Restlet-based extension

SuperMap iServer not only provides REST services and publishes a large number of GIS functionalities as resource, but also provides a suit of extension mechanism, which can help users add their own applications to SuperMap iServer, implement custom resources and publish domain services.

Currently, SuperMap iServer has two methods to provide REST service, namely the mechanism based on Restlet and JAX-RS. When performing the extension, you should adopt the different extension methods according to the implementation method of functional modules.

The modules that implement REST resource based on the Restlet mechanism have: map module, data module, transporation analyst module and 3D module. SuperMap iServer provides the following extensions:

- Extending resource by using REST SDK and inheriting the abstract resource class provided in SDK.
- Extending the encoder. The resource on the server can be published in new output formats.
- Extending the decoder. The server can recognize new parameter passing formats.
- Extending HTTP Handler. The processing procedure of the HTTP request on the server can be customized.
- Extending security. Users can configure their own security mechanism to the server.

Below shows how an HTTP request sent by a client to access SuperMap iServer REST service is processed on SuperMap iServer. In green is places where the extension can be performed.

Encoder extension, decoder extension, and HTTP request extension can be performed for original resources on the server and those resources formed by extending through REST SDK or publishing domain components.

Follow the links below to know how to extend REST services.

◆ REST service publishing mechanism
Extending a Simple Algorithm

Extending an Encoder

Extending a Decoder

Extending an HTTP Request Handler

Extending a Security Mechanism

REST service publishing mechanism

SuperMap iServer publishes REST services through Restlet. Firstly, construct a REST application, and then deploy the application object in an HTTP environment, such as Servlet container, OSGi core, Spring, Guice Ioc container, independent JVMS, Groovy, Scala, JAIN/SLEE, etc. Currently, the application object is deployed in the Servlet container. After that, SuperMap iServer can publish REST services through the REST application object, as illustrated below.

![Diagram of REST service publishing mechanism](image)

As shown above, an HTTP request firstly arrives at the HTTP environment, then the HTTP environment passes the request to the REST application object, the corresponding class of which inherits from Application in Restlet. The HTTP request will be handled according to the context and the response will be sent back to the client through the HTTP environment.

When an HTTP request arrives at the REST application object, the application object will search the resource configuration information in the REST application context and compare the URI for the HTTP request with those templates. The best matched resource will be found and the implementation of the resource is expected to handle the HTTP request. The procedure is as follows:

![Diagram of resource configuration](image)

The REST application context contains the information about the resource configuration, REST application configuration, core component configuration, etc. needed by the REST application. The REST application context is obtained from the configuration file when starting the server.

The resource configuration information includes the resource ID, resource type, URI template, implementation
class, etc. Resources are implemented through the REST application object and are published in the form of the corresponding URI templates.

The implementation class of a resource is used to handle all operations on the resource. Each implementation class has an HTTP MethodHandler, which aims at managing the HTTP request handling. The HTTP request handling has the following steps:

1. Extract parameters from the HTTP request (it is possible that there are no parameters in the HTTP request);
2. Get the correct Decoder according to the HTTP request header, parsing the parameters in the HTTP request into Java objects;
3. Handle the business logic according to the parameters (there can be no parameters) and return the result;
4. Obtain the correct Encoder according to URI, encode the result and write the result into the HTTP response object.

In short, when an HTTP request arrives at the REST application object, a proper resource will be found to handle the request. After that the result will be returned to the client through the HTTP environment as an HTTP response.

**Extending a Simple Algorithm**

The REST service publishing mechanism section gives you an introduction to the REST service publishing mechanism. The resource information is saved in the resource configuration files and transferred to the REST application object through the REST application context. Therefore, when the HTTP request arrives at the REST application object, the proper resource can be found to handle the request. Through the process summarized above, we can know that we need to take two steps to extend a simple algorithm resource.

**1. Implementing a Simple Algorithm Class**

In the REST implementation framework of SuperMap iServer, the highest-level resource class is `resourceBase`, from which all resources can directly or indirectly inherit.

When extending REST resource, you need to get the favorable class by inheriting from the `ResourceBase` class or its child class. SuperMap iServer provides a lot of abstract classes except for the `ResourceBase` class and the final implementation class, representing different types of resources, which provides great convenience for REST resource extension. You can find the proper abstract class according to the resource functions for extension. Also, you can
decide to directly inherit from an existing resource class if the target resource is similar to an existing resource.

Please refer to the com.supermap.services.rest.resources package in Javadoc for more information about the abstract classes for REST resources in SuperMap iServer.

As a reference for resource extension, here we list the types of implemented resource classes (see the ResourceType enumeration in Javadoc).

<table>
<thead>
<tr>
<th>Name</th>
<th>Parent class</th>
<th>Implemented resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple algorithm resource</td>
<td>SimpleAlgorithmResultResourceBase</td>
<td>symbol, clearCache, etc. of map, statistic of data</td>
</tr>
<tr>
<td>ImageResourceBase</td>
<td></td>
<td>entireimage, highlightimage, image, overview, tileImage, etc.</td>
</tr>
<tr>
<td>MeasureResourceBase</td>
<td></td>
<td>area, distance, etc.</td>
</tr>
<tr>
<td>Algorithm resource</td>
<td>AlgorithmResultResource</td>
<td>queryResult of map, featureResult of data</td>
</tr>
<tr>
<td>Algorithm result resource</td>
<td>AlgorithmResultSetResource</td>
<td>queryResults of map, featureResults of data</td>
</tr>
<tr>
<td>Directory resource</td>
<td>CatalogListResourceBase</td>
<td>maps, root, tempLayersSet, trackingLayers, domainComponents, domainComponentData etc. of map, data, datasets, etc. of data, networkanalyst of networkanalyst, 3D, scenes, datas etc. of 3D</td>
</tr>
<tr>
<td>Static resource</td>
<td>StaticResource</td>
<td>layers, map, templayers, etc. of map, networkDataName of networkanalyst</td>
</tr>
</tbody>
</table>

