Prolifics
Global Technology Solutions Provider

Introducing REST Support for Panther Web Application Broker

Overview

If you already have a JAM or Panther application, you may want to expose some of its functionality to a non-Panther web or mobile application. Or, even if you already have a Panther Web application, you may want to have your user interface interact with it without having it return an HTML page that replaces the one the user is working with. For example, you might use a Panther application's JPL procedure that searches for a part or other item in your inventory. You can expose this functionality as a RESTful web service that can be seamless integrated into your non Panther web application for processing orders.

What is REST?

REST stands for "Representational State Transfer." It is a style of software architecture, not a specific technology. REST is often used as a light weight alternative to a traditional web services implementation. It does not use SOAP (Simple Object Access Protocol) or WSDL (Web Services Description Language), making it simpler to use. It is also easier to maintain a RESTful implementation. Traditional web services make use of a WSDL document that fully defines each web service. However, with REST, there is no well-defined service contract. This often allows for services to be augmented without the need to change existing clients.

When used on the web, REST uses the HTTP methods, POST, GET, PUT, and DELETE, and maps them to the CRUD operations, Create, Read, Update, and Delete, respectively. These are the operations one uses with a back end resource, such as a database. When using the RESTful style, you access the web based resource in a similar manner as you would access a database. A RESTful URI (Universal Resource Identifier) typically ends in a noun rather than a verb. Conceptually, one may think of a RESTful URI as referring to a database table and an HTTP GET operation on that table will retrieve all of the records in that table. Usually, one may append an item ID to that URI, and retrieve a single record associated with just that item ID. The data is typically returned as JSON (JavaScript Object Notation), or XML (Extensible Markup Language). Web Browsers can make these RESTful requests and update portions of a web page without the need to update the entire page. This is known as AJAX (Asynchronous JavaScript and XML). It is also possible for server side code, such as the Java code in a servlet, to make RESTful style requests to some remote server in order to gather and organize the data to be presented to the client. JSON is most readily consumed by client side JavaScript, and many existing JavaScript API's and UI component technologies are designed to work with JSON data.

Overview

REST support for Panther Web (aka Panther Web Application Broker) should not be confused with the technology Prolifics developed for Panther Enterprise Gateway (PEG). PEG is a preview technology based on the Panther Java Component Server (Panther JCO Server), and does not facilitate the use of RESTful URIs. It is essentially a technology for RMI/HTTP (Remote Method invocation over HTTP) that is unencumbered by SOAP and WSDL. For applications where SOAP and WSDL are preferred, Prolifics provides Panther JCO Application Server for Web Services, and also Panther Application Server for WebSphere and JEE.

REST support in Panther Web offers an alternative to these other Panther technologies for accessing legacy Panther functionality from Web, Mobile or any type of application that can act as an HTTP client. While traditional web services, based on WSDL and SOAP/HTTP, are appropriate for some applications, the RESTful style of access to backend resources is often a more suitable and simpler approach.

Background

First, lets define some terms associated with Panther Web technology:

Jserver - A Panther process that runs on the server machine. It opens Panther screens and executes JPL code, etc.

Requester - A light weight process supplied by Panther Web, usually a CGI program or Servlet that works with the HTTP Server. It receives HTTP requests, passes them to the Jserver, and returns responses from the Jserver.

Dispatcher - A process supplied by Panther Web that starts up JServers as needed and controls the traffic between the Requesters and JServer's.

The traditional architecture for a Panther Web application is one in which Panther screens are opened on the Jserver in order to provide both backend functionality and UI components. Statefullness and application navigation are also typically provided by the Jserver. The Jserver usually generates HTML documents from Panther Screens, either fully or by means of Panther HTML templates. The templates contain embedded template tags for which the Jserver generates content. That content is usually HTML, but may be just the data from Panther widgets or variables. In fact, Panther HTML templates need not contain any HTML at all. They may contain JSON or XML, for example, and that JSON or XML may contain embedded template tags, such as *{{value:variable}}*. This allows the application to embed only the Panther data contained within a widget or variable within the generated JSON or XML document that is the response to the HTTP request.

Clever Panther Web application developers have previously made use of this feature for making AJAX calls from JavaScript. Usually these would be in the style of a remote method invocation (RMI) and might require statefullness to be maintained by passing in Panther's *webid* parameter. This type of AJAX request would be

either an HTTP GET or a POST. DELETE and PUT were previously rejected by the Jserver. On a GET, parameters would be sent in the query string. However, one cannot send encrypted data in this way, even when SSL is used. On a POST, the data would need to be formatted as required for POSTed form data, with an HTTP *Content-Type* header of *'application/x-www-form-urlencoded'*. Another complication is that the returned data would always contain a Content-*Type* header of *'text/html'*. This made it difficult to work with client APIs that expected *'application/json'*.

REST for Panther Web

The latest version of Panther Web allows application code to set both the *Status* and *Content-type* headers using the Panther library function, *sm_web_set_http_header()*. Only if the *Content-type* header is not already set by application code will the Jserver return a *Content-type* of *'text/html'*. The ability to set the HTTP Status header for the returned data allows the application code to return standard REST status codes for success or failure.

