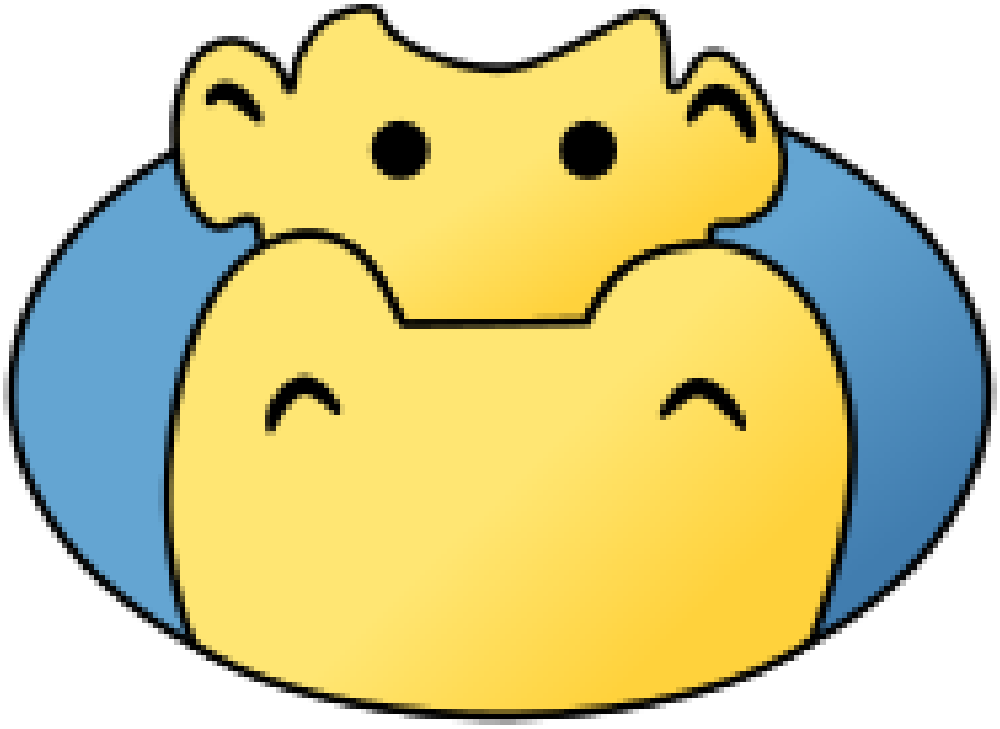
iPOPO Documentation

Release 1.0.0

Thomas Calmant

Jan 19, 2020

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 *Quick-start* 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.

iPOPO is released under the terms of the Apache Software License 2.0. It depends on a fork of jsonrpclib, called jsonrpclib-pelix. The documentation of this library is available on GitHub.

CHAPTER 1

About this documentation

The previous documentation was provided as a wiki, which has been shut down for various reasons. A copy of the previous content is available in the convert_doc branch, even though it's starting to age. The documentation is now hosted on 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.

If you have any question which hasn't been answered in the documentation, please ask on the users' mailing list.

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

CHAPTER 2

User's Guide

This chapter details how to install and use iPOPO.

2.1 Foreword

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

2.1.1 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*.

By the way, I pronounce the name iPOPO the french way, *i.e.* /i.p.p/ (International Phonetic Alphabet). The english way, *i.e.* /a.p.p/, is the most commonly used by the users I had the chance to talk to.

2.1.2 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 developed 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.

As of 2018, the development of iPOPO is still active. iPOPO 1.0 will come out when some features, existing or currently in development, will have been completed, tested and polished.

2.1.3 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 instantiated 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 *Quick-start* or the *Tutorials*.

2.2 Installation

iPOPO strongly depends on only one external library, jsonrpclib-pelix, which provides some utility methods and is required to enable remote services based on JSON-RPC. It relies on other libraries for extended features, which are listed in the requirements file.

To install iPOPO, you will need Python 2.7, Python 3.4 or newer. iPOPO is constantly tested, using Tox and Travis-CI, on the following interpreters:

- Python 2.7
- Python 3.4, 3.5 and 3.6

Support for Python 2.6 has been dropped with iPOPO 0.6. The framework should run on Python 3.2 and 3.3 and also on Pypy, but this is not guaranteed. Any feedback on those platforms is welcome.

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

2.2.1 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
```

2.2.2 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.

Older Python 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, the `python` and `pip` commands you type in the shell should be those of your virtual environment. The shell prompt indicates the name of the virtual environment currently in use.

Now you can install iPOPO using `pip`. As you are in 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
```

2.2.3 Development version

If you want to work with the 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.

You can now continue to *Quick-start*

2.3 Quick-start

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.

2.3.1 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 Pelix 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: 14
$ start 14
Starting bundle 14 (pelix.shell.remote)...
$ bl
+----+----------------------------------------+--------+---------+
| ID |                  Name                  | State  | Version |
+====+========================================+========+=========+
| 0  | pelix.framework                        | ACTIVE | 1.0.0   |
+----+----------------------------------------+--------+---------+
...
+----+----------------------------------------+--------+---------+
| 14 | pelix.shell.remote                     | ACTIVE | 1.0.0   |
+----+----------------------------------------+--------+---------+
15 bundles installed
$ update 14
Updating bundle 14 (pelix.shell.remote)...
$ stop 14
Stopping bundle 14 (pelix.shell.remote)...
$ uninstall 14
Uninstalling bundle 14 (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       |
+----+--------------------------+-------------------------------------------------
↪+---------+
...
```

(continues on next page)

(continued from previous page)

```
16 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 an erroneous component

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

```
$ install pelix.shell.remote
Bundle ID: 15
$ start 15
Starting bundle 15 (pelix.shell.remote)...
$ factories
+------------------------------+----------------------------------------+
|           Factory            |                 Bundle                 |
+==============================+========================================+
| ipopo-remote-shell-factory   | Bundle(ID=15, 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 **
```

(continues on next page)

(continued from previous page)

```
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`.

2.3.2 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.

```
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: 14
$ start 14
Starting bundle 14 (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`).

```
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
```

(continues on next page)

(continued from previous page)

```
@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: 15
$ start 15
Starting bundle 15 (consumer)...
Component validated, calling the service...
Hello, World !
Done.
$ update 14
Updating bundle 14 (provider)...
Component invalidated, the service is gone
Component validated, calling the service...
Hello, World !
Done.
$ uninstall 14
Uninstalling bundle 14 (provider)...
Component invalidated, the service is gone
```

2.3.3 Hello from somewhere else!

This section reuses the bundles written in the *Hello World* sample and starts them into two distinct frameworks. The consumer will use the service provided from the other framework.

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: this will be the job of the 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 be instantiated to answer to those requests:

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

(continues on next page)

(continued from previous page)

```
$ install pelix.remote.registry
Bundle ID: 14
$ start 14
Starting bundle 14 (pelix.remote.registry)...
$ install pelix.remote.dispatcher
Bundle ID: 15
$ start 15
Starting bundle 15 (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: 16
$ start 16
Starting bundle 16 (pelix.http.basic)...
$ instantiate pelix.http.service.basic.factory httpd pelix.http.port=8000
INFO:httpd:Starting HTTP server: [0.0.0.0]:8000 ...
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 frameworks 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: 17
$ start 17
Starting bundle 17 (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 handling the export of services and one handling their 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: 18
$ start 18
Starting bundle 18 (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: 19
$ start 19
Starting bundle 19 (provider)...
```

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

```
$ install consumer
Bundle ID: 19
$ start 19
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.

You can now continue to the *Tutorials*

2.4 Tutorials

This section provides tutorials for various parts of iPOPO.

2.4.1 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.

```
#!/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.
```

(continues on next page)

(continued from previous page)

```
        """
        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):
        """
        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.

```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 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.
    """

    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.
        """
```

(continues on next page)

(continued from previous page)

```
        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.

```
#!/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

# 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

```

(continues on next page)

(continued from previous page)

```
    @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
    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]
```

(continues on next page)

(continued from previous page)

```
        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.

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

(continues on next page)

(continued from previous page)

```
        """
        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.
        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
```

(continues on next page)

(continued from previous page)

```
                # 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.

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

# Pelix framework module and utility methods
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()

```

(continues on next page)

(continued from previous page)

```
    # 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...
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:
```

(continues on next page)

(continued from previous page)

```
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
```

You can now go back to see other *Tutorials* or take a look at the *Reference Cards*.

2.4.2 RSA Remote Services between Python and Java

Authors Scott Lewis, Thomas Calmant

Introduction

This tutorial shows how to launch and use the sample application for OSGi R7 Remote Services Admin (RSA) between Python and Java. This sample shows how to use the *iPOPO RSA implementation* to export and/or import remote services from/to a OSGi/Java process to a Python iPOPO process.

