ColanderAlchemy Documentation

Release 0.3.3.dev1

Stefano Fontanelli

April 23, 2015

Contents

ColanderAlchemy helps you to automatically generate Colander schemas based on SQLAlchemy mapped classes.

CHAPTER 1

Quick start

In order to get started with *ColanderAlchemy*, you can either use `colanderalchemy.setup_schema()` to automatically create and attach a schema to a mapped class for you, or else you can use `colanderalchemy.SQLAlchemySchemaNode` to have more control over the auto-generated schema.

The easiest way to get going is to set up an SQLAlchemy event listener. There are two ways in which to have schemas automatically generated for your models.

1. For individual SQLAlchemy models, configure the `colanderalchemy.setup_schema()` method to listen for the `mapper_configured` event for your model class:

   ```
   from sqlalchemy import event
   from colanderalchemy import setup_schema
   # MyModel is your SQLAlchemy model class
   event.listen(MyModel, 'mapper_configured', setup_schema)
   ```

   This is simplest and most efficient option if you know specifically which models require Colander schemas attached.

2. To automatically create schemas for *all* mapped models, configure the `colanderalchemy.setup_schema()` method to listen for the `mapper_configured` event for `sqlalchemy.orm.mapper`:

   ```
   from sqlalchemy import event
   from sqlalchemy.orm import mapper
   from colanderalchemy import setup_schema
   event.listen(mapper, 'mapper_configured', setup_schema)
   ```

   Consider which Colander schemas you use directly because `setup_schema` will attach schemas to all models automatically. This may result in extra overhead from generating Colander schemas that you do not use.

In both cases, this will create a Colander schema from the given SQLAlchemy model, and attach it to the given class as the attribute `__colanderalchemy__`. This event fires when the mapper for the given class is fully configured.

Note: Keep in mind that you should configure the event listener as soon as possible in your application, especially if you're using declarative definitions. Adding the above code immediately after your SQLAlchemy model class definition is advised.

By associating `ColanderAlchemy` configuration with your mapped class, its columns, and its relationships, you can tell `ColanderAlchemy` how to generate each and every part of your mapped schema - including things like titles, descriptions, preparers, validators, widgets, and more. See *Configuring within SQLAlchemy models* for more information on how to customise this process.

CHAPTER 2

Usage

Beyond the event listener methodology above, you can use `colanderalchemy.setup_schema()` manually. Simply pass it a SQLAlchemy mapped class like so:

```
from sqlalchemy import Column, Integer, String, Text
from sqlalchemy.ext.declarative import declarative_base
from colanderalchemy import setup_schema

Base = declarative_base()

class SomeClass(Base):
    __tablename__ = 'some_table'
    id = Column(Integer, primary_key=True)
    name =  Column(String(50))
    biography =  Column(Text())

setup_schema(None, SomeClass)
SomeClass.__colanderalchemy__  # A Colander schema for you to use
```

If you already have a mapped class available, you can just pass it as is - you don't need to redefine another schema.

Also, if you'd like even more control over your generated schema, then use `colanderalchemy.SQLAlchemySchemaNode` directly like so:

```
from colanderalchemy import SQLAlchemySchemaNode
from my.project import SomeClass

schema = SQLAlchemySchemaNode(SomeClass,
                              includes=['name', 'biography'],
                              excludes=['id'],
                              title='Some class')
```

Or include custom field:

```
import deform
import colander
from colanderalchemy import SQLAlchemySchemaNode
from my.project import SomeClass

typ = colander.String()
widget = deform.widget.SelectWidget(values=(('foo', 'a'),
                                            ('bar', 'b'),
                                            ('baz', 'c')))
column = colander.SchemaNode(typ,
                             name='customfield',
```

```
                                widget=widget)
schema = SQLAlchemySchemaNode(SomeClass,
                              includes=['name', column, 'biography'],
                              excludes=['id'],
                              title='Some class')
```

Note the various arguments you can pass when creating your mapped schema - you have full control over how the schema is generated and what fields are included, which are excluded, and more. See the `colanderalchemy.SQLAlchemySchemaNode` API documentation for more information. For more information you should read the section *Examples* to see how use *ColanderAlchemy*.

In either situation, you can now pass the resulting `Colander` schema to anything that needs it. For instance, this works well with `Deform` and you can read more about this later in this documentation: *Examples: using ColanderAlchemy with Deform*.

