iPOPO Documentation

Release 0.7.0

Thomas Calmant

May 12, 2018

Contents

iPOPO is a Python-based Service-Oriented Component Model (SOCM) based on Pelix, a dynamic service platform. They are inspired by two popular Java technologies for the development of long-lived applications: the iPOJO component model and the OSGi Service Platform. iPOPO enables the conception of long-running and modular IT services.

This documentation is divided into three main parts. The *Quickstart* will guide you to install iPOPO and write your first components. The *Reference Cards* section details the various concepts of iPOPO. Finally, the *Tutorials* explain how to use the various built-in services of iPOPO. You can also take a look at the slides of the iPOPO tutorial to have a quick overview of iPOPO.

This documentation is inspired by the Flask's one.

iPOPO depends on a fork of jsonrpclib, called jsonrpclib-pelix. The documentation of this library is available on GitHub.

CHAPTER 1

Usage survey

In order to gain insight from the iPOPO community, I've put a really short survey on Google Forms (no login required).

Please, feel free to answer it, the more answers, the better. All feedback is really appreciated, and I'll write about the aggregated results on the users' mailing list, once enough answers will have been received.

CHAPTER 2

State of this documentation

This documentation is a work in progress, starting nearly from scratch.

The previous documentation was provided as a wiki on a dedicated server which I had to take down due to many reasons (DoS attacks, update issues, . . .). As a result, the documentation is now hosted by Read the Docs. The main advantages are that it is now included in the Git repository of the project, and it can include *docstrings* directly from the source code.

Alas, the wiki content must be completely rewritten in reStructuredText format. I take this opportunity to update the documentation, but it takes a lot of time, and I can't work on this project as much as I'd like to. So, if you have any question which hasn't been answered in the current documentation, please ask on the users' mailing list.

As always, all contributions to the documentation and the code are very appreciated.

CHAPTER 3

User's Guide

This chapter details how to install and use iPOPO.

3.1 Foreword

This section describes the purpose and goals of the iPOPO project, as well as some background history.

3.1.1 What is iPOPO ?

iPOPO is a Python-based Service-Oriented Component Model (SOCM). It is split into two parts:

- Pelix, a dynamic service platform
- iPOPO, the SOCM framework, hence the name.

Both are inspired on two popular Java technologies for the development of long-lived applications: the OSGi Service Platform and the iPOJO component model.

iPOPO allows to conceive long-running and modular IT services in Python.

About the name, iPOPO is inspired from iPOJO, which stands for *injected Plain Old Java Object*. Java being replaced by Python, the name became iPOPO. The logo comes from the similarity of pronunciation with the french word for the hippo: *hippopotame*.

3.1.2 A bit of history

During my PhD thesis, I had to monitor applications built as multiple instances of OSGi frameworks and based on iPOJO components. This required to access some OS-specific low-level methods and was initially done in Java with JNA.

To ease the development of probes, the monitoring code has been translated to Python. At first, it was only a set of scripts without any relations, but as the project grown, it was necessary to develop a framework to handle those various parts and to link them together. In order to be consistent, I decided to develop a component model similar to what was used used in Java, *i.e.* iPOJO, and keeping the concepts of OSGi.

A first draft, called `python.injections` was developped in December 2011. It was a proof of concept which was good enough for my employer, isandlaTech (now Cohorte Technologies), to allow the development of what would become iPOPO.

The first public release was version 0.3 in April 2012, under the GPLv3 license. In November 2013, iPOPO adopts the Apache Software License 2.0 with release 0.5.5.

On March 2015, release 0.6 dropped support for Python 2.6. Since then, the development slowed down as the core framework is considered stable.

iPOPO 1.0 should be released mid-2017, when the remote services engine will be upgraded.

3.1.3 SOA and SOCM in Python

The Service-Oriented Architecture (SOA) consists in linking objects through provided contracts (services) registered in a service registry.

A service is an object associated to properties describing it, including the names of the contracts it implements. It is stored in the service registry of the framework by the service provider. The provider or the service itself (they are often the same) must handle the requirements, *i.e.* looking for the services required to work and handling their late un/registration.

A component is an object instanciated and handled by an instance manager created by iPOPO. The manager handles the life cycle of the component, looking for its dependencies and handling their late registration, unregistration and replacement. It eases the development and allows a lot of dynamism in an application.

The conclusion is that the parts of an application which only provide a feature can be written as a simple service, whereas parts using other elements of the application should be written as components.

Continue to *Installation*, the *Quickstart* or the *Tutorials*.

3.2 Installation

iPOPO depends on only one external library, jsonrpclib-pelix, which provides some utility methods and is required to enable remote services.

To install iPOPO, you will need Python 2.7, Python 3.3 or newer. iPOPO is constantly tested, using Tox and Travis-CI, on the following interpreters:

- Python 2.7, 3.3, 3.4, 3.5
- Pypy 2 et 3

Support for Python 2.6 has been dropped with iPOPO 0.6. The framework should run on Python 3.2, but it is not officialy supported.

There are many ways to install iPOPO, so let's have a look to some of them.

3.2.1 System-Wide Installation

This is the easiest way to install iPOPO, even though using virtual environments is recommended to develop your applications.

For a system-wide installation, just run `pip` with root privileges:

```
$ sudo pip install iPOPO
```

If you don't have root privileges and you can't or don't want to use virtual environments, you can install iPOPO for your user only:

```
$ pip install --user iPOPO
```

3.2.2 Virtual Environment

Using virtual environments is the recommended way to install libraries in Python. It allows to try and develop with specific versions of libraries, to test some packages, etc. without messing with your Python installation, nor your main development environment.

It is also useful in production, as virtual environment allows to isolate libraries, avoiding incompatibilities.

Python 3.3+

Python 3.3 introduced the `venv` module, introducing a standard way to handle virtual environments. As this module is included in the Python standard library, you shouldn't have to install it manually.

Now you can create a new virtual environment, here called *ipopo-venv*:

```
$ python3 -m venv ipopo-venv
```

Continue to *Then…* to activate your new environment.

Previous versions

Before Python 3.3, virtual environments were handled by a third-party package, `virtualenv`, which must be installed alongside Python.

If you are on Linux or Mac OS X, the following command should work:

```
$ sudo pip install virtualenv
```

On Linux, virtualenv is probably provided by your distribution. For example, you can use the following command on Debian or Ubuntu:

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

Once virtualenv is installed, you can create you first virtual environment:

```
$ virtualenv ipopo-venv
New python executable in ipopo-venv/bin/python
Installing setuptools, pip............done.
```

Then…

Now, whenever you want to work on this project, you will have to activate the virtual environment:

```
$ . ipopo-venv/bin/activate
```

If you are a Windows user, the following command is for you:

```
> ipopo-venv\Scripts\activate
```

Either way, you show now be using your virtual environment. The shell prompt should indicate it.

Now you can install iPOPO using `pip`. As this is a virtual environment, you don't need administration rights:

```
$ pip install iPOPO
```

iPOPO is now installed and can be used in this environment. You can now try it and develop your components.

Once you are done, you can get out of the virtual environment using the following command (both on Linux and Windows):

```
$ deactivate
```

3.2.3 Development version

If you want to work with latest version of iPOPO, there are two ways: you can either let `pip` pull in the development version, or you can tell it to operate on a git checkout. Either way, a virtual environment is recommended.

Get the git checkout in a new virtual environment and run in development mode:

```
$ git clone https://github.com/tcalmant/ipopo.git
# Cloning into 'ipopo'...
$ cd ipopo
$ python3 -m venv ipopo-venv
New python executable in ipopo-venv/bin/python
Installing setuptools, pip...........done.
$ . ipopo-venv/bin/activate
$ python setup.py develop
# ...
Finished processing dependencies for iPOPO
```

This will pull the dependency (*jsonrpclib-pelix*) and activate the git head as the current version inside the virtual environment. As the *develop* installation mode uses symbolic links, you simply have to run `git pull origin` to update to the latest version of iPOPO in your virtual environment.

3.3 Quickstart

Eager to get started? This page gives a good introduction to iPOPO. It assumes you already have iPOPO installed. If you do not, head over to the *Installation* section.

3.3.1 Play with the shell

The easiest way to see how iPOPO works is by playing with the builtin shell.

To start the shell locally, you can run the following command:

```
bash$ python -m pelix.shell
** Pelix Shell prompt **
$
```

Survival Kit

As always, the life-saving command is `help`:

```
$ help
=== Name space 'default' ===
- ? [<command>]
                Prints the available methods and their documentation,
                or the documentation of the given command.
- bd <bundle_id>
                Prints the details of the bundle with the given ID
                or name
- bl [<name>]
                Lists the bundles in the framework and their state.
                Possibility to filter on the bundle name.
...
$
```

The must-be-known shell commands of iPOPO are the following:

Command	Description
help	Shows the help
loglevel	Prints/Changes the log level
exit	Quits the shell (and stops the framework in console UI)
threads	Prints the stack trace of all threads
run	Runs a shell script

Bundle commands

The following commands can be used to handle bundles in the framework:

Command	Description
install	Installs a module as a bundle
start	Starts the given bundle
update	Updates the given bundle (restarts it if necessary)
uninstall	Uninstalls the given bundle (stops it if necessary)
bl	Lists the installed bundles and their state
bd	Prints the details of a bundle

In the following example, we install the `pelix.shell.remote` bundle, and play a little with it:

```
$ install pelix.shell.remote
Bundle ID: 12
$ start 12
Starting bundle 12 (pelix.shell.remote)...
$ bl
+----+----------------------------------------+--------+---------+
| ID |                  Name                  | State  | Version |
+====+========================================+========+=========+
| 0  | pelix.framework                        | ACTIVE | 0.6.4   |
+----+----------------------------------------+--------+---------+
...
+----+----------------------------------------+--------+---------+
| 12 | pelix.shell.remote                     | ACTIVE | 0.6.4   |
```

(continues on next page)

(continued from previous page)

```
+----+----------------------------------------+---------+----------+
13 bundles installed
$ update 12
Updating bundle 12 (pelix.shell.remote)...
$ stop 12
Stopping bundle 12 (pelix.shell.remote)...
$ uninstall 12
Uninstalling bundle 12 (pelix.shell.remote)...
$
```

While the `install` command requires the name of a module as argument, all other commands accepts a bundle ID as argument.

Service Commands

Services are handles by bundles and can't be modified using the shell. The following commands can be used to check the state of the service registry:

Command	Description
sl	Lists the registered services
sd	Prints the details of a services

This sample prints the details about the iPOPO core service:

```
$ sl
+----+---------------------------+----------------------------------------------------
↪+---------+
| ID |      Specifications       |                       Bundle                       ␣
↪| Ranking |
+====+===========================+====================================================+=========+
| 1  | ['ipopo.handler.factory'] | Bundle(ID=5, Name=pelix.ipopo.handlers.properties)␣
↪| 0       |
+----+---------------------------+----------------------------------------------------
↪+---------+
...
+----+---------------------------+----------------------------------------------------
↪+---------+
| 8  | ['pelix.ipopo.core']      | Bundle(ID=1, Name=pelix.ipopo.core)                ␣
↪| 0       |
+----+---------------------------+----------------------------------------------------
↪+---------+
...
11 services registered
$ sd 8
ID............: 8
Rank..........: 0
Specifications: ['pelix.ipopo.core']
Bundle........: Bundle(ID=1, Name=pelix.ipopo.core)
Properties....:
        objectClass = ['pelix.ipopo.core']
        service.id = 8
        service.ranking = 0
Bundles using this service:
        Bundle(ID=4, Name=pelix.shell.ipopo)
$
```

iPOPO Commands

iPOPO provides a set of commands to handle the components and their factories:

Command	Description
factories	Lists registered component factories
factory	Prints the details of a factory
instances	Lists components instances
instance	Prints the details of a component
waiting	Lists the components waiting for an handler
instantiate	Starts a new component instance
kill	Kills a component
retry	Retry the validation of a component

This snippets installs the `pelix.shell.remote` bundle and instantiate a new remote shell component:

```
$ install pelix.shell.remote
Bundle ID: 12
$ start 12
Starting bundle 12 (pelix.shell.remote)...
$ factories
+------------------------------+---------------------------------------+
|           Factory            |                Bundle                 |
+==============================+=======================================+
| ipopo-remote-shell-factory   | Bundle(ID=12, Name=pelix.shell.remote) |
+------------------------------+---------------------------------------+
| ipopo-shell-commands-factory | Bundle(ID=4, Name=pelix.shell.ipopo)   |
+------------------------------+---------------------------------------+
2 factories available
$ instantiate ipopo-remote-shell-factory rshell pelix.shell.address=0.0.0.0 pelix.
↪shell.port=9000
Component 'rshell' instantiated.
```

A remote shell as been started on port 9000 and can be accessed using Netcat:

```
bash$ nc localhost 9000
------------------------------------------------------------------------
** Pelix Shell prompt **

iPOPO Remote Shell
------------------------------------------------------------------------
$
```

The remote shell gives access to the same commands as the console UI. Note that an XMPP version of the shell also exists.

To stop the remote shell, you have to kill the component:

```
$ kill rshell
Component 'rshell' killed.
```

Finally, to stop the shell, simply run the `exit` command or press `Ctrl+D`.

3.3.2 Hello World!

In this section, we will create a service provider and its consumer using iPOPO. The consumer will use the provider to print a greeting message as soon as it is bound to it. To simplify this first sample, the consumer can only be bound to a single service and its life-cycle is highly tied to the availability of this service.

Here is the code of the provider component, which should be store in the `provider` module (`provider.py`). The component will provide a service with of the `hello.world` specification.

```python
from pelix.ipopo.decorators import ComponentFactory, Provides, Instantiate

# Define the component factory, with a given name
@ComponentFactory("service-provider-factory")
# Defines the service to provide when the component is active
@Provides("hello.world")
# A component must be instantiated as soon as the bundle is active
@Instantiate("provider")
# Don't forget to inherit from object, for Python 2.x compatibility
class Greetings(object):
      def hello(self, name="World"):
          print("Hello,", name, "!")
```

Start a Pelix shell like shown in the previous section, then install and start the provider bundle:

```
** Pelix Shell prompt **
$ install provider
Bundle ID: 12
$ start 12
Starting bundle 12 (provider)...
$
```

The consumer will require the `hello.world` service and use it when it is validated, *i.e.* once this service has been injected. Here is the code of this component, which should be store in the `consumer` module (`consumer.py`).

```python
from pelix.ipopo.decorators import ComponentFactory, Requires, Instantiate, \
     Validate, Invalidate

# Define the component factory, with a given name
@ComponentFactory("service-consumer-factory")
# Defines the service required by the component to be active
# The service will be injected in the '_svc' field
@Requires("_svc", "hello.world")
# A component must be instantiated as soon as the bundle is active
@Instantiate("consumer")
# Don't forget to inherit from object, for Python 2.x compatibility
class Consumer(object):
      @Validate
      def validate(self, context):
          print("Component validated, calling the service...")
          self._svc.hello("World")
          print("Done.")

      @Invalidate
      def invalidate(self, context):
         print("Component invalidated, the service is gone")
```

Install and start the `consumer` bundle in the active Pelix shell and play with the various commands described in the *previous section*:

```
$ install consumer
Bundle ID: 13
$ start 13
Starting bundle 13 (consumer)...
Component validated, calling the service...
Hello, World !
Done.
$ update 12
Updating bundle 12 (provider)...
Component invalidated, the service is gone
Component validated, calling the service...
Hello, World !
Done.
$ uninstall 12
Uninstalling bundle 12 (provider)...
Component invalidated, the service is gone
```

3.3.3 Hello from somewhere!

This section reuses the bundles written in the *Hello World* sample, and install them in two distinct frameworks. To achieve that, we will use the *Pelix Remote Services*, a set of bundles intending to share services across multiple Pelix frameworks. A *reference card* provides more information about this feature.

Core bundles

First, we must install the core bundles of the *remote services* implementation: the *Imports Registry* (`pelix.remote.registry`) and the *Exports Dispatcher* (`pelix.remote.dispatcher`). Both handle the description of the shared services, not their link with the framework: it will be the job of discovery and transport providers. The discovery provider we will use requires to access the content of the *Exports Dispatcher* of the frameworks it finds, through HTTP requests. A component, the *dispatcher servlet*, must therefore be instantiate to answer to those requests:

```
bash$ python -m pelix.shell
** Pelix Shell prompt **
$ install pelix.remote.registry
Bundle ID: 11
$ start 11
Starting bundle 11 (pelix.remote.registry)...
$ install pelix.remote.dispatcher
Bundle ID: 12
$ start 12
Starting bundle 12 (pelix.remote.dispatcher)...
$ instantiate pelix-remote-dispatcher-servlet-factory dispatcher-servlet
Component 'dispatcher-servlet' instantiated.
```

The protocols we will use for discovery and transport depends on an HTTP server. As we are using two framework on the same machine, don’t forget to use different HTTP ports for each framework:

```
$ install pelix.http.basic
Bundle ID: 13
$ start 13
Starting bundle 13 (pelix.http.basic)...
$ instantiate pelix.http.service.basic.factory httpd pelix.http.port=8000
INFO:httpd:Starting HTTP server: [0.0.0.0]:8000 ...
```

(continues on next page)

(continued from previous page)

```
INFO:httpd:HTTP server started: [0.0.0.0]:8000
Component 'httpd' instantiated.
```

The *dispatcher servlet* will be discovered by the newly started HTTP server and will be able to answer to clients.

Discovery and Transport

Next, it is necessary to setup the remote service discovery layer. Here, we'll use a Pelix-specific protocol based on UDP multicast packets. By default, this protocol uses the UDP port 42000, which must therefore be accessible on any machine providing or consuming a remote service.

Start two Pelix framework with their shell and, in each one, install the `pelix.remote.discovery.multicast` bundle then instantiate the discovery component:

```
$ install pelix.remote.discovery.multicast
Bundle ID: 14
$ start 14
Starting bundle 14 (pelix.remote.discovery.multicast)...
$ instantiate pelix-remote-discovery-multicast-factory discovery
Component 'discovery' instantiated.
```

Finally, you will have to install the transport layer that will be used to send requests and to wait for their responses. Here, we'll use the JSON-RPC protocol (`pelix.remote.json_rpc`), which is the easiest to use (*e.g.* XML-RPC has problems handling dictionaries of complex types). Transport providers often require to instantiate two components: one for the export and one for the import. This allows to instantiate the export part only, avoiding every single framework to know about all available services.

> $ install pelix.remote.json_rpc Bundle ID: 15 $ start 15 Starting bundle 15 (pelix.remote.json_rpc)... $ instantiate pelix-jsonrpc-importer-factory importer Component 'importer' instantiated. $ instantiate pelix-jsonrpc-exporter-factory exporter Component 'exporter' instantiated.

Now, the frameworks you ran have all the necessary bundles and services to detect and use the services of their peers.

Export a service

Exporting a service is as simple as providing it: just add the `service.exported.interfaces` property while registering it and will be exported automatically. To avoid typos, this property is defined in the `pelix.remote.PROP_EXPORTED_INTERFACES` constant. This property can contain either a list of names of interfaces/contracts or a star (`*`) to indicate that all services interfaces are exported.

Here is the new version of the *hello world* provider, with the export property:

```
from pelix.ipopo.decorators import ComponentFactory, Provides, \
    Instantiate, Property
from pelix.remote import PROP_EXPORTED_INTERFACES

@ComponentFactory("service-provider-factory")
@Provides("hello.world")
# Here is the new property, to authorize the export
@Property('_export_itfs', PROP_EXPORTED_INTERFACES, '*')
@Instantiate("provider")
class Greetings(object):
      def hello(self, name="World"):
          print("Hello,", name, "!")
```

That's all!

Now you can install this provider in a framework, using:

```
$ install provider
Bundle ID: 16
$ start 16
Starting bundle 16 (provider)...
```

When installing a consumer in another framework, it will see the provider and use it:

```
$ install consumer
Bundle ID: 16
$ start 16
Component validated, calling the service...
Done.
```

You should then see the greeting message (*Hello, World !*) in the shell of the provider that has been used by the consumer.

3.4 Tutorials

This section provides tutorials for various parts of iPOPO.

3.4.1 iPOPO in 10 minutes

Authors Shadi Abras, Thomas Calmant

This tutorial presents how to use the iPOPO framework and its associated service-oriented component model. The concepts of the service-oriented component model are introduced, followed by a simple example that demonstrates the features of iPOPO. This framework uses decorators to describe components.

Introduction

iPOPO aims to simplify service-oriented programming on OSGi frameworks in Python language; the name iPOPO is an abbreviation for *injected POPO*, where *POPO* would stand for Plain Old Python Object. The name is in fact a simple modification of the Apache iPOJO project, which stands for *injected Plain Old Java Object*

iPOPO provides a new way to develop OSGi/iPOJO-like service components in Python, simplifying service component implementation by transparently managing the dynamics of the environment as well as other non-functional requirements. The iPOPO framework allows developers to more clearly separate functional code (*i.e.* POPOs) from the non-functional code (*i.e.* dependency management, service provision, configuration, etc.). At run time, iPOPO combines the functional and non-functional aspects. To achieve this, iPOPO provides a simple and extensible service component model based on POPOs.

Basic concepts

iPOPO is separated into two parts:

- Pelix, the underlying bundle and service registry
- iPOPO, the service-oriented component framework

It also defines three major concepts:

- A *bundle* is a single Python module, *i.e.* a `.py` file, that is loaded using the Pelix API.
- A *service* is a Python object that is registered to service registry using the Pelix API, associated to a set of specifications and to a dictionary of properties.
- A *component* is an instance of *component factory*, *i.e.* a class manipulated by iPOPO decorators. Those decorators injects information into the class that are later used by iPOPO to manage the components. Components are defined inside bundles.

Simple example

In this tutorial we will present how to:

- Publish a service
- Require a service
- Use lifecycle callbacks to activate and deactivate components

Presentation of the Spell application

To illustrate some of iPOPO features, we will implement a very simple application. Three bundles compose this application:

- A bundle that defines a component implementing a dictionary service (an English and a French dictionaries).
- One with a component requiring the dictionary service and providing a spell checker service.
- One that defines a component requiring the spell checker and providing a user line interface.

