The Apache Software Foundation, Inc.
Jakarta/Pluto

Architecture Overview

Let's begin by examining Pluto's architecture and underlying concepts. First, we briefly explain the portal that runs the RI, and see where to find a portlet container inside a portal architecture. Next, we investigate Pluto's architecture in detail. Last, we look at how it solves one challenging item of the portlet container: portlet deployment.

The portal

Pluto normally serves to show how the Portlet API works and offers developers a working example platform from which they can test their portlets. However, it's cumbersome to execute and test the portlet container without a driver, in this case, the portal. Pluto's simple portal component is built only on the portlet container's and the JSR 168's requirements. (In contrast, the more sophisticated, open source Apache Jetspeed project concentrates on the portal itself rather than the portlet container, and considers requirements from other groups.)

Figure 1 depicts the portal's basic architecture. The portal Web application processes the client request, retrieves the portlets on the user's current page, and then calls the portlet container to retrieve each portlet's content. The portal accesses the portlet container with the Portlet Container Invoker API, representing the portlet container's main interface supporting request-based methods to call portlets from a portal's viewpoint. The container's user must implement the portlet container's Container Provider SPI (Service Provider Interface) callback interface to get portal-related information. Finally, the portlet container calls all portlets via the Portlet API.

The portlet container

The portlet container, the portlets' runtime environment and a core component of each portal, requires knowledge about the portal itself and must reuse common code from it. Consequently, the portlet container remains completely separated from every other portal component. That said, you can embed the standalone portlet container in any portal by complying with the portlet container's requirements, such as implementing all SPIs.

The Portlet Container Invoker API, also called an entrance point, acts as the portlet container's main calling interface. The API combines a portlet container's lifecycle (init, destroy) with request-based calling methods (initPage(), performTitle(), portletService(), and so on). Because the portlet container calls a portlet in the end, the method signature resembles the Portlet API's main portlet interface, except that a portlet identifier must be passed. With this additional portlet identifier, the container can determine the portlet and call it accordingly.

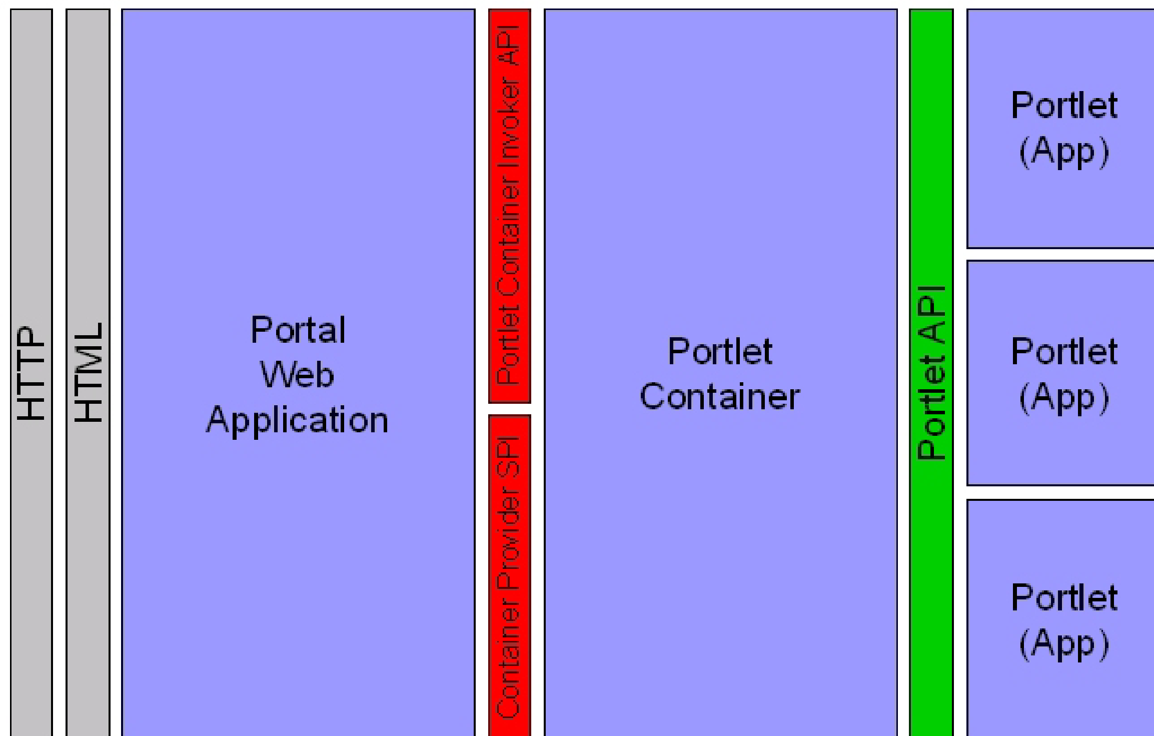

Figure 1. The simple portal included with Pluto.

