Tori Documentation

Release 2.1

Juti Noppornpitak

November 29, 2013

Contents

Author Juti Noppornpitak <jnopporn@shiroyuki.com>

Tori is a collection of libraries, micro web framework based on Facebook's Tornado framework 2.x and the ORM for MongoDB and supports Python 2.7+ and Python 3.3+. (Read more from the *Introduction*.)

CHAPTER 1

What next?

Read *Manual* if you want to get started and learn how to use it.

Read *API Reference* if you want to get the documentation of APIs and methods.

1.1 Manual

The manual is for the better understanding on how each part is designed and used in the human language.

1.1.1 Introduction

Tori is a collection of libraries, a micro web framework based on Facebook's Tornado framework 2.x and the ORM for MongoDB and supports Python 2.7+ and Python 3.3+.

Before using

Please note that this framework/library is released under MIT license copyrighted by Juti Noppornpitak. The distributed version of this license is available at https://github.com/shiroyuki/Tori/blob/master/readme.md.

Differences in Idea

As there already exists many web framework for Python, Tori Framework is made for specific purposes.

1. It is to simplify the setup process and customizable.
2. Everything in Tori, beside what Tornado provides, is designed with the concepts of aspect-oriented programming (AOP) and dependency injections (DI) which heavily relies on Imagination Framework.
3. There is no guideline on how developers want to use the code.
4. Many libraries/components are designed for re-usability with or without the web framework part or Imagination Framework (AOP part).

Differences in Code

Even though Tori is based on Tornado, there are a few elements that differ from the Tornado.

1. The setup script is different as the setup process of Tori Framework is designed to be a wrapper for Tornado's Application.
2. Tori Framework overrides the default template engine with Jinja2.
3. Tori Framework's controller extends Tornado's `RequestHandler` with the integration with Tori's session controller and the template engine.
4. Tori Framework can handle more than one route to static resource.
5. Provide a simple way to define routes. (added in 2.1)

Prerequisites

Module	Required Third-party Modules
`tori.application`	tornado 2.4+/3+
`tori.controller`	tornado 2.4+/3+
`tori.socket`	tornado 2.4+/3+
`tori.db`	pymongo 2.3+ / sqlalchemy 0.7+
`tori.session`	redis 2.7+
`tori.template`	jinja2 2.6+

Note: It is not required to have all of them. You can keep only what you need.

Installation

You can install via **PIP** command or **easy_install** command or you can download the source code and run `python setup.py install` or `make install`.

Warning: There is no plan on supporting the legacy releases of Python as the project moves forward to **Python 3.3 or higher**. **Python 2.7** is the last series of Python 2 being supported by the project. **Python 2.6** seems to be working but ßthe framework is not tested.

1.1.2 Concept and Philosophy

Tori Framework is designed to incorporates:

- the adapted version of **PEP 8** with Object Calisthenics
- **the aspect-oriented programming pattern**
- **the dependency injection pattern**

altogether. Despite of that, there are a few irregular things: the controller-repository-model pattern, standalone sub-modules and non-circular dependency graph.

Controller-Repository-Model Pattern (CRM)

If a sub module has to deal with static or indexed data, the controller-repository-model pattern (CRM) will be used where:

- **controllers** are front-end interfaces used to provide data in the general way
- **repositories** are back-end interfaces used to access data specifically for a particular type of data
- **models** or **entities** are models representing the data retrieved by the repositories and known by the controllers.

For instance, the session module has `tori.session.controller.Controller` as the only controller, any classes in `tori.session.repository` as a repository and any classes in `tori.session.entity` as an entity (or data structure) if required by the repository.

Standalone Sub-modules

Some sub-modules are designed to work independently without the need of other sub-modules. This only applies to low-level modules like navigation (`tori.navigation`), ORM (`tori.db`) and templating module (`tori.template`).

Non-circular Dependency Graph

All modules in Tori Framework have unidirectional relationship at the module and code level. The reasons beside all of other cool reasons, many of which you may have heard somewhere else, of doing this is for easy maintenance, easy testing and infinite-loop prevention.

1.1.3 Getting Started

Installation

There are options for the installation:

- Use PyPI to install by running `sudo pip install tori`,
- Download or clone the source code and run `make install` or `python setup.py install`.

Tip: The second option is preserved for development or installing the pre-release package from the source code.

Hello, world... again?

In our imagination, we want to write a dead simple app to just say “Hello” to someone.

Suppose we have **Tori** on the system path and the following project structure:

```
project/
    app/
        __init__.py
        error.py
        views/
            (empty)
    resources/
        readme.txt
        city.jpg
```

First, we write a controller `project/app/controller.py` with:

```
from tori.controller           import Controller
from tori.decorator.controller import renderer

@renderer('app.views')
class MainController(Controller):
    def get(self, name):
        self.render('index.html', name=name)
```

`@renderer('app.views')` is to indicate that the template is in `app.views` and `self.render` is to render a template at `index.html` in `app.views` with a context variable `name` from a query string.

