Portals allow users easy access to information by integrating heterogeneous applications or data sources in a consistent way. First-generation portals tend to present a monolithic software architecture that compromises portal development and maintenance. With roots in Internet portals such as My Yahoo (my.yahoo.com), second-generation portals let users create one or more personal pages composed of portlets — interactive Web mini-applications. Unlike first-generation portals, second-generation portals present a component-oriented architecture in which each portlet is a component that can be easily plugged into the portal, thus improving development, maintenance, and reusability.

This portal model is also beginning to appear in corporate intranets, giving users a personalized and restricted view of the company’s information. In the past three years, both the industry and open-source communities have provided software platforms — portal servers — to support the construction of second-generation portals. When portal servers first appeared, standards for developing portlets had not yet been proposed, making it impossible to consume remote portlets in a generic way or to deploy portlets in one portal server that were developed in a different one has been impossible. Two standards released in Fall 2003 — the Web Services for Remote Portlets (WSRP) and the Java portlet specification — address these problems. This article outlines these standards and presents a typical architecture for a standards-based portal.

**The Need for Standards**

Figure 1 shows the classic approach to implementing the portal shown in Figure 2. In this scenario, each portlet is a software component local to the portal, providing a complete user interface through which the portal can interact with the user and display content (provided by it or by an external service). The problem with this approach is that any portal wishing to consume the services provided by the BBC, Xignite, or others must reimplement each portlet’s user interface because such services provide only content.

It would be much more convenient if the remote services were remote portlets, returning HTML markup fragments rather than plain data. Likewise, if we used the same portal server to develop another portal that used the My Book-
marks and Tip of the Day portlets, we would have to redeploy the portlets in the new portal; if we used a different server, we would have to develop new portlets from scratch. Although unrealistic in this example, the situation is common in large corporations that have intranet portals for each division and wish to share portlets among them.

Figure 3 (next page) shows a solution to this problem: each remote portlet producer implements a standard Web service interface (defined in Web Service Definition Language) through which portals can interact with the remote producer’s portlets (currently the BBC and Xignite do not provide remote portlets, but rather services like those illustrated in Figure 1). Among the operations specified in the interface, one takes as a parameter the data corresponding to an event (for example, click on a portlet link or click on a portlet form-submit button) and returns the new markup. Whenever the user performs an interaction with a given portlet in a page, the portal calls on the corresponding producer’s Web service to get the new markup. To export some of its local portlets to other portals, a portal need only provide an implementation of the standard interface that wraps the portlets. As all portlet producers implement the same interface, a portal administrator could add a remote portlet to a portal without programming effort.

In September 2003, the Organization for the Advancement of Structured Information Standards (Oasis) released the first version of the Web Services for Remote Portlets (WSRP) standard (see the “Portal Resources” sidebar on p. 60), which specifies the interfaces a remote portlet producer must implement to allow another application (typically a portal) to consume its portlets, regardless of the technology the producer and consumer use (J2EE, .NET, and so on).

Developers should be able to take portlets developed in one portal server and deploy them in a different server, provided the servers are built with the same technology. Such an approach avoids dependence on portal server manufacturers. Thus, we need a standard API for each language used to develop portlets. It might seem that the interfaces specified in WSRP make up such an API. After all, many tools implement the standard WSDL-to-Java mapping, for example, and .NET provides a similar tool for its languages. However, several factors make this approach unsuitable:

- Most portlets will be local, in part because many will be domain specific, but also because using local resources is more efficient. However, WSRP was not designed to be an API for developing local portlets; it is for exporting a producer’s portlets to remote applications.
- WSRP interfaces contain many low-level details that make them difficult to implement directly.
- Implementing a portlet involves developing a component that generates markup (HTML, Wireless Markup Language, and so on) as in other Web applications.

Therefore, we need a portlet API that lets us use the same technologies we use to generate markup when developing normal Web applications (for
example, Java server pages or ASP.NET). In October 2003, the Java Community Process released the first version of the Java portlet specification (JSR 168), standardizing a Java API for implementing WSRP-compatible portlets that can be deployed in any standard Java portlet container.

**Web Services for Remote Portlets**

In the WSRP standard, *portlet modes* refer to the types of functionality a portlet can perform. The standard defines four modes: *view, edit, help,* and *preview.* It also defines *window states,* which indicate the amount of space a portlet will be assigned in the page. Window states can be *normal, minimized, maximized,* or *solo.*

**Service Interfaces**

A producer must implement two Web service interfaces — *service description* and *markup* — and may implement two others — *registration* and *portlet management.*

With the service description interface, consumers can discover metadata about the producer (supported custom portlet modes, window states, and so on) and the portlets it hosts (supported markup types, window title, description, and keywords, for example). The markup interface defines operations for requesting markup generation and for interacting with the markup. The operations getMarkup and performBlockingInteraction best represent this interface. Whenever the consumer needs to get the markup of a portlet (for example, when a user accesses a personal page at the beginning of the session), it invokes getMarkup. When the user performs an interaction with a portlet that might result in a change to a shared data area (such as a database), the consumer calls performBlockingInteraction on the portlet. This call is followed by a call to getMarkup on the portlet as well as the remaining portlets in that page from the same producer, because the change can affect them. When performing an isolated interaction with a portlet, the consumer invokes getMarkup to request the new markup of the portlet. To improve efficiency, the return value of getMarkup can indicate how long (in seconds) the markup can be cached.

