



Practical introduction to ZOO: The Open WPS Platform



FOSS4G 2010
Barcelona
Free and Open Source Software
for Geospatial CONFERENCE
SEP 6th - 9th

FOSS4G 2010 WORKSHOP



Mon 6 Sep 15-18h - C6.S309
Facultat d'Informàtica de Barcelona
Barcelona Campus Nord



General information

Workshop site:	Facultat d'Informàtica de Barcelona Barcelona Campus Nord
Classroom:	C6.S309
Total length:	3h
Instructors:	Gérald FENOY (GeoLabs) gerald.fenoy@geolabs.fr Nicolas BOZON (3LIZ) nbozon@3liz.com

Prerequisites

OSGeoLive Virtual Machine image disk (including pre-compiled ZOO 1.0 and data sample)

Basic knowledge of OGC Web Processing Service specification (WPS 1.0)

Basic knowledge of C and/or Python languages

Basics knowledge of JavaScript and OpenLayers

Useful links

ZOO Workshop HTML version: <http://zoo-project.org/trac/wiki/ZooWorkshopFoss4g2010>

ZOO Workshop PDF version: <http://zoo-project.org/dl/foss4g2010ws.pdf>

ZOO Project official website: <http://zoo-project.org>

ZOO Project official Trac: <http://zoo-project.org/trac>

OGC WPS 1.0 specification: <http://opengeospatial.org/standards/wps>

OSGeoLive official website: <http://live.osgeo.org/>

GDAL/OGR official website: <http://www.gdal.org/>

OpenLayers official website: <http://openlayers.org>

Contents

1) Introduction.....	4
1.1) What is ZOO?.....	4
1.2) How does ZOO work?.....	4
1.3) What are we going to do in this workshop?.....	5
2) Using ZOO from an OSGeoLive virtual machine.....	6
2.1) Development environment description.....	6
2.2) Testing the ZOO installation with GetCapabilities.....	8
2.3) Preparing your ZOO ServiceProvider directory.....	9
3) Creating WPS compliant OGR based Web Services.....	10
3.1) Introduction.....	10
3.2) Preparing ZOO metadata file.....	10
3.3) Implementing single geometry processes.....	14
3.3.1) Boundary.....	14
3.3.1.1) C Version.....	14
3.3.1.2) Python Version.....	20
3.3.1.3) Testing the Service using Execute Request.....	21
3.3.2) Creating Services for other functions (ConvexHull and Centroid).....	24
3.3.2.1) C Version.....	24
3.3.2.2) Python Version.....	26
3.3.3) Create the Buffer Service.....	28
3.3.3.1) C Version.....	28
3.3.3.2) Python Version.....	30
3.3.3.3) The Buffer MetadataFile file.....	30
4) Building a WPS client using OpenLayers.....	32
4.1) Creating a simple map showing the dataset as WMS.....	32
4.2) Fetching the data layer as WFS and adding selection controls.....	33
4.3) Calling the single geometrie processes from JavaScript.....	34
4.4) Calling the multiples geometries processes from JavaScript.....	36
5) Exercise.....	38
5.1) C version.....	38
5.2) Python Version.....	39
5.3) Testing your services.....	39

1 Introduction

1.1 What is ZOO?

ZOO is a WPS (Web Processing Service) open source project recently released under a [MIT/X-11](#) style license. It provides an OGC WPS compliant developer-friendly framework to create and chain WPS Web services. ZOO is made of three parts:

- ✓ **ZOO Kernel**: A powerful server-side C Kernel which makes it possible to manage and chain Web services coded in different programming languages.
- ✓ **ZOO Services**: A growing suite of example Web Services based on various open source libraries.
- ✓ **ZOO API**: A server-side JavaScript API able to call and chain the ZOO Services, which makes the development and chaining processes easier.

ZOO is designed to make the WPS server-side development easier by providing a powerful system able to understand and execute WPS compliant queries. It supports several programming languages, thus allowing you to create Web Services in your favorite language and from existing code. Further information on the project is available on the [ZOO Project official website](#).

1.2 How does ZOO work?

ZOO is based on a 'WPS Service Kernel' which constitutes the ZOO's core system (aka ZOO Kernel). The latter is able to load dynamic libraries and to handle them as on-demand Web services. The **ZOO Kernel** is written in C language, but supports several common programming languages for creating **ZOO Services**.

A **ZOO Service** is a link composed of a ZOO metadata file (.zcfg) and the code for the corresponding implementation. The metadata file describes all the available functions which can be called using a WPS Exec Request, as well as the desired input/output. Services contain the algorithms and functions, and can now be implemented in C/C++, Fortran, Java, Python, PHP and JavaScript.

ZOO Kernel works with Apache and can communicate with cartographic engines and Web mapping clients. It simply adds the WPS support to your spatial data infrastructure and your Web mapping application. It can use every GDAL/OGR supported formats as input data and create suitable vector or raster output for your cartographic engine and/or your web-mapping client application.

1.3 What are we going to do in this workshop?

This workshop aims to present the ZOO Project and its features, and to explain its capabilities regarding the WPS 1.0.0 specification. The participants will learn in 3 hours how to use ZOO Kernel, how to create ZOO Services and their configuration files and finally how to link the created Service with a client-side webmapping application.

A pre-compiled ZOO 1.0 version is provided inside OSGeoLive, the OSGeo official Live DVD. For the sake of simplicity, an OSGeoLive Virtual Machine image disk is already installed on your computers. This will be used during this workshop, so the participants won't have to compile and install ZOO Kernel manually. Running and testing ZOO Kernel from this OSGeoLive image disk is thus the first step of the workshop, and every participants should get a working ZOO Kernel in less than 30 minutes.

Once ZOO Kernel will be tested from a Web browser using GetCapabilities requests, participants will be invited to create an OGR based ZOO Service Provider aiming to enable simple spatial operations on vector data. Participants will first have to choose whether they will create the service using C or Python language. Every programming step of the ZOO Service Provider and the related Services will be each time detailed in C and Python.

Once the ZOO Services will be ready and callable by ZOO Kernel, participants will finally learn how to use its different functions from an OpenLayers simple application. A sample dataset from Geoserver will be displayed on a simple map using WMS/WFS standards and used as input data by the ZOO Services. Then, some specific selection and execution controls will be added in the JavaScript code in order to execute single and multiple geometries on the displayed polygons.

Once again, the whole procedure will be organized step-by-step and detailed with numerous code snippets and their respective explanations. The instructors will check the ZOO Kernel functioning on each machine and will assist you while coding. Technical questions are of course welcome during the workshop.

Usefull tips for reading:

Codes snippets are included in yellow blocks

Code changes are included in grey blocks

Code snippets included in sentences are displayed this way

HTTP and XML Requests are included in blue blocks

Let's go !

2 Using ZOO from an OSGeoLive virtual machine

2.1 Development environment description

OSGeoLive is a live DVD and virtual machine based on **Xubuntu** that allows you to try a wide variety of open source geospatial software without installing anything. It is composed entirely of free software and include ZOO 1.0 this year, for testing purpose.

As already said in introduction, an OSGeoLive virtual machine image disk has been installed on your computer, allowing you to use ZOO Kernel in a development environment directly. Using a virtual machine image disk seems to be the simplest way to use ZOO Kernel and to develop ZOO Services locally, as we can ensure that everything requested for compile C Services and run Python Services is available and ready to use. Every ZOO related material and source code have been placed in `/home/user/zoows` directory. We will work inside it during this workshop. As the binary version of ZOO Kernel is already compiled and stored in `/home/user/zoows/sources/zoo-kernel`, you only have to copy two important files inside the `/usr/lib/cgi-bin` directory : `zoo_loader.cgi` and the `main.cfg` in order to make ZOO Kernel available, using the following commands:

```
sudo cp ~/zoows/sources/zoo-kernel/zoo_loader.cgi /usr/lib/cgi-bin
sudo cp ~/zoows/sources/zoo-kernel/main.cfg /usr/lib/cgi-bin
```

Please note that we will talk about ZOO Kernel or `zoo_loader.cgi` script without any distinction during this workshop .

