Apache Felix HTTP Service

Apache Felix HTTP Service

This is an implementation of the HTTP Service Specification as described in chapter 102 of the OSGi Compendium. The goal is to provide a standard and simplified way to register servlets and resources in a Servlet container, and to associate them with URIs. It also implement a non-standard extension for registering servlet filters as well as a whiteboard implementation. Complete set of features:

- Standard HTTP Service implementation.
- Extended HTTP Service implementation that allows for servlet filter registration.
- Run either with Jetty or inside your own application server using the servlet bridge.
- A whiteboard implementation for easy registration of servlets and filters.
- One complete bundle that includes everything to simplify deployment.

Installing

The Apache Felix HTTP Service project includes several bundles.

- org.apache.felix.http.jetty - HTTP Service implementation that is embedding Jetty server.
- org.apache.felix.http.whiteboard - Whiteboard implementation that uses any HTTP Service implementation.
- org.apache.felix.http.bridge - HTTP Service implementation that uses the host applicaiton server (bridged mode). Must be used with proxy.
- org.apache.felix.http.bundle - All in one bundle that includes all of the above.
- org.apache.felix.http.proxy - Proxy that is needed inside WAR when deployed inside an application server.

So, in most cases you could just use **org.apache.felix.http.bundle** and forget about all the other ones.

Using the HttpService

The main components provided by the Apache Felix HTTP Service bundle are:

- `HttpService` - Service used to dynamically register resources and servlets
- `HttpContext` - Additional (optional) component to handle authentication, resource and mime type mappings

Servlets created for the OSGi HTTP service don't need to have any reference to the OSGi specification (they only need to conform to the Servlet specification), like in the example:

```
public class HelloWorld extends HttpServlet
{
  @Override
  protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException
  {
    resp.getWriter().write("Hello World");
  }
}
```

To register a Servlet and map it to a URI, you need to retrieve the `HttpService` and call its `registerServlet` method:

```
public class Activator implements BundleActivator
{
  public void start(BundleContext context) throws Exception
  {
    ServiceReference sRef = context.getServiceReference(HttpService.class.getName());
    if (sRef != null)
    {
      HttpService service = (HttpService) context.getService(sRef);
      service.registerServlet("/hello", new HelloWorld(), null, null);
    }
  }
}
```

In the same way, you can unregister a Servlet (for instance, in the `stop` method of the Bundle Activator) calling the `HttpService.unregister` method.

As you notice in the example above, the `registerServlet` method accepts four parameters:

- the Servlet alias
- the Servlet instance
- an additional configuration Map
- an HttpContext

The Servlet alias must begin with a slash and must not end with a slash. When a request is processed, the HTTP Service will try to exact match the requested URI with a registered Servlet. If not existent, it will remove the last '/' in the URI and everything that follows, and try to match the remaining part, and so on.

An additional configuration Map can be optionally specified; if present, all the parameters contained will be copied in the ServletContext object.

Finally, an HttpContext object can be optionally specified to handle authentication, mime type and resource mapping. The `HttpContext` interface is quite simple:

```
public interface HttpContext
{
  String getMimeType(java.lang.String name); //Returns the mime type of the specified resource
  URL getResource(java.lang.String name);   //Returns the URL to retrieve the specified resource
  boolean handleSecurity(HttpServletRequest request, HttpServletResponse response); //Manages security for the
specified request
}
```

The use of a custom HttpContext is typical when you want to serve static contents with the HTTP Service. Let's see first the simplest example of resource registration (without `HttpContext`)

```
public class Activator implements BundleActivator
{
  public void start(BundleContext context) throws Exception
  {
    ServiceReference sRef = context.getServiceReference(HttpService.class.getName());
    if (sRef != null)
    {
      HttpService service = (HttpService) context.getService(sRef);
      service.registerResources("/static", "/etc/www", null);
    }
  }
}
```

As a result of the `service.registerResources("/static", "/etc/www", null)` code, all the files available under `/etc/www` will be exposed under `/static` (f.i. http://localhost:8080/static/001.jpg will render the /etc/www/001.jpg). However, the example above can be simplistic in practice; the HttpContext object is the solution to customize the resource handling.

