Flask-Registry Documentation

Release 0.2.1.dev20140801

CERN

July 11, 2016

Contents

CHAPTER 1

Flask-Registry

Flask-Registry is a Flask extension that allows frameworks to dynamically assemble Flask application from reusable packages consisting of blueprints, extensions, and configurations.

CHAPTER 2

User's Guide

This part of the documentation will show you how to get started in using Flask-Registry with Flask.

2.1 Installation

Install Flask-Registry with `pip`:

```
$ pip install flask-registry
```

The development version can be downloaded from its page at GitHub.

```
$ git clone https://github.com/inveniosoftware/flask-registry.git
$ cd flask-registry
$ python setup.py develop
$ ./run-tests.sh
```

2.1.1 Requirements

Flask-Registry has the following dependencies:

- Flask
- six

Flask-Registry requires Python version 2.6, 2.7 or 3.3+

2.2 Quickstart

This guide assumes you have successfully installed Flask-Registry and a working understanding of Flask. If not, follow the installation steps and read about Flask at http://flask.pocoo.org/docs/.

2.2.1 A Minimal Example

A minimal Flask-Registry usage example looks like this. First create the application and initialize the extension:

```
>>> from flask import Flask
>>> from flask_registry import Registry
>>> from flask_registry import ListRegistry
>>> app = Flask('myapp')
>>> r = Registry(app=app)
```

Then, we can create a simple ListRegistry that just keeps a list of objects:

```
>>> r['my_namespace'] = ListRegistry()
>>> r['my_namespace'].register("something")
>>> r['my_namespace'].register("something else")
>>> for obj in r['my_namespace']:
...     print(obj)
something
something else
```

2.2.2 Application Discovery Example

Flask-Registry also has support for dynamically discovering Python modules, resources, entry points and the like. All this can be put together in your Flask application factory to create and easily extensible application.

Following is a small example how a Flask application can be assemble from reusable packages that each provides configuration, extensions and blueprints:

```
from flask import Flask

from flask_registry import (BlueprintAutoDiscoveryRegistry,
                            ConfigurationRegistry, ExtensionRegistry,
                            PackageRegistry, Registry)


class Config(object):
    PACKAGES = ['registry_module']
    EXTENSIONS = ['registry_module.mockext']
    USER_CFG = True


def create_app(config):
    app = Flask('myapp')
    app.config.from_object(config)
    r = Registry(app=app)
    r['packages'] = PackageRegistry(app)
    r['extensions'] = ExtensionRegistry(app)
    r['config'] = ConfigurationRegistry(app)
    r['blueprints'] = BlueprintAutoDiscoveryRegistry(app=app)
    return app

if __name__ == '__main__':
    config = Config()
    app = create_app(config)
    app.run(debug=True)
```

Save this in a file named `app.py` next to the `tests` folder in the Flask-Registry distribution and run it using your Python interpreter.

```
$ python app.py
 * Running on http://127.0.0.1:5000/
```

```
$ curl http://localhost:5000
Hello from Flask-Registry
```

The blueprint is loaded from `tests.views` and only works if the extension `registry_module.mockext` and the configuration in `registry_module.config` has been loaded.

See *Application Discovery* for full explanation on what is happening in the example.

2.3 User Guide

Flask extension to dynamically assemble your Flask application from packages.

Flask-Registry is initialized like this:

```
>>> from flask import Flask
>>> from flask_registry import Registry, ListRegistry
>>> app = Flask('myapp')
>>> r = Registry(app=app)
```

A simple usage example of `ListRegistry` looks like this:

```
>>> app.extensions['registry']['my.namespace'] = ListRegistry()
>>> len(app.extensions['registry'])
1
>>> app.extensions['registry']['my.namespace'].register("something")
>>> app.extensions['registry']['my.namespace'].register("something else")
>>> len(app.extensions['registry']['my.namespace'])
2
>>> for obj in app.extensions['registry']['my.namespace']:
...     print(obj)
something
something else
```

2.3.1 Module Discovery

The module discovery registries.

They provide discovery functionality useful for searching a list of Python packages for a specific module name, and afterwards registering the module. This is used to e.g. load and register Flask blueprints by `BlueprintAutoDiscoveryRegistry`.