The `main.cfg` file contains metadata informations about the identification and provider but also some important settings. The file is composed of various sections, namely `main`, `identification` and `provider` per default. Obviously, you are free to add new sections to the files if you need them for a specific Service. Nevertheless, you have to know that the `env` section name is used in a specific way. It lets you define environment variables that your Service requires during its runtime. For instance, if your Service requires to access to a X server running on framebuffer, you can add `DISPLAY=:1` line in your `env` section to take this specificity into account.

Please have a look to this file. Three important parameters are commented bellow:

- ✓ `serverAddress` : The url to access to the ZOO Kernel
- ✓ `tmpPath` : The full path to store temporary files
- ✓ `tmpUrl` : The url path relative to `serverAddress` to access temporary directory.

The values of the `main.cfg` file used from the running virtual machine are the following :

```
serverAddress=http://localhost/zoo
```

```
tmpPath=/var/www/temp
tmpUrl=../temp/
```

You could have noticed that the `tmpUrl` is a relative url from `serverAddress`, so it must be a directory. Even if ZOO Kernel can be used with the full url of the `zoo_loader.cgi` script, for better readability and fully functional ZOO Kernel, you have to modify the default Apache configuration in order to be able to use the <http://localhost/zoo/> url directly.

First, please create a `zoo` directory in the existing `/var/www` which is used by Apache as the `DirectoryIndex`. Then, please edit the `/etc/apache2/sites-available/default` configuration file and add the following lines after the `Directory` block related to `/var/www` directory :

```
<Directory /var/www/zoo/>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride All
    Order allow,deny
    allow from all
</Directory>
```

Now create a small `.htaccess` file in the `/var/www/zoo` containing the following lines:

```
RewriteEngine on
RewriteRule (.*)/(.*) /cgi-bin/zoo_loader.cgi?metapath=$1 [L,QSA]
RewriteRule (.*) /cgi-bin/zoo_loader.cgi [L,QSA]
```

For this last file to be taken into account by Apache, you must activate the rewrite Apache module by copying a load file as bellow :

```
sudo cp /etc/apache2/mods-available/rewrite.load /etc/apache2/mods-enabled/
```

Now you should be able to access the ZOO Kernel using a simplified by restarting your Apache Web server :

```
sudo /etc/init.d/apache2 restart
```

Two other softwares from the OSGeoLive environment will be used during this workshop. Geoserver will first be used to provide WFS input data for the ZOO Services we are going to develop. The Geoserver sample dataset (United States polygons) will be passed to our service during section 3. So please start the Geoserver using the corresponding launcher in the `Servers` folder, as illustrated in the following screenshot :



OpenLayers library is also available on the OSGeoLive virtual machine image disk, and it will be used during section 4, for building a simple WPS client application able to query the newly developed ZOO Services.

As we planned to use OGR C-API and Python module of the GDAL library, we will need the corresponding header files, libraries and associated files. Hopefully everything was already available per default and so ready to use on the OSGeoLive packaging.

2.2 Testing the ZOO installation with GetCapabilities

You can now simply query ZOO Kernel using the following request from your Internet browser:

```
http://localhost/cgi-bin/zoo_loader.cgi?Request=GetCapabilities&Service=WPS
```

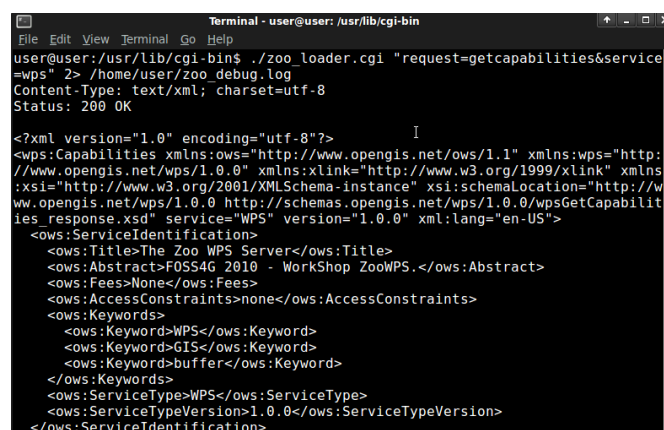
You should then get a valid Capabilities XML document, as the following :

```
-<wps:Capabilities xsi:schemaLocation="http://www.opengis.net/wps/1.0.0 http://schemas.opengis.net/wps/1.0.0/wpsGetCapabilities_response.xsd" service="WPS" version="1.0.0" xml:lang="en-US">
  -<ows:ServiceIdentification>
    <ows:Title>The Zoo WPS Server</ows:Title>
    <ows:Abstract>FOSS4G 2010 - WorkShop ZooWPS.</ows:Abstract>
    <ows:Fees>None</ows:Fees>
    <ows:AccessConstraints>none</ows:AccessConstraints>
  -<ows:Keywords>
    <ows:Keyword>WPS</ows:Keyword>
    <ows:Keyword>GIS</ows:Keyword>
    <ows:Keyword>buffer</ows:Keyword>
  </ows:Keywords>
  <ows:ServiceType>WPS</ows:ServiceType>
  <ows:ServiceTypeVersion>1.0.0</ows:ServiceTypeVersion>
</ows:ServiceIdentification>
```

Please note that no Process node is returned in the ProcessOfferings section, as no ZOO Service is available yet. You can also proceed to a GetCapabilities request from the command line, using the following command:

```
./zoo_loader.cgi "request=GetCapabilities&service=WPS"
```

The same result as in your browser will be returned, as shown in the following screenshot:



```
Terminal - user@user: /usr/lib/cgi-bin
user@user:/usr/lib/cgi-bin$ ./zoo_loader.cgi "request=getcapabilities&service=wps" 2> /home/user/zoo debug.log
Content-Type: text/xml; charset=utf-8
Status: 200 OK

<?xml version="1.0" encoding="utf-8"?>
<wps:Capabilities xmlns:ows="http://www.opengis.net/ows/1.1" xmlns:wps="http://www.opengis.net/wps/1.0.0" xmlns:xlink="http://www.w3.org/1999/xlink" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.opengis.net/wps/1.0.0 http://schemas.opengis.net/wps/1.0.0/wpsGetCapabilities_response.xsd" service="WPS" version="1.0.0" xml:lang="en-US">
  <ows:ServiceIdentification>
    <ows:Title>The Zoo WPS Server</ows:Title>
    <ows:Abstract>FOSS4G 2010 - WorkShop ZooWPS.</ows:Abstract>
    <ows:Fees>None</ows:Fees>
    <ows:AccessConstraints>none</ows:AccessConstraints>
  <ows:Keywords>
    <ows:Keyword>WPS</ows:Keyword>
    <ows:Keyword>GIS</ows:Keyword>
    <ows:Keyword>buffer</ows:Keyword>
  </ows:Keywords>
  <ows:ServiceType>WPS</ows:ServiceType>
  <ows:ServiceTypeVersion>1.0.0</ows:ServiceTypeVersion>
</ows:ServiceIdentification>
```

2.3 Preparing your ZOO ServiceProvider directory

In order to simplify the task, we will first comment the directory structure which should be used when creating a new Services Provider :

- ✓ The main Services Provider directory including :
 - ✓ A `cgi-env` directory which will contain all the `zcfg` metadata files and the service shared object (C Shared Library or Python module)
 - ✓ The `Makefile` and the `*c` files needed to compile your Services Provider.

Please create a `ws_sp` main Services Provider directory in the existing `zoo-services` one located in `/home/user/zoows/sources/`, respecting the tree above .

```
mkdir -p /home/user/zoows/sources/zoo-services/ws_sp/cgi-env
```

The `Makefile` and the code of the C and Python Service Shared Object will be detailed in the next sections.

3 Creating WPS compliant OGR based Web Services

3.1 Introduction

In this part, we are going to create a ZOO ServiceProvider containing several Services based on the OGR C API or on the OGR Python module, which have also been placed in the ZOO installation on OSGeoLive. The intended goal is to use OGR and its GEOS based simple spatial functions as WPS Services.