For instance, you can set the define more complex URI to file mappings overriding the `HttpContext.getResource` method, or the correct mime type implementing the method `HttpContext.getMimeType` like in the example:

```
//....

public String getMimeType(String file)
{
  if (file.endsWith(".jpg")
  {
    return "image/jpeg";
  }
  else if (file.endsWith(".png"))
  {
    return "image/png";
  }
  else
  {
    return "text/html";
  }
}

//....
```

If you implement a customized HttpContext object, don't forget to specify it as third parameter of the `registerResources` method invocation:

```
public class Activator implements BundleActivator
{
  public void start(BundleContext context) throws Exception
  {
    ServiceReference sRef = context.getServiceReference(HttpService.class.getName());
    if (sRef != null)
    {
      HttpService service = (HttpService) context.getService(sRef);
      HttpContext myHttpContext = new MyHttpContext());
      service.registerResources("/static", "/etc/www", myHttpContext);
    }
  }
}
```

Using the ExtHttpService

To be able to register filters, it is possible to get hold of `org.apache.felix.http.api.ExtHttpService`. This is exported by both jetty and the bridged implementation. Let's see the simplest example of a filter registration.

```
public class Activator implements BundleActivator
{
  public void start(BundleContext context) throws Exception
  {
    ServiceReference sRef = context.getServiceReference(ExtHttpService.class.getName());
    if (sRef != null)
    {
      ExtHttpService service = (ExtHttpService) context.getService(sRef);
      service.registerFilter(new HelloWorldFilter(), "/hello/.*", null, 0, null);
    }
  }
}
```

Notice the pattern for filters is using regular expressions. So `.*` is the same as a simple `*` using standard servlet patterns.

Using the Whiteboard

The whiteboard implementation simplifies the task of registering servlets and filters. A servlet (or filter) can be registered by exporting it as a service. The whiteboard implementation detects all `javax.servlet.Servlet`, `javax.servlet.Filter` and `org.osgi.service.http.HttpContext` services with the right service properties. Let us illustrate the usage by registering a servlet:

```
public class Activator implements BundleActivator
{
  private ServiceRegistration registration;

  public void start(BundleContext context) throws Exception
  {
    Hashtable props = new Hashtable();
    props.put("alias", "/hello");
    props.put("init.message", "Hello World!");

    this.registration = context.registerService(Servlet.class.getName(), new HelloWorldServlet(), props);
  }

  public void stop(BundleContext context) throws Exception
  {
    this.registration.unregister();
  }
}
```

Servlet service properties:

- `alias` - Servlet alias to register with.
- `contextId` - Id of context to register with.
- `init.*` - Servlet initialization values.

Filter service properties:

- `pattern` - Regular expression pattern to register filter with.
- `contextId` - Id of context to register with.
- `service.ranking` - Where in the chain this filter should be placed.
- `init.*` - Filter initialization values.

HttpContext service properties:

- `contextId` - Id of context to be referenced by a servlet or filter service.

Using the Servlet Bridge

The servlet bridge is used if you want to use the Http service inside a WAR deployed on a 3rd part applicaiton server. A little setup is needed for this to work:

- Deploy `org.apache.felix.http.proxy` jar file inside the web applicaiton (WEB-INF/lib).
- In a startup listener (like ServletContextListener) set the BundleContext as a servlet context attribute (see example).
- Define `org.apache.felix.http.proxy.ProxyServlet` inside your `web.xml` and register it to serve on all requests `/*` (see example).
- Define `org.apache.felix.http.proxy.ProxyListener` as a `<listener>` in your `web.xml` to allow HTTP Session related events to be forwarded (see the section of Servlet API Event forwarding below and example).
- Be sure to add `javax.servlet;javax.servlet.http;version=2.5` to OSGi system packages (`org.osgi.framework.system.packages`).
- Deploy `org.apache.felix.http.bridge` (or `org.apache.felix.http.bundle`) inside the OSGi framework.

A detailed example can be found here.