The registration interface enables a relationship between a consumer and a producer (for billing, for example). A producer providing personalizable portlets must implement the portlet management interface, which allows creation and destruction of portlet instances, among other things. Each instance represents a unique personalization.

**Operations**

Figure 4 illustrates the interactions a user performs with the Stock Quote portlet in a page that also includes the Stock News portlet. I have augmented the Stock Quote portlet interface to include a stock-quote search facility. The user personalizes the Stock Quote portlet by adding a new stock symbol to the list of preferred stocks. The user clicks on the Stock Quote Edit button (e) in the top toolbar, then the Stock Quote Add button, and, finally, the Stock Quote View button (v).
new markup for the Stock Quote portlet. The getMarkup invocation specifies that the portlet should render markup in edit mode and assume a normal window state. When the user adds the new stock symbol, the portal calls performBlockingInteraction (2.1) on the Stock Quote portlet, followed by an invocation to getMarkup (2.2 and 2.3) on each portlet. This action modifies the Stock Quote portlet preferences, which could impact the markup generated by the Stock News portlet. Finally, when the user clicks on the Stock Quote portlet’s View button, the portal calls getMarkup (3.1) on that portlet, specifying the view mode.

Clicking the portlet’s Search button would cause an invocation of getMarkup only on that portlet because this action does not modify shared state.

It’s important to note that the internal URLs a portlet inserts into the generated markup cannot be direct links to the producer. First, the producer’s firewall can prevent client browsers from directly accessing the portlet. In addition, the portal creates the final markup returned to client browsers, and it might aggregate the markup of the remaining portlets in the page (as in the example) or include other information (such as a portal-specific header and footer). Therefore, the internal URLs generated by a portlet must be rewritten, so that they point to the portal rather than to the portlet itself. This way, the portal will intercept user interactions with a given portlet, translating them into appropriate invocations on producers. The WSRP standard defines two techniques for URL rewriting, one occurring in the consumer (the consumer rewrites the URLs embedded in the markup generated by the portlet) and the other in the producer (the consumer indicates to the producer how to generate URLs that point to the consumer).

Another interesting issue involves the markup returned by portlets. To allow portlets to generate consumer-independent markup, the WSRP specification defines a set of standard Cascading Stylesheet (CSS) styles. Portlets should use these styles, and consumers must provide a CSS with concrete definitions for each style, thus achieving a consistent and consumer-specific look-and-feel in the generated pages.

Java Portlet Specification

Similar to servlets, Java portlets run in a portlet container, a portal server component that provides portlets with a runtime environment. Although a portlet differs from a servlet, the Java portlet API (javax.portlet) borrows many concepts from the servlet API. In fact, a portlet container is an extension of a servlet container. Likewise, a portlet application is an extension of a Web application, which, apart from servlets, JSP pages, and other static resources (HTML files, images, and so on), also includes one or more portlets.

Figure 6 (next page) shows a code snippet of the Stock Quote portlet implementation. Typically, portlets are implemented by extending GenericPortlet, which provides a default implementation of the Portlet interface. As in WSRP, the portlet API distinguishes between action and render requests. Action requests modify the portlet’s state or cause a redirection, and are processed by redefining processAction. The remaining requests are render requests and are processed by the implementation of the render method provided by GenericPortlet. This method in turn invokes doDispatch to dispatch the request to doView, doEdit, or doHelp, depending on the selected portlet mode.

Only render methods can generate markup using the RenderResponse object. Thus, an action request directed to a portlet is always followed by a render request to the portlet and to the remaining portlets in the page whose content is not cached. A render request directed to a portlet also causes a
render request to the remaining portlets in the page whose content is not cached. The implementation of a render method (doView, doEdit or doHelp) can delegate markup generation to a JSP or a servlet.

In the example in Figure 6, processAction handles requests for adding or removing stock symbols from the list of preferred stocks. Each action is implemented in a helper method, which gets the portlet preferences and modifies the stockSymbols preference. The doView method handles two kinds of render requests: one for printing the preferred stock quotes and another for displaying a quote when a user types the stock symbol in the search form. Again, a helper method implements the processing of each event. For instance, showPreferredStockQuotes gets the preferred stock symbols, obtains the stock quote collection by using a facade of the business logic, adds it to the request as an attribute, and includes the content generated by a JSP displaying the stock quotes and the search form.