We will first start with the Boundary spatial function, which will be explained, coded and tested gradually as a ZOO Service. The same procedure will then be used to enable the Buffer, Centroid and Convex Hull functions. Once done, some multiple geometries processes such as Intersection, Union, Difference and Symetric Difference will be implemented through an exercise at the end of the workshop.

As already said in the introduction, you have the choice to code your service in C or Python (or both!) during this workshop. Explanations will be based on the C part, but will be very helpful for those who will choose Python. Please decide according to your habits and preferences and tell your choice to the instructors. The results will be the same in both case.

3.2 Preparing ZOO metadata file

A **ZOO Service** is a combination of a ZOO metadata file (.zcfg) and the runtime module for the corresponding implementation, which is commonly called ZOO Service Provider. We will first prepare a .zcfg file step-by-step. Please open your preferred text editor and edit a file named `Boundary.zcfg` in your `/home/user/zoows/sources/zoo-services/ws_sp` directory. First, you need to name the service between brackets at the top of the file, as the following:

```
[Boundary]
```

This name is very important, it is the name of the Service and so the name of the function defined in the Services Provider. A title and a brief abstract must then be added to inform clients on what the service can do :

```
Title = Compute boundary.  
Abstract = Returns the boundary of the geometry on which the method is invoked.
```

Such metadata informations will be returned by a `GetCapabilities` request.

You can also add other specific informations like the `processVersion`. You can set if your ZOO Service can store its results, by setting the `storeSupported` parameter to true or false. You can also decide if the function can be run as a background task and inform on its current status, according to the `statusSupported` value :

```
processVersion = 1  
storeSupported = true  
statusSupported = true
```

In the main section of the ZOO Service metadata file, you must also specify two important things:

- ✓ `serviceProvider`, which is the name of the C shared library containing the Service function or the Python module name.
- ✓ `serviceType`, which defines the programming language to be used for the Service. (value can be C or Python depending on what language you have decided to use)

C ServiceProvider Example :

```
serviceProvider=ogr_ws_service_provider.zo
serviceType=C
```

In this case you will get an `ogr_ws_service_provider.zo` shared library containing the Boundary function, placed in the same directory than ZOO Kernel.

Python ServiceProvider Example :

```
serviceProvider=ogr_ws_service_provider
serviceType=Python
```

In this case, you will get an `ogr_ws_service_provider.py` file containing the Python code of your Boundary function.

In the main section you can also add any other metadata information, as the following:

```
<MetaData>
  Title = Demo
</MetaData>
```

The main metadata informations have been declared, so you can now define data input which will be used by the ZOO Service. You can define any input needed by the Service. Please note that you can request ZOO Kernel using more data input than defined in the `.zcfg` file without any problem, those values will be passed to your service without filtering. In the Boundary Service example, a single polygon will be used as input, the one on which to apply the Boundary function.

The data input declarations are included in a `DataInputs` block. They use the same syntax as the Service itself and the input name is between brackets. You can also fill a title, an abstract and a `MetaData` section for the input. You must set values for the `minOccurs` and `maxOccurs` parameters, as they will inform ZOO Kernel which parameters are required to be able to run the Service function.

```
[InputPolygon]
Title = Polygon to compute boundary
Abstract = URI to a set of GML that describes the polygon.
minOccurs = 1
maxOccurs = 1
<MetaData lang="en">
  Test = My test
</MetaData>
```

The metadata defines what type of data the Service supports. In the Boundary example, the input polygon can be provided as a GML file or as a JSON string. Next step is thus to define the default and supported input formats. Both formats should be declared in a `LiteralData` or `ComplexData` block depending on their types. For this first example we will use `ComplexData` blocks only.

```
<ComplexData>
  <Default>
    mimeType = text/xml
    encoding = UTF-8
  </Default>
  <Supported>
    mimeType = application/json
    encoding = UTF-8
  </Supported>
</ComplexData>
```

Then, the same metadata information must be defined for the output of the Service, inside a `DataOutputs` block, as the following:

```
[Result]
  Title = The created geometry
  Abstract = The geometry containing the boundary of the geometry on which the method
was invoked.
  <MetaData lang="en">
    Title = Result
  </MetaData>
  <ComplexData>
    <Default>
      mimeType = application/json
      encoding = UTF-8
    </Default>
    <Supported>
      mimeType = text/xml
      encoding = UTF-8
    </Supported>
  </ComplexData>
```

A complete copy of this `.zcfg` file can be found at the following URL :

<http://zoo-project.org/trac/browser/trunk/zoo-services/ogr/base-vect-ops/cgi-env/Boundary.zcfg>

Once the ZOO metadata file is modified, you have to copy it in the same directory than your ZOO Kernel (so in your case `/usr/lib/cgi-bin`). Then you should be able to run the following request :

<http://localhost/zoo/?Request=DescribeProcess&Service=WPS&Identifier=Boundary&version=1.0.0>

The returned ProcessDescriptions XML document should look like the following :

```
-<wps:ProcessDescriptions xsi:schemaLocation="http://www.opengis.net/wps/1.0.0 http://schemas.opengis.net/wpsDescribeProcess_response.xsd" service="WPS" version="1.0.0" xml:lang="en">
  -<ProcessDescription wps:processVersion="1" storeSupported="true" statusSupported="true">
    <ows:Identifier>Boundary</ows:Identifier>
    <ows:Title>Compute boundary.</ows:Title>
    -<ows:Abstract>
      A new geometry object is created and returned containing the boundary of the geometry on which the me
    </ows:Abstract>
    <ows:Metadata xlink:Test="Demo"/>
    -<DataInputs>
      -<Input minOccurs="1" maxOccurs="1">
        <ows:Identifier>InputPolygon</ows:Identifier>
        <ows:Title>Polygon to compute boundary</ows:Title>
        <ows:Abstract>URI to a set of GML that describes the polygon.</ows:Abstract>
      -<ComplexData>
```

Please note that the `GetCapabilities` and `DescribeProcess` only need a `.zcfg` file to be completed. **Simple, isn't it ?** At this step, if you request ZOO Kernel for an `Execute`, you will get an `ExceptionReport` document as response, looking as the following :

```
-<ows:ExceptionReport xsi:schemaLocation="http://www.opengis.net/ows/1.1 http://schemas.opengis.net/ows/1.1.0/owsExceptionReport.xsd" xml:lan="en" service="WPS" version="1.0.0">
  -<ows:Exception exceptionCode="NoApplicableCode">
    -<ows:ExceptionText>
      C Library can't be loaded /usr/lib/cgi-bin//ogr_ws_service_provider.zo: cannot open shared object file: No such file or directory
    </ows:ExceptionText>
  </ows:Exception>
</ows:ExceptionReport>
```

A similar error message will be returned if you try to run your Python Service :

```
-<ows:ExceptionReport xsi:schemaLocation="http://www.opengis.net/ows/1.1 http://schemas.opengis.net/ows/1.1.0/owsExceptionReport.xsd" xml:lan="en" service="WPS" version="1.0.0">
  -<ows:Exception exceptionCode="NoApplicableCode">
    -<ows:ExceptionText>
      Python module ogr_ws_service_provider cannot be loaded.
    </ows:ExceptionText>
  </ows:Exception>
</ows:ExceptionReport>
```

3.3 Implementing single geometry processes

In order to learn the Services Provider creation and deployment step-by-step, we will first focus on creating a very simple one dedicated to the `Boundary` function. Similar procedure will then be used for the `Buffer`, `Centroid` and `ConvexHull` implementation.

Your metadata is now ok, so you now must create the code of your Service. The most important thing you must be aware of when coding ZOO Services is that the function corresponding to your Service takes three parameters (internal `maps` datatype or **Python dictionaries**) and returns an integer value representing the status of execution (`SERVICE_FAILED` or `SERVICE_SUCCEEDED`):

- ✓ `conf` : The main environment configuration (corresponding to the `main.cfg` content)
- ✓ `inputs` : The requested / default inputs
- ✓ `outputs` : The requested / default outputs

3.3.1 Boundary

3.3.1.1 C Version

As explained before, ZOO Kernel will pass the parameters to your Service function in a specific datatype called `maps`. In order to code your Service in C language, you also need to learn how to access this datatype in read/write mode.

