JAX-RS-based extension

SuperMap iServer not only provides REST services, publishing a large number of GIS functions as resource, but also provides an extension mechanism, adding users' applications to SuperMap iServer server, which can implement the custom resources to be a part of REST services.

SuperMap iServer has two ways to provide REST services: based on Restlet and JAX-RS. When extending, you should use different extending methods according to the implementation of function modules.

The REST resource module of SuperMap iServer implemented based on the JAX-RS is the spatial analyst module.

SuperMap iServer provides the resource extension mechanism based on the JAX-RS framework (refer to JAX-RS Introduction), which is used for the REST service extension:

1. It provides the resource base class: JaxrsResourceBase and JaxAlgorithmResultSetResource<T>, for users to extend and implement the new resource.
2. SuperMap iServer provides several output formats like xml, json and rjson. Resource created based on the base class support the output formats by default.
3. It provides the @Template annotation, and a series of FreeMarker variables, for users to customize the *.ftl template and configure it to the resource. Therefore, the resource can support the html output format.
4. It provides the JSON decoder. The default identification type of the resource created based on the resource base class is the JSON format.
5. It provides the service interface context (InterfaceContext). The resource created based on the base class can get the service component and other information through InterfaceContext.
6. The domain functionalities are tightly attached to the three-layer architecture of SuperMap iServer, which induces the creation of domain resources through the combination with the current GIS functionalities.

Therefore, the REST resource extensions based on JAX-RS include:

- Extending a new Resource
- Extending an Encoder
- Extending a Decoder

The extend flow is as follows:


New resources implement based on com.supermap.services.rest.resources.JaxAlgorithmResultSetResource or com.supermap.services.rest.resources.JaxrsResourceBase;

The Encoder implement based on javax.ws.rs.ext.MessageBodyWriter;

The Decoder implement based on javax.ws.rs.ext.MessageBodyReader.

2. Configuration

- Extending the Encoder (consistent with Extending the Encoder for existing modules)
- Extending the Decoder (consistent with Extending the Decoder for existing modules)

Here it introduces FakeKMLEncoder, FakeJsonDecoder and myBuffer. The source codes locate in: %SuperMap iServer_Home%/samples/code\ExtendExist_JSR. It is an overall project file. Users only need to import this project
and compile a JAR (See extendexist.jsr.jar). Then put Jar into %SuperMap
iServer_HOME%/webapps/iserver/WEB-INF/lib, and restart iServer.

Extending a Resource

Here is an example of extending the spatial analyst function to implement a special buffer analysis function, which is used to get the buffer of the desired line L1. The coordinates of the endpoints of L1 are: (116.40, 39.97), (116.40, 40.00), and (116.38, 40.00).

The resource <root_uri>/spatialanalyst/geometry/mybuffer is implemented, and a POST request is performed on the resource to implement the buffer analysis. The request body parameter is designed as a Double number, indicating the buffer radius. The analysis result is the round head buffer of L1.

Note: The analysis result is saved in the storage repository provided by iServer SDK as the child resource of the mybuffer resource. It can be got by performing a GET request on <root_uri>/spatialanalyst/geometry/mybuffer/[id].

As to the extension of resources support the POST request, it is recommended to use the base class com.supermap.services.rest.resources.JaxAlgorithmResultSetResource provided by SuperMap iServer for extension.

Please see the sample code: %SuperMap iServer Java_HOME%/samples/code/ExtendExist_JSR/MyBufferResultsResource.java.

Resource implementation

The type of the request body parameter of the POST request is double. Compose a MyBufferResultsResource class as follows:

```java
@Path("/spatialanalyst/geometry/mybuffer")

public class MyBufferResultsResource extends JaxAlgorithmResultSetResource<Double> {
  ...... 
}
```

@Path specifies the URI of mybuffer resource. In iServer6R, the root directory of the REST resource implemented based on the JAX-RS framework is: http://<server>:<port>/iserver/services/<servicecomponent>/serviceinterface, where servicecomponent is the service component, serviceinterface is the service interface (see Setting the Service URI), and restjsr is the default JAX-RS REST service interface. The example is expected to extend the spatial analyst function, and the spatial analyst service component is used.