Note: Read more on *Controller*.

Next, we write a Jinja2 template `project/app/views/index.html` with:

```
<!doctype html>
<html>
<head>
    <title>Example</title>
</head>
<body>
    Hello, {{ name }}
</body>
</html>
```

Note: Read more on *Templates*.

Then, we need to write a configuration file. For this example, we will save to at `project/server.xml` containing:

```
<?xml version="1.0" encoding="utf-8"?>
<application>
    <server>
        <port>8000</port>
    </server>
    <routes>
        <controller
            class="demo.app.controller.main.MainController"
            pattern="/{name}"
            regexp="false"
        />
        <resource
            location="resources"
            pattern="/resources/**"
            cache="true"
            regexp="false"
        />
        <redirection
            destination="http://shiroyuki.com"
            pattern="/about-shiroyuki"
        />
    </routes>
    <services/>
</application>
```

Tip: This example uses the simple routing pattern introduced in Tori 2.1. See *Routing* for more detail.

Note: See *Configuration* for more information on the configuration.

Then, we write a bootstrap file at `project/server.py` containing:

```
from tori.application import Application

application = Application('server.xml')
application.start()
```

Now, to run the server, you can simply just execute:

```
python server.py
```

You should see it running.

1.1.4 Configuration

Author Juti Noppornpitak

The configuration in Tori framework is written on XML. The only reason is because it is validable and flexible. It is largely influenced by the extensive uses of JavaBeans in Spring Framework (Java) and the lazy loading in Doctrine (PHP).

Specification

Here is the complete specification of the configuration file:

```
permitive_boolean ::= 'true' | 'false'

root_node ::= '<application>' include_node server_node routing_node service_node '</application>'

# Include other configuration files
include_node ::= '<include src="' include_file_path '"/>' include_node | ''
# "include_file_path" is a string representing either an absolute path or a relative path to the curr
# working directory of the script.

# Server-specific configuration
server_node ::= '<server>' server_debug_node server_port_node server_error_node '</server>' | ''

server_config_node ::= (
        server_debug_node
        | server_port_node
        | server_error_node
    )
    server_config_node

# Debug switch (which can be overridden by the app constructor)
server_debug_node ::= '<debug>' permitive_boolean '</debug>' | ''
# Default to "true"

# Server port number
server_port_node ::= '<port>' server_port_number '</port>' | ''
# E.g., 80, 443, 8000 (default) etc.

# Custom error delegate/handler as a controller.
```

```
server_error_node ::= '<error>' server_error_class '</error>' | ''
# "server_error_class" is a string representing the full name of the error controller class, for inst
# com.shiroyuki.www.controller.ErrorController. If not specified, the default handler will be decided
# Tornado's code.

# Routing configuration
routing_node ::= '<routes>' routing_route_node '</routes>'

routing_route_node ::= (
        routing_route_controller_node
        | routing_route_redirection_node
        | routing_route_resource_node
    )
    routing_route_node
    | ''

tornado_route_pattern ::= 'pattern="' tornado_route_pattern_regexp '"'

# "controller_class" is a string representing the full name of the controller class, for instance,
# com.shiroyuki.www.controller.HomeController.

# Controller
routing_route_controller_node ::= '<controller class="' controller_class '" ' tornado_route_pattern '

# Redirection
routing_route_redirection_node ::= '<redirection destination="' tornado_route_pattern '" ' tornado_ro

# Resource
routing_route_resource_node ::= '<resource location="' file_path_pattern '" ' tornado_route_pattern '

# Service configuration
service_node ::= '<service>' include_file_path '</service>' service_node | ''
```

Note: DTD will be provided as soon as someone is willing to help out on writing.

You can see the example from the configuration of The Council Project on GitHub.

See More

Routing

Routing Order and Priority The routes (`<routes>`) is prioritized by the order in the routing list.

Types of Directives There are *3 types* of routes being supported.

Directive	Description
controller	A routing directive for dynamic content handled by a controller.
resource	A routing directive for static content/resource.
redirection	A routing directive for relaying requests with redirection.

Common Attributes

Attribute	Description	Expected Values
pattern	the routing pattern	*regular expression* or *simple patter* (string)
regexp	the flag to indicate whether the given routing pattern is simplified	`true` or `false` (boolean)

Regular-expression Routing Pattern In general, the attribute `pattern` of any routing directives is to indicate the routing pattern where the directive intercepts, process and respond to any requests to the pattern. Each routing pattern is unique from each other.

