Watson - DB

Release 2.6.3

Mar 30, 2017

Contents

SqlAlchemy integration for Watson-Framework.

CHAPTER 1

Build Status

CHAPTER 2

Dependencies

- watson-framework
- sqlalchemy
- alembic

CHAPTER 3

Installation

```
pip install watson-db
```

CHAPTER 4

Testing

Watson can be tested with py.test. Simply activate your virtualenv and run `python setup.py test`.

CHAPTER 5

Contributing

If you would like to contribute to Watson, please feel free to issue a pull request via Github with the associated tests for your code. Your name will be added to the AUTHORS file under contributors.

CHAPTER 6

Table of Contents

Usage

Configuration

Before being able to integrate SqlAlchemy with Watson, there are a few things that must be implemented first within your applications config.

1. Add the init event to your applications configuration.

```
'events': {
     events.INIT: [
         ('watson.db.listeners.Init', 1, True)
     ],
 }
```

2. Create a default configuration for a database session.

```
db = {
    'connections': {
      'default': {
          'connection_string': 'sqlite:///:memory:',
          'engine_options': {},
          'session_options': {}
       }
     }
 }
```

`engine_options` and `session_options` are optional values and can contain any kwarg values that `create_session` and `sessionmaker` from SqlAlchemy take.

A full example configuration might look like this:

```
db = {
   'connections': {
```

```
        'default': {
            'connection_string': 'sqlite:///../data/db/default.db',
            'metadata': 'app.models.Base',
            'engine_options': {
                'encoding': 'utf-8',
                'echo': False,
                'pool_recycle': 3600
            }
        },
    },
    'migrations': {
        'path': '../data/db/migrations',
        'use_twophase': False
    },
    'fixtures': {
        'path': '../data/db/fixtures',
        'data': (
            # Fixtures will be located in ../data/db/fixtures/model.json
            # and will be inserted into the 'default' database.
            ('model', None),
        )
    }
}
```

Fixtures

Fixtures are a way of inserting some initial data into a database to populate it. They are stored in basic JSON format, and can be defined as follows.

```
[
    {
        "class": "app.models.Model",
        "fields": {
            "id": 1,
            "column": "Value"
        }
    }
    // .. more records
]
```

Each fixture that is to be loaded via the *populate* command should be included in the `data` value of the `fixtures` in the format `(FIXTURE_NAME, DATABASE_CONNECTION_NAME)`. If DATABASE_CONNECTION_NAME is set to None, then the default connection will be used.

Migrations

Watson DB utilizes Alembic to handle migrations, which can be run via the command line. See the commands section of this document for more information on the individual commands.

Commands

The commands available to you are split into two namespaces, db, and db:migrate. These can be accessed via `./console.py db` and `./console.py db:migrate` respectively.

db

create

Creates the databases against the associated model metadata and connections.

dump

Prints out the SQL statements used to create the database.

populate

Inserts the data from the fixtures into the databases.

db:migrate

These commands are essentially wrappers to the Alembic command line. Additional arguments that can be specified can be found by appending –help to the command.

branches

current

downgrade

history

init

revision

stamp

upgrade

Services

Services provide a straightforward way to interact with the models in your application without having to directly call against the SqlAlchemy session itself. Each service should be defined within the configuration to use the relevant SqlAlchemy session in it's constructor.

```
dependencies = {
    'definitions': {
        'myservice': {
            'item': 'myapp.services.MyService',
            'init': ['sqlalchemy_session_default']
        },
        'mycontroller': {
            'item': 'myapp.controllers.MyController',
            'property': {
                'service': 'myservice'
            }
        }
    }
}
```

Example

Once configured, the session can be retrieved from the container via `container.get('sqlalchemy_session_[SESSION_NAME]')`.

watson.db also provides a paginator class for paginating a set of results back from SQLAlchemy. Basic usage includes:

```
# within myapp.models
from watson.db import models

class MyModel(models.Model):
    # .. columns

# within myapp.services
from watson.db import services
from myapp import models

class MyService(services.Base):
    __model__ = models.MyModel

# within myapp.controllers, assuming the MyService object has
# been injected into the controller as the `service` attribute.
from watson.db import utils
from watson.framework import controllers

class MyController(controllers.Rest):
    def GET(self):
        return {
            'paginator': utils.Pagination(self.service.query, limit=50)
        }
```

```
# within view
{% for item in paginator %}
{% endfor %}
<div class="pagination">
{% for page in paginator.iter_pages() %}
    {% if page == paginator.page %}
    <a href="#" class="current">{{ page }}</a>
    {% else %}
    <a href="#">{{ page }}</a>
    {% endif %}
{% endfor %}
</div>
```

Reference Library

watson.db.commands

class watson.db.commands.**Database**(*config*)

Database commands.

_session_or_engine(*type_*)

Retrieves all the sessions or engines from the container.

create(*drop*)

Create the relevant databases.

dump()
Print the Schema of the database.

generate_fixtures(*models*, *output_to_stdout*)
Generate fixture data in json format.

