A service-oriented programming model can simplify the development of program-to-human interactions by abstracting interfaces, standardizing messages, and aggregating independent information sources at the presentation layer under the control of a user or administrator. This fifth article in the series on the programming model for IBM's SOA covers services that are user facing and, as well as services provided by users through the Human Task Manager. Previous articles in this series introduced a language-neutral data access and programming model for Web services based on service-oriented architecture (SOA) concepts.

User interfaces for the service-oriented architecture

In the literature, Web services are generally discussed in the context of program-to-program interactions. However, this article demonstrates how an SOA programming model for Web services can also expedite the development of human-facing services. There are several approaches and programming models you can use to create user-facing UI components and dynamically aggregate them to present a consistent UI.

Moreover, the new Human Task Manager component of WebSphere Process Choreographer enables a startling new twist, by enabling a new kind of services: services that are provided by humans within the framework of a business process implemented in software. We'll cover that topic at the end of the article.

The Model-View-Controller (MVC) paradigm underlies most modern UI application frameworks. SOA operations provide the model layer, and UIs occupy the view layer. UI technologies can render information on devices ranging from gadgets and smart phones to browsers and rich clients capable of considerable client-side processing. Middleware and tools connect view-layer UI technologies to model-layer Web services and data.

In the SOA approach, the environments that host components are abstracted as containers, which provide well-known sets of infrastructure services. From a UI perspective, the three major containers for hosting client-side UI components are the:
• Basic Web browser.
• Web browser augmented with Java™Script and dynamic HTML.
• IBM Workplace™ Client Technology™ -- the Eclipse-rich client plus native IBM WebSphere® Application Server client support.

These containers can be augmented by supporting technologies such as servlets, JavaServer Pages (JSP), and JSP Tags; Struts for page sequencing; JavaServer Faces (JSF) for advanced page composition; and portlets to combine views of multiple applications on the same page.

Frameworks for UI development

UI development frameworks can simplify the creation of complex user-facing applications. The following UI frameworks are often used to create UI components:

• **Struts**, with the largest developer community and exceptional tools support, is an Apache open source project predating the Java Portlet Specification, JSR 168 (see Resources for a link to the Struts Web site). Struts is a multi-page MVC framework for server-based UI development using the servlet/JSP paradigm. A special version of the Struts, Version 1.1 library supports JSR 168 portlets on IBM WebSphere Portal.

• **JavaServer Faces** is an MVC realization for Java Web applications and builds incrementally on prior technologies. It is well suited for portlet development, offering portlets and servlets, state handling, validation, and eventing. A JSF page has one or more local models that interact with UI controls on the page. These controls render UI properties into outputs, and sophisticated logic ensures their presentation is at the "right" place. The client-side model can be wired into the Enterprise Service Bus to send and receive events.

• **Java Widget Library** (JWL), an extended widget set usable by portal and portlet programmers, adds JavaScript client-side processing to JSF and will be supported by IBM Rational® Suite® DevelopmentStudio. Updating the view locally on the client saves round trips to the server, shortens response time by orders of magnitude, and dramatically improves the user experience.

Portals provide first-class UI support. In the portal architecture, a **portlet** (typically developed using one of the above-mentioned UI frameworks) is the basic building block. This architecture lets developers focus on unique aspects of their application and delegate common functions for life cycle, per-user customization, aggregation, and integration with other components to the middleware.

The following sections describe portlet components for individual services and portals as a service aggregation mechanism.

Portlets for the service-oriented UI

The portlet component implements a standardized service interface and protocol. The Java Portlet Specification and Web Services for Remote Portlet (WSRP) standard define this interface for Java and Web services, respectively (see Resources for more information on WSRP). The two standards are similar enough that portlets written to either interface are interchangeable if the proper containers or proxies are present.
Java portlet example

Every Java portlet implements the portlet interface or extends a class that implements it. This interface defines the service contract between the portlet and its container and defines the portlet life cycle:

- Initializing the portlet and putting it into service (init method)
- Request handling (processAction and render methods)
- Taking the portlet out of service (destroy method)

During request handling, the portlet container calls the portlet's:

- processAction method to notify the portlet of a user action. Only one user-based action per client request is triggered. The portlet can issue a redirect, change its portlet mode or window state, or modify its state.
- render method to request a markup fragment.

Portlets can also invoke further services to perform desired functions. The Listing 1 sample uses a Web service to retrieve and display stock quotes for a specific user.