The spell dictionary components provide the `spell_dictionary_service` specification. The spell checker provides a `spell_checker_service` specification.

Preparing the tutorial

The example contains several bundles:

- spell_dictionary_EN.py defines a component that implements the Dictionary service, containing some English words.
- spell_dictionary_FR.py defines a component that implements the Dictionary service, containing some French words.
- spell_checker.py contains an implementation of a Spell Checker. The spell checker requires a dictionary service and checks if an input passage is correct, according to the words contained in the wished dictionary.
- spell_client.py provides commands for the *Pelix shell service*. This component uses a spell checker service. The user can interact with the spell checker with this command line interface.

Finally, a main_pelix_launcher.py script starts the Pelix framework. It is not considered as a bundle as it is not loaded by the framework, but it can control the latter.

The English dictionary bundle: Providing a service

The `spell_dictionary_EN` bundle is a simple implementation of the Dictionary service. It contains few English words.

```python
#!/usr/bin/python
# -- Content-Encoding: UTF-8 --
"""
This bundle provides a component that is a simple implementation of the
Dictionary service. It contains some English words.
"""

# iPOPO decorators
from pelix.ipopo.decorators import ComponentFactory, Property, Provides, \
    Validate, Invalidate, Instantiate


# Name the iPOPO component factory
@ComponentFactory("spell_dictionary_en_factory")
# This component provides a dictionary service
@Provides("spell_dictionary_service")
# It is the English dictionary
@Property("_language", "language", "EN")
# Automatically instantiate a component when this factory is loaded
@Instantiate("spell_dictionary_en_instance")
class SpellDictionary(object):
    """
    Implementation of a spell dictionary, for English language.
    """

    def __init__(self):
        """
        Declares members, to respect PEP-8.
        """
        self.dictionary = None

    @Validate
    def validate(self, context):
        """
        The component is validated. This method is called right before the
        provided service is registered to the framework.
        """
        # All setup should be done here
        self.dictionary = {"hello", "world", "welcome", "to", "the", "ipopo",
                           "tutorial"}
        print('An English dictionary has been added')

    @Invalidate
    def invalidate(self, context):
        """
        The component has been invalidated. This method is called right after
        the provided service has been removed from the framework.
        """
        self.dictionary = None

    def check_word(self, word):
        """
```

(continues on next page)

(continued from previous page)

```
        Determines if the given word is contained in the dictionary.

        @param word the word to be checked.
        @return True if the word is in the dictionary, False otherwise.
        """
        word = word.lower().strip()
        return not word or word in self.dictionary
```

- The `@Component` decorator is used to declare an iPOPO component. It must always be on top of other decorators.
- The `@Provides` decorator indicates that the component provides a service.
- The `@Instantiate` decorator instructs iPOPO to automatically create an instance of our component. The relation between components and instances is the same than between classes and objects in the object-oriented programming.
- The `@Property` decorator indicates the properties associated to this component and to its services (*e.g.* French or English language).
- The method decorated with `@Validate` will be called when the instance becomes valid.
- The method decorated with `@Invalidate` will be called when the instance becomes invalid (*e.g.* when one its dependencies goes away) or is stopped.

For more information about decorators, see :ref:refcard_decorators.

The French dictionary bundle: Providing a service

The `spell_dictionary_FR` bundle is a similar to the `spell_dictionary_EN` one. It only differs in the `language` component property, as it checks some French words declared during component validation.

```
#!/usr/bin/python
# -- Content-Encoding: UTF-8 --
"""
This bundle provides a component that is a simple implementation of the
Dictionary service. It contains some French words.
"""

# iPOPO decorators
from pelix.ipopo.decorators import ComponentFactory, Property, Provides, \
    Validate, Invalidate, Instantiate


# Name the iPOPO component factory
@ComponentFactory("spell_dictionary_fr_factory")
# This component provides a dictionary service
@Provides("spell_dictionary_service")
# It is the French dictionary
@Property("_language", "language", "FR")
# Automatically instantiate a component when this factory is loaded
@Instantiate("spell_dictionary_fr_instance")
class SpellDictionary(object):
    """
    Implementation of a spell dictionary, for French language.
    """

```

(continues on next page)

(continued from previous page)

```python
    def __init__(self):
        """
        Declares members, to respect PEP-8.
        """
        self.dictionary = None

    @Validate
    def validate(self, context):
        """
        The component is validated. This method is called right before the
        provided service is registered to the framework.
        """
        # All setup should be done here
        self.dictionary = {"bonjour", "le", "monde", "au", "a", "ipopo",
                           "tutoriel"}
        print('A French dictionary has been added')

    @Invalidate
    def invalidate(self, context):
        """
        The component has been invalidated. This method is called right after
        the provided service has been removed from the framework.
        """
        self.dictionary = None

    def check_word(self, word):
        """
        Determines if the given word is contained in the dictionary.

        @param word the word to be checked.
        @return True if the word is in the dictionary, False otherwise.
        """
        word = word.lower().strip()
        return not word or word in self.dictionary
```

It is important to note that the iPOPO factory name must be unique in a framework: only the first one to be registered with a given name will be taken into account. The name of component instances follows the same rule.

The spell checker bundle: Requiring a service

The `spell_checker` bundle aims to provide a spell checker service. However, to serve this service, this implementation requires a dictionary service. During this step, we will create an iPOPO component requiring a Dictionary service and providing the Spell Checker service.

```python
#!/usr/bin/python
# -- Content-Encoding: UTF-8 --
"""
The spell_checker component uses the dictionary services to check the spell of
a given text.
"""

# iPOPO decorators
from pelix.ipopo.decorators import ComponentFactory, Provides, \
    Validate, Invalidate, Requires, Instantiate, BindField, UnbindField

```

(continues on next page)

(continued from previous page)

```python
# Standard library
import re


# Name the component factory
@ComponentFactory("spell_checker_factory")
# Provide a Spell Checker service
@Provides("spell_checker_service")
# Consume all Spell Dictionary services available (aggregate them)
@Requires("_spell_dictionaries", "spell_dictionary_service", aggregate=True)
# Automatic instantiation
@Instantiate("spell_checker_instance")
class SpellChecker(object):
    """
    A component that uses spell dictionary services to check the spelling of
    given texts.
    """

    def __init__(self):
        """
        Define class members
        """
        # the spell dictionary service, injected list
        self._spell_dictionaries = []

        # the list of available dictionaries, constructed
        self.languages = {}

        # list of some punctuation marks could be found in the given passage,
        # internal
        self.punctuation_marks = None

    @BindField('_spell_dictionaries')
    def bind_dict(self, field, service, svc_ref):
        """
        Called by iPOPO when a spell dictionary service is bound to this
        component
        """
        # Extract the dictionary language from its properties
        language = svc_ref.get_property('language')

        # Store the service according to its language
        self.languages[language] = service

    @UnbindField('_spell_dictionaries')
    def unbind_dict(self, field, service, svc_ref):
        """
        Called by iPOPO when a dictionary service has gone away
        """
        # Extract the dictionary language from its properties
        language = svc_ref.get_property('language')

        # Remove it from the computed storage
        # The injected list of services is updated by iPOPO
        del self.languages[language]

    @Validate
```

(continues on next page)

(continued from previous page)

```python
    def validate(self, context):
        """
        This spell checker has been validated, i.e. at least one dictionary
        service has been bound.
        """
        # Set up internal members
        self.punctuation_marks = {',', ';', '.', '?', '!', ':', ' '}
        print('A spell checker has been started')

    @Invalidate
    def invalidate(self, context):
        """
        The component has been invalidated
        """
        self.punctuation_marks = None
        print('A spell checker has been stopped')

    def check(self, passage, language="EN"):
        """
        Checks the given passage for misspelled words.

        :param passage: the passage to spell check.
        :param language: language of the spell dictionary to use
        :return: An array of misspelled words or null if no words are misspelled
        :raise KeyError: No dictionary for this language
        """
        # list of words to be checked in the given passage
        # without the punctuation marks
        checked_list = re.split("([!,?.:; ])", passage)
        try:
            # Get the dictionary corresponding to the requested language
            dictionary = self.languages[language]
        except KeyError:
            # Not found
            raise KeyError('Unknown language: {0}'.format(language))

        # Do the job, calling the found service
        return [word for word in checked_list
                if word not in self.punctuation_marks
                and not dictionary.check_word(word)]
```

- The `@Requires` decorator specifies a service dependency. This required service is injected in a local variable in this bundle. Its `aggregate` attribute tells iPOPO to collect the list of services providing the required specification, instead of the first one.
- The `@BindField` decorator indicates that a new required service bounded to the platform.
- The `@UnbindField` decorator indicates that one of required service has gone away.

The spell client bundle

The `spell_client` bundle contains a very simple user interface allowing a user to interact with a spell checker service.

```python
#!/usr/bin/python
# -- Content-Encoding: UTF-8 --
```

(continues on next page)

(continued from previous page)

```python
"""
This bundle defines a component that consumes a spell checker.
It provides a shell command service, registering a "spell" command that can be
used in the shell of Pelix.

It uses a dictionary service to check for the proper spelling of a word by check
for its existence in the dictionary.
"""

# iPOPO decorators
from pelix.ipopo.decorators import ComponentFactory, Provides, \
    Validate, Invalidate, Requires, Instantiate

# Specification of a command service for the Pelix shell
from pelix.shell import SHELL_COMMAND_SPEC


# Name the component factory
@ComponentFactory("spell_client_factory")
# Consume a single Spell Checker service
@Requires("_spell_checker", "spell_checker_service")
# Provide a shell command service
@Provides(SHELL_COMMAND_SPEC)
# Automatic instantiation
@Instantiate("spell_client_instance")
class SpellClient(object):
    """
    A component that provides a shell command (spell.spell), using a
    Spell Checker service.
    """

    def __init__(self):
        """
        Defines class members
        """
        # the spell checker service
        self._spell_checker = None

    @Validate
    def validate(self, context):
        """
        Component validated, just print a trace to visualize the event.
        Between this call and the call to invalidate, the _spell_checker member
        will point to a valid spell checker service.
        """
        print('A client for spell checker has been started')

    @Invalidate
    def invalidate(self, context):
        """
        Component invalidated, just print a trace to visualize the event
        """
        print('A spell client has been stopped')

    def get_namespace(self):
        """
        Retrieves the name space of this shell command provider.
```

(continues on next page)

(continued from previous page)

```python
        Look at the shell tutorial for more information.
        """
        return "spell"

    def get_methods(self):
        """
        Retrieves the list of (command, method) tuples for all shell commands
        provided by this component.
        Look at the shell tutorial for more information.
        """
        return [("spell", self.spell)]

    def spell(self, io_handler):
        """
        Reads words from the standard input and checks for their existence
        from the selected dictionary.

        :param io_handler: A utility object given by the shell to interact with
                           the user.
        """
        # Request the language of the text to the user
        passage = None
        language = io_handler.prompt("Please enter your language, EN or FR: ")
        language = language.upper()

        while passage != 'quit':
            # Request the text to check
            passage = io_handler.prompt(
                "Please enter your paragraph, or 'quit' to exit:\n")

            if passage and passage != 'quit':
                # A text has been given: call the spell checker, which have been
                # injected by iPOPO.
                misspelled_words = self._spell_checker.check(passage, language)
                if not misspelled_words:
                    io_handler.write_line("All words are well spelled!")
                else:
                    io_handler.write_line(
                        "The misspelled words are: {0}", misspelled_words)
```

The component defined here implements and provides a shell command service, which will be consumed by the Pelix Shell Core Service. It registers a `spell` shell command.

Main script: Launching the framework

We have all the bundles required to start playing with the application. To run the example, we have to start Pelix, then all the required bundles.

```python
#!/usr/bin/python
# -- Content-Encoding: UTF-8 --
"""
Starts a Pelix framework and installs the Spell Checker bundles
"""

# Pelix framework module and utility methods
```

(continues on next page)

(continued from previous page)

```python
import pelix.framework
from pelix.utilities import use_service

# Standard library
import logging


def main():
    """
    Starts a Pelix framework and waits for it to stop
    """
    # Prepare the framework, with iPOPO and the shell console
    # Warning: we only use the first argument of this method, a list of bundles
    framework = pelix.framework.create_framework((
        # iPOPO
        "pelix.ipopo.core",
        # Shell core (engine)
        "pelix.shell.core",
        # Text console
        "pelix.shell.console"))

    # Start the framework, and the pre-installed bundles
    framework.start()

    # Get the bundle context of the framework, i.e. the link between the
    # framework starter and its content.
    context = framework.get_bundle_context()

    # Start the spell dictionary bundles, which provide the dictionary services
    context.install_bundle("spell_dictionary_EN").start()
    context.install_bundle("spell_dictionary_FR").start()

    # Start the spell checker bundle, which provides the spell checker service.
    context.install_bundle("spell_checker").start()

    # Sample usage of the spell checker service
    # 1. get its service reference, that describes the service itself
    ref_config = context.get_service_reference("spell_checker_service")

    # 2. the use_service method allows to grab a service and to use it inside a
    # with block. It automatically releases the service when exiting the block,
    # even if an exception was raised
    with use_service(context, ref_config) as svc_config:
        # Here, svc_config points to the spell checker service
        passage = "Welcome to our framwork iPOPO"
        print("1. Testing Spell Checker:", passage)
        misspelled_words = svc_config.check(passage)
        print(">  Misspelled_words are:", misspelled_words)

    # Start the spell client bundle, which provides a shell command
    context.install_bundle("spell_client").start()

    # Wait for the framework to stop
    framework.wait_for_stop()


# Classic entry point...
```

(continues on next page)

(continued from previous page)

```python
if __name__ == "__main__":
    logging.basicConfig(level=logging.DEBUG)
    main()
```

Running the application

Launch the `main_pelix_launcher.py` script. When the framework is running, type in the console: **spell** to enter your language choice and then your passage.

Here is a sample run, calling `python main_pelix_launcher.py`:

```
INFO:pelix.shell.core:Shell services registered
An English dictionary has been added
** Pelix Shell prompt **
A French dictionary has been added
A dictionary checker has been started
1. Testing Spell Checker: Welcome to our framwork iPOPO
>  Misspelled_words are: ['our', 'framwork']
A client for spell checker has been started

$ spell
Please enter your language, EN or FR: FR
Please enter your paragraph, or 'quit' to exit:
Bonjour le monde !
All words are well spelled !
Please enter your paragraph, or 'quit' to exit:
quit
$ spell
Please enter your language, EN or FR: EN
Please enter your paragraph, or 'quit' to exit:
Hello, world !
All words are well spelled !
Please enter your paragraph, or 'quit' to exit:
Bonjour le monde !
The misspelled words are: ['Bonjour', 'le', 'monde']
Please enter your paragraph, or 'quit' to exit:
quit
$ quit
Bye !
A spell client has been stopped
INFO:pelix.shell.core:Shell services unregistered
```

3.5 Reference Cards

This section contains some short introductions to the services provided by Pelix/iPOPO.

3.5.1 Bundles

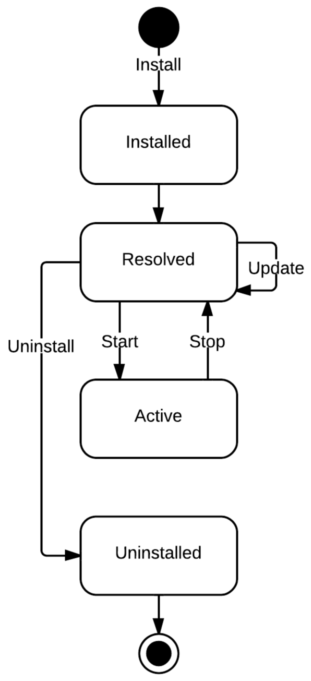
A bundle is a Python module installed using the Pelix *`Framework`* instance or a *`BundleContext`* object.

Each bundle is associated to an ID, an integer that is unique for a framework instance, and to a symbolic name, *i.e.* its module name. The framework itself is seen as the bundle which ID is always 0.

Because installing a bundle is in fact importing a module, **no code should be written to be executed at module-level** (except the definition of constants, the import of dependencies, . . .). Initialization must be done in the bundle activator (see below).

Life-cycle

Unlike a module, a bundle has a life-cycle and can be in one of the following states:



State	Description
INSTALLED	The Python module has been correctly imported, the bundle goes to the RESOLVED state
RESOLVED	The bundle has not been started yet or has been stopped
STARTING	The `start()` method of the bundle activator is being called (transition to ACTIVE or RESOLVED)
ACTIVE	The bundle activator has been called and didn't raise any error
STOPPING	The `stop()` method of the bundle activator is being called (transition to RESOLVED)
UNINSTALLED	The bundle has been removed from the framework (only visible by remaining references to the bundle)

The update process of a bundle is simple:

- if it was active, the bundle is stopped: other bundles are notified of this transition, and its services are unregistered
- the module is updated, using the imp.reload() method
  - if the update fails, the previous version of the module is kept, but the bundle is not restarted.
- if the update succeeds and the bundle was active, the bundle its restarted

Bundle Activator

A bundle activator is a class defining the `start()` and `stop()` methods, which are called by the framework according to the bundle life-cycle.

class `pelix.constants.`**`BundleActivator`**

This decorator must be applied to the class that will be notified of the life-cycle events concerning the bundle. A bundle can only have one activator, which must implement the following methods:

`start`(*context*)

This method is called when the bundle is in *STARTING* state. If this method doesn't raise an exception, the bundle goes immediately into the *ACTIVE* state. If an exception is raised, the bundle is stopped.

During the call of this method, the framework is locked. It is therefore necessary that this method returns as soon as possible: all time-consuming tasks should be executed in a new thread.

`stop`(*context*)

This method is called when the bundle is in *STOPPING* state. After this method returns or raises an exception, the bundle goes into the *RESOLVED* state.

All resources consumed by the bundle should be released before this method returns.

A class is defined as the bundle activator if it is decorated with `@BundleActivator`, as shown in the following snippet:

```python
import pelix.constants

@pelix.constants.BundleActivator
class Activator(object):
    """
    Bundle activator template
    """
    def start(self, context):
        """
        Bundle is starting
        """
        print("Start")

    def stop(self, context):
         """
         Bundle is stopping
         """
         print("Stop")
```

Note: The previous declaration of the activator, *i.e.* declaring an `activator` module member, is deprecated and its support will be removed in version 1.0.

Bundle Context

A context is associated to each bundle, and allows it to interact with the framework. It is unique for a bundle until it is removed from the framework. It must be used to register and to look up services, to request framework information, etc.

All the available methods are described in the *API chapter*. Here are the most used ones concerning the handling of bundles:

class `pelix.framework.`**`BundleContext`**(*framework*, *bundle*)

The bundle context is the link between a bundle and the framework. It is unique for a bundle and is created by the framework once the bundle is installed.

Parameters

- **`framework`** – Hosting framework
- **`bundle`** – The associated bundle

`add_bundle_listener`(*listener*)

Registers a bundle listener, which will be notified each time a bundle is installed, started, stopped or updated.

The listener must be a callable accepting a single parameter:

- **event** – The description of the event (a `BundleEvent` object).

Parameters **`listener`** – The bundle listener to register

Returns True if the listener has been registered, False if it already was

`get_bundle`(*bundle_id=None*)

Retrieves the *`Bundle`* object for the bundle matching the given ID (int). If no ID is given (None), the bundle associated to this context is returned.

Parameters **`bundle_id`** – A bundle ID (optional)

Returns The requested *`Bundle`* object

Raises **`BundleException`** – The given ID doesn't exist or is invalid

`get_bundles`()

Returns the list of all installed bundles

Returns A list of *`Bundle`* objects

`install_bundle`(*name*, *path=None*)

Installs the bundle (module) with the given name.

If a path is given, it is inserted in first place in the Python loading path (`sys.path`). All modules loaded alongside this bundle, *i.e.* by this bundle or its dependencies, will be looked after in this path in priority.

Note: Before Pelix 0.5.0, this method returned the ID of the installed bundle, instead of the Bundle object.

Warning: The behavior of the loading process is subject to changes, as it does not allow to safely run multiple frameworks in the same Python interpreter, as they might share global module values.

Parameters

- **`name`** – The name of the bundle to install
- **`path`** – Preferred path to load the module (optional)

Returns The *`Bundle`* object of the installed bundle

Raises **`BundleException`** – Error importing the module or one of its dependencies

`install_package`(*path*, *recursive=False*)

Installs all the modules found in the given package (directory). It is a utility method working like *`install_visiting()`*, with a visitor accepting every module found.

Parameters

- **`path`** – Path of the package (folder)
- **`recursive`** – If True, installs the modules found in sub-directories

Returns A 2-tuple, with the list of installed bundles (*`Bundle`*) and the list of the names of the modules which import failed.

Raises `ValueError` – The given path is invalid

`install_visiting`(*path*, *visitor*)
Looks for modules in the given path and installs those accepted by the given visitor.

The visitor must be a callable accepting 3 parameters:

- **fullname** – The full name of the module
- **is_package** – If True, the module is a package
- **module_path** – The path to the module file

Parameters

- **`path`** – Root search path (folder)
- **`visitor`** – The visiting callable

Returns A 2-tuple, with the list of installed bundles (*`Bundle`*) and the list of the names of the modules which import failed.

Raises `ValueError` – Invalid path or visitor

`remove_bundle_listener`(*listener*)
Unregisters the given bundle listener

Parameters `listener` – The bundle listener to remove

Returns True if the listener has been unregistered, False if it wasn't registered

3.5.2 Services

A service is an object that is registered to the framework service registry, associated with a set of specifications it implements and to properties.

The bundle that registers the service must keep the *`ServiceRegistration`* object returned by the framework. It allows to update the service properties and to unregister the service. This object **shall not** be accessible by other bundles/services. Finally, all services must be unregistered when their bundle is stopped.

A consumer can look for a service in the framework that matches a specification and a set of properties. The framework will return a *`ServiceReference`* object, which provides a read-only access to the description of its associated service: properties, registering bundle, bundles using it…

Properties

When registered and while it is available, the properties of a service can be set and updated by its provider.