Simple Routing Pattern New in version 2.1. By default, similar to Tornado, Tori Framework uses the normal regular expression for routing. However, this could introduce an error-prone routing table for anyone that does not know the regular expression. Here is the syntax where the routing resolver considers in the following presented order.

Simple Pattern Syntax	Equivalent Regular Expression
`**`	`(.+)`
`*`	`([^/]+)`
`{name}`	`(?P<name>.+)`

Here are the simple versions of routing patterns.

Simple Pattern	Equivalent Regular Expression	Expected Parameter List/Map
`/abc/def/ghi/**`	`/abc/def/ghi/(.+)`	index `0` or the first key
`/abc/def/ghi/*/jkl`	`/abc/def/ghi/([^/]+)/jkl`	index `0` or the first key
`/abc/def/ghi/{key}/jkl`	`/abc/def/ghi/(?P<key>.+)/jkl`	key `key`

To enable the simple routing pattern, the `regexp` attribute must be `false` (not default).

Default Routes for FAVICON New in version 2.1. In addition to the simple routing, the default route for `/favicon.ico` is available if not assigned.

Controller For a routing directive `controller`, the attribute `class` is a class reference to a particular controller where the controller must be on the system path (for Python).

```
<controller class="app.note.controller.IndexController" pattern="/notes/(.*)"/>
```

Redirection For a routing directive `redirection`, the attribute `destination` is a string indicating the destination of the redirection, and the attribute `permanent` is a boolean indicating whether the redirection is permanent.

```
<redirection destination="/notes/" pattern="/notes"/>
```

Resource For a routing directive `resource`, the attribute `location` is either:

- an absolute or relative path to static resource,
- a module name containing static resource.

the attribute `cache` is a boolean to indicate whether the resource should be cache.

```
<resource location="resources/favicon.ico" pattern="/favicon.ico" cache="true"/>
```

Service

Author Juti Noppornpitak

The services is prioritized by the appearing order of `<service>` in the file.

The content of the `<service>` block is the absolute or relative path to the service configuration file and follows *‘the specification <https://imagination.readthedocs.org/en/latest/api/helper.assembler.html#xml-schema>‘_* of Imagination Framework.

1.1.5 Object-relational Mapping (ORM)

Tori Framework introduces the object-relational mapping module for MongoDB 2.0 or newer.

Introduction

The object-relational mapping (ORM) module in Tori is designed for non-relational databases. The current version of ORM is designed only for MongoDB 2.2 or newer. There are plans for other kinds of databases but there are not enough resources.

Definitions

In this documentation, let’s define:

Entity Document

Object ID An primitive identifier (string, integer, or floating number) or an instance of `bson.ObjectId`

Pseudo ID an instance of `tori.db.common.PseudoObjectId`

Architecture

There are a few points to highlight.

- The lazy-loading strategy and proxy objects are used for loading data wherever applicable.
- The ORM uses **the Unit Of Work pattern** as used by:
  - Hibernate (Java)
  - Doctrine (PHP)
  - SQLAlchemy (Python)
- Although MongoDB does not has transaction support like MySQL, the ORM has sessions to manage the object graph within the same memory space.
- 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.

- 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**.

Limitation

- 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.
- Sessions cannot merge together.
- **Cascading operations on deleting** forces the ORM to load the whole graph which potentially introduces performance issue on a large data set.
- **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.

Getting Started

The chapter illustrates how to define entities and set up an entity manager.

Define an Entity

First, we define the entity (document) class.

```
from tori.db.entity import entity

# Alternatively, @entity('name_of_collection') is to set the name of the collection.
@entity
class Character(object):
    def __init__(self, name):
        self.name = name
```

where an entity of class `Character` automatically has a readable and writable property `id` which can be set only once.

Warning: It is not recommended to the ID manually. Leave setting the ID to the backend database.

Define the Entity Manager

Then, define the entity manager.

```
from pymongo         import MongoClient
from tori.db.manager import Manager

connection     = MongoClient()
entity_manager = Manager('default', connection)
```

Basic Usage

Create a new entity

Suppose two characters: “Ramza”, and “Alma”, are to created. The ORM provides two ways to create a new entity.

Using the constructor directly

```
ramza = Character('Ramza')
alma  = Character('Alma')

character_collection.post(ramza)
character_collection.post(alma)
```

Using the “new” method

```
session = entity_manager.open_session(supervised=False)
collection = session.collection(Character)

ramza = collection.new(name = 'Ramza')
alma = collection.new(name = 'Alma')

collection.post(ramza)
collection.post(alma)
```

Note: for the following example, assume that `ramza.id` is `1` and `alma.id` is `2`.

