sqlalchemy-seeder Documentation

Release 0.3.1

Kevin CY Tang

Oct 23, 2022

Contents

Seed SQLAlchemy database with a simple data format. Supports references to other entities (and their fields) that are defined alongside it or persisted in the database.

You can install the library from PyPI:

```
pip install sqlalchemy-seeder
```

CHAPTER 1

Table of contents

1.1 Seeders

1.1.1 Basic seeder

If you only need to create an object using a simple field->value mapping you can do so with the `BasicSeeder` methods.

1.1.2 Resolving seeder

Once you want to be able to reference other entities you'll need to use a `ResolvingSeeder`. This allows for entity attributes to point to other entities (in case of relationships) or reference another entity's field (for foreign keys or attributes).

This requires the seed file to be formatted in a specific way which we will detail in the next section.

Data format

Currently supported formats:

- JSON
- YAML

The top structure is composed out of one or more **entity group** objects which define a target class and a data block. The data block in turn contains one or more **entity data** blocks which then contains field-value pairs alongside the special *!refs* key where references are defined.

The general structure is outlined here (using JSON), for some complete examples *Format examples*.

- Entity Group

Either a list of entity groups or a single entity group should form the root node of your data.

For the target class you can provide a class name (eg. *MyClass*) or a path to the class (eg. *path.to.module:MyClass*)

A single entity group:

```
{
    "target_class": "MyClass",
    "data": {}
}
```

A list of entity groups:

```
[{
    "target_class": "MyClass",
    "data": {}
},{
    "target_class": "my.module.OtherClass",
    "data": []
}]
```

- Entity Data

An entity data node defines a single entity. The *!refs* field is an optional key where you can define field values as references to other entities. These reference definitions are outlined in the next section.

A simple data block, without references, would simple be like this:

```
{
    "my_field": "my_value",
    "my_number": 123
}
```

An example with references:

```
{
    "my_field": "my_value",
    "my_number": 123,
    "!refs": {
        "my_other_class": {}
    }
}
```

In this example, the resolved reference is assigned to the attribute *my_other_class* of the defined entity.

- Reference Description

The reference description defines which entity is being referenced based on some provided criteria and a target class. Optionally, a field can be provided which corresponds to a referenced attribute of the matched entity. If no field is defined the entire object is used as a reference (eg. for relationships).

```
{
    "target_class": "OtherClass",
    "criteria": {
        "name": "My Name"
    }
}
```

Specifying a specific field:

```
{
    "target_class": "my.module.OtherClass",
    "criteria": {
        "length": 4,
        "width": 6
    },
    "field": "name"
}
```

Format examples

Examples will be built up using JSON, the final example in each section will include a YAML version. The examples use the following model classes (in a module called "example.model"):

```
# In module example.model
class Country(Base):
    __tablename__ = 'country'

    id = Column(Integer, primary_key=True)
    short = Column(String(5))
    name = Column(String(100))

    airports = relationship("Airport", back_populates="country")

class Airport(Base):
    __tablename__ = 'airport'

    id = Column(Integer, primary_key=True)
    icao = Column(String(4))
    name = Column(String(100))
    altitude = Column(Integer)

    country_id = Column(Integer, ForeignKey("country.id"), nullable=False)
    country = relationship("Country", back_populates="airports")
```

Basic examples

Let's start with defining just a single country:

```
{
    "target_class": "Country",
    "data": {
        "name": "United Kingdom",
        "short": "UK"
    }
}
```

Defining multiple countries is fairly trivial as well:

```
{
    "target_class": "example.module:Country",
    "data": [
        {
            "name": "United Kingdom",
```

(continues on next page)

(continued from previous page)

```
            "short": "UK"
        }, {
            "name": "Belgium",
            "short": "BE"
        }
    ]
}
```

You could define them separately if preferred:

```
[
    {
        "target_class": "Country",
        "data":
            {
                "name": "United Kingdom",
                "short": "UK"
            }

    },
    {
        "target_class": "Country",
        "data": {
            "name": "Belgium",
            "short": "BE"
        }
    }
]
```

In yaml these would be:

```
--- # Compact
target_class: example.module:Country
data:
- name: United Kingdom
  short: UK
- name: Belgium
  short: BE
```

```
--- # Separate
- target_class: Country
  data:
    name: United Kingdom
    short: UK
- target_class: Country
  data:
    name: Belgium
    short: BE
```

Referencing other entities

When referencing other entities you specify a number of criteria to find the matching entity. This can use any of the fields that are defined in the referenced entity class.