Requirements

This sample requires Python 3 and launching the Java sample prior to proceeding with Starting the Python Sample below.

It is also required to have installed the osgiservicebridge package (using `pip` or `easy_install`) before continuing.

This ECF tutorial page describes how to launch the Java-side sample. One can start via Bndtools project template, or start via Apache Karaf.

Once the Java sample has been successfully started, proceed below.

Note: You may skip this part if you executed the Java sample following the instructions above.

It is recommended to read the whole Java part of the tutorial before continuing. However, for those who just want to see things working, here are the commands to execute the Java sample:

```
# Download Karaf from https://karaf.apache.org/download.html
wget http://archive.apache.org/dist/karaf/4.2.1/apache-karaf-4.2.1.tar.gz
tar xzf apache-karaf-4.2.1.tar.gz
cd apache-karaf-4.2.1.tar.gz
./bin/karaf
```

Once inside karaf, run the following commands:

```
feature:repo-add ecf
feature:install -v ecf-rs-examples-python.java-hello
```

This will add the ECF repository to the Karaf framework, then install and start all the necessary Java bundles.

Wait for a XML representation of an endpoint (in EDEF format) to be printed out: the Java side of the tutorial is now ready.

Starting the Python Sample

In the iPOPO project root directory, start the top-level script for this sample:

```
$ python samples/run_rsa_py4java.py
```

This should produce output to the console like the following:

```
** Pelix Shell prompt **
Python IHello service consumer received sync response: Java says: Hi PythonSync, nice␣
↪to see you
done with sayHelloAsync method
done with sayHelloPromise method
async response: JavaAsync says: Hi PythonAsync, nice to see you
promise response: JavaPromise says: Hi PythonPromise, nice to see you
```

This output indicates that

1. The Python process connected to the Java process using the Py4j distribution provider
2. RSA discovered and imported the Java-exported `HelloImpl` service
3. RSA created a Python proxy for the `IHello` service instance hosted from Java
4. iPOPO injected the `IHello` proxy into the sample consumer by setting the `self._helloservice` requirement to the `IHello` proxy
5. iPOPO then called the `_validate` method of the `RemoteHelloConsumer` class (in `samples/rsa/helloconsumer.py`)

Here is the source code of the `helloconsumer.py` file, from the `samples/rsa` folder:

```
from pelix.ipopo.decorators import (
    ComponentFactory,
    Instantiate,
    Requires,
    Validate,
)


@ComponentFactory("remote-hello-consumer-factory")
# The '(service.imported=*)' filter only allows remote services to be injected
@Requires(
    "_helloservice",
    "org.eclipse.ecf.examples.hello.IHello",
    False,
    False,
    "(service.imported=*)",
    False,
)
@Instantiate("remote-hello-consumer")
class RemoteHelloConsumer(object):
    def __init__(self):
        self._helloservice = None
```

(continues on next page)

(continued from previous page)

```
      self._name = "Python"
      self._msg = "Hello Java"

    @Validate
    def _validate(self, bundle_context):
        # call it!
      resp = self._helloservice.sayHello(self._name + "Sync", self._msg)
        print(
            self._name, "IHello service consumer received sync response:", resp
      )

        # call sayHelloAsync which returns Future and we add lambda to print
        # the result when done
        self._helloservice.sayHelloAsync(
            self._name + "Async", self._msg
      ).add_done_callback(lambda f: print("async response:", f.result()))
        print("done with sayHelloAsync method")

        # call sayHelloAsync which returns Future and we add lambda to print
        # the result when done
        self._helloservice.sayHelloPromise(
            self._name + "Promise", self._msg
      ).add_done_callback(lambda f: print("promise response:", f.result()))
        print("done with sayHelloPromise method")
```

When the `_validate` method is called by iPOPO, it calls the `self._helloservice.sayHello` synchronous method and prints out the result (`resp`) to the console:

```
@Validate
def _validate(self, bundle_context):
    # call it!
    resp = self._helloservice.sayHello(self._name + "Sync", self._msg)
    print(
        self._name, "IHello service consumer received sync response:", resp
    )
```

The print in the code above is responsible for the console output:

```
Python IHello service consumer received sync response:
Java says: Hi PythonSync, nice to see you
```

Then the `sayHelloAsync` method is called:

```
 self._helloservice.sayHelloAsync(
   self._name + "Async", self._msg
).add_done_callback(lambda f: print("async response:", f.result()))
 print("done with sayHelloAsync method")
```

The print is responsible for the console output:

```
done with sayHelloAsync method
```

Then the `sayHelloPromise` method is called:

```
 self._helloservice.sayHelloPromise(
   self._name + "Promise", self._msg
```

(continues on next page)

(continued from previous page)

```
).add_done_callback(lambda f: print("promise response:", f.result()))
 print("done with sayHelloPromise method")
```

Resulting in the console output:

```
done with sayHelloPromise method
```

Note that the async response and promise response are received after the `print('done with sayHelloPromise')` statement. Once the remote (Java) call is completed, the lambda expression callback is executed via `Future.add_done_callback`. This results in the output ordering of:

```
Python IHello service consumer received sync response: Java says: Hi PythonSync, nice␣
↪to see you
done with sayHelloAsync method
done with sayHelloPromise method
async response: JavaAsync says: Hi PythonAsync, nice to see you
promise response: JavaPromise says: Hi PythonPromise, nice to see you
```

The 'done…' prints out prior to the execution of the print in the lambda expression callback passed to Future.add_done_callback.

Note that at the same time as the Python-side console output above, in the Java console this will appear:

```
Java.sayHello called by PythonSync with message: 'Hello Java'
Java.sayHelloAsync called by PythonAsync with message: 'Hello Java'
Java.sayHelloPromise called by PythonPromise with message: 'Hello Java'
```

This is the output from the Java `HelloImpl` implementation code:

```
public String sayHello(String from, String message) {
    System.out.println("Java.sayHello called by "+from+" with message: '"+message+"'
↪");
    return "Java says: Hi "+from + ", nice to see you";
}
```

Exporting a Hello implementation from Python to Java

In the iPOPO console, give the following command to register and export a `IHello` service instance from Python impl to Java consumer.

```
$ start samples.rsa.helloimpl_py4j
```

This should result in the Python console output

```
$ start samples.rsa.helloimpl_py4j
Bundle ID: 18
Starting bundle 18 (samples.rsa.helloimpl_py4j)...
Python.sayHello called by: Java with message: 'Hello Python'
Python.sayHelloAsync called by: JavaAsync with message: 'Howdy Python'
Python.sayHelloPromise called by: JavaPromise with message: 'Howdy Python'
```

Here is the Python hello implementation from `samples/helloimpl_py4j.py`:

```
from pelix.ipopo.decorators import Instantiate, ComponentFactory, Provides
from samples.rsa.helloimpl import HelloImpl


@ComponentFactory("helloimpl-py4j-factory")
# Provides IHello interface as specified by Java interface.
@Provides("org.eclipse.ecf.examples.hello.IHello")
# See https://github.com/ECF/Py4j-RemoteServicesProvider/blob/master/examples/org.
↪eclipse.ecf.examples.hello/src/org/eclipse/ecf/examples/hello/IHello.java
@Instantiate(
    "helloimpl-py4j",
    {
        "service.exported.interfaces": "*",  # Required for export
        # Required to use py4j python provider for export
        "service.exported.configs": "ecf.py4j.host.python",
        # Required to use osgi.async intent
        "service.intents": ["osgi.async"],
        "osgi.basic.timeout": 30000,
    },
)  # Timeout associated with remote calls (in ms)
class Py4jHelloImpl(HelloImpl):
    """
    All method implementations handled by HelloImpl super-class.

    See samples.rsa.helloimpl module.
    """
    pass
```

and here is the `HelloImpl` super-class from `samples/helloimpl.py`:

```
class HelloImpl(object):
    """
    Implementation of Java org.eclipse.ecf.examples.hello.IHello service
    interface.
    This interface declares on normal/synchronous method ('sayHello') and two
    async methods as defined by the OSGi Remote Services osgi.async intent.

    Note that the service.intents property above includes the 'osgi.async'
    intent. It also declares a property 'osgi.basic.timeout' which will be used
    to assure that the remote methods timeout after the given number of
    milliseconds.

    See the OSGi Remote Services specification at:
    https://osgi.org/specification/osgi.cmpn/7.0.0/service.remoteservices.html

    The specification defines the standard properties given above.
    """

    def sayHello(self, name="Not given", message="nothing"):
        """
        Synchronous implementation of IHello.sayHello synchronous method.
        The remote calling thread will be blocked until this is executed and
        responds.
        """
        print(
            "Python.sayHello called by: {0} with message: '{1}'".format(
                name, message
```

(continues on next page)

(continued from previous page)

```
            )
        )
        return "PythonSync says: Howdy {0} that's a nice runtime you got there".format(
            name
        )

    def sayHelloAsync(self, name="Not given", message="nothing"):
        """
        Implementation of IHello.sayHelloAsync.
        This method will be executed via some thread, and the remote caller
        will not block.
        This method should return either a String result (since the return type
        of IHello.sayHelloAsync is CompletableFuture<String>, OR a Future that
        returns a python string.  In this case, it returns the string directly.
        """
        print(
            "Python.sayHelloAsync called by: {0} with message: '{1}'".format(
                  name, message
            )
        )
        return "PythonAsync says: Howdy {0} that's a nice runtime you got there".format(
            name
        )

    def sayHelloPromise(self, name="Not given", message="nothing"):
        """
        Implementation of IHello.sayHelloPromise.
        This method will be executed via some thread, and the remote caller
        will not block.
        """
        print(
            "Python.sayHelloPromise called by: {0} with message: '{1}'".format(
                  name, message
            )
        )
        return "PythonPromise says: Howdy {0} that's a nice runtime you got there".
↪format(
            name
        )
```

You can now go back to see other *Tutorials* or take a look at the *Reference Cards*.