Assume e.g. we want to discover the `helpers` module from the `tests` package. First we initialize the registry:

```
>>> from flask import Flask
>>> from flask_registry import Registry, ModuleDiscoveryRegistry
>>> from flask_registry import ImportPathRegistry
>>> app = Flask('myapp')
>>> r = Registry(app=app)
```

We then create the list of packages to search through using an `ImportPathRegistry`:

```
>>> r['mypackages'] = ImportPathRegistry(initial=['registry_module'])
```

Then, initialize the `ModuleDiscoveryRegistry` and run the discovery:

```
>>> r['mydiscoveredmodules'] = ModuleDiscoveryRegistry(
...     'helpers', registry_namespace='mypackages')
>>> len(r['mydiscoveredmodules'])
0
>>> r['mydiscoveredmodules'].discover(app=app)
>>> len(r['mydiscoveredmodules'])
1
```

Lazy discovery

Using `RegistryProxy` you may lazily discover modules. Above example using lazy loading looks like this:

```
>>> from flask_registry import RegistryProxy
>>> app = Flask('myapp')
>>> r = Registry(app=app)
>>> pkg_proxy = RegistryProxy('mypackages', ImportPathRegistry,
...     initial=['registry_module'])
>>> mod_proxy = RegistryProxy('mydiscoveredmodules',
...     ModuleDiscoveryRegistry,
...     'helpers',
...     registry_namespace=pkg_proxy)
>>> 'mypackages' in r
False
>>> 'mydiscoveredmodules' in r
False
>>> with app.app_context():
...     mod_proxy.discover(app=app)
>>> 'mypackages' in r
True
>>> 'mydiscoveredmodules' in r
True
```

2.3.2 Application Discovery

Application discovery registries.

They provide discovery functionality useful for dynamically constructing Flask applications based on configuration variables. This allows a developer to package config, blueprints and extensions into isolated and reusable packages which a framework can dynamically install into a Flask application.

Such a package (named `registry_module`) could look like and it is located in `tests` directory:

- `registry_module.views` – contains blueprints which should be registered on the application object.
- `registry_module.mockext` – contains a `setup_app()` method which be used to install any Flask extensions on the application object.
- `registry_module.config` – contains configuration variables specific for this module.

Following is a simplified example of a Flask application factory, that will load config, extensions and blueprints:

```
>>> from flask import Flask, Blueprint
>>> from flask_registry import Registry, PackageRegistry
>>> from flask_registry import ExtensionRegistry
>>> from flask_registry import ConfigurationRegistry
>>> from flask_registry import BlueprintAutoDiscoveryRegistry
>>> class Config(object):
...     PACKAGES = ['registry_module']
```

```
...     EXTENSIONS = ['registry_module.mockext']
...     USER_CFG = True
>>> def create_app(config):
...     app = Flask('myapp')
...     app.config.from_object(config)
...     r = Registry(app=app)
...     r['packages'] = PackageRegistry(app)
...     r['extensions'] = ExtensionRegistry(app)
...     r['config'] = ConfigurationRegistry(app)
...     r['blueprints'] = BlueprintAutoDiscoveryRegistry(app=app)
...     return app
>>> config = Config()
>>> app = create_app(config)
```

Packages

The config variable `PACKAGES` specifies the list of Python packages, which `ConfigurationRegistry` and `BlueprintAutoDiscoveryRegistry` will search for `config.py` and `views.py` modules inside.

```
>>> for pkg in app.extensions['registry']['packages']:
...     print(pkg)
registry_module
```

Extensions

The config variable `EXTENSIONS` specifies the list of Python packages, which the `ExtensionRegistry` will load and call `setup_app(app)` on, to dynamically initialize Flask extensions.

```
>>> for pkg in app.extensions['registry']['extensions']:
...     print(pkg)
registry_module.mockext
```

Configuration

The `ConfigurationRegistry` will merge any package defined config, with the application config without overwriting already set variables in the application config:

```
>>> config.USER_CFG
True
>>> import registry_module.config
>>> registry_module.config.USER_CFG
False
>>> app.config['USER_CFG']
True
```

Blueprints

The `BlueprintAutoDiscoveryRegistry` will search for blueprints defined inside a `views` module in each package defined in `PACKAGES`. It will also register the discovered blueprints on the Flask application. Each `views` module should define either a single blueprint in the variable `blueprint` and/or multiple blueprints in the variable `blueprints`:

```
>>> from registry_module import views
>>> isinstance(views.blueprint, Blueprint)
True
>>> len(views.blueprints)
2
>>> for k in sorted(app.blueprints.keys()):
...     print(k)
test
test1
test2
```

2.3.3 Package Resources

Package Resources

Package resource registries may be used to discover e.g. package resources as well as loading entry points.

Entry points

`setuptools` entry points are a simple way for packages to "advertise" Python objects, so that frameworks can search for these entry points. `setup.py` files for instance allows you to specify `console_scripts` entry points, which will install scripts into system path for you.