CHAPTER 3

How it works

ColanderAlchemy auto-generates *Colander* schemas following these rules:

1. The type of the schema is `colander.MappingSchema`,
2. The schema has a `colander.SchemaNode` for each `sqlalchemy.Column` in the mapped object:
   - The type of `colander.SchemaNode` is based on the type of `sqlalchemy.Column`
   - The `colander.SchemaNode` has a validator if the `sqlalchemy.Column` is an instance of either `sqlalchemy.types.Enum` or `sqlalchemy.types.String`. `Enum` is checked with `colander.OneOf` and `String` is checked with `colander.Length`
   - Customization stored in the `__colanderalchemy_config__` attribute of the SQLAlchemy type are applied.
   - `colander.SchemaNode` has `missing=colander.required` except for the when `default` is set, `nullable=True`, there's a `server_default`, or the field is an auto incrementing integer used as part of a primary key. Essentially it's required unless SQLAlchemy can derive a value for you automatically if it's missing.
   - `colander.SchemaNode` has `default=colander.null` unless there is a column default which is a static scalar value. Callable function defaults and server defaults are ignored for the purposes of generating a colander schema default value.
   - Customisations to the resulting `colander.SchemaNode` are applied, if defined as part of the `info` structure on the `sqlalchemy.Column`.
3. The schema has a `colander.SchemaNode` for each *relationship* (`sqlalchemy.orm.relationship` or those from `sqlalchemy.orm.backref`) in the mapped object:
   - The `colander.SchemaNode` has `missing=None`
   - **The type of `colander.SchemaNode` is:**
     - A `colander.Mapping` for *ManyToOne* and *OneToOne* relationships
     - A `colander.Sequence` of `colander.Mapping` for *ManyToMany* and *OneToMany* relationships
     - Customisations to the resulting `colander.SchemaNode` are applied, if defined as part of the `info` structure on the `sqlalchemy.orm.relationship`.

   For both kind of relationships, the `colander.Mapping` is built recursively by applying this same set of rules to the mapped class referenced by the relationship.
4. Customisations to the resulting `Colander` schema are applied using configuration stored in the `__colanderalchemy_config__` attribute on the class definition.

Read the section *Customization* to see how change these rules and how to customize the Colander schema returned by ColanderAlchemy.

CHAPTER 4

Contents

4.1 Examples

4.1.1 Less boilerplate

The best way to illustrate the benefit of using ColanderAlchemy is to show a comparison between the code required to represent SQLAlchemy model as a Colander schema.

Suppose you have these SQLAlchemy mapped classes:

```
from sqlalchemy import Column, Enum, ForeignKey, Integer, Unicode
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship


Base = declarative_base()

class Phone(Base):
    __tablename__ = 'phones'

    person_id = Column(Integer, ForeignKey('persons.id'),
                       primary_key=True)
    number = Column(Unicode(128), primary_key=True)
    location = Column(Enum('home', 'work'))

class Friend(Base):
    __tablename__ = 'friends'

    person_id = Column(Integer, ForeignKey('persons.id'),
                       primary_key=True)
    friend_of = Column(Integer, ForeignKey('persons.id'),
                       primary_key=True)
    rank = Column(Integer, default=0)

class Person(Base):
    __tablename__ = 'persons'

    id = Column(Integer, primary_key=True)
    name = Column(Unicode(128), nullable=False)
    surname = Column(Unicode(128), nullable=False)
    gender = Column(Enum('M', 'F'))
    age = Column(Integer)
```

```
    phones = relationship(Phone)
    friends = relationship(Friend, foreign_keys=[Friend.person_id])
```

