Passerine Documentation

Release 1.4.0

Juti Noppornpitak

May 05, 2016

Contents

Author Juti Noppornpitak <jnopporn@shiroyuki.com>

A generic object relational mapper (ORM) and data abstraction layer (DAL) primarily designed for NoSQL databases.

You can **install Passerine** by running `pip install passerine`.

CHAPTER 1

Architecture

Passerine is primarily designed for non-relational databases. Currently, the only driver shipped with the library is MongoDB. The next one will be **Riak 2.0** (a distributed database) and **MySQL** (a relational databases).

There are a few points to highlight.

- The lazy-loading strategy and proxy objects are used to load data wherever applicable.
- The ORM uses **the Unit Of Work pattern** as used by:
  - Hibernate (Java)
  - Doctrine (PHP)
  - SQLAlchemy (Python)
- By containing a similar logic to determine whether a given entity is new or old, the following condition are used:
  - If a given entity is identified with an **object ID**, the given entity will be considered as an existing entity.
  - Otherwise, it will be a new entity.
- The object ID cannot be changed via the ORM interfaces.
- The ORM supports cascading operations on deleting, persisting, and refreshing.
- To ensure the performance, heavily rely on **public properties**, which does not have leading underscores (_) to map between class properties and document keys, except the property **id** will be converted to the key **_id**.
- The class design is heavily influenced by dependency injection.

1.1 Limitations

- **Cascading operations on persisting** force the ORM to load the data of all proxy objects but commiting changes will still be made only if there are changes.
- **Cascading operations on refreshing** force the ORM to reset the data and status of all entities, including proxy objects. However, the status of any entities marked for deletion will not be reset.

1.1.1 Common for Non-relational or Distributed Databases

- Sessions cannot merge together.
- **Cascading operations on deleting** forces the ORM to load the whole data graph which degrades the performance.

1.1.2 MongoDB Only

- For the MongoDB driver, as MongoDB does not has transaction support like MySQL, the ORM has sessions to manage the object graph within the same memory space.
- As **sessions** are not supported by MongoDB, the ORM cannot roll back in case that an exception are raisen or a writing operation is interrupted.

CHAPTER 2

API Reference

2.1 passerine.common

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at `passerine/common.py`.

Author Juti Noppornpitak

This package contains classes and functions for common use.

class `passerine.common.`**`Enigma`**
Hashlib wrapper

`hash`(**data_list*)
Make a hash out of the given `value`.

Parameters **`data_list`** (*`list of string`*) – the list of the data being hashed.

Returns the hashed data string

static `instance`()
Get a singleton instance.

Note: This class is capable to act as a singleton class by invoking this method.

class `passerine.common.`**`Finder`**
File System API Wrapper

`read`(*file_path*, *is_binary=False*)
Read a file from *file_path*.

By default, read a file normally. If *is_binary* is `True`, the method will read in binary mode.

2.2 passerine.data.base

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at `passerine/data/base.py`.

class passerine.data.base.**ResourceEntity**(*path*, *cacheable=False*)
Static resource entity representing the real static resource which is already loaded to the memory.

Parameters **path** – the path to the static resource.

Note: This is for internal use only.

content
Get the content of the entity.

2.3 passerine.data.compressor

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at passerine/data/compressor.py.

2.4 passerine.data.exception

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at passerine/data/exception.py.

2.5 passerine.data.serializer

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at passerine/data/serializer.py.

2.6 passerine.db.common

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at passerine/db/common.py.

Author Juti Noppornpitak <jnopporn@shiroyuki.com>

Stability Stable

class passerine.db.common.**ProxyCollection**(*session*, *origin*, *guide*)
Proxy Collection

This collection is extended from the built-in class `list`, designed to only load the associated data whenever is required.

Parameters

- **session** (passerine.db.session.Session) – the managed session
- **origin** (*object*) – the origin of the association
- **guide** (passerine.db.mapper.RelatingGuide) – the relational guide

---

Note: To replace with criteria and driver

---

reload()
Reload the data list

> **Warning:** This method is **not recommended** to be called directly. Use *passerine.db.session.Session.refresh()* on the owned object instead.

class passerine.db.common.**ProxyFactory**
Proxy Factory

This factory is to create a proxy object.

Parameters

- **session** (passerine.db.session.Session) – the managed session
- **id** – the object ID
- **mapping_guide** (passerine.db.mapper.RelatingGuide) – the relational guide

class passerine.db.common.**ProxyObject**(*session*, *cls*, *object_id*, *read_only*, *cascading_options*, *is_reverse_proxy*)
Proxy Collection

This class is designed to only load the entity whenever the data access is required.

Parameters

- **session** (passerine.db.session.Session) – the managed session
- **cls** (*type*) – the class to map the data
- **object_id** – the object ID
- **read_only** (*bool*) – the read-only flag
- **cascading_options** (*list or tuple*) – the cascading options
- **is_reverse_proxy** (*bool*) – the reverse proxy flag

class passerine.db.common.**PseudoObjectId**(*oid=None*)
Pseudo Object ID

This class extends from bson.objectid.ObjectId.

This is used to differentiate stored entities and new entities.

class passerine.db.common.**Serializer**(*max_depth=2*)
Object Serializer for Entity

encode(*data*, *stack_depth=0*, *convert_object_id_to_str=False*)
Encode data into dictionary and list.

Parameters

- **data** – the data to encode
- **stack_depth** – traversal depth limit
- **convert_object_id_to_str** – flag to convert object ID into string

2.7 passerine.db.driver.interface

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at `passerine/db/driver/interface.py`.

class passerine.db.driver.interface.**DialectInterface**
Dialect interface

It is used to translate a generic query into a native query.

get_alias_to_native_query_map(*query*)
Retrieve a map from alias to native query.

Parameters **passerine.db.query.Query** – the query object

Return type dict

get_iterating_constrains(*query*)
Retrieve the query constrains.

Raises **NotImplemented** – only if the interface is not overridden.

get_native_operand(*generic_operand*)
Translate a generic operand into a corresponding native operand.

Parameters **generic_operand** – a generic operand

Returns a native operand

Return type str

process_join_conditions(*alias_to_conditions_map*, *alias*, *join_config*, *parent_alias*)
Process the join conditions.

Parameters

- **alias_to_conditions_map** (*dict*) – a alias-to-conditions map
- **join_config** (*dict*) – a join config map
- **alias** (*str*) – an alias of the given join map
- **parent_alias** (*str*) – the parent alias of the given join map

Raises **NotImplemented** – only if the interface is not overridden.

process_non_join_conditions(*alias_to_conditions_map*, *definition_map*, *left*, *right*, *operand*)
Process the non-join conditions.

Parameters

- **alias_to_conditions_map** (*dict*) – a alias-to-conditions map
- **definition_map** (*dict*) – a parameter-to-value map
- **left** (passerine.db.expression.ExpressionPart) – the left expression
- **right** (passerine.db.expression.ExpressionPart) – the right expression
- **operand** – the native operand

Raises **NotImplemented** – only if the interface is not overridden.

class passerine.db.driver.interface.**DriverInterface**(*config*, *dialect*)

The abstract driver interface

Parameters

- **config** (*dict*) – the configuration used to initialize the database connection / client
- **dialect** (passerine.db.driver.interface.DialectInterface) – the corresponding dialect

client

Driver Connection / Client

collection(*name*)

Low-level Collection-class API

Returns the low-level collection-class API

Raises **NotImplemented** – only if the interface is not overridden.

config

Driver configuration

connect(*config*)

Connect the client to the server.