Parameters

- **models** (*string*) – A comma separated list of models to output
- **output_to_stdout** (*boolean*) – Whether or not to output to the stdout

generate_models(*connection_string=None*, *tables=None*, *outfile=None*)
Generate models from an existing database schema.

Parameters

- **connection_string** (*string*) – The database to connect to
- **tables** (*string*) – Tables to process (comma-separated, default: all)
- **outfile** (*string*) – File to write output to (default: stdout)

populate()
Add data from fixtures to the database(s).

class watson.db.commands.**Migrate**(*config*)
Alembic integration with Watson.

branches()
Show current un-spliced branch points.

current()
Display the current revision for each database.

downgrade(*sql=False*, *tag=None*, *revision='-1'*)
Revert to a previous version.

history(*rev_range*)
List changeset scripts in chronological order.

Parameters **rev_range** – Revision range in format [start]:[end]

init()
Initializes Alembic migrations for the project.

revision(*sql=False*, *autogenerate=False*, *message=None*)
Create a new revision file.

Parameters

- **sql** (*bool*) – Don't emit SQL to database - dump to standard output instead
- **autogenerate** (*bool*) – Populate revision script with andidate migration operatons, based on comparison of database to model
- **message** (*string*) – Message string to use with 'revision'

stamp(*sql=False*, *tag=None*, *revision='head'*)
'stamp' the revision table with the given revision; don't run any migrations.

Parameters

- **sql** (*bool*) – Don't emit SQL to database - dump to standard output instead
- **tag** (*string*) – Arbitrary 'tag' name - can be used by custom env.py scripts

- **revision** (*string*) – Revision identifier

upgrade(*sql=False*, *tag=None*, *revision='head'*)
Upgrade to a later version.

Parameters

- **sql** (*bool*) – Don't emit SQL to database - dump to standard output instead
- **tag** (*string*) – Arbitrary 'tag' name - can be used by custom env.py scripts
- **revision** (*string*) – Revision identifier

watson.db.contextmanagers

watson.db.contextmanagers.**transaction_scope**(*session*, *should_close=False*)
Provides a transactional scope for session calls.

See:

- http://docs.sqlalchemy.org/en/latest/orm/session.html

Example:

```
class MyController(controllers.Rest):

    def GET(self):
        with transaction_scope(self.db):
            session.add(Model())
```

watson.db.engine

watson.db.engine.**create_db**(*engine*, *model*, *drop=False*)
Creates a new database on the given engine based on the models metadata.

Parameters

- **engine** (*Engine*) – A SQLAlchemy engine object
- **model** (*object*) – The model base containing the associated metadata.

watson.db.engine.**make_engine**(***kwargs*)
Create a new engine for SqlAlchemy.

watson.db.fixtures

watson.db.listeners

class watson.db.listeners.**Complete**
Cleanups the db session at the end of each request.

class watson.db.listeners.**Init**
Bootstraps watson.db into the event system of watson.

Each session and engine can be retrieved from the container by using sqlalchemy_engine_[name of engine] and sqlalchemy_session_[name of session] respectively.

_load_default_commands(*config*)
Load default database commands and append to application config.

_validate_config(*config*, *container*)
Validates the config and sets some standard defaults.

watson.db.meta

class watson.db.meta.**_DeclarativeMeta**(*classname*, *bases*, *dict_*)
Responsible for automatically assigning a tablename to a model.

Tablenames will be pluralized.

watson.db.models

watson.db.panels

watson.db.session

watson.db.services

watson.db.utils

class watson.db.utils.**Page**(*id*)
A single page object that is returned from the paginator.

Provides the ability to automatically generate a query string.

__init__(*id*)

class watson.db.utils.**Pagination**(*query*, *page=1*, *limit=20*)
Provides simple pagination for query results.

query
Query – The SQLAlchemy query to be paginated

page
int – The page to be displayed

limit
int – The maximum number of results to be displayed on a page

total
int – The total number of results

items
list – The items returned from the query

Example:

```
# within controller
query = session.query(Model)
paginator = Pagination(query, limit=50)

# within view
{% for item in paginator %}
{% endfor %}
<div class="pagination">
{% for page in paginator.iter_pages() %}
    {% if page == paginator.page %}
```

```
    <a href="{{ page }}" class="current">{{ page.id }}</a>
    {% else %}
    <a href="{{ page }}">{{ page.id }}</a>
    {% endif %}
{% endfor %}
</div>
```

__init__(*query*, *page=1*, *limit=20*)
Initialize the paginator and set some default values.

has_next
Return whether or not there are more pages from the currently displayed page.

Returns boolean

has_previous
Return whether or not there are previous pages from the currently displayed page.

Returns boolean

iter_pages()
An iterable containing the number of pages to be displayed.

Example:

```
{% for page in paginator.iter_pages() %}{% endfor %}
```

next
Return the next page object if another page exists.

Returns Page

pages
The total amount of pages to be displayed based on the number of results and the limit being displayed.

Returns int

previous
Return the previous page object if the page exists.

Returns Page

Python Module Index

W

Index

Symbols

B

C

D

G

H

I

L

M

N

P

Q

R

S

T

U

W