The code you need to create the Colander schema for `Person` would be:

```
import colander


class Friend(colander.MappingSchema):
    person_id = colander.SchemaNode(colander.Int())
    friend_of = colander.SchemaNode(colander.Int())
    rank = colander.SchemaNode(colander.Int(),
                               missing=0,
                               default=0)

class Phone(colander.MappingSchema):
    person_id = colander.SchemaNode(colander.Int())
    number = colander.SchemaNode(
        colander.String(),
        validator=colander.Length(0, 128)
    )
    location = colander.SchemaNode(
        colander.String(),
        validator=colander.OneOf(['home', 'work']),
        missing=colander.null
    )

class Friends(colander.SequenceSchema):
    friends = Friend(missing=[])

class Phones(colander.SequenceSchema):
    phones = Phone(missing=[])

class Person(colander.MappingSchema):
    id = colander.SchemaNode(
        colander.Int(),
        missing=colander.drop
    )
    name = colander.SchemaNode(
        colander.String(),
        validator=colander.Length(0, 128)
    )
    surname = colander.SchemaNode(
        colander.String(),
        validator=colander.Length(0, 128)
    )
    gender = colander.SchemaNode(
        colander.String(),
        validator=colander.OneOf(['M', 'F']),
        missing=colander.null
    )
    age = colander.SchemaNode(
        colander.Int(),
        missing=colander.null
    )
    phones = Phones(missing=[])
    friends = Friends(missing=[])
```

```
person = Person()
```

By contrast, all you need to obtain the same Colander schema for the `Person` mapped class using ColanderAlchemy is simply:

```
from colanderalchemy import setup_schema

setup_schema(None, Person)
schema = Person.__colanderalchemy__
```

Or alternatively, you may do this:

```
from colanderalchemy import SQLAlchemySchemaNode

schema = SQLAlchemySchemaNode(Person)
```

As you can see, it's a lot simpler.

4.2 Examples: using ColanderAlchemy with Deform

When using ColanderAlchemy, the resulting Colander schema will reflect the configuration on the mapped class, as shown in the code below:

```
from colanderalchemy import SQLAlchemySchemaNode

from sqlalchemy import Enum, ForeignKey, Integer, Unicode
from sqlalchemy.ext.declarative import declarative_base


Base = declarative_base()

class Phone(Base):
    __tablename__ = 'phones'

    person_id = Column(Integer, ForeignKey('persons.id'),
                       primary_key=True)
    number = Column(Unicode(128), primary_key=True)
    location = Column(Enum('home', 'work'))

class Person(Base):
    __tablename__ = 'persons'

    id = Column(Integer, primary_key=True)
    name = Column(Unicode(128), nullable=False)
    surname = Column(Unicode(128), nullable=False)
    phones = relationship(Phone)

schema = SQLAlchemySchemaNode(Person)
```

The resulting schema from the code above is the same as what would be produced by constructing the following Colander schema by hand:

```
import colander


class Phone(colander.MappingSchema):
    person_id = colander.SchemaNode(colander.Int())
```

```
    number = colander.SchemaNode(
        colander.String(),
        validator=colander.Length(0, 128)
    )
    location = colander.SchemaNode(
        colander.String(),
        validator=colander.OneOf(['home', 'work']),
        missing=colander.null
    )

class Phones(colander.SequenceSchema):
    phones = Phone(missing=[])

class Person(colander.MappingSchema):
    id = colander.SchemaNode(colander.Int(),
                             missing=colander.drop)
    name = colander.SchemaNode(
        colander.String(),
        validator=colander.Length(0, 128)
    )
    surname = colander.SchemaNode(
        colander.String(),
        validator=colander.Length(0, 128)
    )
    phones = Phones(missing=[])

schema = Person()
```

Note the various configuration aspects like field length and the like will automatically be mapped. This means that getting a Deform form to use ColanderAlchemy is as simple as using any other Colander schema:

```
from colanderalchemy import SQLAlchemySchemaNode
from deform import Form

# Using Colander requires manually constructing the schema
# person = Person()

# Using ColanderAlchemy is easy!
person = SQLAlchemySchemaNode(Person)

form = Form(person, buttons=('submit',))
```

Keep in mind that if you want additional control over the resulting Colander schema and nodes produced (such as controlling a node's `title`, `description`, `widget` or more), you are able to provide appropriate keyword arguments declaratively within the SQLAlchemy model as part of the respective `info` argument to a `sqlalchemy.Column` or `sqlalchemy.orm.relationship()` declaration. For more information, see *Customization*.

4.3 Customization

4.3.1 Changing auto-generation rules

The default `Colander` schema generated using `colanderalchemy.SQLAlchemySchemaNode` follows certain rules seen in *How it works*. You can change the default behaviour of `colanderalchemy.SQLAlchemySchemaNode` by specifying the keyword arguments `includes`, `excludes`, and `overrides`.

Refer to the API for `colanderalchemy.SQLAlchemySchemaNode` and the tests to understand how they work.