Raises **NotImplemented** – only if the interface is not overridden.

database_name

The name of provisioned database

db(*name*)

Low-level Database-class API

Returns the low-level database-class API

Raises **NotImplemented** – only if the interface is not overridden.

dialect

Driver dialect

disconnect()

Disconnect the client.

Raises **NotImplemented** – only if the interface is not overridden.

index_count()

Retrieve the number of indexes.

Raises **NotImplemented** – only if the interface is not overridden.

indice()

Retrieve the indice.

Raises **NotImplemented** – only if the interface is not overridden.

insert(*collection_name*, *data*)
Low-level insert function

Raises **NotImplemented** – only if the interface is not overridden.

class passerine.db.driver.interface.**QueryIteration**(*alias*, *native_query*)
Driver Query Iteration

This is a metadata class representing an iteration in complex queries.

Parameters

- **alias** (*str*) – the alias of the rewritten target
- **native_query** (*dict*) – the native query for a specific engine

---

Note: Internal use only

---

class passerine.db.driver.interface.**QuerySequence**
Driver Query Sequence

The collection represents the sequence of sub queries.

add(*iteration*)
Append the the iteration

Parameters **iteration** (passerine.db.driver.interface.QueryIteration) – the query iteration

each()
Get the sequence iterator.

2.8 passerine.db.driver.mongodriver

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at passerine/db/driver/mongodriver.py.

---

exception passerine.db.driver.mongodriver.**UnsupportedExpressionError**
MongoDB-specific Unsupported Expression Error

This is due to that the expression may be unsafe (e.g., 1 = 2) or result in unnecessary complex computation (e.g., e.mobile_phone = e.home_phone).

2.9 passerine.db.driver.registrar

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at passerine/db/driver/registrar.py.

2.10 passerine.db.driver.riakdriver

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at `passerine/db/driver/riakdriver.py`.

2.11 passerine.db.entity

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at `passerine/db/entity.py`.

Author Juti Noppornpitak <jnopporn@shiroyuki.com>

class passerine.db.entity.**BasicAssociation**(*origin*, *destination*)
Basic Association

Parameters

- **origin** (*object*) – The origin of the association
- **destination** (*object*) – The destination (endpoint) of the association

Note: This class is used automatically by the association mapper.

class passerine.db.entity.**Entity**(***attributes*)
Dynamic-attribute Basic Entity

Parameters **attributes** (*dict*) – key-value dictionary

Here is an example on how to use this class.

```
@entity
class Note(Entity): pass
```

class passerine.db.entity.**Index**(*field_map*, *unique=False*)

Parameters

- **field_map** (*dict*) – the map of field to index type
- **unique** (*bool*) – the unique flag

Unless a field is not in the map of fixed orders, the index will instruct the repository to ensure all combinations of indexes are defined whenever is necessary.

passerine.db.entity.**entity**(**args*, ***kwargs*)
Entity decorator

Parameters **collection_name** (*str*) – the name of the collection

Returns the decorated object

Return type object

passerine.db.entity.**prepare_entity_class**(*cls*, *collection_name=None*, *indexes=[]*)
Create a entity class

Parameters

- **cls** (*object*) – the document class
- **collection_name** (*str*) – the name of the corresponding collection where the default is the lowercase version of the name of the given class (cls)

The object decorated with this decorator will be automatically provided with a few additional attributes.

Attribute	Access	Description	Read	Write
id	Instance	Document Identifier	Yes	Yes, ONLY `id` is undefined.
__t3_orm_meta__	Static	Tori 3's Metadata	Yes	ONLY the property of the metadata
__session__	Instance	DB Session	Yes	Yes, but NOT recommended.

The following attributes might stay around but are deprecated as soon as the stable Tori 3.0 is released.

Attribute	Access	Description	Read	Write
__collection_name__	Static	Collection Name	Yes	Yes, but NOT recommended.
__relational_map__	Static	Relational Map	Yes	Yes, but NOT recommended.
__indexes__	Static	Indexing List	Yes	Yes, but NOT recommended.

`__session__` is used to resolve the managing rights in case of using multiple sessions simutaneously.

For example,

```
@entity
class Note(object):
    def __init__(self, content, title=''):
        self.content = content
        self.title   = title
```

where the collection name is automatically defined as "note".

Changed in version 3.0: The way Tori stores metadata objects in `__collection_name__`, `__relational_map__` and `__indexes__` are now ignored by the ORM in favour of `__t3_orm_meta__` which is an entity metadata object.

This change is made to allow easier future development.

Tip: You can define it as "notes" by replacing `@entity` with `@entity('notes')`.

2.12 passerine.db.exception

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at `passerine/db/exception.py`.

exception passerine.db.exception.**DuplicatedRelationalMapping**
Exception thrown when the property is already mapped.

exception passerine.db.exception.**EntityAlreadyRecognized**
Warning raised when the entity with either a designated ID or a designated session is provided to Repository.post

exception passerine.db.exception.**EntityNotRecognized**
Warning raised when the entity without either a designated ID or a designated session is provided to Repository.put or Repository.delete

exception passerine.db.exception.**IntegrityConstraintError**
Runtime Error raised when the given value violates a integrity constraint.

exception passerine.db.exception.**InvalidUrlError**
Invalid DB URL Error

exception passerine.db.exception.**LockedIdException**
Exception thrown when the ID is tempted to change.

exception passerine.db.exception.**MissingObjectIdException**
Exception raised when the object Id is not specified during data retrieval.

exception passerine.db.exception.**NonRefreshableEntity**
Exception thrown when the UOW attempts to refresh a non-refreshable entity

exception passerine.db.exception.**ReadOnlyProxyException**
Exception raised when the proxy is for read only.

exception passerine.db.exception.**UOWRepeatedRegistrationError**
Error thrown when the given reference is already registered as a new reference or already existed.

exception passerine.db.exception.**UOWUnknownRecordError**
Error thrown when the given reference is already registered as a new reference or already existed.

exception passerine.db.exception.**UOWUpdateError**
Error thrown when the given reference is already registered as a new reference or already existed.

exception passerine.db.exception.**UnavailableCollectionException**
Exception thrown when the collection is not available.

exception passerine.db.exception.**UnknownDriverError**
Unknown Driver Error

exception passerine.db.exception.**UnsupportedRepositoryReferenceError**
Unsupported Repository Reference Error

2.13 passerine.db.expression

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at passerine/db/expression.py.

class passerine.db.expression.**Criteria**
Expression Criteria

Support operands: =, <=, <, >, >=, in, like (SQL-like string pattern), rlike (Regular-expression pattern), indexed with (only for Riak)

class passerine.db.expression.**Expression**(*left*, *operand*, *right*)
Query Expression

Parameters

- **left** (passerine.db.expression.ExpressionPart) – the left part
- **right** (passerine.db.expression.ExpressionPart) – the right part

- **operand** (*str*) – the generic operand

class passerine.db.expression.**ExpressionPart**(*original*, *kind*, *value*, *alias*)
Query Expression

Parameters

- **original** (*str*) – the original query
- **kind** (*str*) – the type of the part
- **value** – the parameter value only for a data part
- **alias** (*str*) – the entity alias for a property part or the name of the parameter of a parameter part

class passerine.db.expression.**ExpressionSet**(*expressions*)
Representation of Analyzed Expression

exception passerine.db.expression.**InvalidExpressionError**
Generic Invalid Expression Error

2.14 passerine.db.fixture

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at passerine/db/fixture.py.