The `maps` are simple `map` named linked list containing a name, a content `map` and a pointer to the next `map` in the list (or `NULL` if there is no more `map` in the list). Here is the datatype definition as you can find in the `zoo-kernel/service.h` file :

```
typedef struct maps{
    char* name;
    struct map* content;
    struct maps* next;
} maps;
```

The `map` included in the `maps` is also a simple linked list and is used to store Key Value Pair values. A `map` is thus a couple of name and value and a pointer to the next element in the list. Here is the datatype definition you can find in the `zoo-kernel/service.h` file :

```
typedef struct map{
    char* name; /* The key */
    char* value; /* The value */
    struct map* next; /* Next couple */
} map;
```

As partially or fully filled datastructures will be passed by the ZOO Kernel to your Services, this means that you do not need to deal with `maps` creation but directly with existing `map`, in other words the content of each `maps`. The first function you need to know is `getMapFromMaps` (defined in the `zoo-kernel/service.h` file) which let you access to a specific `map` of a `maps`.

This function takes three parameters listed bellow :

- ✓ `m` : a `maps` pointer representing the `maps` used to search the specific `map`
- ✓ `name` : a `char*` representing the name of the `map` you are searching for
- ✓ `key` : a specific key in the `map` named `name`

For example, the following syntax will be used to access the `InputPolygon` value `map` of a `maps` named `inputs`, your C code should be :

```
map* tmp=getMapFromMaps(inputs,"InputPolygon","value");
```

Once you get the `map`, you can access the `name` or the `value` fields, using the following syntax :

```
tmp->name  
tmp->value
```

As you know how to read and access the `map` fields from a `maps`, you can now learn how to write in such a datastructure. This is done by using the simple `addToMap` function once again defined in `zoo-kernel/service.h`. The `addToMap` function also takes three parameters :

- ✓ `m` : a `map` pointer you want to update,
- ✓ `n` : the name of the `map` you want to add or update the value,
- ✓ `v` : the value you want to set for this `map`.

Here is an example of how to add or edit the `content` `map` of a `maps` called `outputs` :

```
addToMap(outputs->content,"value","Hello from the C World !");  
addToMap(outputs->content,"mimeType","text/plain");  
addToMap(outputs->content,"encoding","UTF-8");
```

Please note that the `addToMap` function is able to create or update an existing `map`. Indeed, if a `map` called « `value` » allready exists, then its value will be updated automatically.

This datatype is really important cause it is used in every C based ZOO Services. It is also the same representation used in other languages but using their respectives datatypes. For Example in Python, the `dictionaries` datatype is used, so manipulation is much easier.

Here is an example of a maps used in Python language (this is a summarized version of the main configuration maps) :

```
main={
  "main": { "encoding": "utf-8",
            "version": "1.0.0",
            "serverAddress": "http://www.zoo-project.org/zoo/",
            "lang": "fr-FR,en-CA"},
  "identification": { "title": "The Zoo WPS Development Server",
                     "abstract": "Development version of ZooWPS.",
                     "fees": "None",
                     "accessConstraints": "none",
                     "keywords": "WPS, GIS, buffer"}
}
```

As you know how to deal with maps and map, you are ready to code the first ZOO Service by using the OGR Boundary function.

As already said in introduction we will use the Geoserver WFS server available on OSGeoLive, so full WFS Response will be used as inputs values. As we will use the simple OGR Geometry functions like [OGR_G_GetBoundary](#), only the Geometry object will be used rather than a full WFS Response. The first thing to do is to write a function which will extract the geometry definition from the full WFS Response. We will call it `createGeometryFromWFS`.

Here is the code of such a function :

```
OGRGeometryH createGeometryFromWFS (maps* conf, char* inputStr) {
  xmlInitParser();
  xmlDocPtr doc = xmlParseMemory(inputStr, strlen(inputStr));
  xmlChar *xmlbuff;
  int buffersize;
  xmlXPathContextPtr xpathCtx;
  xmlXPathObjectPtr xpathObj;
  char * xpathExpr="/*/*/*/*/*[local-name()='Polygon' or local-name()='MultiPolygon']";
  xpathCtx = xmlXPathNewContext(doc);
  xpathObj = xmlXPathEvalExpression(BAD_CAST xpathExpr, xpathCtx);
  if(!xpathObj->nodelist) {
    errorException(conf, "Unable to parse Input Polygon", "InvalidParameterValue");
    exit(0);
  }
  int size = (xpathObj->nodelist) ? xpathObj->nodelist->nodeNr : 0;
  xmlDocPtr ndoc = xmlNewDoc(BAD_CAST "1.0");
  for(int k=size-1; k>=0; k--){
    xmlDocSetRootElement(ndoc, xpathObj->nodelist->nodeTab[k]);
  }
  xmlDocDumpFormatMemory(ndoc, &xmlbuff, &buffersize, 1);
  char *tmp=strdup(strstr((char*)xmlbuff, ">>")+2);
  xmlXPathFreeObject(xpathObj);
  xmlXPathFreeContext(xpathCtx);
  xmlFree(xmlbuff);
  xmlFreeDoc(doc);
  xmlCleanupParser();
  OGRGeometryH res=OGR_G_CreateFromGML(tmp);
  if(res==NULL){
    map* tmp=createMap("text", "Unable to call OGR_G_CreatFromGML");
    addToMap(tmp, "code", "NoApplicableCode");

    exit(0);
  }
  else
```

```
return res;
}
```

The only thing we will focus on is the call to the `errorException` function used in the function body. This function is declared in the `zoo-kernel/service_internal.h` and defined in `zoo-kernel/service_internal.c` file. It takes three parameters as follow :

- ✓ the main environment maps,
- ✓ a `char*` representing the error message to display,
- ✓ a `char*` representing the error code (as defined in the WPS specification – Table 62).

In other words, if the WFS response cannot be parsed properly, then you will return an `ExceptionReport` document informing the client that a problem occurred.

The function to extract the geometry object from a WFS Response is written, so you can now start defining the Boundary Service. Here is the full code for the Boundary Service :

```
int Boundary (maps*& conf, maps*& inputs, maps*& outputs) {
    OGRGeometryH geometry, res;
    map* tmp=getMapFromMaps (inputs, "InputPolygon", "value");
    if (tmp==NULL)
        return SERVICE_FAILED;
    map* tmp1=getMapFromMaps (inputs, "InputPolygon", "mimeType");
    if (strncmp (tmp1->value, "application/json", 16)==0)
        geometry=OGR_G_CreateGeometryFromJson (tmp->value);
    else
        geometry=createGeometryFromWFS (conf, tmp->value);
    res=OGR_G_GetBoundary (geometry);
    tmp1=getMapFromMaps (outputs, "Result", "mimeType");
    if (strncmp (tmp1->value, "application/json", 16)==0) {
        addToMap (outputs->content, "value", OGR_G_ExportToJson (res));
        addToMap (outputs->content, "mimeType", "text/plain");
    }
    else{
        addToMap (outputs->content, "value", OGR_G_ExportToGML (res));
    }
    outputs->next=NULL;
    OGR_G_DestroyGeometry (geometry);
    OGR_G_DestroyGeometry (res);
    return SERVICE_SUCCEEDED;
}
```

As you can see in the code above, the `mimeType` of the data inputs passed to our Service is first checked :

```
map* tmp1=getMapFromMaps (inputs, "InputPolygon", "mimeType");
if (strncmp (tmp1->value, "application/json", 16)==0)
    geometry=OGR_G_CreateGeometryFromJson (tmp->value);
else
    geometry=createGeometryFromGML (conf, tmp->value);
```

Basically, if we get an input with a `mimeType` set to `application/json`, then we will use our `OGR_G_CreateGeometryFromJson` in other case, our `createGeometryFromGML` local function.

Please note that in some sense the data inputs are not really of the same kind. Indeed as we used directly `OGR_G_CreateGeometryFromJson` it means that the JSON string include only the geometry object and not the full GeoJSON string. Nevertheless, you can easily change this code to be able to use a full GeoJSON string, simply by creating a function which will extract the geometry object from the GeoJSON string (using the `json-c` library for instance, which is also used by the OGR GeoJSON Driver).