Although, some properties are reserved for the framework; each service has at least the following properties:

Name	Type	Description
objectClass	list of str	List of the specifications implemented by this service
service.id	int	Identifier of the service. Unique in a framework instance

The framework also uses the following property to sort the result of a service look up:

Name	Type	Description
service.ranking	int	The rank/priority of the service. The lower the rank, the more priority

Service Factory

Warning: Service factories are a very recent feature of iPOPO and might be prone to bugs: please report any bug encounter on the project GitHub.

A service factory is a pseudo-service with a specific flag, which can create individual instances of service objects for different bundles. Sometimes a service needs to be differently configured depending on which bundle uses the service. For example, the log service needs to be able to print the logging bundle's id, otherwise the log would be hard to read.

A service factory is registered in exactly the same way as a normal service, using *register_service()*, with the `factory` argument set to True``. The only difference is an indirection step before the actual service object is handed out.

The client using the service need not, and should not, care if a service is generated by a factory or by a plain object.

A simple service factory example

```python
class ServiceInstance:
    def __init__(self, value):
        self.__value = value

    def cleanup(self):
        self.__value = None

    def get_value(self):
        return self.__value

class ServiceFactory:
    def __init__(self):
        # Bundle -> Instance
        self._instances = {}

    def get_service(self, bundle, registration):
        """
        Called each time a new bundle requires the service
        """
        instance = ServiceInstance(bundle.get_bundle_id())
        self._instances[bundle] = instance
        return instance

    def unget_service(self, bundle, registration):
        """
        Called when a bundle has released all its references
        to the service
        """
        # Release connections, ...
        self._instances.pop(bundle).cleanup()

bundle_context.register_service(
    "sample.factory", ServiceFactory(), {}, factory=True)
```

Note: The framework will cache generated service objects. Thus, at most one service can be generated per client bundle.

Prototype Service Factory

Warning: Prototype Service factories are a very recent feature of iPOPO and might be prone to bugs: please report any bug encounter on the project GitHub.

A prototype service factory is a pseudo-service with a specific flag, which can create multiple instances of service objects for different bundles.

Each time a bundle requires the service, the prototype service factory is called and can return a different instance. When called, the framework gives the factory the *Bundle* object requesting the service and the *ServiceRegistration* of the requested service. This allows a single factory to be registered for multiple services.

Note that there is no Prototype Service Factory implemented in the core Pelix/iPOPO Framework (unlike the *Log Service simple* service factory).

A Prototype Service Factory is registered in exactly the same way as a normal service, using *register_service()*, with the `prototype` argument set to `True`.

A simple prototype service factory example:

```python
class ServiceInstance:
    def __init__(self, value):
        self.__value = value

    def cleanup(self):
        self.__value = None

    def get_value(self):
        return self.__value

class PrototypeServiceFactory:
    def __init__(self):
        # Bundle -> [instances]
        self._instances = {}

    def get_service(self, bundle, registration):
        """
        Called each time ``get_service()`` is called
        """
        bnd_instances = self._instances.setdefault(bundle, [])
        instance = ServiceInstance(
            [bundle.get_bundle_id(), len(bnd_instances)])
        bnd_instances.append(instance)
        return instance

    def unget_service_instance(self, bundle, registration, service):
        """
        Called when a bundle releases an instance of the service
        """
        bnd_instances[bundle].remove(service)
```

(continues on next page)

(continued from previous page)

```
        service.cleanup()

    def unget_service(self, bundle, registration):
        """
        Called when a bundle has released all its references
        to the service
        """
        # Release global resources...

        # When this method is called, all instances will have been cleaned
        # up individually in ``unget_service_instance``
        if len(self._instances.pop(bundle)) != 0:
           raise ValueError("Should never happen")

bundle_context.register_service(
    "sample.proto", PrototypeServiceFactory(), {}, factory=True)
```

Note: A Prototype Service Factory is considered as a Service Factory, hence both *`is_factory()`* and *`is_prototype()`* will return `True` for this kind of service

API

The service provider has access to the *`ServiceRegistration`* object created by the framework when *`register_service()`* is called.

class `pelix.framework.`**`ServiceRegistration`**(*framework*, *reference*, *properties*, *update_callback*)

Represents a service registration object

Parameters

- **`framework`** – The host framework
- **`reference`** – A service reference
- **`properties`** – A reference to the ServiceReference properties dictionary object
- **`update_callback`** – Method to call when the sort key is modified

`get_reference`()

Returns the reference associated to this registration

Returns A ServiceReference object

`set_properties`(*properties*)

Updates the service properties

Parameters **`properties`** – The new properties

Raises **`TypeError`** – The argument is not a dictionary

`unregister`()

Unregisters the service

Consumers can access the service using its *`ServiceReference`* object, unique and constant for each service. This object can be retrieved using the *`BundleContext`* and its `get_service_reference*` methods. A consumer can check the properties of a service through this object, before consuming it.

class pelix.framework.**ServiceReference**(*bundle*, *properties*)
Represents a reference to a service

Parameters

- **bundle** – The bundle registering the service
- **properties** – The service properties

Raises **BundleException** – The properties doesn't contain mandatory entries

get_bundle()
Returns the bundle that registered this service

Returns the bundle that registered this service

get_properties()
Returns a copy of the service properties

Returns A copy of the service properties

get_property(*name*)
Retrieves the property value for the given name

Returns The property value, None if not found

get_property_keys()
Returns an array of the keys in the properties of the service

Returns An array of property keys.

get_using_bundles()
Returns the list of bundles that use this service

Returns A list of Bundle objects

is_factory()
Returns True if this reference points to a service factory

Returns True if the service provides from a factory

is_prototype()
Returns True if this reference points to a prototype service factory

Returns True if the service provides from a prototype factory

Finally, here are the methods of the *BundleContext* class that can be used to handle services:

class pelix.framework.**BundleContext**(*framework*, *bundle*)
The bundle context is the link between a bundle and the framework. It is unique for a bundle and is created by the framework once the bundle is installed.

Parameters

- **framework** – Hosting framework
- **bundle** – The associated bundle

add_service_listener(*listener*, *ldap_filter=None*, *specification=None*)
Registers a service listener

The service listener must have a method with the following prototype:

```
def service_changed(self, event):
    '''
    Called by Pelix when some service properties changes
```

(continues on next page)

(continued from previous page)

```
    event: A ServiceEvent object
    '''
    # ...
```

Parameters

- **`listener`** – The listener to register
- **`ldap_filter`** – Filter that must match the service properties (optional, None to accept all services)
- **`specification`** – The specification that must provide the service (optional, None to accept all services)

Returns True if the listener has been successfully registered

`get_all_service_references`(*clazz*, *ldap_filter=None*)
Returns an array of ServiceReference objects. The returned array of ServiceReference objects contains services that were registered under the specified class and match the specified filter expression.

Parameters

- **`clazz`** – Class implemented by the service
- **`ldap_filter`** – Service filter

Returns The sorted list of all matching service references, or None

`get_service`(*reference*)
Returns the service described with the given reference

Parameters **`reference`** – A ServiceReference object

Returns The service object itself

`get_service_reference`(*clazz*, *ldap_filter=None*)
Returns a ServiceReference object for a service that implements and was registered under the specified class

Parameters

- **`clazz`** – The class name with which the service was registered.
- **`ldap_filter`** – A filter on service properties

Returns A service reference, None if not found

`get_service_references`(*clazz*, *ldap_filter=None*)
Returns the service references for services that were registered under the specified class by this bundle and matching the given filter

Parameters

- **`clazz`** – The class name with which the service was registered.
- **`ldap_filter`** – A filter on service properties

Returns The list of references to the services registered by the calling bundle and matching the filters.

`register_service`(*clazz*, *service*, *properties*, *send_event=True*, *factory=False*, *prototype=False*)
Registers a service

Parameters

- **`clazz`** – Class or Classes (list) implemented by this service
- **`service`** – The service instance
- **`properties`** – The services properties (dictionary)
- **`send_event`** – If not, doesn't trigger a service registered event
- **`factory`** – If True, the given service is a service factory
- **`prototype`** – If True, the given service is a prototype service factory (the factory argument is considered True)

Returns A ServiceRegistration object

Raises **`BundleException`** – An error occurred while registering the service

`remove_service_listener`(*listener*)
Unregisters a service listener

Parameters **`listener`** – The service listener

Returns True if the listener has been unregistered

`unget_service`(*reference*)
Disables a reference to the service

Returns True if the bundle was using this reference, else False

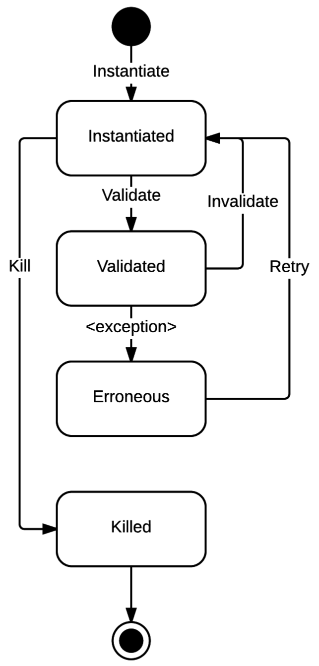
3.5.3 iPOPO Components

A component is an object with a life-cycle, requiring services and providing ones, and associated to properties. The code of a component is reduced to its functional purpose: its life-cycle, dependencies, etc. are handled by iPOPO. In iPOPO, a component is an instance of component factory, *i.e.* a Python class manipulated with the iPOPO decorators.

Note: Due to the use of Python properties, all component factories must be new-style classes. It is the case of all Python 3 classes, but Python 2.x classes must explicitly inherit from the `object` class.

Life-cycle

The component life cycle is handled by an instance manager created by the iPOPO service. This instance manager will inject control methods, run-time dependencies, and will register the component services. All changes will be notified to the component using the callback methods it decorated.



State	Description
INSTANTI-ATED	The component has been instantiated. Its constructor has been called and the control methods have been injected
VALI-DATED	All required dependencies have been injected. All services provided by the component will be registered right after this method returned
KILLED	The component has been invalidated and won't be usable again
ERRO-NEOUS	The component raised an error during its validation. It is not destroyed and a validation can be retried manually

API

iPOPO components are handled through the iPOPO core service, which can itself be accessed through the Pelix API or the utility context manager `use_ipopo()`. The core service provides the `pelix.ipopo.core` specification.

`pelix.ipopo.constants.`**`use_ipopo`**`(`*bundle_context*`)`

Utility context to use the iPOPO service safely in a "with" block. It looks after the the iPOPO service and releases its reference when exiting the context.

Parameters **`bundle_context`** – The calling bundle context

Returns The iPOPO service

Raises **`BundleException`** – Service not found

The following snippet shows how to use this method:

```python
from pelix.ipopo.constants import use_ipopo

# ... considering "context" being a BundleContext object
with use_ipopo(context) as ipopo:
    # use the iPOPO core service with the "ipopo" variable
```

(continues on next page)

(continued from previous page)

```
    ipopo.instantiate("my.factory", "my.component",
                      {"some.property": [1, 2, 3], "answer": 42})

# ... out of the "with" context, the iPOPO service has been released
# and shouldn't be used
```

Here are the most commonly used methods from the iPOPO core service to handle components and factories:

class `pelix.ipopo.core.`**`_IPopoService`**(*bundle_context*)

The iPOPO registry and service

Parameters **`bundle_context`** – The iPOPO bundle context

`add_listener`(*listener*)

Register an iPOPO event listener.

The event listener must have a method with the following prototype:

```
def handle_ipopo_event(self, event):
    '''
    event: A IPopoEvent object
    '''
    # ...
```

Parameters **`listener`** – The listener to register

Returns True if the listener has been added to the registry

`get_factories`()

Retrieves the names of the registered factories

Returns A list of factories. Can be empty.

`get_factory_details`(*name*)

Retrieves a dictionary with details about the given factory

- name: The factory name
- bundle: The Bundle object of the bundle providing the factory
- properties: Copy of the components properties defined by the factory
- requirements: List of the requirements defined by the factory
  - id: Requirement ID (field where it is injected)
  - specification: Specification of the required service
  - aggregate: If True, multiple services will be injected
  - optional: If True, the requirement is optional
- services: List of the specifications of the services provided by components of this factory
- handlers: Dictionary of the non-built-in handlers required by this factory. The dictionary keys are handler IDs, and it contains a tuple with:
  - A copy of the configuration of the handler (0)
  - A flag indicating if the handler is present or not

Parameters **`name`** – The name of a factory

Returns A dictionary describing the factory

Raises **`ValueError`** – Invalid factory

`get_instance_details`(*name*)

Retrieves a snapshot of the given component instance. The result dictionary has the following keys:

- name: The component name
- factory: The name of the component factory
- bundle_id: The ID of the bundle providing the component factory
- state: The current component state
- services: A {Service ID → Service reference} dictionary, with all services provided by the component
- dependencies: A dictionary associating field names with the following dictionary:
  - handler: The name of the type of the dependency handler
  - filter (optional): The requirement LDAP filter
  - optional: A flag indicating whether the requirement is optional or not
  - aggregate: A flag indicating whether the requirement is a set of services or not
  - binding: A list of the ServiceReference the component is bound to
- properties: A dictionary key → value, with all properties of the component. The value is converted to its string representation, to avoid unexpected behaviors.

Parameters **`name`** – The name of a component instance

Returns A dictionary of details

Raises **`ValueError`** – Invalid component name

`get_instances`()

Retrieves the list of the currently registered component instances

Returns A list of (name, factory name, state) tuples.

`instantiate`(*factory_name*, *name*, *properties=None*)

Instantiates a component from the given factory, with the given name

Parameters

- **`factory_name`** – Name of the component factory
- **`name`** – Name of the instance to be started
- **`properties`** – Initial properties of the component instance

Returns The component instance

Raises

- **`TypeError`** – The given factory is unknown
- **`ValueError`** – The given name or factory name is invalid, or an instance with the given name already exists
- **`Exception`** – Something wrong occurred in the factory

`kill`(*name*)

Kills the given component

Parameters **name** – Name of the component to kill

Raises **ValueError** – Invalid component name

remove_listener(*listener*)
Unregister an iPOPO event listener.

Parameters **listener** – The listener to register

Returns True if the listener has been removed from the registry

retry_erroneous(*name*, *properties_update=None*)
Removes the ERRONEOUS state of the given component, and retries a validation

Parameters

- **name** – Name of the component to retry
- **properties_update** – A dictionary to update the initial properties of the component

Returns The new state of the component

Raises **ValueError** – Invalid component name

3.5.4 iPOPO Decorators

Component definition

Those decorators describe the component. They must decorate the factory class itself.

Factory definition

The factory definition decorator must be unique per class and must always be the last one executed, *i.e.* the top one in the source code.

class `pelix.ipopo.decorators.`**ComponentFactory**(*name=None*, *excluded=None*)
Manipulates the component class according to a `FactoryContext` object filled by other decorators.

This **must** be the last executed decorator, *i.e.* the one on top of others in the source code.

It accepts the following arguments:

Parameters

- **name** – the name of the component factory
- **excluded** – the list of the IDs of the handlers which configuration must **not** be inherited from a parent component class

If no factory name is given, it will be generated as `ClassNameFactory`, *e.g.* a `Foo` class will have the factory name `FooFactory`.

The `__init__()` method of a component factory must not require any parameter.

```
@ComponentFactory()
class Foo(object):
    def __init__(self):
        pass

@ComponentFactory('my-factory')
```

(continues on next page)

(continued from previous page)

```
class Bar(object):
   pass
```

Parameters

- **name** – Name of the component factory
- **excluded** – List of IDs of handlers which configuration must not be inherited from the parent class

class pelix.ipopo.decorators.**SingletonFactory**(*name=None*, *excluded=None*)

This decorator is a specialization of the *ComponentFactory*: it accepts the same arguments and follows the same rule, but it allows only one instance of component from this factory at a time.

If the factory is instantiated while another already exist, a `ValueError` will be raised.

```
@SingletonFactory()
class Foo(object):
    def __init__(self):
        pass

@SingletonFactory('my-factory')
class Bar(object):
   pass
```

Parameters

- **name** – Name of the component factory
- **excluded** – List of IDs of handlers which configuration must not be inherited from the parent class

Component properties

class pelix.ipopo.decorators.**Property**(*field*, *name=None*, *value=None*)

The `@Property` decorator defines a component property. A property can be used to configure the component at validation time and to expose the state of a component. Note that component properties are exposed in the properties of the services it provides.

This decorator accepts the following parameters:

Parameters

- **field** – The property field in the class (can't be None nor empty)
- **name** – The property name (if None, this will be the field name)
- **value** – The property value (None by default)

Handler ID `pelix.ipopo.constants.HANDLER_PROPERTY`

If no initial value is given, the value stored in the field in the `__init__()` method will be used.

Warning: In Python 2, it is required that the component class inherits `object` for properties to work.

```
@ComponentFactory()
@Property('_answer', 'some.answer', 42)
class Foo(object):
    pass
```

Parameters

- **field** – The property field in the class (can't be None nor empty)
- **name** – The property name (if None, this will be the field name)
- **value** – The property value (None by default)

Raises

- **TypeError** – Invalid argument type
- **ValueError** – If the name or the name is None or empty

class pelix.ipopo.decorators.**HiddenProperty**(*field*, *name=None*, *value=None*)

The `@HiddenProperty` decorator defines a component property which won't be visible in the properties of the services it provides. This kind of property is also not accessible using iPOPO reflection methods.

This decorator accepts the same parameters and follows the same rules as *Property*.

```
@ComponentFactory()
@HiddenProperty('_password', 'some.password', "secret")
class Foo(object):
    pass
```

Parameters

- **field** – The property field in the class (can't be None nor empty)
- **name** – The property name (if None, this will be the field name)
- **value** – The property value (None by default)

Raises

- **TypeError** – Invalid argument type
- **ValueError** – If the name or the name is None or empty

Special properties

Note that some properties have a special meaning for iPOPO and Pelix.

Name	Type	Description
`instance.name`	`str`	The name of the iPOPO component instance
`service.id`	`int`	The registration number of a service
`service.ranking`	`int`	The rank (priority) of the services provided by this component

```
@ComponentFactory()
@Property('_name', 'instance.name')     # Special property
@Property('_value', 'my.value')         # Some property
@Property('_answer', 'the.answer', 42)  # Some property, with a default value
```

(continues on next page)

(continued from previous page)

```python
class Foo(object):
   def __init__(self):
       self._name = None     # This will overwritten by iPOPO
       self._value = 12      # 12 will be used if this property is not configured
       self._answer = None   # 42 will be used by default
```

Provided Services

class pelix.ipopo.decorators.**Provides**(*specifications*, *controller=None*, *factory=False*)

The `@Provides` decorator defines a service to be exposed by component instances. This service will be registered (visible) in the Pelix service registry while the component is valid and the service controller is set to `True`.

This decorator accepts the following parameters:

Parameters

- **specifications** – A list of provided specification(s), or the single provided specification (can't be empty)
- **controller** – The name of the service controller class field (optional)

Handler ID `pelix.ipopo.constants.HANDLER_PROVIDES`

All the properties of the component defined with the *Property* decorator will be visible in the service properties.

The controller is a Python *property* that must contain a boolean. By default, the controller is set to `True`, *i.e.* the service will be provided by the component when it is validated.

```python
@ComponentFactory()
# 'answer.value' will be a property of the service
@Property('_answer', 'answer.value')
@Provides('hello.world')
class Foo(object):
    pass

@ComponentFactory()
# This service will provide multiple specifications
@Provides(['hello.world', 'hello.world.extended'], '_svc_flag')
class Bar(object):
    # self._svc_flag = False ; to forbid the service to be provided
    # self._svc_flag = True  ; to provide the service
    pass
```

Sets up a provided service. A service controller can be defined to enable or disable the service.

Parameters

- **specifications** – A list of provided interface(s) name(s) (can't be empty)
- **controller** – Name of the service controller class field (optional)
- **factory** – If True, this service is a service factory

Raises ValueError – If the specifications are invalid

Requirements

class pelix.ipopo.decorators.**Requires**(*field*, *specification*, *aggregate=False*, *optional=False*, *spec_filter=None*, *immediate_rebind=False*)

The `@Requires` decorator defines the requirement of a service. It accepts the following parameters:

Parameters

- **field** – The field where to inject the requirement
- **specification** – The specification of the service to inject
- **aggregate** – If True, injects a list of services, else the first matching service
- **optional** – If True, this injection is optional: the component can be valid without it
- **spec_filter** – An LDAP query to filter injected services according to their properties
- **immediate_rebind** – If True, the component won't be invalidated then re-validated if a matching service is available when the injected dependency is unbound

Handler ID `pelix.ipopo.constants.HANDLER_REQUIRES`

The `field` and `specification` attributes are mandatory. By default, a requirement is neither aggregated nor optional (both are set to `False` and no specification filter is used.

Note: Since iPOPO 0.5.4, only one specification can be given.

```
@ComponentFactory()
@Requires('_hello', 'hello.world')
class Foo(object):
    pass

@ComponentFactory()
@Requires('_hello', 'hello.world', aggregate=True, optional=False,
          spec_filter='(language=fr)')
class Bar(object):
    pass
```

Parameters

- **field** – The injected field
- **specification** – The injected service specification
- **aggregate** – If True, injects a list
- **optional** – If True, this injection is optional
- **spec_filter** – An LDAP query to filter injected services upon their properties
- **immediate_rebind** – If True, the component won't be invalidated then re-validated if a matching service is available when the injected dependency is unbound

Raises

- **TypeError** – A parameter has an invalid type
- **ValueError** – An error occurred while parsing the filter or an argument is incorrect

class `pelix.ipopo.decorators.`**`Temporal`**(*field, specification, optional=False, spec_filter=None, timeout=10*)

The `@Temporal` decorator defines a single immediate rebind requirement with a grace time when the injected service disappears.