List, query or filter entities

To list all characters (documents),

```
characters = collection.filter()

for character in characters:
    print('{}: {}'.format(character.id, character.name))
```

Then, you should see:

```
1: Ramza
2: Alma
```

Now, to find “Ramza”,

```
characters = collection.filter({'name': 'Ramza'})

for character in characters:
    print('{}: {}'.format(character.id, character.name))
```

Then, you should only see:

```
1: Ramza
```

Note: The criteria (e.g., in this case `{'name': 'Ramza'}`) is the same one used by `pymongo.collection.Collection`.

Retrieve an entity by ID

Now, to retrieve an entity by ID,

```
alma = collection.get(2)
```

Note: There is no auto-conversion from any given ID to `bson.ObjectId` as the ID can be anything. If the ID of the target entity is of type `bson.ObjectId`, e.g., `"2"` is a string representation of the `ObjectId`, the code has to be `alma = collection.get(bson.ObjectId('2'))`. (Assume that instantiating is okay.)

Update entities

Let's say you want to rename "Alma" to "Luso".

```
alma = collection.get(2)
```

```
alma.name = 'Luso'
```

You can update this by

```
collection.put(character)
```

Delete entities

```
collection.delete(alma)
```

Working with Associations

This chapter introduces association mappings which directly use object IDs to refer to the corresponding objects.

Tori only uses **decorators** (or annotations in some other languages) to define the association mapping.

Instead of working with the object IDs directly, you will always work with references to objects:

- A reference to a single object is represented by object IDs.
- A collection of objects is represented by many object IDs pointing to the object holding the collection

Note: As **lazy loading** is the heart of architectural design of the ORM, when an entity is mapped to an existing document, each property of the entity *in the clean state* will be a reference to either `tori.db.common.ProxyObject`, which loads the data on demand for any **non-many-to-many mappings**, or `tori.db.common.ProxyCollection`, which loads the list of proxy objects to the respective entities on demand only for any **many-to-many mappings**.

There are two sections in this chapter:

- types of associations
- options for associations

Types of Associations

In general, the decorator `tori.db.mapper.link()` is used to define the association a property of the decorated class to the another class.

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

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

Before getting started, here is the general table of abilities which will be explained later on in this chapter.

Ability	**Origin**	**Destination**	
Unidirectional		**Bidirectional**	
Map a property to object	Yes	N/A	Yes
Cascade opeations	Yes	N/A	No, Ignored
Force read-only mode	Yes	N/A	Yes

where available operations are “merge”, “delete”, “persist”, and “refresh”.

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 Under development for Tori 2.1 (https://github.com/shiroyuki/Tori/issues/27).

Options for Associations

The decorator `tori.db.mapper.link()` has the following options:

Option	Description
association	the type of associations (See `tori.db.mapper.AssociationType`.)
cascading	the list of allowed cascading operations (See *Cascading* `tori.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 `tori.db.common.ProxyObject`.)
target	the full name of class or the actual class

See Also:

Database APIs

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 `tori.db.orm.repository.Repository` object, which sends queries to the database using the current database connection of the session (`tori.db.orm.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** (`tori.db.orm.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.

Supported Operations

Supported Operation	Supported Version
Persist	2.1
Delete	2.1
Refresh	2.1
Merge	Planned for 2014
Detach	Planned for 2014

Example

First, define the entity manager.

```
from pymongo              import Connection
from tori.db.orm.manager import Manager

connection     = Connection()
entity_manager = Manager('default', connection)
```

Tip: Alternatively, you can write just `entity_manager = Manager('default')` where the manager will use the default settings of `Connection`, which is for **localhost** on the default port.

Then, open a session:

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

Then, try to query for "Bob" (`User`) with `tori.db.orm.repository.Repository`:

```
bob = session.collection(User).filter_one({'name', 'Bob'})
print(bob.address)
```

The output should show:

```
Bangkok, Thailand
```

Then, update his address:

```
bob.address = 'London, UK'
session.persist(bob)
```

Or, delete `bob`:

```
session.delete(bob)
```

Or, refresh `bob`:

```
session.refresh(bob)
```

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

```
session.flush(bob)
```

Drawbacks Introduced by Either MongoDB or Tori

1. Even though MongoDB does not support transactions, like some relational database engines, such as, InnoDB, Tori provides software-based transactions. However, as mentioned earlier, Tori **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.