Once you can access the input geometry object, you can use the `OGR_G_GetBoundary` function and store the result in the `res` geometry variable. Then, you only have to store the value in the right format : GeoJSON per default or GML as we declared it as a supported output format.

Please note that ZOO Kernel will give you pre-filled `outputs` values, so you will only have to fill the value for the key named `value`, even if in our example we override the `mimeType` using the `text/plain` value rather than the `application/json` (to show that we can also edit other fields of a map). Indeed, depending on the format requested by the client (or the default one) we will provide JSON or GML representation of the geometry.

```
tmp1=getMapFromMaps(outputs,"Result","mimeType");
if(strncmp(tmp1->value,"application/json",16)==0){
    addToMap(outputs->content,"value",OGR_G_ExportToJson(res));
    addToMap(outputs->content,"mimeType","text/plain");
}
else{
    addToMap(outputs->content,"value",OGR_G_ExportToGML(res));
}
```

The Boundary ZOO Service is now implemented and you need to compile it to produce a Shared Library. As you just used functions defined in `service.h` (`getMapFromMaps` and `addToMap`), you must include this file in your C code. The same requirement is needed to be able to use the `errorException` function declared in `zoo-kernel/service_internal.h`, you also must link your service object file to the `zoo-kernel/service_internal.o` in order to use `errorException` on runtime. You must then include the required files to access the `libxml2` and OGR C-API.

For the need of the Shared Library, you have to put your code in a block declared as `extern "C"`. The final Service code should be stored in a `service.c` file located in the root of the Services Provider directory (so in `/home/zoows/sources/zoo-services/ws_sp`). It should look like this :

```
#include "ogr_api.h"
#include "service.h"
extern "C" {
#include <libxml/tree.h>
#include <libxml/parser.h>
#include <libxml/xpath.h>
#include <libxml/xpathInternals.h>
<YOUR SERVICE CODE AND OTHER UTILITIES FUNCTIONS>
}
```

The full source code of your Service is now ready and you must produce the corresponding Service Shared Object by compiling the code as a Shared Library. This can be done using the following command:

```
g++ $CFLAGS -shared -fpic -o cgi-env/ServiceProvider.zo ./service.c $LDFLAGS
```

Please note that the `CLFAGS` and `LDFLAGS` environment variables values must be set before.

The `CFLAGS` must contain all the requested paths to find included headers, so the path to the directories where the `ogr_api.h`, `libxml2` directory, `service.h` and `service_internal.h` files are located. Thanks to the OSGeoLive environment, some of the provided tools can be used to retrieve those values : `xml2-config` and `gdal-config`, both used with the `--cflags` argument. They will produce the desired paths for you.

If you follow the instructions to create your ZOO Services Provider main directory in `zoo-services`, then you should find the ZOO Kernel headers and source tree which is located in the `../../zoo-kernel` directory relatively to your current path (`/home/user/zoows/sources/zoo-services/ws_sp`). Note that you can also use a full path to the `zoo-kernel` directory but using relative path will let you move your sources tree somewhere else and keep your code compiling using exactly the same command line. So you must add a `-I../../zoo-kernel` to your `CFLAGS` to make the compiler able to find the `service.h` and `service_internal.h` files.

The full `CFLAGS` definition should look like this :

```
CFLAGS=`gdal-config --cflags` `xml2-config --cflags` -I../../zoo-kernel/
```

Once you get the included paths correctly set in your `CFLAGS` , it is time to concentrate on the library we have to link against (defined in the `LDFLAGS` env variable). In order to link against the `gdal` and `libxml2` libraries, you can use the same tools than above using the `--libs` argument rather than `--cflags`. The full `LDFLAGS` definition must look like this :

```
LDFLAGS=`gdal-config --libs` `xml2-config --libs` ../../zoo-kernel/service_internal.o
```

Let's now create a `Makefile` which will help you compiling your code over the time. Please write a short `Makefile` in the root of your ZOO Services Provider directory, containing the following lines :

```
ZOO_SRC_ROOT=../../zoo-kernel/
CFLAGS=-I${ZOO_SRC_ROOT} `xml2-config --cflags` `gdal-config --cflags`
LDFLAGS=`xml2-config --libs` `gdal-config --libs` ${ZOO_SRC_ROOT}/service_internal.o

cgi-env/ogr_ws_service_provider.zo: service.c
    g++ ${CFLAGS} -shared -fpic -o cgi-env/ogr_ws_service_provider.zo ./service.c $
    {LDFLAGS}

clean:
    rm -f cgi-env/ogr_ws_service_provider.zo
```

Using this `Makefile`, you should be able to run `make` from your ZOO Service Provider main directory and to get the resulting `ogr_ws_service_provider.zo` file located in the `cgi-env` directory.

The metadata file and the ZOO Service Shared Object are now both located in the `cgi-env` directory. In order to deploy your new Service Provider, you only have to copy the ZOO Service Shared Object and its corresponding metadata file in the directory where ZOO Kernel is located, so in `/usr/lib/cgi-bin`. You must use a `sudo` command to achieve this task:

```
sudo cp ./cgi-env/* /usr/lib/cgi-bin
```

You should now understand more clearly the meanings of the ZOO Service Provider source tree ! The `cgi-env` directory will let you deploy your new Services or Services Provider in an easy way , simply by copying the whole `cgi-env` content in your `cgi-bin` directory.

Please note that you can add the following lines to your Makefile to be able to type `make install` directly and to get your new Services Provider available for use from ZOO Kernel :

```
install:
    sudo cp ./cgi-env/* /usr/lib/cgi-bin
```

Your ZOO Services Provider is now ready to use from an `Execute` request passed to ZOO Kernel.

3.3.1.2 Python Version

For those using Python to implement their ZOO Services Provider, the full code to copy in `ogr_ws_service_provider.py` in `cgi-env` directory is shown below. Indeed, as Python is an interpreted language, you do not have to compile anything before deploying your service which makes the deployment step much easier :

```
import osgeo.ogr
import libxml2

def createGeometryFromWFS(my_wfs_response):
    doc=libxml2.parseMemory(my_wfs_response,len(my_wfs_response))
    ctxt = doc.xpathNewContext()
    res=ctxt.xpathEval ("/*/*/*/*/*[local-name()='Polygon' or local-
name()='MultiPolygon']")
    for node in res:
        geometry_as_string=node.serialize()
        geometry=osgeo.ogr.CreateGeometryFromGML(geometry_as_string)
    return geometry

def Boundary(conf,inputs,outputs):
    if inputs["InputPolygon"]["mimeType"]=="application/json":
        geometry=osgeo.ogr.CreateGeometryFromJson(inputs["InputPolygon"]["value"])
    else:
        geometry=createGeometryFromWFS(inputs["InputPolygon"]["value"])
    rgeom=geometry.GetBoundary()
    if outputs["Result"]["mimeType"]=="application/json":
        outputs["Result"]["value"]=rgeom.ExportToJson()
        outputs["Result"]["mimeType"]="text/plain"
    else:
        outputs["Result"]["value"]=rgeom.ExportToGML()
    geometry.Destroy()
    rgeom.Destroy()
    return 3
```

We do not discuss the functions body here as we already gave all the details before and the code was voluntarily made in a similar way.

As done before, you only have to copy the `cgi-env` files into your `cgi-bin` directory :

```
sudo cp ./cgi-env/* /usr/lib/cgi-bin
```

A simple Makefile containing the install section can be written as the following :

```
install:
    sudo cp ./cgi-env/* /usr/lib/cgi-bin/
```

Finally, simply run `make install` from the ZOO Services Provider main directory, in order to deploy your ZOO Service Provider.

