The POCO Open Service Platform

Specification

Version 1.0

Purpose of This Document

This document specifies the programming interfaces and implementation details of the Applied Informatics POCO Open Service Platform.

The document is targeted at developers implementing the POCO Open Service Platform.

Validity of This Document

This document covers release 1.0 and later releases of the Applied Informatics POCO Open Service Platform.

Copyright, Trademarks, Disclaimer

Table of Contents

1 Introduction

Today's applications are becoming ever more complex. A large part of this complexity stems from the need to support different configurations, e.g. for different devices, operating system environments or customer requirements. Huge monolithic applications like the ones developed in the past do not (or only at an enormous cost) provide the necessary flexibility required today.

Many companies have noticed this, and are basing their latest software architectures on a dynamic plug-in model. In this model, they implement a very basic skeleton application merely acting as a loader and execution environment (or container) for plug-ins – shared libraries loaded dynamically as required at run-time. All the features of an application are then provided by plug-ins, which can be added to an application on demand. While Java developers have been able – for a few years now – to choose between the various readily available implementations of such a framework based on the OSGi Service Platform specification[1] developed by the OSGi Foundation, companies relying on C++ have no such standard environment readily available and are forced to implement their own system.

This is where the POCO Open Service Platform (OSP) comes in. OSP is a C++ based middleware providing a service-oriented and component-based environment for developing, deploying, running and managing modular network-based applications. As such, OSP is to C++ to what the OSGi Service Platform is to Java. In fact, the design of OSP draws many ideas and inspirations from the OSGi Service Platform.

POCO OSP is based on the POCO C++ Libraries[2]. The POCO C++ Libraries are open-source class libraries that simplify the development of network-centric, portable applications in C++. The libraries integrate perfectly with the C++ Standard Library and fill many of the functional gaps left open by it.

The classes provided by the POCO C++ Libraries provide support for multi-threading, streams, accessing the file system, shared libraries and class loading, configuration file and command line handling, security, network programming (TCP/IP sockets, HTTP, FTP, SMTP, SSL/TLS, etc.), XML parsing (SAX2 and DOM) and generation, as well as access to SQL databases. Applied Informatics provides various additional C++ class libraries based on POCO, providing features such as distributed objects and web services, automatic discovery of network services, as well as remote configuration capabilities based on the NETCONF protocols.

Using the POCO C++ Libraries as a foundation makes OSP available on a variety of platforms, from Windows and Solaris based enterprise systems to embedded Linux based smart devices. Applications based on the POCO C++ Libraries – and therefore, applications based on POCO OSP – can be

[1] http://www.osgi.org
[2] See http://pocoproject.org for more information on the POCO C++ Libraries

compiled for and executed on different platforms, all from the same source code base.

This component-based architecture of OSP addresses an increasing problem in software development: The large number of application configurations that need to be developed and maintained. The standardized OSP component architecture simplifies this configuration process significantly.

2 OSP Architecture Overview

The POCO Open Service Platform is based on a layered architecture, depicted in Figure 1. At the core of OSP is the Portable Runtime Environment, consisting of the C and C++ standard libraries and the POCO Core Libraries (Foundation, XML, Util and Net). Layered above the Portable Runtime Environment is the OSP Framework, consisting of Service Registry, Life Cycle Management, Bundle Management and Standard Services. Application-specific bundles based on the OSP Framework implement the actual application logic.

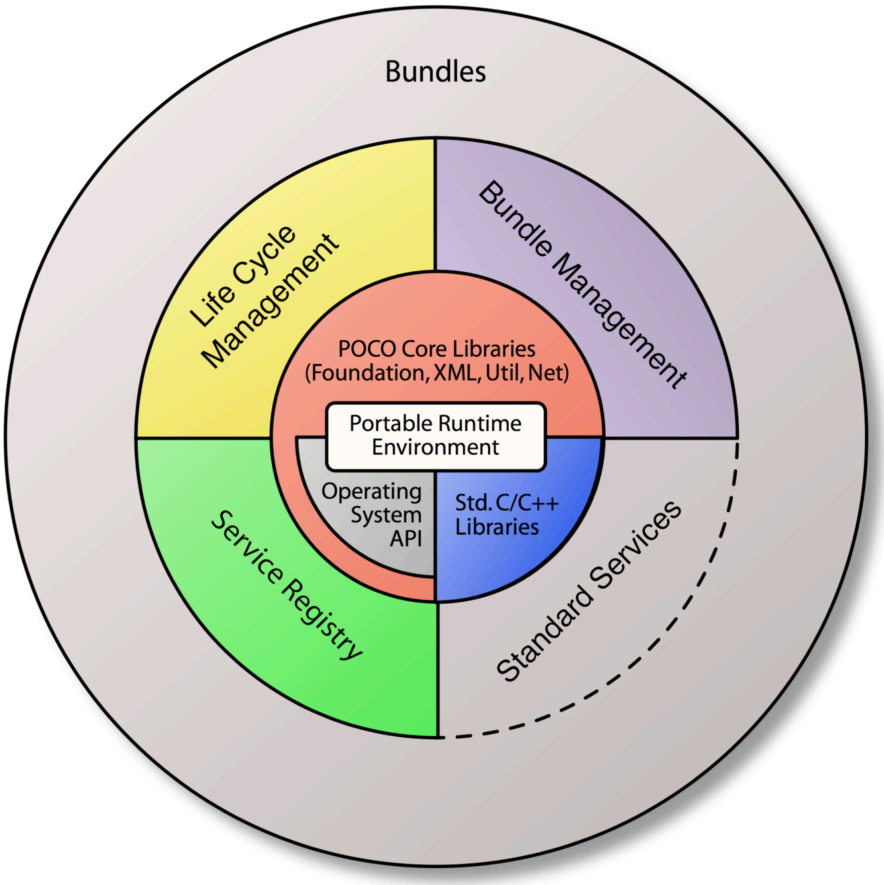


Figure 1: Open Service Platform layered architecture

2.1 Portable Runtime Environment

The Portable Runtime Environment sits at the center of the OSP architecture. Based on the C and C++ Standard Libraries, as well as on the POCO Core Libraries, it provides platform-independent low-level services to the upper OSP layers, such as:

- access to the file system
- multithreading support
- shared library and class loading
- notifications and events
- logging
- XML parsing
- configuration data handling
- TCP/IP sockets and support for various network protocols (HTTP, FTP, SMTP, POP3, etc.)
- various utility classes and functions

By isolating applications from the operating system interfaces, the Portable Runtime Environment makes it possible to write applications that can be compiled for and run on different operating system platforms and processor architectures, all from the same source code.