This class also accepts all keyword arguments that could normally be passed to a basic `colander.SchemaNode`, such as `title`, `description`, `preparer`, and more. Read more about basic Colander customisation at http://docs.pylonsproject.org/projects/colander/en/latest/basics.html.

If the available customisation isn't sufficient, then you can subclass the following `colanderalchemy.SQLAlchemySchemaNode` methods when you need more control:

- `SQLAlchemySchemaNode.get_schema_from_column()`, which returns a `colander.SchemaNode` given a `sqlachemy.schema.Column`
- `SQLAlchemySchemaNode.get_schema_from_relationship()`, which returns a `colander.SchemaNode` given a `sqlalchemy.orm.relationship()`.

4.3.2 Configuring within SQLAlchemy models

One of the most useful aspects of ColanderAlchemy is the ability to customize the schema being built by including hints directly in your SQLAlchemy models. This means you can define just one SQLAlchemy model and have it translate to a fully-customised Colander schema, and do so purely using declarative code. Alternatively, since the resulting schema is just a `colander.SchemaNode`, you can configure it imperatively too, if you prefer.

Colander options can be specified declaratively in SQLAlchemy models using the `info` argument that you can pass to either `sqlalchemy.Column` or `sqlalchemy.orm.relationship()`. `info` accepts any and all options that `colander.SchemaNode` objects do and should be specified like so:

```
name = Column(
    'name',
    info={
        'colanderalchemy': {
            'title': 'Your name',
            'description': 'Test',
            'missing': 'Anonymous',
            # ...  add your own!
        }
    }
)
```

and you can add any number of other options into the `dict` structure as described above. So, anything you want passed to the resulting mapped `colander.SchemaNode` should be added here. This also includes arbitrary attributes like `widget`, which, whilst not part of Colander by default, is useful for a library like Deform.

Note that for a relationship, these configured attributes will only apply to the outer mapped `colander.SchemaNode`; this *outer* node being a `colander.Sequence` or `colander.Mapping`, depending on whether the SQLAlchemy relationship is x-to-many or x-to-one, respectively.

To customise the inner mapped class, use the special attribute `__colanderalchemy_config__` on the class itself and define this as a dict-like structure of options that will be passed to `colander.SchemaNode`, like so:

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

def address_validator(node, value):
   # Validate address node
   pass

class Address(Base):
    __colanderalchemy_config__ = {'title': 'An address',
```

```
                                    'description': 'Enter an address.',
                                    'validator': address_validator,
                                    'unknown': 'preserve'}
    # Other SQLAlchemy columns are defined here
```

Note that, in contrast to the other options in `__colanderalchemy_config__`, the `unknown` option is not directly passed to `colander.SchemaNode`. Instead, it is passed to the `colander.Mapping` object, which itself is passed to `colander.SchemaNode`.

It is also possible to customize the column type, this is done in the same manner as above, using the `__colanderalchemy_config__` attribute, like so:

```
from sqlalchemy import types

def email_validator(node, value):
    # Validate an e-mail address
    pass

class Email(types.TypeDecorator):

    impl = types.String

    __colanderalchemy_config__ = {'validator': email_validator}
```

It should be noted that the `default` and `missing` colander options can not be set in a SQLAlchemy type.

4.3.3 Worked example

A full worked example could be like this:

```
from sqlalchemy import Integer
from sqlalchemy import Unicode
from sqlalchemy.ext.declarative import declarative_base

import colander


Base = declarative_base()

class Person(Base):
    __tablename__ = 'person'
    # Fully customised schema node
    id = Column(sqlalchemy.Integer,
                primary_key=True,
                info={'colanderalchemy': {
                    'typ': colander.Float(),
                    'title': 'Person ID',
                    'description': 'The Person identifier.',
                    'widget': 'Empty Widget'
                }})
    # Explicitly set as a default field
    name = Column(sqlalchemy.Unicode(128),
                  nullable=False,
                  info={'colanderalchemy': {
                      'default': colander.required
                  }})
    # Explicitly excluded from resulting schema
    surname = Column(sqlalchemy.Unicode(128),
```

