Call Control Web Service Plugins

The SipXrest container - a restful service container for exporting REST interfaces to sipXecs call control services

As sipXecs grows to integrate with other products, there is a need to export call control services using a web services model so that these services may be invoked by other components and from other products. An example of such a service would be the third party call controller. Call control services are traditionally written in sipXecs as stand alone processes. SipXrest does not aim to supplant this model. Rather, what we aim to do is to provide a web services interface to interact with such call control services that are useful to export for business integration purposes. Web services (and recently REST-based services)are exported by sipXconfig for configuration functions. We aim to do something similar for call control functions. This has worked well thus far by placing such integration code into sipxconfig. **SipXconfig** thus exports the third party call controller as a REST service in SIPXECS 4.0. However, as the number of these services grows, so do the associated problems of putting every bit of call control integration functionality into *sipXconfig*. As the complexity of the product grows, so does the number of such services and hence so do the number of integration issues.

The **SipXrest** service container attempts to solve this problem. Users write JAVA service plugins for the **sipXrest** container. Each plugin is a converged SIP/HTTP service. The code of a plugin is invoked via HTTP GET/POST or by SIP signaling. It may in turn interact with other SIP components and accomplish the desired task and present a result to the user. The SIP component of a plugin is a user agent (UAC or UAS) that provides a specialized function. HTTP services run using the *Restlet* model. All plugins run in the same JVM.

SipXrest is a simple (almost trivial) converged container. (It does not aim to be a re-invention of the **JAIN-SLEE** or anything nearly that complicated.) There is a single SIP stack and a single HTTP server that is shared by all the plugins that run in **sipXrest. SipXrest** provides no service isolation of plugins (they all use the same classloader) and provides only very simple lifecycle management. Plugin authors are expected to be well mannered and not trod on each other's static data. Services are created (instantiated) when they are loaded. Services may initiate SIP signaling via the SIP Service bean that is shared by all such services bundled as part of the service container. Each service may define its own security policy and HTTP authentication method for interaction with the outside world. SipXrest will access the validusers.xml account database and take care of HTTP and SIP request authentication. A simple *SipHelper* class is provided to the SipListner part of the service (if one exists) and exposes a subset of the JAIN-SIP *SipProvider* functionality. This allows plugins to create SIP protocol objects such as SIP Transactions and SIP Dialogs. The supported SIP signaling model is dialog stateful. Each Plugin may have a single SIP endpoint. The goal is to simplify the task of building such converged services while allowing them to co-exist in a single JVM and constraining them to a single style of SIP service (i.e. the SIP UA). No SIP proxy or SIP back to back user agent function is supported.

How to write a plugin

A plugin is a Jar file. It is placed in the directory ${prefix}/share/java/sipXecs/sipXrest/plugin. Each plugin has a file : plugin.xml that indicates the behavior of the plugin,
its main (Plugin) class and other aspects of its behavior.

Plugin Jar Structure
-java package
-plugin.xml

Note: plugin.xml is NOT *in the java package*

Here is what that plugin.xml descriptor file contains :

plugin-class : The fully qualified class path of the main plugin class. This class must implement the Plugin interface.

uri-prefix : The HTTP URI prefix to invoke this REST service. URI prefixes must be unique. They do not include the transport part. For example the third party call controller uses the URI prefix /callcontroller/. It can be invoked using either HTTP or HTTPS transport.

security-level : The security level. Can be LOCAL-AND-REMOTE or
LOCAL-ONLY. A LOCAL-ONLY service can only be accessed from the sipXecs proxy domain. LOCAL-AND-REMOTE services can be accessed using HTTP interactions from outside the proxy domain.

remote-authentication-method : This determines how remote callers (i.e. those outside
the sipXecs domain) are authenticated. The available choices are HTTP-DIGEST or HTTPS-BASIC. If the service has a security of LOCAL-ONLY then this field is ignored.

service-description: HTML description of what this service does. The sipXrest container consolidates these and presents an information page.

sip-listener-class : Fully qualified class path of the SipListener (if any) for this service. If the service does not include a SIP stack you can leave out this field. The SipListener
must extend the AbstractSipListener. It is similar in function to the JAIN-SIP listener.

sip-convergence-name : This is the "convergence name" of the service. This field is relevant if the Dialog that instantiates the service will be instantiated by
an out of Dialog inbound INVITE (i.e. runs as a UAS). The userName part of the request URI is used to identify the appropriate service to instantiate. If the SIP service is not triggered \by an incoming SIP request (i.e works as a UAC) you can leave the field blank.