In addition, CommonAlgorithmResultResource is a child class of the SimpleAlgorithmResultResourceBase resource. Resources implemented based on CommonAlgorithmResultResource include closestfacilities, weightMatrix, TSPPaths, serviceAreas, MTSPPaths, location, paths, etc. of the closestfacilities module, tileData, etc. of the 3D module. SimpleResource is a child class of ResourceBase. Resources implemented based on SimpleResource include scene, layers, layer, data, config, modelIndex, tileDataVersion, etc. of the 3D module.

Here, we extend the rectangleArea resource to implement the functionality of measuring a rectangular area on the map and returning the result.

Since we are going to measure area on a map, we take rectangleArea as a child resource of the map resource. The designed URI is as follows:

```
http://<server>:<port>/iserver/services/components-rest/rest/maps/{mapName}/rectangleArea[.<format>]
```

The request parameters are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rect2D</td>
<td>Rectangle2D</td>
<td>The rectangular area to be measured on the map.</td>
</tr>
<tr>
<td>unit</td>
<td>Unit</td>
<td>The unit of the measurement result.</td>
</tr>
</tbody>
</table>

The response structure is:

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>area</td>
<td>double</td>
<td>The area of the rectangular area.</td>
</tr>
<tr>
<td>unit</td>
<td>Unit</td>
<td>The unit of the measurement result.</td>
</tr>
</tbody>
</table>

Since rectangleArea resource is similar to the SimpleAlgorithmResultResourceBase resource, we construct the implementation class of rectangleAreaResource by inheriting from the implementation class of the SimpleAlgorithmResultResourceBase abstract class. The abstract methods of the SimpleAlgorithmResultResourceBase class, as shown below, are required to be implemented. For more details, please refer to Javadoc.
The business logic, i.e., the measuring functionality, can be implemented by the implementation class `com.supermap.services.components.impl.MapImpl` (already provided by SuperMap iServer) of the map component (`com.supermap.services.components.Map`). In the sample class `RectangleAreaResource`, the constructor gets the map name and the MapImpl component. The map name is retrieved from the URI, and the MapImpl object is retrieved from the REST application context. Sample code:

```java
package com.supermap.sample.extendREST;
import java.util.HashMap;
import org.restlet.Context;
import org.restlet.Request;
import org.restlet.Response;
import com.supermap.services.components.Map;
import com.supermap.services.components.MapException;
import com.supermap.services.components.commontypes.MeasureParameter;
import com.supermap.services.components.commontypes.MeasureResult;
import com.supermap.services.components.commontypes.Point2D;
import com.supermap.services.components.commontypes.Rectangle2D;
import com.supermap.services.components.commontypes.Unit;
import com.supermap.services.rest.resources.SimpleAlgorithmResultResourceBase;
import com.supermap.services.rest.util.MappingUtil;
public class RectangleAreaResource extends SimpleAlgorithmResultResourceBase{
    private String mapName;
    private MappingUtil mappingUtil;
    public RectangleAreaResource(Context context, Request request, Response response) {
        super(context, request, response);
        mappingUtil = new MappingUtil(this);
        this.mapName = this.mappingUtil.getMapName(this.getRequest());
        this.mapName = this.mapName.trim();
    }
    // Determines whether the resource exists
    public boolean isResourceExist(){
        // Logger.info("rectangle area resource exists");
        boolean flag = false;
        flag = this.mappingUtil.isMapExist(this.mapName);
        return flag;
    }
    // Run the algorithm and get the result
    protected Object runArithmetic(java.util.Map params){
        Object arithResult=null;
        // If the parameters are null, the area will be returned as -1
        if(params==null||0==params.size()){  
            MeasureResult result = new MeasureResult();
            result.area = -1;
            arithResult = result;
        }else{
            Rectangle2D rect = (Rectangle2D)params.get("rect2D");
        }
    }
}
```
Unit unit = (Unit)params.get("unit");
//The unit can be not transferred. The default is meters
if(unit == null){
    unit = Unit.METER;
}
//The measurement parameter class with the unit information stored
MeasureParameter measureParam = new MeasureParameter();
measureParam.unit = unit;
//Convert Rectangle2D into Point2Ds
Point2D leftBottom=rect.leftBottom;
Point2D rightTop=rect.rightTop;
Point2D leftTop=new Point2D(leftBottom.x,rightTop.y);
Point2D rightBottom=new Point2D(rightTop.x,leftBottom.y);
Point2D[] point2Ds={leftBottom,leftTop,rightTop,rightBottom};
//The measurement result
try {
    Map mapComponent = this.mappingUtil.getMapComponent(mapName);
    arithResult = mapComponent.measureArea(mapName, point2Ds, measureParam);
} catch (MapException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
return arithResult;
}

protected java.util.Map<String, Class> createArithParamClassMappings(){
    java.util.Map<String, Class> paramClassMapping = new HashMap<String, Class>();
    paramClassMapping.put("rect2D", Rectangle2D.class);
    paramClassMapping.put("unit", Unit.class);
    return paramClassMapping;
}

protected void checkUrlParamValid(java.util.Map arg0) {
    // TODO Auto-generated method stub
}

You can find the source code for this example in %SuperMap iServer_HOME%/samples/code/ExtendREST.