This decorator acts like :class:Requires except it doesn't support `immediate_rebind` (set to `True`) nor `aggregate`. It also adds the following argument:

Parameters **`timeout`** – Temporal timeout, in seconds (must be greater than 0)

Handler ID `pelix.ipopo.constants.HANDLER_TEMPORAL`

When the injected service disappears, the component won't be invalidated before the given timeout. If a matching is found, it is injected in-place and the component instance continues its operations. If the service is used while no service is available, the call is put in hold and blocks until a new service is injected or until the timeout is reached. In the latter case, a `TemporalException` is raised.

```
@ComponentFactory()
@Temporal('_hello', 'hello.world', timeout=5)
class Bar(object):
    pass
```

Parameters

- **`field`** – The injected field
- **`specification`** – The injected service specification
- **`optional`** – If true, this injection is optional
- **`spec_filter`** – An LDAP query to filter injected services upon their properties
- **`timeout`** – Temporal timeout, in seconds (must be greater than 0)

Raises

- **`TypeError`** – A parameter has an invalid type
- **`ValueError`** – An error occurred while parsing the filter or an argument is incorrect

class `pelix.ipopo.decorators.`**`RequiresBest`**(*field, specification, optional=False, spec_filter=None, immediate_rebind=True*)

The `@RequiresBest` decorator acts like *`Requires`*, but it always injects the service with the best rank (`service.ranking` property).

Unlike most of the other requirement decorators, `@RequiresBest` doesn't support the injection of a list of services: only the best service can be injected.

Handler ID `pelix.ipopo.constants.HANDLER_REQUIRES_BEST`

```
@ComponentFactory()
@RequiresBest('_hello', 'hello.world')
class Foo(object):
    pass

@ComponentFactory()
@RequiresBest('_hello', 'hello.world', optional=True,
              spec_filter='(language=fr)')
class Bar(object):
    pass
```

Parameters

- **`field`** – The injected field
- **`specification`** – The injected service specification
- **`optional`** – If true, this injection is optional
- **`spec_filter`** – An LDAP query to filter injected services upon their properties
- **`immediate_rebind`** – If True, the component won't be invalidated then re-validated if a matching service is available when the injected dependency is unbound

Raises

- **`TypeError`** – A parameter has an invalid type
- **`ValueError`** – An error occurred while parsing the filter or an argument is incorrect

class `pelix.ipopo.decorators.`**`RequiresMap`**(*field*, *specification*, *key*, *allow_none=False*, *aggregate=False*, *optional=False*, *spec_filter=None*)

The `@RequiresMap` decorator defines a requirement that must be injected in a dictionary.

In addition to the arguments of :class:Requires, this decorator also accepts or redefines the following ones:

Parameters

- **`key`** – The name of the service property to use as a dictionary key
- **`allow_none`** – If True, also injects services with the property value set to None or missing
- **`aggregate`** – If true, injects a list of services with the same property value, else injects only one service per value

Handler ID `pelix.ipopo.constants.HANDLER_REQUIRES_MAP`

```python
@ComponentFactory()
@RequiresMap('_hello', 'hello.world', 'language')
class Bar(object):
    # self._hello['fr'].hello('le monde')
    pass
```

Parameters

- **`field`** – The injected field
- **`specification`** – The injected service specification
- **`key`** – Name of the service property to use as a dictionary key
- **`allow_none`** – If True, inject services with a None property value
- **`aggregate`** – If true, injects a list
- **`optional`** – If true, this injection is optional
- **`spec_filter`** – An LDAP query to filter injected services upon their properties

Raises

- **`TypeError`** – A parameter has an invalid type
- **`ValueError`** – An error occurred while parsing the filter or an argument is incorrect

class pelix.ipopo.decorators.**RequiresVarFilter**(*field*, *specification*, *aggregate=False*, *optional=False*, *spec_filter=None*, *immediate_rebind=False*)

The @RequiresVarFilter decorator acts like *Requires*, but its LDAP filter dynamically adapts to the properties of this component.

Handler ID pelix.ipopo.constants.HANDLER_REQUIRES_VARIABLE_FILTER

```
@ComponentFactory()
@Property('_lang', 'lang', 'fr')
@RequiresVarFilter('_hello', 'hello.world', optional=True,
                   spec_filter='(language={lang})')
class Bar(object):
    pass
```

Parameters

- **field** – The injected field
- **specification** – The injected service specification
- **aggregate** – If True, injects a list
- **optional** – If True, this injection is optional
- **spec_filter** – An LDAP query to filter injected services upon their properties
- **immediate_rebind** – If True, the component won't be invalidated then re-validated if a matching service is available when the injected dependency is unbound

Raises

- **TypeError** – A parameter has an invalid type
- **ValueError** – An error occurred while parsing the filter or an argument is incorrect

Instance definition

class pelix.ipopo.decorators.**Instantiate**(*name*, *properties=None*)

This decorator tells iPOPO to instantiate a component instance from this factory as soon as its bundle is in **ACTIVE** state.

It accepts the following arguments:

Parameters

- **name** – The name of the component instance (**mandatory**)
- **properties** – The initial properties of the instance

If no properties are given, the default value declared in @Property decorators will be used.

The properties are associated to the component instance but not added to it. This means that new (meta-) properties can be added to add information to the component (like the Remote Services export properties), but those won't be accessible directly by the component. Those extra properties will be visible in component's services properties and in the instance properties returned by the iPOPO get_instance_details() method, but no new field will be injected in the component instance.

```
@ComponentFactory()
@Property('_name', 'name', 'foo')
@Instantiate('component-1')
```

(continues on next page)

(continued from previous page)

```
@Instantiate('component-2', {'name': 'bar'})
class Foo(object):
    pass
```

Parameters

- **name** – Instance name
- **properties** – Instance properties

Life-cycle events

Those decorators store behavioral information on component methods. They must decorate methods in the component class.

Component state

pelix.ipopo.decorators.**Validate**(*method*)

The validation callback decorator is called when a component becomes valid, *i.e.* if all of its required dependencies has been injected.

The decorated method must accept the bundle's *BundleContext* as argument:

```
@Validate
def validation_method(self, bundle_context):
    '''
    bundle_context: The component's bundle context
    '''
    # ...
```

If the validation callback raises an exception, the component goes into **ERRONEOUS** state.

If the component provides a service, the validation method is called before the provided service is registered to the framework.

Parameters **method** – The validation method

Raises **TypeError** – The decorated element is not a valid function

pelix.ipopo.decorators.**Invalidate**(*method*)

The invalidation callback decorator is called when a component becomes invalid, *i.e.* if one of its required dependencies disappeared.

The decorated method must accept the bundle's *BundleContext* as argument:

```
@Invalidate
def invalidation_method(self, bundle_context):
    '''
    bundle_context: The component's bundle context
    '''
    # ...
```

Exceptions raised by an invalidation callback are ignored.

If the component provides a service, the invalidation method is called after the provided service has been unregistered to the framework.

Parameters **`method`** – The decorated method

Raises **`TypeError`** – The decorated element is not a function

Injections

`pelix.ipopo.decorators.`**`Bind`**(*method*)

The `@Bind` callback decorator is called when a component is bound to a dependency.

The decorated method must accept the injected service object and its *`ServiceReference`* as arguments:

```
@Bind
def bind_method(self, service, service_reference):
    '''
    service: The injected service instance.
    service_reference: The injected service ServiceReference
    '''
    # ...
```

If the service is a required one, the bind callback is called **before** the component is validated.

The service reference can be stored *if it is released on unbind*.

Exceptions raised by a bind callback are ignored.

Parameters **`method`** – The decorated method

Raises **`TypeError`** – The decorated element is not a valid function

`class` `pelix.ipopo.decorators.`**`BindField`**(*field*, *if_valid=False*)

The `@BindField` callback decorator is called when a component is bound to a dependency, injected in the given field.

This decorator accepts the following arguments:

Parameters

- **`field`** – The field associated to the binding
- **`if_valid`** – If True, call the decorated method only when the component is valid

The decorated method must accept the field where the service has been injected, the service object and its *`ServiceReference`* as arguments:

```
@BindField('_hello')
def bind_method(self, field, service, service_reference):
    '''
    field: Field wherein the dependency is injected
    service: The injected service instance.
    service_reference: The injected service ServiceReference
    '''
    # ...
```

If the service is a required one, the bind callback is called **before** the component is validated. The bind field callback is called **after** the global bind method.

The service reference can be stored *if it is released on unbind*.

Exceptions raised by a bind callback are ignored.

Parameters

- **`field`** – Field associated to the binding

- **if_valid** – Call the method only if the component is valid

pelix.ipopo.decorators.**Update**(*method*)

The @Update callback decorator is called when the properties of an injected service have been modified.

The decorated method must accept the injected service object and its *ServiceReference* and the previous properties as arguments:

```
@Update
def update_method(self, service, service_reference, old_properties):
    '''
    service: The injected service instance.
    service_reference: The injected service ServiceReference
    old_properties: The previous service properties
    '''
    # ...
```

Exceptions raised by an update callback are ignored.

Parameters **method** – The decorated method

Raises **TypeError** – The decorated element is not a valid function

class pelix.ipopo.decorators.**UpdateField**(*field*, *if_valid=False*)

The @UpdateField callback decorator is called when the properties of a service injected in the given field have been updated.

This decorator accepts the following arguments:

Parameters

- **field** – The field associated to the binding
- **if_valid** – If True, call the decorated method only when the component is valid

The decorated method must accept the field where the service has been injected, the service object, its *ServiceReference* and its previous properties as arguments:

```
@UpdateField('_hello')
def update_method(self, service, service_reference, old_properties):
    '''
    field: Field wherein the dependency is injected
    service: The injected service instance.
    service_reference: The injected service ServiceReference
    old_properties: The previous service properties
    '''
    # ...
```

Exceptions raised by an update callback are ignored.

Parameters

- **field** – Field associated to the binding
- **if_valid** – Call the method only if the component is valid

pelix.ipopo.decorators.**Unbind**(*method*)

The @Unbind callback decorator is called when a component dependency is unbound.

The decorated method must accept the injected service object and its *ServiceReference* as arguments:

```
@Unbind
def unbind_method(self, service, service_reference):
    '''
    service: The previously injected service instance.
    service_reference: Its ServiceReference
    '''
    # ...
```

If the service is a required one, the unbind callback is called **after** the component has been invalidated.

Exceptions raised by an unbind callback are ignored.

Parameters **`method`** – The decorated method

Raises **`TypeError`** – The decorated element is not a valid function

class `pelix.ipopo.decorators.`**`UnbindField`**(*field*, *if_valid=False*)

The `@UnbindField` callback decorator is called when an injected dependency is unbound.

This decorator accepts the following arguments:

Parameters

- **`field`** – The field associated to the binding
- **`if_valid`** – If True, call the decorated method only when the component is valid

The decorated method must accept the field where the service has been injected, the service object, its `ServiceReference` and its previous properties as arguments:

```
@UnbindField('_hello')
def unbind_method(self, field, service, service_reference):
    '''
    field: Field wherein the dependency was injected
    service: The injected service instance.
    service_reference: The injected service ServiceReference
    '''
    # ...
```

If the service is a required one, the unbind callback is called **after** the component has been invalidated. The unbind field callback is called **before** the global unbind method.

Exceptions raised by an unbind callback are ignored.

Parameters

- **`field`** – Field associated to the binding
- **`if_valid`** – Call the method only if the component is valid

Service state

`pelix.ipopo.decorators.`**`PostRegistration`**(*method*)

The service post-registration callback decorator is called after a service of the component has been registered to the framework.

The decorated method must accept the `ServiceReference` of the registered service as argument:

```
@PostRegistration
def callback_method(self, service_reference):
    '''
    service_reference: The ServiceReference of the provided service
    '''
    # ...
```

Parameters **`method`** – The decorated method

Raises **`TypeError`** – The decorated element is not a valid function

`pelix.ipopo.decorators.`**`PostUnregistration`**(*method*)

The service post-unregistration callback decorator is called after a service of the component has been unregistered from the framework.

The decorated method must accept the *`ServiceReference`* of the registered service as argument:

```
@PostUnregistration
def callback_method(self, service_reference):
    '''
    service_reference: The ServiceReference of the provided service
    '''
    # ...
```

Parameters **`method`** – The decorated method

Raises **`TypeError`** – The decorated element is not a valid function

3.5.5 Initial Configuration File

The `pelix.misc.init_handler` module provides the *`InitFileHandler`* class. It is able to load the configuration of an iPOPO framework, from one or multiple files.

This configuration allows to setup environment variables, additional Python paths, framework properties, a list of bundles to start with the framework and a list of components to instantiate.

File Format

Configuration files are in JSON format, with a root object which can contain the following entries:

- `properties`: a JSON object defining the initial properties of the framework. The object keys must be strings, but can be associated to any valid JSON value.
- `environment`: a JSON object defining new environment variables for the process running the framework. Both keys and values must be strings.
- `paths`: a JSON array containing paths to add to the Python lookup paths. The given paths will be prioritized, *i.e.* if a path was already defined in `sys.path`, it will be moved forward. The given paths can contains environment variables and the user path marker (~).

  Note that the current working directory (*cwd*) will always be the first element of `sys.path` when using an initial configuration handler.
- `bundles`: a JSON array containing the names of the bundles to install and start with the framework.
- `components`: a JSON array of JSON objects defining the components to instantiate. Each component description has the following entries:

  - `factory`: the name of the component factory
  - `name`: the name of the instance
  - `properties` (optional): a JSON object defining the initial properties of the component

Here is a sample initial configuration file:

```
{
  "properties": {
    "some.value": 42,
    "framework.uuid": "custom-uuid",
    "arrays": ['they', 'work', 'too', 123],
    "dicts": {"why": "not?"}
  },
  "environment": {
    "new_path": "/opt/foo",
    "LANG": "en_US.UTF-8"
  }
  "paths": [
    "/opt/bar",
    "$new_path/mylib.zip"
  ],
  "bundles": [
    "pelix.misc.log",
    "pelix.shell.log",
    "pelix.http.basic"
  ],
  "components": [
    {
      "factory": "pelix.http.service.basic.factory",
      "name": "httpd",
      "properties": {
         "pelix.http.address": "127.0.0.1"
      }
    }
  ]
}
```

Moreover, if the root object contains a `reset_<name>` entry, then the previously loaded configuration for the `<name>` entry are forgotten: the current configuration will replace the old one instead of updating it.

For example:

```
{
  "bundles": [
     "pelix.http.basic"
  ],
  "reset_bundles": true
}
```

When this file will be loaded, the list of bundles declared by previously loaded configuration files will be cleared and replaced by the one in this file.

File lookup

A *InitFileHandler* object updates its internal state with the content of the files it parses. As a result, multiple configuration files can be used to start framework with a common basic configuration.

When calling `load()` without argument, the handler will try to load all the files named `.pelix.conf` in the following folders and order:

- `/etc/default`
- `/etc`
- `/usr/local/etc`
- `~/.local/pelix`
- `~/.config`
- `~` (user directory)
- `.` (current working directory)

When giving a file name to `load()`, the handler will merge the configuration it contains with its current state.

Finally, after having updated a configuration, the `InitFileHandler` will remove duplicated in Python path and bundles configurations.

Support in Pelix shell

The framework doesn't starts a `InitFileHandler` on its own: it must be created and loaded before creating the framework.

Currently, only the Pelix Shell Console supports the initial configuration, using the following arguments:

- *no argument*: the *.pelix.conf* files are loaded as described in *File lookup*.
- `-e`, `--empty-conf`: no initial configuration file will be loaded
- `-c <filename>`, `--conf <filename>`: the default configuration files, then given one will be loaded.
- `-C <filename>`, `--exclusive-conf <filename>`: only the given configuration file will be loaded.

It is planned that the support for initial configuration files will be added to other shells in future iPOPO versions.

API

class `pelix.misc.init_handler.`**`InitFileHandler`**

Parses and handles the instructions of initial configuration files

`clear`()

Clears the current internal state (cleans up all loaded content)

`instantiate_components`(*context*)

Instantiate the defined components

Parameters **`context`** – A `BundleContext` object

Raises **`BundleException`** – Error starting a component

`load`(*filename=None*)

Loads the given file and adds its content to the current state. This method can be called multiple times to merge different files.

If no filename is given, this method loads all default files found. It returns False if no default configuration file has been found

Parameters **`filename`** – The file to load

Returns True if the file has been correctly parsed, False if no file was given and no default file exist

Raises **IOError** – Error loading file

normalize()

Normalizes environment variables and the Python path.

This method first updates the environment variables (`os.environ`). Then, it normalizes the Python path (`sys.path`) by resolving all references to the user directory and environment variables.

bundles

Returns The list of names of bundles to install and start

properties

Returns The initial framework properties

Sample API Usage

This sample starts a framework based on the default configuration files, plus a given one named *some_file.json*.

```
import pelix.framework as pelix
from pelix.misc.init_handler import InitFileHandler

# Read the initial configuration script
init = InitFileHandler()

# Load default configuration
init.load()

# Load the given configuration file
init.load("some_file.json")

# Normalize configuration (forge sys.path)
init.normalize()

# Use the utility method to create, run and delete the framework
framework = pelix.create_framework(init.bundles, init.properties)
framework.start()

# Instantiate configured components
init.instantiate_components(framework.get_bundle_context())

# Let the framework live
try:
    framework.wait_for_stop()
except KeyboardInterrupt:
    framework.stop()
```

3.5.6 Logging

The best way to log traces in iPOPO is to use the logging module from the Python Standad Library. Pelix/iPOPO relies on this module for its own logs, using a module level constant providing a logger with the name of the module, like this:

```
import logging
_logger = logging.getLogger(__name__)
```

That being said, Pelix/iPOPO provides a utility log service matching the OSGi *LogService* specification, which logs to and reads traces from the standard Python logging system.

The log service is provided by the `pelix.misc.log` bundle. It handles `LogEntry` object keeping track of the log timestamp, source bundle and message. It also registers as a handler to the Python logging system, which means it can also keep track of all traces logged with the `logging` module.

API

Once install and started, the `pelix.misc.log` bundle provides two services:

- `pelix.log`: The main log service, which allows to log entries;
- `pelix.log.reader`: The log reader service, which gives a read-only access to previous log entries. Those entries can be stored using either the log service or the Python logging system.

Log Service

The log service provides the following method:

class pelix.misc.log.**LogServiceInstance**(*reader*, *bundle*)

Instance of the log service given to a bundle by the factory

Parameters

- **reader** – The Log Reader service
- **bundle** – Bundle associated to this instance

log(*level*, *message*, *exc_info=None*, *reference=None*)

Logs a message, possibly with an exception

Parameters

- **level** – Severity of the message (Python logging level)
- **message** – Human readable message
- **exc_info** – The exception context (sys.exc_info()), if any
- **reference** – The ServiceReference associated to the log

Log Reader Service

The log reader provides the following methods:

class pelix.misc.log.**LogReaderService**(*context*, *max_entries*)

The LogReader service

Parameters

- **context** – The bundle context
- **max_entries** – Maximum stored entries

add_log_listener(*listener*)
Subscribes a listener to log events.

A log listener is an object providing with a `logged` method, with the following signature:

```python
def logged(self, log_entry):
    '''
    A log entry (LogEntry) has been added to the log service
    '''
    # ...
```

Parameters **listener** – A new listener

get_log()
Returns the logs events kept by the service

Returns A tuple of log entries

remove_log_listener(*listener*)
Unsubscribes a listener from log events.

Parameters **listener** – The listener to remove

The result of *get_log()* and the argument to listeners registered with *add_log_listener()* is a *LogEntry* object, giving read-only access to the following properties:

class pelix.misc.log.**LogEntry**(*level*, *message*, *exception*, *bundle*, *reference*)
Represents a log entry

Parameters

- **level** – The Python log level of the entry
- **message** – A human readable message
- **exception** – The exception associated to the entry
- **bundle** – The bundle that created the entry
- **reference** – The service reference associated to the entry

bundle
The bundle that created this entry

exception
The exception associated to this entry

level
The log level of this entry (Python constant)

message
The message associated to this entry

osgi_level
The log level of this entry (OSGi constant)

reference
The reference to the service associated to this entry

time
The timestamp of this entry

Note: `LogEntry` is a read-only bean which can't be un-marshalled by Pelix Remote Services transport providers. As a consequence, it is not possible to get the content of a remote log service as is.

Sample Usage

Using the shell is pretty straightforward, as it can be seen in the `pelix.shell.log` bundle.

```python
import logging

from pelix.ipopo.decorators import ComponentFactory, Requires, Instantiate, \
   Validate, Invalidate
from pelix.misc import LOG_SERVICE, LOG_READER_SERVICE

@ComponentFactory("log-sample-factory")
@Requires("_logger", LOG_SERVICE)
@Requires("_reader", LOG_READER_SERVICE)
@Instantiate("log-sample")
class SampleLog(object):
    """
    Provides shell commands to print the content of the log service
    """
    def __init__(self):
        self._logger = None
        self._reader = None

    @Validate
    def _validate(self, context):
        self._reader.add_log_listener(self)
        self._logger.log(logging.INFO, "Component validated")

    @Invalidate
    def _invalidate(self, context):
        self._logger.log(logging.WARNING, "Component invalidated")
        self._reader.remove_log_listener(self)

    def logged(self, entry):
        print("Got a log:", entry.message, "at level", entry.level)
```

The log service is provided by a service factory, therefore the components of a same bundle share the same service, and each bundle has a different instance of the logger. The log reader service is a singleton service.