2.2 Service Registry

The Service Registry allows a bundle to register its services to make them available to other bundles, as well as to discover the services provided by other bundles. Since bundles can appear and disappear in an application at any time, the Service Registry also provides notification mechanisms so that a bundle can be informed when another bundles it uses disappears from the system.

A bundle can discover a certain service by using a simple query language to query the Service Registry for a service with certain properties.

2.3 Life Cycle Management

Bundles adhere to a well-defined life cycle, shown in Figure 2. The Life Cycle Management in OSP ensures that every bundle in the system follows this life cycle, which is outlined in the following.

A bundle is installed into an application and starts its life cycle in the Installed state. From the Installed state, a bundle can either be uninstalled, or resolved. Resolving a bundle means determining all its dependencies on other bundles, and verifying that all required bundles are available. Only a successfully resolved bundle can be started. Starting a bundle causes a special class provided by the bundle, the *Bundle Activator*, to be loaded and invoked. The Bundle Activator then takes care of registering the bundle's services and does all the necessary steps so that the bundle can provide its services. After the Bundle Activator has successfully done its work, the bundle is in Active state. Eventually, an active bundle will be stopped again, and possibly removed from the system (uninstall).

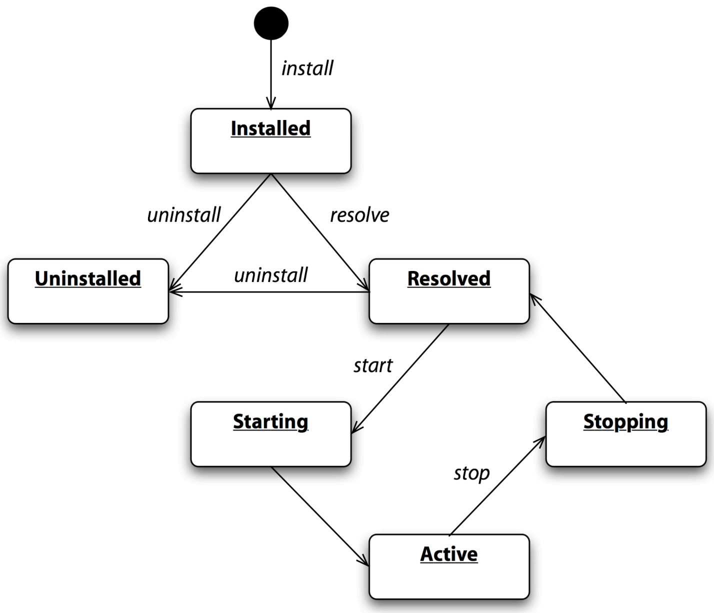


Figure 2: The bundle life cycle

2.4 Bundle Management

The Bundle Management layer in OSP takes care of dealing with the physical storage of bundles in the file system, as well as the loading of shared libraries contained in bundles.

Bundles are a collection of files (manifest, shared libraries, data files, configuration files, resource files, etc.) stored in a certain directory structure. For easier deployment, a bundle can be packed into an archive, using the Zip file format. In a future version of OSP it will also be possible to cryptographically sign a bundle and to encrypt the contents of a bundle.

Bundles can contain multiple versions of a shared library for different operating system platforms and hardware architectures. Bundle Management ensures that the correct version of a shared library for the current operating system and hardware architecture is loaded. This makes it possible to provide a single bundle file that can be, for example, deployed on both a Windows and a Linux system.

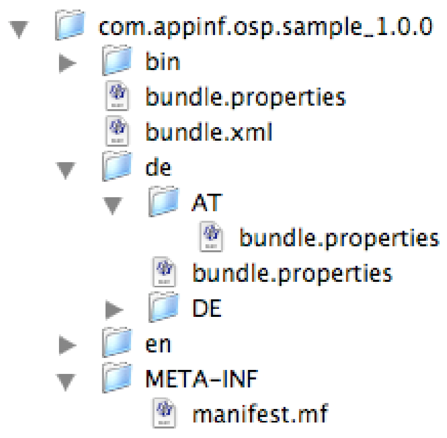
2.5 Standard Services

OSP comes with a number of standard services implementing commonly required features. Examples include a HTTP server, support for sending email messages, Web-based bundle management, as well as a configuration and preferences storage service.

3 Bundles

Bundles are the unit of deployment in OSP. A bundle is a directory containing all files that make up a bundle, in a well-defined directory hierarchy. These files can be configuration files, shared libraries, HTML files, etc. For easier deployment, a bundle can be packed into a Zip[3] file. The format of a bundle is loosely based on the format of OSGi bundles; however, there are some differences.

Figure 3 gives an overview of the general directory layout for a bundle.



```
▼ com.appinf.osp.sample_1.0.0
  ▶ bin
    bundle.properties
    bundle.xml
  ▼ de
    ▼ AT
        bundle.properties
      bundle.properties
    ▶ DE
  ▶ en
  ▼ META-INF
      manifest.mf
```

Figure 3: Bundle directory hierarchy example

3.1 Bundle Naming

The name of a bundle must conform to certain conventions. Bundle names must be unique across different vendors. To ensure this, the bundle name employs the reverse domain name scheme known for example from Java namespaces. The name consists of a number of parts, separated by periods. The first part is the top-level domain of the vendor (e.g., “com”). The second part is the domain name of the company (e.g., “appinf”). The remaining parts can be freely specified by the vendor, and usually include a product name, subsystem name, module name, etc. There is no limit to the number of parts in a name, although for practical reasons, a bundle name should not consist of more than five parts. For maximum portability across different platforms, a name part must not contain any characters other than upper- and lowercase alphabetic characters (‘A’ – ‘Z’), digits (‘0’ – ‘9’) and dash (‘-’).

Appended to the bundle name, separated with an underscore, is the version designation of the bundle. The version designation consists of a major version number, minor version number and revision number, separated by a period. Appended to the revision number, separated by a period can be an optional vendor specific release designation that must conform to the same naming rules as a name part.

[3] The specification for the Zip file format can be downloaded from http://www.pkware.com/documents/casestudies/APPNOTE.TXT

```
name-version     ::= bundle-name "_" bundle-version
bundle-name      ::= bundle-id ("." bundle-id)*
bundle-version   ::= version ["_" bundle-id]
version          ::= major "." minor "." revision
bundle-id        ::= bundle-char+
bundle-char      ::= letter | digit | "-"
major            ::= digit+
minor            ::= digit+
revision         ::= digit+
letter           ::= 'A' .. 'Z' | 'a' .. 'z'
digit            ::= '0' .. '9'
```

Bundle names in the `osp.*` namespace are reserved for use by OSP.