If there is more than one match, or no matches are found an error will be thrown.

From our example model, *Airport's require a reference to a country, either through the 'country_id* foreign key or via the *country* relationship. Here are several ways to fulfil this requirement by reference:

```
{
    "target_class": "Airport",
    "data": {
        "icao": "EGLL",
        "name": "London Heathrow",
        "!refs": {
            "country_id": {
                "target_class": "Country",
                "criteria": {
                    "short": "UK"
                },
                "field": "id"
            }
        }
    }
}
```

You can also do it via the relationship:

```
{
    "target_class": "Airport",
    "data": {
        "icao": "EGLL",
        "name": "London Heathrow",
        "!refs": {
            "country": {
                "target_class": "Country",
                "criteria": {
                    "short": "UK"
                }
            }
        }
    }
}
```

You can also reference entities that are inserted from the same file. Here the *country* relationship in the Airport entity is populated with the object that is created from this schema.

```
[
    {
        "target_class": "Country",
        "data":
            {
                "name": "United Kingdom",
                "short": "UK"
            }
    },
    {
        "target_class": "Airport",
        "data": {
            "icao": "EGLL",
            "name": "London Heathrow",
            "!refs": {
                "country": {
                    "target_class": "Country",
```

(continues on next page)

(continued from previous page)

```
                    "criteria": {
                        "short": "UK"
                    }
                }
            }
        }
    }
]
```

This same example in yaml:

```
---
- target_class: Country
  data:
    name: United Kingdom
    short: UK
- target_class: Airport,
  data:
    icao: EGLL
    name: London Heathrow
    '!refs':                    # <-- Due to the '!' symbol it has to be surrounded␣
↪in quotes.
      country:
        target_class: Country,
        criteria:
          short: UK
```

Comprehensive example

Three countries each with a single airport.

```
[
  {
    "target_class": "example.module:Country",
    "data": [
      {
        "name": "United Kingdom",
        "short": "UK"
      },
      {
        "name": "Belgium",
        "short": "BE"
      },
      {
        "name": "Netherlands",
        "short": "NL"
      }
    ]
  },
  {
    "target_class": "example.module:Airport",
    "data": [
      {
        "icao": "EGLL",
        "name": "London Heathrow",
```

(continues on next page)

(continued from previous page)

```
        "!refs": {
          "country": {
            "target_class": "Country,",
            "criteria": {
              "short": "UK"
            }
          }
        }
      },
      {
        "icao": "EBBR",
        "name": "Brussels Zaventem",
        "!refs": {
          "country_id": {
            "target_class": "Country,",
            "criteria": {
              "short": "BE"
            },
            "field": "id"
          }
        }
      },
      {
        "icao": "EHAM",
        "name": "Amsterdam Schiphol",
        "!refs": {
          "country": {
            "target_class": "Country,",
            "criteria": {
              "name": "Netherlands"
            }
          }
        }
      }
    ]
  }
]
```

```
---
- target_class: example.module:Country
  data:
  - name: United Kingdom
    short: UK
  - name: Belgium
    short: BE
  - name: Netherlands
    short: NL
- target_class: example.module:Airport
  data:
  - icao: EGLL
    name: London Heathrow
    '!refs':
        country:
                target_class: Country,
        criteria:
            short: UK
```

(continues on next page)

(continued from previous page)

```
  - icao: EBBR
    name: Brussels Zaventem
    '!refs':
        country_id:
                target_class: Country,
        criteria:
            short: BE
        field: id
    - icao: EHAM
    name: Amsterdam Schiphol
    '!refs':
        country:
                target_class: Country,
        criteria:
            name: Netherlands
```

Using the resolving seeder

A `ResolvingSeeder` needs access to a session (provided on initialization) which it uses to resolve references.

A basic usage example:

```
from sqlalchemyseeder import ResolvingSeeder
from db import Session  # Or wherever you would get your session

session = Session()
seeder = ResolvingSeeder(session)
# See API reference for more options
new_entities = seeder.load_entities_from_yaml_file("path/to/file.yaml")
session.commit()
```

1.2 API Reference

1.2.1 Basic seeder

class `sqlalchemyseeder.basic_seeder.`**`BasicSeeder`**

Directly converts objects from dictionary without any further processing.

static entity_from_dict(*entity_dict*, *entity_class*)

Created an entity using the dictionary as initializer arguments.

static entity_from_json_string(*json_string*, *entity_class*)

Extract entity from given json string.

static entity_from_yaml_string(*yaml_string*, *entity_class*)

Extract entity from given yaml string.

