Device Interface Extensibility

Version: 1.0 | 01/24/2012

Status: PDR DRAFT

Copyright 2012 Barco Inc.

All rights reserved.

This document is protected by copyright and distributed under licenses restricting its use, copying, distribution and decompilation. No part of this document may be reproduced in any form by any means without prior written authorization of Barco.

Contents

[Document History 2](#_Toc346878913)

[Introduction 2](#_Toc346878914)

[Application Extensibility 3](#_Toc346878915)

[Service Provider Architecture 3](#_Toc346878916)

[SPI in Java 4](#_Toc346878917)

[SPI in Device Interface 4](#_Toc346878918)

[Creating new Service Provider 6](#_Toc346878919)

[Alternatives to Java SPI 6](#_Toc346878920)

# Document History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Reason for change |
| 1.0 | 01/24/2012 | NEHUT | Initial Draft |

# Introduction

This document provides a high level overview of the provision of extensibility in Device Interface

# Application Extensibility

An *extensible* application is one that can be extended easily without modifying its original code base. One can enhance its functionality with new plug-ins or modules. Developers, software vendors, and even customers can add new functionality or application programming interfaces (APIs) by simply adding a new Java Archive (JAR) file onto the application classpath or into an application-specific extension directory.

This document intends to discuss extensibility of Device Interface and how it is achieved using Service Provider Architecture

# Service Provider Architecture

A *service* is a set of programming interfaces and classes that provide access to some specific application functionality or feature.

The Service Provider Architecture comprises of

* Service Interface:
  + The set of public interfaces and abstract classes that a service defines. The SPI defines the classes and methods available to your application.
* Service Provider:
  + Implements the SPI. An application with extensible services will allow you, vendors, and perhaps even customers to add service providers without modifying the original application.
* Service Access API:
  + Clients use Service Access API to obtain an instance of the service. The service access API typically allows but does not require the client to specify some criteria for choosing a provider. In the absence of such a specification, the API returns an instance of a default implementation.

The following figure shows a conceptual behavior of SPI architecture:

Figure 1: Working of SPI Architecture

*Bind*

*Publish*

*Find*

## SPI in Java

In Java application SPI architecture can be achieved by

* Defining Service interface
  + This can be a simple interface or abstract class describing what the service provider need to implement
* Service Access provisioning
  + The [Java SE 6](http://www.oracle.com/technetwork/java/javase/overview/index.html) platform provides a new API java.util.ServiceLoader that helps you find, load, and use service providers.
  + The ServiceLoader class searches for service providers on your application's classpath. It loads them and allows your application to use the provider's APIs.
* Service Provider
  + Implements the Service Interface. For ServiceLoader to instantiate the provider, it must contain zero argument public constructor. If the provider does not contain the constructor, it will result in runtime exception.
  + Usually, this will be a JAR file to hold provider. To register the provider, you must create a provider configuration file in the JAR file's META-INF/services directory.
  + The configuration file name should be the fully qualified binary name of the service's type.
  + On separate lines within the file, list the fully qualified binary names of your concrete implementations. The file must be UTF-8 encoded. Additionally, you can include comments in the file by beginning the comment line with the # character.

## SPI in Device Interface

Following is the Service Interface

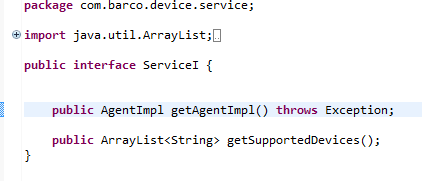


Figure 2: Service Interface

getSupportedDevices() method will return the array list of devices supported by the particular provider. This method helps to decide which provider to use at run time depending upon the device xml provided.

Device Interface also has a dummy service provider which acts as a default provider and will be called when no other provider is present, so the application runs at all times.

