Tori Documentation

Release 3.1.0

Juti Noppornpitak

August 15, 2016

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.)

How to Install?

See **Installation** from Getting Started.

What's next?

Read:

- Manual to get started and learn how to use it.
- API Reference to get the reference on APIs and methods.
- Change Logs to see what have been changed.

2.1 Manual

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

2.1.1 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.

2.1.2 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 submodules 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.

2.1.3 Getting Started

Installation

Just run `sudo pip install tori`

Hello, world... again?

Set up for Python 3.3+

Please install Tori with **Python 2.7** and use `nest` to create the skeleton app for now as the command due a known compatibility issue.

Set up for Python 2.7

After the installation, you can begin creating a app with the supplied `nest` command. In this example, we create an app called "konota web".

```
nest tori.app.create -p 5000 konataweb
```

What just happened?

- The command generate a basic based on an app skeleton **at the working directory**.

- The option `-p` sets the default port to 5000.

At your current directory, you should see:

```
(app root path)
    config/             <- config folder
        dev.xml         <- app config (routing)
        service.xml     <- config for service containers
        settings.json   <- app settings
    konataweb/          <- app module
        controller.py   <- dummy controller
    Makefile
    server.py           <- Bootstrap file
    static/             <- Just like Flask
        js/     (empty)
        image/  (empty)
        css/    (empty)
        scss/   (empty)
    templates           <- Just like Flask
        home.html       <- dummy template
```

If you take a look at `konataweb/controller.py`, you will see:

```python
from tori.controller import Controller

class Home(Controller):
    def get(self):
        self.render('home.html', name = 'konataweb')
```

where the base path of the template folder is at `templates`.

Note: The template engine is Jinja2.

Note: You can re-define the base path of your template either by module (e.g., *konataweb.templates* or `/opt/templates`). For example, if you happened to have a template in `konataweb/templates`, you can re-define by:

1. import a decorator.

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

2. decorate the controller to re-define the base path.

```python
@renderer('konataweb.templates')
class Home(Controller):
    ...
```

Run `make service` to start the web service (not in the background). You should now be able to access to http://localhost:5000.

What is a service container?

In Tori Framework, you may define global variables for reusability. This part of the framework is relied on Project Imagination (see more information from the documentation).

For example, if we want to create a container (or known as **entity** in Project Imagination) to do some calculation, first create `konataweb.calculator.py`.

```python
class EasyCalculator(object):
    def sum(self, *items):
        summation = 0

        for item in items:
            summation = item


        return item
```

Then, in `config/service.xml`, just define an **entity** tag for a container under `<imagination>`.

```xml
<entity id="easycalc" class="konataweb.calculator.EasyCalculator"/>
<!-- You may define more than one container of the same class -->
<entity id="different_easycalc" class="konataweb.calculator.EasyCalculator"/>
```

To use the container in the controller or websocket handler, you can simply retrieve the global instance of the container **easycalc** by calling *self.component*.

```python
# In konataweb/controller.py
import re
class CalculatorAPI(Controller):
    def get(self, operation):
        raw_nums = self.get_argument('num_sequence', '') # tornado.web.RequestHandler's original
        numbers  = [int(str_num) for str_num in re.split(',', raw_nums)]

        if operation != 'sum':
            return self.set_status(405) # tornado.web.RequestHandler's original

        sum = self.component('easycalc').sum(*numbers) # tori.controller.Controller's extra

        self.finish(sum)
```

Just now, we happen to have a new controller. We need to make it accessible.

Add a route

To add a new route, just add a `<controller>` tag under `<routes>`.

```xml
<controller id="api.calculator" class="konataweb.controller.CalculatorAPI" pattern="/api/{operation}"
```

You should see the following result after you send a GET request to http://localhost:5000/api/sum?num_sequence=1,3,5,7:

```
16
```

Application Settings (NEW since 3.0)

Instead of overriding the service container **session**, you achieve the same thing by defining the section **session**. For example, we change to use the file-based session.

```json
{
    "session": {
        "class": "tori.session.repository.file.FileSessionRepository",
        "params": {
            "location": "session.json"
        }
    },
```

```
    ...
}
```

Router in the template (NEW since 3.0)

In Tori 3, you can refer to any routes by ID. For instance, we add a link to the calculator API.

```
<a href="{{ app.path('api.calculator', operation = 'sum') }}?num_sequence=1,3,5,7">Test link</a>
```

Read more

- Controller
- Object-relational Mapping (ORM)
- Routing
- Configuration

2.1.4 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 cur
# 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

Predefined Configuration

New in version 3.0.