After compiling, package into a Jar package, ExtendREST.jar for instance, and place the package in %SuperMap iServer_HOME%/webapps/iserver/WEB-INF/lib. Please note that all implementation classes, including the implementation of the extensions that will be introduced later, need to be packaged and placed in %SuperMap iServer_HOME%/webapps/iserver/WEB-INF/lib.

Note: SuperMap iServer REST framework uses FreeMarker to generate HTML format representation. To make the rectangleArea resource support HTML format representation, we still need to compose the *.ftl files, i.e., templates. Please refer to the existing template files in the templates directory in SuperMap iServer_HOME%/webapps/iserver/WEB-INF/lib/iserver-all-{version}.jar.

FreeMarker is a generic tool to generate text output. In SuperMap iServer REST framework, $resource.content.* in the *.ftl template file is the representation result field and the file name must be identical to the resource ID.
Compose the FreeMarker template file for the rectangleArea, as shown below. rectangleArea.ftl must be placed in a Jar package in %SuperMap iServer_HOME%/webapps/iserver/WEB-INF/lib. It is suggested that you place the template file in the same package with the implementation class of the resource, i.e., ExtendREST.jar. The Jar package holding the template file must be placed in the [Jar package]/templates directory.

```freemarker
<#include "head.ftl">
<h1>rectangleArea resource</h1>
<br>
<b>Description: </b>The area measurement result resource, used to represent an area measurement result.
<br>
<#if resource.content.area!=-1>
<p>
<table width="150px" id="userTable">
  <tr align="left">
    <#setting number_format="#0.000000000#">
    <td style="width:50px">area:</td>
    <td style="width:50px">${resource.content.area}</td>
  </tr>
  <tr align="left">
    <td style="width:50px">unit:</td>
    <td>${resource.content.unit}</td>
  </tr>
</table>
</p>
<#else>
<br><br>
Please set the rect2D parameter. If not set, -1 will be returned, indicating measurement failure.
</#if>
<p>
<form method="get">
<table style="border:1px solid #000000;width:800;">
  <#assign prerect2D>{leftBottom:{x:23,y:34},rightTop:{x:40,y:50}}</#assign>
  <tr><td>rect2D</td><td><textarea type="text" style="width:255px;height:50px" name="rect2D">${prerect2D}</textarea></td></tr>
  <tr><td>unit</td><td>
    <select name="unit">
      <option value="METER" selected="selected">METER</option>
      <option value="KILOMETER">KILOMETER</option>
    </select>
  </td></tr>
  <tr><td></td><td><input type="submit" value="rect2DareaMeasure"/></td></tr>
</table>
</form>
</p>
```

Add: 6/F, Building 107, No. A10, Jiuxianqiao North Road, Chaoyang District, Beijing, 100015, CHINA, 100015
E-mail: request@supermap.com   Website: www.supermap.com
2. Configuring the Algorithm

There are two ways to add the resource information to configuration file:

2.1. Add to external iserver-rest-resources.xml file

The configuration information of the extended resource needs to be placed in %SuperMap iServer_HOME%/webapps/iserver/WEB-INF/iserver-rest-resources.xml. Therefore, the resource can be found and published as REST service when the SuperMap iServer REST service starts. The structure of the iserver-rest-resources.xml file is as follows:

```xml
<resources>
    ......
    <resource>
        ......
    </resource>
</resources>
```

where a <resource/> node corresponds to an item of resource information. The <components/> node is used for domain component configuration and will be introduced in detail in Publishing Domain Services as Resources section.

The structure of the <resource/> nodes for the rectangleArea resource is as follows:

```xml
<resource>
    <configID>rectangleArea</configID>
    <urlTemplate>/maps/{mapName}/rectangleArea</urlTemplate>
    <resourceType>ArithmeticResource</resourceType>
    <implementClass>com.supermap.sample.extendREST.RectangleAreaResource</implementClass>
    <extensionEncoderBeanNames></extensionEncoderBeanNames>
    <extensionDecoderBeanNames></extensionDecoderBeanNames>
    <extensionHttpActionHandlerBeanName></extensionHttpActionHandlerBeanName>
</resource>
```

The <resource/> nodes are introduced in detail below:

- **configID:** The name of the resource.
- **urlTemplate:** The URI of the resource (the path after the root node).
- **resourceType:** The type of the resource. The resource types include ArithmeticResource, ArithResultResource, ArithResultSetResource, CatalogList, DomainArithmeticResource, DomainArithResultResource, StaticResource, etc. For more details, please refer to the ResourceType enumeration in the JavaDoc API section.
- **implementClass:** The implementation class of the resource. The class is included in the Jar package placed in %SuperMap iServer_HOME%/webapps/iserver/WEB-INF/lib, ExtendREST.jar for instance.
- **extensionEncoderBeanNames:** The name of the encoder component.
extensionDecoderBeanNames: The name of the decoder component.

extensionActionHandlerBeanName: The name of the HTTP request handler component.

Note: In the configuration, the extensionEncoderBeanNames, extensionDecoderBeanNames, extensionActionHandlerBeanName nodes, indicating the implementations of the extended encoder, the extended decoder, and the HTTP request handler respectively, can be absent or null. If it is the case, SuperMap iServer will employ the default values.