Implement the abstract method as shown below:

```java
/**
 * The algorithm name, used for accessing the algorithm result.
 */
protected String getAlgorithmName() {
  return "MyBuffer";
}
/**
 * Implement the runArithmetic of the parent class. The business logic to be processed when the client sends the HTTP POST method.
 * The Post method of the parent class will call the interface.
 */
protected Object runArithmetic(Double leftBufferDistance) {
  //Construct buffer analyst parameters. The left buffer radius is leftBufferDistance, the right buffer radius is identical to left buffer radius, and the buffer type is ROUND
```
BufferAnalystParameter bufferParameter = this.createGeometryBufferPostParameter(leftBufferDistance);
//Construct the line geometry object as source for creating the buffer. The endpoints are (116.40, 39.97), (116.40, 40.00) and (116.38, 40.00).
Geometry sourceGeometry = this.createSourceGeometry();
// Get the business component for the buffer analysis.
SpatialAnalyst spatialAnalyst = null;
//Get all spatial analyst business components from the interface context.
List<SpatialAnalyst> spatialAnalystList = this.getInterfaceContext().getComponents(SpatialAnalyst.class);
//By default, there is only one spatial analyst business component in the interface context.
//There will be one business component in the interface context if the business component is bound to the REST interface.
//There will be zero or multiple spatial analyst business components in the interface context if the business component collection is bound to the REST interface.
spatialAnalyst = spatialAnalystList.get(0);
// Buffer analysis
return spatialAnalyst.buffer(sourceGeometry, bufferParameter, new GeometrySpatialAnalystResultSetting()).resultGeometry;
}

createGeometryBufferPostParameter and createSourceGeometry are the creation parameters and the code is as follows:

Code
The service component is got from the service interface context. In order to get the correct service interface component, it needs to inherit the constructor of the parent class (JaxAlgorithmResultSetResource):
/**
 * The constructor of the resource implementation class.
 * The parameters of servletConfig and httpRequest will be injected dynamically when processing the request by the Jax-rs runtime.
 * @param servletConfig The unique global Servlet configuration
 */
public MyBufferResultsResource(@Context ServletConfig servletConfig) {
// If the spatial analyst business component is expected to get from the resource, the corresponding constructor of the parent class should be overloaded.
super(servletConfig);
}

The POST request is usually used to create a child resource, and here a buffer analysis result resource is created. In order to get the algorithm resource, a child resource supporting the GET method is implemented in MyBufferResultsResource.
/**
 * Returns the specified analysis result.
 * @param id The ID of the analysis result.
 * @return The region buffer.
 */
@GET
@Path("{id}")
public Geometry getBufferResultGeometry(@PathParam("id") String id) {
// Get the object storage repository, where the buffer analysis result is stored.
// There are multiple child repositories in this repository, differentiated by the key value. Here the key value is MyBuffer.
TempObjRepository objRepository = this.getRepository();
return (Geometry) objRepository.getArithResult(ALGORITHMNAME, id);
}

Where @GET represents that this method corresponds to the GET request, @Path ("{id}") is the URI of the results resource, here, from the URI of the mybuffer resource. [id] corresponds to @PathParam("id") String id, indicating passing the URI [id] to String id. If the GET request URI is <mybuffer_uri>/12, the analysis result with the ID of 12
will be got.

Up until now, the sample is finished. The parent resource `<root_uri>/spatialanalyst/geometry/mybuffer` supports the POST request. The response of the POST request will include the URIs of the child resources. The child resource `<root_uri>/spatialanalyst/geometry/mybuffer/{id}` supports the GET request.

Note: The @Template annotation is not supported here. By default, the HTTP output formats like xml, json and rjson are supported. The default message body format supported by mybuffer is JSON.