2.4.3 RSA Remote Services using XmlRpc transport

Authors Scott Lewis, Thomas Calmant

Introduction

This tutorial shows how to create and run a simple remote service using the XmlRpc provider. The XmlRpc distribution provider is one of several supported by the RSA Remote Services (OSGi R7-compliant) implementation.

Requirements

This tutorial sample requires Python 3.4+ or Python 2.7, and version 0.8.0+ of iPOPO.

Defining the Remote Service with a Python Class

We'll start by defining a Python *hello* service that can to be exported by RSA for remote access.

In the `sample.rsa` package is the `helloimpl_xmlrpc` module, containing the `XmlRpcHelloImpl` class

```
@ComponentFactory("helloimpl-xmlrpc-factory")
@Provides(
    "org.eclipse.ecf.examples.hello.IHello"
)
@Instantiate(
    "helloimpl-xmlrpc",
    {
        # uncomment to automatically export upon creation
        # "service.exported.interfaces":"*",
        "osgi.basic.timeout": 60000,
    },
)
class XmlRpcHelloImpl(HelloImpl):
    pass
```

The `XmlRpcHelloImpl` class has no body/implementation as it inherits its implementation from the `HelloImpl` class, which we will discuss in a moment.

The important parts of this class declaration for remote services are the `@Provides` class decorator and the commented-out **service.exported.interfaces** and **osgi.basic.timeout** properties in the `@Instantiate` decorator.

The `@Provides` class decorator gives the **name** of the service specification provided by this instance. This is the name that both local and remote consumers use to lookup this service, even if it's local-only (*i.e.* not a remote service). In this case, since the original `IHello` interface is a java interface class, the fully-qualified name of the interface class is used. For an example of JavaPython remote services see this tutorial.

For Python-only remote services it's not really necessary for this service specification be the name of a Java class, any unique String could have been used.

The **osgi.basic.timeout** is an optional property that gives a maximum time (in milliseconds) that the consumer will wait for a response before timing out.

The **service.exported.interfaces** property is a required property for remote service export. If one wants to have a remote service exported immediately upon instantiation and registration as an iPOPO service, this property can be set to value `*` which means to export all service interfaces.

The **service.exported.interfaces** property is commented out so that it is **not** exported immediately upon instantiation and registration. Instead, for this tutorial the export is performed via iPOPO console commands. If these comments were to be removed, the RSA impl will export this service as soon as it is instantiated and registered, making it unnecessary to explicitly export the service as shown in *Exporting the XmlRpcHelloImpl as a Remote Service* section below.

The HelloImpl Implementation

The `XmlRpcHelloImpl` class delegates all the actual implementation to the `HelloImpl` class, which has the code for the methods defined for the *"org.eclipse.ecf.examples.hello.IHello"* service specification name, with the main method `sayHello`:

```
class HelloImpl(object):
    def sayHello(self, name='Not given', message='nothing'):
        print(
            "Python.sayHello called by: {0} with message: '{1}'".format(
```

(continues on next page)

(continued from previous page)

```
                name, message))
        return "PythonSync says: Howdy {0} that's a nice runtime you got there".
↪format(
            name)
```

The `sayHello` method is invoked via a remote service consumer once the service has been exporting.

Exporting the XmlRpcHelloImpl as a Remote Service

Go to the pelix home directory and start the `run_rsa_xmlrpc.py` main program

```
ipopo-1.0.0$ python -m samples.run_rsa_xmlrpc
** Pelix Shell prompt **
$
```

To load the module and instantiate and register an `XmlRpcHelloImpl` instance type

```
$ start samples.rsa.helloimpl_xmlrpc
Bundle ID: 18
Starting bundle 18 (samples.rsa.helloimpl_xmlrpc)...
```

In your environment, bundle number might not be 18... that is fine.

If you list services using the `sl` console command you should see an instance of `IHello` service

```
$ sl org.eclipse.ecf.examples.hello.IHello
+----+-----------------------------------------+-----------------------------------
↪--------------+---------+
| ID |              Specifications             |                      Bundle
↪              | Ranking |
+====+=========================================+===================================
| 20 | ['org.eclipse.ecf.examples.hello.IHello'] | Bundle(ID=18, Name=samples.rsa.
↪helloimpl_xmlrpc) | 0       |
+----+-----------------------------------------+-----------------------------------
↪--------------+---------+
1 services registered
```

The service ID (20 in this case) may not be the same in your environment... again that is ok... but **make a note of what the service ID is**.

To export this service instance as remote service and make it available for remote access, use the `exportservice` command in the pelix console, giving the number (20 from above) of the service to export:

```
$ exportservice 20        # use the service id for the org.eclipse.ecf.examples.hello.
↪IHello service if not 20
Service=ServiceReference(ID=20, Bundle=18, Specs=['org.eclipse.ecf.examples.hello.
↪IHello']) exported by 1 providers. EDEF written to file=edef.xml
$
```

This means that the service has been successfully exported. To see this use the `listexports` console command:

```
$ listexports
+--------------------------------------+------------------------------+------------+
|             Endpoint ID              |         Container ID         | Service ID |
+======================================+==============================+============+
| b96927ad-1d00-45ad-848a-716d6cde8443 | http://127.0.0.1:8181/xml-rpc | 20         |
```

(continues on next page)

(continued from previous page)

```
+-------------------------------------+------------------------------+------------+
$ listexports b96927ad-1d00-45ad-848a-716d6cde8443
Endpoint description for endpoint.id=b96927ad-1d00-45ad-848a-716d6cde8443:
<?xml version='1.0' encoding='cp1252'?>
<endpoint-descriptions xmlns="http://www.osgi.org/xmlns/rsa/v1.0.0">
       <endpoint-description>
               <property name="objectClass" value-type="String">
                       <array>
                               <value>org.eclipse.ecf.examples.hello.IHello</value>
                       </array>
               </property>
               <property name="remote.configs.supported" value-type="String">
                       <array>
                               <value>ecf.xmlrpc.server</value>
                       </array>
               </property>
               <property name="service.imported.configs" value-type="String">
                       <array>
                               <value>ecf.xmlrpc.server</value>
                       </array>
               </property>
               <property name="remote.intents.supported" value-type="String">
                       <array>
                               <value>osgi.basic</value>
                               <value>osgi.async</value>
                       </array>
               </property>
               <property name="service.intents" value-type="String">
                       <array>
                               <value>osgi.async</value>
                       </array>
               </property>
               <property name="endpoint.service.id" value="20" value-type="Long">
                       </property>
               <property name="service.id" value="20" value-type="Long">
                       </property>
               <property name="endpoint.framework.uuid" value="4d541077-ee2a-4d68-
↪85f5-be529f89bec0" value-type="String">
                       </property>
               <property name="endpoint.id" value="b96927ad-1d00-45ad-848a-
↪716d6cde8443" value-type="String">
                       </property>
               <property name="service.imported" value="true" value-type="String">
                       </property>
               <property name="ecf.endpoint.id" value="http://127.0.0.1:8181/xml-rpc"␣
↪value-type="String">
                       </property>
               <property name="ecf.endpoint.id.ns" value="ecf.namespace.xmlrpc" value-
↪type="String">
                       </property>
               <property name="ecf.rsvc.id" value="3" value-type="Long">
                       </property>
               <property name="ecf.endpoint.ts" value="1534119904514" value-type="Long
↪">
                       </property>
               <property name="osgi.basic.timeout" value="60000" value-type="Long">
                       </property>
```

(continues on next page)

(continued from previous page)

```
      </endpoint-description>
</endpoint-descriptions>
$
```

Note that `listexports` produced a small table with **Endpoint ID**, **Container ID**, and **Service ID** columns. As shown above, if the Endpoint ID is copied and used in listexports, it will then print out the endpoint description (XML) for the newly-created endpoint.