Author Juti Noppornpitak

Warning: Not in used.

class passerine.db.fixture.**Fixture**(*repository*)
Foundation of the council

Note: this must be used at most once.

Warning: this class is not tested.

set(*kind*, *fixtures*)
Define the fixtures.

Parameters

- **kind** (*unicode | str*) – a string represent the kind
- **fixtures** (*dict*) – the data dictionary keyed by the alias

```
fixture = Fixture()

fixture.set(
    'council.security.model.Provider',
    {
        'ldap': { 'name': 'ldap' }
    }
)
```

```
        fixture.set(
            'council.user.model.User', {
                'admin': { 'name': 'Juti Noppornpitak' }
            }
        )
        fixture.set(
            'council.security.model.Credential',
            {
                'shiroyuki': {
                    'login':    'admin',
                    'user':     'proxy/council.user.model.User/admin',
                    'provider': 'proxy/council.security.model.Provider/ldap'
                }
            }
        )
```

2.15 passerine.db.manager

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at `passerine/db/manager.py`.

class `passerine.db.manager.`**Manager**(*driver*)

Entity Manager

This is to manage the unit-of-work session.

Parameters **driver** (`passerine.db.driver.interface.DriverInterface`) – the driver interface

close_session(*id*)

Close the managed session

Warning: This method is designed to bypass errors when the given ID is unavailable or already closed.

driver

Driver API

Return type *passerine.db.driver.interface.DriverInterface*

get_repository(*ref_class*, *session_id=None*)

Retrieve the repository.

Parameters

- **ref_class** (*type*) – The reference class
- **session_id** – The supervised session ID

Return type *passerine.db.repository.Repository*

Note: This is designed for Imagination's factorization.

Note: With an unsupervised session, the memory usage may be higher than usual as the memory reference may not be freed as long as the reference to the returned repository continues to exist in active threads.

open_session(*id=None*, *supervised=False*)
Open a session

Parameters

- **id** – the session ID
- **supervised** (*bool*) – the flag to indicate that the opening session will be observed and supervised by the manager. This allows the session to be reused by multiple components. However, it is not **thread-safe**. It is disabled by default.

session(**args*, ***kwds*)
Open a session in the context manager.

Parameters

- **id** – the session ID
- **supervised** (*bool*) – the flag to indicate that the opening session will be observed and supervised by the manager. This allows the session to be reused by multiple components. However, it is not **thread-safe**. It is disabled by default.

Note: The end of the context will close the open session.

class passerine.db.manager.**ManagerFactory**(*urls=None*, *protocols=None*)
Manager Factory

Parameters

- **urls** (*dict*) – the alias-to-endpoint-URL map
- **protocols** (*dict*) – the protocol-to-fully-qualified-module-path map

get(*alias*)
Retrieve the database by the manager alias.

Parameters **alias** (*str*) – the alias of the manager.

Return type *passerine.db.manager.Manager*

register(*protocol*, *driver_class*)
Register the protocol to the driver class.

Parameters

- **protocol** (*str*) – the protocol string (e.g., mongodb, riak)
- **driver_class** (passerine.db.driver.interface.DriverInterface) – a DriverInterface-based class (type)

set(*alias*, *url*)
Define the database endpoint URL to a manager (identified by the given alias).

Parameters

- **alias** (*str*) – the alias of the manager.
- **url** (*str*) – the URL to the endpoint

2.16 passerine.db.mapper

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at `passerine/db/mapper.py`.

Note: The current implementation doesn't support merging or detaching a document simultaneously observed by at least two entity manager.

class passerine.db.mapper.**AssociationFactory**(*origin*, *guide*, *cascading_options*, *is_reverse_mapping*)

Association Factory

class_name

Auto-generated Association Class Name

Return type str

Note: This is a read-only property.

cls

Auto-generated Association Class

Return type type

Note: This is a read-only property.

collection_name

Auto-generated Collection Name

Return type str

Note: This is a read-only property.

destination

Destination

Return type type

origin

Origin

Return type type

class passerine.db.mapper.**AssociationType**

Association Type

AUTO_DETECT = 1

Auto detection (default, disabled and raising exception)

MANY_TO_MANY = 5

Many-to-many association mode

MANY_TO_ONE = 4
Many-to-one association mode

ONE_TO_MANY = 3
One-to-many association mode

ONE_TO_ONE = 2
One-to-one association mode

static known_type(*t*)
Check if it is a known type

Parameters **t** (*int*) – type

Returns `True` if it is a known type.

Return type bool

class passerine.db.mapper.**BasicGuide**(*target_class*, *association*)
Basic Relation Guide

This class is abstract and used with the relational map of the given entity class.

Parameters

- **target_class** (*object*) – the target class or class name (e.g., acme.entity.User)
- **association** (*int*) – the type of association

target_class
The target class

Return type type

class passerine.db.mapper.**CascadingType**
Cascading Type

DELETE = 2
Cascade on delete operation

DETACH = 4
Cascade on detach operation

Note: Supported in Tori 2.2

MERGE = 3
Cascade on merge operation

Note: Supported in Tori 2.2

PERSIST = 1
Cascade on persist operation

REFRESH = 5
Cascade on refresh operation

class passerine.db.mapper.**RelatingGuide**(*entity_class*, *target_class*, *inverted_by*, *association*, *read_only*, *cascading_options*)
Relation Guide

This class is used with the relational map of the given entity class.

Parameters

- **entity_class** (*type*) – the reference of the current class
- **mapped_by** (*str*) – the name of property of the current class
- **target_class** (*type*) – the target class or class name (e.g., acme.entity.User)
- **inverted_by** (*str*) – the name of property of the target class
- **association** (*int*) – the type of association
- **read_only** (*bool*) – the flag to indicate whether this is for read only.
- **cascading_options** (*list or tuple*) – the list of actions on cascading

passerine.db.mapper.**link**(*mapped_by=None*, *target=None*, *inverted_by=None*, *association=1*, *read_only=False*, *cascading=[]*)

Association decorator

New in version 2.1.

This is to map a property of the current class to the target class.

Parameters

- **mapped_by** (*str*) – the name of property of the current class
- **target** (*type*) – the target class or class name (e.g., acme.entity.User)
- **inverted_by** (*str*) – the name of property of the target class
- **association** (*int*) – the type of association
- **read_only** (*bool*) – the flag to indicate whether this is for read only.
- **cascading** (*list or tuple*) – the list of actions on cascading

Returns the decorated class

Return type type

Tip: If `target` is not defined, the default target will be the reference class.

---

passerine.db.mapper.**map**(*cls*, *mapped_by=None*, *target=None*, *inverted_by=None*, *association=1*, *read_only=False*, *cascading=[]*)

Map the given class property to the target class.

New in version 2.1.