**Add configuration file**

Create the MyBufferAnalystRest.xml file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<resources>
    <resource>
        <configID>mybuffer</configID>
        <implementClass>com.supermap.sample.extend.MyBufferResultsResource</implementClass>
    </resource>
</resources>
```

Create the JAX-RS resource configuration file. The example is the extension of the spatial analyst module. Therefore, the configuration file name should be consistent with that of the spatial analyst module configuration file, SpatialAnalystRest (See Table Module and Configuration), shown as below:

```
resourceFiles=MyBufferAnalystRest.xml
```

Create a jar package with the configuration file and the compiled result of implementation class of the Mybuffer resource (see extendexist_jsr.jar), put it under `%SuperMap iServer Java_HOME%/webapps/iserver/WEB-INF/libm` and restart SuperMap iServer6R.

MyBufferAnalystRest.xml is located in Jar://MyBufferAnalystRest.xml, and SpatialAnalystRest is located in Jar://META-INF/extensions/services/rest/SpatialAnalystRest.

**Resource access**

After restarting the server, `http://localhost:8090/iserver/services/spatialanalyst-sample/restjsr` is an existing spatial analyst module. Access the newly extended MyBuffer resource by implementing a POST request on `http://localhost:8090/iserver/services/spatialanalyst-sample/restjsr/spatialanalyst/geometry/mybuffer.rjsonl`, and the request body is "4".

```
POST http://localhost:8090/iserver/services/spatialanalyst-sample/restjsr/spatialanalyst/geometry/mybuffer.rjson HTTP/1.1
Content-Length: 1
Content-Type:application/json
4
```

The response in rjson format is as follows:

```
{
    "newResourceId": "4",
    "newResourceLocation": "http://localhost:8090/iserver/services/spatialanalyst-sample/restjsr/spatialanalyst/geometry/mybuffer/1",
    "postResultType": "CreateChild",
    "succeed": true
}
```

Where newResourceLocation is the child resource (address of the algorithm result). Implement a GET request on `http://localhost:8090/iserver/services/spatialanalyst-sample/restjsr/spatialanalyst/geometry/mybuffer/1.rjson` to get the response in rjson output format, as shown below.
The result is a region object (com.supermap.services.components.commonetypes.Geometry).
Extending an Encoder

Here is an example of extending the spatial analysis output format of the geometry object (like geometryBufferResult). A pseudo KML format is added. That is to say, a simple KML region is generated according to the result region.

Implementation of new encoder pseudo KML

Firstly, determine the media type of the KML representation is "application/mykml". And in the iServer6R, if ".kml" is appended to the end of the URI when performing a GET request, the iServer6R server will return the HTTP response in "application/mykml" (To know how to explicitly configure the media type, please refer to Extending Media Types);

Secondly, use @Provider to register the customized MessageBodyProvider, namely FakeKMLEncoder, based on javax.ws.rs.ext.MessageBodyWriter and use @Produces to specify the media type as "application/mykml";

Thirdly, customize a KML file to represent a region object.

Fourthly, judges whether the response can be parsed into "application/mykml". If yes, a kml file will be returned.

The code for the above process are as follows, and the code file can be found in %SuperMap iServer Java_HOME%/samples/code/ExtendExist_JSR/FakeKMLEncoder.java.

```java
package com.supermap.sample.extend;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.Writer;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;
import javax.ws.rs.Produces;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.ext.MessageBodyWriter;
import javax.ws.rs.ext.Provider;
import com.supermap.services.components.commontypes.Geometry;
import com.supermap.services.components.commontypes.GeometrySpatialAnalystResult;
import com.supermap.services.components.commontypes.Point2D;
@Provider
@Produces("application/mykml")
public class FakeKMLEncoder implements MessageBodyWriter {
    // Determine the length of the object to be written in after serialization, with -1 indicating unknown.
    public long getSize(Object object, Class type, Type genericType, Annotation[] annotations, MediaType mediaType) {
        return -1;
    }