Configuration Properties

The service can both be configured using OSGi environment properties and using Configuration Admin. The service PID for this service is "`org.apache.felix.http`". If you use both methods, Configuration Admin takes precedence. The following properties can be used (some legacy property names still exist but are not documented here on purpose):

- `org.osgi.service.http.port` - The port used for servlets and resources available via HTTP. The default is `80`. A negative port number has the same effect as setting `org.apache.felix.http.enable` to `false`.
- `org.osgi.service.http.port.secure` - The port used for servlets and resources available via HTTPS. The default is `443`. A negative port number has the same effect as setting `org.apache.felix.https.enable` to `false`.
- `org.apache.felix.http.nio` - Flag to enable the use of NIO instead of traditional IO for HTTP. One consequence of using NIO with HTTP is that the bundle needs at least a Java 5 runtime. The default is `true`.
- `org.apache.felix.https.nio` - Flag to enable the use of NIO instead of traditional IO for HTTPS. One consequence of using NIO with HTTPS is that the bundle needs at least a Java 5 runtime. If this property is not set the (default) value of the `org.apache.felix.http.nio` property is used.
- `org.apache.felix.http.enable` - Flag to enable the use of HTTP. The default is `true`.
- `org.apache.felix.https.enable` - Flag to enable the user of HTTPS. The default is `false`.
- `org.apache.felix.https.keystore` - The name of the file containing the keystore.
- `org.apache.felix.https.keystore.password` - The password for the keystore.
- `org.apache.felix.https.keystore.key.password` - The password for the key in the keystore.
- `org.apache.felix.https.truststore` - The name of the file containing the truststore.
- `org.apache.felix.https.truststore.password` - The password for the truststore.
- `org.apache.felix.https.clientcertificate` - Flag to determine if the HTTPS protocol requires, wants or does not use client certificates. Legal values are `needs`, `wants` and `none`. The default is `none`.
- `org.apache.felix.http.debug` - Flag to enable debugging for this service implementation. The default is `false`.

Additionally, the all-in-one bundle uses the following environment properties:

- `org.apache.felix.http.jettyEnabled` - True to enable jetty as the http container. The default is `false`.
- `org.apache.felix.http.whiteboardEnabled` - True to enable the whiteboard implementation. The default is `false`.

The Jetty based implementation supports the following Jetty specific configuration as of Http Service Jetty Bundle 2.4:

- `org.apache.felix.http.host` - Host name or IP Address of the interface to listen on. The default is `null` causing Jetty to listen on all interfaces.
- `org.apache.felix.http.context_path` - The Servlet Context Path to use for the Http Service. If this property is not configured it defaults to "/". This must be a valid path starting with a slash and not ending with a slash (unless it is the root context).
- `org.apache.felix.http.timeout` - Connection timeout in milliseconds. The default is 60000 (60 seconds).
- `org.apache.felix.http.session.timeout` - Allows for the specification of the Session life time as a number of minutes. This property serves the same purpose as the `session-timeout` element in a Web Application descriptor. The default is zero for no timeout at all.
- `org.mortbay.jetty.servlet.SessionCookie` - Name of the cookie used to transport the Session ID. The default is `JSESSIONID`.
- `org.mortbay.jetty.servlet.SessionURL` - Name of the request parameter to transport the Session ID. The default is `jsessionid`.
- `org.mortbay.jetty.servlet.SessionDomain` - Domain to set on the session cookie. The default is `null`.
- `org.mortbay.jetty.servlet.SessionPath` - The path to set on the session cookie. The default is the configured session context path (`/`).
- `org.mortbay.jetty.servlet.MaxAge` - The maximum age value to set on the cookie. The default is `-1`.

- `org.apache.felix.http.jetty.headerBufferSize` - Size of the buffer for request and response headers. Default is 16KB.
- `org.apache.felix.http.jetty.requestBufferSize` - Size of the buffer for requests not fitting the header buffer. Default is 8KB.
- `org.apache.felix.http.jetty.responseBufferSize` - Size of the buffer for responses. Default is 24KB.