The `EntryPointRegistry` allows you to easily register these entry points into your Flask application:

```
>>> from flask import Flask
>>> from flask_registry import Registry, EntryPointRegistry
>>> app = Flask('myapp')
>>> r = Registry(app=app)
>>> r['scripts'] = EntryPointRegistry('console_scripts')
>>> 'easy_install' in r['scripts']
True
```

Entry points are specified in you `setup.py`, e.g.:

```
setup(
    # ...
    entry_points={
        'flask_registry.test_entry': [
            'testcase = flask_registry:RegistryBase',
        ]
    },
    # ...
)
```

```
>>> r['entrypoints'] = EntryPointRegistry(
...     'flask_registry.test_entry', load=True)
>>> 'testcase' in r['entrypoints']
True
>>> from flask_registry import RegistryBase
>>> r['entrypoints']['testcase'][0] == RegistryBase
True
```

See http://pythonhosted.org/setuptools/pkg_resources.html#entry-points for more information on entry points.

Resource files

The `PkgResourcesDirDiscoveryRegistry` will search a list of Python packages for a specific resource directory and register all files found in the directories.

Assume e.g. a package `tests` have a directory `resources` with one file in it called `testresource.cfg`. This file can be discovered in the following manner:

```
>>> import os
>>> app = Flask('myapp')
>>> r = Registry(app=app)
>>> from flask_registry import ImportPathRegistry
>>> from flask_registry import PkgResourcesDirDiscoveryRegistry
>>> r['packages'] = ImportPathRegistry(initial=['registry_module'])
>>> r['res'] = PkgResourcesDirDiscoveryRegistry('resources', app=app)
>>> os.path.basename(r['res'][0]) == 'testresource.cfg'
True
```

2.3.4 Extending Flask-Registry

Flask-Registry extensions.

Extending Flask-Registry

You can easily create your own type of registries by subclassing one of the existing registries found in the modules under `flask_registry.registries`.

If you for instance want to create a list registry that only accepts integers, you could create it like this:

```
>>> from flask import Flask
>>> from flask_registry import Registry, RegistryError, ListRegistry
>>> class IntListRegistry(ListRegistry):
...     def register(self, item):
...         if not isinstance(item, int):
...             raise ValueError("Object must be of type int")
>>> app = Flask('myapp')
>>> r = Registry(app=app)
>>> r['myns'] = IntListRegistry()
>>> r['myns'].register(1)
>>> r['myns'].register("some string")
Traceback (most recent call last):
  File "/usr/lib/python2.7/doctest.py", line 1289, in __run
    compileflags, 1) in test.globs
  File "<doctest default[7]>", line 1, in <module>
    r['myns'].register("some string")
  File "<doctest default[2]>", line 4, in register
    raise ValueError("Object must be of type int")
ValueError: Object must be of type int
```

CHAPTER 3

API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

3.1 API Docs

Flask extension to dynamically assemble your Flask application from packages.

Flask-Registry is initialized like this:

```
>>> from flask import Flask
>>> from flask_registry import Registry, ListRegistry
>>> app = Flask('myapp')
>>> r = Registry(app=app)
```

A simple usage example of `ListRegistry` looks like this:

```
>>> app.extensions['registry']['my.namespace'] = ListRegistry()
>>> len(app.extensions['registry'])
1
>>> app.extensions['registry']['my.namespace'].register("something")
>>> app.extensions['registry']['my.namespace'].register("something else")
>>> len(app.extensions['registry']['my.namespace'])
2
>>> for obj in app.extensions['registry']['my.namespace']:
...     print(obj)
something
something else
```

class `flask_registry.`**`Registry`**(*app=None*)

Bases: `_abcoll.MutableMapping`

Flask extension.

Initialization of the extension:

```
>>> from flask import Flask
>>> from flask_registry import Registry
>>> app = Flask('myapp')
>>> r = Registry(app)
>>> app.extensions['registry']
<Registry ()>
```

or alternatively using the factory pattern:

```
>>> app = Flask('myapp')
>>> r = Registry()
>>> r.init_app(app)
>>> r
<Registry ()>
```

init_app(*app*)
Initialize a Flask application.

Only one Registry per application is allowed.