Parameters

- **cls** (*type*) – the reference of the current class
- **mapped_by** (*str*) – the name of property of the current class
- **target** (*type*) – the target class or class name (e.g., acme.entity.User)
- **inverted_by** (*str*) – the name of property of the target class
- **association** (*int*) – the type of association
- **read_only** (*bool*) – the flag to indicate whether this is for read only.
- **cascading** (*list or tuple*) – the list of actions on cascading

2.17 passerine.db.metadata.entity

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at `passerine/db/metadata/entity.py`.

class `passerine.db.metadata.entity.`**`EntityMetadata`**
Entity Metadata

`cls`
Entity Class

`collection_name`
Collection / Bucket / Table Name

`index_list`
Index List

`relational_map`
Relational Map

2.18 passerine.db.metadata.helper

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at `passerine/db/metadata/helper.py`.

class `passerine.db.metadata.helper.`**`EntityMetadataHelper`**
Entity Metadata Helper

static `extract``()`
Extract the metadata of the given class

Parameters **`cls`** (*type*) – the entity class

Return type *passerine.db.metadata.entity.EntityMetadata*

static `hasMetadata``()`
Check if the given class *cls* has a metadata

Parameters **`cls`** (*type*) – the entity class

Return type bool

static `imprint`(*collection_name*, *indexes*)
Imprint the entity metadata to the class (type)

Parameters

- **`cls`** (*type*) – the entity class
- **`collection_name`** (*str*) – the name of the collection (known as table, bucket etc.)
- **`indexes`** (*list*) – the list of indexes

2.19 passerine.db.mochi

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at `passerine/db/mochi.py`.

2.19.1 Mochi Non-relational Database Library

Author Juti Noppornpitak <jnopporn@shiroyuki.com>

This library is designed to work like SQLite but be compatible with MongoDB instructions (and PyMongo interfaces).

2.20 passerine.db.query

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at `passerine/db/query.py`.

2.20.1 `passerine.db.criteria` – Query Criteria

class passerine.db.query.**Order**
Sorting Order Definition

ASC
Ascending Order

alias of `ASCENDING`

DESC
Descending Order

alias of `DESCENDING`

class passerine.db.query.**Query**(*alias*)
Criteria

Note: The current implementation does not support filtering on associated entities.

criteria
Expression Criteria

define(*variable_name=None*, *value=None*, ***definition_map*)
Define the value of one or more variables (known as parameters).

Parameters

- **variable_name** (*str*) – the name of the variable (for single assignment)
- **value** – the value of the variable (for single assignment)
- **definition_map** – the variable-to-value dictionary

This method is usually recommended be used to define multiple variables like the following example.

```
criteria.define(foo = 'foo', bar = 2)
```

However, it is designed to support the assign of a single user. For instance,

```
criteria.define('foo', 'foo').define('bar', 2)
```

expect (*statement*)
Define the condition / expectation of the main expression.

Parameters **statement** (*str*) – the conditional statement

This is a shortcut expression to define expectation of the main expression. The main expression will be defined automatically if it is undefined. For example,

```
c = Query()
c.expect('foo = 123')
```

is the same thing as

```
c = Query()
c.criteria = c.new_criteria()
c.criteria.expect('foo = 123')
```

join (*property_path*, *alias*)
Define a join path

join_map
A join map

limit (*limit*)
Define the filter limit

Parameters **limit** (*int*) – the filter limit

new_criteria ()
Get a new expression for this criteria

Return type *passerine.db.expression.Criteria*

order (*field*, *direction=<class 'ASCENDING'>*)
Define the returning order

Parameters

- **field** (*str*) – the sorting field
- **direction** – the sorting direction

start (*offset*)
Define the filter offset

Parameters **offset** (*int*) – the filter offset

2.21 passerine.db.repository

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at `passerine/db/repository.py`.

Author Juti Noppornpitak <jnopporn@shiroyuki.com>

Status Stable

class passerine.db.repository.**Repository**(*session*, *representing_class*)
Repository (Entity AbstractRepository) for Mongo DB

Parameters

- **session** (passerine.db.session.Session) – the entity manager
- **representing_class** (*type*) – the representing class

A repository may automatically attempt to create an index if *auto_index()* define the auto-index flag. Please note that the auto-index feature is only invoked when it tries to use a criteria with sorting or filtering with a certain type of conditions.

auto_index(*auto_index*)
Enable the auto-index feature

Parameters **auto_index** (*bool*) – the index flag

count(*criteria*)
Count the number of entities satisfied the given criteria

Parameters **criteria** (*passerine.db.criteria.Query*) – the search criteria

Return type int

filter(*condition={}*, *force_loading=False*)
Shortcut method for **:method:'find'**.

filter_one(*condition={}*, *force_loading=False*)
Shortcut method for **:method:'find'**.

find(*criteria=None*, *force_loading=False*)
Find entity with criteria

Parameters

- **criteria** (*passerine.db.criteria.Query*) – the search criteria
- **force_loading** (*bool*) – the flag to force loading all references behind the proxy

Returns the result based on the given criteria

Return type object or list of objects

index(*index*, *force_index=False*)
Index data

Parameters

- **index** (*list, passerine.db.entity.Index or str*) – the index
- **force_index** (*bool*) – force indexing if necessary

name
Collection name

Return type str

new(***attributes*)
Create a new document/entity

Parameters **attributes** – attribute map

Returns object

Note: This method deal with data mapping.

new_criteria(*alias='e'*)
Create a criteria

Return type passerine.db.criteria.Query

session
Session

Return type *passerine.db.session.Session*

setup_index()
Set up index for the entity based on the entity and link decorators

2.22 passerine.db.session

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at passerine/db/session.py.

class passerine.db.session.**Session**(*driver*)
Database Session

Parameters

- **database_name** – the database name
- **driver** – the driver API

apply_relational_map(*entity*)
Wire connections according to the relational map

collection(*entity_class*)
Alias to repository()

Deprecated since version 2.2.

delete(**entities*)
Delete entities

Parameters **entities** (*type of list of type*) – one or more entities

flush(**args*, ***kwargs*)
Flush all changes of the session.

See the flag from **:method:'passerine.db.uow.UnitOfWork.commit'**.

persist(**entities*)
Persist entities

Parameters **entities** (*type of list of type*) – one or more entities

query(*query*)
Query the data

Parameters **query** (passerine.db.query.Query) – the query object

Returns the list of matched entities

Return type list

refresh(**entities*)
Refresh entities

Parameters **entities** (*type of list of type*) – one or more entities

register_class(*entity_class*)
Register the entity class

Parameters **entity_class** (*type*) – the class of document/entity

Return type *passerine.db.repository.Repository*

Note: This is for internal operation only. As it seems to be just a residual from the prototype stage, the follow-up investigation in order to remove the method will be for Tori 3.1.

repositories()
Retrieve the list of collections

Return type list

repository(*reference*)
Retrieve the collection

Parameters **reference** – the entity class or entity metadata of the target repository / collection

Return type *passerine.db.repository.Repository*

2.23 passerine.db.uow

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at `passerine/db/uow.py`.

class passerine.db.uow.**DependencyNode**(*record*)
Dependency Node

This is designed to be bi-directional to maximize flexibility on traversing the graph.

class passerine.db.uow.**UnitOfWork**(*entity_manager*)
Unit of Work

This Unit of Work (UOW) is designed specifically for non-relational databases.