Besides using the APIs to access the portlet container, the portal must implement SPIs defined for the portlet container. Therefore, the RI introduces container services: pluggable components that can be registered at the container to either extend or provide basic functionality. The RI includes the following built-in container services (the first four must be implemented to run the portlet container, while the fifth is optional):

- Information provider: Gives the portlet container information about the portal and its framework. Only known information or information that should be stored within the portal is present through this interface. Such information includes URL generation with navigational state, portlet context, portlet mode, and window-state handling
- Factory manager: Defines how to get an implementation through a factory. (A normal portal should already own such an implementation.)
- Log service: Defines a logging facility. (A normal portal should already own such an implementation.)
- Config service: Defines how to get configuration values. (A normal portal should already own such an implementation.)
- Property manager (optional): A property manager interface implementation lets a portal handle properties as defined in the JSR 168 specification

Strictly speaking, the Portlet Object Model also acts as an SPI, but has an exceptional position among the SPIs. Therefore, don't consider it part of the container services as it deals with all portlet objects and comprises a collection of interwoven interfaces.

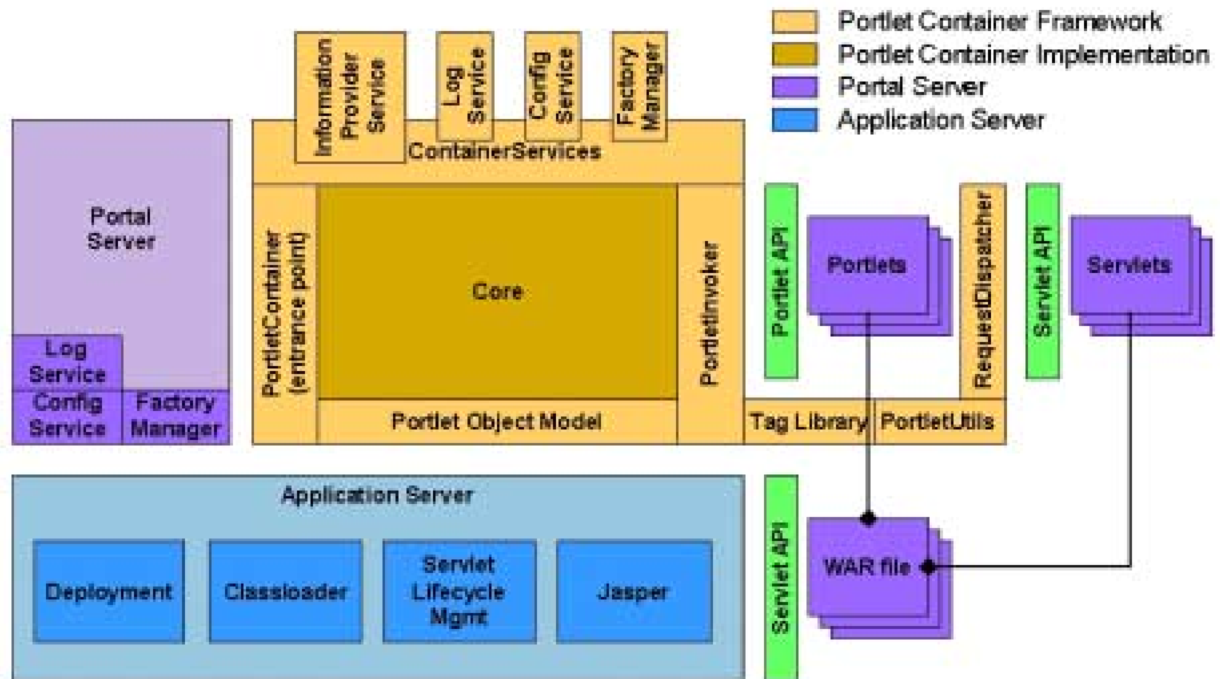

Figure 2. The portlet container's architecture.

