RFP 117 - Parameterized Services

Final

6 Pages

Abstract

Service Factories have been part of the OSGi specifications since release 1. However, the service factories were scope per bundle. This scoping is confusing to newcomers and requires that real factories are implemented with factory services. These factory services are such a common pattern in OSGi that this RFP investigates the need for a more general feature.

0 Document Information

0.1 Table of Contents

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 6.1.

```
Source code is shown in this typeface.
```

0.3 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	*NOV 24 2008*	*Peter Kriens, aQute, Peter.Kriens@aQute.biz*
Final	*04/20/10*	Updated from Mar 17 CPEG meeting. *BJ Hargrave, hargrave@us.ibm.com*

1 Introduction

This RFP investigates the need for a new model of service factories because the existing model of scoping the instances per bundle has shown some serious shortcomings. These shortcomings have led to systems that implement factories on top of OSGi using special factory services. These patterns are so common that the question arises if the OSGi should not address this pattern directly in the framework. This RFP investigates the issues around a different model of service factories.

2 Application Domain

The core service model of OSGi is about sharing services. It therefore has a *service registry*. A bundle can register an object with this registry and other bundles can then find these objects without requiring any a-priori knowledge of the registering bundle.

During the development of OSGi R1, it was noted that in many cases it was necessary to return a service that was specific for the requesting bundle. At least, it was found out that this could make the implementations of services significantly easier. For example, the HTTP service implementation must track registrations of servlets and resources. Being able to associate the registration requests directly with the requesting bundle greatly simplifies the required garbage collection when the requesting bundle is stopped. This resulted in the magic ServiceFactory interfaces. Any service registered with the service registry that implemented this interface changed the semantics of getting the service. Instead of returning the registered object, the framework would call the registering bundle for a new object. It provided the requesting bundle and the service registration as a parameter. After the Service Factory was introduced it was discovered that it actually was an absolute necessity because this was the only reliable way to get the Service Registration object. The registration of a service usually triggers the getting of the service before the Service Registration object is returned. Several crucial scenarios required access to this Service Registration in handling the service calls.

Once a requesting bundle no longer has a need for a service, it must return it by calling ungetservice();

The Service Factory became heavily used in lazy patterns because the registration of the service was disconnected from the creation of the instance. This allowed systems like Declarative Services to register a service on behalf of a third party while the third party did not even have a class loader created.

Service Factory services are scoped per bundle. That is, only the first time a bundle gets a service is the registering bundle asked to create a service. The framework hangs on to this instance and will return this same object in subsequent calls.

This model does not match the model for more general factories. In a more general non-singleton factory, each call must return a new instance. This pattern can be implemented by a service that acts as a factory. That is,to create an instance, a requesting bundle must first get a service and than call a specific method in that service to obtain an instance. The providing bundle must then track any provided instances and the requesting bundle must track that its instances remain valid by verifying that the factory service remains valid.

2.1 Terminology + Abbreviations

3 Problem Description

3.1 Impossible to customize a service for a specific use case

There are many patterns where it is necessary to get a unique instance per call that is not shared with any other bundle or even inside the bundle. For example, when one needs a thread from a thread pool, it is wrong to reuse the same thread. Each call, even from the same bundle, requires a unique thread instance. In the current model, this example would require a ThreadPool service that would have a getThread method.

3.2 Potentially different behavior if bundles are merged

When two bundles are merged the different parts of the bundle will suddenly share a single instance. This can subtly change the interaction.

3.3 Service based factories are cumbersome

At this moment, the only way to implement a genuine service factory is to register a Factory service. As in the thread example, it is possible to register a ThreadPool service and then let this service create the instances. However, this creates an unnecessary additional interface. It would be a lot simpler if the OSGi registry could abstract this concept.

Though an additional interface may sound like a small price to pay for this functionality, it tends to add up. Also, subsystems like iPOJO, Declarative Services, and Spring DM all invent their own factory mechanisms. By not having a standard for this need, we make it harder to collaborate on the same semantics.

4 Use Cases

4.1 Thread Pool

A Thread Pool service registers a factory service for threads. Clients of the thread pool create threads from this pool and delete them once they are done with them.

4.2 Http Servlet Factory

As an alternative to the Http Service, the Factory service could allow the creation of Http Name Handlers. In this model, the Http Service would register a Factory Service for HttpNameHandler services. A client would then create a name handler for a specific URI, providing a Servlet or resource for the content. The client could then delete the name handler if it no longer needed it.

5 Requirements

- A0001 – The Framework must provide a mechanism that allows a provider bundle to register a factory service that enables a new instance for each invocation of a get service method.
- A0002 – The instance creation method should use a different API from the normal getService to minimize confusion
- A0003 – The service factory must be implemented with the existing concepts of service lifecycle, ServiceReference, ServiceRegistration, ServiceListener and service hooks.
- A0004 – Clients must be able to garbage collect any instances when the provider unregisters the service. The instances must follow the same life cycle as the service factory.
- A0005 – Providers must be able to customize the requested instance based on the requesting bundle as well as any number of parameters given by the requesting bundle.
- A0006 – The traditional way of using services must remain possible. That is, getService must have well defined semantics, for example returning a generic factory interface. The interaction of “classic” services and parameterized services must be defined.

6 Document Support

6.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

6.2 Author's Address

Name	Peter Kriens
Company	aQute
Address	9c, Avenue St. Drezery
Voice	+33467542167
e-mail	Peter.Kriens@aQute.biz

6.3 End of Document