3.2 Bundle Versioning

Every bundle has a version specification consisting of a major version number, a minor version number, a revision number, and an optional release designation. When comparing versions, only major version, minor version and revision number are significant; the release designation is ignored.

If multiple versions of a bundle are installed, only the highest version of a bundle will be considered. All other versions will be ignored.

3.3 Bundle Directory Layout

Every bundle has one mandatory subdirectory named `META-INF`. This directory contains the bundle's Manifest file (`MANIFEST.MF`) and, optionally, a few other files (e.g., for signed bundles). A bundle containing executable binary code in the form of shared libraries also has a directory named `bin`, with subdirectories therein containing the platform-specific shared libraries. A bundle can contain code for multiple platforms and the subdirectories under `bin` reflect this. For every supported operating system, there is a directory named after the operating system. Under the operating system directory, there is a directory for each supported processor architecture.

The name of the operating system directory is derived from the name returned by `Poco::Environment::osName()`, by replacing whitespace characters with underscore characters. The name of the processor architecture directory is derived from the name returned by `Poco::Environment::osArchitecture()`, by replacing whitespace characters with underscore characters.

The shared library files should follow the same naming convention as bundles. During startup of OSP, the shared libraries for the current platform from all installed bundles will be copied into a common cache directory, so extra care must be taken to prevent the shared libraries from one bundle overwrite the shared libraries from another one.

3.4 Bundle Files

Bundles stored in a Zip file have the same name as the top-most bundle directory, with the extension `.bndl` appended. (e.g., `com.appinf.osp.sample_1.0.0.bndl`). The top-most bundle directory is not represented in the ZIP file, only its subdirectories and files (e.g., `META-INF`, `bin`, `bundle.properties`) are.

3.5 Bundle Manifest

The bundle manifest contains information about a bundle that allows the bundle management machinery of OSP to load a bundle into an application. Examples for information stored in the manifest are:

- the bundle's name
- the bundle's human-readable display name
- the bundle's version
- a list of bundles that must be available for the bundle to run
- the class name and library name where the bundle's `BundleActivator` class can be found

The bundle manifest is contained in a UTF-8 encoded text file that conforms to the file format specified by the `PropertyFileConfiguration` class in the Util library[4]. The name of the file is `MANIFEST.MF` and the file is always located in the `META-INF` directory. Following is an example for a manifest file.

```
# This is a sample manifest file
Manifest-Version: 1.0
Bundle-Name: OSP Sample Bundle
Bundle-SymbolicName: com.appinf.osp.sample
Bundle-Version: 1.0.0
Bundle-Vendor: Applied Informatics
Bundle-Copyright: Copyright (c) 2007
Bundle-Activator: OSP::Sample::BundleActivator
```

A property value in a manifest file can contain references to the bundle's properties (stored in `bundle.properties` and its localized variants). A list of currently defined manifest properties is given in the next sections. If a property is not recognized by OSP, it is ignored.

3.5.1 Manifest-Version

This property specifies the version of the file format of the manifest file. Currently, this is always "1.0". This property is required.

```
manifest-version ::= digit+ "." digit+
```

[4] This is a major difference to OSGi, which uses a file format that's similar to, but not identical to Java property files.

3.5.2 Bundle-Name

This property contains the human-readable name of the bundle. It is only used for display purposes and can be an arbitrary string.

3.5.3 Bundle-Vendor

This property contains the name of the organization or company that created the bundle. It is only used for display purposes and can be an arbitrary string.

3.5.4 Bundle-Copyright

This property contains copyright information for the bundle. It is only used for display purposes and can be an arbitrary string.

3.5.5 Bundle-SymbolicName

The symbolic name of the bundle, but without any version information. See 3.1 for details. This property is required.

3.5.6 Bundle-Version

The version of the bundle. See 3.1 and 0 for more information. This property is required.

3.5.7 Bundle-Activator

The fully-qualified name of the class implementing the `IBundleActivator` interface for this bundle. Optionally, the name of the shared library containing the class can be specified. If no shared library name is specified, the name of the library is assumed to be the same as the name of the bundle.

```
bundle-activator ::= class-name [“;” “library” “=” library-name]
class-name       ::= id {“::” id}
library-name     ::= bundle-name
id               ::= letterUS (letterUS | digit)*
letterUS         ::= letter | “_”
```

3.5.8 Require-Bundle

A list of bundles that must be available for this bundle to run. Both the name and the possible versions of required bundles can be specified. It is possible to specify either a list of required versions, or an interval of required versions.

```
require-bundle   ::= req-name-version (“,” req-name-version)
req-name-version ::= bundle-name [“;” “bundle-version” “=” req-version]
req-version      ::= part-version | interval
interval         ::= (“[“ | “(“) part-version “,” part-version (“]” | “)”)
```

```
part-version     ::= major [“.” minor [“.” revision]]
```

If only a single version number is given, the version of the bundle must be *at least* the given version. If no version number is given, any version of the bundle can be used. An interval can be, mathematically speaking, open, closed or half-closed. The open interval `(1.0.0,1.1.0)` means any version greater than 1.0.0 and less than 1.1.0. The closed interval `[1.0.0,1.1.0]` means any version greater than or equal to 1.0.0, and less than, or equal to 1.1.0. The half-closed interval `[1.0.0,1.1.0)` means any version greater than or equal to 1.0.0 and less than 1.1.0.

3.5.9 Bundle-LazyStart

This property specifies whether the bundle should be started automatically at application startup. The default is `false`, which means the bundle will be automatically started. If set to `true`, the bundle will be started whenever another bundle depending on it is also started, or when it is started explicitly.

```
bundle-lazyStart ::= “true” | “false”
```

3.5.10 Bundle-RunLevel

The `Bundle-RunLevel` property specifies the run level in which the bundle should be started. Its value is used by the bundle loader to start bundles in a certain order. A run level is specified in the form *nnn-description*, where *nnn* is a three-digit string denoting the order, and *description* is an optional textual description of the run level. When starting bundles, the bundle loader sorts bundles by their run level. A simple string comparison is used for sorting, so the specified format for run level strings should be strictly followed to avoid surprises.

Run levels 000 to 099 are reserved for use by OSP.