Portlet deployment

The portlet container can leverage the servlet container's functionality, upon which the portlet container is built. To accomplish that, the portlet container must inject servlet artifacts into each portlet-application war file, as Figure 3 shows. The portlet component, Deployment, takes the original war file, then injects a new or modified web.xml and a servlet to wrap each portlet and uses it as a calling point. Then the portlet deployment passes the modified war file to the application server deployment, which deploys it into the application server's system. During the portlet's invocation, the portlet container calls the injected servlet as an entrance point into the deployed portlet war file.

Pluto and the WSRP standard

The JSR 168 aligns closely with the Web Services for Remote Portlets (WSRP) standard. Both standards, which emerged at the same time, released open source implementations capable of all necessary functions described in the respective specifications. As a mutual goal, both standards strive to work well together. As a result, the portlet container can run WSRP portlets as a consumer as well as a producer.

Pluto must be able to run multiple portlet containers in one portal. Consequently, Pluto's portlet container can be instantiated multiple times and, more importantly, it can be instrumented in different ways. Each portlet container, therefore, can use different implementations for SPIs.

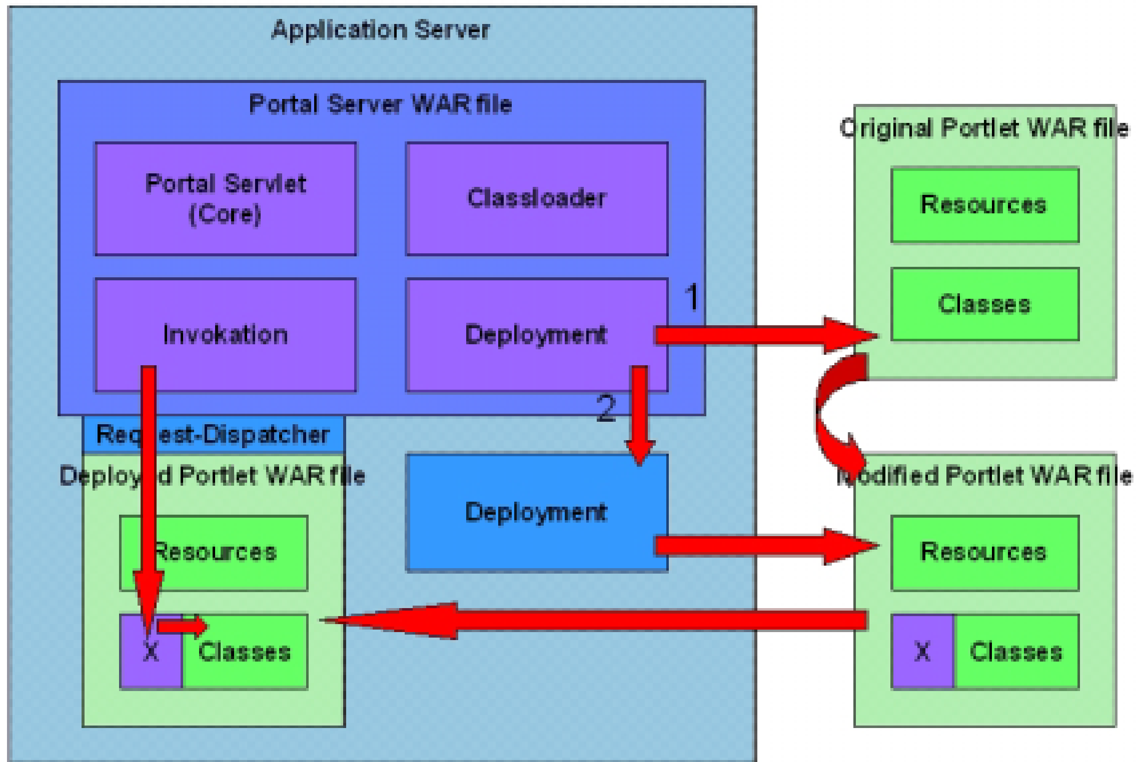

Figure 3. Portlet deployment in the RI.

Copyright © 2003, Apache Software Foundation