1.2.2 Resolving seeder

class `sqlalchemyseeder.resolving_seeder.`**`ResolvingSeeder`**(*session*)

Seeder that can resolve entities with references to other entities.

This requires the data to be formatted in a custom *Data format* to define the references.

As entities have to define their target class they must be registered so the sqlalchemyseeder can retrieve them during the seeding process. This is typically done using `register()`, `register_class()` or `register_module()` which are hoisted methods from `ClassRegistry`. If a classpath is encountered but not recognized it will be resolved before continuing.

The session passed to this sqlalchemyseeder is used to resolve references. Flushes may occur depending on the session configuration and the passed parameters. The default behaviour when loading entities is to perform flushes but not to commit.

load_entities_from_data_dict(*seed_data*, *separate_by_class=False*, *flush_on_create=True*, *commit=False*)

Create entities from the given dictionary.

By default each entity is flushed into the provided session when it is created. This is useful if you want to reference them by id in other entities.

If this behaviour is not wanted (eg. the created entities are incomplete) you can disable it by setting *flush_on_create* to False when loading entities. The provided session can still flush if it is configured with *autoflush=True*.

No commit is issued unless *commit* is set to True.

Parameters

- **seed_data** – The formatted entity dict or list. This collection can be modified by the resolver.
- **separate_by_class** – Whether the output should separate entities by class (in a dict).
- **flush_on_create** – Whether entities should be flushed once they are created.
- **commit** – Whether the session should be committed after entities are generated.

Returns List of entities or a dictionary mapping of classes to a list of entities based on *separate_by_class*.

Raises **ValidationError** – If the provided data does not conform to the expected data structure.

load_entities_from_json_file(*seed_file*, *separate_by_class=False*, *flush_on_create=True*, *commit=False*)

Convenience method to read the given file and parse it as json.

See: `load_entities_from_data_dict`

load_entities_from_json_string(*json_string*, *separate_by_class=False*, *flush_on_create=True*, *commit=False*)

Parse the given string as json.

See: `load_entities_from_data_dict`

load_entities_from_yaml_file(*seed_file*, *separate_by_class=False*, *flush_on_create=True*, *commit=False*)

Convenience method to read the given file and parse it as yaml.

See: `load_entities_from_data_dict`

load_entities_from_yaml_string(*yaml_string*, *separate_by_class=False*, *flush_on_create=True*, *commit=False*)

Parse the given string as yaml.

See: `load_entities_from_data_dict`

class sqlalchemyseeder.resolving_seeder.**ClassRegistry**

A cache of mappable classes used by `ResolvingSeeder`.

get_class_for_string(*target*)

Look for class in the cache. If it cannot be found and a full classpath is provided, it is first registered before returning.

Parameters **target** – The class name or full classpath.

Returns The class defined by the target.

Raises **AttributeError** – If there is no registered class for the given target.

register(*target*)

Register module or class defined by target.

Parameters **target** – If *target* is a class, it is registered directly using `register_class`.

If *target* is a module, it registers all mappable classes using `register_module`.

If *target* is a string, it is first resolved into either a module or a class. Which look like:

Module path: "path.to.module"

Class path: "path.to.module:MyClass"

Raises

- **ValueError** – If target string could not be parsed.
- **AttributeError** – If target string references a class that does not exist.

register_class(*cls*)

Registers the given class with its full class path in the cache.

Parameters **cls** – The class to register.

Returns The class that was passed.

Raises **ValueError** – If the class is not mappable (no associated SQLAlchemy mapper).

register_module(*module_*)

Retrieves all classes from the given module that are mappable.

Parameters **module** – The module to inspect.

Returns A set of all mappable classes that were found.

1.2.3 Exceptions

exception sqlalchemyseeder.exceptions.**AmbiguousReferenceError**

Raised when a reference matches more than one entity.

exception sqlalchemyseeder.exceptions.**EntityBuildError**

Internal error to signify that an entity cannot be built.

exception sqlalchemyseeder.exceptions.**UnresolvedReferencesError**

Raised when a reference could not be resolved during the seeding process.

CHAPTER 2

Contribute / Issues

This project is hosted on GitHub at https://github.com/RiceKab/sqlalchemy-seeder.

CHAPTER 3

License

MIT License

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index