2.2. Add to function module XML file

This method can pack the configuration information into Jar, which is convenient to administrator to configure. This sample extends the map map module resources. Add the Jar here: //config/resource/rest/mappingResources.xml file (See The correspondence of the resource module and the configuration file) is as follows:

```xml
<resources>
  <resource>
    <configID>rectangleArea</configID>
    <urlTemplate>/maps/{mapName}/rectangleArea</urlTemplate>
    <resourceType>ArithmeticResource</resourceType>
    <implementClass>com.supermap.sample.extendREST.RectangleAreaResource</implementClass>
  </resource>
</resources>
```

You can configure multiple <resources/> in the <resource/> label. This step can be in the Java project that implements the resource class. It can be packed in the same Jar Package, such as the ExtendREST.jar.

Up till now, an extended resource is finished. The rectangleArea resource implements the area measurement of a rectangular area on a map. Start the service and use the URI below http://localhost:8090/iserver/services/components-rest/rest/maps/WorldMap/rectangleArea.html, taking the HTML output format as an example, to access the rectangleArea resource of the WorldMap, as shown below.

![Rectangle Area Resource](image-url)
Input the coordinates of the bottom left and top right points of the rectangle in the rect2D box, select a unit, and then click rect2DareaMeasure to display the result, as shown below:

![Rectangle area measurement interface](image)

**Extending an Encoder**

The REST Service Publishing Mechanism section gives you an introduction to the REST service publishing mechanism. We can know that when the HTTP request arrives at the REST application object and handled by the implementation class, the response needs to be processed by the encoder before the server sends the representation back to the client.

The encoders provided by SuperMap iServer include: TemplateEncoder (mainly used for HTML format representation), JsonEncoder, RJsonEncoder, XMLEncoder, ImageEncoder (BMP, GIF, JPEG, JPG, PNG, etc.), etc. The resource will select an encoder according to the specified representation format and to process the result. Then the result will be returned in the expected representation format.

SuperMap iServer also provides the extension mechanism for the encoder. If users want to publish the resource as a representation format not supported by SuperMap iServer, users need to extend an encoder.

The two steps for extending the encoder is as follows:

1. **Implementing an Encoder Class**

The REST implementation framework of SuperMap iServer provides the abstract class com.supermap.services.rest.encoders.Encoder, from which all encoders inherit. Encoders provided by SuperMap iServer include: ImageEncoder, JsonEncoder, SceneEncoder, StreamEncoder, TemplateEncoder, XMLEncoder, etc.

When extending the encoder, users need to inherit from the Encoder abstract class and implement the abstract methods in the class, or users can inherit from an existing encoder and overwrite the methods in it. Below is a list of some important methods involved.
To construct a custom encoder, inherit from the Encoder abstract class or its child class, and implement or overwrite the methods listed in the above table.

For illustration, here we implement MyEncoder similar to JsonEncoder, with the media type changed to application/cjson. The representation in cjson format is identical to that in json format. The only difference is that the media type in the HTTP response is application/cjson, but not application/json. In practical applications, users can define the specific implementation according to needs.

The hierarchy of MyEncoder is as follows:

```
  Encoder
   \___ JsonEncoder
        \___ MyEncoder
```

The sample code of MyEncoder:

```java
package com.supermap.sample.extendREST;
import java.util.ArrayList;
import java.util.List;
import org.restlet.data.MediaType;
import org.restlet.representation.Representation;
import com.supermap.services.rest.encoders.JsonEncoder;
public class MyEncoder extends JsonEncoder{
    public Representation toRepresentation(MediaType mediaType,
            Object resourceObj) {
        //Get the representation, which is identical to that in JSON format
        Representation rep = super.toRepresentation(mediaType, resourceObj);
        if(rep != null){
            //Set the media type in the HTTP response to application/cjson
            rep.setMediaType(new MediaType("application/cjson"));
        }
        return rep ;
    }
    protected List<MediaType> createSupportedMediaTypes() {
        List<MediaType> supportedMediaTypes = new ArrayList<MediaType>(); // NOPMD
        //The media type supported by MyEncoder is application/cjson
        supportedMediaTypes.add(new MediaType("application/cjson"));
        return supportedMediaTypes;
    }
}
```

Up till now, a simple encoder is finished. You can find the sample code in %SuperMap iServer_HOME%/samples/code/ExtendREST.
After compiling, the MyEncoder class needs to be packaged into a Jar package and place the package in %SuperMap iServer_HOME%/webapps/iserver/WEB-INF/lib. In this example, the MyEncoder class is packaged into extendREST.jar.

2. Configuring MyEncoder to REST Service
There are two ways for configuring MyEncoder:

2.1 Add to the external configuration file

2.2 Add to the XML configuration file of your own module

Please note that the server will choose an encoder according to the suffix of the request URI, and return the representation of the corresponding type. By default, the encoder is chosen according to the media type with the value "application/suffix". For instance, if the suffix of the URI is cjson, the server will choose the encoder that supports the media type “application/cjson”. Here MyEncoder will be found by the server. After MyEncoder handles the result, the server will return the response, with the media type being "application/cjson", to the client.

If you don't want the server to follow the corresponding relationship between the URI suffix and the media type, you can show the configuration on iServer. See The relations between URI suffix and media type.

