IoC Documentation

Release 0.0.16

Thomas Rabaix

April 15, 2015

Contents

Contents:

CHAPTER 1

Installation

First, obtain Python and virtualenv if you do not already have them. Using a virtual environment will make the installation easier, and will help to avoid clutter in your system-wide libraries. You will also need Git in order to clone the repository.

Once you have these, create a virtual environment somewhere on your disk, then activate it:

```
virtualenv myproject
cd myproject
source bin/activate
```

Now you can install the related required packages:

```
pip install ioc
```

CHAPTER 2

References

2.1 Bootstrapping

Here a quick example on how to use the ioc to initialize a project.

First, create a `start.py` file with the following code.

```
import sys, logging, optparse

import ioc

def main():
    parser = optparse.OptionParser()
    parser.add_option("-e", "--env", dest="env", help="Define the environment", default='dev')
    parser.add_option("-d", "--debug", dest="debug", action="store_true", default=False)

    options, args = parser.parse_args()

    if options.debug:
        logging.basicConfig(level=logging.DEBUG)

    container = ioc.build([
        'config/services.yml',
        'config/parameters_%s.yml' % options.env,
    ])

    ## adapt this line depends on your need
    container.get('myservice').start()


if __name__ == "__main__":
    main()
```

Now you can create a `services.yml` containing services definitions:

```
parameters:
    app_name: My App

services:
    my.service:
        class: module.ClassName
        arg: [arg1, @my.second.service]
        kwargs:
```

```
            api_key:  '%external.service.api_key%'
            app_name: '%app.name%'

    my.second.service:
        class: logging.getLogger
        arguments:
            - 'logger_name'
```

If you need to have different configurations, another files can be defined. The switch will be done by the `start.py` script with the `env` option.

```
# configuration parameters_prod.yml
parameters:
    external.service.api_key: XXXXXX

# configuration parameters_dev.yml
parameters:
    external.service.api_key: YYYYYY
```

The project can be started by using:

```
python start.py -e prod
```

2.2 Extension

An extension is a class used to configure services. A vendor might want to expose a configuration file to automatically generated valid services.

Here a flask extension, `ioc.extra.flask.di.Extension`

```
import ioc.loader, ioc.component
import os

class Extension(ioc.component.Extension):
    def load(self, config, container_builder):

        path = os.path.dirname(os.path.abspath(__file__))

        # load an external file defining services
        loader = ioc.loader.YamlLoader()
        loader.load("%s/resources/config/flask.yml" % path, container_builder)

        # set default parameters into the container to be reuse by the container builder
        # or by external services
        container_builder.parameters.set('ioc.extra.flask.app.name', config.get('name', ''))
        container_builder.parameters.set('ioc.extra.flask.app.static_path', config.get('static_path',
        container_builder.parameters.set('ioc.extra.flask.app.static_url_path', config.get('static_ur
        container_builder.parameters.set('ioc.extra.flask.app.instance_path', config.get('instance_pa
        container_builder.parameters.set('ioc.extra.flask.app.template_folder', config.get('template_
        container_builder.parameters.set('ioc.extra.flask.app.instance_relative_config', config.get('
        container_builder.parameters.set('ioc.extra.flask.app.port', config.get('port', 8080))
```

2.2.1 How to use an extensions

An extension is declared in the top yaml file by using its module name (`di.Extension` is added by the `ioc`), so in order to generate a flask instance just do:

```
ioc.extra.flask:
    port:               8080
    name:               ''
    static_path:        ''
    static_url_path:    ''
    static_folder:      'static'
    template_folder:    'templates'
    instance_path:      ''
    instance_relative_config: false
```

and to use it:

```
import ioc

container = ioc.build(['config.yml'])

app = container.get('ioc.extra.flask.app')

__name__ == '__main__':
    app.run()
```

2.2.2 Going further

The previous example is a bit overkill as Flask itself required a minimun amount of code to run. However the previous code allows to configure the default port which can be usefull for running different configurations.

Now, the `ioc.extra.flask.app` is available inside the container, so other services can use it.

The `shirka` project exposes some `flask` actions as service:

```
shirka.flask.view.shirka_proc_list:
    class: [shirka.control.proc, ProcListView.as_view]
    arguments:
        - shirka_proc_list
        - "%shirka.data.dir%/proc"

shirka.flask.view.shirka_proc_view:
    class: [shirka.control.proc, ProcView.as_view]
    arguments:
        - shirka_proc_view
        - "%shirka.data.dir%/proc"
```

So there are 2 actions as a service defined here : `shirka.flask.view.shirka_proc_view` and `shirka.flask.view.shirka_proc_list`. As you can note, we are injected custom parameters into each service, these parameters can be configured by the user in an external file.

The `shirka` project also provide a custom extension `shirka.di.Extension`, this extension will register theses services as methods call to the `ioc.extra.flask` service.

```
import ioc
import os

class Extension(ioc.component.Extension):
```

```
    def pre_build(self, container_builder, container):

        # if the service does not exist, then avoid registering services
        if not container_builder.has('ioc.extra.flask.app'):
            return

        definition = container_builder.get('ioc.extra.flask.app')

        base_url = container_builder.parameters.get('shirka.web.api.base_url')
        definition.method_calls.append([
            'add_url_rule',
            ['%s/process' % base_url],
            {'view_func': ioc.component.Reference('shirka.flask.view.shirka_proc_list')}
        ])

        definition.method_calls.append([
            'add_url_rule',
            ['%s/process/<id>' % base_url],
            {'view_func': ioc.component.Reference('shirka.flask.view.shirka_proc_view')}
        ])
```

The `pre_build` method is called after all extensions are loaded, this allow extensions to alter service definitions.

shirka configuration defined inside the `config.yml` file:

```
shirka:
    # public_dir:
    api:
        base_url: '/shirka/api'
    data_dir: '%base_dir%/data'
```

So through some configuration, the user can configure how the Flask action will be expose `/shirka/api`.