    /**
     * <p>
     * Judges whether the application/kml media type is supported
     * </p>
     */
    public boolean isWriteable(Class type, Type genericType, Annotation[] annotations, MediaType mediaType) {
        if (mediaType.equals(new MediaType("application", "mykml"))) {
            return true;
        }
    }
}
```
public void writeTo(Object resourceObj, Class type, Type genericType, Annotation[] annotations, MediaType mediaType, MultivaluedMap httpHeaders, OutputStream entityStream) throws IOException, WebApplicationException {
    String result;
    // KML name (<name/> tag)
    String resultName = "result.kml";
    if (resourceObj instanceof Geometry) {
        Geometry geometry = (Geometry) resourceObj;
        result = this.createKMLRegion(resultName, geometry);
    } else if (resourceObj instanceof GeometrySpatialAnalystResult) {
        Geometry geometry = ((GeometrySpatialAnalystResult) resourceObj).resultGeometry;
        result = this.createKMLRegion(resultName, geometry);
    } else {
        result = "The response is not a valid Geometry object";
    }
    // Write the KML file into entityStream, as the message body of the KML representation.
    Writer writer = new BufferedWriter(new OutputStreamWriter(entityStream, "utf-8"));
    writer.write(result.toString(), 0, result.length());
    writer.close();
}

private String createKMLRegion(String filename, Geometry geometry) {
    Point2D[] points = geometry.points;
    // Convert into point set (<coordinates/> tag content)
    String coordinates = "";
    for (Point2D point : points) {
        coordinates += point.x + "," + point.y + "," + "0 \n";
    }
    // The customized KML file, a region object
    StringBuffer sb = new StringBuffer();
    sb.append("<kml xmlns="http://www.opengis.net/kml/2.2"/>
    
    <Placemark>
    
    <name>" + filename + "</name>
    
    <Polygon>
    
    <outerBoundaryIs>
    
    <LinearRing>
    
    <coordinates>" + coordinates + "0 \n"
    
    </coordinates>
    
    </LinearRing>
    
    </outerBoundaryIs>
    
    </Polygon>
    
    </Placemark>
    
    </kml>"
    
    return sb.toString();
}

Pseudo KML representation configuration

Please refer to Configuration Files to learn the configuration method of the module configuration file. Add the content below to [Jar package] \META-INF\extensions\services\restSpatialAnalystRest:

encoders=com.supermap.sample.extend.FakeKMLEncoder
resouceFiles=MyBufferAnalystRest.xml

Complete the configuration by exporting the project ExtendExist_JSR where the code is located into JAR package (see extendexist_jsr.jar) and placing the package in %SuperMap iServer Java_HOME%\webapps\iserver\WEB-INF\lib.
The sample code can be seen in %SuperMap iServer Java_HOME%\samples\code\ExtendExist_JSR. It can be used in Eclipse by directly importing it. Note that when exporting the Jar package in Eclipse, check "Add directory entries", as shown below:

![JAR Export screen](image)

**Use of the pseudo KML representation**

Here is an example of introducing the KML representation usage through Geometry buffer analysis. Start the SuperMap iServer service, access the resource: http://supermapiserver:8090/iserver/services/spatialanalyst-sample/restjsr/spatialanalyst/geometry/buffer, and change the buffer distance to 1, with all the other setting unchanged. Implement a POST request and get the response: http://supermapiserver:8090/iserver/services/spatialanalyst-sample/restjsr/spatialanalyst/geometry/buffer/1.mykml.

Save the 1.mykml file to local. After revising the suffix, the 3D scene effect is as follows:

Note: You can choose to load the KML file in SuperMap iDesktop: open an existing scene or create a new scene, right click on the General Layers node in Layer Manager, and click Add KML Layer to add the data to the scene.
Extending a Decoder

Implementation of new decoder myjson

As an example, here we define a pseudo JSON format to make SuperMap iServer support the decoding of this type. The pseudo JSON format uses "(" to replace "{", and ")" to replace "}", and other definitions are identical to those of the JSON format.