Also as indicated in the `exportservice` command output, a file *edef.xml* has also been written to the filesystem containing the endpoint description XML known as EDEF). EDEF is a standardized XML format that gives all of the remote service meta-data required for a consumer to import an endpoint. The *edef.xml* file will contain the same XML printed to the console via the `listexports b96927ad-1d00-45ad-848a-716d6cde8443` console command.

Importing the XmlRpcHelloImpl Remote Service

For a consumer to use this remote service, another python process should be started using the same command:

```
ipopo-1.0.0$ python -m samples.run_rsa_xmlrpc
** Pelix Shell prompt **
$
```

If you have started this second python process from the same location, all that's necessary to trigger the import of the remote service, and have a consumer sample start to call it's methods is to use the `importservice` console command:

```
$ importservice
Imported 1 endpoints from EDEF file=edef.xml
Python IHello service consumer received sync response: PythonSync says: Howdy␣
↪PythonSync that's a nice runtime you got there
done with sayHelloAsync method
done with sayHelloPromise method
Proxy service=ServiceReference(ID=21, Bundle=7, Specs=['org.eclipse.ecf.examples.
↪hello.IHello']) imported. rsid=http://127.0.0.1:8181/xml-rpc:3
$ async response: PythonAsync says: Howdy PythonAsync that's a nice runtime you got␣
↪there
promise response: PythonPromise says: Howdy PythonPromise that's a nice runtime you␣
↪got there
```

This indicates that the remote service was imported, and the methods on the remote service were called by the consumer.

Here is the code for the consumer (also in `samples/rsa/helloconsumer_xmlrpc.py`)

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

from concurrent.futures import ThreadPoolExecutor

@ComponentFactory("remote-hello-consumer-factory")
# The '(service.imported=*)' filter only allows remote services to be injected
@Requires("_helloservice", "org.eclipse.ecf.examples.hello.IHello",
          False, False, "(service.imported=*)", False)
@Instantiate("remote-hello-consumer")
class RemoteHelloConsumer(object):

```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        self._helloservice = None
        self._name = 'Python'
        self._msg = 'Hello Java'
        self._executor = ThreadPoolExecutor()

    @Validate
    def _validate(self, bundle_context):
        # call it!
        resp = self._helloservice.sayHello(self._name + 'Sync', self._msg)
        print(
            "{0} IHello service consumer received sync response: {1}".format(
                self._name,
                resp))
        # call sayHelloAsync which returns Future and we add lambda to print
        # the result when done
        self._executor.submit(
            self._helloservice.sayHelloAsync,
            self._name + 'Async',
            self._msg).add_done_callback(
            lambda f: print(
                'async response: {0}'.format(
                    f.result())))
        print("done with sayHelloAsync method")
        # call sayHelloAsync which returns Future and we add lambda to print
        # the result when done
        self._executor.submit(
            self._helloservice.sayHelloPromise,
            self._name + 'Promise',
            self._msg).add_done_callback(
            lambda f: print(
                'promise response: {0}'.format(
                    f.result())))
        print("done with sayHelloPromise method")
```

For having this remote service injected, the important part of things is the `@Requires` decorator

```
@Requires("_helloservice", "org.eclipse.ecf.examples.hello.IHello",
          False, False, "(service.imported=*)", False)
```

This gives the specification name required **org.eclipse.ecf.examples.hello.IHello**, and it also gives an OSGi filter

```
"(service.imported=*)"
```

As per the Remote Service spec this requires that the `IHello` service is a remote service, as all proxies must have the **service.imported** property set, indicating that it was imported.

When `importservice` is executed the RSA implementation does the following:

1. Reads the edef.xml from filesystem (i.e. 'discovers the service')
2. Create a local proxy for the remote service using the edef.xml file
3. The proxy is injected by iPOPO into the `RemoteHelloConsumer._helloservice` member
4. The `_activated` method is called by iPOPO, which uses the `self._helloservice` proxy to send the method calls to the remote service, using HTTP and XML-RPC to serialize the `sayHello` method arguments, send the request via HTTP, get the return value back, and print the return value to the consumer's console.

Note that with Export, rather than using the console's `exportservice` command, it may be invoked programmatically, or automatically by the topology manager (for example upon service registration). For Import, the `importservice` command may also be invoked automatically, or via remote service discovery (e.g. etcd, zookeeper, zeroconf, custom, etc). The use of the console commands in this example was to demonstrate the dynamics and flexibility provided by the OSGi R7-compliant RSA implementation.

Exporting Automatically upon Service Registration

To export automatically upon service registration, all that need be done is to un-comment the setting the **service.exported.interfaces** property in the `Instantiate` decorator:

```
@ComponentFactory("helloimpl-xmlrpc-factory")
@Provides(
    "org.eclipse.ecf.examples.hello.IHello"
)
@Instantiate(
    "helloimpl-xmlrpc",
    {
        "service.exported.interfaces": "*",
        "osgi.basic.timeout": 60000,
    },
)
class XmlRpcHelloImpl(HelloImpl):
    pass
```

Unlike in the example above, when this service is instantiated and registered, it will also be automatically exported, making unnecessary to use the `exportservice` command.

Using Etcd Discovery

Rather than importing remote services manually via the `importservice` command, it's also possible to import using supported network discovery protocols. One discovery mechanism used in systems like kubernetes is etcd, and there is an etcd discovery provider available in the `pelix.rsa.providers.discovery.discovery_etcd` module.

This is the list of bundles included in the `samples.run_rsa_etcd_xmlrpc` program:

```
bundles = ['pelix.ipopo.core',
           'pelix.shell.core',
           'pelix.shell.ipopo',
           'pelix.shell.console',
           'pelix.rsa.remoteserviceadmin',  # RSA implementation
           'pelix.http.basic',  # httpservice
           # xmlrpc distribution provider (opt)
           'pelix.rsa.providers.distribution.xmlrpc',
           # etcd discovery provider (opt)
           'pelix.rsa.providers.discovery.discovery_etcd',
           # basic topology manager (opt)
           'pelix.rsa.topologymanagers.basic',
           'pelix.rsa.shell',  # RSA shell commands (opt)
           'samples.rsa.helloconsumer_xmlrpc']  # Example helloconsumer.  Only uses␣
↪remote proxies
```

Note the presence of the etcd discovery provider: `pelix.rsa.providers.discovery.discovery_etcd`

To start a consumer with etcd discovery run the `samples.run_rsa_etcd_xmlrpc` program:

```
$ python -m samples.run_rsa_etcd_xmlrpc
** Pelix Shell prompt **
$ start samples.rsa.helloimpl_xmlrpc
Bundle ID: 19
Starting bundle 19 (samples.rsa.helloimpl_xmlrpc)...
$ sl org.eclipse.ecf.examples.hello.IHello
+----+-----------------------------------------+-----------------------------------
↪--------------+---------+
| ID |              Specifications             |                      Bundle
↪               | Ranking |
+====+=========================================+===================================
| 21 | ['org.eclipse.ecf.examples.hello.IHello'] | Bundle(ID=19, Name=samples.rsa.
↪helloimpl_xmlrpc) | 0       |
+----+-----------------------------------------+-----------------------------------
↪--------------+---------+
1 services registered
$ exportservice 21
Service=ServiceReference(ID=21, Bundle=19, Specs=['org.eclipse.ecf.examples.hello.
↪IHello']) exported by 1 providers. EDEF written to file=edef.xml
$ lexps
+--------------------------------------+------------------------------+------------+
|             Endpoint ID              |         Container ID         | Service ID |
+======================================+==============================+============+
| 0b5a6bf1-494e-41ef-861c-4c302ae75141 | http://127.0.0.1:8181/xml-rpc | 21         |
+--------------------------------------+------------------------------+------------+
$
```

Then start a consumer process

```
$ python -m samples.run_rsa_etcd_xmlrpc
** Pelix Shell prompt **
$ Python IHello service consumer received sync response: PythonSync says: Howdy␣
↪PythonSync that's a nice runtime you got there
done with sayHelloAsync method
done with sayHelloPromise method
async response: PythonAsync says: Howdy PythonAsync that's a nice runtime you got␣
↪there
promise response: PythonPromise says: Howdy PythonPromise that's a nice runtime you␣
↪got there
```

This consumer uses etcd to discover the `IHello` remote service, a proxy is created and injected into the consumer (using the same consumer code shown above), and the consumer calls this proxy producing the text output above on the consumer and this output on the remote service implementation:

```
$ Python.sayHello called by: PythonSync with message: 'Hello Java'
Python.sayHelloAsync called by: PythonAsync with message: 'Hello Java'
Python.sayHelloPromise called by: PythonPromise with message: 'Hello Java'
```

The consumer discovered the `org.eclipse.ecf.examples.hello.IHello` service published via etcd discovery, injected it into the consumer and the consumer called the methods on the `IHello` remote service, producing output on both the consumer and the remote service implementation.