Note: It is the design decision to make sub-commit methods available so that when it is used with Imagination Framework, the other Imagination entity may intercept before or after actually committing data. In the other word, Imagination Framework acts as an event controller for any actions (public methods) of this class.

commit(*sync_mode=True*, *check_associations=True*)
Commit the changes.

Warning: Both flags should not be set to False.

Parameters

- **sync_mode** (*bool*) – Enable the synchronous mode. (default: True)
- **check_associations** (*bool*) – Check the associations. (default: True)

refresh(*entity*)
Refresh the entity

Note: This method

Parameters **entity** (*object*) – the target entity

register_clean(*entity*)
Register the entity with the clean bit

Parameters **entity** (*object*) – the entity to register

register_deleted(*entity*)
Register the entity with the removal bit

Parameters **entity** (*object*) – the entity to register

register_dirty(*entity*)
Register the entity with the dirty bit

Parameters **entity** (*object*) – the entity to register

register_new(*entity*)
Register a new entity

Parameters **entity** (*object*) – the entity to register

2.24 passerine.db.wrapper

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at `passerine/db/wrapper.py`.

2.25 passerine.decorator.common

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at `passerine/decorator/common.py`.

Author Juti Noppornpitak

This package contains decorators for common use.

class passerine.decorator.common.**BaseDecoratorForCallableObject**(*reference*)
Base decorator based from an example at http://www.artima.com/weblogs/viewpost.jsp?thread=240808.

passerine.decorator.common.**make_singleton_class**(*class_reference*, **args*, ***kwargs*)
Make the given class a singleton class.

class_reference is a reference to a class type, not an instance of a class.

args and *kwargs* are parameters used to instantiate a singleton instance.

To use this, suppose we have a class called `DummyClass` and later instantiate a variable `dummy_instnace` as an instance of class `DummyClass`. `class_reference` will be `DummyClass`, not `dummy_instance`.

Note that this method is not for direct use. Always use `@singleton` or `@singleton_with`.

`passerine.decorator.common.`**`singleton`**(**args*, ***kwargs*)

Decorator to make a class to be a singleton class. This decorator is designed to be able to take parameters for the construction of the singleton instance.

Please note that this decorator doesn't support the first parameter as a class reference. If you are using that way, please try to use `@singleton_with` instead.

Example:

```
# Declaration
@singleton
class MyFirstClass(ParentClass):
    def __init__(self):
        self.number = 0
    def call(self):
        self.number += 1
        echo self.number
# Or
@singleton(20)
class MySecondClass(ParentClass):
    def __init__(self, init_number):
        self.number = init_number
    def call(self):
        self.number += 1
        echo self.number

# Executing
for i in range(10):
    MyFirstClass.instance().call()
# Expecting 1-10 to be printed on the console.
for i in range(10):
    MySecondClass.instance().call()
# Expecting 11-20 to be printed on the console.
```

The end result is that the console will show the number from 1 to 10.

`passerine.decorator.common.`**`singleton_with`**(**args*, ***kwargs*)

Decorator to make a class to be a singleton class with given parameters for the constructor.

Please note that this decorator always requires parameters. Not giving one may result errors. Additionally, it is designed to solve the problem where the first parameter is a class reference. For normal usage, please use *@singleton* instead.

Example:

```
# Declaration
class MyAdapter(AdapterClass):
    def broadcast(self):
        print "Hello, world."

@singleton_with(MyAdapter)
class MyClass(ParentClass):
    def __init__(self, adapter):
```

```
            self.adapter = adapter()
        def take_action(self):
            self.adapter.broadcast()

    # Executing
    MyClass.instance().take_action() # expecting the message on the console.
```

The end result is that the console will show the number from 1 to 10.

2.26 passerine.exception

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at `passerine/exception.py`.

exception `passerine.exception.`**DuplicatedPortError**
Exception thrown only when the port config is duplicated within the same configuration file.

exception `passerine.exception.`**DuplicatedRouteError**
Exception used when the routing pattern is already registered.

exception `passerine.exception.`**FutureFeatureException**
Exception used when the future feature is used where it is not properly implemented.

exception `passerine.exception.`**InvalidConfigurationError**
Exception thrown only when the configuration is invalid.

exception `passerine.exception.`**InvalidControllerDirectiveError**
Exception used when the controller directive is incomplete due to missing parameter

exception `passerine.exception.`**InvalidInput**
Exception used when the given input is invalid or incompatible to the requirement.

exception `passerine.exception.`**InvalidRedirectionDirectiveError**
Exception used when the redirection directive is incomplete because some parameters aren't provided or incompatible.

exception `passerine.exception.`**LoadedFixtureException**
Exception raised when the fixture is loaded.

exception `passerine.exception.`**RendererNotFoundError**
Exception thrown when the unknown template repository is used.

exception `passerine.exception.`**RendererSetupError**
Exception thrown when there exists errors during setting up the template.

exception `passerine.exception.`**RenderingSourceMissingError**
Exception used when the rendering source is not set.

exception `passerine.exception.`**RoutingPatternNotFoundError**
Exception used when the routing pattern is not specified in the configuration file.

exception `passerine.exception.`**RoutingTypeNotFoundError**
Exception used when the routing type is not specified in the configuration file.

exception `passerine.exception.`**SessionError**
Exception thrown when there is an error with session component.

exception passerine.exception.**SingletonInitializationException**
This exception is used when the target class contain a special attribute *_singleton_instance* not a reference to its own class.

exception passerine.exception.**UnexpectedComputationError**
Exception used when the code runs mistakenly unexpectedly.

exception passerine.exception.**UnknownRoutingTypeError**
Exception used when the routing type is not unknown.

exception passerine.exception.**UnknownServiceError**
Exception thrown when the requested service is unknown or not found.

exception passerine.exception.**UnsupportObjectTypeError**
Exception used when the unsupported object type is used in an inappropriate place.

Please note that this is a general exception.

exception passerine.exception.**UnsupportedRendererError**
Exception thrown when the unsupported renderer is being registered.

2.27 passerine.graph

Note: This page is automatically generated. If you don't see anything, this means this sub-module is not meant to be used. If you really want to know what it is, please check out the source code at passerine/graph.py.

class passerine.graph.**DependencyNode**
Dependency Node

This is designed to be bi-directional to maximize flexibility on traversing the graph.

CHAPTER 3

Guide

This section will guide you on how to use the library properly.

3.1 1. Getting Started

Now, you get through the boring stuff. Let's write some code.

3.1.1 Set Up the Connection and Entity Manager

```
from passerine.db.manager import ManagerFactory
manager_factory = ManagerFactory()

# Define the connection URL
manager_factory.set('default', 'mongodb://localhost/sample_db')

# Get the "default" entity manager
entity_manager = manager_factory.get('default')
```

The first three two is to set up the entity manager factory (`ManagerFactory`).

Then, on the next line, we define the **default** connection to `mongodb://localhost/sample_database`.

Then, use the entity manager factory to get an instance of the **default** entity manager.

3.1.2 Define an Entity

Note: From this section on, **collection** (used by NoSQL DBs), **bucket** (used by distributed DBs) and **table** (used by relational DBs) are collectively the same thing for the library.