Listing 1. Stock quote portlet code sample

```java
public class StockQuotePortlet extends GenericPortlet {
    private ServiceManager serviceManager;
    public void init(PortletConfig config) throws PortletException {
        serviceManager = new ServiceManager();
    }
    public void doView(RenderRequest request, RenderResponse response) throws PortletException, IOException {
        response.setContentType("text/html");
        // invoke autogenerated wrapper to locate service
        NetXmethodsServicesStockquoteStockQuoteServiceLocator loc = new NetXmethodsServicesStockquoteStockQuoteServiceLocator();
        NetXmethodsServicesStockquoteStockQuotePortType port = loc.getNetXmethodsServicesStockquoteStockQuotePort();
        // loop through all stock quotes the user is interested in
        PortletPreferences prefs = request.getPreferences();
        Iterator quoteKeys = prefs.getMap().keys().iterator();
        String key; Float quote; StockBean quoteBean = new StockBean();
        while (quoteKeys.hasNext()) {
            key = quoteKeys.next();
            quote = port.getQuote(key);
            quoteBean.add(key, quote);
        }
        request.setAttribute("StockQuoteBean", quoteBean);
        // render stock quotes using a JSP
        PortletRequestDispatcher rd = getPortletContext().getRequestDispatcher("jsp/View.jsp");
        rd.include(request, response);
    }
}
```

This section illustrated how you can implement a UI service using the Java Portlet Specification, and how your portlet can invoke additional Web services. The next section shows how to publish your UI as a Web service using WSRP.

Web Services for Remote Portlets

WSRP is the standard for remote rendering of portlets, enabling a portal to aggregate content from multiple sources. WSRP extends the integration capabilities of Web services to presentation-oriented components, and exposes the view layer to sharing across platforms, implementation languages, and vendors. Content and application provider services can be discovered and plugged into standards-compliant applications without any extra programming effort.

Typical Web services employ a remote presentation paradigm, meaning all view logic executes on the client, whereas the application logic and the data layer (controller and model) reside on the server. By contrast, WSRP splits presentation between the client and the server in a distributed paradigm.
Figure 1. Data-oriented Web services compared to the WSRP presentation-oriented Web service

The image above depicts this difference. On the left is a typical data-oriented Web service that provides unformatted data; it must rely entirely on client-side rendering code for data presentation. (This implies the need to install and manage a client-side application component on the client.) On the right is a WSRP service; its distributed presentation logic splits the presentation task into:

- Generation of markup language.
- Aggregation of markup fragments into a Web page (not shown).
- Rendering of the markup language by a standard client-side container.

**WSRP example**

Listing 2 shows an example WSRP getMarkup request issued through Simple Object Access Protocol (SOAP) from a WSRP consumer.