2.5 Reference Cards

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

2.5.1 Bundles

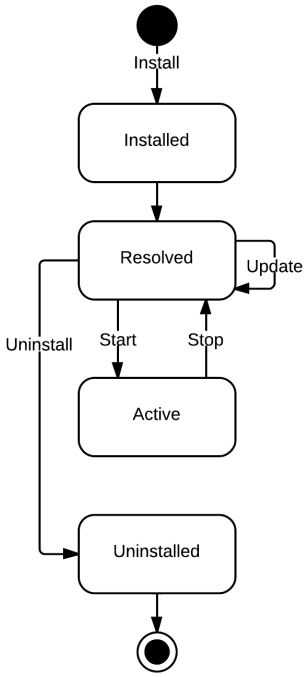
A bundle is a Python module installed using the Pelix `Framework` instance or a `BundleContext` object.

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 importlib.reload() method (or imp.reload() when not available)
- 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.

The framework is locked during transitions in bundles states, which means during the calls to *start()* and *stop()*. Therefore, it is heavily recommended to return fast from those methods. For example, it may be necessary to use threads to complete the initialization before registering services when the bundle starts. On the other hand, it is recommended to wait for all resources to be released before exiting the *stop()* , *e.g.* to wait for all threads started by the bundle to terminate.

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:

```
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 module member named `activator`, 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 and can be used until the latter is removed from the framework. It is not recommended to keep references to `BundleContext` objects as they can imply a stall reference to the bundle they describe. A bundle must use its context 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:

2.5.2 Services

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

The bundle that registers the service must keep track of 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, as it gives access and control over the life cycle of the service it represents. Finally, all services must be unregistered when their bundle is stopped.

A consumer can look for a service that matches a specification and a set of properties, using its `BundleContext`. 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, etc..

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

```
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:

```
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)
        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.

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.

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

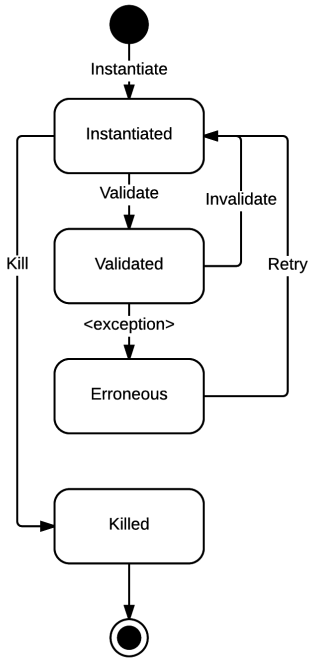
2.5.3 iPOPO Components

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
INSTANTIATED	The component has been instantiated. Its constructor has been called and the control methods have been injected
VALIDATED	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
ERRONEOUS	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.

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

2.5.4 A word on Python 3.7 Data classes

These indications have to be taken into account when using iPOPO decorators on data classes. They are also valid when using the dataclasses package for Python 3.6.

Important notes

- **All** fields of the Data Class **must** have a default value. This will let the `@dataclass` decorator generate an `__init__` method without explicit arguments, which is a requirement for iPOPO.
- If the `init=False` argument is given to `@dataclass`, it is necessary to implement your own `__init__`, defining all fields, otherwise generated methods like `__repr__` won't work.

Good to know

- Injected fields (`@Property`, `@Requires`, ...) will lose the default value given in the class definition, in favor to the ones given to the iPOPO decorators. This is due to the redefinition of the fields by those decorators. Other fields are not touched at all.
- The `@dataclass` decorator can be used before or after the iPOPO decorators

2.5.5 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.

Component properties

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
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

Requirements

Instance definition

Life-cycle events

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

Component state

When all its requirements are fulfilled, the component goes into the *VALID* state. During the transition, it is in *VALIDATING* state and the following decorators indicate which method must be called at that time. If the decorated method raises an exception, the component goes into the *ERRONEOUS* state.

When one of its requirements is missing, or when it is killed, the component goes into the *INVALID* state. During the transition, it is in *INVALIDATING* state and the following decorators indicate which method must be called at that time.

Exceptions raised by the decorated method are ignored.

Injections

Service state

2.5.6 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

---

Note: The `components` entry requires iPOPO to work. Therefore, the `pelix.ipopo.core` bundle must be declared in the `bundles` entry of the initial configuration file.

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"
```

(continues on next page)

(continued from previous page)

```
  ],
  "bundles": [
    "pelix.ipopo.core",
    "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"
      }
    }
  ]
}
```

Configuration override

The initial configuration can be split in multiple files in order to ease the specialisation of frameworks sharing a common base configuration. This is explained in the following *File Lookup* section.

Sometimes, it can be necessary to redefine some entries, *e.g* in order to change a set of components but keeping the bundles to be started. 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.ipopo.core",
     "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

An `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 newly loaded configuration with the current state of the handler.

Finally, after having updated a configuration, the `InitFileHandler` will remove the duplicated elements of the Python path and bundles entries.

Support in Pelix shell

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

Currently, all the Pelix Shell interfaces (local console, remote shell and XMPP) support 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.

API

Note that the `pelix.shell.console` module provides a `handle_common_arguments()` method to automate the use of an initial configuration with the arguments common to all Pelix Shell scripts:

Sample API Usage

This sample starts a framework based on the default configuration files (see *File lookup*), 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, if possible
if "pelix.ipopo.core" in init.bundles:
```

(continues on next page)

(continued from previous page)

```
    init.instantiate_components(framework.get_bundle_context())
else:
    print("iPOPO has not been setup in the configuration file.")

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

2.5.7 Logging

The best way to log traces in iPOPO is to use the logging module from the Python Standard 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:

Log Reader Service

The log reader provides the following methods:

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:

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.

```
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 $?
```

(continues on next page)

(continued from previous page)

```
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
$
```

2.5.8 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:

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

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.

HTTP response

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

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):
  """
  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.

2.5.9 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.

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

```
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, HttpPut

@ComponentFactory()
@Provides(HTTP_SERVLET)
@Property('_path', HTTP_SERVLET_PATH, '/api/v0')
```

(continues on next page)

(continued from previous page)

```
@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):
      response.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):
      response.send_content(200, "<p>Handled {} : {}</p>" \
         .format(some_id, filename))
```

2.5.10 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 protocols), 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 implementation, 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
```

(continues on next page)

(continued from previous page)

```
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 $?

# 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
```

(continues on next page)

(continued from previous page)

```
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 developed 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.

`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.

Core Services

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

The *import registry* service provides the `pelix.remote.registry` (constant string stored in `pelix.remote.SERVICE_REGISTRY`) service, with the following API:

2.5.11 Remote Service Admin

Pelix/iPOPO now includes an implementation of the Remote Service Admin OSGi specification. It has been contributed by Scott Lewis, leader of the Eclipse Communication Framework project.

This feature can be use to let multiple iPOPO and OSGi frameworks share their services. Note that Java is not mandatory when used only between iPOPO frameworks.

Note: This is a brand new feature, which might still contain some bugs and might not work with all versions of Python (especially 2.7).

As always, feedback is welcome: don't hesitate to report bugs on GitHub.

Links to ECF

The Remote Service Admin implementation in iPOPO is based an architecture similar to the Eclipse Communication Framework (implemented in Java). Most of the concepts have been kept in the Python implementation, it is therefore useful to check the documentation of this Eclipse project.

- ECF project page, the formal project page
- ECF wiki, where most of the documentation can be found
- ECF blog, providing news and description of new features

Some pages of the wiki are related to the links between Java and Python worlds:

- OSGi R7 Remote Services between Python and Java describes how to share remote services between an iPOPO Framework and an OSGi Framework.

Package description

The implementation of Remote Service Admin is provided by the `pelix.rsa` package, which is organized as follows (all names must be prefixed by `pelix.rsa`):

Module / Package	Description
`edef`	Definition of the EDEF XML endpoint description format
`endpointdescription`	`EndpointDescription` beans
`remoteserviceadmin`	Core implementation of RSA
`shell`	Shell commands to control/debug RSA
`providers.discovery`	Package of discovery providers
`providers.distribution`	Package of transport providers
`topologymanagers.basic`	Basic implementation of a Topology Manager

Providers included with Pelix/iPOPO

iPOPO includes some discovery and transport providers. More of them will be added in future releases.

etcd Discovery

Bundle pelix.rsa.providers.discovery.discovery_etcd

Requires *none*

Libraries python-etcd

This discovery provider uses etcd as a store of descriptions of endpoints. It depends on the python-etcd third-party package.