Parameters **app** (*flask.Flask*) – Flask application

Raises ***flask_registry.RegistryError*** – if the registry is already initialized

class flask_registry.**RegistryProxy**(*namespace*, *registry_class*, **args*, ***kwargs*)
Bases: werkzeug.local.LocalProxy

Lazy proxy object to a registry in the current_app

Allows you to define a registry in your local module without needing to initialize it first. Once accessed the first time, the registry will be initialized in the current_app, thus you must be working in either the Flask application context or request context.

```
>>> from flask import Flask
>>> app = Flask('myapp')
>>> from flask_registry import Registry, RegistryProxy, RegistryBase
>>> r = Registry(app=app)
>>> proxy = RegistryProxy('myns', RegistryBase)
>>> 'myns' in app.extensions['registry']
False
>>> with app.app_context():
...     print(proxy.namespace)
...
myns
>>> 'myns' in app.extensions['registry']
True
```

Parameters

- **namespace** – Namespace for registry
- **registry_class** – The registry class - i.e. a sublcass of RegistryBase.
- **args** – Arguments passed to registry_class on initialization.
- **kwargs** – Keyword arguments passed to registry_class on initialization.

class flask_registry.**RegistryError**
Bases: exceptions.Exception

Exception class raised for user errors.

e.g. creating two registries in the same namespace)

Registry base module.

class flask_registry.base.**RegistryBase**
Bases: object

Abstract base class for all registries.

Each subclass must implement the `register()` method. Each subclass may implement the `unregister()` method.

Once a registry is registered in the Flask application, the namespace under which it is available is injected into it self.

Please see `flask_registry.registries.core` for simple examples of subclasses.

namespace
Namespace. Used only by the Flask extension to inject the namespace under which this instance is registered in the Flask application. Defaults to `None` if not registered in a Flask application.

register(**args*, ***kwargs*)
Abstract method which MUST be overwritten by subclasses. A subclass does not need to take the same number of arguments as the abstract base class.

unregister(**args*, ***kwargs*)
Abstract method which MAY be overwritten by subclasses. A subclass does not need to take the same number of arguments as the abstract base class.

Core registries.

3.1.1 Core Registries

The core registries are useful to use as subclasses for other more advanced registries. The provide the basic functionality for `list` and `dict` style registries, as well as simple import path and module style registries.

class `flask_registry.registries.core.`**ListRegistry**
Bases: *flask_registry.base.RegistryBase*, `_abcoll.Sequence`

Basic registry that just keeps a list of objects.

Provides normal list-style access to the registry:

```
>>> from flask import Flask
>>> from flask_registry import Registry, ListRegistry
>>> app = Flask('myapp')
>>> r = Registry(app=app)
>>> r['myns'] = ListRegistry()
>>> r['myns'].register("something")
>>> len(r['myns'])
1
>>> r['myns'][0]
'something'
>>> "something" in r['myns']
True
>>> for obj in r['myns']:
...     print(obj)
something
```

register(*item*)
Register a new object

Parameters **item** – Object to register

unregister(*item*)
Unregister an existing object. Raises a `ValueError` in case object does not exists. If the same object was registered twice, only the first registered object will be unregister.

Parameters **item** – Object to unregister

class flask_registry.registries.core.**DictRegistry**
Bases: *flask_registry.base.RegistryBase*, _abcoll.MutableMapping

Basic registry that just keeps a key, value pairs.

Provides normal dict-style access to the registry:

```
>>> from flask import Flask
>>> from flask_registry import Registry, DictRegistry
>>> app = Flask('myapp')
>>> r = Registry(app=app)
>>> r['myns'] = DictRegistry()
>>> r['myns'].register("mykey", "something")
>>> len(r['myns'])
1
>>> r['myns']["mykey"]
'something'
>>> "mykey" in r['myns']
True
>>> for k, v in r['myns'].items():
...     print("%s: %s" % (k,v))
mykey: something
```

register(*key*, *value*)
Register a new object under a given key.

Parameters

- **key** – Key to register object under
- **item** – Object to register

unregister(*key*)
Unregister an object under a given key. Raises KeyError in case the given key doesn't exists.

class flask_registry.registries.core.**SingletonRegistry**
Bases: *flask_registry.base.RegistryBase*

Basic registry that just keeps a single object.

```
>>> from flask import Flask
>>> from flask_registry import Registry, SingletonRegistry
>>> app = Flask('myapp')
>>> r = Registry(app=app)
>>> r['singleton'] = SingletonRegistry()
>>> r['singleton'].register("test string")
>>> r['singleton'].get()
'test string'
>>> r['singleton'].register("another string")
Traceback (most recent call last):
    ...
RegistryError: Object already registered.
>>> r['singleton'].unregister()
>>> r['singleton'].get() is None
True
>>> r['singleton'].unregister()
Traceback (most recent call last):
    ...
RegistryError: No object to unregister.
```

get()
Get the registered object

register(*obj*)
Register a new singleton object

Parameters **obj** – The object to register

unregister()
Unregister the singleton object

class flask_registry.registries.core.**ImportPathRegistry**(*initial=None*, *exclude=None*, *load_modules=False*)
Bases: *flask_registry.registries.core.ListRegistry*

Registry of Python import paths.