3.3.1.3 Testing the Service using Execute Request

3.3.1.3.1 The simple and unreadable way

Everybody should now get his own copy of the OGR Boundary Service stored as a ZOO Services Provider called `ogr_ws_service_provider` and deployed in the ZOO Kernel tree, so the following `Execute` request can be used to test the Service :

```
http://localhost/cgi-bin/zoo_loader.cgi?
request=Execute&service=WPS&version=1.0.0&Identifier=Boundary&DataInputs=InputPolygon=Reference@xlink:href=
http%3A%2F%2Flocalhost%3A8082%2Fgeoserver%2Fows%3FSERVICE%3DWFS%26REQUEST%3DGetFeature
%26VERSION%3D1.0.0%26typename%3Dtopp%3Astates%26SRS%3DEPSG%3A4326%26FeatureID%3Dstates.15
```

As you can see in the url above, we use an URLEncoded WFS request to the Geoserver WFS server available on OSGeoLive as a `xlink:href` key in the `DataInputs` KVP value, and set the `InputPolygon` value to `Reference`. The corresponding non encoded WFS request is as follow :

```
http://localhost:8082/geoserver/ows/?
SERVICE=WFS&REQUEST=GetFeature&VERSION=1.0.0&typename=topp:states&SRS=EPSG:4326&FeatureID=state
s.15
```

Please note that you can add `lineage=true` to the previous request if you need to get information about the input values used to run your Service. Furthermore, you may need to store the `ExecuteResponse` document of your ZOO Service to re-use it later. In this case you must add `storeExecuteResponse=true` to the previous request. Note that is an important thing as the behavior of ZOO Kernel is not exactly the same than when running without this parameter settled to true. Indeed, in such a request, ZOO Kernel will give you an `ExecuteResponse` document which will contain the attribute `statusLocation`, which inform the client where the ongoing status or the final `ExecuteResponse` will be located.

Here is an example of what the `ExecuteResponse` using `storeExecuteResponse=true` in the request would look like :

```
-<wps:ExecuteResponse xsi:schemaLocation="http://www.opengis.net/wps/1.0.0 http://schemas.opengis.net/wps/1.0.0/wpsExecute_response.xsd"
service="WPS" version="1.0.0" xml:lang="en" serviceInstance="http://localhost/zoo/" statusLocation="http://localhost/zoo/..
/temp/ogr_service.zo_21487.xml">
-<wps:Process wps:processVersion="1">
  <ows:Identifier>Boundary</ows:Identifier>
  <ows:Title>Compute boundary.</ows:Title>
-<ows:Abstract>
  A new geometry object is created and returned containing the boundary of the geometry on which the method is invoked.
</ows:Abstract>
</wps:Process>
-<wps:Status creationTime="2010-09-02T09:36:16Z">
  <wps:ProcessAccepted/>
</wps:Status>
</wps:ExecuteResponse>
```

Then, according to the `statusLocation`, you should get the `ExecuteResponse` as you get before using the previous request. Note that can be really useful to provide some caching system for a client application.

You didn't specify any `ResponseForm` in the previous request, it is not requested and should return a `ResponseDocument` per default using the `application/json` mimeType as you defined in your `zcfg` file. Nevertheless, you can tell ZOO Kernel what kind of data you want to get in result of your query adding the attribute `mimeType=text/xml` to your `ResponseDocument` parameter. Adding this parameter to the previous request will give us the result as its GML representation :

```
http://localhost/cgi-bin/zoo_loader.cgi?
request=Execute&service=WPS&version=1.0.0&Identifier=Boundary&DataInputs=InputPolygon=Reference@xlink:href=
http%3A%2F%2Flocalhost%3A8082%2Fgeoserver%2Fows%3FSERVICE%3DWFS%26REQUEST%3DGetFeature
%26VERSION%3D1.0.0%26typename%3Dtopp%3Astates%26SRS%3DEPSG%3A4326%26FeatureID
%3Dstates.15&ResponseDocument=Result@mimeType=text/xml
```

As defined by the WPS specifications, you can also ask for a `RawDataOutput` to get only the data without the full `ResponseDocument`. To do that, you only have to replace the `ResponseDocument` of your request by `RawDataOutput`, like in the following request :

```
http://localhost/cgi-bin/zoo_loader.cgi?
request=Execute&service=WPS&version=1.0.0&Identifier=Boundary&DataInputs=InputPolygon=Reference@xlink:href=
http%3A%2F%2Flocalhost%3A8082%2Fgeoserver%2Fows%3FSERVICE%3DWFS%26REQUEST%3DGetFeature
%26VERSION%3D1.0.0%26typename%3Dtopp%3Astates%26SRS%3DEPSG%3A4326%26FeatureID
%3Dstates.15&RawDataOutput=Result@mimeType=application/json
```

Finally, please note that we go back to the default `mimeType` to directly obtain the JSON string as we will use this kind of request to develop our client application in the next section of this workshop.

3.3.1.3.2 Simplification and readability of request

As you can see in the simple example we used since the beginning of this workshop, it is sometimes hard to write the `Execute` requests using the GET method as it makes really long and complex URLs. In the next requests examples, we will thus use the POST XML requests. First, here is the XML request corresponding to the previous `Execute` we used :

```
<wps:Execute service="WPS" version="1.0.0" xmlns:wps="http://www.opengis.net/wps/1.0.0"
xmlns:ows="http://www.opengis.net/ows/1.1" xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.opengis.net/wps/1.0.0
../wpsExecute_request.xsd">
  <ows:Identifier>Boundary</ows:Identifier>
  <wps:DataInputs>
    <wps:Input>
      <ows:Identifier>InputPolygon</ows:Identifier>
      <ows:Title>Playground area</ows:Title>
      <wps:Reference xlink:href="http://localhost:8082/geoserver/ows/?
SERVICE=WFS&REQUEST=GetFeature&VERSION=1.0.0&typename=topp:states&SRS=EPSG:43
26&FeatureID=states.15"/>
    </wps:Input>
  </wps:DataInputs>
  <wps:ResponseForm>
    <wps:ResponseDocument>
      <wps:Output>
        <ows:Identifier>Result</ows:Identifier>
        <ows:Title>Area serviced by playground.</ows:Title>
        <ows:Abstract>Area within which most users of this playground will live.</ows:Abstract>
      </wps:Output>
    </wps:ResponseDocument>
  </wps:ResponseForm>
</wps:Execute>
```

In order to let you easily run the XML requests, a simple HTML form called `test_services.html` is available in your `/var/www` directory. You can access it using the following link : http://localhost/test_services.html.

Please open this page in your browser, simply fill the XML request content into the textarea field and click the « run using XML Request » submit button. You will get exactly the same result as when running your Service using the GET request. The screenshot above show the HTML form including the request and the ExecuteResponse document displayed in the iframe at the bottom of the page :

Use your own XML request :

run using XML Request

```
<wps:Execute service="WPS" version="1.0.0" xmlns:wps="http://www.opengis.net/wps/1.0.0"
xmlns:ows="http://www.opengis.net/ows/1.1" xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.opengis.net
/wps/1.0.0 ../wpsExecute_request.xsd">
  <ows:Identifier>Boundary</ows:Identifier>
  <wps:DataInputs>
    <wps:Input>
      <ows:Identifier>InputPolygon</ows:Identifier>
      <ows:Data>
        <ows:ComplexData mimeType="application/json">
          { "type": "MultiPolygon", "coordinates": [ [ [ [ -105.998360, 31.393818 ], [ -106.212753,
31.478128 ], [ -106.383041, 31.733763 ], [ -106.538971, 31.786198 ], [ -106.614441,
31.817728 ], [ -105.769730, 31.170780 ], [ -105.998360, 31.393818 ] ] ], [ [ [
-94.913429, 29.257572 ], [ -94.767380, 29.342451 ], [ -94.748405, 29.319490 ], [
-95.105415, 29.096958 ], [ -94.913429, 29.257572 ] ] ] ] }
        </ows:ComplexData>
      </ows:Data>
    </wps:Input>
  </wps:DataInputs>
  <wps:ResponseForm>
    <wps:ResponseDocument>
      <wps:Output>
        <ows:Identifier>Result</ows:Identifier>
        <ows:Title>Area serviced by playground.</ows:Title>
        <ows:Abstract>Area within which most users of this playground will
live.</ows:Abstract>
      </wps:Output>
    </wps:ResponseDocument>
  </wps:ResponseForm>
</wps:Execute>
```

```
-<wps:ExecuteResponse xsi:schemaLocation="http://www.opengis.net/wps/1.0.0 http://schemas.opengis.net/wps/1.0.0
/wpsExecute_response.xsd" service="WPS" version="1.0.0" xml:lang="en" serviceInstance="http://localhost/zoo/">
-<wps:Process wps:processVersion="1">
  <ows:Identifier>Boundary</ows:Identifier>
  <ows:Title>Compute boundary.</ows:Title>
-<ows:Abstract>
  A new geometry object is created and returned containing the boundary of the geometry on which the method is invoked.
</ows:Abstract>
```