This discovery provider is instantiated immediately as the bundle is started. The instance configuration must therefore be given as Framework properties. Another solution is to kill the `etcd-endpoint-discovery` component and restart it with custom properties.

This provider can be configured with the following properties:

Property	Default value	Description
`etcd.hostname`	localhost	Address of the etcd server
`etcd.port`	2379	Port of the etcd server
`etcd.toppath`	/org.eclipse.ecf.provider.etcd.EtcdDiscoveryContainer	Path in etcd where to store endpoints
`etcd.sessionttl`	30	Session Time To Live

XML-RPC Distribution

Bundle pelix.rsa.providers.transport.xmlrpc

Requires HTTP Service

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

The XML-RPC distribution is the recommended provider for inter-Python communications. Note that it also supports communications between Python and Java applications. Its main advantage is that is doesn't depend on an external library, XML-RPC being supported by the Python Standard Library.

All components of this provider are automatically instantiated when the bundle starts. They can be configured using framework properties or by killing and restarting its components with custom properties.

Property	Default value	Description
`ecf.xmlrpc.server.hostname`	localhost	Hostname of the HTTP server (`None` for auto-detection)
`ecf.xmlrpc.server.path`	/xml-rpc	Path to use in the HTTP server
`ecf.xmlrpc.server.timeout`	30	XML-RPC requests timeout

Other properties are available but not presented here as they describe constants used to mimic the Java side configuration.

A sample usage of this provider can be found in the tutorial section: *RSA Remote Services using XmlRpc transport*.

Py4J Distribution

Bundle pelix.rsa.providers.transport.py4j

Requires HTTP Service

Libraries py4j, osgiservicebridge

Note: This provider works only in Python 3

This provider allows to discover and share a Python service with its Py4J gateway and vice versa.

It can be configured with the following properties:

Property	Default value	Description
`ecf.py4j.javaport`	25333	Port of the Java proxy
`ecf.py4j.pythonport`	25334	Port of the Python proxy
`ecf.py4j.defaultservicetimeout`	30	Timeout before gateway timeout

A sample usage of this provider can be found in the tutorial section: *RSA Remote Services between Python and Java*.

2.5.12 Pelix Shell

Most of the time, it is necessary to access a Pelix application locally or remotely in order to monitor it, to update its components or simply to check its sanity. The easiest to do those tasks is to use the Pelix Shell: it provides an extensible set of commands that allows to work on bundles, iPOPO components, ...

The shell is split into two parts:

- the core shell, handling and executing commands
- the UI, which handles input/output operations with the user

Pelix comes with some bundles providing shell commands for various actions, and a few UI implementations. Feel free to implement and, maybe, publish new commands UIs according to your needs.

In order to use the shell, the `pelix.shell.core` bundle must be installed and running. It doesn't require iPOPO and can therefore be used in minimalist applications.

Provided user interfaces

Pelix includes 3 main user interfaces:

- Text UI: the one to use when running a basic Pelix application
- Remote Shell: useful when managing an application running on a server
- XMPP Shell: useful to access applications behind firewalls

Common script arguments

Before looking at the available user interfaces, note that all of them support arguments to handle the Initial Configuration files (see *Initial Configuration File*).

In addition to their specific arguments, the scripts starting the user interfaces also accept the following ones:

Argument	Description
`-h`, `--help`	Prints the script usage
`--version`	Prints the script version
`-D KEY=VALUE`	Sets up a framework property
`-v`, `--verbose`	Sets the logger to DEBUG mode
`--init FILE`	Start by running a Pelix shell script
`--run FILE`	Run a Pelix shell script then exit
`-c FILE`, `--conf FILE`	Use a configuration file, above the system configuration
`-C FILE`, `--exclusive-conf FILE`	Use a configuration file, ignore the system configuration
`-e`, `--empty-conf`	Don't use any initial configuration

Text UI

The Text UI is the easiest way to manage or test your programs with Pelix/iPOPO. It provides the most basic yet complete interaction with the Pelix Shell core service.

If it is available, the Text UI relies on `readline` to provide command and arguments completion.

Script startup

The text (or console) UI can be started using the `python -m pelix.shell` command. This command will start a Pelix framework, with iPOPO and the most commonly used shell command providers.

This script only accepts the common shell parameters.

Programmatic startup

This UI is provided by the `pelix.shell.console` bundle. It is a raw bundle, which does not provide a component factory: the UI is available while the bundle is active. There is no configuration available when starting the Text UI programmatically.

Remote Shell

Pelix frameworks are often started on remote locations, but still need to be managed with the Pelix shell. Instead of using an SSH connection to work on a foreground server, you can use the Pelix Remote Shell.

The Pelix Remote Shell is a simple interface to the Pelix Shell core service of its framework instance, based on a TCP server. Unlike the console UI, multiple users can connect the framework at the same time, each with his own shell session (variables, ...).

By default, the remote shell starts a TCP server listening the local interface (*localhost*) on port 9000. It is possible to enforce the server by setting up OpenSSL certificates. The server will have its own certificate, which should be checked by the clients, and each client will have to connect with its own certificate, signed by an authority recognized by the server. See *How to prepare certificates for the Remote Shell* for more information on how to setup this kind of certificates.

Note: TLS features and arguments are available only if the Python interpreters fully provides the `ssl` module, *i.e.* if it has been built with OpenSSL.

Script startup

The remote shell UI can be started using the `python -m pelix.shell.remote` command. This command will start a Pelix framework with iPOPO, and will start a Python console locally.

In addition to the common parameters, the script accepts the following ones:

Argument	Default	Description
`--no-input`	*not set*	If set, don't start the Python console (useful for server/daemon mode)
`-a ADDR`, `--address ADDR`	local-host	Server binding address
`-p PORT`, `--port PORT`	9000	Server binding port
`--ca-chain FILE`	`None`	Path to the certificate authority chain file (to authenticate clients)
`--cert FILE`	`None`	Path to the server certificate file
`--key FILE`	`None`	Path to the server private key file
`--key-password PASSWORD`	`None`	Password of the server private key

Programmatic startup

The remote shell is provided as the `ipopo-remote-shell-factory` component factory defined in the `pelix.shell.remote` bundle. You should use the constant `pelix.shell.FACTORY_REMOTE_SHELL` instead of the factory name when instantiating the component.

This factory accepts the following properties:

Name	Default	Description
`pelix.shell.address`	localhost	Server binding address
`pelix.shell.port`	9000	Server binding port
`pelix.shell.ssl.ca`	`None`	Path to the clients certificate authority chain file
`pelix.shell.ssl.cert`	`None`	Path to the server's SSL certificate file
`pelix.shell.ssl.key`	`None`	Path to the server's private key
`pelix.shell.ssl.key_password`	`None`	Password of the server's private key

XMPP Shell

The XMPP shell interface allows to communicate with a Pelix framework using an XMPP client, e.g. Pidgin, Psi. The biggest advantages of this interface are the possibility to use TLS to encrypt conversations and the fact that it is an output-only communication. This allows to protect Pelix applications behind hardened firewalls, letting them only to connect the XMPP server.

It requires an XMPP account to connect an XMPP server. Early tests of this bundle were made against Google Talk (with a GMail account, not to be confused with Google Hangout) and a private OpenFire server.

Script startup

The XMPP UI can be started using the `python -m pelix.shell.xmpp` command. This command will start a Pelix framework with iPOPO, and will start a Pelix console UI locally.

In addition to the common parameters, the script accepts the following ones:

Argument	Default	Description
`-j JID`, `--jid JID`	`None`	Jabber ID (user account)
`--password PASSWORD`	`None`	Account password
`-s ADDR`, `--server ADDR`	`None`	Address of the XMPP server (found in the Jabber ID by default)
`-p PORT`, `--port PORT`	5222	Port of the XMPP server
`--tls`	*not set*	If set, use a STARTTLS connection
`--ssl`	*not set*	If set, use an SSL connection

Programmatic startup

This UI depends on the `sleekxmpp` third-party package, which can be installed using the following command:

```
pip install sleekxmpp
```

The XMPP shell is provided as the `ipopo-xmpp-shell-factory` component factory defined in the `pelix.shell.xmpp` bundle. You should use the constant `pelix.shell.FACTORY_XMPP_SHELL` instead of the factory name when instantiating the component.

This factory accepts the following properties:

Name	Default	Description
`shell.xmpp.server`	localhost	XMPP server hostname
`shell.xmpp.port`	5222	XMPP server port
`shell.xmpp.jid`	`None`	JID (XMPP account) to use
`shell.xmpp.password`	`None`	User password
`shell.xmpp.tls`	1	Use a STARTTLS connection
`shell.xmpp.ssl`	0	Use an SSL connection

Provided command bundles

Pelix/iPOPO comes with some batteries included. Here is the list of the bundles which provide commands for specific services.