2.1 Add to the external configuration file

How to configure MyEncoder to the REST service? First, register an implementation class to a component; Second, configure this component to the resources (you can either configure it to a single resource that possesses this encoder, or configure it to all resources that supports this encoder). The workflow is as follows:

To register the MyEncoder class as a Bean component, we need to add a <bean/> node in the REST application configuration file--AppContext.xml, which is located in %SuperMap iServer_HOME%/webapps/iserver/WEB-INF. As shown below, the MyEncoder class is registered as the MyEncoder Bean component:

<bean id="MyEncoder" class="com.supermap.sample.extendREST.MyEncoder"/>

Configuring for a single resource
The configuration of the encoder for a single resource should be performed in Resources.xml. The structure of the resource configuration information is introduced in step 2. Configuring the Algorithm for Extending a Simple Algorithm, where the <extensionEncoderBeanNames/> node under the <resource/> node is mentioned. Each resource configuration item, i.e., <resource/>, can have a child node <extensionEncoderBeanNames/>, specifying the extended encoders via the IDs of the registered Bean components. If there are multiple extended encoders, separate the IDs with ",". Below is the sample code for configuring MyEncoder for the rectangleArea resource:

<resource>
  <configID>rectangleArea</configID>
  <urlTemplate>/maps/{mapName}/rectangleArea</urlTemplate>
  <resourceType>ArithmeticResource</resourceType>
  <implementClass>com.supermap.sample.extendREST.rectangleAreaResource</implementClass>
  <extensionEncoderBeanNames>MyEncoder</extensionEncoderBeanNames>
  <extensionDecoderBeanNames/></extensionEncoderBeanNames>
Now, the rectangleArea resource has one more encoder--MyEncoder, therefore, the representation in application/cjson format is supported.

Restart the service, and access the URI below to simulate the GET request and get the representation in cjson format:

http://localhost:8090/iserver/services/components-rest/rest/maps/WorldMap/rectangleArea.cjson?rect2D={leftBottom:{x:23,y:34},rightTop:{x:40,y:50}}&unit=MET

Since application/cjson is a custom media type and IE cannot parse it, you will be prompted to save the file, as shown below:

After saving the file, open it with the text editor. Then you will find you get the measurement result, which is identical to the representation in json format.

**Configuring for all resources**

The configuration of the encoder for all resources should be performed in AppContext.xml, i.e., adding a default encoder. In AppContext.xml, the default encoder is specified by the <entry/> node of key="systemEncoders". The sample code for registering the MyEncoder component for all resources in AppContext.xml is shown as below:

```xml
<beans>
    ......  
    <util:map id="restConfig">
        <entry key="systemEncoders" value="xmlEncoder, jsonEncoder, pjsonEncoder, templateEncoder, MyEncoder"/>
    </util:map>
    ......  
</beans>
```

Now, all resources can use MyEncoder. The effect is similar to the cjson of rectangleArea.

Note: Resources.xml and AppContext.xml are located in %SuperMap iServer_HOME%/webapps/iserver/WEB-INF.

**2.2 Add to the XML configuration file of your own module**

In this way, the configuration information can be packaged to Jar so as to facilitate administrators to configure.

This sample extends the resource of the map module. The configure file is: Jar://META-
INF/extensions/services/rest/mapRest (See Configure file introduction).

mapRest configuration is shown as follows:

```
encoders=com.supermap.sample.extendREST.MyEncoder
decoders=
verifiers=
resourceFiles=config/resource/rest/mappingResources.xml
```

resourceFiles corresponds to the resource config file.

Jar://config/resource/rest/mappingResources.xml is configured as follows:

```
<resources>
  <resource>
    <configID>rectangleArea</configID>
    <urlTemplate>/maps/{mapName}/rectangleArea</urlTemplate>
    <resourceType>ArithmeticResource</resourceType>
    <implementClass>com.supermap.sample.extendREST.RectangleAreaResource</implementClass>
    <extensionEncoderBeanNames>MyEncoder</extensionEncoderBeanNames>
    <extensionDecoderBeanNames>MyEncoder</extensionDecoderBeanNames>
  </resource>
<resources>
```

This step can be packaged to the same Jar package.

Now, the extension of an Encoder is finished. The workflow of both the extension parameter parser and the extension HTTP request processor is similar to the encoder.

Note that MyEncoder is very simple. It has no practical meanings. Here we only focus on how to create an extension encoder.

### Extending a Decoder

The REST Service Publishing Mechanism section gives you an introduction to the REST service publishing mechanism. We can know that when the HTTP request arrives at the REST application object, the parameters in the HTTP request need to be parsed before the business logic handles the request. Parameter passing is performed by the Decoder, which can converts request parameters into java object.

The decoders provided by SuperMap iServer include: JsonDecoder for Json format parameter parsing and XMLDecoder for XML format parameter parsing. The parameter types are identified by the X-RequestEntity-ContentType and X-UrlEntity-ContentType request headers in the HTTP request. X-RequestEntity-ContentType and X-RequestEntity-ContentType identifies the type of the request body parameters and X-UrlEntity-ContentType identifies the type of the URI query parameters. If the values of the request headers are application/json, the HTTP request parameters are in Json format and will be parsed with JsonDecoder; If the values are application/xml, SuperMap iServer will parse the parameters with XMLDecoder. If the X-RequestEntity-ContentType or X-UrlEntity-ContentType request header is null, the server will use the default decoder JsonDecoder to parse the parameters.

SuperMap iServer also provides the extension mechanism for the decoder. If users want to use another format for HTTP request parameters, that is, a format other than Json and XML, users need to extend the decoder. The two steps for extending the decoder in SuperMap iServer are as follows:

1. Implementing a Decoder Class
2. Configuring the Decoder
1. Implementing a Decoder Class

The REST implementation framework of SuperMap iServer provides the abstract class com.supermap.services.rest.decoders.Decoder, from which all decoders inherit, to parse parameters. The decoders provided by SuperMap iServer include: JsonDecoder, XMLDecoder, etc.