The `xlink:href` value is used in the simplest way to deal with such data input. Obviously, you can also use a full JSON string of the geometry, as shown in the following XML Request example :

```
<wps:Execute service="WPS" version="1.0.0" xmlns:wps="http://www.opengis.net/wps/1.0.0"
xmlns:ows="http://www.opengis.net/ows/1.1" xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.opengis.net/wps/1.0.0 ../wpsExecute_request.xsd">
  <ows:Identifier>Boundary</ows:Identifier>
  <wps:DataInputs>
    <wps:Input>
      <ows:Identifier>InputPolygon</ows:Identifier>
      <wps:Data>
        <ows:ComplexData mimeType="application/json">
          { "type": "MultiPolygon", "coordinates": [ [ [ [ -105.998360, 31.393818 ], [ -106.212753,
31.478128 ], [ -106.383041, 31.733763 ], [ -106.538971, 31.786198 ], [ -106.614441,
31.817728 ], [ -105.769730, 31.170780 ], [ -105.998360, 31.393818 ] ] ], [ [ [
-94.913429, 29.257572 ], [ -94.767380, 29.342451 ], [ -94.748405, 29.319490 ], [
-95.105415, 29.096958 ], [ -94.913429, 29.257572 ] ] ] ] }
        </ows:ComplexData>
      </wps:Data>
    </wps:Input>
  </wps:DataInputs>
  <wps:ResponseForm>
    <wps:ResponseDocument>
      <wps:Output>
        <ows:Identifier>Result</ows:Identifier>
        <ows:Title>Area serviced by playground.</ows:Title>
        <ows:Abstract>Area within which most users of this
playground will live.</ows:Abstract>
      </wps:Output>
    </wps:ResponseDocument>
  </wps:ResponseForm>
</wps:Execute>
```

If everything went well, you should get the Boundary of the JSON geometry passed as argument, and so be sure that your Service support both GML and JSON as input data. Note that in the previous request, we added a mimeType attribute to the ComplexData node to specify that the input data is not in the default text/xml mimeType but passed as an application/json string directly. It is similar to add @mimeType=application/json as we discussed before.

3.3.2 Creating Services for other functions (ConvexHull and Centroid)

As the Boundary sample service code is available, you can now easily add ConvexHull and Centroid functions as they take exactly the same number of arguments : Only one geometry. The details for implementing and deploying the ConvexHull Service are provided bellow, and we will let you do the same thing for the Centroid one.

3.3.2.1 C Version

Please add first the following code to the service.c source code :

```
int ConvexHull (maps*& conf,maps*& inputs,maps*& outputs) {
    OGRGeometryH geometry, res;
    map* tmp=getMapFromMaps (inputs, "InputPolygon", "value");
    if (tmp==NULL)
        return SERVICE_FAILED;
    map* tmp1=getMapFromMaps (inputs, "InputPolygon", "mimeType");
    if (strcmp (tmp1->value, "application/json", 16)==0)
        geometry=OGR_G_CreateGeometryFromJson (tmp->value);
    else
        geometry=createGeometryFromWFS (conf, tmp->value);
    res=OGR_G_ConvexHull (geometry);
    tmp1=getMapFromMaps (outputs, "Result", "mimeType");
    if (strcmp (tmp1->value, "application/json", 16)==0) {
        addToMap (outputs->content, "value", OGR_G_ExportToJson (res));
        addToMap (outputs->content, "mimeType", "text/plain");
    }
    else{
        addToMap (outputs->content, "value", OGR_G_ExportToGML (res));
    }
    outputs->next=NULL;
    OGR_G_DestroyGeometry (geometry);
    OGR_G_DestroyGeometry (res);
    return SERVICE_SUCCEEDED;
}
```

This new code is exactly the same as for the Boundary Service. The only thing we modified is the line where the `OGR_G_ConvexHull` function is called (rather than the `OGR_G_GetBoundary` you used before). It is better to not copy and paste the whole function and find a more generic way to define your new Services as the function body will be the same in every case. The following generic function is proposed to make things simpler :

```
int applyOne (maps*& conf,maps*& inputs,maps*& outputs, OGRGeometryH (*myFunc)
(OGRGeometryH)) {
    OGRGeometryH geometry, res;
    map* tmp=getMapFromMaps (inputs, "InputPolygon", "value");
    if (tmp==NULL)
        return SERVICE_FAILED;
    map* tmp1=getMapFromMaps (inputs, "InputPolygon", "mimeType");
    if (strcmp (tmp1->value, "application/json", 16)==0)
```

```

    geometry=OGR_G_CreateGeometryFromJson(tmp->value);
else
    geometry=createGeometryFromWFS(conf,tmp->value);
res>(*myFunc)(geometry);
tmp1=getMapFromMaps(outputs,"Result","mimeType");
if(strncmp(tmp1->value,"application/json",16)==0){
    addToMap(outputs->content,"value",OGR_G_ExportToJson(res));
    addToMap(outputs->content,"mimeType","text/plain");
}
else{
    addToMap(outputs->content,"value",OGR_G_ExportToGML(res));
}
outputs->next=NULL;
OGR_G_DestroyGeometry(geometry);
OGR_G_DestroyGeometry(res);
return SERVICE_SUCCEEDED;
}

```

Then, a function pointer called `myFunc` rather than the full function name can be used. This way we can re-implement our Boundary Service this way :

```

int Boundary(maps*& conf,maps*& inputs,maps*& outputs){
    return applyOne(conf,inputs,outputs,&OGR_G_GetBoundary);
}

```

Using this `applyOne` local function defined in the `service.c` source code, we can define other Services this way :

```

int ConvexHull(maps*& conf,maps*& inputs,maps*& outputs){
    return applyOne(conf,inputs,outputs,&OGR_G_ConvexHull);
}
int Centroid(maps*& conf,maps*& inputs,maps*& outputs){
    return applyOne(conf,inputs,outputs,&MY_OGR_G_Centroid);
}

```

The genericity of the `applyOne` function let you add two new Services in your ZOO Services Provider : `ConvexHull` and `Centroid`.

Note that you should define `MY_OGR_G_Centroid` function before the `Centroid` one as `OGR_G_Centroid` don't return a geometry object but set the value to an already existing one and support only Polygon as input, so to ensure we use the `ConvexHull` for MultiPolygon. So please use the code bellow :

```

OGRGeometryH MY_OGR_G_Centroid(OGRGeometryH hTarget){
    OGRGeometryH res;
    res=OGR_G_CreateGeometryFromJson("{\"type\": \"Point\", \"coordinates\": [0,0] }");
    OGRwkbGeometryType gtype=OGR_G_GetGeometryType(hTarget);
    if(gtype!=wkbPolygon){
        hTarget=OGR_G_ConvexHull(hTarget);
    }
    OGR_G_Centroid(hTarget,res);
    return res;
}

```

To deploy your Services, you only have to copy the `Boundary.zcfg` metadata file from your `cgi-env` directory as `ConvexHull.zcfg` and `Centroid.zcfg`. Then, you must rename the Service name on the first line to be able to run and test the `Execute` request in the same way you did before. You only have to set the `Identifier` value to `ConvexHull` or `Centroid` in your request depending on the Service you want to run.

Note here that the `GetCapabilities` and `DescribeProcess` requests will return odd results as we didn't modified any metadata informations, you can edit the `.zcfg` files to set correct values. By the way it can be used for testing purpose, as the input and output get the same name and default/supported formats.