Supports simple discovery of modules without loading them.

```
>>> from flask import Flask
>>> from flask_registry import Registry, ImportPathRegistry
>>> app = Flask('myapp')
>>> r = Registry(app=app)
>>> r['myns'] = ImportPathRegistry(initial=[
... 'flask_registry.registries.*',
... 'flask_registry'])
>>> for imp_path in r['myns']:
...     print(imp_path)
flask_registry.registries.appdiscovery
flask_registry.registries.core
flask_registry.registries.moduleddiscovery
flask_registry.registries.pkgresources
flask_registry
```

When using star imports it is sometimes useful to exclude certain imports:

```
>>> r['myns2'] = ImportPathRegistry(
... initial=['flask_registry.registries.*',     ],
... exclude=['flask_registry.registries.core']
... )
>>> for imp_path in r['myns2']:
...     print(imp_path)
flask_registry.registries.appdiscovery
flask_registry.registries.moduleddiscovery
flask_registry.registries.pkgresources
```

Parameters

- **initial** – List of initial import paths.
- **exclude** – A list of import paths to not register. Useful together with star imports (`'*'`). Defaults to `[]`.
- **load_modules** – Load the modules instead of just registering the import path. Defaults to `False`.

register(*import_path*)
Register a new import path.

Parameters **import_path** – A full Python import path (e.g. `somepackge.somemodule`) or Python star import path to find all modules inside a package (e.g. `somepackge.*`).

unregister(**args*, ***kwargs*)
It is not possible to unregister import paths.

class flask_registry.registries.core.**ModuleRegistry**(*with_setup=True*)
Bases: *flask_registry.registries.core.ListRegistry*

Registry for Python modules with setup and teardown functionality.

Each module may provide a `setup()` and `teardown()` function which will be called when the module is registered. The name of the methods can be customized by subclassing and setting the class attributes `setup_func_name` and `teardown_func_name`.

Any extra arguments and keyword arguments to `register` and `unregister` is passed to the setup and teardown functions.

Example:

```
import mod

registry = ModuleRegistry(with_setup=True)
registry.register(mod, arg1, arg2, kw1=...)
# Will call mod.setup(arg1, arg2, kw1=...)
```

Parameters **with_setup** – Call setup/teardown function when registering/unregistering modules. Defaults to `True`.

register(*module*, **args*, ***kwargs*)
TODO.

Parameters

- **module** – Module to register.
- **args** – Argument passed to the module setup function.
- **kwargs** – Keyword argument passed to the module setup function.

setup_func_name = 'setup'
Name of setup function. Defaults to `setup`.

teardown_func_name = 'teardown'
Name of teardown function. Defaults to `teardown`.

unregister(*module*, **args*, ***kwargs*)
TODO.

Parameters

- **module** – Module to unregister.
- **args** – Argument passed to the module teardown function.
- **kwargs** – Keyword argument passed to the module teardown function.

Application discovery registries.

They provide discovery functionality useful for dynamically constructing Flask applications based on configuration variables. This allows a developer to package config, blueprints and extensions into isolated and reusable packages which a framework can dynamically install into a Flask application.

Such a package (named `registry_module`) could look like and it is located in `tests` directory:

- `registry_module.views` – contains blueprints which should be registered on the application object.
- `registry_module.mockext` – contains a `setup_app()` method which be used to install any Flask extensions on the application object.
- `registry_module.config` – contains configuration variables specific for this module.

Following is a simplified example of a Flask application factory, that will load config, extensions and blueprints:

```
>>> from flask import Flask, Blueprint
>>> from flask_registry import Registry, PackageRegistry
>>> from flask_registry import ExtensionRegistry
>>> from flask_registry import ConfigurationRegistry
>>> from flask_registry import BlueprintAutoDiscoveryRegistry
>>> class Config(object):
...     PACKAGES = ['registry_module']
...     EXTENSIONS = ['registry_module.mockext']
...     USER_CFG = True
>>> def create_app(config):
...     app = Flask('myapp')
...     app.config.from_object(config)
...     r = Registry(app=app)
...     r['packages'] = PackageRegistry(app)
...     r['extensions'] = ExtensionRegistry(app)
...     r['config'] = ConfigurationRegistry(app)
...     r['blueprints'] = BlueprintAutoDiscoveryRegistry(app=app)
...     return app
>>> config = Config()
>>> app = create_app(config)
```