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.

```python
from tori.db.entity import entity
from tori.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.

Testing Environments

The ORM is tested with the following configurations.

MongoDB Version	Operating System / Platform
2.2+	Mac OS X 10.8 Server
2.2+	GNU/Linux Debian*
2.2+	Fedora Core*
anything versions	Travis CI

Note: Only test on the latest stable version of OSs running on the latest version of VirtualBox 4.2 on Mac OS X.

See also

- *Database APIs*

1.1.6 Controller

Tori's framework ships with a based controller, extending from `tornado.web.RequestHandler`. So, the usage is pretty much the same as you can find in Tornado's documentation.

Suppose we have the following file structure.:

```
web/
    __init__.py
    controller.py
    views/
        index.html
        error.html
```

Create a Controller

Let's start with create a controller in **web/controller.py**

```
# Module: web.controller (web/controller)
from tori.controller import Controller

class HomeController(Controller):
    def get(self, name):
        self.write('Hello, {}.'.format(name))
```

However, as mentioned earlier, the rendering engine is replaced with Jinja2. By default, the methods `render` and `render_template` of *Controller* are not ready to use.

Enable the Template Engine

Template Engine in Tori Framework is **totally optional** but enabling is not a big problem.

Before getting started, the integration between the rendering part and the controller part is based on the concept of flexibility where each controller can use any template engine or any source. For instance, two controllers may use two different engines or sources.

First, the decorator `tori.decorator.controller.renderer` (or `@renderer` for short) must be imported.

```
from tori.decorator.controller import renderer
```

where the only parameter of `@renderer` is either the name of the package (`web.views`) or the file path (**web/views**). In this example, we use the package.

```
@renderer('web.views')
class HomeController(Controller):
    pass
```

Note: The file path can be either relative with regard of the current working directory or absolute. However, using the package option is recommended.

Suppose the content of **web/views/index.html** is

```
Hello, {{ name }}
```

Then, we replace `self.write(...)` with

```
self.render('index.html', name=name)
```

There is only one default and reserved variable `app` with two attributes:

- `app.request`: an instance of controller's request `tornado.httpserver.HTTPRequest`
- `app.session`: a reference to controller's session getter `tori.session.controller.Controller`

Using Session

Where Tornado framework provide nothing regarding to session management, Tori integrates the cookie-based session controller.

Note: The session controller works with both secure and non-secure cookies. The secure cookies are highly recommended.

The session controller for the session data for a particular session ID is accessible via the read-only property `session` of the controller. For example, to get a session key "userId", you can do by

```
self.session.get('userId')
```

from any method of the controller. Please read more from `tori.session.controller.Controller`.

REST Controller

Tori provides the base controller `tori.controller.RestController` for CRUD operations. It is however designed strictly for querying, creating, retrieving, updating and deleting data.

To use it, the route pattern must accept only one parameter where it is optional. For example, the route can be

```
<controller class="web.controller.BlogEntryRestController" pattern="/blog/rest/entry/(.*)"/>
```

where `web.controller.BlogEntryRestController` is

```
class BlogEntryRestController(RestController):
    def list(self):
        # GET /blog/rest/entry/
        # query the list of entries
        pass

    def create(self):
        # POST /blog/rest/entry/
        # create a new entry
        pass

    def retrieve(self, id):
        # GET /blog/rest/entry/ID
        # retrieve the entry by ID
        pass

    def update(self, id):
        # PUT /blog/rest/entry/ID
        # update the entry by ID
        pass

    def remove(self, id)
        # DELETE /blog/rest/entry/ID
        # delete the entry by ID
        pass
```

Note: The `remove` method is actual the replacement of the `delete` method but to minimize the need of users to call the parent/ancestors version of the overridden method, the `delete` method is tended to be left untouched where the deleting implementation should be placed in the `remove` method.

Customize Error Page

There are types of custom error pages for normal controllers and error controllers where any custom error pages will receive three variables: `message`, `code` (HTTP Response Code) and `debug_info` (the text version of stack trace).

Custom Error Pages for Unattended Exceptions

When exceptions are raised unexpectedly, to handle the exceptions not handled by normal controllers, you need something similar to the following code.

```
@custom_error('error.html')
@renderer('app.view')
class ErrorController(BaseErrorController): pass
```

Then, add a single `<error>` tag under the `<server>` tag. For example,

```
<?xml version="1.0" encoding="utf-8"?>
<application>
    <!-- ... -->
    <server>
        <!-- ... -->
        <error>app.controller.ErrorController</error>
        <!-- ... -->
    </server>
    <!-- ... -->
</application>
```

Controller-specific Custom Error Pages

When exceptions are raised on a normal controller (e.g., any controller based on `tori.controller.Controller` and `tori.controller.RestController`), what you need is just add the decorator `tori.decorator.controller.custom_error()` to the controller. For example,