Based on the feedback, despite of maximizing the customization, the XML configuration schema is pretty hard to work with or remember, especially for something important like session management and databases (MongoDB).

In version 3.0, it provides the predefined configuration (in JSON format). Here is the sample of all available configuration.

```
{
    "session": {
```

```
        "class": "tori.session.repository.memory.Memory"
        "params": {}
    }
    "db": {
        "managers": {}
    }
}
```

Database Configuration ("db")   This section currently can only tell the entity manager factory from Passerine ORM (https://github.com/shiroyuki/passerine) to automatically prepare for the connection to

For example, we have `config.json`.

```
{
    "db": {
        "managers": {
            "directory": "mongodb://localhost/directory"
        }
    }
}
```

Add this line to the XML configuration file.

```
<use src="config.json"/>
```

And you can call the service by either:

```
from tori.centre import services

services.get('db.directory') # to get the entity manager
```

Session Configuration   The following are usable session adapters.

tori.session.repository.memory.Memory   This is the default option. This uses the process memory as a storage. **No parameters.**

tori.session.repository.file.File   This adapter uses a single JSON file to store session data. It writes to the file on every save.

Parameters:

> **param str location** The location of the file. If the given location is a relative path, the base path will be based from where the main script is.

tori.session.repository.xredis.Redis   This adapter uses a single JSON file to store session data. It writes to the file on every save.

> **param str prefix** The key prefix for all session entries. By default, the prefix is set to **tori/session**.
>
> **param redis_client** The redis connection client from **redis** (python package). By default, it is set to a connection client bounded to **localhost** without credential.
>
> **param bool use_localhost_as_fallback** The flag to use **localhost** as a fallback connection. It is set to use this feature by default.

Your own adapter?   Just extends your adapter from `tori.session.repository.base.Base`

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 pattern* (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	Equvalent 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.

2.1.5 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.
- Some database operations are not supported or optimized due to the non-generalized interfaces as shown on the table in the next section. (Introduced in Tori 3.0)
- **LevelDB** will only be supported for **Python 2.7** as the underlying library leveldb only supports Python 2.7 due to its dependency.

Supported SQL-equivalent Querying Operations

New in version 3.0.

SQL-equivalent Operation	MongoDB 2.4+	Riak 1.4+	Riak 2.0+	LevelDB	Redis
CRUD operations	Yes	Yes	Yes	Yes	Yes
Simple query	Yes	No	Unknown	No	No
AND compound statement	Yes	No	Unknown	No	No
OR compound statement	Yes/SW	Yes/SW	Yes/SW	No	No
Filter with regular expression	Yes	No	Unknown	No	No
Range filter	Yes	No	Unknown	No	No
Query optimization with index	Yes	Yes	Yes	No	No
Directly use indice for query	No	Yes	Yes	No	No
Store the data as they are*	Yes	Yes	Yes	No	No

Note: Some databases may store a complex-structured data, which is always the case when the ORM stores the structured data of the entity.

Note: New in version 3.1.

The OR statements are not available by any of supported NoSQL databases. This is planned for Tori 3.1 to support by the ORM, similar to **session**.

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.

```python
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, team=None):
        self.name = name
        self.team = None

@entity
class Team(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.

```python
from tori.db.manager import ManagerFactory

manager_factory = ManagerFactory()
manager_factory.set('ff_game', 'mongodb://db_host/db_name')

entity_manager = manager_factory.get('ff_game')
```

Basic Usage

Suppose we have:

```python
session   = entity_manager.open_session()
char_repo = session.repository(Character)
```

Create a new entity

Suppose two characters: "Ramza", and "Alma", are to created.

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

character_repo.post(ramza)
character_repo.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),

```python
query      = char_repo.new_criteria('c')
characters = char_repo.find(query)

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

Then, you should see:

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

Now, to find "Ramza",

```python
query = char_repo.new_criteria('c')
query.expect('c.name = :name')
query.define('name', 'Ramza')

characters = char_repo.find(query)

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

Then, you should only see:

```
1: Ramza
```

Note: The queries use a simple query language. (If you see this message and see no explaination on the query language, please contact @shiroyuki on Twitter.)

Retrieve an entity by ID

Now, to retrieve an entity by ID,

```
alma = char_repo.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

```
char_repo.put(character)
```

Delete entities

```
char_repo.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:

```python
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`.

```python
@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

```python
@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
```

```python
@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 Implemented 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

How to make a join query

New in version 3.0.

