defernia Documentation

Release 0.1.0

Hong Minhee

May 24, 2013

CONTENTS

Defernia is a large complex system for defining the Ernia, a fictional world. Its name is an abbreviation of *Defining Ernia*. It is also a name of a website; see http://defernia.org/ also.

CHAPTER

ONE

DEVELOPER GUIDES

1.1 Installation

1.1.1 Requirements

Defernia is made with several open source softwares. Defernia depends on the following softwares:

Python 2.6–2.7 or PyPy 1.5+ The Defernia system is mostly written in Python programming language. There's big incompatibility between Python 2.x and Python 3.x, so you must not use Python 3.0 nor more.

It works on PyPy 1.5+ as well.

PostgreSQL 8.3+ or SQLite 3+ Defernia uses RDBMS (relational databases) to store data. Recommend PostgreSQL for production use and SQLite for development-purpose.

Note: It probably works well on MySQL also, but we don't recommend it.

Any other many Python packages Defernia depends on any other many Python packages listed following, but you can be free from these libraries. Because these libraries are installed automatically.

- Mercurial
- SQLAlchemy
- Flask (Werkzeug and Jinja)

Note: How to install the above softwares from Debian/Ubuntu Linux:

```
$ apt-get install python postgresql  # production
$ apt-get install python sqlite3     # development
```

1.1.2 Easy way

The easiest way to install Defernia is just using pip. It downloads Defernia itself and resolves dependencies also automatically.

```
$ pip install hg+https://bitbucket.org/dahlia/defernia
```

1.1.3 Hard way

Defernia is an ordinary Python package that follows the standard Python distribution way. It means that you can download the Defernia source code and install it by yourself.

```
$ hg clone https://bitbucket.org/dahlia/defernia
$ cd defernia/
defernia$ python setup.py install
```

If setuptools or Distribute is installed in your system, `setup.py` must have resolved its dependencies automatically as well.

1.1.4 Using virtualenv

If you have an idea to develop Defernia (and contribute to it), virtualenv would be helpful. It helps to isolate the working Python site-packages environemt from the system global site-packages environment. If there's no installed **virtualenv** command in your system yet, install it first: (It might need a system administrator permission: use **sudo** then.)

```
$ easy_install virtualenv
```

Note: Most of Linux distributions provide **virtualenv** as package. For example, in Debian or Ubuntu like APT-based distributions you can install it like:

```
$ apt-get install python-virtualenv
```

And then, make your isolated environment for Defernia development via **virtualenv**:

```
$ virtualenv --distribute defernia-env
defernia-env$ cd defernia-env/
(defernia-env)defernia-env$ source bin/activate
```

The last command makes your command line prompt to enter the created `defernia-env` enviroment. The `(defernia-env)` prefix indicates where you are in. When you want to back from here, type **deactivate** in the prompt.

What you have to do next is to checkout the Defernia source code.

```
(defernia-env)defernia-env$ hg clone https://bitbucket.org/dahlia/defernia
(defernia-env)defernia-env$ cd defernia/
```

To install Defernia in development mode and resolve the dependencies, use **setup.py develop** subcommand instead of **setup.py install**.

```
(defernia-env)defernia$ python setup.py develop
```

It's finished. Now you can hack the Defernia system.

Note: **hg** is a command provided by Mercurial.

1.2 Getting started

If Defernia has *installed*, there might be **manage_defernia.py** command. It helps you to make a configuration, initialize a database, or run a web server.

See Also:

Script *manage_defernia.py*

1.2.1 Configuration file

Defernia is a system that supports multiple instances, and instances' metadata are stored in the configuration file. From here, we assume that our configuration filename is `instance.cfg`. (Of course, there's no such file currently.) You can name it freely like `dev.cfg` or `prod.cfg`.

You can pass a filename that doesn't exist into `--config` option, and the script will confirm would you want to create a such configuration file.

```
$ manage_defernia.py shell --config instance.cfg
instance.cfg doesn't exist yet; would you create it? [y]
```

There's some fields to be set like database URL:

Repository directory path (`REPODIR_PATH`) The directory that would contain Mercurial repositories. By default it is a directory named `repos` located in the current directory.

```
Repository directory path [/home/dahlia/defernia-env/defernia/repos]:
```

Database URL (`DATABASE_URL`) The database to be used. By default it uses SQLite with a database file (`db.sqlite`) located in the current directory.

```
Database URL [sqlite:////home/dahlia/defernia-env/db.sqlite]:
```

See Also:

SQLAlchemy — Database Urls

Secret key for secure session (`SECRET_KEY`) The HMAC secret key. The default key is randomly generated, so skip this if you don't know about HMAC or secure session.

```
Secret key for secure cookies [ab03199d87db101aa07fd18e3dc2599a]:
```

See Also:

Flask — *Sessions* Flask provdes client-side secure sessions.

Context Local `flask.session` The session object works pretty much like an ordinary dict, with the difference that it keeps track on modifications.

RFC 2104 — HMAC: Keyed-Hashing for Message Authentication This document describes HMAC, a mechanism for message authentication using cryptographic hash functions. HMAC can be used with any iterative cryptographic hash function, e.g., MD5, SHA-1, in combination with a secret shared key. The cryptographic strength of HMAC depends on the properties of the underlying hash function.

Facebook App ID, Facebook App key, Facebook App secret key Keys of Facebook application used for login. You can create a new Facebook application from Facebook Developers home.

```
Facebook App ID: 123456789012345
Facebook App key: 6753a27847d7e4e3518b1837c2f0e716
Facebook App secret key: edd661737bf101806acb51d83e65c5c1
```

See Also:

Facebook Developers — Create Application

Twitter App key, Twitter App secret key Key pair of Twitter application used for login. You can create a new Twitter application from Twitter Developers home.

```
Twitter App key: X0DS1WP71Mhs8NN0r7paRg
Twitter App secret key: AuJyVWiQm9Jvm61koDP0mv3Gsjgf6GDRrNsvqm5qL
```

See Also:

Twitter Developers — Register an Application Create your own Twitter app.

Tiwtter Developers — Authenticating Requests with OAuth Twitter uses the open authentication standard OAuth for authentication.

See Also:

Flask — *Configuration Handling*

1.2.2 Database initialization

What you have to do next is creating tables into your relational database. There are to recommended relational databases:

SQLite 3+ SQLite is a small and powerful file-based relational database. It is recommended for development-purpose.

PostgreSQL 8.3+ PostgreSQL is a powerful object-relational database system. We recommend it for production-use.

You make a decision, and then, initialize the database via **manage_defernia.py initdb** command:

```
$ manage_defernia.py initdb --config instance.cfg
```

No news is good news. It doesn't print anything unless errors happen.