Shell Commands

The `pelix.shell.log` bundle provides a set of commands in the `log` shell namespace, to interact with the log services:

Command	Description
log	Prints the last `N` entries with level higher than the given one (`WARNING` by default)
debug	Logs a message at `DEBUG` level
info	Logs a message at `INFO` level
warning	Logs a message at `WARNING` level
warn	An alias of the `warning` command
error	Logs a message at `ERROR` level

```
$ install pelix.misc.log
Bundle ID: 12
$ start $?
Starting bundle 12 (pelix.misc.log)...
$ install pelix.shell.log
Bundle ID: 13
$ start $?
Starting bundle 13 (pelix.shell.log)...
$ debug "Some debug log"
$ info "..INFO.."
$ warning !!WARN!!
$ error oops
$ log 3
WARNING :: 2017-03-10 12:06:29.131131 :: pelix.shell.log :: !!WARN!!
 ERROR  :: 2017-03-10 12:06:31.884023 :: pelix.shell.log :: oops
$ log info
 INFO   :: 2017-03-10 12:06:26.331350 :: pelix.shell.log :: ..INFO..
WARNING :: 2017-03-10 12:06:29.131131 :: pelix.shell.log :: !!WARN!!
 ERROR  :: 2017-03-10 12:06:31.884023 :: pelix.shell.log :: oops
$ log info 2
WARNING :: 2017-03-10 12:06:29.131131 :: pelix.shell.log :: !!WARN!!
 ERROR  :: 2017-03-10 12:06:31.884023 :: pelix.shell.log :: oops
$
```

3.5.7 HTTP Service

The HTTP service is a basic servlet container, dispatching HTTP requests to the handler registered for the given path. A servlet can be a simple class or a component, registered programmatically to the HTTP service, or a service registered in the Pelix framework and automatically registered by the HTTP service.

Note: Even if it borrows the concept of *servlets* from Java, the Pelix HTTP service doesn't follow the OSGi specification. The latter inherits a lot from the existing Java APIs, while this is an uncommon way to work in Python.

The basic implementation of the HTTP service is defined in `pelix.http.basic`. It is based on the HTTP server available in the standard Python library (see http.server). Future implementations might appear in the future Pelix implementations, based on more robust requests handlers.

Configuration properties

All implementations of the HTTP service must support the following property:

Property	Default	Description
pelix.http.address	0.0.0.0	The address the HTTP server is bound to
pelix.http.port	8080	The port the HTTP server is bound to

Instantiation

The HTTP bundle defines a component factory which name is implementation-dependent. The HTTP service factory provided by Pelix/iPOPO is `pelix.http.service.basic.factory`.

Here is a snippet that starts a HTTP server component, named `http-server`, which only accepts local clients on port 9000:

```
from pelix.framework import FrameworkFactory
from pelix.ipopo.constants import use_ipopo

# Start the framework
framework = FrameworkFactory.get_framework()
framework.start()
context = framework.get_bundle_context()

# Install & start iPOPO
context.install_bundle('pelix.ipopo.core').start()

# Install & start the basic HTTP service
context.install_bundle('pelix.http.basic').start()

# Instantiate a HTTP service component
with use_ipopo(context) as ipopo:
   ipopo.instantiate(
       'pelix.http.service.basic.factory', 'http-server',
       {'pelix.http.address': 'localhost',
        'pelix.http.port': 9000})
```

This code starts an HTTP server which will be listening on port 9000 and the HTTP service will be ready to handle requests. As no servlet service has been registered, the server will only return 404 errors.

API

HTTP service

The HTTP service provides the following interface:

class `pelix.http.basic.`**`HttpService`**

Basic HTTP service component

`get_access``()`

Retrieves the (address, port) tuple to access the server

static **`get_hostname`**`()`

Retrieves the server host name

Returns The server host name

`get_registered_paths``()`

Returns the paths registered by servlets

Returns The paths registered by servlets (sorted list)

`get_servlet``(`*path*`)`

Retrieves the servlet matching the given path and its parameters. Returns None if no servlet matches the given path.

Parameters **`path`** – A request URI

Returns A tuple (servlet, parameters, prefix) or None

`is_https``()`

Returns True if this is an HTTPS server

Returns True if this server uses SSL

register_servlet (*path*, *servlet*, *parameters=None*)
Registers a servlet

Parameters

- **path** – Path handled by this servlet
- **servlet** – The servlet instance
- **parameters** – The parameters associated to this path

Returns True if the servlet has been registered, False if it refused the binding.

Raises **ValueError** – Invalid path or handler

unregister (*path*, *servlet=None*)
Unregisters the servlet for the given path

Parameters

- **path** – The path to a servlet
- **servlet** – If given, unregisters all the paths handled by this servlet

Returns True if at least one path as been unregistered, else False

The service also provides two utility methods to ease the display of error pages:

class `pelix.http.basic.`**HttpService**
Basic HTTP service component

make_exception_page (*path*, *stack*)
Prepares a page printing an exception stack trace in a 500 error

Parameters

- **path** – Request path
- **stack** – Exception stack trace

Returns A HTML page

make_not_found_page (*path*)
Prepares a "page not found" page for a 404 error

Parameters **path** – Request path

Returns A HTML page

Servlet service

To use the whiteboard pattern, a servlet can be registered as a service providing the `pelix.http.servlet` specification. It must also have a valid `pelix.http.path` property, or it will be ignored.

The binding methods described below have a `parameters` argument, which represents a set of properties of the server, given as a dictionary. Some parameters can also be given when using the *`register_servlet()`* method, with the `parameters` argument.

In any case, the following entries must be set by all implementations of the HTTP service and can't be overridden when register a servlet. Note that their content and liability is implementation-dependent:

- `http.address`: the binding address (*str*) of the HTTP server;
- `http.port`: the real listening port (*int*) of the HTTP server;
- `http.https`: a boolean flag indicating if the server is listening to HTTP (False) or HTTPS (True) requests;

- `http.name`: the name (*str*) of the server. If the server is an iPOPO component, it should be the instance name;
- `http.extra`: an implementation dependent set of properties.

A servlet for the Pelix HTTP service has the following methods:

class HttpServlet

These are the methods that the HTTP service can call in a servlet. Note that it is not necessary to implement them all: the service has a default behaviour for missing methods.

accept_binding (*path*, *parameters*)

This method is called before trying to bind the servlet. If it returns False, the servlet won't be bound to the server. This allows a servlet service to be bound to a specific server.

If this method doesn't exist or returns None or anything else but False, the calling HTTP service will consider that the servlet accepts to be bound to it.

Parameters

- **path** (*str*) – The path of the servlet in the server
- **parameters** (*dict*) – The parameters of the server

bound_to (*path*, *parameters*)

This method is called when the servlet is bound to a path. If it returns False or raises an Exception, the registration is aborted.

Parameters

- **path** (*str*) – The path of the servlet in the server
- **parameters** (*dict*) – The parameters of the server

unbound_from (*path*, *parameters*)

This method is called when the servlet is bound to a path. The parameters are the ones given in *accept_binding()* and *bound_to()*.

Parameters

- **path** (*str*) – The path of the servlet in the server
- **parameters** (*dict*) – The parameters of the server

do_GET (*request*, *response*)

Each request is handled by the method call `do_XXX` where `XXX` is the name of an HTTP method (`do_GET`, `do_POST`, `do_PUT`, `do_HEAD`, . . .).

If it raises an exception, the server automatically sends an HTTP 500 error page. In nominal behaviour, the method must use the `response` argument to send a reply to the client.

Parameters

- **request** – A *AbstractHTTPServletRequest* representation of the request
- **response** – The *AbstractHTTPServletResponse* object to use to reply to the client

HTTP request

Each request method has a request helper argument, which implements the *AbstractHTTPServletRequest* abstract class.

class `pelix.http.`**AbstractHTTPServletRequest**

Abstract HTTP Servlet request helper

get_client_address()
Returns the address of the client

Returns A (host, port) tuple

get_command()
Returns the HTTP verb (GET, POST, . . .) used for the request

get_header(*name*, *default=None*)
Returns the value of a header

Parameters

- **name** – Header name
- **default** – Default value if the header doesn't exist

Returns The header value or the default one

get_headers()
Returns a copy all headers, with a dictionary interface

Returns A dictionary-like object

get_path()
Returns the request full path

Returns A request full path (string)

get_prefix_path()
Returns the path to the servlet root

Returns A request path (string)

get_rfile()
Returns the request input as a file stream

Returns A file-like input stream

get_sub_path()
Returns the servlet-relative path, i.e. after the prefix

Returns A request path (string)

read_data()
Reads all the data in the input stream

Returns The read data

HTTP response

Each request method also has a response helper argument, which implements the *AbstractHTTPServletResponse* abstract class.

class pelix.http.**AbstractHTTPServletResponse**
HTTP Servlet response helper

end_headers()
Ends the headers part

get_wfile()
Retrieves the output as a file stream. `end_headers()` should have been called before, except if you want to write your own headers.

Returns A file-like output stream

is_header_set(*name*)
Checks if the given header has already been set

Parameters **name** – Header name

Returns True if it has already been set

send_content(*http_code*, *content*, *mime_type='text/html'*, *http_message=None*, *content_length=-1*)
Utility method to send the given content as an answer. You can still use get_wfile or write afterwards, if you forced the content length.

If content_length is negative (default), it will be computed as the length of the content; if it is positive, the given value will be used; if it is None, the content-length header won't be sent.

Parameters

- **http_code** – HTTP result code
- **content** – Data to be sent (must be a string)
- **mime_type** – Content MIME type (content-type)
- **http_message** – HTTP code description
- **content_length** – Forced content length

set_header(*name*, *value*)
Sets the value of a header. This method should not be called after `end_headers()`.

Parameters

- **name** – Header name
- **value** – Header value

set_response(*code*, *message=None*)
Sets the response line. This method should be the first called when sending an answer.

Parameters

- **code** – HTTP result code
- **message** – Associated message

write(*data*)
Writes the given data. `end_headers()` should have been called before, except if you want to write your own headers.

Parameters **data** – Data to be written

Write a servlet

This snippet shows how to write a component providing the servlet service:

```
from pelix.ipopo.decorators import ComponentFactory, Property, Provides, \
    Requires, Validate, Invalidate, Unbind, Bind, Instantiate

@ComponentFactory(name='simple-servlet-factory')
@Instantiate('simple-servlet')
@Provides(specifications='pelix.http.servlet')
@Property('_path', 'pelix.http.path', "/servlet")
class SimpleServletFactory(object):
```

(continues on next page)

(continued from previous page)

```
    """
    Simple servlet factory
    """
    def __init__(self):
        self._path = None

    def bound_to(self, path, params):
        """
        Servlet bound to a path
        """
        print('Bound to ' + path)
        return True

    def unbound_from(self, path, params):
        """
        Servlet unbound from a path
        """
        print('Unbound from ' + path)
        return None

    def do_GET(self, request, response):
        """
        Handle a GET
        """
        content = """<html>
<head>
<title>Test SimpleServlet</title>
</head>
<body>
<ul>
<li>Client address: {clt_addr[0]}</li>
<li>Client port: {clt_addr[1]}</li>
<li>Host: {host}</li>
<li>Keys: {keys}</li>
</ul>
</body>
</html>""".format(clt_addr=request.get_client_address(),
                  host=request.get_header('host', 0),
                  keys=request.get_headers().keys())

        response.send_content(200, content)
```

To test this snippet, install and start this bundle and the HTTP service bundle in a framework, then open a browser to the servlet URL. If you used the HTTP service instantiation sample, this URL should be http://localhost:9000/servlet.

3.5.8 HTTP Routing utilities

The `pelix.http.routing` module provides a utility class and a set of decorators to ease the development of REST-like servlets.

Decorators

Important: A servlet which uses the utility decorators **must** inherit from the `pelix.http.routing.RestDispatcher` class.

The `pelix.http.routing.RestDispatcher` class handles all `do_*` methods and calls the corresponding decorated methods in the child class.

The child class can declare as many methods as necessary, with any name (public, protected or private) and decorate them with the following decorators. Note that a method can be decorated multiple times.

class `pelix.http.routing.`**`Http`**(*route*, *methods=None*)

Decorator indicating which route a method handles

Parameters

- **`route`** – Path handled by the method (beginning with a '/')
- **`methods`** – List of HTTP methods allowed (GET, POST, . . .)

class `pelix.http.routing.`**`HttpGet`**(*route*)

Bases: *`pelix.http.routing.Http`*

Decorates a method handling GET requests

Parameters **`route`** – Path handled by the method (beginning with a '/')

class `pelix.http.routing.`**`HttpPost`**(*route*)

Bases: *`pelix.http.routing.Http`*

Decorates a method handling POST requests

Parameters **`route`** – Path handled by the method (beginning with a '/')

class `pelix.http.routing.`**`HttpPut`**(*route*)

Bases: *`pelix.http.routing.Http`*

Decorates a method handling PUT requests

Parameters **`route`** – Path handled by the method (beginning with a '/')

class `pelix.http.routing.`**`HttpHead`**(*route*)

Bases: *`pelix.http.routing.Http`*

Decorates a method handling HEAD requests

Parameters **`route`** – Path handled by the method (beginning with a '/')

class `pelix.http.routing.`**`HttpDelete`**(*route*)

Bases: *`pelix.http.routing.Http`*

Decorates a method handling DELETE requests

Parameters **`route`** – Path handled by the method (beginning with a '/')

The decorated methods muse have the following signature:

`decorated_method`(*request*, *response*, ***kwargs*)

Called by the dispatcher to handle a request.

The keyword arguments must have the same name as the ones given in the URL pattern in the decorators.

Parameters

- **`request`** – An *`AbstractHTTPServletRequest`* object
- **`response`** – An *`AbstractHTTPServletResponse`* object

Supported types

Each argument in the URL can be automatically converted to the requested type. If the conversion fails, an error 500 is automatically sent back to the client.

Type	Description
string	Simple string used as is. The string can't contain a slash (`/`)
int	The argument is converted to an integer. The input must be of base 10. Floats are rejected.
float	The argument is converted to a float. The input must be of base 10.
path	A string representing a path, containing slashes.
uuid	The argument is converted to a `uuid.UUID` class.

Multiple arguments can be given at a time, but can only be of one type.

Sample

```python
from pelix.ipopo.decorators import ComponentFactory, Provides, Property, \
   Instantiate
from pelix.http import HTTP_SERVLET, HTTP_SERVLET_PATH
from pelix.http.routing import RestDispatcher, HttpGet, HttpPost

@ComponentFactory()
@Provides(HTTP_SERVLET)
@Property('_path', HTTP_SERVLET_PATH, '/api/v0')
@Instantiate("some-servlet")
class SomeServlet(RestDispatcher):
   @HttpGet("/list")
   def list_elements(self, request, response):
      response.send_content(200, "<p>The list</p>")

   @HttpPost("/form/<form_id:uuid>")
   def handle_form(self, request, response, form_id):
      reponse.send_content(200, "<p>Handled {}</p>".format(form_id))

   @HttpPut("/upload/<some_id:int>/<filename:path>")
   @HttpPut("/upload/<filename:path>")
   def handle_upload(
   self, request, response,
                  some_id=None, filename=None):
      reponse.send_content(200, "<p>Handled {} : {}</p>" \
         .format(some_id, filename))
```

3.5.9 Remote Services

Pelix/iPOPO provides support for *remote services*, *i.e.* consuming services provided from another framework instance. This provider can run on the same machine as the consumer, or on another one.

Concepts

Pelix/iPOPO remote services implementation is a based on a set of services. This architecture eases the development of new providers and allows to plug in or update protocols providers at run time.

In this section, we will shortly describe the basic concepts of Pelix Remote Services, *i.e.*:

- the concept of import and export endpoints
- the core services required to activate remote services
- the discovery providers
- the transport providers

The big picture of the Pelix Remote Services can be seen as:

Note that Pelix Remote Services implementation has been inspired from the OSGi Remote Services specification, and tries to reuse most of its constants, to ease compatibility.

Before that, it is necessary to see the big picture: how does Pelix Remote Services works.

How does it work?

The export and import of a service follows this sequence diagram, described below:

When a service declares it can be exported, the *export dispatcher* detects it (as it is a service listener) notifies all *transport providers* which matches the service properties. Each transport provider then tests if it can/must create an endpoint for it and, if so, returns an *export endpoint* description to the *exports dispatcher*. The endpoint implementation is transport-dependent: it can be a servlet (HTTP-based procotols), a serial-port listener, . . . As a result, there can be multiple *export endpoints* for a single service: (at least) one per transport provider. The description of each *export endpoint* is then stored in the *exports dispatcher*, one of the core services of Pelix Remote Services.

When an endpoint (or a set of endpoints) is stored in the *exports dispatcher*, the discovery providers are notified and send there protocol-specific events. They can target other Pelix frameworks, but also any other kind of frameworks (OSGi/Java, . . .) or of software (like a Node.js server with mDNS support). Those events indicate that new export endpoints are available: they can point to the description of this endpoint or contain its serialized form. Note that the description sent over the network must be an import-side description: it should contain all information required to connect and use the endpoint, stored in import properties so that the newly imported services don't get exported by mistake.

Another framework using the same discovery provider can capture this event and handle the new set of *import endpoints*. Those endpoints will be stored in the *imports registry*, the other core service of Pelix Remote Services. If multiple discovery providers find the same endpoints, don't worry, they will be filtered out according to their unique identifier (UUID).

The *imports registry* then notifies the *transport providers* to let them create a local proxy to the remote service and register it as a local service (with import properties). This remote service is now usable by local consumers.

Note: In the current implementation of Pelix Remote Services, the same remote service can be imported multiple times by the same consumer framework. This is due to the fact that the imported service is created by the transport providers and not by the centralized imports registry.

This behaviour is useful when you want to consume a service from a specific provider, or if you can sort transport providers by efficiency. This has to been taken into account in some cases, like when consuming multiple services of the same specification while multiple transport providers are active.

This behaviour is subject to debate but is also used in some projects. It could be modified if enough problems are reported either on the mailing list or in GitHub issues.

Finally, Pelix Remote Services also supports the update of service properties, which can be handled as a minimalist event by the discovery providers, *e.g.* containing only the endpoint UID and the new properties. The unregistration is often the simplest event of a discovery provider, sending only the endpoint UID.

Export/Import Endpoints

The endpoints objects are declared in `pelix.remote.beans` by the *`ExportEndpoint`* and *`ImportEndpoint`* classes.

Both contain the following information:

- UID: the unique identifier of the endpoint. It is a class-4 UUID, which should be unique across frameworks.
- Framework: the UID of the framework providing the endpoint. It is mainly used to clean up the endpoints of a lost framework. If too many endpoint UID collisions are reported, it could be used as a secondary key.
- Name: the name of the endpoint. It can have a meaning for the transport provider, but isn't used by Pelix itself.
- Properties: a copy of the current properties of the remote service.
- Specifications: the list of service exported specifications. A service can choose to export a subset of its specifications, as some could be private or using non-serializable types.
- Configurations: the list of transports allowed to export this endpoint or used for importing it.

Finally, the *`ExportEndpoint`* object also gives access to the service reference and implemnetation, in order to let transport providers access the methods and properties of the service.

Core Services

The core services of the Pelix Remote Services implementation is based on two services:

- the *exports dispatcher* which keeps track of and notifies the discovery providers about the export endpoints created/updated/deleted by transport providers. If a discovery provider appears after the creation of an export endpoint, it will still be notified by the exports dispatcher.

  This service is provided by an auto-instantiated component from the `pelix.remote.dispatcher` bundle. It provides a *`pelix.remote.dispatcher`* service.

- the *imports registry* which keeps track of and notifies the transports providers about the import endpoints, according to the notifications from the discovery providers. If a transport provider appears after the registration of an import endpoint, it will nevertheless be notified by the imports registry of existing endpoints.

  This service is provided by an auto-instantiated component from the `pelix.remote.registry` bundle. It provides a *`pelix.remote.registry`* service.

Dispatcher Servlet

The content of the *exports dispatcher* can be exposed by the *dispatcher servlet*, provided by the same bundle as the *exports dispatcher*, `pelix.remote.dispatcher`. Most discovery providers rely on this servlet as it allows to get the list of exported endpoints, or the details of a single one, in JSON format.

This servlet must be instantiated explicitly using its `pelix-remote-dispatcher-servlet-factory` factory. As it is a servlet, it requires the HTTP service to be up and running to provide it to clients.

Its API is very simple:

- `/framework`: returns the framework UID as a JSON string

- `/endpoints`: returns the whole list of the export endpoints registered in the exports dispatcher, as a JSON array of JSON objects.
- `/endpoint/<uid>`: returns the export endpoint with the given UID as a JSON object.

Discovery Providers

A framework must discover a service before being able to use it. Pelix/iPOPO provides a set of discovery protocols:

- a home-made protocol based on UDP multicast packets, which supports addition, update and removal of services;
- a home-made protocol based on MQTT, which supports addition, update and removal of services;
- mDNS, which is a standard but doesn't support service update;
- a discovery service based on Redis.

Transport Providers

The *remote services* implementation supports XML-RPC (using the xmlrpc standard package), but it is recommended to use JSON-RPC instead (using the jsonrpclib-pelix third-party module). Indeed, the JSON-RPC layer has a better handling of dictionaries and custom types. iPOPO also supports a variant of JSON-RPC, *Jabsorb-RPC*, which adds Java type information to the arguments and results. As long as a Java interface is correctly implementing, this protocol allows a Python service to be used by a remote OSGi Java framework, and vice-versa. The OSGi framework must host the Java implementation of the Pelix Remote Services.

All those protocols require the HTTP service to be up and running to work. Finally, iPOPO also supports a kind of *MQTT-RPC* protocol, *i.e.* JSON-RPC over MQTT.

Providers included with Pelix/iPOPO

This section gives more details about the usage of the discovery and transport providers included in Pelix/iPOPO. You'll need at least a discovery and a compatible transport provider for Pelix Remote Services to work.

Apart MQTT, the discovery and transport providers are independent and can be used with one another.

Multicast Discovery

Bundle pelix.remote.discovery.multicast

Factory pelix-remote-discovery-multicast-factory

Requires HTTP Service, Dispatcher Servlet

Libraries *nothing* (based on the Python Standard Library)

Pelix comes with a home-made UDP multicast discovery protocol, implemented in the `pelix.remote.discovery.multicast` bundle. This is the original discovery protocol of Pelix/iPOPO and the most reliable one in small local area networks. A Java version of this protocol is provided by the Cohorte Remote Services implementation.

This protocol consists in minimalist packets on remote service registration, update and unregistration. They mainly contain the notification event type, the port of the HTTP server of the framework and the path to the dispatcher servlet. The IP of the framework is the source IP of the multicast packet: this allows to get a valid address for frameworks on servers with multiple network interfaces.