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)
```

Warning: In Tori 3.0, we only test for one-to-one and many-to-one relationships.

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	No plan at the moment
Detach	No plan at the moment

Example

First, define the entity manager.

```python
from tori.db.manager import ManagerFactory

manager_factory = ManagerFactory()
manager_factory.set('default', 'mongodb://db_host/db_name')

entity_manager = manager_factory.get('default')
```

Then, open a session:

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

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

```python
repo = session.collection(User)

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

bob = repo.find(query)
```

```
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()
```

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

2.1.6 Console and CLI Framework

New in version 2.2.

Setup

Similar to how you set up a server (see Getting Started), you need to add `tori.cli.console.Console` into the mix, for instance, we have a script name nep

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
Nameless Education Platform

.. codeauthor:: Juti Noppornpitak <jnopporn@shiroyuki.com>
"""
from tori.application import Application
from tori.cli.console import Console
from tori.cli.exception import TerminationSignal

app     = Application('config/app.xml')
console = Console('NEP')
```

```
console.load(app);

try:
    console.handle()
except TerminationSignal as e:
    pass
```

where you can see the list of all registered commands by executing *nep*.

Configuration

As commands are treated as reusable components (with Imagination Framework), they must be defined first with tag "command" and then any thing with prefix "command:". For example,

```
<!-- From https://github.com/nepteam/nep -->
<entity
    id="command.db"
    class="neptune.command.Database"
    tags="command command:db">
    <param name="db" type="entity">db</param>
    <interception before="me" do="execute" with="init"/>
    <interception after="me" do="execute" with="clean_up"/>
</entity>
```

Then, the command will be referenced with anything after ":". From the previous example, the command "command.db" will be referred as "db" and executed as:

```
./nep db -d # in this example, this command is to reset the databases.
```

Implement Commands

Just write a class extending from *tori.cli.command.Command*.

There are two methods that mush be overridden:

define_arguments(*argument_parser*)
>   Define the arguements. Override the method with the keyword *pass* if there is no argument to define.
>
>   Parameters **argument_parser** (*argparse.ArgumentParser*) – the argument parser
>
>   For more information on how to define the arguments, see http://docs.python.org/3.3/library/argparse.html.

execute(*args*)
>   Execute the command.
>
>   Parameters **args** (*argparse.Namespace*) – the arguments

2.1.7 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**

```python
# 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.

```python
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.

```python
@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

```python
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

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

where `web.controller.BlogEntryRestController` is

```python
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
<?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

2.1.8 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: The option `--dry-run` is added to prevent the command from performing any write operations, including running **pip**.

Deprecated since version 2.1.2: The prompt for UI library installation is removed in flavour to other package managers or any automation scripts like **puppet** and **chef**.

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

2.1.9 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.

2.1.10 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

```python
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.

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

Then, the server will reply with.

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

See More

- Web Socket (Reference)

2.2 API Reference

Author Juti Noppornpitak <jnopporn@shiroyuki.com>

This section is all about the reference for Tori API.

2.2.1 tori.cli

class `tori.cli.command.`**`Command`**
> Abstract class for all Tori-based commands

> **`define_arguments`**(*argument_parser*)
> > Define the arguments

> **`execute`**(*args*)
> > Execute the command

class `tori.cli.console.`**`Console`**(*namespace=None*)
> Main Console

> The commands must be defined first with tag "command" and then any thing with prefix "command:". For example,

```
<!-- From https://github.com/nepteam/nep -->
<entity
    id="command.db"
    class="neptune.command.Database"
    tags="command command:db">
    <param name="db" type="entity">db</param>
    <interception before="me" do="execute" with="init"/>
    <interception after="me" do="execute" with="clean_up"/>
</entity>
```

> Then, the command will be referenced with anything after ":". From the previous example, the command "command:db" will be referred as "db".

exception `tori.cli.exception.`**`CommandNotFound`**
> Command Not Found

exception `tori.cli.exception.`**`InterfaceException`**
> Interface Exception. Require implementation

exception `tori.cli.exception.`**`NotConfigured`**
> Not-configured exception

exception `tori.cli.exception.`**`TerminationSignal`**
> Termination Signal

2.2.2 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.