Note: If you would use SQLite, the data file will be automatically created. But if you would use PostgreSQL, the database to be used have to be created first. Create a database via the **createdb** command PostgreSQL provides:

```
$ createdb -U postgres -E utf8 -T postgres defernia_db
```

See Also:

Command *manage_defernia.py initdb*

1.2.3 Running unit tests

To check whether the installation has sucessful, we can run the unit tests. **setup.py test** command runs the unit tests.

```
$ python setup.py test
running test
[100%] 2 of 2 Time: 0:00:00

Failures: 0/2 (6 assertions)
```

1.2.4 Web server

We finished configuring an instance. Now we can run the development web server from command line:

```
$ manage_defernia.py runserver --config instance.cfg
```

See Also:

Command *manage_defernia.py runserver*

1.2.5 How to serve on WSGI servers

Note: It explains advanced details. If you don't know about WSGI, skip this section and follow *Web server* section.

Defernia web application is WSGI-compliant, so it can be served on WSGI servers. For example, in order to serve it on Meinheld server, make a script:

```
import defernia.web
import meinheld.server

app = defernia.web.create_app(config_filename='instance.cfg')
meinheld.server.listen(('0.0.0.0', 8080))
meinheld.server.run(app)
```

Let's cut to the chase. `defernia.web.create_app()` makes a WSGI application and returns it. It takes a `config_filename` optionally (and it have to be passed by keyword, not positional). And then, pass the created WSGI application into your favorite WSGI server.

1.3 Contributors' guide

Note: It is **not** saying about contents of the fictional world *Ernia*. If you are interested in contributing to contents of the *Ernia* world, just go to http://defernia.org/ and then contribute to there.

If you want to contribute to Defernia system, there are several ways to do it. The most end-level contributions are bug reporting and feature request. In these case you can use *Issue tracker*. The deeper, quicker and more advanced contribution is to submit patches. It is available only if you have some programming skills like programming languages. In this case read *Patches* section.

1.3.1 Issue tracker

Do you have some ideas for enhancements of Defernia system? Did you face a bug of Defernia system? So, you can report it to our **issue tracker** and then we will really thank you.

Issue tracker is a thing like customer service for open source softwares. Defernia system is not developed by a commercial company for enterprise, but free developers for hobby. Nobody pays, nobody makes any profit. There are still users but they are *not customer*. Issue tracker is the official channel of feedback by users to developers and support by developers to users.

Our issue tracker is here:

https://bitbucket.org/dahlia/defernia/issues

To request a feature or report a bug, you should create a new **issue**:

https://bitbucket.org/dahlia/defernia/issues/new

Before do that, there are some instructions you check first: search already reported issues first. There possibly are the same issues already. If you ignore this and just create an yet another one, it will become just a duplicate reporting that doesn't help us but just obstructs us.

If you'd search but can't find anything, creating a new issue is just okay.

Note: Currently the officially allowed languages are English and Korean. There are no developers who speak any other languages in our team yet.

1.3.2 Patches

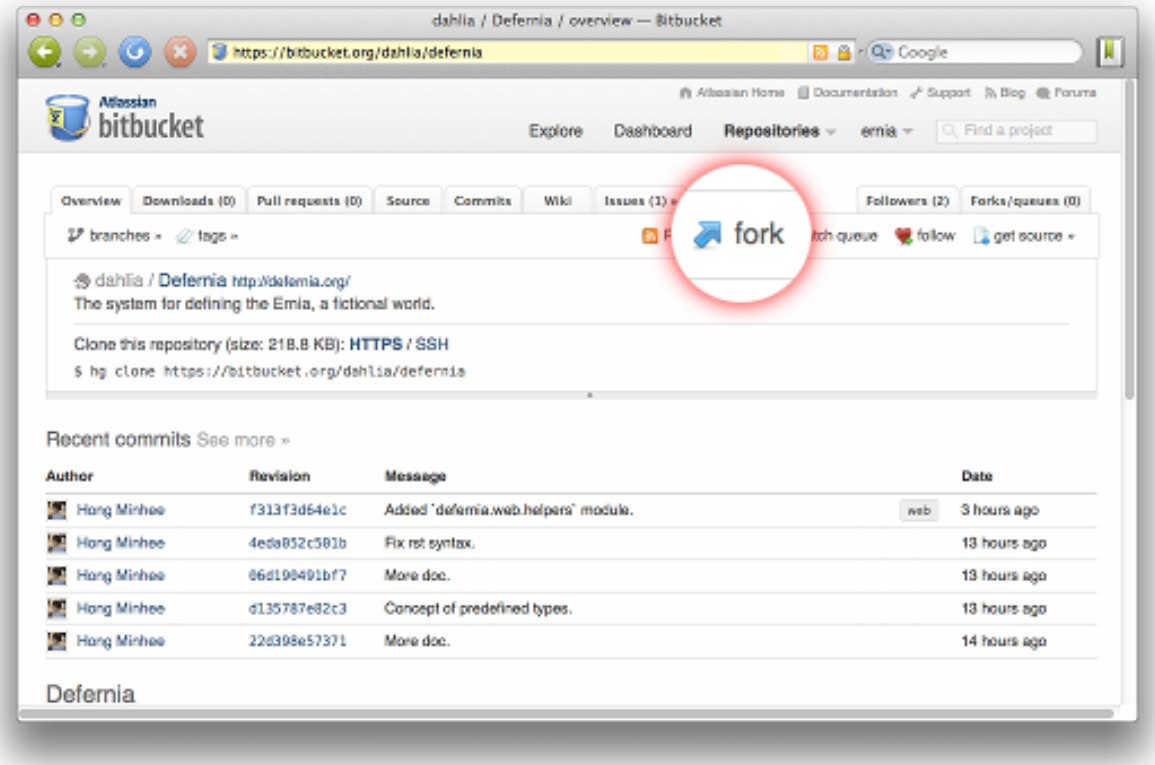
If you have some programming skills, you can hack Defernia system by yourself. To setup the local development environment of Defernia, read *Using virtualenv* and *Getting started* first.

