableAttributeError (class in sqlathanor.errors), 137
Serialization, **159**
Serialization Function, **160**
SerializationError (class in sqlathanor.errors), 136
set_attribute_serialization_config() (sqlathanor.declarative.BaseModel class method), 90
SQLAlchemySupportError (class in sqlathanor.errors), 136
sqlathanor.attributes (module), 118
sqlathanor.automap (module), 126
sqlathanor.declarative (module), 74

sqlathanor.errors (module), 133
sqlathanor.flask_sqlathanor (module), 126
sqlathanor.schema (module), 108
SQLAthanorError (class in sqlathanor.errors), 136
SQLAthanorWarning (class in sqlathanor.errors), 138
supports_csv (AttributeConfiguration attribute), 125
supports_dict (AttributeConfiguration attribute), 125
supports_json (AttributeConfiguration attribute), 125
supports_yaml (AttributeConfiguration attribute), 125

T

Table (class in sqlathanor.schema), 111
tests (module), 145
to_csv() (BaseModel method), 44, 93
to_dict() (BaseModel method), 47, 93
to_json() (BaseModel method), 45, 94
to_yaml() (BaseModel method), 46, 94

U

UnsupportedDeserializationError (class in sqlathanor.errors), 138
UnsupportedSerializationError (class in sqlathanor.errors), 137
UnsupportedValueTypeError (class in sqlathanor.errors), 138
update_from_csv() (BaseModel method), 54, 95
update_from_dict() (BaseModel method), 57, 96
update_from_json() (BaseModel method), 55, 96
update_from_yaml() (BaseModel method), 56, 97

V

validate_serialization_config() (in module sqlathanor.attributes), 126
ValueDeserializationError (class in sqlathanor.errors), 138
ValueSerializationError (class in sqlathanor.errors), 136

Y

YAML Ain't a Markup Language (YAML), **160**
YAMLParseError (class in sqlathanor.errors), 137