3.1.2 Packages

The config variable `PACKAGES` specifies the list of Python packages, which `ConfigurationRegistry` and `BlueprintAutoDiscoveryRegistry` will search for `config.py` and `views.py` modules inside.

```
>>> for pkg in app.extensions['registry']['packages']:
...     print(pkg)
registry_module
```

3.1.3 Extensions

The config variable `EXTENSIONS` specifies the list of Python packages, which the `ExtensionRegistry` will load and call `setup_app(app)` on, to dynamically initialize Flask extensions.

```
>>> for pkg in app.extensions['registry']['extensions']:
...     print(pkg)
registry_module.mockext
```

3.1.4 Configuration

The `ConfigurationRegistry` will merge any package defined config, with the application config without overwriting already set variables in the application config:

```
>>> config.USER_CFG
True
>>> import registry_module.config
>>> registry_module.config.USER_CFG
False
>>> app.config['USER_CFG']
True
```

3.1.5 Blueprints

The `BlueprintAutoDiscoveryRegistry` will search for blueprints defined inside a `views` module in each package defined in `PACKAGES`. It will also register the discovered blueprints on the Flask application. Each `views` module should define either a single blueprint in the variable `blueprint` and/or multiple blueprints in the variable `blueprints`:

```
>>> from registry_module import views
>>> isinstance(views.blueprint, Blueprint)
True
>>> len(views.blueprints)
2
>>> for k in sorted(app.blueprints.keys()):
...     print(k)
test
test1
test2
```

class `flask_registry.registries.appdiscovery.`**`PackageRegistry`**(*app*)
Bases: *flask_registry.registries.core.ImportPathRegistry*

Specialized `ImportPathRegistry` that takes the initial list of import paths from the `PACKAGES` configuration variable in the application.

Parameters **app** – The Flask application object from which includes a `PACKAGES` variable in it's configuration.

class `flask_registry.registries.appdiscovery.`**`ExtensionRegistry`**(*app*)
Bases: *flask_registry.registries.core.ListRegistry*

Flask extensions registry.

Loads all extensions specified by `EXTENSIONS` configuration variable. The registry will look for a `setup_app` function in the extension and call it if it exists.

Example configuration:

```
EXTENSIONS = [
    'invenio.ext.debug_toolbar',
    'invenio.ext.menu:MenuAlchemy',
]
```

Parameters **app** – Flask application to get configuration from.

`register`(*app*, *ext_name*)
Register a Flask extensions and call `setup_app()` on it.

Parameters

- **app** – Flask application object
- **ext_name** – An import path (e.g. a package, module, object) which when loaded has an method `setup_app()`.

`unregister`()
It is not possible to unregister configuration.

class `flask_registry.registries.appdiscovery.`**`ConfigurationRegistry`**(*app*, *registry_namespace=None*)
Bases: *flask_registry.registries.moduledicovery.ModuleDiscoveryRegistry*

Specialized `ModuleDiscoveryRegistry` that search for `config` modules in a list of Python packages and merge them into the Flask application config without overwriting already set variables.

Parameters

- **app** – A Flask application
- **registry_namespace** – The registry namespace of an `ImportPathRegistry` with a list Python packages to search for `config` modules in. Defaults to `packages`.

register(*new_object*)
Register a new `config` module.

Parameters **new_object** – The configuration module. `app.config.from_object()` will be called on it.

unregister(**args*, ***kwargs*)
It is not possible to unregister configuration.

class flask_registry.registries.appdiscovery.**BlueprintAutoDiscoveryRegistry**(*module_name=None*, *app=None*, *with_setup=False*, *silent=False*)

Bases: *flask_registry.registries.moduleddiscovery.ModuleAutoDiscoveryRegistry*

Specialized `ModuleAutoDiscoveryRegistry` that search for `views` modules in a list of Python packages and register blueprints found inside them.

Blueprints are loaded by searching for a variable `blueprints` (list of Blueprint instances) or `blueprint` (a Blueprint instance). If found, the blueprint will be registered on the Flask application.

A blueprint URL prefix can be overwritten using the `BLUEPRINTS_URL_PREFIXES` variable in the application configuration:

```
BLUEPRINTS_URL_PREFIXES = {
    '<blueprint name>': '<new url prefix>',
    # ...
}
```

The module discovery registries.

They provide discovery functionality useful for searching a list of Python packages for a specific module name, and afterwards registering the module. This is used to e.g. load and register Flask blueprints by `BlueprintAutoDiscoveryRegistry`.