When extending the decoder, users need to inherit from the Decoder abstract class and implement the abstract methods in the class, or users can inherit from an existing decoder and overwrite the methods in it. Below is a list of some important methods involved.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createSupportedMediaTypes()</td>
<td>Creates the list of media types supported by the decoder.</td>
</tr>
<tr>
<td>toObject(String, Class)</td>
<td>Converts the parameter string into Java object.</td>
</tr>
<tr>
<td>toList(String, Class)</td>
<td>Converts the parameter string into java.util.List object.</td>
</tr>
<tr>
<td>toMap(String, Map&lt;String, Class&gt;)</td>
<td>Converts the parameter string into java.util.Map mapping set.</td>
</tr>
<tr>
<td>toSet(String, Class)</td>
<td>Converts the parameter string into java.util.Set object.</td>
</tr>
</tbody>
</table>

To construct a custom decoder, inherit from the Decoder abstract class or its child class, and implement or overwrite the methods listed in the above table.

For illustration, here we implement the GET request on the distance resource and parse the custom parameters for the distance resource.

point2Ds=[(x= 23.00,y=34.00);(x= 53.55,y=12.66);(x= 73.88,y=12.6)]&unit=RADIAN

Note: Below is the normal Json format parameter.

point2Ds=[[x: 23.00,y:34.00],[x: 53.55,y:12.66],[x: 73.88,y:12.6]]&unit=RADIAN

The hierarchy of MyDecoder:

```
<table>
<thead>
<tr>
<th>Decoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>MYDECODER</td>
</tr>
</tbody>
</table>
```

The sample code of MyDecoder:

```java
package com.supermap.sample.extendREST;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TreeSet;
import org.restlet.data.MediaType;
```
import com.supermap.services.components.common.types.Point2D;
import com.supermap.services.rest.decoders.Decoder;
/**
 * Extend the decoder for the distance resource.
 * JSON foramt parameter: point2Ds=[{x: 23.00,y:34.00},{x: 53.55,y:12.66},{x: 73.88,y:12.6}]&unit=METER
 * Expected parsible parameter: point2Ds=[(x:23.00,y:34.00);(x:53.55,y:12.66);(x:73.88,y:12.6)]&unit=METER
 */
public class MyDecoder extends Decoder{
    @Override
    protected List<MediaType> createSupportedMediaTypes() {
        // TODO Auto-generated method stub
        List<MediaType> mediaTypes = new ArrayList<MediaType>();
        mediaTypes.add(new MediaType("application/custom"));
        return mediaTypes;
    }
    @Override
    public List toList(String text, Class elementClass) throws Exception {
        List result = null;
        Object obj = this.toObject(text, Point2D.class);
        if(obj == null){
            return null;
        }
        Class eleClz = obj.getClass().getComponentType();
        if(eleClz.equals(Point2D.class)){
            List list = new ArrayList();
            Point2D[] points = (Point2D[])(obj);
            for(Point2D point: points){
                list.add(point);
            }
            result = list;
        }
        return result;
    }
    @Override
    public Map<String, Object> toMap(String str, Map<String, Class> nameClassMapping) {
        Map<String, Object> mapping = new HashMap();
        try{
            mapping = (Map<String, Object>)this.toObject(str, Point2D.class);
        }catch(Exception e){
            e.printStackTrace();
        }
        return mapping;
    }
    @Override
    public Object toObject(String mytypeStr, Class targetClass) throws Exception {
        if(targetClass == null){
            throw new NullPointerException("param targetClass can not be null");
        }
        if(mytypeStr == null){
            return null;
        }
        mytypeStr = mytypeStr.trim();
        Object resultObj = null;
        //Convert the string
        //If mytypeStr starts with “[“ and ends with “]”, it is a point2Ds
if (mytypeStr.startsWith("[") && mytypeStr.endsWith("]")){
    mytypeStr = mytypeStr.substring(1);
    mytypeStr = mytypeStr.substring(0,mytypeStr.length() - 1);
    mytypeStr = mytypeStr.trim();
    if (mytypeStr.equals("")) {
        return new Object[0];
    } else {
        //There is only one Point2D in the array
        if (mytypeStr.lastIndexOf("\") == 0) {
            Point2D[] points = new Point2D[1];
            //
            Point2D point = (Point2D) this.toObject(mytypeStr, Point2D.class);
            points[0] = point;
            resultObj = points;
        } else {
            String[] objValueStrs = mytypeStr.split(";");
            Point2D[] points = new Point2D[objValueStrs.length];
            for (int i = 0; i < points.length; i++) {
                String objValueStr = objValueStrs[i];
                Point2D point = (Point2D) this.toObject(objValueStr, Point2D.class);
                points[i] = point;
            }
            resultObj = points;
        }
    }
} else if (Enum.class.isAssignableFrom(targetClass)) {
    //If targetClass is an enumeration, parse mytypeStr as an enumeration type
    //If targetClass is an Unit, converts mytypeStr into an Unit enumeration object
    resultObj = Enum.valueOf(targetClass, mytypeStr);
} else {
    Map<String, String> nameValueMapping = this.getMap(mytypeStr);
    if (nameValueMapping == null) {
        return null;
    }
    if (targetClass.equals(Point2D.class)) {
        if (nameValueMapping.size() == 0) {
            return new Point2D();
        } else {
            Point2D pp = new Point2D();
            String strX = nameValueMapping.get("x");
            String strY = nameValueMapping.get("y");
            try {
                double x = Double.parseDouble(strX);
                double y = Double.parseDouble(strY);
                pp.x = x;
                pp.y = y;
                resultObj = pp;
            } catch (NumberFormatException e) {
                e.printStackTrace();
            }
        } 
    }
}
return resultObj;