This provider relies on the HTTP server and the *dispatcher servlet*. It doesn't have external dependencies.

The bundle provides a `pelix-remote-discovery-multicast-factory` iPOPO factory, which **must** be instantiated to work. It can be configured with the following properties:

Property	Default value	Description
multicast.group	239.0.0.1	The multicast group (address) to join to send and receive discovery messages.
multicast.port	42000	The multicast port to listen to

To use this discovery provider, you'll need to install the following bundles and instantiate the associated components:

```
# Start the HTTP service with default parameters
install pelix.http.basic
start $?
instantiate pelix.http.service.basic.factory httpd

# Install Remote Services Core
install pelix.remote.registry
start $?
install pelix.remote.dispatcher
start $?

# Instantiate the dispatcher servlet
instantiate pelix-remote-dispatcher-servlet-factory dispatcher-servlet

# Install and start the multicast discovery with the default parameters
install pelix.remote.discovery.multicast
start $?
instantiate pelix-remote-discovery-multicast-factory discovery-mcast
```

mDNS Discovery

Bundle pelix.remote.discovery.mdns

Factory pelix-remote-discovery-zeroconf-factory

Requires HTTP Service, Dispatcher Servlet

Libraries pyzeroconf

The mDNS protocol, also known as Zeroconf, is a standard protocol based on multicast packets. It provides a Service Discovery layer (mDNS-SD) based on the DNS-SD specification.

Unlike the home-made multicast protocol, this one doesn't support service updates and gives troubles with service unregistrations (frameworks lost, . . .). As a result, it should be used only if it is required to interact with other mDNS devices.

In order to work with the mDNS discovery from the Eclipse Communication Framework, the `pyzeroconf` library must be patched: the `.local.` check in `zeroconf.mdns.DNSQuestion` must be removed (around line 220).

This provider is implemented in the `pelix.remote.discovery.mdns` bundle, which provides a `pelix-remote-discovery-zeroconf-factory` iPOPO factory, which **must** be instantiated to work. It can be configured with the following properties:

Property	Default value	Description
zeroconf.service.type	_pelix_rs._tcp.local.	Zeroconf service type of exported services
zeroconf.ttl	60	Time To Live of services (in seconds)

To use this discovery provider, you'll need to install the following bundles and instantiate the associated components:

```
# Start the HTTP service with default parameters
install pelix.http.basic
start $?
instantiate pelix.http.service.basic.factory httpd

# Install Remote Services Core
install pelix.remote.registry
start $?
install pelix.remote.dispatcher
start $?

# Instantiate the dispatcher servlet
instantiate pelix-remote-dispatcher-servlet-factory dispatcher-servlet

# Install and start the mDNS discovery with the default parameters
install pelix.remote.discovery.mdns
start $?
instantiate pelix-remote-discovery-zeroconf-factory discovery-mdns
```

Redis Discovery

Bundle pelix.remote.discovery.redis

Factory pelix-remote-discovery-redis-factory

Requires *nothing* (all is stored in the Redis database)

Libraries redis

The Redis discovery is the only one working well in Docker (Swarm) networks. It uses a Redis database to store the host name of each framework and the description of each exported endpoint of each framework. Those description are stored in the OSGi standard EDEF XML format, so it should be possible to implement a Java version of this discovery provider. The Redis discovery uses the *key events* of the database to be notified by the latter when a framework or an exported service is registered, updated, unregistered or timed out, which makes it both robust and reactive.

This provider is implemented in the `pelix.remote.discovery.redis` bundle, which provides a `pelix-remote-discovery-redis-factory` iPOPO factory, which **must** be instantiated to work. It can be configured with the following properties:

Property	Default value	Description
redis.host	localhost	The hostname of the Redis server
redis.port	46379	The port the Redis server listens to
redis.db	0	The Redis database to use (integer)
redis.password	None	Password to access the Redis database
heartbeat.delay	10	Delay in seconds between framework heart beats

To use this discovery provider, you'll need to install the following bundles and instantiate the associated components:

```
# Install Remote Services Core
install pelix.remote.registry
start $?
install pelix.remote.dispatcher
start $?
```

(continues on next page)

(continued from previous page)

```
# Install and start the Redis discovery with the default parameters
install pelix.remote.discovery.redis
start $?
instantiate pelix-remote-discovery-redis-factory discovery-redis
```

XML-RPC Transport

Bundle pelix.remote.xml_rpc

Factories pelix-xmlrpc-exporter-factory, pelix-xmlrpc-importer-factory

Requires HTTP Service

Libraries *nothing* (based on the Python Standard Library)

The XML-RPC transport is the first one having been implemented in Pelix/iPOPO. Its main advantage is that is doesn't depend on an external library, XML-RPC being supported by the Python Standard Library.

It has some troubles with complex and custom types (dictionaries, . . .), but can be used without problems on primitive types. The JSON-RPC transport can be preferred in most cases.

Like most of the transport providers, this one is split in two components: the exporter and the importer. Both must be instantiated manually.

The exporter instance can be configured with the following property:

Property	Default value	Description
pelix.http.path	/XML-RPC	The path to the XML-RPC exporter servlet

To use this transport provider, you'll need to install the following bundles and instantiate the associated components:

```
# Start the HTTP service with default parameters
install pelix.http.basic
start $?
instantiate pelix.http.service.basic.factory httpd

# Install Remote Services Core
install pelix.remote.registry
start $?
install pelix.remote.dispatcher
start $?

# Install and start the XML-RPC importer and exporter with the default
# parameters
install pelix.remote.xml_rpc
start $?
instantiate pelix-xmlrpc-exporter-factory xmlrpc-exporter
instantiate pelix-xmlrpc-importer-factory xmlrpc-importer
```

JSON-RPC Transport

Bundle pelix.remote.json_rpc

Factories pelix-jsonrpc-exporter-factory, pelix-jsonrpc-importer-factory

Requires HTTP Service

Libraries jsonrpclib-pelix (installation requirement of iPOPO)

The JSON-RPC transport is the recommended one in Pelix/iPOPO. It depends on an external library, jsonrpclib-pelix which has no transient dependency. It has way less troubles with complex and custom types than the XML-RPC transport, which eases the development of most of Pelix/iPOPO applications.

Like most of the transport providers, this one is split in two components: the exporter and the importer. Both must be instantiated manually.

The exporter instance can be configured with the following property:

Property	Default value	Description
pelix.http.path	/JSON-RPC	The path to the JSON-RPC exporter servlet

To use this transport provider, you'll need to install the following bundles and instantiate the associated components:

```
# Start the HTTP service with default parameters
install pelix.http.basic
start $?
instantiate pelix.http.service.basic.factory httpd

# Install Remote Services Core
install pelix.remote.registry
start $?
install pelix.remote.dispatcher
start $?

# Install and start the JSON-RPC importer and exporter with the default
# parameters
install pelix.remote.json_rpc
start $?
instantiate pelix-jsonrpc-exporter-factory jsonrpc-exporter
instantiate pelix-jsonrpc-importer-factory jsonrpc-importer
```

Jabsorb-RPC Transport

Bundle pelix.remote.transport.jabsorb_rpc

Factories pelix-jabsorbrpc-exporter-factory, pelix-jabsorbrpc-importer-factory

Requires HTTP Service

Libraries jsonrpclib-pelix (installation requirement of iPOPO)

The JABSORB-RPC transport is based on a variant of the JSON-RPC protocol. It adds Java typing hints to ease unmarshalling on Java clients, like the Cohorte Remote Services implementation. The additional information comes at small cost, but this transport shouldn't be used when no Java frameworks are expected: it doesn't provide more features than JSON-RPC in a 100% Python environment.

Like the JSON-RPC transport, it depends on an external library, jsonrpclib-pelix which has no transient dependency.

Like most of the transport providers, this one is split in two components: the exporter and the importer. Both must be instantiated manually.

The exporter instance can be configured with the following property:

Property	Default value	Description
pelix.http.path	/JABSORB-RPC	The path to the JABSORB-RPC exporter servlet

To use this transport provider, you'll need to install the following bundles and instantiate the associated components:

```
# Start the HTTP service with default parameters
install pelix.http.basic
start $?
instantiate pelix.http.service.basic.factory httpd

# Install Remote Services Core
install pelix.remote.registry
start $?
install pelix.remote.dispatcher
start $?

# Install and start the JABSORB-RPC importer and exporter with the default
# parameters
install pelix.remote.transport.jabsorb_rpc
start $?
instantiate pelix-jabsorbrpc-exporter-factory jabsorbrpc-exporter
instantiate pelix-jabsorbrpc-importer-factory jabsorbrpc-importer
```

MQTT discovery and MQTT-RPC Transport

Bundle pelix.remote.discovery.mqtt, pelix.remote.transport.mqtt_rpc

Factories pelix-remote-discovery-mqtt-factory, pelix-mqttrpc-exporter-factory, pelix-mqttrpc-importer-factory

Requires *nothing* (everything goes through MQTT messages)

Libraries paho

Finally, the MQTT discovery and transport protocols have been developped as a proof of concept with the fabMSTIC fablab of the Grenoble Alps University.

The idea was to rely on the lightweight MQTT messages to provide both discovery and transport mechanisms, and to let them be handled by low-power devices like small Arduino boards. Mixed results were obtained: it worked but the performances were not those intended, mainly in terms of latencies.

Those providers are kept in Pelix/iPOPO as they work and provide a non-HTTP way to communicate, but they won't be updated without new contributions (pull requests, . . .).

They rely on the Eclipse Paho library, previously known as the Mosquitto library.

The discovery instance can be configured with the following properties:

Property	Default value Description	
mqtt.host	localhost	Host of the MQTT server
mqtt.port	1883	Port of the MQTT server
topic.prefix	pelix/{appid}/remote-services	Prefix of all MQTT messages (format string accepting the `appid` entry)
application.id	None	Application ID, to allow multiple applications on the same server

The transport exporter and importer instances should be configured with the same `mqtt.host` and `mqtt.port` properties as the discovery service.

To use the MQTT providers, you'll need to install the following bundles and instantiate the associated components:

```
# Install Remote Services Core
install pelix.remote.registry
start $?
install pelix.remote.dispatcher
start $?

# Install and start the MQTT discovery and the MQTT-RPC importer and exporter
# with the default parameters
install pelix.remote.discovery.mqtt
start $?
instantiate pelix-remote-discovery-mqtt-factory mqttrpc-discovery

install pelix.remote.transport.mqtt_rpc
start $?
instantiate pelix-mqttrpc-exporter-factory mqttrpc-exporter
instantiate pelix-mqttrpc-importer-factory mqttrpc-importer
```

API

Endpoints

`ExportEndpoint` objects are created by transport providers and stored in the registry of the *exports dispatcher*. It is used by discovery providers to create a description of the endpoint to send over the network and suitable for the import-side.

class `pelix.remote.beans.`**`ExportEndpoint`**(*uid*, *fw_uid*, *configurations*, *name*, *svc_ref*, *service*, *properties*)

Represents an export end point (one per group of configuration types)

Parameters

- **`uid`** – Unique identified of the end point
- **`fw_uid`** – The framework UID
- **`configurations`** – Kinds of end point (xmlrpc, . . .)
- **`name`** – Name of the end point
- **`svc_ref`** – ServiceReference of the exported service
- **`service`** – Instance of the exported service
- **`properties`** – Extra properties

Raises `ValueError` – Invalid UID or the end point exports nothing (all specifications have been filtered)

`get_properties`()

Returns merged properties

Returns Endpoint merged properties

`make_import_properties`()

Returns the properties of this endpoint where export properties have been replaced by import ones

Returns A dictionary with import properties

`rename`(*new_name*)

Updates the endpoint name

Parameters **`new_name`** – The new name of the endpoint

`configurations`
Configurations of this end point

`framework`
Framework UID

`instance`
Service instance

`name`
Name of the end point

`reference`
Service reference

`specifications`
Returns the exported specifications

`uid`
End point unique identifier

`ImportEndpoint` objects are the description of an endpoint on the consumer side. They are given by the *imports registry* to the *transport providers* on the import side.

class `pelix.remote.beans.`**`ImportEndpoint`**(*uid*, *framework*, *configurations*, *name*, *specifications*, *properties*)
Represents an end point to access an imported service

Parameters

- **`uid`** – Unique identified of the end point
- **`framework`** – UID of the framework exporting the end point (can be None)
- **`configurations`** – Kinds of end point (xmlrpc, . . .)
- **`name`** – Name of the end point
- **`specifications`** – Specifications of the exported service
- **`properties`** – Properties of the service

`configurations`
Kind of end point

`framework`
UID of the framework exporting this end point

`name`
Name of the end point

`properties`
Properties of the imported service

`specifications`
Specifications of the service

`uid`
End point unique identifier

Core Services

The *exports dispatcher* service provides the `pelix.remote.dispatcher` (constant string stored in `pelix.remote.SERVICE_DISPATCHER`) service, with the following API:

class `pelix.remote.dispatcher.`**`Dispatcher`**

Common dispatcher for all exporters

`get_endpoint`(*uid*)

Retrieves an end point description, selected by its UID. Returns None if the UID is unknown.

Parameters **`uid`** – UID of an end point

Returns An *`ExportEndpoint`* or None.

`get_endpoints`(*kind=None*, *name=None*)

Retrieves all end points matching the given kind and/or name

Parameters

- **`kind`** – A kind of end point
- **`name`** – The name of the end point

Returns A list of *`ExportEndpoint`* matching the parameters

The *import registry* service provides the `pelix.remote.registry` (constant string stored in `pelix.remote.SERVICE_REGISTRY`) service, with the following API:

class `pelix.remote.registry.`**`ImportsRegistry`**

Registry of discovered end points. End points are identified by their UID

`add`(*endpoint*)

Registers an end point and notifies listeners. Does nothing if the endpoint UID was already known.

Parameters **`endpoint`** – An *`ImportEndpoint`* object

Returns True if the end point has been added

`contains`(*endpoint*)

Checks if an endpoint is in the registry

Parameters **`endpoint`** – An endpoint UID or an *`ImportEndpoint`* object

Returns True if the endpoint is known, else False

`lost_framework`(*uid*)

Unregisters all the end points associated to the given framework UID

Parameters **`uid`** – The UID of a framework

`remove`(*uid*)

Unregisters an end point and notifies listeners

Parameters **`uid`** – The UID of the end point to unregister

Returns True if the endpoint was known

`update`(*uid*, *new_properties*)

Updates an end point and notifies listeners

Parameters

- **`uid`** – The UID of the end point
- **`new_properties`** – The new properties of the end point

Returns True if the endpoint is known, else False

3.5.10 Configuration Admin

Concept

The Configuration Admin service allows to easily set, update and delete the configuration (a dictionary) of managed services.

The Configuration Admin service can be used by any bundle to configure a service, either by creating, updating or deleting a Configuration object.

The Configuration Admin service handles the persistence of configurations and distributes them to their target services.

Two kinds of managed services exist: * Managed Services, which handle the configuration as is * Managed Service Factories, which can handle multiple configuration of a kind

Note: Even if iPOPO doesn't fully respect it, you can find details about the Configuration Admin Service Specification in the chapter 104 of the OSGi Compendium Services Specification.

Note: This page is highly inspired from the Configuration Admin tutorial from the Apache Felix project.

Basic Usage

Here is a bery basic example of a managed service able to handle a single configuration. This configuration contains a single entry: the length of a pretty printer.

The managed service must provide the `pelix.configadmin.managed` specification, associated to a persistent ID (PID) identifying its configuration (`service.pid`).

The PID is just a string, which must be globally unique. Assuming a simple case where your pretty printer configurator receives the configuration has a unique class name, you may well use that name.

So lets assume, our managed service is called `PrettyPrinter` and that name is also used as the PID. The class would be:

```python
class PrettyPrinter:
    def updated(self, props):
        """
        A configuration has been updated
        """
        if props is None:
            # Configuration have been deleted
            pass
        else:
            # Apply configuration from config admin
            pass
```

Now, in your bundle activator's `start()` method you can register `PrettyPrinter` as a managed service:

```python
@BundleActivator
class Activator:
    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        self.svc_reg = None

    def start(self, context):
        svc_props = {"service.pid": "pretty.printer"}
        self.svc_reg = context.register_service(
            "pelix.configadmin.managed", PrettyPrinter(), svc_props)

    def stop(self, context):
        if self.svc_reg is not None:
            self.svc_reg.unregister()
            self.svc_reg = None
```

That's more or less it. You may now go on to use your favourite tool to create and edit the configuration for the Pretty Printer, for example something like this:

```
# Get the current configuration
pid = "pretty.printer"
config = config_admin_svc.get_configuration(pid)
props = config.get_properties()
if props is None:
   props = {}

# Set properties
props.put("key", "value")

# Update the configuration
config.update(props)
```

After the call to `update()` the Configuration Admin service persists the new configuration data and sends an update to the managed service registered with the service PID `pretty.printer`, which happens to be our PrettyPrinter class as expected.

Managed Service Factory example

Registering a service as a Managed Service Factory means that it will be able to receive several different configuration dictionaries. This can be useful when used by a Service Factory, that is, a service responsible for creating a distinct instance of a service according to the bundle consuming it.

A Managed Service Factory needs to provide the `pelix.configadmin.managed.factory` specification, as shown below:

```
class SmsSenderFactory:
    def __init__(self):
        self.existing = {}

    def updated(pid, props):
        """
        Called when a configuration has been created or updated
        """
        if pid in self.existing:
            # Service already exist
            self.existing[pid].configure(props)
        else:
            # Create the service
            svc = self.create_instance()
```

(continues on next page)

(continued from previous page)

```
            svc.configure(props)
            self.existing[pid] = service

    def deleted(pid):
        """
        Called when a configuration has been deleted
        """
        self.existing[pid].close()
        del self.existing[pid]
```

The example above shows that, differently from a managed service, the managed service factory is designed to manage multiple instances of a service.

In fact, the `updated` method accept a PID and a dictionary as arguments, thus allowing to associate a certain configuration dictionary to a particular service instance (identified by the PID).

Note also that the managed service factory specification requires to implement (besides the getName method) a `deleted` method: this method is invoked when the Configuration Admin service asks the managed service factory to delete a specific instance.

The registration of a managed service factory follows the same steps of the managed service sample:

```
@BundleActivator
class Activator:
    def __init__(self):
        self.svc_reg = None

    def start(self, context):
        svc_props = {"service.pid": "sms.sender"}
        self.svc_reg = context.register_service(
            "pelix.configadmin.managed.factory", SmsSenderFactory(),
            svc_props)

    def stop(self, context):
        if self.svc_reg is not None:
            self.svc_reg.unregister()
            self.svc_reg = None
```

Finally, using the ConfigurationAdmin interface, it is possible to send new or updated configuration dictionaries to the newly created managed service factory:

```
@BundleActivator
class Activator:
    def __init__(self):
        self.configs = {}

    def start(self, context):
        svc_ref = context.get_service_reference("pelix.configadmin")
        if svc_ref is not None:
            # Get the configuration admin service
            config_admin_svc = context.get_service(svc_ref)

            # Create a new configuration for the given factory
            config = config_admin_svc.create_factory_configuration(
                "sms.sender")

            # Update it
```

(continues on next page)

(continued from previous page)

```
            props = {"key": "value"}
            config.update(props)

            # Store it for future use
            self.configs[config.get_pid()] = config

    def stop(self, context):
        # Clear all configurations (for this example)
        for config in self.configs:
            config.delete()

        self.configs.clear()
```

3.5.11 EventAdmin service

Description

The EventAdmin service defines an inter-bundle communication mechanism.

Note: This service is inspired from the EventAdmin specification in OSGi, but without the `Event` class.

It is a publish/subscribe communication service, using the whiteboard pattern, that allows to send an *event*:

- the publisher of an event uses the EventAdmin service to send its event
- the handler (or subscriber or listener) publishes a service with filtering properties

An event is the association of:

- a topic, a URI-like string that defines the nature of the event
- a set of properties associated to the event

Some properties are defined by the EventAdmin service:

Property	Type	Description
event.sender.framework.uid	str	UID of the framework that emitted the event. Useful in remote services
event.timestamp	float	Time stamp of the event, computed when the event is given to EventAdmin

Usage

Instantiation

The EventAdmin service is implemented in `pelix.services.eventadmin` bundle, as a single iPOPO component. This component must be instantiated programmatically, by using the iPOPO service and the `pelix-services-eventadmin-factory` factory name.

```
from pelix.ipopo.constants import use_ipopo
import pelix.framework

# Start the framework (with iPOPO)
framework = pelix.framework.create_framework(['pelix.ipopo.core'])
```

(continues on next page)

(continued from previous page)

```
framework.start()
context = framework.get_bundle_context()

# Install & start the EventAdmin bundle
context.install_bundle('pelix.services.eventadmin').start()

# Get the iPOPO the service
with use_ipopo(context) as ipopo:
   # Instantiate the EventAdmin component
   ipopo.instantiate('pelix-services-eventadmin-factory',
                     'EventAdmin', {})
```

It can also be instantiated via the Pelix Shell:

```
$ install pelix.services.eventadmin
Bundle ID: 12
$ start 12
Starting bundle 12 (pelix.services.eventadmin)...
$ instantiate pelix-services-eventadmin-factory eventadmin
Component 'eventadmin' instantiated.
```

The EventAdmin component accepts the following property as a configuration:

Property	Default value	Description
pool.threads	10	Number of threads in the pool used for asynchronous delivery

Interfaces

EventAdmin service

The EventAdmin service provides the `pelix.services.eventadmin` specification:

class `pelix.services.eventadmin.`**`EventAdmin`**
The EventAdmin implementation

`post`(*topic*, *properties=None*)
Sends asynchronously the given event

Parameters

- **`topic`** – Topic of event
- **`properties`** – Associated properties

`send`(*topic*, *properties=None*)
Sends synchronously the given event

Parameters

- **`topic`** – Topic of event
- **`properties`** – Associated properties

Both `send` and `post` methods get the topic as first parameter, which must be a URI-like string, *e.g.* `sensor/temperature/changed` and a dictionary as second parameter, which can be `None`.