Now, we have an entity manager to work with. Next, we need to define the data structure. For instance, we define two entity classes: **Character** and **Team**.

```
from passerine.db.entity import entity

@entity
class Player(object):
    def __init__(self, name, level, team=None):
        self.name  = name
```

```
        self.team  = team
        self.level = level

    def __repr__(self):
        attrs = {
            'name':  self.name,
            'team':  self.team,
            'level': self.level
        }

        return '<{} {}>'.format(self.__class__.__name__, attrs)

@entity('teams')
class Team(object):
    def __init__(self, name, location):
        self.name = name
        self.location = location

    def __repr__(self):
        attrs = {
            'name': self.name,
            'location': self.location
        }

        return '<{} {}>'.format(self.__class__.__name__, attrs)
```

1. `@entity` is to define the decorated class as an entity class. If all entities of this class will be saved in the collection, named **character**.
2. `@entity('teams')` is similar to `@entity` except that the name of the destination collection is **teams**.
3. The constructor must never take an identifier (e.g. `id` or `_id`) as a parameter.
4. The name of the parameters of the constructor must be the same as the property.

Note: Currently, data-object mapping is very straight forward. If the given data is

```
// mongodb: sample_db.teams
team_raw_data = {
    '_id':       123,
    'name':     'SEED 8',
    'location': 'Wakayama, Japan'
}
```

then, the data mapper will try to do somthing similar to the following code:

```
team_entity = Team(
    name     = team_raw_data['name'],
    location = team_raw_data['location'],
)
team_entity.id = team_raw_data['_id']
```

The proper data mapping mechanism will be introduced in later releases.

You may use a property getter or setter to restrict the access to the property if needed.

Now, we have **Entity** classes and a working entity manager. What can we do next?

3.2 2. Create a New Entity

In the previous step, we have an entity manager (`entity_manager`) and two entity classes (`Player` and `Team`). Now, we are going to create a new entity.

Let's create **one player**.

```
sid = Player('Sid', 5)

session = entity_manager.open_session()

repository = session.repository(Player)
repository.persist(sid)
repository.commit()
```

From the code, `sid = Player('Sid')` is to create an object.

The rest are pretty simple.

1. We open a DB (pseudo) session with `open_session()`,
2. find the corresponding repository, referred as `repository`,
3. tell the repository to push the new object on commit with `persist()`,
4. and finally commit the change to the database.

Note: Alternatively, you may forget `repository` by using `session` directly by replacing everything about `repository` with the following code.

```
session.persist(sid)
session.flush()
```

where the session's `persist` and `commit` are the same as the repository's `persist` and `flush` respectively. **For the sake of tutorial, we will keep using the repository approach.**

Warning: The database session is pseudo for unsupported databases.

Note: The way we use `open_session()` here is to open an unsupervised session. The session connection cannot be closed by the supervising entity manager but the connection is still closed as soon as the process is terminated.

After this process, you can verify with MongoDB CLI (`mongo sample_db`) by running `db.player.find()`. You should be able to see the result on the screen like the following (except `_id`).

```
{
    "_id" : ObjectId("abc"),
    "name" : "Sid",
    "level": 5,
    "team" : null
}
```

3.3 3. Find entities (Basic)

Previously, we added an entity. Now, we just need to find it properly.

However, just before we move onto the next step, let's add more Player entities.

```
# Add more entities
repository.persist(Player('Ramza', 9))
repository.persist(Player('Tsunemori', 6))
repository.commit()
```

3.3.1 Find many entities without constrains

Suppose we want to retrieve everyone. There are two ways you can do it.

First, just invoke the `find` method of `repository`.

```
result = repository.find()
```

Or alternatively, you can create a new *passerine.db.query.Query* via `repository`.

```
query = repository.new_criteria('p')

result = repository.find(query)
```

You should then see the same output (`result`) like the following:

```
[<Player {'level': 5, 'name': 'Sid', 'team': None}>, <Player {'level': 6, 'name': 'Tsunemori', 'team
```

Warning: *passerine.db.query.Query* must always be instantiated via *passerine.db.repository.Repository*. Please do not try to instantiate the **Query** class directly.

3.3.2 Find many entities with constrains

Suppose we want to retrieve the information of the player named "Sid". There are two ways you can do it.

Hand-coded expectation

First, just hand-code the expected value.

```
query = repository.new_criteria('p')
query.expect('p.name = "Sid"')

result = repository.find(query)
```

`query = repository.new_criteria('p')` is to instantiate a query object (*passerine.db.query.Query*) where the alias of the expected entitiy is `p`. Then, `query.expect('p.name = "Sid"')` set the expectation that the attribute `name` of the expected entities must be equal to `Sid`. `result = repository.find(query)` will perform the query with the specification defined in `query`. If you execute the code, you should see the output (`result`) like this:

```
[<Player {'level': 5, 'name': 'Sid', 'team': None}>]
```

Note: The syntax for the expectation is ironically similar to SQL and DQL (Doctrine Query Language). It is by design to generalize the interface between different types of drivers.

Parameterized expectation (exact match)

However, the previous `query` is not reusable with the dynamic data. We will create the Query object with **parameters** to allow use to reuse the same query object for different parameters.

```
query = repository.new_criteria('p')
query.expect('p.name = :name')
query.define('name', 'Sid')

result = repository.find(query)
```

In this snippet, you will see that in `query.expect('p.name = :name')`, `:name` replaces `"Sid"` and `query.define('name', 'Sid')` defines the value of the parameter `name`. This is equivalent to `query.expect('p.name = "Sid"')` as you will see the same output (`result`):

```
[<Player {'level': 5, 'name': 'Sid', 'team': None}>]
```

However, without recreating the Query object, if I re-define the parameter `name` with `Tsunemori` by adding:

```
query.define('name', 'Tsunemori')
```

you will now get the different result:

```
[<Player {'level': 6, 'name': 'Tsunemori', 'team': None}>]
```

Note: This is the recommended way to query.

Range query

To do range search, just like the previous examples, you can either hand-code the expectation or rely on the parameterization. The following example uses the latter:

```
# Range-query multiple
query = repository.new_criteria('p')
query.expect('p.level >= :min')
query.expect('p.level <= :max')
query.define('min', 5)
query.define('max', 6)

result = repository.find(query)
```

And `result` becomes:

```
[
    <Player {'level': 5, 'name': 'Sid', 'team': None}>,
    <Player {'level': 6, 'name': 'Tsunemori', 'team': None}>
]
```

In-set query (IN operator)

To do in-set search, unlike the previous examples, you can only rely on the parameterization. For example:

```
query = repository.new_criteria('p')
query.expect('p.level IN :expected_level')
query.define('expected_level', (6, 9)) # or [6, 9]
```

```
result = repository.find(query)
```

And `result` becomes:

```
[
    <Player {'level': 9, 'name': 'Ramza', 'team': None}>,
    <Player {'level': 6, 'name': 'Tsunemori', 'team': None}>
]
```

Regular-expression query (LIKE operator)

To do in-set search, unlike the previous examples, you can only rely on the parameterization. For example:

```
query = repository.new_criteria('p')
query.expect('p.name LIKE :name')
query.define('name', '^Ra')

result = repository.find(query)
```

And `result` becomes:

```
[<Player {'level': 9, 'name': 'Ramza', 'team': None}>]
```

To just retrieve everything, you can either do:

```
query = repository.new_criteria()
repository.find(query)
```

or just go with:

```
repository.find(query)
```

3.4 4. Associations and Relational Mappings

Usually data from different collections are related and may co-exist. As the result, even though some database softwares do not allow associations, Passerine allows the software-based relational mapping. This ORM supports all types of relational mapping either unidirectionally or bidirectionally.