Assume e.g. we want to discover the `helpers` module from the `tests` package. First we initialize the registry:

```
>>> from flask import Flask
>>> from flask_registry import Registry, ModuleDiscoveryRegistry
>>> from flask_registry import ImportPathRegistry
>>> app = Flask('myapp')
>>> r = Registry(app=app)
```

We then create the list of packages to search through using an `ImportPathRegistry`:

```
>>> r['mypackages'] = ImportPathRegistry(initial=['registry_module'])
```

Then, initialize the `ModuleDiscoveryRegistry` and run the discovery:

```
>>> r['mydiscoveredmodules'] = ModuleDiscoveryRegistry(
...     'helpers', registry_namespace='mypackages')
>>> len(r['mydiscoveredmodules'])
0
```

```
>>> r['mydiscoveredmodules'].discover(app=app)
>>> len(r['mydiscoveredmodules'])
1
```

3.1.6 Lazy discovery

Using `RegistryProxy` you may lazily discover modules. Above example using lazy loading looks like this:

```
>>> from flask_registry import RegistryProxy
>>> app = Flask('myapp')
>>> r = Registry(app=app)
>>> pkg_proxy = RegistryProxy('mypackages', ImportPathRegistry,
...     initial=['registry_module'])
>>> mod_proxy = RegistryProxy('mydiscoveredmodules',
...     ModuleDiscoveryRegistry,
...     'helpers',
...     registry_namespace=pkg_proxy)
>>> 'mypackages' in r
False
>>> 'mydiscoveredmodules' in r
False
>>> with app.app_context():
...     mod_proxy.discover(app=app)
>>> 'mypackages' in r
True
>>> 'mydiscoveredmodules' in r
True
```

class `flask_registry.registries.modulediscovery.`**`ModuleDiscoveryRegistry`**(*module_name*, *registry_namespace=None*, *with_setup=False*, *silent=False*)

Bases: *flask_registry.registries.core.ModuleRegistry*

Specialized `ModuleRegistry` that will search a list of Python packages in an `ImportPathRegistry` or `ModuleRegistry` for a specific module name. By default the list of Python packages is read from the `packages` registry namespace.

Packages may be excluded during the discovery using a configuration variables constructed according to the following pattern:

```
<NAMESPACE>_<MODULE_NAME>_EXCLUDE
```

where `<NAMESPACE>` should be replaced by the `registry_namepsace`, and `<MOUDLE_NAME>` should be replaced with `module_name`. Example: `PACKAGES_VIEWS_EXCLUDE`. All namespaces are capitalized and have dots replaced with underscores.

Subclasses of `ModuleDiscoveryRegistry` may overwrite the internal `_discover_module()` method to provide specialized discovery (see e.g. `BlueprintAutoDiscoveryRegistry`).

Parameters

- **`module_name`** – Name of module to search for in packages.
- **`registry_namespace`** – The registry namespace of an `ImportPathRegistry` or `ModuleRegistry` with a list Python packages to search for `module_name` modules in. Alternatively to a registry namespace an instance of a `RegistryProxy` or `Registry` may also be used. Defaults to `packages`.

- **with_setup** – Call setup and teardown function on discovered modules. Defaults to `False` (see `ModuleRegistry`).
- **silent** – if set to True import errors are ignored. Defaults to `False`.

discover(*app=None*)
Perform module discovery.

It does so by iterating over the list of Python packages in the order they are specified.

Parameters **app** – Flask application object from where the list of Python packages is loaded (from the `registry_namespace`). Defaults to `current_app` if not specified (thus requires you are working in the Flask application context).

class `flask_registry.registries.moduleautodiscovery.`**ModuleAutoDiscoveryRegistry**(*module_name*, *app=None*, *registry_namespace=None*, *with_setup=False*, *silent=False*)
Bases: *flask_registry.registries.moduleautodiscovery.ModuleDiscoveryRegistry*

Specialized `ModuleDiscoveryRegistry` that will discover modules immediately on initialization.

Parameters

- **module_name** – Name of module to search for in packages.
- **app** – Flask application object
- **registry_namespace** – The registry namespace of an `ImportPathRegistry` or `ModuleRegistry` with a list Python packages to search for `module_name` modules in. Alternatively to a registry namespace an instance of a `RegistryProxy` or `Registry` may also be used. Defaults to `packages`.
- **with_setup** – Call setup and teardown function on discovered modules. Defaults to `False` (see `ModuleRegistry`).
- **silent** – if set to True import errors are ignored. Defaults to `False`.

3.1.7 Package Resources

Package resource registries may be used to discover e.g. package resources as well as loading entry points.

Entry points

`setuptools` entry points are a simple way for packages to "advertise" Python objects, so that frameworks can search for these entry points. `setup.py` files for instance allows you to specify `console_scripts` entry points, which will install scripts into system path for you.