Also, the Jserver will now accept PUT requests and perform processing in the same way as for POST. DELETE requests will now be accepted and perform processing in the same manner as for GET. Application code may inspect the CGI *REQUEST_METHOD* header in order to determine what functionality to perform. In JPL, for example, the built-in variable, *@cgi_request_method*, can be referenced.

The latest Requester Servlet contains special processing for REST and JSON. No such processing has been added to the other types of Requesters. Prolifics recommends using only the servlet form of the Requester in all cases. When JSON data is POSTed using a *Content-type header* of *'application/json'*, the Requester Servlet parses the JSON data and sends the *name/value* pairs to the Jserver as though it were normal field data for a POSTed form, and as though *'application/x-www-form-urlencoded'* were the *Content-type*. In order to do this mapping, structural elements of the JSON data are ignored. Names whose values are not other JSON nodes are treated as field names. These are the names in the leaf nodes. They may be the names of Panther widgets of any widget type that are valid Panther fields or arrays on the target screen. JSON arrays are copied to Panther arrays. A JSON array of objects, where each object contains several *name/value* pairs, can map to a set of Panther arrays having those same names. This is done such that corresponding occurrences represent a single record for each JSON object in the JSON array. The Panther arrays can be members of a grid widget, though it is not required. If they are members of a grid widget, the name of the grid widget is unused by the mapping mechanism.

Also, in order to better support RESTful style URIs, the Jserver will now accept URIs with multiple path components following a screen name. Application code that is used by the screen can parse the value of the variable, @cgi_path_info, in order to apply filters to the data resource represented by the screen. So, for example, if the 'parts' screen represents a table of parts by means of the grid widget it contains, then a URI ending in '*parts/500'* may represent a single part whose part number is 500. When the HTTP GET operation is performed on that URI, the *'parts'* screen is opened, and a screen entry function can find *'parts/500'* in the *@cgi_path_info* variable. It should then strip off the number following the last slash character, if any, so that it

can retrieve the data for only part number 500. It may even dynamically change the *html_template* property of the 'parts' screen, if necessary, in order to properly return JSON data for just the one part requested. Use of this feature is solely at the discretion of the Panther application developer. Query parameters in the URI may be used to accomplish the same thing.

Client State

One of the hallmarks of the REST style of architecture is that the server does not maintain client state information from one request to the next. This presents a hurdle for the migration of typical application code that is designed to function properly only within a particular state that had been achieved by earlier processing, such as, for example, a user having logged into the application. It is not efficient for each service request to require reauthentication and reauthorization, yet to do otherwise is inconsistent with REST constraints. For its part, a Jserver process is designed to be reusable, and must, therefore, not transfer any state information from one request to the next. In fact, some care must be taken by the Panther Web application developer that application code does not accidentally do this. After handling a request from one user, a microsecond later it may handle a request from a different user, and then the next request may be from the initial user once again. Traditional Panther Web applications usually deal with this by making use of Panther Web's server-side cache mechanism. Client-side caching is also possible by means of cookies and hidden HTML elements that the Jserver generates, which can then be POSTed back to the Jserver when the user interacts with the Panther generated UI. The later is not suitable for a REST and JSON web services implementation, however, and the former violates REST constraints.

In normal processing, the Jserver purges any application state it knows about after responding to a request, in order to be prepared for the next request. If server-side caching is enabled, and it receives a Panther *webid* from the client in a new HTTP request, it uses that *webid* to locate and load the cache containing the previous application state before processing the new request. For a GET operation, the *webid* may be passed as a query parameter, like, '*http://...myscreen?@webid=pLCaWhf-1545421892-192525-49-*'. For a POST, the POSTed screen would contain a hidden input element named '__server_data__', whose value contains the *webid* value. It looks something like this:

```
<input name=__server_data__ type=hidden value="pLCaWhf-1545421892-192525-49-">
```

There is nothing to prevent a GET operation from returning the webid within JSON data, and for the following POST of JSON data to provide that same webid in order to trigger the use of a server-side cache, assuming server-side caching is enabled. The POSTed data could look something like this:

```
{ "HolidayBonus" : 1000000, "__server_data__":"pLCaWhf-1545421892-192525-49-" }
```

This is not recommended. Note that query parameters are supported by the Jserver only for GET and DELETE

operations. For POST and PUT parameters should be contained in the JSON data.

With regard to login state, however, it is typical for REST servers to accept tokens in an HTTP *Authorization* header that allow the authentication and authorization state to be reinstated for each service request without there having to be any undue overhead, and without sacrificing security. Strictly speaking, however, if the server caches anything that allows it to automatically reauthenticate and reauthorize, it is violating a constraint of REST. Nevertheless, this one constraint violation is often done for improved performance.

REST support in Panther Web does not impose any such constraint violation. The application code can access the HTTP *Authorization* header using the variable, *@cgi_authorization*, and do whatever it chooses to do with that. However, the Requester Servlet can be configured to provide support for OAuth2, by means of Auth0, and this mechanism will cache the JWT (Java Web Token) so that it can be recognized again, before it expires. Thus, the authentication and authorization flow does not need to be repeated for each service request. One benefit of using Auth0 is that one can authenticate to various Identity Managers, including various Social Media platforms that support Open Identity. Authenticating to a Social Media platform is a way to provide single sign-on capability with other applications that do the same.