Passerine ORM utilizes two patterns to implement association.

- Decorators (AKA annotations) to define associations.
- Lazy loading to load data when it is requested.
- Proxy object as direct result of lazy loading.

In general, the decorator *`passerine.db.mapper.link()`* is used to define association by mapping decorated fields to another classes by primary key (or object ID). The ID-to-object happens automatically during data mapping.

3.4.1 Types of Associations

For the sake of the simplicity of this chapter, all examples are assumed to be in the module `sampleapp.model`, and begin with:

```
from passerine.db.entity import entity
from passerine.db.mapper import link, AssociationType as t, CascadingType as c
```

One-to-one

Suppose there are two entities: `Owner` and `Restaurant`, **one-to-one associations** imply the relationship between two entities as described in the following UML:

```
Owner (1) ----- (1) Restaurant
```

Unidirectional

UML:

```
Owner (1) <--x- (1) Restaurant
```

Suppose we have two classes: `Owner` and `Restaurant`, where `Restaurant` has the one-to-one unidirectional relationship with `Owner`.

```
@entity
class Owner(object):
    def __init__(self, name):
        self.name  = name

@link(
    target      = 'sampleapp.model.Owner',
    mapped_by   = 'owner',
    association = t.ONE_TO_ONE
)
@entity
class Restaurant(object):
    def __init__(self, name, owner):
        self.name  = name
        self.owner = owner
```

where the sample of the stored documents will be:

```
// collection: owner
{'_id': 'o-1', 'name': 'siamese'}

// collection: restaurant
{'_id': 'rest-1', 'name': 'green curry', 'owner': 'o-1'}
```

Tip: To avoid the issue with the order of declaration, the full namespace in string is recommended to define the target class. However, the type reference can also be. For example, `@link(target = Owner, ...)`.

Bidirectional

UML:

```
Owner (1) <---> (1) Restaurant
```

Now, let's allow `Owner` to have a reference back to `Restaurant` where the information about the reference is not kept with `Owner`. So, the

```
@link(
    target      = 'sampleapp.model.Restaurant'
    inverted_by = 'owner',
    mapped_by   = 'restaurant',
    association = t.ONE_TO_ONE
)
@entity
class Owner(object):
    def __init__(self, name, restaurant):
        self.name       = name
        self.restaurant = restaurant
```

where the the stored documents will be the same as the previous example.

`inverted_by` means this class (`Owner`) maps `Restaurant` to the property *restaurant* where the value of the property *owner* of the corresponding entity of Restaurant must equal the *ID* of this class.

Note: The option `inverted_by` only maps `Owner.restaurant` to `Restaurant` virtually but the reference is stored in the **restaurant** collection.

Many-to-one

Suppose a `Customer` can have many `Reward`'s as illustrated:

```
Customer (1) ----- (0..n) Reward
```

Unidirectional

UML:

```
Customer (1) <--x- (0..n) Reward
```

```
@entity
class Customer(object):
    def __init__(self, name):
        self.name    = name

@link(
    target      = 'sampleapp.model.Customer',
    mapped_by   = 'customer',
    association = t.MANY_TO_ONE
)
@entity
class Reward(object):
    def __init__(self, point, customer):
        self.point    = point
        self.customer = customer
```

where the data stored in the database can be like this:

```
// collection: customer
{'_id': 'c-1', 'name': 'panda'}
```

```
// collection: reward
{'_id': 'rew-1', 'point': 2, 'customer': 'c-1'}
{'_id': 'rew-2', 'point': 13, 'customer': 'c-1'}
```

Bidirectional

UML:

```
Customer (1) <---> (0..n) Reward
```

Just change `Customer`.

```
@link(
    target      = 'sampleapp.model.Reward',
    inverted_by = 'customer',
    mapped_by   = 'rewards',
    association = t.ONE_TO_MANY
)
@entity
class Customer(object):
    def __init__(self, name, rewards):
        self.name    = name
        self.rewards = rewards
```

where the property *rewards* refers to a list of rewards but the stored data remains unchanged.

Note: This mapping is equivalent to a **bidirectional one-to-many mapping**.

One-to-many

Let's restart the example from the many-to-one section.

Unidirectional with Built-in List

The one-to-many unidirectional mapping takes advantage of the built-in list.

UML:

```
Customer (1) -x--> (0..n) Reward
```

```
@link(
    target      = 'sampleapp.model.Reward',
    mapped_by   = 'rewards',
    association = t.ONE_TO_MANY
)
@entity
class Customer(object):
    def __init__(self, name, rewards):
        self.name    = name
        self.rewards = rewards

@entity
class Reward(object):
```

```
    def __init__(self, point):
        self.point = point
```

where the property `rewards` is a unsorted iterable list of `Reward` objects and the data stored in the database can be like this:

```
// collection: customer
{'_id': 'c-1', 'name': 'panda', 'reward': ['rew-1', 'rew-2']}

// collection: reward
{'_id': 'rew-1', 'point': 2}
{'_id': 'rew-2', 'point': 13}
```

Warning: As there is no way to enforce relationships with built-in functionality of MongoDB and there will be constant checks for every write operation, it is not recommended to use unless it is for **reverse mapping** via the option `inverted_by` (see below for more information).
Without a proper checker, which is not provided for performance sake, this mapping can be used like the **many-to-many join-collection mapping**.

Bidirectional

See *Many-to-one Bidirectional Association*.

Many-to-many

Suppose there are `Teacher` and `Student` where students can have many teachers and vise versa:

```
Teacher (*) ----- (*) Student
```

Similar other ORMs, the many-to-many mapping uses the corresponding join collection.

Unidirectional with Join Collection

UML:

```
Teacher (*) <--x- (*) Student
```

```
@entity('teachers')
class Teacher(object):
    def __init__(self, name):
        self.name = name

@link(
    mapped_by   = 'teachers',
    target      = Teacher,
    association = AssociationType.MANY_TO_MANY,
    cascading   = [c.DELETE, c.PERSIST]
)
@entity('students')
class Student(object):
    def __init__(self, name, teachers=[]):
        self.name     = name
        self.teachers = teachers
```

where the stored data can be like the following example:

```
// db.students.find()
{'_id': 1, 'name': 'Shirou'}
{'_id': 2, 'name': 'Shun'}
{'_id': 3, 'name': 'Bob'}

// db.teachers.find()
{'_id': 1, 'name': 'John McCain'}
{'_id': 2, 'name': 'Onizuka'}

// db.students_teachers.find() // -> join collection
{'_id': 1, 'origin': 1, 'destination': 1}
{'_id': 2, 'origin': 1, 'destination': 2}
{'_id': 3, 'origin': 2, 'destination': 2}
{'_id': 4, 'origin': 3, 'destination': 1}
```