Note that the commands themselves won't be described here: it is recommended to use the `help` command in the shell to have the latest usage information.

Bundle name	Description
`pelix.shell.ipopo`	Handles iPOPO factories and instances.
`pelix.shell.configadmin`	Handles the Configuration Admin service (provided by `pelix.misc.configadmin`). See *Configuration Admin*.
`pelix.shell.eventadmin`	Handles the Event Admin service (provided by `pelix.misc.eventadmin`). See *EventAdmin service*.
`pelix.shell.log`	Looks into the Log Service (provided by `pelix.misc.log`). See *Logging*.
`pelix.shell.report`	Generates framework state reports. See *Shell reports*.

How to provide commands

Shell Command service

Shell commands are detected by the Shell Core Service when a Shell Command service (use the `pelix.shell.SERVICE_SHELL_COMMAND` constant) is registered.

First, the Shell Core calls the `get_namespace()` method of the new service, in order to prepare the (potentially new) command namespace. Each shell command provider **should** have a unique, human-readable, namespace. Sometimes it can be interesting to have multiple services providing sets of optional commands in the same namespace, but this can lead to some unexpected behaviour, *e.g.* when trying to provide the same command name twice in the same namespace. A namespace must not contain spaces nor separator characters (dot, comma, ...).

Then, the Shell Core calls `get_methods()`, which must a return a list of (command name, command method) couples. Like its namespace, a command name must not contain spaces nor separator characters (dot, comma, ...).

Each command method must accept at least one argument: the `ShellSession` object representing the current session and handling interactions with the client. Note that the Python *docstring* of the method will be what is shown by the core *help* command.

The shell core bundle also provides a utility service, `pelix.shell.SERVICE_SHELL_UTILS`, which can be used to generate ASCII tables to print out to the user. This is the service used by the core method to print the list of bundles, services, iPOPO instances, etc..

Here is an example of a simple command service providing the *echo* and *hello* shell commands. *echo* accepts an unlimited list of arguments and prints it back to the client. *hello* asks a name if it wasn't given as parameter then says

hello.

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

@ComponentFactory("sample-commands-factory")
@Provides(pelix.shell.SERVICE_SHELL_COMMAND)
@Instantiate("sample-shell-commands")
class SampleCommands(object):
    """
    Sample shell commands
    """
    @staticmethod
    def get_namespace():
        """
        Retrieves the name space of this command handler
        """
        return "sample"

    def get_methods(self):
        """
        Retrieves the list of tuples (command, method) for this command handler
        """
        return [("echo", self.echo), ("hello", self.hello)]

    def hello(self, session, name=None):
        """
        Says hello
        """
        if not name:
            # Name not given as parameter, ask for it
            name = session.prompt("What's your name? ")

        session.write_line("Hello, {0} !", name)

    def echo(self, session, *words):
        """
        Prints back the words it has been given
        """
        session.write_line(" ".join(words))
```

To use this sample, simply start a framework with the Shell Core, a Shell UI and iPOPO, then install and start the sample bundle. For example:

```
bash:~ $ python -m pelix.shell
** Pelix Shell prompt **
$ start pelix.shell.toto
Bundle ID: 14
Starting bundle 14 (pelix.shell.toto)...
$ sample.echo Hello, world !
Hello, world !
$ hello World
Hello, World !
$ hello
What's your name? Thomas
Hello, Thomas !
```

The I/O handling of the `session` argument is implemented by the shell UI and hides the ways used to communicate with the client. The code of this example works with all UIs: local text UI, remote shell and XMPP shell.

API

How to prepare certificates for the Remote Shell

In order to use certificate-based client authentication with the Remote Shell in TLS mode, you will have to prepare a certificate authority, which will be used to sign server and clients certificates.

The following commands are a summary of OpenSSL Certificate Authority page by Jamie Nguyen.

Prepare the root certificate

- Prepare the environment of the root certificate:

```
mkdir ca
cd ca/
mkdir certs crl newcerts private
chmod 700 private/
touch index.txt
echo 1000 > serial
```

- Download the sample openssl.cnf file to the `ca/` directory and edit it to fit your needs.
- Create the root certificate. The following snippet creates a 4096 bits private key and creates a certificate valid for 7300 days (20 years). The `v3_ca` extension allows to use the certificate as an authority.

```
openssl genrsa -aes256 -out private/ca.key.pem 4096
chmod 400 private/ca.key.pem

openssl req -config openssl.cnf -key private/ca.key.pem \
    -new -x509 -days 7300 -sha256 -extensions v3_ca \
    -out certs/ca.cert.pem
chmod 444 certs/ca.cert.pem

openssl x509 -noout -text -in certs/ca.cert.pem
```

Prepare an intermediate certificate

Using intermediate certificates allows to hide the root certificate private key from the network: once the intermediate certificate has signed, the root certificate private key should be hidden in a server somewhere not accessible from the outside. If your intermediate certificate is compromised, you can use the root certificate to revoke it.

- Prepare the environment of the intermediate certificate:

```
mkdir intermediate
cd intermediate/
mkdir certs crl csr newcerts private
chmod 700 private/
touch index.txt
echo 1000 > serial
echo 1000 > crlnumber
```

- Download the sample intermediate/openssl.cnf file to the `ca/intermediate` folder and edit it to your needs.
- Generate the intermediate certificate and sign it with the root certificate. The `v3_intermediate_ca` extension allows to use the certificate as an intermediate authority. Intermediate certificates are valid less time than the root certificate. Here we consider a validity of 10 years.

```
openssl genrsa -aes256 -out intermediate/private/intermediate.key.pem 4096
chmod 400 intermediate/private/intermediate.key.pem

openssl req -config intermediate/openssl.cnf \
    -new -sha256 -key intermediate/private/intermediate.key.pem \
    -out intermediate/csr/intermediate.csr.pem

openssl ca -config openssl.cnf -extensions v3_intermediate_ca \
    -days 3650 -notext -md sha256 \
    -in intermediate/csr/intermediate.csr.pem \
    -out intermediate/certs/intermediate.cert.pem
chmod 444 intermediate/certs/intermediate.cert.pem

openssl x509 -noout -text -in intermediate/certs/intermediate.cert.pem

openssl verify -CAfile certs/ca.cert.pem \
    intermediate/certs/intermediate.cert.pem
```

- Generate the Certificate Authority chain file. This is simply the bottom list of certificates of your authority:

```
cat intermediate/certs/intermediate.cert.pem certs/ca.cert.pem \
    > intermediate/certs/ca-chain.cert.pem

chmod 444 intermediate/certs/ca-chain.cert.pem
```

Prepare the server certificate

The steps to generate the certificate is simple. For simplicity, we consider we are in the same folder hierarchy as before.

This certificate must has a validity period shorter than the intermediate certificate.

1. Generate a server private key. This can be done on any machine:

```
openssl genrsa -aes256 -out intermediate/private/server.key.pem 2048
openssl genrsa -out intermediate/private/server.key.pem 2048
chmod 400 intermediate/private/server.key.pem
```

2. Prepare a certificate signing request

```
openssl req -config intermediate/openssl.cnf \
    -key intermediate/private/server.key.pem -new -sha256 \
    -out intermediate/csr/server.csr.pem
```

3. Sign the certificate with the intermediate certificate. The `server_cert` extension indicates a server certificate which can't sign other ones.

```
openssl ca -config intermediate/openssl.cnf -extensions server_cert \
    -days 375 -notext -md sha256 \
    -in intermediate/csr/server.csr.pem \
    -out intermediate/certs/server.cert.pem
chmod 444 intermediate/certs/server.cert.pem

openssl x509 -noout -text -in intermediate/certs/server.cert.pem
```

(continues on next page)

(continued from previous page)

```
openssl verify -CAfile intermediate/certs/ca-chain.cert.pem \
    intermediate/certs/server.cert.pem
```

Prepare a client certificate

The steps to generate the client certificates are the same as for the server.

1. Generate a client private key. This can be done on any machine:

```
openssl genrsa -out intermediate/private/client1.key.pem 2048
chmod 400 intermediate/private/client1.key.pem
```

2. Prepare a certificate signing request

```
openssl req -config intermediate/openssl.cnf \
    -key intermediate/private/client1.key.pem -new -sha256 \
    -out intermediate/csr/client1.csr.pem
```

3. Sign the certificate with the intermediate certificate. The `usr_cert` extension indicates this is a client certificate, which cannot be used to sign other certificates.