If a bundle does not specify a run level, the default is `999-user`.

```
bundle-runLevel ::= digit digit digit [“-” id]
```

3.6 BundleManifest Class

The `BundleManifest` class is a helper class used by the `Bundle` class to parse and hold the contents of the bundle manifest file.

```
class BundleManifest: public Poco::RefCountedObject
{
public:
    typedef Poco::AutoPtr<BundleManifest> Ptr;
    typedef const Ptr ConstPtr;

    struct Dependency
    {
        std::string  name;
        VersionRange versions;
    };
    typedef std::vector<std::string> Dependencies;

    const std::string& name() const;
        /// Returns the bundle's name.

    const std::string& symbolicName() const;
        /// Returns the bundle's symbolic name.

    const Version& version() const;
        /// Returns the bundle's version.

    const std::string& vendor() const;
        /// Returns the bundle's vendor, or an empty
        /// string if no vendor has been defined
        /// in the bundle's manifest.

    const std::string& copyright() const;
        /// Returns the bundle's copyright notice,
        /// or an empty string, if no copyright
        /// notice has been defined in the bundle's
        /// manifest.

    const std::string& activatorClass() const;
        /// Returns the bundle's activator class
        /// name, or an empty string if no activator
        /// has been specified in the bundle's manifest.

    const std::string& activatorLibrary() const;
        /// Returns the bundle's activator library
        /// (the name of the library containing
        /// the bundle's activator class).
        ///
        /// Defaults to the bundle's symbolic name.

    bool lazyStart() const;
        /// Returns true iff lazy startup has been specified
        /// in the bundle manifest.

    const std::string& runLevel() const;
        /// Returns the bundle's run level.
        /// If the bundle does not specify a run level,
        /// the default is "999-user".

    void requiredBundles(std::set<Dependency>& reqs) const;
        /// Returns a set containing information about all
        /// bundles that this bundle requires to function.
};
```

3.7 Bundle Class

In the OSP framework, a bundle is represented by the `Bundle` class, the interface of it is given below.

```
class Bundle: public Poco::RefCountedObject
{
public:
    typedef Poco::AutoPtr<Bundle> Ptr;
    typedef const Ptr ConstPtr;
    enum State
    {
        BUNDLE_INSTALLED,
        BUNDLE_UNINSTALLED,
        BUNDLE_RESOLVED,
        BUNDLE_STARTING,
        BUNDLE_ACTIVE,
        BUNDLE_STOPPING
    };

    State state() const;
        /// Returns the bundle’s state.

    bool isResolved() const;
        /// Returns true if the bundle has been successfully resolved.
        /// In other words, the bundle’s state is either “resolved”,
        /// “starting”, “active” or “stopping”.

    bool isActive() const;
        /// Returns true if the bundle’s state is “active”.

    bool isStarted() const;
        /// Returns true if the bundle’s state is either “starting”,
        /// “active” or “stopping”.

    const std::string& name() const;
        /// Returns the bundle’s name.

    const std::string& symbolicName() const;
        /// Returns the bundle’s symbolic name.

    const Version& version() const;
        /// Returns the bundle’s version.

    const std::string& vendor() const;
        /// Returns the bundle’s vendor, or an empty
        /// string if no vendor has been defined
        /// in the bundle’s manifest.

    const std::string& copyright() const;
        /// Returns the bundle’s copyright notice,
        /// or an empty string, if no copyright
        /// notice has been defined in the bundle’s
        /// manifest.

    const std::string& activatorClass() const;
        /// Returns the bundle’s activator class
        /// name, or an empty string if no activator
        /// has been specified in the bundle’s manifest.

    const std::string& activatorLibrary() const;
        /// Returns the bundle’s activator library
        /// (the name of the library containing
        /// the bundle’s activator class).
        ///
        /// Defaults to the bundle’s symbolic name.

    BundleActivator* activator() const;
        /// If the bundle is active, returns a pointer
        /// to the bundle’s activator, otherwise
        /// returns NULL.

    bool lazyStart() const;
```

```
        /// Returns true iff lazy startup has been specified
        /// in the bundle manifest.

    const std::string& runLevel() const;
        /// Returns the bundle’s run level.
        /// If the bundle does not specify a run level,
        /// the default is “999-user”.

    const Poco::Util::AbstractConfiguration& properties() const;
        /// Returns the bundle’s properties, which are
        /// obtained from the bundle’s bundle.properties
        /// file (and its optional localizations).

    std::istream* getResource(const std::string& name) const;
        /// Creates and returns an input stream for reading
        /// the bundle’s (non-localized) resource with the given name.
        ///
        /// Resources are ordinary files stored within
        /// the bundle’s directory or archive file.
        ///
        /// Returns NULL if the resource does not exist.

    std::istream* getLocalizedResource(const std::string& name) const;
        /// Creates and returns an input stream for reading
        /// the bundle’s localized resource with the given name.
        ///
        /// Resources are ordinary files stored within
        /// the bundle’s directory or archive file.
        ///
        /// Returns NULL if the resource does not exist.

    const std::string& path() const;
        /// Returns the path to the bundle’s directory
        /// or archive file.

    void resolve();
        /// Resolves the bundle by checking the availability of
        /// all required bundles. The bundle must be in “installed”
        /// state. A successful resolve() operation puts the bundle
        /// into “resolved” state.
        /// Throws an exception if the bundle cannot be resolved.

    void start();
        /// Starts the bundle. The bundle’s state must be “resolved”.
        /// Puts the bundle into “starting” state, loads the
        /// bundle’s activator and runs it, and finally puts the
        /// bundle in “active” state.

    void stop();
        /// Stops the bundle. The bundle’s state must be “active”.
        /// Puts the bundle into “stopping” state, stops the
        /// bundle activator and finally puts the bundle in
        /// “resolved” state.

    void uninstall();
        /// Uninstalls the bundle. The bundle’s state must be “resolved”
        /// or “installed”. The bundle is also removed from the file
        /// system.

    void requiredBundles(BundleManifest::Dependencies& reqs) const;
        /// Returns a set containing information about all
        /// bundles that this bundle requires to function.
    void dependentBundles(std::set<Bundle*>& deps) const;
        /// Returns a set containing pointers to all bundles
        /// depending upon this bundle.
};
```