When sending an event, each handler is notified with a different copy of the property dictionary, avoiding to propagate changes done by a handler.

EventHandler service

An event handler must provide the `pelix.services.eventadmin.handler` specification, which defines by the following method:

`handle_event` (*topic*, *properties*)
Called by the EventAdmin service to notify a handler of a new event

Parameters

- **`topic`** – The topic of the event (str)
- **`properties`** – The properties associated to the event (dict)

Warning: Events sent using the `post()` are delivered from another thread. It is unlikely but possible that sometimes the `handle_event()` method may be called whereas the handler service has been unregistered, for example after the handler component has been invalidated.

It is therefore recommended to check that the injected dependencies used in this method are not `None` before handling the event.

An event handler must associate at least one the following properties to its service:

Property	Type	Description
event.topics	List of str	A list of strings that indicates the topics the topics this handler expects. EventAdmin supports "file name" filters, i.e. with `*` or `?` jokers.
event.filter	str	A LDAP filter string that will be tested on the event properties

Example

In this example, a component will publish an event when it is validated or invalidated. These events will be:

- `example/publisher/validated`
- `example/publisher/invalidated`

The event handler component will provide a service with a topic filter that accepts both topics: `example/publisher/*`

Publisher

The publisher requires the EventAdmin service, which specification is defined in the `pelix.services` module.

```
# iPOPO
from pelix.ipopo.decorators import *
import pelix.ipopo.constants as constants

# EventAdmin constants
import pelix.services

@ComponentFactory('publisher-factory')
# Require the EventAdmin service
@Requires('_event', pelix.services.SERVICE_EVENT_ADMIN)
```

(continues on next page)

(continued from previous page)

```python
# Inject our component name in a field
@Property('_name', constants.IPOPO_INSTANCE_NAME)
# Auto-instantiation
@Instantiate('publisher')
class Publisher(object):
    """
    A sample publisher
    """
    def __init__(self):
        """
        Set up members, to be OK with PEP-8
        """
        # EventAdmin (injected)
        self._event = None

        # Component name (injected property)
        self._name = None

    @Validate
    def validate(self, context):
        """
        Component validated
        """
        # Send a "validated" event
        self._event.send("example/publisher/validated",
                         {"name": self._name})

    @Invalidate
    def invalidate(self, context):
        """
        Component invalidated
        """
        # Post an "invalidated" event
        self._event.send("example/publisher/invalidated",
                         {"name": self._name})
```

Handler

The event handler has no dependency requirement. It has to provide the EventHandler specification, which is defined in the `pelix.services` module.

```python
# iPOPO
from pelix.ipopo.decorators import *
import pelix.ipopo.constants as constants

# EventAdmin constants
import pelix.services

@ComponentFactory('handler-factory')
# Provide the EventHandler service
@Provides(pelix.services.SERVICE_EVENT_HANDLER)
# The event topic filters, injected as a component property that will be
# propagated to its services
@Property('_event_handler_topic', pelix.services.PROP_EVENT_TOPICS,
        ['example/publisher/*'])
```

(continues on next page)

(continued from previous page)

```python
# The event properties filter (optional, here set to None by default)
@Property('_event_handler_filter', pelix.services.PROP_EVENT_FILTER)
# Auto-instantiation
@Instantiate('handler')
class Handler(object):
    """
    A sample event handler
    """
    def __init__(self):
        """
        Set up members
        """
        self._event_handler_topic = None
        self._event_handler_filter = None


    def handle_event(self, topic, properties):
        """
        Event received
        """
        print('Got a {0} event from {1} at {2}' \
              .format(topic, properties['name'],
                      properties[pelix.services.EVENT_PROP_TIMESTAMP]))
```

It is recommended to define an event filter property, even if it is set to `None` by default: it allows to customize the event handler when it is instantiated using the iPOPO API:

```python
# This handler will be notified only of events with a topic matching
# 'example/publisher/*' (default value of 'event.topics'), and in which
# the 'name' property is 'foobar'.
ipopo.instantiate('handler-factory', 'customized-handler',
                  {pelix.services.PROP_EVENT_FILTER: '(name=foobar)'})
```

Shell Commands

It is possible to send events from the Pelix shell, after installing the `pelix.shell.eventadmin` bundle.

This bundle defines two commands, in the `event` scope:

Command	Description
`post <topic> [<property=value> [...]]`	Posts an event on the given topic, with the given properties
`send <topic> [<property=value> [...]]`	Sends an event on the given topic, with the given properties

Here is a sample shell session, considering the sample event handler above has been started. It installs and start the EventAdmin shell bundle:

```
$ install pelix.shell.eventadmin
13
$ start 13
$ event.send example/publisher/activated name=foobar
Got a example/publisher/activated from foobar at 1369125501.028135
```

Events printer utility component

A `pelix-misc-eventadmin-printer-factory` component factory is provided by the `pelix.misc.eventadmin_printer` bundle. It can be used to instantiate components that will print and/or log the event matching a given filter.

Here is a Pelix Shell snippet to instantiate a printer and to send it some events:

```
$ install pelix.shell.eventadmin
13
$ start 13
$ install pelix.misc.eventadmin_printer
14
$ start 14
$ instantiate pelix-misc-eventadmin-printer-factory printerA event.topics=foo/*
Component 'printerA' instantiated.
$ instantiate pelix-misc-eventadmin-printer-factory printerB evt.log=True event.
↪topics=foo/bar/*
Component 'printerB' instantiated.
$ send foo/abc
Event: foo/abc
Properties:
{'event.sender.framework.uid': 'aa180e9b-bb45-4cbf-8092-d45fbe12464f',
 'event.timestamp': 1492698306.1903257}
$ send foo/bar/def
Event: foo/bar/def
Properties:
{'event.sender.framework.uid': 'aa180e9b-bb45-4cbf-8092-d45fbe12464f',
 'event.timestamp': 1492698324.9549854}
Event: foo/bar/def
Properties:
{'event.sender.framework.uid': 'aa180e9b-bb45-4cbf-8092-d45fbe12464f',
 'event.timestamp': 1492698324.9549854}
```

The second event is printed twice as it is handled by both printers.

MQTT Bridge

Pelix provides a bridge to send EventAdmin events to an MQTT server and vice-versa. This can be used to send events between various Pelix frameworks, without the need of the remote services layer, or between different entities sharing an MQTT server.

The component factory, `pelix-services-eventadmin-mqtt-factory`, is provided by the `pelix.services.eventadmin_mqtt` bundle. It can be configured with the following properties:

Property	Default Value	Description
event.topics	`*`	The filter to select the events to share
mqtt.host	localhost	The host name of the MQTT server
mqtt.port	1883	The port the MQTT server is bound to
mqtt.topic.prefix	`/pelix/eventadmin`	The prefix to add to events before sending them over MQTT

Events handled by this component, i.e. matching the filter given at instantiation time, and having the `event.propagate` property set to any value (even `False`) will be sent as messages to the MQTT server with the following modifications:

- the MQTT message topic will be the event topic prefixed by the value of the `mqtt.topic.prefix` property

- if the event topic starts with a slash (`/`), a `pelix.eventadmin.mqtt.start_slash` property is added to the event and is set to `True`
- a `pelix.eventadmin.mqtt.source` is added to the event, containing the UUID of the emitting framework, to avoid loops.

The event properties are then converted to JSON and used as the body the MQTT message.

When an MQTT message starting with the configured prefix is received, it is converted back to an event, given to EventAdmin. Loopback messages are detected and ignored to avoid loops.

CHAPTER 4

API Reference

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

4.1 API

This part of the documentation covers all the core classes and services of iPOPO.

4.1.1 BundleContext Object

The bundle context is the link between a bundle and the framework. It's by the context that you can register services, install other bundles.

class `pelix.framework.`**`BundleContext`**(*framework*, *bundle*)

The bundle context is the link between a bundle and the framework. It is unique for a bundle and is created by the framework once the bundle is installed.

Parameters

- **`framework`** – Hosting framework
- **`bundle`** – The associated bundle

`add_bundle_listener`(*listener*)

Registers a bundle listener, which will be notified each time a bundle is installed, started, stopped or updated.

The listener must be a callable accepting a single parameter:

- **event** – The description of the event (a `BundleEvent` object).

Parameters **`listener`** – The bundle listener to register

Returns True if the listener has been registered, False if it already was

add_framework_stop_listener(*listener*)

Registers a listener that will be called back right before the framework stops

The framework listener must have a method with the following prototype:

```
def framework_stopping(self):
    '''
    No parameter given
    '''
    # ...
```

Parameters **listener** – The framework stop listener

Returns True if the listener has been registered

add_service_listener(*listener*, *ldap_filter=None*, *specification=None*)

Registers a service listener

The service listener must have a method with the following prototype:

```
def service_changed(self, event):
    '''
    Called by Pelix when some service properties changes

    event: A ServiceEvent object
    '''
    # ...
```

Parameters

- **listener** – The listener to register
- **ldap_filter** – Filter that must match the service properties (optional, None to accept all services)
- **specification** – The specification that must provide the service (optional, None to accept all services)

Returns True if the listener has been successfully registered

get_all_service_references(*clazz*, *ldap_filter=None*)

Returns an array of ServiceReference objects. The returned array of ServiceReference objects contains services that were registered under the specified class and match the specified filter expression.

Parameters

- **clazz** – Class implemented by the service
- **ldap_filter** – Service filter

Returns The sorted list of all matching service references, or None

get_bundle(*bundle_id=None*)

Retrieves the `Bundle` object for the bundle matching the given ID (int). If no ID is given (None), the bundle associated to this context is returned.

Parameters **bundle_id** – A bundle ID (optional)

Returns The requested `Bundle` object

Raises **BundleException** – The given ID doesn't exist or is invalid

get_bundles()
Returns the list of all installed bundles

Returns A list of *Bundle* objects

get_framework()
Returns the `Framework` that created this bundle context

Returns The *Framework* object

get_property(*name*)
Returns the value of a property of the framework, else returns the OS environment value.

Parameters **name** – A property name

get_service(*reference*)
Returns the service described with the given reference

Parameters **reference** – A ServiceReference object

Returns The service object itself

get_service_objects(*reference*)
Returns the ServiceObjects object for the service referenced by the specified ServiceReference object.

Parameters **reference** – Reference to a prototype service factory

Returns An intermediate object to get more instances of a service

get_service_reference(*clazz*, *ldap_filter=None*)
Returns a ServiceReference object for a service that implements and was registered under the specified class

Parameters

- **clazz** – The class name with which the service was registered.
- **ldap_filter** – A filter on service properties

Returns A service reference, None if not found

get_service_references(*clazz*, *ldap_filter=None*)
Returns the service references for services that were registered under the specified class by this bundle and matching the given filter

Parameters

- **clazz** – The class name with which the service was registered.
- **ldap_filter** – A filter on service properties

Returns The list of references to the services registered by the calling bundle and matching the filters.

install_bundle(*name*, *path=None*)
Installs the bundle (module) with the given name.

If a path is given, it is inserted in first place in the Python loading path (`sys.path`). All modules loaded alongside this bundle, *i.e.* by this bundle or its dependencies, will be looked after in this path in priority.

Note: Before Pelix 0.5.0, this method returned the ID of the installed bundle, instead of the Bundle object.

Warning: The behavior of the loading process is subject to changes, as it does not allow to safely run multiple frameworks in the same Python interpreter, as they might share global module values.

Parameters

- **`name`** – The name of the bundle to install
- **`path`** – Preferred path to load the module (optional)

Returns The *`Bundle`* object of the installed bundle

Raises **`BundleException`** – Error importing the module or one of its dependencies

`install_package`(*path*, *recursive=False*)

Installs all the modules found in the given package (directory). It is a utility method working like *`install_visiting()`*, with a visitor accepting every module found.

Parameters

- **`path`** – Path of the package (folder)
- **`recursive`** – If True, installs the modules found in sub-directories

Returns A 2-tuple, with the list of installed bundles (*`Bundle`*) and the list of the names of the modules which import failed.

Raises **`ValueError`** – The given path is invalid

`install_visiting`(*path*, *visitor*)

Looks for modules in the given path and installs those accepted by the given visitor.

The visitor must be a callable accepting 3 parameters:

- **fullname** – The full name of the module
- **is_package** – If True, the module is a package
- **module_path** – The path to the module file

Parameters

- **`path`** – Root search path (folder)
- **`visitor`** – The visiting callable

Returns A 2-tuple, with the list of installed bundles (*`Bundle`*) and the list of the names of the modules which import failed.

Raises **`ValueError`** – Invalid path or visitor

`register_service`(*clazz*, *service*, *properties*, *send_event=True*, *factory=False*, *prototype=False*)

Registers a service

Parameters

- **`clazz`** – Class or Classes (list) implemented by this service
- **`service`** – The service instance
- **`properties`** – The services properties (dictionary)
- **`send_event`** – If not, doesn't trigger a service registered event
- **`factory`** – If True, the given service is a service factory

- **`prototype`** – If True, the given service is a prototype service factory (the factory argument is considered True)

Returns A ServiceRegistration object

Raises **`BundleException`** – An error occurred while registering the service

`remove_bundle_listener`(*listener*)
Unregisters the given bundle listener

Parameters **`listener`** – The bundle listener to remove

Returns True if the listener has been unregistered, False if it wasn't registered

`remove_framework_stop_listener`(*listener*)
Unregisters a framework stop listener

Parameters **`listener`** – The framework stop listener

Returns True if the listener has been unregistered

`remove_service_listener`(*listener*)
Unregisters a service listener

Parameters **`listener`** – The service listener

Returns True if the listener has been unregistered

`unget_service`(*reference*)
Disables a reference to the service

Returns True if the bundle was using this reference, else False

4.1.2 Framework Object

The Framework object is a singleton and can be accessed using *`get_bundle(0)`*. This class inherits the methods from *`pelix.framework.Bundle`*.

`class` `pelix.framework.`**`Framework`**(*properties=None*)
The Pelix framework (main) class. It must be instantiated using FrameworkFactory

Sets up the framework.

Parameters **`properties`** – The framework properties

`add_property`(*name, value*)
Adds a property to the framework **if it is not yet set**.

If the property already exists (same name), then nothing is done. Properties can't be updated.

Parameters

- **`name`** – The property name
- **`value`** – The value to set

Returns True if the property was stored, else False

`delete`(*force=False*)
Deletes the current framework

Parameters **`force`** – If True, stops the framework before deleting it

Returns True if the framework has been delete, False if is couldn't

find_service_references(*clazz=None*, *ldap_filter=None*, *only_one=False*)
Finds all services references matching the given filter.

Parameters

- **clazz** – Class implemented by the service
- **ldap_filter** – Service filter
- **only_one** – Return the first matching service reference only

Returns A list of found reference, or None

Raises **BundleException** – An error occurred looking for service references

get_bundle_by_id(*bundle_id*)
Retrieves the bundle with the given ID

Parameters **bundle_id** – ID of an installed bundle

Returns The requested bundle

Raises **BundleException** – The ID is invalid

get_bundle_by_name(*bundle_name*)
Retrieves the bundle with the given name

Parameters **bundle_name** – Name of the bundle to look for

Returns The requested bundle, None if not found

get_bundles()
Returns the list of all installed bundles

Returns the list of all installed bundles

get_properties()
Retrieves a copy of the stored framework properties.

get_property(*name*)
Retrieves a framework or system property. As framework properties don't change while it's running, this method don't need to be protected.

Parameters **name** – The property name

get_property_keys()
Returns an array of the keys in the properties of the service

Returns An array of property keys.

get_service(*bundle*, *reference*)
Retrieves the service corresponding to the given reference

Parameters

- **bundle** – The bundle requiring the service
- **reference** – A service reference

Returns The requested service

Raises

- **BundleException** – The service could not be found
- **TypeError** – The argument is not a ServiceReference object

get_symbolic_name()
Retrieves the framework symbolic name

Returns Always “pelix.framework”

install_bundle(*name*, *path=None*)
Installs the bundle with the given name

Note: Before Pelix 0.5.0, this method returned the ID of the installed bundle, instead of the Bundle object.

WARNING: The behavior of the loading process is subject to changes, as it does not allow to safely run multiple frameworks in the same Python interpreter, as they might share global module values.

Parameters

- **name** – A bundle name
- **path** – Preferred path to load the module

Returns The installed Bundle object

Raises **BundleException** – Something happened

install_package(*path*, *recursive=False*, *prefix=None*)
Installs all the modules found in the given package

Parameters

- **path** – Path of the package (folder)
- **recursive** – If True, install the sub-packages too
- **prefix** – (**internal**) Prefix for all found modules

Returns A 2-tuple, with the list of installed bundles and the list of failed modules names

Raises **ValueError** – Invalid path

install_visiting(*path*, *visitor*, *prefix=None*)
Installs all the modules found in the given path if they are accepted by the visitor.

The visitor must be a callable accepting 3 parameters:

- fullname: The full name of the module
- is_package: If True, the module is a package
- module_path: The path to the module file

Parameters

- **path** – Root search path
- **visitor** – The visiting callable
- **prefix** – (**internal**) Prefix for all found modules

Returns A 2-tuple, with the list of installed bundles and the list of failed modules names

Raises **ValueError** – Invalid path or visitor

register_service(*bundle*, *clazz*, *service*, *properties*, *send_event*, *factory=False*, *prototype=False*)
Registers a service and calls the listeners

Parameters

- **bundle** – The bundle registering the service

- **`clazz`** – Name(s) of the interface(s) implemented by service
- **`service`** – The service to register
- **`properties`** – Service properties
- **`send_event`** – If not, doesn't trigger a service registered event
- **`factory`** – If True, the given service is a service factory
- **`prototype`** – If True, the given service is a prototype service factory (the factory argument is considered True)

Returns A ServiceRegistration object

Raises **`BundleException`** – An error occurred while registering the service

`start`()
Starts the framework

Returns True if the bundle has been started, False if it was already running

Raises **`BundleException`** – A bundle failed to start

`stop`()
Stops the framework

Returns True if the framework stopped, False it wasn't running

`uninstall`()
A framework can't be uninstalled

Raises **`BundleException`** – This method must not be called

`uninstall_bundle`(*bundle*)
Ends the uninstallation of the given bundle (must be called by Bundle)

Parameters **`bundle`** – The bundle to uninstall

Raises **`BundleException`** – Invalid bundle

`unregister_service`(*registration*)
Unregisters the given service

Parameters **`registration`** – A ServiceRegistration to the service to unregister

Raises **`BundleException`** – Invalid reference

`update`()
Stops and starts the framework, if the framework is active.

Raises **`BundleException`** – Something wrong occurred while stopping or starting the framework.

`wait_for_stop`(*timeout=None*)
Waits for the framework to stop. Does nothing if the framework bundle is not in ACTIVE state.

Uses a threading.Condition object

Parameters **`timeout`** – The maximum time to wait (in seconds)

Returns True if the framework has stopped, False if the timeout raised

4.1.3 Bundle Object

This object gives access to the description of an installed bundle. It is useful to check the path of the source module, the version, etc.

class `pelix.framework.`**`Bundle`**(*framework*, *bundle_id*, *name*, *module_*)

Represents a "bundle" in Pelix

Sets up the bundle descriptor

Parameters

- **`framework`** – The host framework
- **`bundle_id`** – The ID of the bundle in the host framework
- **`name`** – The bundle symbolic name
- **`module`** – The bundle module

`get_bundle_context`()

Retrieves the bundle context

Returns The bundle context

`get_bundle_id`()

Retrieves the bundle ID

Returns The bundle ID

`get_location`()

Retrieves the location of this module

Returns The location of the Pelix module, or an empty string

`get_module`()

Retrieves the Python module corresponding to the bundle

Returns The Python module

`get_registered_services`()

Returns this bundle's ServiceReference list for all services it has registered or an empty list

The list is valid at the time of the call to this method, however, as the Framework is a very dynamic environment, services can be modified or unregistered at any time.

Returns An array of ServiceReference objects

Raises **`BundleException`** – If the bundle has been uninstalled

`get_services_in_use`()

Returns this bundle's ServiceReference list for all services it is using or an empty list. A bundle is considered to be using a service if its use count for that service is greater than zero.

The list is valid at the time of the call to this method, however, as the Framework is a very dynamic environment, services can be modified or unregistered at any time.

Returns An array of ServiceReference objects

Raises **`BundleException`** – If the bundle has been uninstalled

`get_state`()

Retrieves the bundle state

Returns The bundle state

get_symbolic_name()
Retrieves the bundle symbolic name (its Python module name)

Returns The bundle symbolic name

get_version()
Retrieves the bundle version, using the `__version__` or `__version_info__` attributes of its module.

Returns The bundle version, "0.0.0" by default

start()
Starts the bundle. Does nothing if the bundle is already starting or active.

Raises **BundleException** – The framework is not yet started or the bundle activator failed.

stop()
Stops the bundle. Does nothing if the bundle is already stopped.

Raises **BundleException** – The bundle activator failed.

uninstall()
Uninstalls the bundle

update()
Updates the bundle

ACTIVE = 32
The bundle is now running

INSTALLED = 2
The bundle is installed but not yet resolved

RESOLVED = 4
The bundle is resolved and is able to be started

STARTING = 8
The bundle is in the process of starting

STOPPING = 16
The bundle is in the process of stopping

UNINSTALLED = 1
The bundle is uninstalled and may not be used

CHAPTER 5

Additional Notes

Design notes, legal information and changelog are here for the interested.

5.1 Who uses iPOPO ?

If you want to add your name here, send a mail on the ipopo-users mailing list.

5.1.1 Cohorte Technologies (isandlaTech)

Cohorte Technologies is the main sponsor and user of iPOPO. It uses iPOPO as the basis of all its core developments, like the Cohorte Framework.

5.1.2 G2ELab / G-Scop

PREDIS is a complex of several platforms dedicated for research and education. These platforms gather many industrials and academic partners working around emerging axes of electrical engineering and energy management. PREDIS platforms are part of the Ense3 school which trains high-level engineers and doctors able to take up the challenges associated with the new energy order, with the increasing demand of water, both in quantity and quality, and with the sustainable development and country planning.