Bidirectional

```
@link(
    mapped_by   = 'students',
    inverted_by = 'teachers',
    target      = 'sampleapp.model',
    association = AssociationType.MANY_TO_MANY
)
@entity('teachers')
class Teacher(object):
    def __init__(self, name, students=[]):
        self.name     = name
        self.students = students

@link(
    mapped_by   = 'teachers',
    target      = Teacher,
    association = AssociationType.MANY_TO_MANY,
    cascading   = [c.DELETE, c.PERSIST]
)
@entity('students')
class Student(object):
    def __init__(self, name, teachers=[]):
        self.name     = name
        self.teachers = teachers
```

3.4.2 Options for Associations

The decorator *passerine.db.mapper.link()* has the following options:

Option	Description
association	the type of associations (See *passerine.db.mapper.AssociationType*.)
cascading	the list of allowed cascading operations (See 6. Cascading *passerine.db.mapper.CascadingType*.)
inverted_by	the name of property used where **enable the reverse mapping if defined**
mapped_by	the name of property to be map
read_only	the flag to disable property setters (only usable with *passerine.db.common.ProxyObject*.)
target	the full name of class or the actual class

3.4.3 How to make a join query

From the customer-reward example, if we want to find all rewards of a particular user, the query will be:

```
query = reward_repository.new_criteria('r')
query.join('r.customer', 'c')
query.expect('c.name = "Bob"')

rewards = reward_repository.find(query)
```

All features for querying is usable with joined entities.

3.5 5. Updating, Deleting and Handling Transactions (Sessions)

Similar to Sessions in SQLAlchemy.

In the most general sense, the session establishes all conversations with the database and represents a "holding zone" for all the objects which you've loaded or associated with it during its lifespan. It provides the entrypoint to acquire a *passerine.db.repository.Repository* object, which sends queries to the database using the current database connection of the session (*passerine.db.session.Session*), populating result rows into objects that are then stored in the session, inside a structure called the identity map (internally being the combination of "the record map" and "the object ID map") - a data structure that maintains unique copies of each object, where "unique" means "only one object with a particular primary key".

The session begins in an essentially stateless form. Once queries are issued or other objects are persisted with it, it requests a connection resource from an manager that is associated with the session itself. This connection represents an ongoing transaction, which remains in effect until the session is instructed to commit.

All changes to objects maintained by a session are tracked - before the database is queried again or before the current transaction is committed, it flushes all pending changes to the database. This is known as **the Unit of Work pattern**.

When using a session, it's important to note that the objects which are associated with it are **proxy objects** (*passerine.db.common.ProxyObject*) to the transaction being held by the session - there are a variety of events that will cause objects to re-access the database in order to keep synchronized. It is possible to "detach" objects from a session, and to continue using them, though this practice has its caveats. It's intended that usually, you'd re-associate detached objects with another Session when you want to work with them again, so that they can resume their normal task of representing database state.

3.5.1 Supported Operations

Supported Operation	Supported Version
Persist	2.1
Delete	2.1
Refresh	2.1
Merge	No plan at the moment
Detach	No plan at the moment

3.5.2 Open a Session

As you might guess from 2. Create a New Entity, Passerine always requires the code to open a session by:

```
session = entity_manager.open_session()
```

3.5.3 Getting Started (again)

Then, try to query a player called **"Sid"**, we created in 2. Create a New Entity with *passerine.db.repository.Repository*:

```
repo = session.collection(Player)

query = repo.new_criteria('c')
query.expect('c.name = :name')
query.define('name', 'Sid')

player = repo.find(query)

# For demonstration only
print('Sid/level: {}'.format(player.level))
```

The output should show:

```
Sid/level: 5
```

3.5.4 Update an Entity

Suppose we want to update his level:

```
player.level = 99
repo.persist(player)
```

3.5.5 Delete an Entity

Suppose we want to delete the entity:

```
session.delete(player)
```

Note: Until you commit the changes, other queries still can return entities you mark as deleted.

3.5.6 Commit Changes

Once I am satisfied with changes, commit the changes with:

```
repo.commit()
```

or:

```
session.flush()
```

Note: We will stick with `session.flush()` as `repo.commit()` is just an alias to `session.flush()`.

Note: To discard all changes at any points, just close the session (mentioned later in this section).

3.5.7 Refresh/revert Changes on One Entity

Or, refresh `player`:

```
session.refresh(player)
```

Then, if `player` is either **persisted** or **deleted**, to flush/commit the change, simply run:

```
session.flush()
```

3.5.8 Close a Session

Closing a session is to end the session or discard all changes to the session. You can simply close the session by:

```
entity_manager.close_session(session)
```

3.5.9 Handle a Session in the Context

To ensure all sessions are closed properly, you can open session as a context manager by doing this:

```
with entity_manager.session() as session:
    ... # do whatever
# session closed
```

3.5.10 Drawbacks Introduced by Either MongoDB or Passerine

1. Even though MongoDB does not support transactions, like some relational database engines, such as, InnoDB, Passerine provides software-based transactions. However, as mentioned earlier, Passerine **does not provide roll-back operations**.
2. **Merging** and **detaching** operations are currently not supported in 2013 unless someone provides the supporting code.
3. Any querying operations cannot find any uncommitted changes.

3.6 6. Cascading

This is the one toughest section to write.

MongoDB, as far as everyone knows, does not support cascading operations like the way MySQL and other vendors do with cascading deletion. Nevertheless, Tori supports cascading through the database abstraction layer (DBAL).

> **Warning:** Cascading persistence and removal via DBAL has high probability of degrading performance with large dataset as in order to calculate a dependency graph, all data must be loaded into the memory space of the computing process. This introduces a spike in memory and network usage.
> This feature is introduced for convenience sake but should be used sparingly or accounted for potential performance degration.

Here is a sample scenario.

Suppose I have two types of objects: a sport team and a player. When a team is updated, removed or refreshed, the associated player should be treated the same way as the team. Here is a sample code.

```
from passerine.db.entity import entity
from passerine.db.mapper import CascadingType as c

@entity
class Player(object):
    pass # omit the usual setup decribed in the basic usage.

@link(
    target=Player,
    mapped_by='player',
    cascading=[c.PERSIST, c.DELETE, c.REFRESH]
)
@entity
class Team(object):
    pass # omit the usual setup decribed in the basic usage.
```

Now, whatever operation is used on a Team entity, associated Player entites are subject to the same operation.

CHAPTER 4

Change Logs

4.1 Version 1.4

4.1.1 Improvements

ManagerFactory

(Short for *passerine.db.manager.ManagerFactory*)

- The alias-to-endpoint-URL map can be passed to the constructor to simplify the usage.
- Shorten the name of each parameters of the constructor.

```
from passerine.db.manager import ManagerFactory

mf = ManagerFactory(urls = {'default': 'mongodb://localhost/sample'})

mf.get('default') # -> a Manager object
```

Others

- The default parameter of the constructor of an Entity class is now respected.

4.1.2 House Cleaning

- Made Python 3 (<3.4) the primary target.
- Cleaned up the test folder.
- Simplified the test running procedure.

4.2 Version 1.1

- (Support Riak 2.0)

4.3 Version 1.0

- A fork of Tori ORM (from Tori 3.0) with lots of little improvements.

CHAPTER 5

Special Thanks

This project is not possible without helps and guidance from Guilherme Blanco from Doctrine Project.

CHAPTER 6

Indices and Modules

- genindex
- modindex

p

A

B

C

D

E

P

Q

R

S

T

U