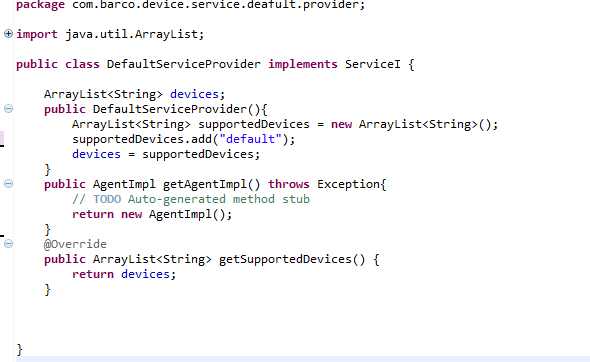


Figure 3: Default Service Provider

For ServiceLoader class to acknowledge presence of this provider, in the application META-INF folder ‘services’ folder is created and following line is added to the file named: com.barco.device.service.ServiceI

*com.barco.device.service.deafult.provider.DefaultServiceProvider #default provider*

As this provider is already in the application classpath, it will be loaded by the ServiceLoader class

Following is the Service Access API:

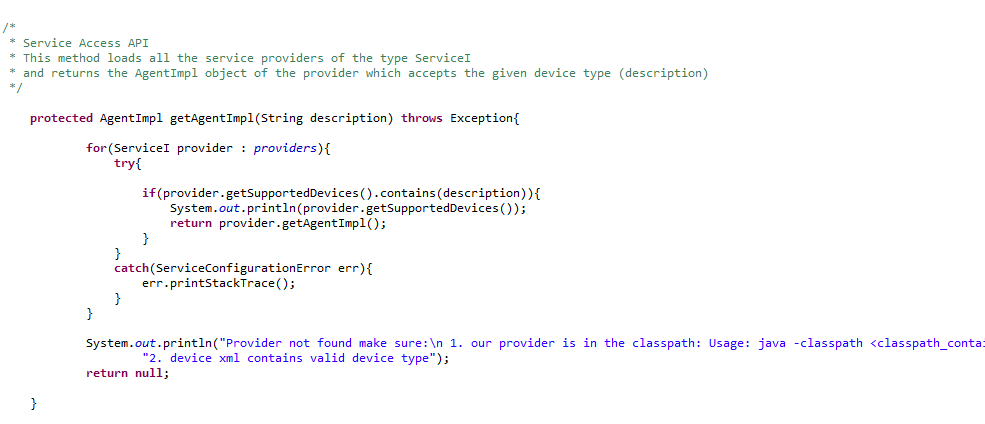


Figure 4: Service Access API

## Creating new Service Provider

Follow same procedure as in above example of default service provider. Make sure that the configuration file named com.barco.device.service.ServiceI is present in the META-INF/services folder of the jar

To bring the new provider in the Device Interface application’s class path use following command:

*java -classpath <classpath\_containing\_implementations\_of\_service> com.barco.device.test.DeviceTest <args>*

# Alternatives to Java SPI

There are other options to achieve extensibility of java application one of them is OSGI (an acronym for *Open Services Gateway Initiative*) framework.

* It is a module system and [service](http://en.wikipedia.org/wiki/Service_(systems_architecture)) platform for the [Java](http://en.wikipedia.org/wiki/Java_(programming_language)) programming language that implements a complete and dynamic [component model](http://en.wikipedia.org/wiki/Component_model)

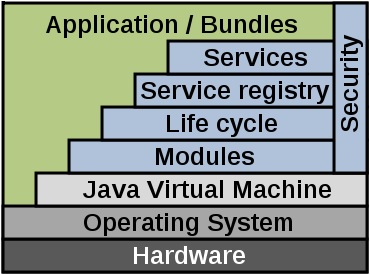


Figure 5: OSGI Framework

For an application to OSGI compliant all of its components need to be in OSGI framework. I.e. all the jars should be bundles. The services in OSGI are bundles published as services in the service registry of the OSGI framework.

Though OSGI is a good framework to implement extensibility, it particularly comes in handy when hot plugin is required, that is when new service provider needs to be plugged in without restarting rest of the application. This is not the requirement in device Interface.

For our existing application to be OSGI compliant lot of code rework will need to take place, which is not desirable as extensibility is easily achieved using SPI. When OSGI is used all the providers will be forced to use OSGI framework which might introduce additional learning curve for all the implementers.