Firstly, determine the media type of the pseudo JSON format--"application/myjson". The media type corresponds to the Content-Type request header of the HTTP request, which is used for the server to read and identify the parameters in the message body and call the corresponding decoder for decoding;

Secondly, use @Provider to register the customized MessageBodyProvider, namely FakeJsonDecoder, based on avax.ws.rs.ext.MessageBodyReader and use @Produces to specify the media type as "application/mykml";

Thirdly, judges whether or not the received request body is in "application/myjson" media type;

Fourthly, read the request body and replace the "(" and ")" in the pseudo JSON string with "{" and "}" respectively, converting to true JSON string;

Fifthly, decode the true JSON string through the JSON converter provided by SuperMap iServer 6R and get the response.

The code for the above process are as follows, and the code file can be found in %SuperMap iServer Java_HOME%\samples\code\ExtendExist_JSR\FakeJsonDecoder.java.

package com.supermap.sample.extend;
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;
import javax.ws.rs.Consumes;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.ext.MessageBodyReader;
import javax.ws.rs.ext.Provider;
import org.json.JSONException;
import com.supermap.services.rest.util.JsonConverter;

// Support the pseudo JSON format, with "[]" being replaced by "()".
@Provider
@Consumes("application/myjson")
public class FakeJsonDecoder implements MessageBodyReader<Object> {
    public boolean isReadable(Class<?> type, Type genericType, Annotation[] annotations, MediaType mediaType) {
        // TODO Auto-generated method stub
        if (mediaType.equals(new MediaType("application", "myjson"))) {
            return true;
        }
        return false;
    }

    public Object readFrom(Class<Object> type, Type genericType, Annotation[] annotations, MediaType mediaType, MultivaluedMap<String, String> httpHeaders, InputStream entityStream) throws java.io.IOException, WebApplicationException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(entityStream, "utf-8"));
        String data;
        StringBuffer buffer = new StringBuffer();
        while((data = reader.readLine()) != null) {
            buffer.append(data);
        }
        String text = buffer.toString();
        // Replace the "(" and ")"s in the pseudo JSON string with "{"s and "}"s respectively, converting to true JSON, and parse the true JSON string.
        text=text.replace(')', '}');
        text=text.replace('(', '{');
        try {
            // Convert the string into Java object and return it.
            // The Json converter provided by iServer 6R
            JsonConverter converter = new JsonConverter();
            return converter.to(text, type);
        } catch (JSONException e) {
            throw new RuntimeException("The request body content is not valid myjson string");
        }
    }
}

Configuration and use of myjson
Please refer to Configuration Files to learn the configuration method of the module configuration file. Add the content below to [Jar package] \META-INF\extensions\services\restSpatialAnalystRest:

decoders=com.supermap.sample.extend.FakeJsonDecoder
resourceFiles=MyBufferAnalystRest.xml
Complete the configuration by exporting the project ExtendExist_JSR where the code is located into JAR package (see extendexist_jsr.jar) and placing the package in %SuperMap iServer Java_HOME%/webapps/iserver/WEB-INF/lib.

Start the service. Here we take Geometry buffer analysis as an example and construct a POST request as shown below:

POST http://supermapiserver:8090/iserver/services/spatialanalyst-sample/restjsr/spatialanalyst/geometry/buffer.xml
HTTP/1.1
X-RequestEntity-Content-Type: application/myjson
Host: supermapiserver:8090
Content-Length: 223
{"analystParameter":{"endType":"ROUND", "semicircleLineSegment":4, "leftDistance":{"value":1}, "rightDistance":{"value":1}}, "sourceGeometry":{"type":"LINE", "points":[{"x":23, "y":23}, {"x":33, "y":37}, {"x":43, "y":23}]}}

The response in xml format is as follows:

<?xml version="1.0" encoding="utf-8" standalone="no"?><MethodResult>
<isSucceed>true</isSucceed>
<newResourceId>4</newResourceId>
<postResultType>CreateChild</postResultType>
</MethodResult>