2.2.3 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`

> **`redirect_to`**(*route_id*, *params={}*, *full_url=False*)
>> Redirect to the path by ID

>> **Parameters**

>>> • **`str`** (`id`) – the path ID
>>>
>>> • **`params`** (`dict`) – the variables used in the routing pattern
>>>
>>> • **`full_url`** (`bool`) – option to provide full URL to the server.

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

>> See `tori.renderer.Renderer.render()` for more information on the parameters.

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

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

resolve_route(*route_id*, *params={}*, *full_url=False*)
Resolve the path by ID

> **Parameters**
>
> > - **str** (`id`) – the path ID
> >
> > - **params** (`dict`) – the variables used in the routing pattern
> >
> > - **full_url** (`bool`) – option to provide full URL to the server.

session
Session Controller

> **Return type** *tori.session.controller.Controller*

template_engine
Template Engine

> **Return type** *tori.template.renderer.Renderer*
>
> **Raises** **RenderingSourceMissingError** – if the template base path and the reverse core reference are not defined.

Changed in version 3.0: The exception will not be raised if the reverse core reference is defined.

Note: The reverse core reference is the first instance of `tori.application.Application` in the process.

---

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(*key=None*)
> Handle DELETE requests.

get(*key=None*)
> Handle GET requests.

list()
> Retrieve the list of all entities.

post(*key=None*)
> Handle POST requests.

put(*key=None*)
> Handle PUT requests.

remove(*key*)
> Remove an entity with *key*.

retrieve(*key*)
> Retrieve an entity with *key*.

update(*key*)
> Update an entity with *key*.

class tori.controller.**SimpleController**(**args*, ***kwargs*)
> Simplified Request Controller

2.2.4 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.

2.2.5 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.

2.2.6 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.

2.2.7 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*.

resolve(*id*, ***params*)
> Resolve the path by ID
>
> > **Parameters**
> >
> > - **str** (*id*) – the path ID
> >
> > - **params** – the variables used in the routing pattern

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.

2.2.8 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*

2.2.9 Session API

> **Author** Juti Noppornpitak

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

class `tori.session.controller.`**Controller**(*session_repository*, *id*)
> 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

2.2.10 Web Socket

Generic Web Socket Module

> **Author** Juti Noppornpitak

> **Status** Stable

> **Last Update** August 15, 2016

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** August 15, 2016

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

2.2.11 Database APIs

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

tori.db.common

> **Author** Juti Noppornpitak <jnopporn@shiroyuki.com>
>
> **Stability** Stable

class tori.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** (`tori.db.session.Session`) – the managed session
> >
> > - **origin** (*object*) – the origin of the association
> >
> > - **guide** (`tori.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 *tori.db.session.Session.refresh()* on the owned object instead.

class tori.db.common.**ProxyFactory**
> Proxy Factory

> This factory is to create a proxy object.

> **Parameters**

>> • **session** (tori.db.session.Session) – the managed session

>> • **id** – the object ID

>> • **mapping_guide** (tori.db.mapper.RelatingGuide) – the relational guide

class tori.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** (tori.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 tori.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 tori.db.common.**Serializer** (*max_depth=2*, *mode='forgiven'*)
> 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

tori.db.criteria

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

ASC
        Ascending Order

        alias of `ASCENDING`

DESC
        Descending Order

        alias of `DESCENDING`

class `tori.db.criteria.`**`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,

```
```

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** *tori.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

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

    Deprecated since version 3.1: Starting in Tori 3.0, the new way to query will be.

> **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

tori.db.driver

class tori.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** **tori.db.criteria.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** (`tori.db.expression.ExpressionPart`) – the left expression
> >
> > - **right** (`tori.db.expression.ExpressionPart`) – the right expression
> >
> > - **operand** – the native operand
>
> > **Raises** `NotImplemented` – only if the interface is not overridden.

class tori.db.driver.interface.**DriverInterface**(*config*, *dialect*)
> The abstract driver interface

> > **Parameters**
> >
> > - **config** (`dict`) – the configuration used to initialize the database connection / client
> >
> > - **dialect** (`tori.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 tori.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 tori.db.driver.interface.**QuerySequence**
> Driver Query Sequence

> The collection represents the sequence of sub queries.