```
@custom_error('error.html')
@renderer('web.views')
class HomeController(Controller):
    # Assuming something
    pass
```

References

For more information, please read

- *Templates* (Manual)
- *tori.controller*
- *tori.decorator.controller*
- *Template Engine Modules* (API)
- *Session API*

1.1.7 Nest

Nest is a command-line script to help you quickly setup an app container. By default, it will be installed on under `/usr/local/bin` for most of the system. Run `nest -h` for more information. New in version 2.1.2.Deprecated since version 2.1.2.

Warning: Nest only works on Python 2.6 and 2.7. It would be fixed in the future release.

1.1.8 Templates

Tori Framework uses Jinja2 as the default template engine. It is to minimize the incompatibility between the syntax of the famous Django framework and the irregular syntax of Tornado's default template engine in case of porting code and reduce the learning curve.

1.1.9 Web Socket

The implementation of Web Socket in Tori Framework incorporates Tornado's Web Socket Handler with Tori's cookie-based Session Controller, which is pretty much like working with *Controller*.

Here is an example.

Suppose I want to create a message-relay module

```
from council.common.handler import WSRPCInterface
# where WSRPCInterface inherits from tori.socket.rpc.Interface

class MathAPI(WSRPCInterface):
    def add(self, a, b):
        return a + b
```

Then, the client just has to send the message in JSON format.

```
{
    "id":     12345
    "method": "add"
    "data": {
        "a": 1,
        "b": 2
    }
}
```

Then, the server will reply with.

```
{
    "id":      12345
    "result": 3
}
```

See More

- *Web Socket* (Reference)

1.2 API Reference

Author Juti Noppornpitak <jnopporn@shiroyuki.com>

This section is all about the reference for Tori API.

1.2.1 tori.common

Author Juti Noppornpitak

This package contains classes and functions for common use.

class `tori.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 `tori.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.

1.2.2 tori.controller

Author Juti Noppornpitak

This package contains an abstract controller (based on `tornado.web.RequestHandler`) and built-in controllers.

class `tori.controller.`**`Controller`**`(`**args*, ***kwargs*`)`

The abstract controller for Tori framework which uses Jinja2 as a template engine instead of the default one that comes with Tornado.

component (*name*, *fork_component=False*)
Get the (re-usable) component from the initialized Imagination component locator service.

Parameters

- **name** – the name of the registered re-usable component.
- **fork_component** – the flag to fork the component

Returns module, package registered or `None`

render (*template_name*, ***contexts*)
Render the template with the given contexts and push the output buffer.

See `tori.renderer.Renderer.render()` for more information.

render_template (*template_name*, ***contexts*)
Render the template with the given contexts.

See `tori.renderer.Renderer.render()` for more information.

session
Session Controller

Return type tori.session.controller.Controller

template_engine
Template Engine

Return type tori.template.renderer.Renderer

class `tori.controller.`**ErrorController** (**args*, ***kwargs*)
Generates an error response with status_code for all requests.

class `tori.controller.`**ResourceService** (**args*, ***kwargs*)
Resource service is to serve a static resource via HTTP/S protocol.

static add_pattern (*pattern*, *base_path*, *enable_cache=False*)
Add the routing pattern for the resource path prefix.

Parameters

- **pattern** – a routing pattern. It can be a Python-compatible regular expression.
- **base_path** – a path prefix of the resource corresponding to the routing pattern.
- **enable_cache** – a flag to indicate whether any loaded resources need to be cached on the first request.

get (*path=None*)
Get a particular resource.

Parameters **path** – blocks of path used to composite an actual path.

Note: This method requires refactoring.

class `tori.controller.`**RestController** (**args*, ***kwargs*)
Abstract REST-capable controller based on a single primary key.

create ()
Create an entity.