The `Bundle` class requires the `Version` and `VersionRange` classes, given in the following.

```
class Version // value semantics
{
public:
    int major() const;
        /// Returns the major version number.

    int minor() const;
        /// Returns the minor version number.

    int revision() const;
        /// Returns the revision number.

    const std::string& release() const;
        /// Returns the release designation, which may
        /// be an empty string.

    std::string toString() const;
        /// Returns a string representation of the
        /// version information.

    bool operator == (const Version& version);
    bool operator != (const Version& version);
    bool operator <  (const Version& version);
    bool operator <= (const Version& version);
    bool operator >  (const Version& version);
    bool operator >= (const Version& version);
};
```

```
class VersionRange
{
public:
    const Version& lowerBound() const;
        /// Returns the lower bound of the range.

    const Version& upperBound() const;
        /// Returns the upper bound of the range.

    bool includeLower() const;
        /// Returns true iff the lower bound is included
        /// in the range.

    bool includeUpper() const;
        /// Returns true iff the upper bound is included
        /// in the range.

    bool isInRange(const Version& version) const;
        /// Returns true iff the given version lies within
        /// the range.

    bool isEmpty() const;
        /// Returns true iff the range is empty.
};
```

A bundle can be stored in the file system either in a special directory hierarchy, or in a single, Zip-compressed file. A special interface, `BundleStorage`, manages file I/O for bundles.

```
class BundleStorage: public Poco::RefCountedObject
{
public:
    typedef Poco::AutoPtr<BundleStorage> Ptr;
    typedef const Ptr ConstPtr;

    virtual std::istream* getResource(const std::string& path) const = 0;
        /// Returns an input stream for reading the resource
        /// with the given path, if the resource exists.
        /// Otherwise, returns a NULL pointer.

    virtual void list(
        const std::string& path,
        std::vector<std::string>& files);
        /// List all files in the directory specified by path.

    virtual const std::string& path() const;
        /// Returns the path to the bundle's directory
        /// or archive file.
};
```

Two subclasses of `BundleStorage` are implemented by OSP: `BundleDirectory` manages a bundle stored in a directory hierarchy and `BundleFile` manages a bundle stored in a Zip file.

3.7.1 Resource Localization

The `Bundle` class supports resource localization/internationalization for resources obtained with the `getResource()` member function. Resource localization can be done on two levels – language and country. The first level of localization is the language. For a given language, a number of countries can be defined. For example, if the language is English, two possible countries would be United States and United Kingdom. If the language is German, the countries could be Austria, Germany and Switzerland. Localization is optional. Also, localization for a certain country is optional. If a resource for a certain country cannot be found, the default resource for language is taken. If the resource for a certain language cannot be found, the global default resource is taken.

Localized resources are stored in a certain directory hierarchy within a bundle. For every supported language, there is a directory having the name of the specific ISO language code (e.g., “en” for English, “de” for German) within the bundle directory. Under a language directory can be directories corresponding to supported countries. These directories are named using ISO country codes (e.g., “AT” for Austria, “US” for United States, etc.).

Figure 4 shows a sample directory hierarchy for a localized bundle. The resource named `bundle.properties` is localized for different languages and countries – German/Austria, German/Germany and various English countries.

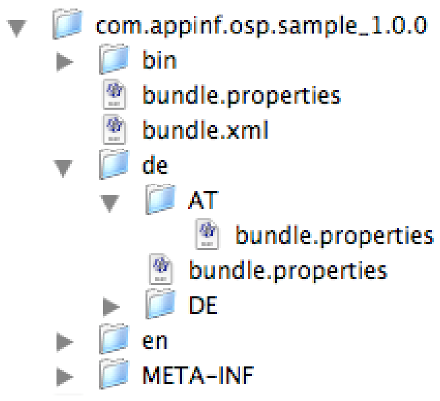


Figure 4: *Bundle resource localization*

If the resource `bundle.properties` is requested from the bundle, and the globally selected localization is German/Austria, the file `de/AT/bundle.properties` will be taken. If the localization is German/Switzerland, the file `de/bundle.properties` will be taken, since there is no `de/CH/bundle.properties` file available. If the localization is French/France, the file `bundle.properties` will be taken, since there neither is a localization for France, nor for French.

The lookup algorithm for localized resources can be summarized as follows:

First, look for the file at `<bundle-path>/<langugage>/<country>/<resource>`. If the file cannot be found there, look for `<bundle-path>/<language>/<resource>`. If the file again cannot be found there, look for `<bundle-path>/<resource>`. If the file still cannot be, the resource does not exist and an exception is thrown.

3.7.2 Bundle Lifecycle

The Bundle class also controls a bundle's lifecycle. The lifecycle has already been presented in Figure 2 in 2.3, but is again shown in Figure 5 for reference.

At startup, OSP looks for bundles at various places. For every bundle that is found, a `Bundle` object is created. The initial state of a bundle is *installed*. Once all bundles have been found, OSP tries to resolve each bundle. Resolving a bundle means checking whether all required bundles of a bundle are available. Every bundle that has been successfully resolved enters the *resolved* state. A resolved bundle will eventually be started. A bundle that is started is first put into *starting* state. Then, all required bundles are started as well. Once all required bundles have entered *active* state, the bundle's activator is invoked. When the activator completes, the bundle is finally put into *active* state. Eventually, at least at shutdown time, the bundle will be stopped. Stopping a bundle means putting the bundle into *stopping* state, invoking the bundle's activator, then putting the bundle into *resolved* state.

A bundle cannot be stopped while other bundles that depend upon this bundle are still running. At shutdown, OSP ensures that all bundles are shutdown in the correct order. A bundle in *resolved* or *installed* state can be uninstalled, which means it is completely removed from the system.

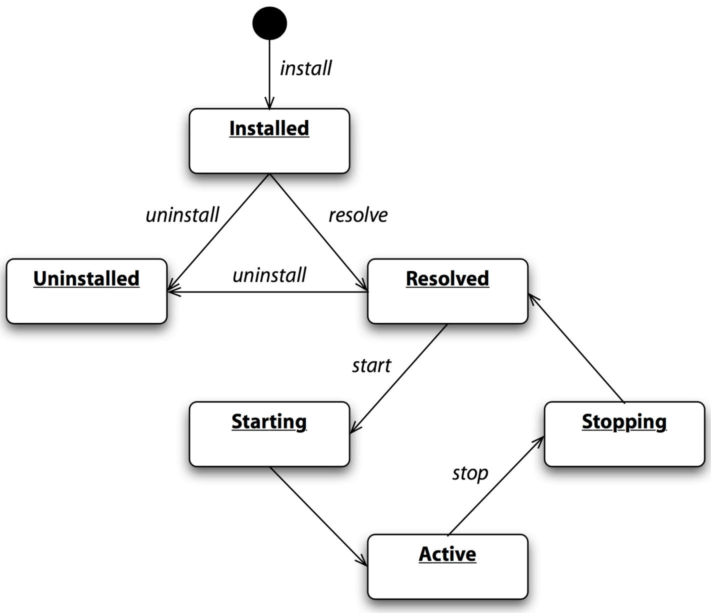


Figure 5: The bundle lifecycle

3.8 Code Cache

Shared libraries provided by a bundle are never loaded from the bundle directly. For bundles stored in a Zip file, this would not be possible anyway. After a bundle has been successfully resolved, all shared libraries of the bundle (for the current operating system and hardware architecture) are copied to a special *code cache* directory (unless they have already been copied, of course). This ensures, that the operating system's dynamic linker is able to find all required shared libraries in a certain place. The shared libraries of a bundle remain in the code cache until the bundle is uninstalled, or the code cache directory is cleared by request of an user or administrator.