@Override
public Set toSet(String arg0, Class arg1) throws Exception {
    Set result = null;
    Object obj = this.toObject(arg0, Point2D.class);
    if(obj == null){
        return null;
    }
    Class eleClz = obj.getClass().getComponentType();
    if(eleClz.equals(Point2D.class)){
        Set set = new TreeSet();
        Point2D[] points = (Point2D[])(obj);
        for(Point2D point: points){
            set.add(point);
        }
        result = set;
    }
    return result;
}

private Map<String,String> getMap(String str){
    Map<String, String> mapping = new HashMap<String, String>();
    if(str == null){
        return null;
    }
    str = str.trim();
    if(str.startsWith("") && str.endsWith("")){
        str = str.substring(1);
        str = str.substring(0,str.length() -1);
        str = str.trim();
        if(str .equals("")){
            return new HashMap();
        }else{
            String[] entryStr = str .split(",");
            for(int i=0;i<entryStr.length;i++){
                entryStr[i]=entryStr[i].trim();
                if(entryStr[i] .indexOf(':' ) == -1){
                    continue;
                }
                String[] element=entryStr[i].split(":");
                if(element.length != 2){
                    System.out.println("String "+entryStr + "contains multiple: ");
                }
                if(element[1]!=null){
                    try {
                        mapping.put(element[0].trim(),element[1].trim());
                    } catch (Exception e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }
            }
        }
    }else{
        throw new IllegalArgumentException("Invalid string format");
    }
    return mapping;
}
Up till now, a simple decoder is finished. You can find the sample code in %SuperMap iServer_HOME%/samples/code/ExtendREST.

After compiling, the MyDecoder class needs to be packaged into a Jar package and place the package in %SuperMap iServer_HOME%/webapps/iserver/WEB-INF/lib. In this example, the DeEncoder class is packaged into extendREST.jar.

2. Configuring the Decoder

Configuring a custom decoder for the REST service is similar to Configuring the Encoder. Firstly, register the decoder as a Bean component. Then, configure for a single resource or all resources.

To register the MyDecoder class as a Bean component, we need to add a node in the REST application configuration file--i-server-rest-appContext.xml, which is located in %SuperMap iServer_HOME%/webapps/iserver/WEB-INF. As shown below, the MyDecoder class is registered as the MyDecoder Bean component:

```
<bean id="MyDecoder" class="com.supermap.sample.extendREST.MyDecoder"/>
```

The configuration of the decoder for a single resource should be performed in <util:map id="restConfig"/> of i-server-rest-appContext.xml. It is controlled by the <entry/> node of key="systemDecoders".

Notes: i-server-rest-resources.xml and i-server-rest-appContext.xml are located in %SuperMap iServer_HOME%/webapps/iserver/WEB-INF.

After registering MyEncoder as a Bean component named MyEncoder, it can only be configured to the distance resource. The configuration information of the distance resource is located in Resource.xml in %SuperMap iServer_HOME%/lib/iserver-all-{version}.jar/config/rest, as shown below.

```
<resource>
  <configID>distance</configID>
  <urlTemplate>/maps/{mapName}/distance</urlTemplate>
  <resourceType>ArithmeticResource</resourceType>
  <implementClass>com.supermap.services.rest.resources.impl.DistanceResource</implementClass>
  <extensionEncoderBeanNames></extensionEncoderBeanNames>
  <extensionDecoderBeanNames>MyDecoder</extensionDecoderBeanNames>
</resource>
```

Up till now, a custom decoder is finished.

Restart the service to let the distance resource accept parameters of specific format.

Please note that at the beginning of this page, it is introduced that the server chooses the decoder based on the values of the RequestEntity-ContentType and X-UrlEntity-ContentType request headers in the HTTP request. From the createSupportedMediaTypes method in the Sample Code, we know that MyDecoder can parse parameters in application/custom. Therefore, when implementing the GET request on the distance resource, the request header needs to be set to identify the media type of the parameter. Since the parameters are placed in the URI, not the request body, we need set the value of the X-UrlEntity-ContentType request header to application/custom, with the value of the X-RequestEntity-ContentType request header unchanged. Then the server will choose the MyDecoder to parse the parameters in the URI and return the correct result.

Using JavaScript to write client program, as shown below (please refer to Sample Overview in the Using REST API section):

```javascript
function getDistance()
{
```

---

Add: 6/F, Building 107, No. A10, Jiuxianqiao North Road, Chaoyang District, Beijing, 100015, CHINA, 100015
E-mail: request@supermap.com Website: www.supermap.com
SuperMap iServer correctly parses the parameter of a special format and returns the format as shown below:

You can find the source code for the client program in %SuperMap iServer_HOME%/samples/code/ExtendREST/distance.

**Extending an HTTP Request Handler**

The REST Service Publishing Mechanism section gives an introduction to the REST service publishing mechanism. We can know that when the HTTP request arrives at the REST application object, the process of handling the request is managed by the HTTP handler. SuperMap iServer provides the default HTTP request handler HTTP implementation (DefaultMethodHandler) to manage this process.