delete (*id=None*)
Handle DELETE requests.

get (*id=None*)
Handle GET requests.

list ()
Retrieve the list of all entities.

post (*id=None*)
Handle POST requests.

put (*id=None*)
Handle PUT requests.

remove (*id*)
Remove an entity with *id*.

retrieve (*id*)
Retrieve an entity with *id*.

update (*id*)
Update an entity with *id*.

1.2.3 tori.decorator.common

Author Juti Noppornpitak

This package contains decorators for common use.

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

`tori.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`.

`tori.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.

tori.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.

1.2.4 tori.decorator.controller

Author Juti Noppornpitak

This package contains decorators for enhancing controllers.

tori.decorator.controller.**custom_error**(*template_name*, ***contexts*)

Set up the controller to handle exceptions with a custom error page.

Note: This decorator is to override the method `write_error`.

Parameters

- **template_name** (*string*) – the name of the template to render.
- **contexts** (*dict*) – map of context variables

`tori.decorator.controller.`**`renderer`**`(`**args*, ***kwargs*`)`
Set up the renderer for a controller.

See `tori.template.renderer.Renderer` for more information.

1.2.5 tori.exception

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Please note that this is a general exception.

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

1.2.6 Navigation APIs

Author Juti Noppornpitaks

Purpose Internal Use Only

The navigation module is designed specifically for the dependency-injectable Application.

Please note that the term *DOMElement* used on this page denotes any of `yotsuba.kotoba.Kotoba`, `yotsuba.kotoba.DOMElements` and `yotsuba.kotoba.DOMElement`.

Additionally, the parameter *route* for any methods mentioned on this page is an instance of *DOMElement*.

class `tori.navigation.`**`DynamicRoute`**`(`*route*`)`
Dynamic route based on class Route handled by a controller.

`controller``()`
Get the controller.

`to_tuple``()`
Convert the route to tuple.

class `tori.navigation.`**`RelayRoute`**`(`*route*`)`
Relay routing directive based on `Route` used for redirection

`destination``()`
Get the relaying destination.

`is_permanent``()`
Check whether the relay route is permanent.

`to_tuple``()`
Convert the route to tuple.

class `tori.navigation.`**`Route`**`(`*route_data*`)`
The abstract class representing a routing directive.

Parameters **route** – an instance of `kotoba.kotoba.Kotoba` representing the route.

`bean_class``()`
Get the class reference for the route.

Return type type

static **`get_pattern`**`(`*route_data*`)`
Get the routing pattern for a given *route*.

static **`get_type`**`(`*route_data*`)`
Get the routing type for a given *route*.

`source``()`
Get the original data for the route.

Return type str

type()
Get the routing type.

Return type str

class tori.navigation.**RoutingMap**
Routing Map

export()
Export the route map as a list of tuple representatives.

Return type list

find_by_pattern(*routing_pattern*)
Get the route by *routing_pattern* where it is a string.

static make(*configuration*, *base_path=None*)
Make a routing table based on the given *configuration*.

Parameters **base_path** – is an optional used by :method *Route.make*:.

register(*route*, *force_action=False*)
Register a *route*.

class tori.navigation.**StaticRoute**(*route*, *base_path*)
Static routing directive based on Route handled by a resource controller

Parameters **base_path** – is a string indicating the base path for the static resource.

cache_enabled()
Check whether the caching option is enabled.

location()
Get the location of the static resource/content.

service()
Get the resource service.

to_tuple()
Convert the route to tuple.

1.2.7 Template Engine Modules

Author Juti Noppornpitak

This package is used for rendering.

class tori.template.renderer.**DefaultRenderer**(**referers*)
The default renderer with Jinja2

Parameters **referers** – the template module path (e.g., com.shiroyuki.view) or multiple base paths of Jinja templates based on the current working directory.

For example:

```
# Instantiate with the module path.
renderer = DefaultRenderer('app.views')

# Instantiate with multiple base paths of Jinja templates.
renderer = DefaultRenderer('/opt/app/ui/template', '/usr/local/tori/module/template')
```

render(*template_path*, ***contexts*)
See Renderer.render() for more information.

class `tori.template.renderer.`**`Renderer`**`(`**args*, ***kwargs*`)`

The abstract renderer for Tori framework.

> **Warning:** This is a non-working renderer. To use the built-in renderer (with Jinja2), try `DefaultRenderer`. Otherwise, you should be expecting `tori.exception.FutureFeatureException`.

`render``(`*template_path*, ***contexts*`)`

Render a template with context variables.

Parameters

- **template_path** (*string or unicode*) – a path to the template
- **contexts** – a dictionary of context variables.

Return type string or unicode

Example:

```
renderer = Renderer()
renderer.render('dummy.html', appname='ikayaki', version=1.0)
```

Author Juti Noppornpitaks

Restriction Internal Use Only

class `tori.template.repository.`**`Repository`**`(`*class_reference*`)`

The template repository used by Rendering Service.

Parameters **class_reference** (*tori.template.service.RenderingService*) – class reference

> **Note:** This class is designed as a strict-type collection and may be refactored to the common area later on.

`get``(`*renderer_name*`)`

Retrieve the renderer by name.

Parameters **renderer_name** (*string or unicode*) – the name of the renderer.

Return type tori.template.renderer.Renderer

`set``(`*renderer*`)`

Register the renderer.

Returns self

Author Juti Noppornpitak

Restriction Internal Use Only

This package contains the rendering service. This is a module automatically loaded by `tori.application.Application`.

class `tori.template.service.`**`RenderingService`**`(`*renderer_class=<class 'tori.template.renderer.Renderer'>*, *repository_class=<class 'tori.template.repository.Repository'>*`)`

The rendering service allows the access to all template repositories.

This acts as a controller.

Parameters

- **renderer_class** (*tori.template.renderer.Renderer*) – a class reference of a renderer

- **repository_class** (*tori.template.repository.Repository*) – a class reference of a template repository

register(*renderer*)
Register a renderer.

Parameters **renderer** (*tori.template.renderer.Renderer*) – the renderer

Returns `self`.

render(*repository_name*, *template_path*, ***contexts*)
Render a template from a repository *repository_name*.

As this method acts as a shortcut and wrapper to the actual renderer for the given repository, see `tori.template.renderer.Renderer.render()` for more information.

Return type string

use(*repository_name*)
Retrieve the renderer by name

Parameters **repository_name** (*str*) – the name of the repository

Return type tori.template.renderer.Renderer

1.2.8 Session API

Author Juti Noppornpitak

This package contains the session controller used with the web controller and socket handler.

class `tori.session.controller.`**Controller**
A session controller for the controller (request handler).

delete(*key*)
Delete the data :param key: data key :type key: str

get(*key*)
Retrieve the data

Parameters **key** (*str*) – data key

Returns the data stored by the given key

id
Administrated Session ID

Returns str

reset()
Clear out all data of the administrated session

set(*key*, *content*)
Define the data

Parameters

- **key** (*str*) – data key
- **content** – data content

1.2.9 Web Socket

Generic Web Socket Module

Author Juti Noppornpitak

Status Stable

Last Update November 29, 2013

class `tori.socket.websocket.`**`WebSocket`**(**args*, ***kwargs*)
Web Socket Handler with extension to session controller

`component`(*name*, *fork_component=False*)
Get the (re-usable) component from the initialized Imagination component locator service.

Parameters

- **name** – the name of the registered re-usable component.
- **fork_component** – the flag to fork the component

Returns module, package registered or `None`

`session`
Session Controller

Return type tori.session.controller.Controller

Remote Procedure Call Module

Author Juti Noppornpitak

Status Stable/Testing

Last Update November 29, 2013

class `tori.socket.rpc.`**`Interface`**(**args*, ***kwargs*)
Remote Interface

Extends from `tori.socket.websocket.WebSocket`

`on_message`(*message*)

The parameter `message` is supposed to be in JSON format:

```
{
    ["id":      unique_id,]
    ["service": service_name,]
    ["data":    parameter_object,]
    "method":  method_name
}
```

When the service is not specified, the interface will act as a service.

class `tori.socket.rpc.`**`Remote`**(*method*, *id=None*, *data=None*, *service=None*)
RPC Request

Parameters

- **method** (*str*) – the name of the method
- **id** – the request ID (default with unix timestamp)

- **data** (*dict*) – method parameters
- **service** (*str*) – the ID of the registered component/service (optional)

call()

Execute the request

Returns the result of the execution

class tori.socket.rpc.**Response**(*result*, *id*)

RPC Response

Parameters

- **result** – the result from RPC
- **id** – the response ID

1.2.10 Database APIs

Tori Framework only provides the object-relational mapping interface for MongoDB databases via PyMongo.

tori.db.common

tori.db.criteria

class tori.db.criteria.**Criteria**

Criteria

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

build_cursor(*repository*, *force_loading=False*, *auto_index=False*)

Build the cursor

Parameters

- **repository** (*tori.db.repository.Repository*) – the repository
- **force_loading** (*bool*) – force loading on any returned entities
- **auto_index** (*bool*) – the flag to automatically index sorting fields

Note: This is mainly used by a repository internally.

limit(*limit*)

Define the filter limit

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

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

`where`(*key_or_full_condition*, *filter_data=None*)
Define the condition

Parameters

- **key_or_full_condition** (*str or dict*) – either the key of the condition (e.g., a field name, $or, $gt etc.)
- **filter_data** – the filter data associating to the key

class `tori.db.criteria.`**`Order`**
Sorting Order Definition

`ASC`
Ascending Order

alias of `ASCENDING`

`DESC`
Descending Order

alias of `DESCENDING`

tori.db.entity

tori.db.exception

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

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

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

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

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

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

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

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

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

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

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

tori.db.fixture

Warning: This feature is added in 2.1 but neither tested nor supported in 2.1.

Author Juti Noppornpitak

class `tori.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'
        }
    }
)
```

tori.db.manager

tori.db.mapper

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

class `tori.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 `tori.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 `tori.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 `tori.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 `tori.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

`tori.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.

`tori.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

tori.db.repository

tori.db.session

tori.db.uow

CHAPTER 2

Not working after upgrade?

File a bug at Tori on GitHub.

CHAPTER 3

What if the documentation is suck or the code is buggy?

If the document is unclear or missing or needs improvement, please help us by contributing to the codebase of Tori on GitHub.

CHAPTER 4

Special Thanks

This project is not possible without helps and guidance from Guilherme Blanco from Doctrine (PHP).

CHAPTER 5

Indices and Modules

- *genindex*
- *modindex*

Python Module Index