Since the shared libraries of all bundles eventually end up in a common directory, great care must be taken when naming the shared libraries in bundles to avoid name conflicts. Therefore, the naming convention for bundles (reverse domain name) should also be used for shared libraries, if possible.

3.9 Bundle Activator

The *bundle activator* class is the code entry point for every bundle contributing executable code and requiring special actions upon startup or shutdown. The interface `BundleActivator` is shown below. Every class acting as bundle activator for a bundle must be derived from this class.

```
class BundleActivator
{
public:
    virtual void start(BundleContext::Ptr pContext) = 0;
        /// Called during the “starting” phase of the bundle, after all
        /// dependencies have been resolved.

    virtual void stop(BundleContext::Ptr pContext) = 0;
        /// Called during the “stopping” phase of the bundle.
};
```

The bundle activator is the only class loaded from the bundle by the OSP framework. Usually, no other dynamic class loading mechanism is needed, since the class loader is responsible for registering all services provided by the bundle with the service registry.

If a bundle does not provide a bundle activator, there is no way for the bundle to provide services to other bundles via the service registry. The only thing the bundle can do is providing common code in shared libraries to other bundles.

3.10 Bundle Context

The *bundle context* gives the bundle activator access to the bundle’s runtime environment. Via the bundle context, the bundle activator can obtain a pointer to the bundle object, to system-wide configuration properties, and to the service registry. The bundle context can also be used to find other bundles in the system.

The bundle context is valid during the entire lifetime of the bundle (until the bundle activator’s `stop()` member function completes), and the bundle activator can store it internally for later use.

```
class BundleContext: public RefCountedObject
{
public:
    typedef Poco::AutoPtr<BundleContext> Ptr;
    typedef const Ptr ConstPtr;

    Bundle::ConstPtr thisBundle() const;
        /// Returns a pointer to the bundle’s Bundle object.

    Bundle::ConstPtr findBundle(const std::string& name) const;
        /// Returns a pointer to the Bundle object for the
        /// bundle with the given name, if the bundle exists.
        /// Otherwise, returns NULL.

    ServiceRegistry& registry() const;
        /// Returns a reference to the system-wide service registry.

    BundleEvents& events() const;
        /// Returns a reference to the global bundle events object, which
        /// can be used to subscribe to events reporting state changes in
        /// installed bundles.

    Poco::Logger& logger() const;
        /// Returns a reference to the logger the activator can use to
        /// emit log messages.
};
```

3.10.1 Bundle Events

A bundle can opt in to be notified about state changes in other installed bundles by subscribing to the various events offered by `BundleEvents` object returned by `BundleContext::events()`. The POCO events framework is used for all OSP events.

```
class BundleEvents
{
public:
    Poco::BasicEvent<BundleEvent> bundleInstalled;
        /// Fired after a bundle has been successfully installed.

    Poco::BasicEvent<BundleEvent> bundleResolving;
        /// Fired when resolving a bundle starts. The bundle is still
        /// in “installed” state.

    Poco::BasicEvent<BundleEvent> bundleResolved;
        /// Fired when resolving a bundle is completed.

    Poco::BasicEvent<BundleEvent> bundleStarting;
        /// Fired when a bundle is started. The bundle is in
        /// “starting” state.

    Poco::BasicEvent<BundleEvent> bundleStarted;
        /// Fired when starting is completed and the bundle is in
        /// “active” state.

    Poco::BasicEvent<BundleEvent> bundleStopping;
        /// Fired when a bundle is stopped. The bundle is in
        /// “stopping” state.

    Poco::BasicEvent<BundleEvent> bundleStopped;
        /// Fired when stopping is completed and the bundle is
        /// in “resolved” state.

    Poco::BasicEvent<BundleEvent> bundleUninstalling;
        /// Fired when uninstalling a bundle starts. The bundle is in
        /// in “installed” state.

    Poco::BasicEvent<BundleEvent> bundleUninstalled;
        /// Fired when uninstalling a bundle is completed and the
        /// bundle is in “uninstalled” state.
};
```

All bundle events use the `BundleEvent` class, which is shown below.

```
class BundleEvent
{
public:
    enum EventKind
    {
        EV_BUNDLE_INSTALLED,
        EV_BUNDLE_RESOLVING,
        EV_BUNDLE_RESOLVED,
        EV_BUNDLE_STARTING,
        EV_BUNDLE_STARTED,
        EV_BUNDLE_STOPPING,
        EV_BUNDLE_STOPPED,
        EV_BUNDLE_UNINSTALLING,
        EV_BUNDLE_UNINSTALLED
    };

    Bundle::ConstPtr bundle();
        /// Returns the bundle that caused the event.

    EventKind what() const;
        /// Returns the kind of the event.
};
```

3.10.2 Logging

The OSP framework supplies the bundle activator with a `Poco::Logger` object which the bundle activator can use to emit log messages. The bundle should use this logger rather than defining its own. The logger's name is `osp.bundle.`*`bundle-name`*, where bundle-name is the symbolic name of the bundle.

3.11 Bundle Loader

The `BundleLoader` class is responsible for creating `Bundle` objects from their file system representation. It also maintains the association between `BundleContext` and `Bundle` objects and ensures that the `BundleActivator` is loaded (from its shared library) and invoked for every bundle at the appropriate times.

3.12 Bundle Repository

The bundle repository manages a directory in the file system containing bundles. At iterates over all files and directories in a specific directory and tries to create `Bundle` objects from them, using the bundle loader.

The `BundleRepository` class also detects and resolves bundle version conflicts – two or more different versions of a bundle are installed. In such a case, it ensures that the most recent version of a bundle is loaded.

Finally, the `BundleRepository` class allows to install a new bundle into it, the bundle's contents read from an input stream.

```
class BundleRepository
{
public:
    void loadBundles();
        /// Loads all available bundles, using a BundleLoader.
        /// If two or more versions of a bundle are found, ensures that
        /// the latest version of the bundle is loaded.

    void installBundle(std::istream& istr);
        /// Reads a bundle archive file from the given stream and
        /// installs it in the repository.
};
```

4 Service Registry

The *service registry* allows a bundle to provide services to other bundles, and allows a bundle to find the services provided by other bundles.