SuperMap iServer also provides the extension mechanism for the HTTP request handler, which gives users access to control the HTTP request handling process. The two steps for extending the HTTP request handler is as follows:

1. **Implementing an HTTP Request Handler Class**

The REST implementation framework of SuperMap iServer provides the abstract class com.supermap.services.rest.AbstractMethodHandler, from which all HTTP request handlers. DefaultMethodHandler is the default HTTP request handler provided by SuperMap iServer.

When extending the HTTP request handler, users need to inherit from the AbstractMethodHandler abstract class and implement the abstract methods in the class, or users can inherit from DefaultMethodHandler and overwrite the methods in it. Below is a list of some important methods involved.
For illustration, here we implement a simple HTTP request handler—MyMethodHandler, with the handleGET method implemented. The logic below is followed:

Do not read the expected representation format included in the HTTP request, and return the XML format representation for all GET requests.

The implementation of MyMethodHandler, which inherits from the AbstractMethodHandler abstract class, is as follows:

Sample code:

```java
package com.supermap.sample.extendREST;
import org.restlet.Request;
import org.restlet.Response;
import org.restlet.data.MediaType;
import org.restlet.data.Status;
import org.restlet.representation.Representation;
import com.supermap.services.rest.AbstractMethodHandler;
import com.supermap.services.rest.encoders.Encoder;
import com.supermap.services.rest.encoders.XMLEncoder;
import com.supermap.services.rest.resources.ResourceBase;
public class MyMethodHandler extends AbstractMethodHandler {
    @Override
    public void handleGet(ResourceBase targetResource, Request request, Response response) {
        // TODO Auto-generated method stub
        Encoder encoder = null;
        if (targetResource.isResourceExist()) {
            //Mandatory use the XML encoder
            encoder = new XMLEncoder();
            Object content = targetResource.getResourceContent();
            if (content != null) {
                //Return the XML format representation regardless of the request
```

---

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handleGet(ResourceBase, Request, Response)</td>
<td>Handles the GET request using the target resource.</td>
</tr>
<tr>
<td>handleHead(ResourceBase, Request, Response)</td>
<td>Handles the HEAD request using the target resource.</td>
</tr>
<tr>
<td>handlePost(ResourceBase, Request, Response)</td>
<td>Handles the POST request using the target resource.</td>
</tr>
<tr>
<td>handlePut(ResourceBase, Request, Response)</td>
<td>Handles the PUT request using the target resource.</td>
</tr>
<tr>
<td>handleDelete(ResourceBase, Request, Response)</td>
<td>Handles the DELETE request using the target resource.</td>
</tr>
<tr>
<td>handleOptions(ResourceBase, Request, Response)</td>
<td>Handles the OPTIONS request using the target resource.</td>
</tr>
</tbody>
</table>

---
Representation entity = encoder.toRepresentation(MediaType.TEXT_XML, content);
response.setEntity(entity);
response.setStatus(Status.SUCCESS_OK);
} else {
    // The resource is not represented.
    response.setStatus(Status.SUCCESS_NO_CONTENT);
} else {
    // Handle the resource not exist exception
    response.setStatus(Status.CLIENT_ERROR_NOT_FOUND);
}

@Override
public void handleHead(ResourceBase arg0, Request arg1, Response arg2) {
    // TODO Auto-generated method stub
}

@Override
public void handleOptions(ResourceBase arg0, Request arg1, Response arg2) {
    // TODO Auto-generated method stub
}

@Override
public void handlePost(ResourceBase arg0, Request arg1, Response arg2) {
    // TODO Auto-generated method stub
}

@Override
public void handlePut(ResourceBase arg0, Request arg1, Response arg2) {
    // TODO Auto-generated method stub
}

@Override
public void handleDelete(ResourceBase arg0, Request arg1, Response arg2) {
    // TODO Auto-generated method stub
}

}

Up till now, an HTTP request handler class is finished. You can find the sample code for this example in %SuperMap IServer_HOME%/samples/code/ExtendREST.

After compiling the MyMethodHandler class, we need to package it into a Jar package and put the Jar package in %SuperMap IServer_HOME%/webapps/iserver/SERVER-INF/lib. In this example, the class is packaged into extendREST.jar.

2. Configuring the HTTP Request Handler

The process for configuring a custom HTTP request handler to the REST service is similar to Configuring the Encoder. First, register a Bean component, and then configure for a single resource or all resources.

Please note that the node configured in iserver-rest-resources.xml turns to <extensionHttpActionHandlerBeanName/>. And when configuring <util:map id="restConfig"/> in iserver-rest-appContext.xml, the key value of <entry/> is <entry/> of defaultHttpActionHandler.

After registering MyMethodHandler as a Bean component named MyHTTPHandler, configure MyHTTPHandler for the maps resource, as shown below. The configuration information for the maps resource is in mappingResources.xml in %SuperMap IServer_HOME%/lib/iserver-all-{version}.jar/config/rest.

<resource>
    <configID>maps</configID>
    <urlTemplate>/maps</urlTemplate>
    <resourceType>CatalogList</resourceType>
</resource>
Up till now, the extension of an HTTP request handler is finished. Note that a resource can have multiple encoders and decoders, but only one HTTP request handler.

Restart the service, and access the URI below:

http://localhost:8090/iserver/services/components-rest/rest/maps.json

You will find the response is in XML format, but not the expected json format. That is because the default HTTP request handler has been replaced with the custom MyHTTPHandler, which mandatorily generates the XML format representation when handling the GET request.

In practical situations, you can extend the HTTP request handler to change the HTTP request handling process, add additional business logic, etc.

**Extending a Security Mechanism**