Our source code is maintained by Mercurial, a popular distributed version control system written in Python, and hosted under Bitbucket, a project hosting service. You can check out the source code by the following command:

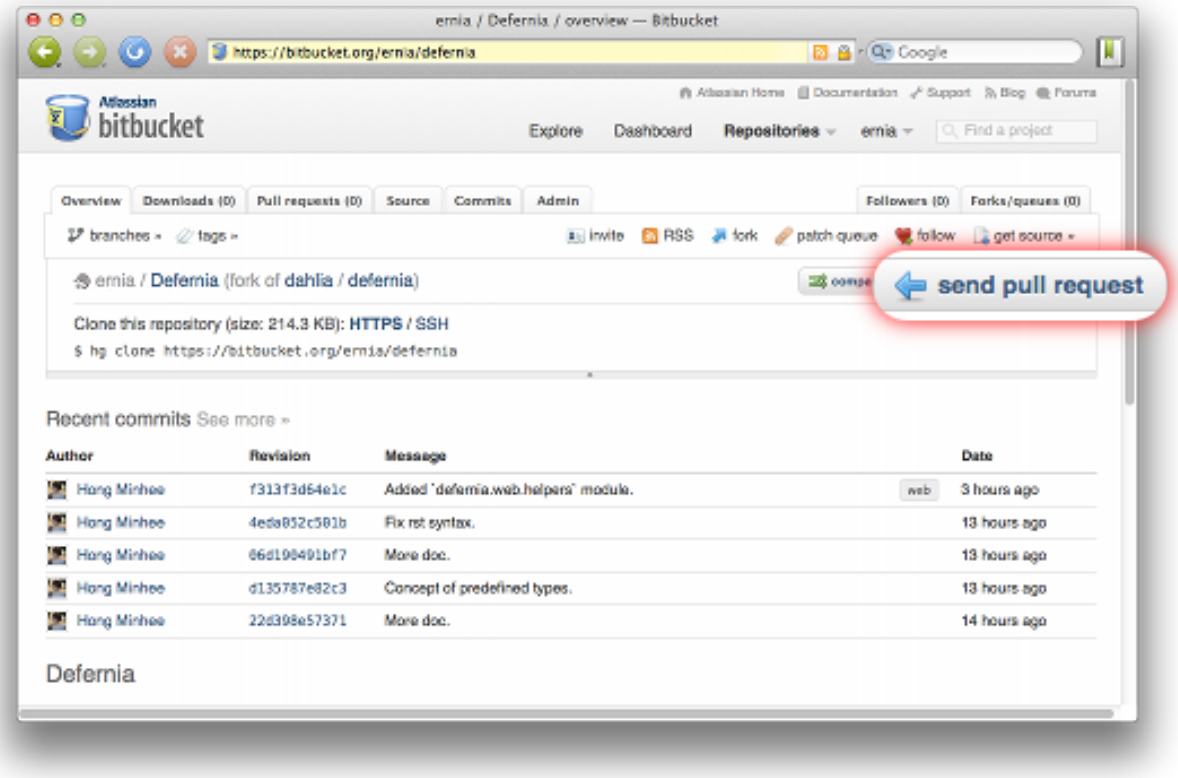
```
$ hg clone https://bitbucket.org/dahlia/defernia
```

For hacking Defernia system there are several *Prerequisite knowledges*.

To submit your patches, you have to use *pull requests* feature (originated by GitHub, anyway) of Bitbucket. If you don't have Bitbucket account, make one first. And then fork our project repository: you can find *fork* button in the web page.



And then push your commits into your forked repository. Now you can send the pull request to the Defernia upstream repository.



If you send the patches by pull requests, we would review your patch and possibly merge these patches into the upstream.

Note: The traditional way to submit patch by attaching diffs into issue tracker is also acceptable but not preferred. There are several reason to prefer pull requests over diff-attached issues/mails:

- Diff-attached patches lost the correct metadata about its contributor. It also makes hard to blame parts of codes in the future.
- Maintainers have to download attached diff files and merges them into the repository manually. It's very boring work.
- Pull requests can archive contributions originated from outside of development team effectively and in integrated way.

1.3.3 Prerequisite knowledges

Here is the minimum list of prerequisite knowledges:

Python Defernia is mostly written in Python, a general-purpose programming language. The version we use is 2.6 or higher. We do not use Python 3 or higher.

Our recommended learning material is Learn Python The Hard Way written by Zed Shaw. You can read it legally free from the web:

http://learnpythonthehardway.org/book/

Mercurial As written below, the source code is mananaged by Mercurial. Moreover, the core of `defernia.world` module heavily depends on Mercurial. (The world data are stored in Mercurial repositories.)

Our recommended learning material is Hg Init: a Mercurial tutorial written by Joel Spolsky. You can read it legally free.

Flask The web frontend part of Defernia system is written in Flask web framework. It is easy and comfortable to learn and use. Of course it's also Python.

The official documentation could be a very good point to start learning Flask.

SQLAlchemy SQLAlchemy is a high level abstraction for relational databases. We use it to deal with data stored in RDBMS. It's also Python but contains a number of black magics, so you should learn about several hidden features under the hood of Python to master this large framework.

So, don't try to master it from scratch. Our recommendation is just to start it from reading *Object Relational Tutorial* and stop learning about it and then just use it. (Of course, in order to hack core parts of Defernia system you should know SQLAlchemy in depth.)

PostgreSQL or SQLite We store any data other than about (con)world, users for example, into RDBMS. We use PostgreSQL in production and recommend one of PostgreSQL or SQLite for development purpose.

Of course you have to know about the legacy codebase of Defernia system as well, but this development guide and *API reference* can help for you.

1.3.4 Convention

Our Python coding style follows the standard of Python: read **PEP 8** first. Don't use hardtab; we only use soft tabs except for Makefile.

We do documentation most of codes in Sphinx and reStructuredText (RST).

Some core parts of Defernia system have unit tests and regression tests in `erniatests` directory. You should attach regression tests also for bug patches.

1.3.5 Unit testing

We encourage to write unit tests including contributed codes as well. You can run unit tests using **tox**. Install tox using **pip** or packaging system software of your operating system (e.g. **apt-get**, **yum**) if you don't have **tox**.

```
$ tox
```

All test codes are in `erniatests/` directory. Add new test cases into that.

1.3.6 Development team

The authors and maintainers of Defernia system are:

- Hong Minhee is the founder of Defernia system and the co-founder of The Chronicle of Ernia. He's the leader of Defernia development team.
- Hyojin Seo is the co-founder of The Chronicle of Ernia. He's even not an advanced programmer, still has some programming knowledges. He mostly helps to decide the big picture and the roadmap of Defernia system.

1.3.7 IRC (Internet Relay Chat)

There is the IRC channel for real-time communication of development team.

irc://irc.ozinger.org/ernia

The IRC network we use is Ozinger. The most of members are always in the channel. You can freely ask us questions related hacking Defernia system. We speak English and Korean both, but we prefer Korean over English frankly.

CHAPTER

TWO

REFERENCES

2.1 manage_defernia.py — Defernia manager script

This scripts provides several subcommands that manage Defernia instances.

-c config
-config config
: Required option. It specify the path of a configruation file.