> **add** (*iteration*)
> > Append the the iteration

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

> **each** ()
> > Get the sequence iterator.

exception tori.db.driver.mongodriver.**InvalidExpressionError**
> MongoDB-specific Invalid Expression Error

exception tori.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).

tori.db.entity

Author Juti Noppornpitak <jnopporn@shiroyuki.com>

class `tori.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 `tori.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 `tori.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.

`tori.db.entity.`**`entity`**(**args*, ***kwargs*)
Entity decorator

>   **Parameters** **collection_name** (`str`) – the name of the collection
>
>   **Returns** the decorated object
>
>   **Return type** object

`tori.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,

```python
@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')`.

---

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.`**`InvalidUrlError`**
    Invalid DB URL Error

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.

exception `tori.db.exception.`**`UnknownDriverError`**
    Unknown Driver Error

exception `tori.db.exception.`**`UnsupportedRepositoryReferenceError`**
    Unsupported Repository Reference Error

tori.db.expression

class `tori.db.expression.`**`Criteria`**
    Expression Criteria

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

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

        Parameters

            • **left** (`tori.db.expression.ExpressionPart`) – the left part

            • **right** (`tori.db.expression.ExpressionPart`) – the right part

            • **operand** (`str`) – the generic operand

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

exception `tori.db.expression.`**`InvalidExpressionError`**
    Generic Invalid Expression Error

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

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

Entity Manager

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

close_session(*id_or_session*)
> 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** *tori.db.driver.interface.DriverInterface*

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.

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.metadata

class `tori.db.metadata.entity.`**EntityMetadata**
> Entity Metadata

> **cls**
> > Entity Class

> **collection_name**
> > Collection / Bucket / Table Name

> **index_list**
> > Index List

> **relational_map**
> > Relational Map

class `tori.db.metadata.helper.`**EntityMetadataHelper**
> Entity Metadata Helper

> **static extract**()
> > Extract the metadata of the given class
> >
> > > **Parameters** **cls** (`type`) – the entity class
> > >
> > > **Return type** *tori.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

tori.db.repository

> **Author** Juti Noppornpitak <jnopporn@shiroyuki.com>
>
> **Status** Stable

class `tori.db.repository.`**`Repository`**(*session*, *representing_class*)

> Repository (Entity AbstractRepository) for Mongo DB
>
> > **Parameters**
> >
> > - **`session`** (`tori.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`** (`tori.db.criteria.Query`) – the search criteria
> > >
> > > **Return type** int
>
> **`find`**(*criteria*, *force_loading=False*)
>
> > Find entity with criteria
> >
> > > **Parameters**
> > >
> > > - **`criteria`** (`tori.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, tori.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** *tori.db.criteria.Query*

session
> Session
>
> > **Return type** *tori.db.session.Session*

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

tori.db.session

class `tori.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()
> Flush all changes of the session.

persist(**entities*)
> Persist entities
>
> > **Parameters entities** (`type of list of type`) – one or more entities

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** *tori.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 *tori.db.repository.Repository*

tori.db.uow

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

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

class tori.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.

---

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

2.3 Change Logs

Code	Definition
BCB-x.y	Backward-compatibility breakage caused by the marked features from version `x.y`

2.3.1 Version 3.1

Release Date TBA

- [Planned] Possibly removed `tori.db.session.Session.register_class(...)`.
- [Planned] Switch from **tori.db** to **Passerine ORM**.

2.3.2 Version 3.0

Release Date 2014.11.23

Note: **tori.db** has been spinned off as project **Passerine ORM** (https://github.com/shiroyuki/passerine). Tori 3.0 only contains the testing version of **Passerine ORM**. The documentation for **Passerine** (http://passerine-orm.readthedocs.org/) is compatible with **tori.db**.