A bundle can provide and register arbitrary services in the form of interface classes. Every service a bundle provides must be implemented in a class that is a derived class of `Service`. Service objects are reference counted, therefore, the Service class is derived from `Poco::RefCountedObject`. The `Service` class does not define any additional members.

```
class Service: public Poco::RefCountedObject
{
public:
    typedef Poco::AutoPtr<Service> Ptr;
    typedef const Ptr ConstPtr;
};
```

A service can be implemented and registered as a singleton, meaning there exists exactly one instance of the service providing services to all bundles in the system, or it can be implemented and registered using a service factory. A service factory provides a unique instance of the service class to everyone requesting an instance of the service.

If a service is to be registered using a service factory, a corresponding service factory object for the service, derived from `ServiceFactory`, must be registered instead of the service object itself. The interface of the `ServiceFactory` class is given in the following.

```
class ServiceFactory: public Service
{
public:
    virtual Service::Ptr createService(BundleContext* pBundleContext);
        /// Create a new instance of the service.
        /// The bundle context of the bundle requesting the service
        /// is given as argument.
};
```

4.1 ServiceRegistry Class

The `ServiceRegistry` class, which implements the service registry functionality, has the following interface.

```
class ServiceRegistry
{
public:
    Poco::BasicEvent<ServiceEvent> serviceRegistered;
        /// Fired when a new service has been registered.

    Poco::BasicEvent<ServiceEvent> serviceUnregistered;
        /// Fired when a service has been unregistered.
    ServiceRef::Ptr registerService(
        const std::string& name,
        Service::Ptr pService,
        const Properties& properties);
        /// Registers the service object given in pService under the given
        /// name and with the given service properties.
        ///
        /// Returns a ServiceRef object for the registered service
        /// that can later be used to update the service properties.
        ///
        /// Throws a Poco::ExistsException if a service with the same name
        /// has already been registered.

    void unregisterService(const std::string& name);
        /// Unregisters the service with the given name.
        ///
        /// Throws a Poco::NotFoundException if no such service has been
        /// registered.

    ServiceRef::ConstPtr findByName(const std::string& name) const;
        /// Looks for a service with the given name.
        /// Returns a pointer to the service information if the service
        /// exists, or a NULL pointer otherwise.

    std::size_t find(
        const std::string& query,
        std::vector<ServiceRef::ConstPtr>& services) const;
        /// Looks for a service with the given properties.
        /// The string given in query must be a valid query according
        /// to the service query language syntax.
        /// Returns the number of services matching the given query
        /// and fills the services vector with ServiceRef objects
        /// describing the found services.
};
```

The `ServiceRegistry` class uses the `ServiceRef` class, which is given below.

```
class ServiceRef: public Poco::RefCountedObject
{
public:
    const std::string& name() const;
        /// Returns the name of the service.

    const Properties& properties() const;
        /// Returns the service properties.

    Properties& properties();
        /// Returns the service properties.

    Service::Ptr instance() const;
        /// If a ServiceFactory has been registered for the service,
        /// creates and returns a new instance of the service class.
        /// Otherwise, returns the registered service object.
};
```

4.2 Registering Services

Services are registered with the `registerService()` member function of the `ServiceRegistry` class. Each service is registered under a unique name. To ensure uniqueness of the name, the service name should follow the same naming conventions as a bundle name. Besides a name, a service can also have an arbitrary number of service properties. The service properties are name-value pairs and can be freely defined by the service. Other bundles can use the service properties to learn about the capabilities of a service, or even search for a service by its properties.

4.2.1 Service Properties

The properties of a service are defined using the `Properties` class, which is a simple wrapper around `std::map<std::string, std::string>`. There is only one predefined property, `name`, which contains the name of the service and is automatically set by the framework.

4.2.2 Updating Services

Once a service has been registered, the service properties can be updated. This is done using the `ServiceRef` object returned by `ServiceRegistry::registerService()`.

4.3 Finding Services

Services can be found by name (`findByName()`) or by property (`find()`). A simple query language can be used for looking up a service by its properties.

4.3.1 Query Language

The query language is similar to C++ expressions and supports comparison (`==`, `!=`, `<`, `<=`, `>`, `>=`), regular expression matching (`=~`) and logical and/or operations (`&&`, `||`). Subexpressions can be grouped with parenthesis. The data types string, integer, float and boolean are supported. The simplified syntax for the query language is given in the following.

```
expr          ::= andExpr [“||” andExpr]
andExpr       ::= relExpr [“&&” relExpr]
relExpr       ::= [“!”] (id [relOp value | “=~” matchExpr]) | subExpr
subExpr       ::= “(“ expr “)”
relOp         ::= “==” | “!=” | “<” | “<=” | “>” | “>=”
value         ::= numLiteral | boolLiteral | stringLiteral
numLiteral    ::= [sign] digit*”.”digit*[“E”[“+” | “-“]digit*]
boolLiteral   ::= “true” | “false”
stringLiteral ::= ‘”’ char* ‘”’
matchExpr     ::= stringLiteral | regExpr
regExpr       ::= delim char+ delim /* valid Perl regular expression,
                                 enclosed in delim */
delim         ::= “/” | “#”
```

Examples for valid queries are given in the following.

- `name == “com.appinf.osp.sample.service”`
  a simple string comparison for equality.
- `majorVersion > 1 && minorVersion >= 5`
  numeric comparisons and logical AND.
- `name =~ “com.appinf.osp.*” && someProperty`
  simple pattern matching and test for existence of `someProperty`.
- `someProperty =~ /[0-9]+/`
  regular expression matching.