  If there's no such file, it confirms would you create a such file.

  ```
  $ manage_defernia.py --config instance.cfg
  instance.cfg doesn't exist yet; would you create it? [y]
  ```

-h
-help
: Show the help message and exit.

2.1.1 manage_defernia.py initdb — Database initialization

Creates tables into a database.

```
$ manage_defernia.py initdb --config instance.cfg
```

It doesn't print anything unless errors happened.

2.1.2 manage_defernia.py runserver — Builtin development web server

Runs the development web server.

```
$ manage_defernia.py runserver --config instance.cfg
 * Running on http://127.0.0.1:5000/
```

-t host
-host host
: The host to bind. Default is `127.0.0.1`.

-p port
-port port
: The port number to bind. Default is 5000.

-d

-no-debug
Disables the debug mode. Debug mode enabled by default.

-r
-no-reload
Don't reload automatically even if a file has changed.

2.1.3 manage_defernia.py shell — Interactive shell

It's similar to Python builtin interactive shell, but it also includes the following variables in the global scope:

engine (**sqlalchemy.engine.base.Engine**) The SQLAlchemy connection to the database spcified by configuration.

session (**defernia.orm.Session**) The SQLAlchemy session bound to the above `engine`.

g The context local globals provided by Flask.

app (**flask.Flask**) The Flask application instance.

defernia (**module**) Defernia top-level package.

User (**class**) Defernia user model class.

```
$ manage_defernia.py shell -c dev.cfg

>>>
```

2.1.4 Internal API

manage_defernia — Defernia manager script

manage_defernia.**create_config_file**(*config_filename*)
Creates a new config file.

manage_defernia.**initdb**()
Creates all tables needed by Defernia.

2.2 defernia — The system for conworld

2.2.1 defernia.orm — Object-relational mapping powered by SQLAlchemy

This module provides object-relational mapping facilities powered by SQLAlchemy.

In order to define persist model class, just subclass `Base`:

```
from sqlalchemy import *
import defernia.orm


class Thing(defernia.orm.Base):
    '''A something object-relationally mapped.'''

    id = Column(Integer, primary_key=True)
    value = Column(UnicodeText, nullable=False)
    __tablename__ = 'things'
```

defernia.orm.**Session = sessionmaker(class_='Session'autoflush=True, bind=None, autocommit=True, expire_on_commit**
SQLAlchemy session class.

See Also:

SQLAlchemy — *Using the Session* `Session` is the primary usage interface for persistence operations.

class defernia.orm.**Base**(***kwargs*)
SQLAlchemy declarative base class.

See Also:

SQLAlchemy — *Declarative* Declarative allows all three to be expressed at once within the class declaration.

defernia.orm.**make_repr**(*self*)
Make a `repr()` string for the given `self` object.

Parameters **self** – an object to make a `repr()` string

Returns a `repr()` string

Return type `str`

2.2.2 `defernia.user` — Defernia users

This module defines the `User` model class.

class defernia.user.**User**(***kwargs*)
Defernia users.

emails
The `set` of `Email` objects the user has.

credentials
The `set` of `Credential` objects for user authentication.

id
The unique primary key.

name
The name. Cannot `None` nor an empty string.

created_at
The created time in `datetime.datetime`.

validate_name(*key*, *name*)
Validates `name` value. It have to be a string longer than 1.

picture_url
The profile picture URL. `None` if not present.

class defernia.user.**Email**(**args*, ***kwargs*)
An email address.

EMAIL_PATTERN = <_sre.SRE_Pattern object at 0x4d05b40>
The `re` pattern that matches to valid email addresses.

user_id
The foreign key `id` of `user`.

user
The owner.

email
The email address.

validate_email (*key*, *email*)
Validates the `email` format.

Raises `ValueError` when it's invalid

2.2.3 `defernia.world` — Conworld

`defernia.world.fact` — Object system for conworld ontologies

This module implements a naive implementation of data structures for ontologies. These are extensible by users, so this module implements only some foundations of it: type system (metaclass) for fact ontologies.

The smallest goal of Defernia is to provide the system for *defining* **fictional facts** (also known as *conworld*). Fictional facts have several properties:

- Most of facts are not momentary but continous; for example, some character's height could be getting taller. Terrains also can be changed.
- Facts have relationships each other; for example, there are two characters A and B, and A is possibly B's father. *Relationships* are also called as **predicates** and such facts are also called as **triples** in ontology terminalogy.
- Facts about relationships can deduce more complex facts by composition for example, "A is B's father and B is C's father; so A is C's grandfather."
- Types of relationships ("predicates") can be getting more.
- Unreachable facts are mostly meaningless (for conworld at least).
- Names are not self-given. Roses don't name themselves as "roses" but just people name them "roses" or "". Roses just have red colors and sweet scents.
- Facts can be incomplete. Some properties of a fact are possibly *unknown*.
- While some relationships ("predicates") can be completely inferenced, these can still exist redundantly to leave extra informations through *unknown* informations. For example, there are only 3 fact objects as A's children, but still there could be needs of explicit data about the explicit number of chidren: 4 or 5, *not 3* (rest of children are not defined yet).
- As a side effect, when unknown fields are filled fully some relationships could have inconsistency. These must be detectable and should be able to be resolved easily.

To achieve our goal this module defines the specialized object system based on Python's object system and metaclasses.

class `defernia.world.fact.`**Type** (**args*, ***kwargs*)
The metaclass of fact objects. For example, the character *Root Gorgias* is an instance of `Character` and the type `Character` is an instance of `Type`.

It is a subtype of Python's `types.TypeType` and `Fact` (which also is an instance of `Type`). It means instances of `Type` (namely, *fact types*) also share the same functionalities and properties with `Fact` instances (namely, *facts*). For example, the character *Root Gorgias* and the fact type `Character` both can be stored in a repository.

fields
(`dict`) The dictionary of all fields. Keys are attribute names and values are descriptors.

reference_fields
(`dict`) The dictionary of reference fields. Keys are attribute names and values are descriptors.

value_fields
(dict) The dictionary of value fields. Keys are attribute nameds and values are descriptors.

class defernia.world.fact.**Fact**(***kwargs*)
The fact object. All fact types are subtypes of Fact class.

Note: Internally it maintains __fact__ attribute (similar concept to __dict__) to store internal status of fields.

---

defernia.world.fact.**Descriptor**
The abstract base class for descriptors defined above.

Parameters

- **name** (basestring) – the human-readable name of the field e.g. 'Date of birth'
- **optional** (bool) – set it False if it cannot be None. True by default

defernia.world.fact.**Field**
The descriptor for fact types. It can store ordinary Python objects. (If you don't want to store ordinary Python objects but fact objects, use Reference instead.)