```
openssl ca -config intermediate/openssl.cnf -extensions usr_cert \
    -days 375 -notext -md sha256 \
    -in intermediate/csr/client1.csr.pem \
    -out intermediate/certs/client1.cert.pem
chmod 444 intermediate/certs/client1.cert.pem

openssl x509 -noout -text -in intermediate/certs/client1.cert.pem

openssl verify -CAfile intermediate/certs/ca-chain.cert.pem \
    intermediate/certs/client1.cert.pem
```

Connect a TLS Remote Shell

To connect a basic remote shell, you can use `netcat`, which is available for nearly all operating systems and all architectures.

To connect a TLS remote shell, you should use the OpenSSL client: `s_client`. It is necessary to indicate the client certificate in order to be accepted by the server. It is also recommended to indicate the authority chain to ensure that the server is not a rogue one.

Here is a sample command line to connect a TLS remote shell on the local host, listening on port 9001.

```
openssl s_client -connect localhost:9001 \
    -cert client1.cert.pem -key client1.key.pem \
    -CAfile ca-chain.cert.pem
```

2.5.13 Shell reports

Pelix/iPOPO comes with a bundle, `pelix.shell.report`, which provides commands to generate reports describing the current framework and its host. This main purpose of this feature is to debug a faulty framework by grabbing all available information. It can also be used to have a quick overview of the operating system, either to check the installation environment or to identify the host machine.

Setup

This feature requires an active Pelix Shell (`pelix.shell.core`) and a UI. See *Pelix Shell* for more information on this subject. The iPOPO service is not required for this feature to work.

It can therefore be started programmatically using the following snippet:

```
# Start the framework, with the required bundles and the report bundle
framework = create_framework(
    ["pelix.shell.core", "pelix.shell.report"])

# ... or install & start it using the BundleContext
bundle_context.install_bundle("pelix.shell.report").start()
```

It can only be installed from a Shell UI using the command `start pelix.shell.report`.

Usage

The bundle provides the following commands in the `report` namespace:

Command	Description
`clear`	Clears the last report
`levels`	Lists the available levels of reporting
`make [<levels ...>]`	Prepares a report with the indicated levels (all levels if none set)
`show [<levels ...>]`	Shows the latest report. Prepares it if levels have been indicated
`write <filename>`	Write the latest report as a JSON file

Report levels

The reports are made of multiple "level information" sections. They describe the current state of the application and its environment.

Here are some of the available levels.

Framework information

Level	Description
`pelix_basic`	Framework properties and version
`pelix_bundles`	Bundles ID, name, version, state and location
`pelix_services`	Services ID, bundle and properties
`ipopo_factories`	Description of iPOPO factories (with their bundle)
`ipopo_instances`	Details of iPOPO instances

Process information

Level	Description
`process`	Details about the current process (PID, user, working directory, ...)
`threads`	Lists the current process threads and their stacktrace

Python information

Level	Description
`python`	Python interpreter details (version, compiler, path, ...)
`python_modules`	Lists all Python modules imported by the application
`python_path`	Lists the content of the Python Path

Host information

Level	Description
`os`	Details about the OS (version, architecture, CPUs, ...) and the host name
`os_env`	Lists the environment variables and their value
`network`	Lists the IPs (v4 and v6) of the host, its name and FQDN.

Group levels

Some levels are groups of lower levels. They are subject to change, therefore the following table is given as an indication. Always refer to the `report.levels` shell command to check available ones.

Level	Description
`pelix`	Combines `pelix_infos`, `pelix_bundles` and `pelix_services`
`ipopo`	Combines `ipopo_factories` and `ipopo_instances`
`app`	Combines `os`, `os_env`, `process`, `python` and `python_path`
`debug`	Combines `app` (except `os_env`), `pelix`, `ipopo` and `python_modules`
`standard`	Like `debug`, but without `pelix_services` nor `ipopo_instances`
`full`	Combines `debug`, `os_env`, `network` and `threads`

Those *groups* were defined according to the most common combinations of levels used during iPOPO development and live setup.

2.5.14 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 very 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:

```
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:

```
@BundleActivator
class Activator:
    def __init__(self):
        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 = {}
```

(continues on next page)

(continued from previous page)

```
# 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()
            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
            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()
```

2.5.15 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'])
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:

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)
# 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.

```
# 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/*'])
# 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:

```
# 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 3

API Reference

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

3.1 API

This part of the documentation covers all the core classes and services of iPOPO.

3.1.1 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.

3.1.2 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`.

3.1.3 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.

3.1.4 Events Objects

Those objects are given to listeners when a bundle or a service event occurs.

CHAPTER 4

Additional Notes

Design notes, legal information and changelog are here for the interested.

4.1 Who uses iPOPO ?

If you want to add your name here, send a mail on the ipopo-users mailing list.

4.1.1 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.

4.1.2 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.

4.1.3 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.

4.2 Release Notes

4.2.1 iPOPO 1.0.0

Release Date 2020-01-19

Project

- The Pelix/iPOPO is now split in two branches: iPOPO (`v1` branch) and ipopo2 (`v2` branch). The `v2` branch requires Python 3.7+, whereas `v1` will keep compatibility with Python 2.7.

Pelix

- Fixed an error when starting the framework after having loaded native modules, *e.g.* `numpy`. These modules don't have a `__path__` set, which case was not handled when the framework normalizes the existing module paths.
- Fixed an invalid import of `collections` abstract classes for Python 3.3+ in `pelix.internal.hooks`.

4.2.2 iPOPO 0.8.1

Release Date 2018-11-17

Pelix

- Fixed a memory leak in the thread pool implementation. The patch comes from issue #35 of the jsonrpclib-pelix project.

Remote Services

- Fixed a deadlock in the Py4J provider (issue #100), contributed by Scott Lewis (@scottslewis). See pull request #101 for more details.
- Use a local `etcd` server in Travis-CI instead of a public one.

4.2.3 iPOPO 1.0.0

Release Date 2018-08-19

Project

- Version bump to 0.8 as the addition of Remote Service Admin is a big step forward.
- Fixed unit tests for `pelix.threadpool`
- Added a word about Python 3.7 dataclasses in the iPOPO reference card
- All the source code has been reformatted with black (`black -l 80 pelix`)

Remote Services

- Added the implementation of Remote Service Admin OSGi specification, contributed by Scott Lewis (@scottslewis). This is a major feature which intends to be used instead of Pelix Remote Services. The latter will be kept for retro-compatibility reasons.

4.2.4 iPOPO 0.7.1

Release Date 2018-06-16

Project

- Added a CONTRIBUTING description file to describe the code style
- The `zeroconf` dependency is now forced to version 0.19, to stay compatible with Python 2.7
- Changed them in the documentation (back to standard ReadTheDocs theme)
- Added some reference cards in the documentation: initial configuration file, shell, shell report

Pelix

- Aded support for Event Listeners Hooks. See pull request #88 for more details.
- Fixed `Framework.delete()` when framework was already stopped.

iPOPO

- Added `@ValidateComponent` and `@InvalidateComponent` decorators. They allow to define callback methods for component in/validation with access to component context and properties (read-only). `@Validate` and `@Invalidate` decorators are now simple aliases to those decorators.
- Checked behaviour with *data classes*, introduced in Python 3.7: all seems to work perfectly. See issue 89 for more details.

Shell

- New shell completion system: completion is now extensible and can work with both commands and arguments. This system relies on `readline`.
- Added a TLS version of the shell. Its usage and the generation of certificates are described in the Pelix Shell reference card in the documentation.
- `ShellSession.write_line()` can now be called without argument (prints an empty line)

Misc

- Fixed the access bug to the Python LogRecord message in the Log Service

4.2.5 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.

4.2.6 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 with 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, ...

4.2.7 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.

4.2.8 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

4.2.9 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.

4.2.10 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.

4.2.11 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.

4.2.12 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.

4.2.13 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*

4.2.14 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.

4.2.15 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.

4.2.16 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

4.2.17 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.

4.2.18 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.

4.2.19 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

4.2.20 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.

4.2.21 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.

4.2.22 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

4.2.23 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

4.2.24 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.

4.2.25 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.

4.2.26 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.

4.3 License

iPOPO is licensed under the terms of the Apache Software License 2.0. All contributions must comply with this license.

4.3.1 File Header

This snippet is added to the module-level documentation:

```
Copyright 2020 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.
```

4.3.2 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,
```

(continues on next page)

(continued from previous page)

```
   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.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
```

(continues on next page)

(continued from previous page)

```
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
```

(continues on next page)

(continued from previous page)

```
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

   To apply the Apache License to your work, attach the following
```

(continues on next page)

(continued from previous page)

```
    boilerplate notice, with the fields enclosed by brackets "[]"
    replaced with your own identifying information. (Don't include
    the brackets!)  The text should be enclosed in the appropriate
    comment syntax for the file format. We also recommend that a
    file or class name and description of purpose be included on the
    same "printed page" as the copyright notice for easier
    identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

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.
```

Python Module Index

p

Index