4.3.2 Service Reference

Information about a service is delivered in `ServiceRef` objects.

```
class ServiceRef
{
public:
    const std::string& name() const;
        /// Returns the name under which the service has been registered.

    const Properties& properties() const;
        /// Returns the service properties.

    Properties& properties();
        /// Returns the service properties.

    Service::Ptr instance() const;
        /// Returns an instance of the service.
        ///
        /// If a ServiceFactory has been registered for the service,
        /// creates and returns a new instance of the Service class.
        /// Otherwise, returns the registered Service object.
};
```

4.4 Service Events

The `ServiceRegistry` class provides a few events to inform interested bundles about changes in service registrations. The `ServiceRegistered` event is fired whenever a new service is registered. The `ServiceUnregistered` event is fired when a service is unregistered. All events use the `ServiceEvent` class as argument.

```
class ServiceEvent
{
public:
    enum EventKind
    {
        EV_SERVICE_REGISTERED,
        EV_SERVICE_UNREGISTERED
    };

    const ServiceInfo& service() const;
        /// Returns information about the service that caused the event.
    EventKind what() const;
        /// Returns the kind of the event.
};
```

5 Standard Services

5.1 Extension Point Service

The Extension Point Service allows a bundle to provide "hooks" that make it possible for other bundles to extend its functionality. An example would be a servlet engine implemented in a bundle, with actual servlets implemented in their own bundles. The actual extension mechanism and how it is implemented is entirely defined by a bundle. The extension point service only provides a standardized configuration mechanism for extension points.

The Extension Point Service is registered under the service name `osp.core.xp` and has the following interface:

```
class ExtensionPointService: public Service
{
public:
    void registerExtensionPoint(
        Bundle::ConstPtr pBundle,
        const std::string& name,
        ExtensionPoint::Ptr pExtensionPoint);
        /// Registers the given extension point, under the given name.
        /// If the bundle registering the extension point is stopped,
        /// the extension point will be automatically unregistered.

    void unregisterExtensionPoint(const std::string& name);
        /// Unregisters the extension point with the given name.
};
```

A bundle that wishes to register an extension point must provide a subclass of `ExtensionPoint`, which has the following interface:

```
class ExtensionPoint: public Poco::RefCountedObject
{
public:
    typedef Poco::AutoPtr<ExtensionPoint> Ptr;
    typedef const Ptr ConstPtr;

    void handleExtension(
        Bundle::ConstPtr pBundle,
        Poco::XML::Element* pExtensionElem);
        /// Handles an extension element in the bundle's
        /// extensions.xml configuration file.
};
```

A bundle wishing to provide an extension point can do so by registering an appropriate `ExtensionPoint` object with the `ExtensionPointService`. This is usually done in the bundle's `BundleActivator`.

A bundle wishing to implement an extension can do so by providing a special configuration file named `extensions.xml` in its bundle's root directory. The file has the following format:

```
<extensions>
    <extension point="name">
        <!-- bundle specific content -->
    </extension>
    <!-- optionally more extension elements -->
</extensions>
```

The actual contents of the `extension` element are defined and interpreted by the `ExtensionPoint` subclass that a bundle defines.

The `ExtensionPointService` registers itself for the `bundleStarted` event. Whenever a bundle is started, the service looks for a `extensions.xml` file in the bundle. If such a file is present, the file is parsed using the DOM parser, and for every `extension` element the corresponding `ExtensionPoint` object is invoked.

5.2 Persistency Service

The Persistency Service provides persistent storage in the system's file system to a bundle. The way the Persistency Service works is quite simple. The persistency service is configured with the path of a directory (persistency root), where the persistent (and transient) data of bundles shall be stored. For every bundle that requests persistent storage, a directory in the persistency root directory is created, having the same name as the bundle's symbolic name. Within that directory, two more directories are created: `persistent` and `transient`. The `transient` directory is cleared whenever the bundle owning it is stopped. The `persistent` directory is never cleared. Within the `persistent` and `transient` directories, a bundle can create arbitrary files and directories.

```
class PersistencyService: public Service
{
public:
    std::string transientDirectory(Bundle::ConstPtr pBundle);
        /// Returns the path to the bundle's transient data directory.
        ///
        /// If the directory does not exist, it is created.
        /// The directory is cleared whenever the bundle that requested
        /// it is stopped.

    std::string persistentDirectory(Bundle::ConstPtr pBundle);
        /// Returns the path to the bundle's persistent data directory.
        /// If the directory does not exist, it is created.
};
```

The Persistency Service is registered under the name `osp.core.persistency`.

5.3 Preferences Service

The Preferences Service provides an easy way for a bundle or service to retrieve and store configuration information. It also gives a bundle read-only access to the global application configuration.

```
class PreferencesService: public Service
{
public:
    Preferences::Ptr preferences(Bundle::ConstPtr pBundle);
        /// Returns the preferences object for the given bundle.

    Configuration::ConstPtr configuration();
        /// Returns the global application configuration.
};
```

A bundle accesses and modifies its preferences via the `Preferences` class:

```
class Preferences: public Poco::Util::AbstractConfiguration
{
public:
    typedef Poco::AutoPtr<Preferences> Ptr;
    typedef const Ptr ConstPtr;

    Poco::BasicEvent<PreferencesEvent> propertyChanged;

    void save();
        /// Saves the preferences to the file system.
};
```

The `propertyChanged` event uses the `PreferencesEvent` class defined below.

```
class PreferencesEvent
{
public:
    const std::string& name() const;
        /// Returns the name of the changed property.

    const std::string& oldValue() const;
        /// Returns the value of the property before the change.

    const std::string& newValue() const;
        /// Returns the new value of the property.
};
```

A bundle accesses the global configuration via the `Configuration` class:

```
class Configuration: public Poco::Util::AbstractConfiguration
{
public:
    typedef Poco::AutoPtr<Configuration> Ptr;
    typedef const Ptr ConstPtr;
};
```

The Preferences Service uses the Persistency Service to persistently store preferences in the form of a properties file.

Internally, the `Preferences` object uses a `PropertyFileConfiguration` object to manage the data. A `Preferences` object automatically persists itself to the file system before it is destroyed.

The Preferences Service is registered under the name `osp.core.preferences`.

5.4 Web Server Service

5.5 User Authentication and Authorization Service

5.6 Management Service

5.7 Remoting Service

The Remoting Service integrates the POCO Remoting framework into OSP. It consists multiple parts:

5.7.1 Remoting Bundle

The Remoting Bundle contains the Remoting shared library and ensures that the Remoting shared library can be found in the code repository. The bundle does not contain a `BundleActivator`.

5.7.2 Binary Transport Bundle

The Binary Transport Bundle contains the Remoting binary transport implementation. It also contains a `BundleActivator` that registers the transport with the Remoting framework.

5.7.3 SOAP Transport Bundle

The Binary Transport Bundle contains the Remoting SOAP transport implementation. It also contains a `BundleActivator` that registers the transport with the Remoting framework.

5.7.4 RemoteGen Extensions

The RemoteGen tool is extended with the capability to generate Service classes for both clients and servers. This is enabled by specifying the `--bundle` (or `/bundle`) option on the command line.

The client service class is a simple wrapper around generated client helper class, exposing its `findObject()` member function.

5.8 Zero Configuration Service