Parameters

- **name** (basestring) – the human-readable name of the field e.g. 'Date of birth'
- **type** (types.TypeType) – the allowed type to set
- **optional** (bool) – set it False if it cannot be None. True by default

defernia.world.fact.**BaseReference**
The abstract base class for Reference and SelfReference.

defernia.world.fact.**Reference**
The reference field. It can store fact object.

Note: Because of evaluation order of Python's class definition, you cannot define a self-referential field by this class. The following class definition doesn't work:

```
class Character(Fact):

    father = Reference(Character)
```

In class definition time, there isn't the class named Character, so it raises a NameError. In this case you have to use SelfReference instead.

Parameters

- **name** (basestring) – the human-readable name of the field e.g. 'Father'
- **type** (Type) – the allowed type to set e.g. Character
- **optional** (bool) – set it False if it cannot be None. True by default

defernia.world.fact.**SelfReference**
The self-referential filed. Shares the same parameters with Descriptor.

defernia.world.fact.**predefined**(*type*)
The class decorator that registers a type as a predefined type. For example:

```
@predefined('character')
class Character(Type):
    pass
```

Or alternatively you can omit its key (`'character'` in the below):

```
@predefined
class Character(Type):
    pass
```

If a key is omitted, it will be set automatically. For example, a class named `ClassName` will be `'class-name'`.

`defernia.world.fact.`**`is_predefined`**(*type*)
Queries whether the given fact `type` is predefined or not.

Parameters **type** (`Type`) – a type to query

Returns `True` only if the `type` is predefined

Return type `bool`

`defernia.world.fact.`**`get_predefined_type`**(*key*)
Returns the predefined type of the passed `key`.

Parameters **key** (`basestring`) – the key of a predefined type

Returns the predefined type

Return type `Type`

Raises `LookupError` if the key doesn't exist

`defernia.world.fact.`**`get_predefined_key`**(*type*)
Returns the key name of the passed predefined `type`.

Parameters **type** (`Type`) – a predefined type

Returns the key name

Return type `basestring`

Raises `ValueError` if `type` is not predefined

`defernia.world.name` — Naming facts

What's in a name? That which we call a rose
By any other name would smell as sweet;

—*Romeo and Juliet*, Act II, Scene II

The concept of naming in Defernia facts is simple: every fact doesn't have its name but references do. Assume that there are two character facts A and B and a relationship that B is a mather of A and A call his mather "Su Gorgias". Thus "Su Gorgias" is a name of B.

In this concept, every fact's names can be multiple. Assume that there are three character facts A, B and C; B is a mother of A and A call his mother "Su Gorgias"; C is a father of A; B and C are married and C call his wife "". Thus B has her two names: "Su Gorgias" and "".

As a result, names can be *weighted* by the number of references. The weightiest names become the **canonical names** automatically.

class defernia.world.name.**NameMap**(*fact*)
The mapping table that contains existing names of the `fact`. Each value has a list of back references and each key has their common name. It implements `collections.Mapping` interface.

For example, assume that there are several relationships between some facts:

- B calls A, his daughter, .
- C calls A, his mother, .
- D calls A, his mother, .
- E calls A, his wife, *Su Gorgias*.

And assume also there is a fact object `a` that represents the below A, and then its `NameMap` will work like: (non-ASCII characters are unescaped for readability)

```
>>> names = NameMap(a)
>>> list(names)
[u'', u'Su Gorgias', u'']
>>> [(name, len(refs)) for name, refs in names.items()]
[(u'', 2), (u'Su Gorgias', 1), (u'', 1)]
```

As you can guess from the below example, it works like a sorted map: these are sorted by their number of references. Moreover there is the property that contains the canonical name: `canon`.

Parameters **fact** (`Fact`) – the fact what the names are of

fact = None
(`Fact`) The fact what the names are of.

canon
(`basestring`) The canonical name of the `fact`. It can be `None` when there are no names for the `fact`.

defernia.world.types — Transcendental fact types

This module provides some predefined built-in fact types.

class defernia.world.types.**Character**(***kwargs*)
The character.

father
(`Character`) The father of the character.

mother
(`Character`) The mother of the character.

defernia.world.serializer — Fact serializer

defernia.world.serializer.**dump**(*fact*, *file*)
Dumps the passed `fact` object into the `file`.

Parameters

- **fact** (`Fact`) – a fact object to serialize
- **file** (*file object*) – a file to be written

defernia.world.serializer.**load**(*file*, *reference_loader=None*)
Loads the fact object from the passed `file` object.

Parameters

- **file** (*file object*) – a file to read
- **reference_loader** (*callable object*) – a function that takes a fact id and returns a fact of it

Returns a loaded fact object

Return type `Fact`

Note: It doesn't initialize loaded fact object's metadata attributes e.g.:

- `__repo__`
- `__rev__`
- `__fact_id__`

`defernia.world.serializer.`**`loads`**(*str*, **args*, ***kwargs*)
Does the same operations as `load()` function does except it takes a string instead of a file object.

`defernia.world.repo` — Revision control of world

Under the hood, `Repository` uses Mercurial as backend to manage revisions. That is, it depends on local filesystem.

class `defernia.world.repo.`**`Repository`**(*path=None*)
World repository.

Parameters **path** (`basestring`) – a repository path. it could be `None` to avoid repository initialization

`facts` = None
(`set`) The set of attached `Fact` objects.

`path`
(`basestring`) The path of the repository. Could be `None` when the repository is not initialized yet.

In order to initialize the repository, set `path` property.

`working`
(`WorkingContext`) The working context of the repository.

`add`(*fact*)
Adds an transient `fact` object into the `repository`. The added `fact` object will start to be tracked by the version control system.

Parameters **fact** (`Fact`) – an transient fact object to add

Note: Actually, it's an alias of `WorkingContext.add()` method.

`commit`(*message*, *user*)
Commits pending changes.

Parameters

- **message** (`basestring`) – a commit message
- **user** – an user string like `'Hong Minhee <minhee@dahlia.kr>'` or an object that implements `__email__()` method

class defernia.world.repo.**BaseChangeContext**(*repository*, *changectx*)
The abstract base class of ChangeContext and WorkingContext.

Additionally it implements collections.Mapping interface also. Fact objects can be gotten by an index operator.