The `EntryPointRegistry` allows you to easily register these entry points into your Flask application:

```
>>> from flask import Flask
>>> from flask_registry import Registry, EntryPointRegistry
>>> app = Flask('myapp')
>>> r = Registry(app=app)
>>> r['scripts'] = EntryPointRegistry('console_scripts')
>>> 'easy_install' in r['scripts']
True
```

Entry points are specified in you `setup.py`, e.g.:

```
setup(
    # ...
    entry_points={
        'flask_registry.test_entry': [
            'testcase = flask_registry:RegistryBase',
        ]
    },
    # ...
)
```

```
>>> r['entrypoints'] = EntryPointRegistry(
...     'flask_registry.test_entry', load=True)
>>> 'testcase' in r['entrypoints']
True
>>> from flask_registry import RegistryBase
>>> r['entrypoints']['testcase'][0] == RegistryBase
True
```

See http://pythonhosted.org/setuptools/pkg_resources.html#entry-points for more information on entry points.

Resource files

The `PkgResourcesDirDiscoveryRegistry` will search a list of Python packages for a specific resource directory and register all files found in the directories.

Assume e.g. a package `tests` have a directory `resources` with one file in it called `testresource.cfg`. This file can be discovered in the following manner:

```
>>> import os
>>> app = Flask('myapp')
>>> r = Registry(app=app)
>>> from flask_registry import ImportPathRegistry
>>> from flask_registry import PkgResourcesDirDiscoveryRegistry
>>> r['packages'] = ImportPathRegistry(initial=['registry_module'])
>>> r['res'] = PkgResourcesDirDiscoveryRegistry('resources', app=app)
>>> os.path.basename(r['res'][0]) == 'testresource.cfg'
True
```

class `flask_registry.registries.pkgresources.`**`EntryPointRegistry`**(*entry_point_ns*, *load=True*, *initial=None*, *exclude=None*, *unique=False*)

Bases: *flask_registry.registries.core.DictRegistry*

Entry point registry. Based on `DictRegistry` with keys being the entry point group, and the value being a list of objects referenced by the entry points.

Parameters

- **`entry_point_ns`** – Entry point namespace
- **`load`** – if False, entry point will not be loaded. Defaults to `True`.
- **`initial`** – List of initial names. If `None` it defaults to all.
- **`exclude`** – A list of names to not register. Useful together with initial equals to `None`. Defaults to `[]`.

- **unique** – Allow only unique options in entry point group if `True`.

register(*entry_point*)
Register a new entry point

Parameters **entry_point** – The entry point

class flask_registry.registries.pkgresources.**PkgResourcesDirDiscoveryRegistry**(*module_name*, *app=None*, *registry_namespace=None*, *with_setup=False*, *silent=False*)

Bases: *flask_registry.registries.moduleDiscovery.ModuleAutoDiscoveryRegistry*

Specialized `ModuleAutoDiscoveryRegistry` that will search a list of Python packages in an `ImportPathRegistry` or `ModuleRegistry` for a specific resource directory and register all files found in the directories. By default the list of Python packages is read from the `packages` registry namespace.

CHAPTER 4

Additional Notes

Notes on how to contribute, legal information and changelog are here for the interested.

4.1 Contributing

See <http://inveniosoftware.org/wiki/Development/Contributing> for now.

4.2 Changelog

Here you can see the full list of changes between each Flask-Registry release.

4.2.1 Version 0.2.0 (released 2014-06-27)

- ListRegistry now fuly behaves as a list.
- DictRegistry now fuly behaves as a dict.
- Fixes issue with app in ModuleAutoDiscoveryRegistry.
- Excludes option for ImportPathRegistry.
- Fixes handling of missing package resource directory.
- Fixes issue in configuration loading.
- Allows removal of registries.
- Fixes ImportError and SyntaxError handling.
- Documentation and code coverage improvements.
- Differentiates between missing and broken modules.
- New BlueprintAutoDiscoveryRegistry.
- New SingletonRegistry.

4.2.2 Version 0.1

- Initial public release

4.3 License

4.3.1 Authors

Flask-Registry is developed for use in Invenio digital library software.

Contact us at info@inveniosoftware.org

Contributors

- Lars Holm Nielsen <lars.holm.nielsen@cern.ch>
- Jiri Kuncar <jiri.kuncar@cern.ch>
- Esteban J. G. Gabancho <esteban.jose.garcia.gabancho@cern.ch>
- Tibor Simko <tibor.simko@cern.ch>
- Yoan Blanc <yoan@dosimple.ch>
- Adrian Moennich <adrian.moennich@cern.ch>

f

B

C

D

E

F

G

I

L

M

N

P

R

S

T

U