**Listing 2. WSRP GetMarkup request issued through SOAP**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <getMarkup xmlns="urn:oasis:names:tc:wsrp:v1:types">
      <registrationContext>
        <registrationHandle>192.168.66.57_1096235652731_0</registrationHandle>
      </registrationContext>
      <portletContext>
        <portletHandle>0.1</portletHandle>
      </portletContext>
      <runtimeContext>
        <userAuthentication>wsrp:none</userAuthentication>
        <portletInstanceKey>ProxyTest_row1_col1_p1</portletInstanceKey>
        <namespacePrefix>Pluto_ProxyTest_row1_col1_p1_</namespacePrefix>
      </runtimeContext>
      <userContext>
        <userContextKey>dummyUserContextKey</userContextKey>
      </userContext>
      <markupParams>
        <secureClientCommunication>false</secureClientCommunication>
        <locales>en</locales>
        <getLocale>de</getLocale>
        <mimeTypes>text/html</mimeTypes>
        <mode>wsrp:view</mode>
        <windowState>wsrp:normal</windowState>
        <userAgent>WSRP4J Proxy Portlet</userAgent>
        <clientData/>
        <userAgent>WSRP4J Proxy Portlet</userAgent>
        <markupCharacterSets=UTF-8&markupCharacterSets=UTF-8&
        <validNewModes>wsrp:view</validNewModes>
        <validNewModes>wsrp:help</validNewModes>
        <validNewModes>wsrp:edit</validNewModes>
        <validNewWindowStates>wsrp:normal</validNewWindowStates>
        <validNewWindowStates>wsrp:maximized</validNewWindowStates>
        <validNewWindowStates>wsrp:minimized</validNewWindowStates>
    </getMarkup>
  </soapenv:Body>
</soapenv:Envelope>
```
The response of the WSRP producer to this request is an HTML fragment that the consumer (typically a portal) can aggregate into a full document, such as a portal page.

Instead of deploying each application or portlet on every server that intends to use it, there are obvious advantages to sharing applications across network boundaries. WSRP enables:

- **Easier administration** -- Instead of managing local deployments of pluggable components, portal administrators can browse a registry for WSRP services to offer. Users benefit from timely availability of new services and content integration on demand.
- **Load distribution** -- Loads are distributed across multiple servers.
- **Reduced infrastructure cost** -- Applications can share hosting infrastructure. For example, distributing just the presentation layer (through WSRP) of a back-end banking application preserves the application provider's secured computing environment, yet users can interact with the shared UI.
- **Control over content presentation** -- Content and application providers can vastly expand their reach to new users as portals redistribute content.

**Portals: dynamic aggregation for the service-oriented UI**

A portal's view-layer integration of several backend services' UIs into a centrally-managed UI can unify the fractured IT infrastructure and give users a single view of IT services with a single UI to master. Applications originally designed separately can be wired together into composite applications, enabling new functions. For example, an e-mail portlet wired to a collaboration portlet could filter the inbox to display received e-mail only when the sender is online and available for a chat -- a capability absent from both original applications.

A surprising consequence of the portal model is improved agility for On Demand Businesses. Administrators become application integrators who create new compound applications -- without programming -- by defining new pages, adding portlets to them, wiring the portlets together, and setting entitlements (access controls). A self-service portal lets users adapt their work environment to their unique needs. The portal architecture frees application developers to concentrate on building new business value.

In the future, portals may even be able to aggregate compound services and, thus, be able to aggregate UIs on a higher level. Portals may seamlessly integrate content from other portals, providing a horizontal, enterprise-wide integration of content.

**Bidirectional user interactions in the SOA**

WebSphere Portal is a gateway for user participation in the service oriented architecture. This participation can be bidirectional: users can both access and provide services through appropriate portlet interfaces.

**Human tasks in a workflow**

The concept of a computer user as a service provider is familiar if we go outside of the Web services realm: we have all asked the telephone operator to look up a phone number. The operator provides a service in response to a verbal service request.
Thus, a person can be a service provider, and the service invocation can be implemented as a Web service using a portal-based user interface. The portal provides information (input data) about a task to the user, then the user performs the task (service) and provides the result (output data) to the portal.

Familiar examples of human tasks within a workflow process include an optical technician inspecting eyeglasses manually before dispensing them to a customer, a safety check by a second mechanic after installing new automobile tires, and a bank manager confirming a customer's identity before allowing a large cash withdrawal. In a typical business process, some steps are performed by computers and some steps are performed by people. Figure 2 illustrates several combinations of human- and machine-implemented services.

Human Task Manager

The Human Task Manager (HTM) provides users participating in a business workflow process with a unified view of their work. By using the HTM, users can provide work and request work that needs to be done independently of the actual implementation of the corresponding service.

The HTM also provides automated Web services with a unified view of work performed by humans. A system can add a task to a task list to be performed by a human. Once the task is performed, the system is notified, can get the result of the task execution, and can continue with its processing.

The HTM manages and coordinates requests for services that are provided by users. The HTM maintains these lists of service requests that are available for a given user for execution as task lists. Task lists can be different for different users, depending on their role within the organization. For example, the task list for a content author contains requests for content to be created, while the task list of his manager contains approval requests for the content created by department members. If a user chooses to work on a certain task on his work list, he claims that task, performs the requested activity, and passes the result back to the HTM.

Users use the HTM to request that work be done. The requested service can be provided by another human or by an automated service. Tasks are typically long running, like Business Processes. The request for the service is independent of the current provider of the service. This allows changing the service provider without changing the service requester. Services that are manually provided by humans can be replaced by an automated service and vice versa. Users can query the status of their requested work using the HTM. The HTM provides the status for tasks they have initiated (either for other tasks performed by humans or for tasks performed by automated Web services).
Figure 2. Human Task Manager scenarios

Summary

Applying SOA concepts to the user interfaces and delegating common software life cycle functions to a UI container using the portal/portlet architecture can improve time-to-value for a software developer. The WSRP standard, delivering UI components through Web services, offers advantages to both the content provider and the content consumer, enabling a type of application aggregation without programming. Portal frameworks can aggregate portlets into a user interface in a dynamic manner. The participation of users in the SOA can be modelled as a Web service, with human interactions facilitated by the Human Task Manager.

© Copyright IBM Corporation 2005
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)