Parameters

- **repository** (Repository) – the repository
- **changectx** (mercurial.context.changectx) – the Mercurial internal change context object

---

Note: Its constructor is internal-use only. Do not instantiate this type directly. Use an index operator of Repository objects instead:

```
repository[rev]
```

---

repository = None
(Repository) The repository of the change context.

changectx = None
(mercurial.context.changectx) The Mercurial internal change context object.

facts = None
(dict) The dictionary of *loaded* fact objects. Keys are __fact_id__ strings and values are Fact objects.

parents
(collections.Sequence) The list of parent nodes.

datetime
(datetime.datetime) The time of the changeset. It is a timezone-aware datetime.datetime value.

class defernia.world.repo.**ChangeContext**(*repository*, *changectx*)
The change context of the revision.

revision
(basestring) The hexadecimal revision.

message
(basestring) Commit message.

class defernia.world.repo.**WorkingContext**(*repository*, *changectx*)
Currently working change context.

ignored_file_globs
(collections.MutableSet) The set of ignored file globs. If you add an additional new glob pattern into the set, it will be added into .hgignore file internally.

add(*fact*)
Adds an transient fact object into the repository. The added fact object will start to be tracked by the version control system.

Parameters **fact** (Fact) – an transient fact object to add

class defernia.world.repo.**FixedOffsetTimezone**(*offset=None*)
The simple datetime.tzinfo implementation that stores just its offset.

Parameters **offset** (datetime.timedelta) – an offset from UTC. UTC by default

class defernia.world.repo.**IgnoredFileGlobSet**(*context*)
The set abstracts `.hgignore` list. It is a subtype of `collections.MutableSet`.

Parameters **context** (`WorkingContext`) – a working context

Note: This object is returned by `WorkingContext.ignored_file_globs` property. Use the property instead of direct creation of this object.

---

path
(`basestring`) The absolute path of `.hgignore` file.

exception defernia.world.repo.**RepositoryError**(**args*, ***kw*)
An abstract base class of `Repository`-related errors. It is a subtype of `mercurial.error.RepoError`.

exception defernia.world.repo.**FactRepositoryError**(*fact*, *repository*, *message=None*)
The exception which rise when the repository of fact has any problem. It is a subtype of `ValueError`.

Parameters

- **fact** (`Fact`) – a fact object related to the error
- **repository** (`Repository`) – a repository related to ther error
- **message** (`basestring`) – an optional error message

fact = None
(`Fact`) The fact object related to the error.

repository = None
(`Repository`) The repository related to the error.

exception defernia.world.repo.**RepositoryLookupError**(*repository*, *revision*, *message=None*)
The exception which rise when the requested revision cannot be found. It is a subtype of `mercurial.error.RepoLookupError` and `LookupError`.

Parameters

- **repository** (`Repository`) – a repository related to ther error
- **revision** – a requested revision
- **message** (`basestring`) – an optional error message

repository = None
(`Repository`) The repository related to the error.

revision = None
The requested revision

defernia.world.repodir — Repositories directory

class defernia.world.repodir.**RepositoryDirectory**(*path*)
A directory that contains one or more `Repository` instances.

Parameters **path** (`basestring`) – a directory path

path = None
The path of the directory repositories belong.

main_name
The name of the `main` repository. It is saved in `.main` file of the repository directory as plain text.

main
The main repository.

2.2.4 defernia.creds — User credentials

Defernia provides various (currently two) ways to sign in: Login with Facebook, Login with Twitter, and other services could be added in the future.

`Credential` abstracts each detail of various login services.

In order to define a new credential service, subclass `Credential` and then define `typeid` class attribute:

```
class WindowsLive(Credential):
    '''The Microsoft's legendary SSO (single sign on) service,
    was Passport before.

    '''

    typeid = 'live.com'
```

Note that `typeid` should be the service's canonical domain name such as `'live.com'`. This attribute will be internally used for service identifier.

class defernia.creds.**Credential**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

Defernia user credentials.

typeid = NotImplemented
The class attribute to be defined for concrete subclasses. It is a default `type` of the class. It should be the service's canonical domain name such as `'facebook.com'`.

user_id
The foreign key `id` of `user`.

user
The owner.

type
The credential type e.g. `'facebook.com'`, `'twitter.com'`. It should be the service's canonical domain name.

identifier
The user identifier used in the service.

data
The extra data for the credential.

defernia.creds.**find_type**(*typeid*)
A class method that finds the subtype of the given `typeid`.

Parameters **typeid** (`basestring`) – internally used `typeid` string e.g. `'facebook.com'`

Returns a subtype of `Credential`

Return type `type`

Rairses `LookupError` when cannot found the class

defernia.creds.**list_types**(*mixin=<class 'defernia.creds.Credential'>*)
Lists all credential types.

Parameters **mixin** (`type`) – an optional mixin class filter. if it is present, lists only subclass of given `mixin`

Returns a dictionary of credential types (keys are `typeid` strings)

Return type `dict`

`defernia.creds.`**`data_property`**(*key*, *doc=None*)

Makes a descriptor that deals with `data` dictionary's specific `key`.

```
class HongMinheeWebsite(Credential):
    typeid = 'dahlia.kr'
    gender = data_property('gender', "The ```male``` or ```female```.")
    birthday = data_property('birthday')
```

Parameters

- **key** – the key of `data` dictionary
- **doc** (`str`) – an optional docstring

Returns a descriptor for `key`

Return type `property`

class `defernia.creds.`**`PictureMixin`**(***kwargs*)

Bases: `defernia.creds.Credential`

The credential mixin class for getting profile pictures.

`picture_url`

The profile picture URL. To be overrided in the subclass. `None` if there's no picture.

class `defernia.creds.`**`Facebook`**(***kwargs*)

Bases: `defernia.creds.PictureMixin`

Login with Facebook.

class `defernia.creds.`**`Twitter`**(***kwargs*)

Bases: `defernia.creds.PictureMixin`

Login with Twitter.

`screen_name`

The Twitter screen name.

2.2.5 `defernia.objsimplify` — Object simplifier for generic serialization

`defernia.objsimplify.`**`simplify`**(*value*, *identifier_map*, *type_map={}*, *url_map=None*, *user=None*, ***extra*)

Simplifies a given `value`.

Parameters

- **value** – an object to simplify
- **identifier_map** (*callable object*) – a map function that normalizes multi-word identifiers
- **type_map** (`dict`) – a type to mapping function dictionary
- **url_map** (*callable object*) – a map function that returns an url of a given `value` object. by default, it is a constant function that just returns `None`, so simplified dictionaries have no url data

- **user** (defernia.user.User) – an user object for signing

defernia.objsimplify.**under_scores**(*identifier*)

Concatenates words of the identifier by underscore ('_').

```
>>> under_scores('key name')
'key_name'
>>> under_scores('encode URL')
'encode_url'
```

Note: Use this function for simplify() function's identifier_map option.