- **ORM/tori.db**: Allow cross-collection (or cross-repository) queries within the same type of backend datastores.
- **ORM/tori.db**: (**BCB-2.1**) Removed the silly preconditions of the setup of ORM.
- **ORM/tori.db**: (**BCB-2.1**) The setup of ORM becomes more generic in order to support multiple drivers.
- **ORM/tori.db**: (**BCB-2.1**) No auto indexing.
- **ORM/tori.db**: (**BCB-2.1**) The query mechanism is changed to reduce the direct access to PyMongo APIs directly. It will be a **BCB** if the code that uses `tori.db.criteria.Criteria` instantiates the class directly.
- **ORM/tori.db**: (**BCB-2.1**) Class **Criteria** has been renamed to **Query** as the internal class will be labeled as **Criteria**. This change is to address the semantic / readability issue. (Hence, all references to Criteria objects are now referred to Query objects.)
- **ORM/tori.db**: Removed unused / tedious code from the ORM.
- **Web Framework**: (**BCB-2.1**) The **simple routing scheme** is now default instead of the **regular expression** originally used by Tornado. (The router class will take care of the translation.)
- **Web Framework**: The first instance of `tori.application.Application` is now self-referenced as `tori.centre.core`.
- **Web Framework**: Add a file-base session repository. This allows the app to store the session data as a json file.
- **Web Framework**: Without specifying the rendering path for each controller, the controller will be looking for templates from `<app_base_path>/templates`.
- **Web Framework**: Introduce `/manual/configuration/predefined-config.rst`. (The old style will be deprecated in 3.2.)
- **Tests**: Reorganized the tests and refactored the ORM tests.

Not working after upgrade?

File a bug at Tori on GitHub.

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.

Special Thanks

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

Indices and Modules

- genindex

- modindex

t

A

add() (tori.db.driver.interface.QuerySequence method), 48

add_pattern() (tori.controller.ResourceService static method), 34

apply_relational_map() (tori.db.session.Session method), 58

ASC (tori.db.criteria.Order attribute), 45

AssociationFactory (class in tori.db.mapper), 53

AssociationType (class in tori.db.mapper), 54

AUTO_DETECT (tori.db.mapper.AssociationType attribute), 54

auto_index() (tori.db.repository.Repository method), 57

B

BaseDecoratorForCallableObject (class in tori.decorator.common), 35

BasicAssociation (class in tori.db.entity), 49

BasicGuide (class in tori.db.mapper), 54

bean_class() (tori.navigation.Route method), 38

C

cache_enabled() (tori.navigation.StaticRoute method), 39

call() (tori.socket.rpc.Remote method), 43

CascadingType (class in tori.db.mapper), 54

class_name (tori.db.mapper.AssociationFactory attribute), 53

client (tori.db.driver.interface.DriverInterface attribute), 47

close_session() (tori.db.manager.Manager method), 52

cls (tori.db.mapper.AssociationFactory attribute), 53

cls (tori.db.metadata.entity.EntityMetadata attribute), 56

collection() (tori.db.driver.interface.DriverInterface method), 47

collection() (tori.db.session.Session method), 58

collection_name (tori.db.mapper.AssociationFactory attribute), 53

collection_name (tori.db.metadata.entity.EntityMetadata attribute), 56

Command (class in tori.cli.command), 32

CommandNotFound, 32

component() (tori.controller.Controller method), 33

component() (tori.socket.websocket.WebSocket method), 42

config (tori.db.driver.interface.DriverInterface attribute), 47

connect() (tori.db.driver.interface.DriverInterface method), 47

Console (class in tori.cli.console), 32

Controller (class in tori.controller), 33

Controller (class in tori.session.controller), 41

controller() (tori.navigation.DynamicRoute method), 38

count() (tori.db.repository.Repository method), 57

create() (tori.controller.RestController method), 34

Criteria (class in tori.db.expression), 51

criteria (tori.db.criteria.Query attribute), 45

custom_error() (in module tori.decorator.controller), 36

D

database_name (tori.db.driver.interface.DriverInterface attribute), 47

db() (tori.db.driver.interface.DriverInterface method), 47

DefaultRenderer (class in tori.template.renderer), 40

define() (tori.db.criteria.Query method), 45

define_arguments(), 27

define_arguments() (tori.cli.command.Command method), 32

DELETE (tori.db.mapper.CascadingType attribute), 54

delete() (tori.controller.RestController method), 35

delete() (tori.db.session.Session method), 58

delete() (tori.session.controller.Controller method), 41

DependencyNode (class in tori.db.uow), 59

DESC (tori.db.criteria.Order attribute), 45

destination (tori.db.mapper.AssociationFactory attribute), 53

destination() (tori.navigation.RelayRoute method), 38

DETACH (tori.db.mapper.CascadingType attribute), 54

dialect (tori.db.driver.interface.DriverInterface attribute), 48

DialectInterface (class in tori.db.driver.interface), 46

T

U

W