Servlet API Events

The Servlet API defines a number of `EventListener` interfaces to catch Servlet API related events. As of HTTP Service 2.1.0 most events generated by the servlet container are forwarded to interested service. To be registered to receive events services must be registered with the respective `EventListener` interface:

Interface	Description
`javax.servlet.ServletContextAttributeListener`	Events on servlet context attribute addition, change and removal.
`javax.servlet.ServletRequestAttributeListener`	Events on request attribute addition, change and removal.
`javax.servlet.ServletRequestListener`	Events on request start and end.
`javax.servlet.http.HttpSessionAttributeListener`	Events on session attribute addition, change and removal. To receive such events in a bridged environment, the `ProxyLister` must be registered with the servlet container. See the *Using the Servlet Bridge* section above.
`javax.servlet.http.HttpSessionListener`	Events on session creation and destroyal. To receive such events in a bridged environment, the `ProxyLister` must be registered with the servlet container. See the *Using the Servlet Bridge* section above.

Of the defined `EventListener` interfaces in the Servlet API, the `javax.servlet.ServletContextListener` events are actually not support. For one thing they do not make much sense in an OSGi environment. On the other hand they are hard to capture and propagate. For example in a bridged environment the `contextInitialized` event may be sent before the framework and any of the contained bundles are actually ready to act. Likewise the `contextDestroyed` event may come to late.

Servlet Context Notes

`ServletContext` instances are managed internally by the Http Service implementation. For each `HttpContext` instance used to register one or more servlets and/or resources a corresponding `ServletContext` instance is created. These `ServletContext` instances is partly based on the single `ServletContext` instance received from the Servlet Container — either embedded Jetty or some external Servlet Container when using the Http Service Bridge — and partly based on the provided `HttpContext` instance:

Method(s)	Based on ...
`getContextPath`, `getContext`, `getMajorVersion`, `getMinorVersion`, `getServerInfo`	Servlet Containers `ServletContext`
`getResourcePaths`	`Bundle.getEntryPaths` of the bundle using the Http Service
`getResource`, `getResourceAsStream`	`HttpContext.getResource`
`getMimeType`	`HttpContext.getMimeType`
`getRequestDispatcher`, `getNamedDispatcher`, `getInitParameter`, `getServlet`, `getRealPath`	Always return `null`
`getInitParameterNames`, `getServlets`, `getServletNames`	Always returns empty `Enumeration`
`getAttribute`, `getAttributeNames`, `setAttribute`, `removeAttribute`	By default maintained for each `ServletContext` managed by the Http Service. If the `org.apache.felix.http.shared_servlet_context_attributes` framework property is set to `true` these methods are actually based on the `ServletContext` provided by the servlet container and thus attributes are shared amongst all `ServlectContext` instances, incl. the `ServletContext` provided by the servlet container

Examples

A set of simple examples illustrating the various features are available.

- Filter registration sample: http://svn.apache.org/repos/asf/felix/trunk/http/samples/filter/
- Servlet bridge sample: http://svn.apache.org/repos/asf/felix/trunk/http/samples/bridge/
- Whiteboard sample: http://svn.apache.org/repos/asf/felix/trunk/http/samples/whiteboard/

Maven Artifacts

```
<dependency>
  <groupId>org.apache.felix</groupId>
  <artifactId>org.apache.felix.http.api</artifactId>
  <version>2.0.4</version>
</dependency>
<dependency>
  <groupId>org.apache.felix</groupId>
  <artifactId>org.apache.felix.http.base</artifactId>
  <version>2.0.4</version>
</dependency>
<dependency>
  <groupId>org.apache.felix</groupId>
  <artifactId>org.apache.felix.http.bridge</artifactId>
  <version>2.0.4</version>
</dependency>
<dependency>
  <groupId>org.apache.felix</groupId>
  <artifactId>org.apache.felix.http.bundle</artifactId>
  <version>2.0.4</version>
</dependency>
<dependency>
  <groupId>org.apache.felix</groupId>
  <artifactId>org.apache.felix.http.jetty</artifactId>
  <version>2.0.4</version>
</dependency>
<dependency>
  <groupId>org.apache.felix</groupId>
  <artifactId>org.apache.felix.http.proxy</artifactId>
  <version>2.0.4</version>
</dependency>
<dependency>
  <groupId>org.apache.felix</groupId>
  <artifactId>org.apache.felix.http.whiteboard</artifactId>
  <version>2.0.4</version>
</dependency>
```