---

defernia.objsimplify.**PascalCase**(*identifier*)

Makes the identifier PascalCase.

```
>>> PascalCase('key name')
'KeyName'
>>> PascalCase('encode URL')
'EncodeURL'
```

Note: Use this function for simplify() function's identifier_map option.

---

defernia.objsimplify.**camelCase**(*identifier*)

Makes the identifier camelCase.

```
>>> camelCase('key name')
'keyName'
>>> camelCase('encode URL')
'encodeUrl'
>>> camelCase('URL encoder')
'urlEncoder'
```

Note: Use this function for simplify() function's identifier_map option.

---

defernia.objsimplify.**transform = <typequery.GenericMethod 'transform' at 0x4dfae10>**

Warning: Internal use only. Use simplify() instead.

The function that really implements simplification per types, without simplify() 's type_map option.

Parameters

- **value** – an object to simplify
- ****options** – extra options

2.2.6 defernia.web — Web frontend

Defernia uses Flask as framework for web frontend. It depends on Werkzeug and Jinja2 also.

See Also:

Module defernia.web.routing Extended Werkzeug routing converters for Defernia.

Blueprint defernia.web.home Website home.

Blueprint defernia.web.user User authentications, personal pages, and so on.

Module `defernia.web.helpers` Helpful template filters, tests and functions for Jinja.

Module `defernia.web.serializers` Object serializers for various content types.

Module `defernia.web.wsgi` Custom WSGI middlewares for Defernia web application.

`flask.g.`**`world_repodir`**
(`defernia.world.repo.RepositoryDirectory`) The global variable that stores the world repository directory object.

`flask.g.`**`session`**
(`defernia.orm.Session`) The global variable that stores the SQLAlchemy session.

`flask.g.`**`database_engine`**
(`sqlalchemy.engine.base.Engine`) The global variable that stores SQLAlchemy dtabase engine.

`defernia.web.`**`blueprints = {'defernia.web.user:user': {'url_prefix': '/users'}, 'defernia.web.world:world': {'url_prefix':`**
The `dict` of blueprints to be registered. Keys are import names in string, and values are keyword arguments for `flask.Flask.register_blueprint()` method.

```
blueprints = {'module.name:var': {'url_prefix': '/path'},
              'module.name2:var2': {}}
```

See Also:

Function `werkzeug.utils.import_string()` The function that imports an object based on a string, provided by Werkzeug.

Flask — *Modular Applications with Blueprints* Flask provides 'blueprint' facilities for large applications.

`defernia.web.`**`wsgi_middlewares = ['defernia.web.wsgi:MethodRewriteMiddleware']`**
The list of WSGI middlewares to hook in. Its elements are import names in string.

```
wsgi_middlewares = ['defernia.web.wsgi:MethodRewriteMiddleware']
```

See Also:

Function `werkzeug.utils.import_string()` The function that imports an object based on a string, provided by Werkzeug.

Module `defernia.web.wsgi` Custom WSGI middlewares for Defernia web application.

Flask — Hooking in WSGI Middlewares Flask provides a way to hook in WSGI middlewares.

`defernia.web.`**`content_types = {'text/xml': '.xml', 'text/html': '.html', 'application/xhtml+xml': '.html', 'application/pl`**
The `dict` of serializers for content types. Keys are MIME types like *`application/json`*, and values are functions that encode a value into the paired type, or a string which is a postfix of the template filename e.g. `'.html'`, `'.xml'`. If value is a string that doesn't start with period (`.`), it will be interpreted as import name.

```
content_types = {'application/json': json.dumps,
                 'text/yaml': 'defernia.web.serializers:yaml',
                 'text/html': '.html',
                 'text/xml': '.xml'}
```

See Also:

Function `render()` The generic content type version of `flask.render_template()`.

Function `werkzeug.utils.import_string()` The function that imports an object based on a string, provided by Werkzeug.

defernia.web.**create_app**(*modifier=None*, *config_filename=None*)
An application factory. It sets up the application then returns the application.

```
app = create_app(config_filename='prod.cfg')
```

Instead you pass an argument `config_filename`, it can be used as decorator-style as well:

```
@create_app
def app(app):
    app.debug = True
    app.config['MAGIC_NUMBER'] = 1234
```

Parameters

- **modifier** (*callable object*) – a function, for decorator-style use
- **config_filename** (`basestring`) – a configuration file name

Returns a WSGI application

Return type `flask.Flask`

defernia.web.**get_world_repodir**(*config*)
Gets `defernia.world.repo.RepositoryDirectory` object from the `config`.

Parameters **config** (`flask.Config`, `dict`) – the configuration that contains `'REPODIR_PATH'`

Returns world repository directory object

Return type `defernia.world.repo.RepositoryDirectory`

defernia.web.**get_database_engine**(*config*)
Gets SQLAlchemy `Engine` object from the `config`.

Parameters **config** (`flask.Config`, `dict`) – the configuration that contains `'DATABASE_URL'` or `'ENGINE'`

Returns SQLAlchemy database engine

Return type `sqlalchemy.engine.base.Engine`

See Also:

SQLAlchemy — *Engine Configuration*

defernia.web.**before_request**(*function*)
The decorator that registers `function` into `before_request_funcs`.

defernia.web.**after_request**(*function*)
The decorator that registers `function` into `after_request_funcs`.

defernia.web.**define_session**()
Sets the `g.world_repodir`, `g.session` and `g.database_engine` global variables before every request.

defernia.web.**render**(*template_name*, *value*, ***context*)
The generic content type version of `flask.render_template()` function. Unlike `flask.render_template()`, it takes one more required parameter, `value`, for generic serialization to JSON-like formats. And `template_name` doesn't include its postfix.

```
render('user/profile', user, user=user)
```

Parameters

- **template_name** (`basestring`) – the name of the template to be rendered, but postfix excluded
- ****context** – the variables that should be available in the context of the template

See Also:

Constant `content_types`

2.2.7 `defernia.web.routing` — Custom routing converters

See Also:

Werkzeug — *URL Routing*

class `defernia.web.routing.`**`CredentialTypeConverter`**(*map*)
Bases: `werkzeug.routing.BaseConverter`

This converter accepts `Credential` types, represented in `typeid` string:

```
Rule('/page/<credtype:service>')
```

Parameters **map** (`werkzeug.routing.Map`) – the `Map`

exception `defernia.web.routing.`**`MovedPermanently`**(*url*, *description=None*)
Bases: `werkzeug.exceptions.HTTPException`

301 *Moved Permanently*

`url` = None
The URL to redirect.

class `defernia.web.routing.`**`UserConverter`**(*map*, *autoredirect=True*)
Bases: `werkzeug.routing.BaseConverter`

This converter accepts `User` objects, represented in `id` with optional slug generated from `name`. For example:

```
123
123/hong.minhee
```

Actually it does not returns a `User` object. Instead, it returns a function that returns a `User` object. So you have to call after give it:

```
@app.route('/<user:user>')
def profile(user):
    user = user()
```

Parameters

- **map** (`werkzeug.routing.Map`) – the `Map`
- **autoredirect** (`bool`) – whether redirects the user to the canonical url when the request url is not canonical

> **Warning:** It has a lot of side effects about errors and redirections that are unexpected in the design of `werkzeug.routing.BaseConverter` class. Please be careful!