In the same way, doEdit gets the array of preferred stock symbols, adds it to the request, and includes the content generated by a JSP page that prints the personalization wizard. To facilitate the creation of action and render URLs from JSPs, the portlet API provides a tag library. This example also suggests the use of the model-view-controller (MVC) architectural pattern to implement portlet applications. The portlet class (controller) receives requests, calls on model facade methods, and delegates content generation to JSP pages.

**Standard-Based Portal Architecture**

Figure 7 (p. 60) shows the typical architecture of a Java portal server that supports WSRP and the
Java portlet specification. The portlet container component runs portlet applications conforming to the Java portlet API. The portal Web application component implements the portal use cases, such as sign in, sign up, select portlet layout, aggregate portlet responses, and so on. To interact with a portlet, the portal Web application component calls on a specific API provided by the portlet container.

The WSRP producer component provides an implementation of the WSRP interfaces, so other consumers can access local portlets. The portal server also provides a WSRP consumer component, implemented as a Java portlet, that acts as a generic proxy of any WSRP producer, allowing the portal to consume remote WSRP portlets.

Looking Ahead
WSRP and the Java portlet specification are in their initial versions. Over the next few years, an increasing number of portals and portlets will use both standards. Future improvements will emerge to overcome issues not currently addressed by the standard. Some improvements have already been suggested. One such improvement refers to the use of Universal Description, Discovery, and Integration (UDDI) for Web services and electronic business using XML (ebXML) registries in WSRP to allow producers to publish information about their

Figure 6 continued.

```java
private void addStockSymbol(ActionRequest request)
    throws PortletException, IOException {
    String stockSymbol = request.getParameter("stockSymbol");
    PortletPreferences prefs = request.getPreferences();
    String[] currentStockSymbols = prefs.getValues("stockSymbols", new String[0]);
    String[] newStockSymbols = addToArray(currentStockSymbols, stockSymbol);
    prefs.setValues("stockSymbols", newStockSymbols);
    prefs.store();
}

private void showPreferredStockQuotes(RenderRequest request,
    RenderResponse response) throws PortletException, IOException {
    PortletPreferences prefs = request.getPreferences();
    String[] stockSymbols = prefs.getValues("stockSymbols", new String[0]);
    StockQuoteFacade stockQuoteFacade = new StockQuoteFacade();
    Collection stockQuotes = stockQuoteFacade.findStockQuotes(stockSymbols);
    request.setAttribute("stockQuotes", stockQuotes);
    includeHTMLResponse(request, response, "/showStockQuotes.jsp");
}

private void includeHTMLResponse(RenderRequest request, RenderResponse response,
    String path) throws PortletException, IOException {
    response.setContentType("text/html");
    PortletRequestDispatcher rd = getPortletContext().getRequestDispatcher(path);
    rd.include(request, response);
}
[...]
}```
A number of interesting resources on portal standards and portal servers are available online.

**WSRP**

**Java Portlet Specification**
- Jakarta Pluto — http://jakarta.apache.org/pluto/
- Jakarta Struts Roadmap — http://jakarta.apache.org/struts/
- Jakstra [ ] [ ]

**Portal Resources**
- Portlet Open Source Trading (POST) — http://sourceforge.net/projects/portlet-opensrc

**Portal Servers**
- BEA WebLogic Portal — www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/portal
- Byelatech’s portal technology resources (collection of links) — www.byelatech.com/resources/portal_resources.asp
- IBM WebSphere Portal — www.ibm.com/software/genservers/portal
- Jakarta Jetspeed — http://jakarta.apache.org/jetspeed/site
- Metadot Portal Server — www.metadot.com/metadot
- Microsoft SharePoint Portal Server — www.microsoft.com/sharepoint
- Oracle Application Server — www.oracle.com/appserver

When they are fully specified and supported, Filters (similar to servlet filters) and event-based portlet intercommunication are among the future improvements to the Java portlet specification. The specification also will impact current MVC Web frameworks, such as Jakarta Struts. Even though the Java portlet API lets us structure a portlet application according to the MVC pattern, its support is more primitive than current MVC Web frameworks. Because these frameworks assume the servlet API, they’ll need adaptation before they’re suitable for building portlet applications. In fact, the Jakarta Struts team has announced support for portlet applications in a future version. And finally, WSRP-compatible APIs for other languages should appear in the next few years.

**Reference**

Fernando Bellas is an associate professor in the Department of Information and Communications Technologies at the University of A Coruña, Spain. His research interests include Web engineering, aspect-oriented development, and content adaptation for handheld devices. He has a PhD in computer science from University of A Coruña. He is member of the IEEE Computer Society and the ACM. Contact him at fbellas@udec.es.