2.3 Tag

Service definition can be tagged in order to be found while the container is being build.

For instance, a jinja filters can be define like this:

```
jinja2.filter.time:
    class: jinja2.extra.filter.Time
    tags:
        jinja2.filter: # a filter can have multiple filter options
            - []
            - []
```

Then, while the container is being build, it is possible to attach new service to the `jinja` instance

```
class Extension(ioc.component.Extension):
    def load(self, config, container_builder):

        # ...

        jinja = container_builder.get('ioc.extra.jinja2')

        for id in container_builder.get_ids_by_tags('jinja2.filter'):
            definition = container_builder.get(id)
```

```
            for options in definition.get_tag('jinja2.filter'):
                jinja.add_call('register_filter', ioc.component.Reference(id))
```

2.4 Resource Locator

The resource locator is a set of classes to find ... resource. A resource is a file located on the filesystem.

2.4.1 Basic Usage

```
import ioc.locator

locator = ioc.locator.FileSystemLocator(['/path/to/templates', '/other/path'])

file = locator.locate("myfile.yml")

# file => is a local path to the file
```

2.5 Event Dispatcher

The ioc package provides an optional Event Dispatcher. The dispatcher is always set if you use the `ioc.build` function.

2.5.1 Basic Usage

```
import ioc.event

def mylistener(event):
    event.get('node')['value'] = event.get('node')['value'] * 60
    event.stop_propagation()

dispatcher = ioc.event.Dispatcher()
dispatcher.add_listener('event.name', mylistener)

event = dispatcher.dispatch('event.name', {
    'node': { 'value': 2 }
})
```

CHAPTER 3

Extra

The ioc package include some integration with some python libs, just edit the `config.yml` file and add the different following `yaml` sections.

3.1 Event Dispatcher

The IoC package includes a small event dispatcher, you can include it by adding this yaml.

3.1.1 Configuration

```
ioc.extra.event:
```

3.2 Flask

Flask is a web micro framework

3.2.1 Configuration

```
ioc.extra.flask:
    app:
        port:               8080
        name:               ''
        static_path:        ''
        static_url_path:    ''
        static_folder:      'static'
        template_folder:    'templates'
        instance_path:      ''
        instance_relative_config: false

    config: # use to populate the instance_relative_config kwargs
        DEBUG:                          False
        TESTING:                        False
        PROPAGATE_EXCEPTIONS:
        PRESERVE_CONTEXT_ON_EXCEPTION:
        SECRET_KEY:
        USE_X_SENDFILE:                 False
```

```
        LOGGER_NAME:
        SERVER_NAME:
        APPLICATION_ROOT:
        SESSION_COOKIE_NAME:            'session'
        SESSION_COOKIE_DOMAIN:
        SESSION_COOKIE_PATH:
        SESSION_COOKIE_HTTPONLY:        True
        SESSION_COOKIE_SECURE:          False
        MAX_CONTENT_LENGTH:
        SEND_FILE_MAX_AGE_DEFAULT:      43200
        TRAP_BAD_REQUEST_ERRORS:        False
        TRAP_HTTP_EXCEPTIONS:           False
        PREFERRED_URL_SCHEME:           'http'
        JSON_AS_ASCII:                  True

    blueprints:
        - element.flask.blueprint
```

3.2.2 Services available

Services available:

- ioc.extra.flask.app : the Flask app

3.3 Mailer

3.3.1 Configuration

```
ioc.extra.mailer:
    host: localhost
    port:
    use_tls: false
    user:
    password:
    use_ssl: false
```

3.4 Redis-Py

Redis-Py is an interface to the Redis key-value store.

3.4.1 Configuration

```
ioc.extra.redis:
    clients:
        default:
            connection: default

    connections:
        default:
            host:               'localhost'
```

```
        port:               6379
        db:                 0
        password:
        socket_timeout:
        encoding:           'utf-8'
        encoding_errors:    'strict'
        decode_responses:   false
```

3.4.2 Services available

- ioc.extra.redis.manager: the Redis manager to retrieve client and connection
- ioc.extra.redis.connection.default: the `default` connection
- ioc.extra.redis.client.default: the `default` client

3.5 redis_wrap

redis-wrap implements a wrapper for Redis datatypes so they mimic the datatypes found in Python

3.5.1 Configuration

```
ioc.extra.redis_wrap:
    clients:
        default: ioc.extra.redis.client.default
```

3.6 Twisted

3.6.1 Configuration

Twisted is an event-driven networking engine written.

```
ioc.extra.twisted:
```

3.6.2 Services available

- ioc.extra.twisted.reactor: the reactor instance
- ioc.extra.twisted.reactor.thread_pool: the reactor thread pool

CHAPTER 4

Indices and tables

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