`autoredirect` = True
Whether redirects the user to the canonical url when the request url is not canonical.

slug(*user*)
Makes a normalized slug for the `user`.

`defernia.web.routing.`**converters = {'credtype': <class 'defernia.web.routing.CredentialTypeConverter'>, 'user': <cl**
The dictionary of extended routing converters. It should be registered when the `Flask` application has created:

```
app = flask.Flask(__name__)
app.url_map.converters.update(defernia.web.routing.converters)
```

See Also:

Attribute `flask.Flask.url_map`

2.2.8 `defernia.web.home` — Website home

`defernia.web.home.`**home = <flask.blueprints.Blueprint object at 0x58a1290>**
Home module.

See Also:

Flask — *Working with Blueprints*

2.2.9 `defernia.web.user` — User web pages

`flask.g.`**current_user**
The global variable that stores the currently signed `User` object.

`defernia.web.user.`**user = <flask.blueprints.Blueprint object at 0x59913d0>**
User web pages module.

See Also:

Flask — *Working with Blueprints*

`defernia.web.user.`**define_current_user**()
Sets the `g.current_user` global variable before every request.

`defernia.web.user.`**inject_current_user**()
Injects the `current_user` for templates.

```
{% if current_user %}
  <p>You are {{ current_user }}.</p>
{% else %}
  <p>Who are you?</p>
{% endif %}
```

`defernia.web.user.`**set_current_user**(*user*)
Sets the `g.current_user`. `user` can be `None` also.

Parameters **user** (`User`, `types.NoneType`) – the user to set. signs out if it is `None`

`defernia.web.user.`**signout**()
Signs out.

`defernia.web.user.`**signin**(*service_cls*)
Starts to sign in. Shows a login form, or redirects the user to a login form of an other service.

`defernia.web.user.`**signin_process**(*service_cls*)
Finishes signing in.

defernia.web.user.**profile**(*user*)
User profile page.

defernia.web.user.**edit_profile_form**(*user*)
User profile edit form.

2.2.10 defernia.web.helpers — Template helpers

This module contains several useful helpers for templating. To install it into the Jinja environment, use `install()` function.

class defernia.web.helpers.**RegisterDecorator**(*dictionary*)
Decorator function template. It is for making `filter()`, `test()` and `func()` decorator functions.

defernia.web.helpers.**filter = <defernia.web.helpers.RegisterDecorator object at 0x573c0d0>**
The decorator function that registers a function as a template filter:

```
@filter
def greet(value):
    return u'Hello, ' + unicode(value)
```

You can specify the custom name for it:

```
@filter('hello')
def greet(value):
    return u'Hello, ' + unicode(value)
```

defernia.web.helpers.**test = <defernia.web.helpers.RegisterDecorator object at 0x573c110>**
The decorator function that registers a function as a template test (predicate). For example:

```
@test
def digit(value):
    return value.isdigit()
```

You can specify the custom name for it:

```
@test('digit')
def is_digit(value):
    return value.isdigit()
```

defernia.web.helpers.**func = <defernia.web.helpers.RegisterDecorator object at 0x573c150>**
The decorator function that registers a function as a template global function:

```
@func
def code(value):
    return u'<code>{0}</code>'.format(value)
```

You can specify the custom name for it as well as `filter()` or `test()` decorators.

defernia.web.helpers.**install**(*environment*)
Install the helpers defined above into the `environment`.

Parameters **environment** (`jinja2.Environment`) – the Jinja environment

defernia.web.helpers.**require**(*import_path*)
Imports a module or an object inside it by its import path.

```
{% set datetime = require('datetime') %}
{% set user = require('defernia.user') %}
```

You can specify an object inside a module as well:

```
{% set date = require('datetime:date') %}
{% set User = require('defernia.user:User') %}
```

Parameters **import_path** (`basestring`) – a dot-separated standard Python import path

Returns an imported module or object

See Also:

Function `werkzeug.utils.import_string()` Imports an object based on a string.

2.2.11 `defernia.web.serializers` — Serializers for various content types

`defernia.web.serializers.`**`to_json`**(*value*)
Serializes a `value` into JSON (*`application/json`*).

Parameters **value** – a value to serialize to JSON

Returns a serialized JSON string

Return type `basestring`

`defernia.web.serializers.`**`to_plist`**(*value*)
Serializes a `value` into property list (plist) format. (*`application/plist+xml`*)

Parameters **value** – a value to serialize to plist

Returns a serialized plist XML

Return type `basestring`

2.2.12 `defernia.web.wsgi` — Custom WSGI middlewares

class `defernia.web.wsgi.`**`MethodRewriteMiddleware`**(*application*)
Bases: `object`

The WSGI middleware that overrides HTTP methods for old browsers. HTML4 and XHTML only specify `POST` and `GET` as HTTP methods that `<form>` elements can use. HTTP itself however supports a wider range of methods, and it makes sense to support them on ther server.

If you however want to make a form submission with `PUT` for instance, and you are using a client that does not support it, you can override it by using this middleware and appending `?__method__=PUT` to the `<form>` `action`.

```
<form action="?__method__=put" method="post">
  ...
</form>
```

Parameters **application** (*callable object*) – WSGI application to wrap

See Also:

Flask — Overriding HTTP Methods for old browsers

class defernia.web.wsgi.**HostRewriteMiddleware**(*application*, *host=None*, *config_name='HOST_REWRITE'*)

Bases: object

A WSGI middleware that overwrites every request's *Host* header (that is, HTTP_HOST environment) to the specific host name. It is useful when WSGI server is running under proxy server.

Parameters

- **application** (callable object, flask.Flask) – WSGI application to wrap
- **host** (basestring) – a host name to rewrite. if not present, HOST_REWRITE configuration may be used (only when application is a flask.Flask instance)

2.2.13 defernia.version — Version data

defernia.version.**VERSION = (0, 1, 0)**
(tuple) The version tuple e.g. (0, 1, 2).

defernia.version.**VERSION_INFO = '0.1.0'**
(basestring) The version string e.g. '0.1.2'.

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

d

m