```
                    nullable=False,
                    info={'colanderalchemy': {'exclude': True}})
```

4.3.4 Customizable Keyword Arguments

`sqlalchemy.Column` and `sqlalchemy.orm.relationship()` can be configured with an `info` argument that ColanderAlchemy will use to customise resulting `colander.SchemaNode` objects for each attribute. The special (magic) key for attributes is `colanderalchemy`, so a `Column` definition should look like how it was mentioned above in *Configuring within SQLAlchemy models*.

This means you can customise options like:

- `typ`
- `children`
- `default`
- `missing`
- `preparer`
- `validator`
- `after_bind`
- `title`
- `description`
- `widget`

Keep in mind the above list isn't exhaustive and you should refer to the complete list of constructor arguments in the Colander API documentation for SchemaNode.

So, as an example, the value of `title` will be passed as the keyword argument `title` when instantiating the `colander.SchemaNode`. For more information about what each of the options can do, see the Colander documentation.

In addition, you can specify the following custom options to control what ColanderAlchemy itself does:

- `exclude` - Boolean value for whether to exclude a given attribute. Extremely useful for keeping a `Column` or `relationship` out of a schema. For instance, an internal field that shouldn't be made available on a Deform form.
- `children` - An iterable (such as a list or tuple) of child nodes that should be used explicitly rather than mapping the current SQLAlchemy aspect.
- `name` - Identifier for the resulting mapped Colander node.
- `typ` - An explicitly-configured Colander node type.

4.4 ColanderAlchemy API

class `colanderalchemy.`**`SQLAlchemySchemaNode`**(*class_*, *includes=None*, *excludes=None*, *overrides=None*, *unknown='ignore'*, ***kw*)

Build a Colander Schema based on the SQLAlchemy mapped class.

__init__(*class_*, *includes=None*, *excludes=None*, *overrides=None*, *unknown='ignore'*, ***kw*)
Initialise the given mapped schema according to options provided.

Arguments/Keywords

class_ An `SQLAlchemy` mapped class that you want a `Colander` schema to be generated for.

To declaratively customise `Colander SchemaNode` options, add a `__colanderalchemy_config__` attribute to your initial class declaration like so:

```
class MyModel(Base):
    __colanderalchemy_config__ = {'title': 'Custom title',
                                  'description': 'Sample'}
    ...
```

includes Iterable of attributes to include from the resulting schema. Using this option will ensure *only* the explicitly mentioned attributes are included and *all others* are excluded.

`includes` can be included in the `__colanderalchemy_config__` dict on a class to declaratively customise the resulting schema. Explicitly passing this option as an argument takes precedence over the declarative configuration.

Incompatible with `excludes`. Default: None.

excludes Iterable of attributes to exclude from the resulting schema. Using this option will ensure *only* the explicitly mentioned attributes are excluded and *all others* are included.

`excludes` can be included in the `__colanderalchemy_config__` dict on a class to declaratively customise the resulting schema. Explicitly passing this option as an argument takes precedence over the declarative configuration.

Incompatible with `includes`. Default: None.

overrides

A dict-like structure that consists of schema attributes to override imperatively. Values provides as part of `overrides` will take precendence over all others.

`overrides` can be included in the `__colanderalchemy_config__` dict on a class to declaratively customise the resulting schema. Explicitly passing this option as an argument takes precedence over the declarative configuration.

Default: None.

unknown Represents the *unknown* argument passed to `colander.Mapping`.

The `unknown` argument passed to `colander.Mapping`, which defaults to `'ignore'`, can be set by adding an `unknown` key to the `__colanderalchemy_config__` dict. For example:

```
class MyModel(Base):
    __colanderalchemy_config__ = {'title': 'Custom title',
                                  'description': 'Sample',
                                  'unknown': 'preserve'}
    ...
```

In contrast to the other options in `__colanderalchemy_config__`, the `unknown` option is not directly passed to `colander.SchemaNode`. Instead, it is passed to the `colander.Mapping` object, which itself is passed to `colander.SchemaNode`.

From Colander:

`unknown` controls the behavior of this type when an unknown key is encountered in the cstruct passed to the deserialize method of this instance.

Default: 'ignore'

****kw** Represents *all* other options able to be passed to a `colander.SchemaNode`. Keywords passed will influence the resulting mapped schema accordingly (for instance, passing `title='My Model'` means the returned schema will have its `title` attribute set accordingly.

See http://docs.pylonsproject.org/projects/colander/en/latest/basics.html for more information.

`dictify`(*obj*)

Return a dictified version of *obj* using schema information.

The schema will be used to choose what attributes will be included in the returned dict.

Thus, the return value of this function is suitable for consumption as a `Deform appstruct` and can be used to pre-populate forms in this specific use case.

Arguments/Keywords

obj An object instance to be converted to a `dict` structure. This object should conform to the given schema. For example, `obj` should be an instance of this schema's mapped class, an instance of a sub-class, or something that has the same attributes.

`objectify`(*dict_*, *context=None*)

Return an object representing `dict_` using schema information.

The schema will be used to choose how the data in the structure will be restored into SQLAlchemy model objects. The incoming `dict_` structure corresponds with one that may be created from the `dictify()` method on the same schema. Relationships and backrefs will be restored in accordance with their specific configurations.

The return value of this function will be suitable for adding into an SQLAlchemy session to be committed to a database.

Arguments/Keywords

dict_ An dictionary or similar data structure to be converted to a an SQLAlchemy object. This data structure should conform to the given schema. For example, `dict_` should be an appstruct (such as that returned from a Deform form submission), result of a call to this schema's `dictify()` method, or a matching structure with relevant keys and nesting, if applicable.

context Optional keyword argument that, if supplied, becomes the base object, with attributes and objects being applied to it.

Specify a `context` in the situation where you already have an object that exists already, such as when you have a pre-existing instance of an SQLAlchemy model. If your model is already bound to a session, then this facilitates directly updating the database – just pass in your dict or appstruct, and your existing SQLAlchemy instance as `context` and this method will update all of its attributes.

This is a perfect fit for something like a CRUD environment.

Default: `None`. Defaults to instantiating a new instance of the mapped class associated with this schema.

`get_schema_from_column`(*prop*, *overrides*)

Build and return a `colander.SchemaNode` for a given Column.

This method uses information stored in the column within the `info` that was passed to the Column on creation. This means that `Colander` options can be specified declaratively in `SQLAlchemy` models using the `info` argument that you can pass to `sqlalchemy.Column`.

Arguments/Keywords

prop A given `sqlalchemy.orm.properties.ColumnProperty` instance that represents the column being mapped.

overrides A dict-like structure that consists of schema attributes to override imperatively. Values provides as part of `overrides` will take precendence over all others.

get_schema_from_relationship(*prop*, *overrides*)

Build and return a `colander.SchemaNode` for a relationship.

The mapping process will translate one-to-many and many-to-many relationships from SQLAlchemy into a `Sequence` of `Mapping` nodes in Colander, and translate one-to-one and many-to-one relationships into a `Mapping` node in Colander. The related class involved in the relationship will be recursively mapped by ColanderAlchemy as part of this process, following the same mapping process.

This method uses information stored in the relationship within the `info` that was passed to the relationship on creation. This means that `Colander` options can be specified declaratively in `SQLAlchemy` models using the `info` argument that you can pass to `sqlalchemy.orm.relationship()`.

For all relationships, the settings will only be applied to the outer Sequence or Mapping. To customise the inner schema node, create the attribute `__colanderalchemy_config__` on the related model with a dict-like structure corresponding to the Colander options that should be customised.

Arguments/Keywords

prop A given `sqlalchemy.orm.properties.RelationshipProperty` instance that represents the relationship being mapped.

overrides A dict-like structure that consists of schema attributes to override imperatively. Values provides as part of `overrides` will take precendence over all others. Example keys include `children`, `includes`, `excludes`, `overrides`.

`colanderalchemy.`**setup_schema**(*mapper*, *class_*)

Build a Colander schema for `class_` and attach it to that class.

This method is designed to be attached to the `mapper_configured` event from SQLAlchemy.

See http://docs.sqlalchemy.org/en/latest/orm/events.html#sqlalchemy.orm.events.MapperEvents.mapper_configured for more information about event handling.

Arguments/Keywords

mapper The mapper associated with the given `class_`. This is typically passed automatically via the SQLAlchemy event handler.

May be specified as `None` if this method is being called manually.

class_ The SQLAlchemy mapped class. This class may have attributes, related mapped classes (via SQLAlchemy relationships) and the like.

CHAPTER 5

Indices and tables

- *genindex*
- *modindex*
- *search*
- *ColanderAlchemy API*

C

Symbols

C

D

G

O

S