The PREDIS Smart Building platform is mainly focused on energy management in buildings such as offices. Two laboratories are developing their research activities in Predis, the Grenoble Electrical Engineering lab (G2Elab) and the Design and Production Sciences laboratories (G-Scop).

The main topics studied in PREDIS SB are:

- Multi-sensor monitoring
- User activities and their energy impact analysis
- Multi-physical modelling, measurement handle and sensitivity analysing
- Optimal control strategies development.

5.1.3 Polytech Grenoble / AIR

AIR means Ambient Intelligence Room.

Ambient intelligence (AmI) is now part of the everyday world of users. It is found in all areas of activity: intelligent building with energy control and maintenance, intelligent electrical grid (*smart grid*), health care with home care, transportation and supply chain, public and private security, culture and entertainment (*infotainment*) with serious games, . . .

The AmI applications development relies primarily pooling of expertise in many areas of computer science and electronics which are generally purchased separately in university curricula and engineering schools. AmI education focuses on developing applications for a wide range of smart objects (the IT server 3G user terminal and the on-board sensor Zigbee/6LoWPAN instrumenting the physical environment). This teaching can be done properly only in the context of experimental practice through group projects and student assignments for various application areas. The experiments can achieve scaled in specialized rooms.

The AIR platform is a *fablab* (Fabrication Laboratory) for engineering students and Grenoble students to invent, create and implement projects and application objects ambient intelligence through their training. The platform of the Grenoble Alps University is housed in the Polytech Grenoble building. AIR is an educational platform of the labex Persyval.

5.2 Release Notes

5.2.1 iPOPO 0.7.0

Release Date 2017-12-30

Project

- Removed Python 2.6 compatibility code
- New version of the logo, with SVG sources in the repository
- Added some tests for `install_package()`

Pelix

- When a bundle is stopped, the framework now automatically releases the services it consumed. This was required to avoid stale references when using (prototype) service factories. **WARNING:** this can lead to issues if you were using stale references to pass information from one bundle version to another (which is bad).
- Added support for Prototype Service Factories, which were missing from issue Service Factories (#75).
- Handle deprecation of the `imp` module (see #85)
- Added a `delete()` method to the `Framework` class. The `FrameworkFactory` class can now be fully avoided by developers.

5.2.2 iPOPO 0.6.5

Release Date 2017-09-17

Project

- Project documentation migrated to Read The Docs as the previous documentation server crashed. All references to the previous server (`coderxpress.net`) have been removed.
- The documentation is being completely rewritten while it is converted from Dokuwiki to Sphinx.
- Removed Pypy 3 from Travis-CI/Tox tests, as it is not compatible with pip.

Pelix

- The import path normalization now ensures that the full path of the initial working directory is stored in the path, and that the current working directory marker (empty string) is kept as the first entry of the Python path.
- Merged pull request #65, to ignore import errors when normalizing the Python path.
- Merged pull request #68, correcting the behaviour of the thread pool.

iPOPO

- The `@Validate` method of components is now always called after the bundle activator has returned. (#66)
- Added a `get_instance(name)` method to access to the component instance object by its name. (#74)

HTTP

- Added some utility methods to `HttpServletRequest`:
  - `get_command()`: get the HTTP command of the request
  - `get_prefix_path()`: get the servlet prefix path
  - `get_sub_path()`: get the part of the path corresponding to the servlet (*i.e.* without the prefix path)
- `get_servlet()` now returns the servlet prefix along with the servlet and the server parameters.
- Added a `pelix.https` service property and an `is_https()` service method to indicate that the server uses HTTPS.

- Added a utility module, `pelix.http.routing`, which eases the routing of HTTP requests whith decorators like `@Http`, `@HttpGet`...
- Merged pull request #70, avoiding remote HTTP servlets to be used by the local HTTP server.

Remote Services

- JSON-RPC and XML-RPC transports providers now support HTTPS.
- Added a Redis-based discovery provider, working with all HTTP-based transport providers.

Shell

- Added the *Configuration Handler*, which allows to give a JSON file to set the initial configuration of a framework: properties, bundles, instances, ...

5.2.3 iPOPO 0.6.4

Release Date 2016-06-12

iPOPO

- Added `@RequiresVariableFilter`, which works like `@Requires` but also supports the use of component properties as variables in LDAP filter.
- Added `@HiddenProperty`, which extends `@Property`, but ensures that the property key and value won't be seen in the description API nor in the shell. (it will stay visible using the standard reflection API of Python)

HTTP Service

- The HTTP basic component now support HTTPS. It is activated when given two files (a certificate and a key) in its component properties. A password can also be given if the key file is encrypted. This is a prototype feature and should be used carefully. Also, it should not be used with remote services.

Services

- A new *log service* has been added to this version, though the `pelix.misc.log` bundle. It provides the OSGi API to log traces, but also keeps track of the traces written with the `logging` module. The log entries can be accessed locally (but not through remote services). They can be printed in the shell using commands provided by pelix.shell.log.

5.2.4 iPOPO 0.6.3

Release Date 2015-10-23

Project

- iPOPO now has a logo ! (thanks to @debbabi)
- README file has been rewritten
- Better PEP-8 compliance
- Updated `jsonrpclib-pelix` requirement version to 0.2.6

Framework

- Optimization of the service registry (less dictionaries, use of sets, . . .)
- Added the `hide_bundle_services()` to the service registry. It is by the framework to hide the services of a stopping bundle from `get_service_reference` methods, and before those services will be unregistered.
- Removed the deprecated `ServiceEvent.get_type()` method

iPOPO

- Optimization of StoredInstance (handlers, use of sets, . . .)

HTTP Service

- Added a `is_header_set()` method to the HTTPServletResponse bean.
- Response headers are now sent on `end_headers()`, not on `set_header()`, to avoid duplicate headers.
- The request queue size of the basic HTTP server can now be set as a component property (`pelix.http.request_queue_size`)

Remote Services

- Added support for keyword arguments in most of remote services transports (all except XML-RPC)
- Added support for `pelix.remote.export.only` and `pelix.remote.export.none` service properties. `pelix.remote.export.only` tells the exporter to export the given specifications only, while `pelix.remote.export.none` forbids the export of the service.

Shell

- The `pelix.shell.console` module can now be run as a main script
- Added the *report* shell command
- Added the name of *varargs* in the signature of commands
- Corrected the signature shown in the help description for static methods
- Corrected the *thread* and *threads* shell commands for Pypy

Utilities

- Updated the MQTT client to follow the new API of Eclipse Paho MQTT Client

Tests

- Travis-CI: Added Python 3.5 and Pypy3 targets
- Better configuration of coverage
- Added tests for the remote shell
- Added tests for the MQTT client and for MQTT-RPC

5.2.5 iPOPO 0.6.2

Release Date 2015-06-17

iPOPO

- The properties of a component can be updated when calling the `retry_erroneous()` method. This allows to modify the configuration of a component before trying to validate it again (HTTP port, . . .).
- The `get_instance_details()` dictionary now always contains a *filter* entry for each of the component requirement description, even if not filter has been set.

HTTP Service

- Protection of the `ServletRequest.read_data()` method against empty or invalid `Content-Length` headers

Shell

- The `ipopo.retry` shell command accepts properties to be reconfigure the instance before trying to validate it again.
- The bundle commands (*start*, *stop*, *update*, *uninstall*) now print the name of the bundle along with its ID.
- The `threads` and `threads` shell commands now accept a stack depth limit argument.

5.2.6 iPOPO 0.6.1

Release Date 2015-04-20

iPOPO

- The stack trace of the exception that caused a component to be in the `ERRONEOUS` state is now kept, as a string. It can be seen through the `instance` shell command.

Shell

- The command parser has been separated from the shell core service. This allows to create custom shells without giving access to Pelix administration commands.
- Added `cd` and `pwd` shell commands, which allow changing the working directory of the framework and printing the current one.

- Corrected the encoding of the shell output string, to avoid exceptions when printing special characters.

Remote Services

- Corrected a bug where an imported service with the same endpoint name as an exported service could be exported after the unregistration of the latter.

5.2.7 iPOPO 0.6.0

Release Date 2015-03-12

Project

- The support of Python 2.6 has been removed

Utilities

- The XMPP bot class now supports anonymous connections using SSL or StartTLS. This is a workaround for issue 351 of SleekXMPP.

5.2.8 iPOPO 0.5.9

Release Date 2015-02-18

Project

- iPOPO now works with IronPython (tested inside Unity 3D)

iPOPO

- Components raising an error during validation goes in the `ERRONEOUS` state, instead of going back to `INVALID`. This avoids trying to validate them automatically.
- The `retry_erroneous()` method of the iPOPO service and the `retry` shell command allows to retry the validation of an `ERRONEOUS` component.
- The `@SingletonFactory` decorator can replace the `@ComponentFactory` one. It ensures that only one component of this factory can be instantiated at a time.
- The `@Temporal` requirement decorator allows to require a service and to wait a given amount of time for its replacement before invalidating the component or while using the requirement.
- `@RequiresBest` ensures that it is always the service with the best ranking that is injected in the component.
- The `@PostRegistration` and `@PreUnregistration` callbacks allows the component to be notified right after one of its services has been registered or will be unregistered.

HTTP Service

- The generated 404 page shows the list of registered servlets paths.
- The 404 and 500 error pages can be customized by a hook service.
- The default binding address is back to "0.0.0.0" instead of "localhost" (for those who used the development version).

Utilities

- The `ThreadPool` class is now a cached thread pool. It now has a minimum and maximum number of threads: only the required threads are alive. A thread waits for a task during 60 seconds (by default) before stopping.

5.2.9 iPOPO 0.5.8

Release Date 2014-10-13

Framework

- `FrameworkFactory.delete_framework()` can be called with `None` or without argument. This simplifies the clean up afters tests, etc.
- The list returned by `Framework.get_bundles()` is always sorted by bundle ID.

iPOPO

- Added the `immediate_rebind` option to the `@Requires` decorator. This indicates iPOPO to not invalidate then re-validate a component if a service can replace an unbound required one. This option only applies to non-optional, non-aggregate requirements.

Shell

- The I/O handler is now part of a `ShellSession` bean. The latter has the same API as the I/O handler so there is no need to update existing commands. I/O Handler write methods are now synchronized.
- The shell supports variables as arguments, *e.g.* `echo $var`. See string.Template for more information. The Template used in Pelix Shell allows `.` (dot) in names.
- A special variable `$?` stores the result of the last command which returned a result, *i.e.* anything but `None` or `False`.
- Added *set* and *unset* commands to work with variables
- Added the *run* command to execute a script file.
- Added protection against `AttributeError` in *threads* and *thread*

5.2.10 iPOPO 0.5.7

Release Date 2014-09-18

Project

- Code review to be more PEP-8 compliant
- jsonrpclib-pelix is now an install requirement (instead of an optional one)

Framework

- Forget about previous global members when calling `Bundle.update()`. This ensures to have a fresh dictionary of members after a bundle update
- Removed `from pelix.constants import *` in `pelix.framework`: only use `pelix.constants` to access constants

Remote Services

- Added support for endpoint name reuse
- Added support for synonyms: specifications that can be used on the remote side, or which describe a specification of another language (*e.g.* a Java interface)
- Added support for a `pelix.remote.export.reject` service property: the specifications it contains won't be exported, event if indicated in `service.exported.interfaces`.
- **JABSORB-RPC:**
  - Use the common `dispatch()` method, like JSON-RPC
- **MQTT(-RPC):**
  - Explicitly stop the reading loop when the MQTT client is disconnecting
  - Handle unknown correlation ID

Shell

- Added a `loglevel` shell command, to update the log level of any logger
- Added a `--verbose` argument to the shell console script
- Remote shell module can be ran as a script

HTTP Service

- Remove double-slashes when looking for a servlet

XMPP

- Added base classes to write a XMPP client based on SleekXMPP
- Added a XMPP shell interface, to control Pelix/iPOPO from XMPP

Miscellaneous

- Added an IPv6 utility module, to setup double-stack and to avoids missing constants bugs in Windows versions of Python
- Added a `EventData` class: it acts like `Event`, but it allows to store a data when setting the event, or to raise an exception in all callers of `wait()`
- Added a `CountdownEvent` class, an `Event` which is set until a given number of calls to `step()` is reached
- `threading.Future` class now supports a callback methods, to avoid to actively wait for a result.

5.2.11 iPOPO 0.5.6

Release Date 2014-04-28

Project

- Added samples to the project repository
- Removed the static website from the repository
- Added the project to Coveralls
- Increased code coverage

Framework

- Added a `@BundleActivator` decorator, to define the bundle activator class. The `activator` module variable should be replaced by this decorator.
- Renamed specifications constants: from `XXX_SPEC` to `SERVICE_XXX`

iPOPO

- Added a *waiting list* service: instantiates components as soon as the iPOPO service and the component factory are registered
- Added `@RequiresMap` handler
- Added an `if_valid` parameter to binding callbacks decorators: `@Bind`, `@Update`, `@Unbind`, `@BindField`, `@UpdateField`, `@UnbindField`. The decorated method will be called if and only if the component valid.
- The `get_factory_context()` from `decorators` becomes public to ease the implementation of new decorators

Remote Services

- **Large rewriting of Remote Service core modules**
  - Now using OSGi Remote Services properties
  - Added support for the OSGi EDEF file format (XML)
- Added an abstract class to easily write RPC implementations

- Added mDNS service discovery
- Added an MQTT discovery protocol
- Added an MQTT-RPC protocol, based on Node.js MQTT-RPC module
- Added a Jabsorb-RPC transport. Pelix can now use Java services and vice-versa, using:
  - Cohorte Remote Services
  - Eclipse ECF and the Jabsorb-RPC provider

Shell

- Enhanced completion with `readline`
- Enhanced commands help generation
- Added arguments to filter the output of `bl`, `sl`, `factories` and `instances`
- Corrected `prompt` when using `readline`
- Corrected `write_lines()` when not giving format arguments
- Added an `echo` command, to test string parsing

Services

- Added support for *managed service factories* in ConfigurationAdmin
- Added an EventAdmin-MQTT bridge: events from EventAdmin with an `event.propagate` property are published over MQTT
- Added an early version of an MQTT Client Factory service

Miscellaneous

- **Added a `misc` package, with utility modules and bundles:**
  - `eventadmin_printer`: an EventAdmin handler that prints or logs the events it receives
  - `jabsorb`: converts dictionary from and to the Jabsorb-RPC format
  - `mqtt_client`: a wrapper for the Paho MQTT client, used in MQTT discovery and MQTT-RPC.

5.2.12 iPOPO 0.5.5

Release Date 2013-11-15

Project

The license of the iPOPO project is now the Apache Software License 2.0.

Framework

- `get_*_service_reference*()` methods have a default LDAP filter set to `None`. Only the service specification is required, event if set to `None`.
- Added a context `use_service(context, svc_ref)`, that allows to consume a service in a `with` block.

iPOPO

- Added the *Handler Factory* pattern: all instance handlers are created by their factory, called by iPOPO according to the handler IDs found in the factory context. This will simplify the creation of new handlers.

Services

- Added the `ConfigurationAdmin` service
- Added the `FileInstall` service

5.2.13 iPOPO 0.5.4

Release Date 2013-10-01

Project

- Global speedup replacing `list.append()` by `bisect.insort()`.
- Optimizations in handling services, components and LDAP filters.
- Some classes of Pelix framework and iPOPO core modules extracted to new modules.
- Fixed support of Python 2.6.
- Replaced Python 3 imports conditions by *try-except* blocks.

iPOPO

- `@Requires` accepts only one specification
- Added a context `use_ipopo(bundle_context)`, to simplify the usage of the iPOPO service, using the keyword `with`.
- `get_factory_details(name)` method now also returns the ID of the bundle provided the component factory, and the component instance properties.
- Protection of the unregistration of factories, as a component can kill another one of the factory during its invalidation.

Remote Services

- Protection of the unregistration loop during the invalidation of JSON-RPC and XML-RPC exporters.
- The *Dispatcher Servlet* now handles the *discovered* part of the discovery process. This simplifies the *Multicast Discovery* component and suppresses a socket bug/feature on BSD (including Mac OS).

Shell

- The help command now uses the `inspect` module to list the required and optional parameters.
- `IOHandler` now has a `prompt()` method to ask the user to enter a line. It replaces the `read()` method, which was to buggy.
- The `make_table()` method now accepts generators as parameters.
- Remote commands handling removed: `get_methods_names()` is not used anymore.

5.2.14 iPOPO 0.5.3

Release Date 2013-08-01

iPOPO

- New `get_factory_details(name)` method in the iPOPO service, acting like `get_instance_details(name)` but for factories. It returns a dictionary describing the given factory.
- New `factory` shell command, which describes a component factory: properties, requirements, provided services, . . .

HTTP Service

- Servlet exceptions are now both sent to the client and logged locally

Remote Services

- Data read from the servlets or sockets are now properly converted from bytes to string before being parsed (Python 3 compatibility).

Shell

- Exceptions are now printed using `str(ex)` instead of `ex.message` (Python 3 compatibility).
- The shell output is now flushed, both by the shell I/O handler and the text console. The remote console was already flushing its output. This allows to run the Pelix shell correctly inside Eclipse.

5.2.15 iPOPO 0.5.2

Release Date 2013-07-19

iPOPO

- An error is now logged if a class is manipulated twice. Decorators executed after the first manipulation, i.e. upon `@ComponentFactory()`, are ignored.
- Better handling of inherited and overridden methods: a decorated method can now be overridden in a child class, with the name, without warnings.

- Better error logs, with indication of the error source file and line

HTTP Service

- **New servlet binding parameters:**
  - `http.name`: Name of HTTP service. The name of component instance in the case of the basic implementation.
  - `http.extra`: Extra properties of the HTTP service. In the basic implementation, this the content of the `http.extra` property of the HTTP server component
- New method `accept_binding(path, params)` in servlets. This allows to refuse the binding with a server before to test the availability of the registration path, thus to avoid raising a meaningless exception.

Remote Services

- End points are stored according to their framework
- Added a method `lost_framework(uid)` in the registry of imported services, which unregisters all the services provided by the given framework.

Shell

- Shell *help* command now accepts a command name to print a specific documentation

5.2.16 iPOPO 0.5.1

Release Date 2013-07-05

Framework

- `Bundle.update()` now logs the SyntaxError exception that be raised in Python 3.

HTTP Service

- The HTTP service now supports the update of servlet services properties. A servlet service can now update its registration path property after having been bound to a HTTP service.
- A *500 server error* page containing an exception trace is now generated when a servlet fails.
- The `bound_to()` method of a servlet is called only after the HTTP service is ready to accept clients.

Shell

- The remote shell now provides a service, `pelix.shell.remote`, with a `get_access()` method that returns the *(host, port)* tuple where the remote shell is waiting for clients.
- Fixed the `threads` command that wasn't working on Python 3.

5.2.17 iPOPO 0.5

Release Date 2013-05-21

Framework

- `BundleContext.install_bundle()` now returns the `Bundle` object instead of the bundle ID. `BundleContext.get_bundle()` has been updated to accept both IDs and `Bundle` objects in order to keep a bit of compatibility
- `Framework.get_symbolic_name()` now returns `pelix.framework` instead of `org.psem2m.pelix`
- `ServiceEvent.get_type()` is renamed `get_kind()`. The other name is still available but is declared deprecated (a warning is logged on its first use).
- `BundleContext.install_visiting(path, visitor)`: visits the given path and installs the found modules if the visitor accepts them
- **`BundleContext.install_package(path)` (*experimental*):**
  - Installs all the modules found in the package at the given path
  - Based on `install_visiting()`

iPOPO

- Components with a `pelix.ipopo.auto_restart` property set to `True` are automatically re-instantiated after their bundle has been updated.

Services

- **Remote Services: use services of a distant Pelix instance**
  - Multicast discovery
  - XML-RPC transport (not fully usable)
  - JSON-RPC transport (based on a patched version of jsonrpclib)
- EventAdmin: send events (a)synchronously

Shell

- Shell command methods now take an `IOHandler` object in parameter instead of input and output file-like streams. This hides the compatibility tricks between Python 2 and 3 and simplifies the output formatting.

5.2.18 iPOPO 0.4

Release Date 2012-11-21

Framework

- New `create_framework()` utility method
- The framework has been refactored, allowing more efficient services and events handling

iPOPO

- A component can provide multiple services
- A service controller can be injected for each provided service, to activate or deactivate its registration
- Dependency injection and service providing mechanisms have been refactored, using a basic handler concept.

Services

- Added a HTTP service component, using the concept of *servlet*
- Added an extensible shell, interactive and remote, simplifying the usage of a framework instance

5.2.19 iPOPO 0.3

Release Date 2012-04-13

Packages have been renamed. As the project goes public, it may not have relations to isandlaTech projects anymore.

Previous name	New name
psem2m	pelix
psem2m.service.pelix	pelix.framework
psem2m.component	pelix.ipopo
psem2m.component.ipopo	pelix.ipopo.core

5.2.20 iPOPO 0.2

Release Date 2012-02-07

Version 0.2 is the first public release of the project, under the terms of the GPLv3 license.

5.2.21 iPOPO 0.1

Release Date 2012-01-20

The first version of the Pelix framework, with packages still named after the `python.injection` and PSEM2M (now named Cohorte) projects by isandlaTech (now named Cohorte Technologies).

Back then, Pelix (bundles and services) was the most advanced part of the project, iPOPO was only an extension of it to handle basic components.

5.2.22 python.injections

Release Date 2011-12-20

The proof-of-concept package trying to mimic the iPOJO framework in Python 2.6. It only supported basic injections described by decorators.

5.3 License

iPOPO is licensed under the terms of the Apache Software License 2.0. All contributions must comply with this license.

5.3.1 File Header

This snippet is added to the module-level documentation:

```
Copyright 2017 Thomas Calmant

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

5.3.2 License Full Text

Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   “License” shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   “Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   “Legal Entity” shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, “control” means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   “You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this License.

Python Module Index

Index