The Plugin Class

The main plugin class must extend the *org.sipfoundry.sipxrest.Plugin* abstract class. The abstract methods of this abstract class that must be implemented are :

```
   /**
    * Return the identity of the agent that is making the request.
    * This is an entry in the validusers.xml database or it can be
    * a special user ID such as ~~id~watcher which is not recorded
    * in the special user database.
    * @param request
    * @return
    */
   public abstract String getAgent(Request request);
   /**
    * Attach the filter and context to the router and route the request
    * to the actual restlet. The Filter is the standard security policy.
    * @param filter
    * @param context
    * @param router
    */
     public abstract void attachContext(Filter filter, Context context, Router router);
```

Here is a very simple plugin :

```
public class PluginA extends Plugin {
  @Override
  public void attachContext(Filter filter, Context context, Router router) {
     filter.setNext(new RestletA());
     Route route = router.attach(getMetaInf().getUriPrefix() + "/{param}",filter);
     route.extractQuery("agent", "agent", true);
  }
  @Override
  public String getAgent(Request request) {
      return (String) request.getAttributes().get("agent");
  }
}
```

The RestletA that does the actual work is

```
public class RestletA extends Restlet {
  public RestletA () {

  }
  @Override
  public void handle(Request request, Response response) {
      System.out.println("RestletA: got a request ");
      System.out.println("Parameter A is " + request.getAttributes().get("param"));
      response.setStatus(Status.SUCCESS_OK);
  }
}
```

Note that the *RestletA* is completely oblivious to any security policy or the url prefix that is used to invoke it.

Here is the plugin.xml file that goes along with this plugin that defines such things :

```
<?xml version="1.0" ?>
<rest-service xmlns="http://www.sipfoundry.org/sipX/schema/xml/sipxrest-service-00-00">
<plugin-class>org.sipfoundry.sipxrest.testa.PluginA</plugin-class>
 <security-level>LOCAL-AND-REMOTE</security-level>
 <uri-prefix>/testplugin/a</uri-prefix>
 <service-description>
     This is test plugin A. You can invoke it using the URI
     testplugin/a?agent=user1 where user1 is a valid user in your user database.
  </service-description>
</rest-service>
```

You compile these java classes, drop them in a jar along with the plugin.xml file and drop the jar in the plugin directory of sipxrest. Restart sipxrest and you are off and running.

Note that we have cleanly separated deployment and security policy issues from what the plugin itself is supposed to be doing, thus liberating the mind of the programmer to indulge in creative activity. If your plugin does not need sip stack support thats all there is to it.

Plugins that need SIP stack support

Life gets a bit more complicated when your plugin needs SIP support. SipXrest bundles a JAIN-SIP stack instance. Each plugin can have a SipListener class that extends the abstract org.sipfoundry.sipxrest.AbstractSipListener class (unlike JAIN-SIP this is not an interface).

Unlike JAIN-SIP, the container supports multiple such Listener classes. There can be one such listener class for each plugin. The container takes care of multiplexing requests and responses and directing these requests and responses to the listeners that expect to see them. To accomplish this goal, the container does not give you direct access to the JAIN-SIP provider. Instead, each plugin is provided with a *org.sipfoundry.sipxrest.SipHelper* instance which works similarly to the JAIN-SIP SipProvider. The *SipHelper* is used to create Client and Server SIP Transactions. The *SipHelper* also provies other utility functions. The JAIN SIP Provider is not directly accessible (or at least it should not be directly accessed) to perform such functions. Using the SipHelper class allows the container to associate JAIN-SIP protocol objects such as transactions and dialogs with specific plugins and route request and response events to those plugins. This works well for services that work as UACs.

For Services that work as UAS, the SipListener has to be associated with an incoming out of dialog INVITE. All such requests arrive over a single port. Hence we need additional information to uniquely identify the service. Each plugin service that needs to function as a UAS has a "convergence-name" that is used for this purpose. The inbound out-of dialog INVITE that is directed to this plugin must have the user part of the request URI matching the convergence name. Convergence names must be uniquely chosen to avoid clashes.

The SIP support in the container will be evolved as needs dictate.

Port and process management

The whole container has a single SIP port, one HTTP port and one HTTPS port. These must be allocated by sipxconfig. The container is hence suitable for services that are structured as simple endpoints or plain HTTP services without SIP signaling requiremnts. We do not envision building back to back user agents that live in this container.