3.3.2.2 Python Version

As we did for the C version, let us implement the `ConvexHull` Service first. Here is the code for such a service :

```
def ConvexHull (conf, inputs, outputs) :
    if inputs["InputPolygon"]["mimeType"]=="application/json":
        geometry=osgeo.ogr.CreateGeometryFromJson (inputs["InputPolygon"]["value"])
    else:
        geometry=createGeometryFromWFS (inputs["InputPolygon"]["value"])
    rgeom=geometry.ConvexHull ()
    if outputs["Result"]["mimeType"]=="application/json":
        outputs["Result"]["value"]=rgeom.ExportToJson ()
        outputs["Result"]["mimeType"]="text/plain"
    else:
        outputs["Result"]["value"]=rgeom.ExportToGML ()
    geometry.Destroy ()
    rgeom.Destroy ()
    return 3
```

Once again, you can easily copy and paste the function for `Boundary` and simply modify the line where the `Geometry` method was called. Nevertheless, as we did for the C language we will give you a simple way to get things more generic.

First of all, the first step which consists in extracting the `InputPolygon` `Geometry` as it will be used in the same way in each `Service` functions, so we will first create a function which will do that for us. The same thing can also be done for filling the output value, so we will define another function to do that automatically. Here is the code of this two functions (`extractInputs` and `outputResult`):

```
def extractInputs (obj) :
    if obj["mimeType"]=="application/json":
        return osgeo.ogr.CreateGeometryFromJson (obj["value"])
    else:
        return createGeometryFromWFS (obj["value"])
    return null

def outputResult (obj, geom) :
    if obj["mimeType"]=="application/json":
        obj["value"]=geom.ExportToJson ()
        obj["mimeType"]="text/plain"
    else:
        obj["value"]=geom.ExportToGML ()
```

We can so minimize the code of the `Boundary` function to make it simpler using the following function definition :

```
def Boundary (conf, inputs, outputs) :
    geometry=extractInputs (inputs["InputPolygon"])
    rgeom=geometry.GetBoundary ()
    outputResult (outputs["Result"], rgeom)
    geometry.Destroy ()
    rgeom.Destroy ()
    return 3
```

Then definition of the ConvexHull and Centroid Services can be achieved using the following code :

```
def ConvexHull (conf, inputs, outputs) :
    geometry=extractInputs (inputs ["InputPolygon"])
    rgeom=geometry.ConvexHull ()
    outputResult (outputs ["Result"], rgeom)
    geometry.Destroy ()
    rgeom.Destroy ()
    return 3

def Centroid (conf, inputs, outputs) :
    geometry=extractInputs (inputs ["InputPolygon"])
    if geometry.GetGeometryType () !=3:
        geometry=geometry.ConvexHull ()
    rgeom=geometry.Centroid ()
    outputResult (outputs ["Result"], rgeom)
    geometry.Destroy ()
    rgeom.Destroy ()
    return 3
```

Note, that in Python you also need to use ConvexHull to deal with MultiPolygons.

You must now copy the Boundary.zcfg file as we explained for the C version in ConvexHull.zcfg and Centroid.zcfg respectively and then, use make install command to re-deploy and test your Services Provider.

3.3.3 Create the Buffer Service

We can now work on the Buffer Service, which takes more arguments than the other ones. Indeed, the code is a bit different from the one used to implement the Boundary, ConvexHull and Centroid Services.

The Buffer service also takes an input geometry, but uses a BufferDistance parameter. It will also allow you to define LitteralData block as the BufferDistance will be simple integer value. The read access to such kind of input value is made using the same function as used before.

3.3.3.1 C Version

If you go back to the first Boundary Service source code, you should not find the following very complicated. Indeed, you simply have to add the access of the BufferDistance argument and modify the line when the `OGR_G_Buffer` must be called (instead of `OGR_G_GetBoundary`). Here is the full code :

```
int Buffer (maps*& conf, maps*& inputs, maps*& outputs) {
    OGRGeometryH geometry, res;
    map* tmp1=getMapFromMaps (inputs, "InputPolygon", "value");
    if (tmp1==NULL)
        return SERVICE_FAILED;
    map* tmp1=getMapFromMaps (inputs, "InputPolygon", "mimeType");
    if (strncmp (tmp1->value, "application/json", 16)==0)
        geometry=OGR_G_CreateGeometryFromJson (tmp1->value);
    else
        geometry=createGeometryFromWFS (conf, tmp1->value);
    int bufferDistance=1;
    tmp=getMapFromMaps (inputs, "BufferDistance", "value");
    if (tmp!=NULL)
        bufferDistance=atoi (tmp->value);
    res=OGR_G_Buffer (geometry, bufferDistance, 30);
    tmp1=getMapFromMaps (outputs, "Result", "mimeType");
    if (strncmp (tmp1->value, "application/json", 16)==0) {
        addToMap (outputs->content, "value", OGR_G_ExportToJson (res));
        addToMap (outputs->content, "mimeType", "text/plain");
    }
    else {
        addToMap (outputs->content, "value", OGR_G_ExportToGML (res));
    }
    outputs->next=NULL;
    OGR_G_DestroyGeometry (geometry);
    OGR_G_DestroyGeometry (res);
    return SERVICE_SUCCEEDED;
}
```

The new code must be inserted in your `service.c` file and need to be recompiled and replace the older version of your ZOO Service Provider in the `/usr/lib/cgi-bin/` directory. You must of course place the corresponding ZOO Metadata File in the same directory.

As we explained before, ZOO Kernel is permissive in the sense that you can pass more arguments than defined in you `zcfg` file, so let's try using a copy of the `Boundary.zcfg` file renamed as `Buffer.zcfg` and containing the `Buffer` identifier. Then, please test your service using an `Execute` request as you did before. You will obtain the buffer result in a `ResponseDocument`.

You may have noted that the above code check if a BufferDistance input was passed to the service. If not, we will use 1 as the default value, which explains why you do not have to use one more input to your previous queries.

You can change the BufferDistance value used by your Service to compute Buffer of your geometry by adding it to the DataInputs value in your request. Note that using KVP syntaxe, each DataInputs are separated by a semicolon.

So, the previous request :

```
DataInputs=InputPolygon=Reference@xlink:href=http%3A%2F%2Flocalhost%3A8082%2Fgeoserver%2Fows%3FSERVICE%3DWFS%26REQUEST%3DGetFeature%26VERSION%3D1.0.0%26typename%3Dtopp%3Astates%26SRS%3DEPSG%3A4326%26FeatureID%3Dstates.15
```

Can now be rewritten this way :

```
DataInputs=InputPolygon=Reference@xlink:href=http%3A%2F%2Flocalhost%3A8082%2Fgeoserver%2Fows%3FSERVICE%3DWFS%26REQUEST%3DGetFeature%26VERSION%3D1.0.0%26typename%3Dtopp%3Astates%26SRS%3DEPSG%3A4326%26FeatureID%3Dstates.15;BufferDistance=2
```

Setting BufferDistance value to 2 would give you a different result, then don't pass any other parameter as we defined 1 as the default value in the source code.

Here you can find the same query in XML format to use from the http://localhost/test_services.html HTML form :

```
<wps:Execute service="WPS" version="1.0.0" xmlns:wps="http://www.opengis.net/wps/1.0.0"
xmlns:ows="http://www.opengis.net/ows/1.1" xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.opengis.net/wps/1.0.0 ../wpsExecute_request.xsd">
  <ows:Identifier>Buffer</ows:Identifier>
  <wps>DataInputs>
    <wps:Input>
      <ows:Identifier>InputPolygon</ows:Identifier>
      <ows:Title>Playground area</ows:Title>
      <wps:Reference xlink:href="http://localhost:8082/geoserver/ows/?
SERVICE=WFS&REQUEST=GetFeature&VERSION=1.0.0&typename=topp:states&SRS=EPS
G:4326&FeatureID=states.15"/>
    </wps:Input>
    <wps:Input>
      <ows:Identifier>BufferDistance</ows:Identifier>
      <wps>Data>
        <wps:LiteralData uom="degree">2</wps:LiteralData>
      </wps>Data>
    </wps:Input>
  </wps>DataInputs>
  <wps:ResponseForm>
    <wps:ResponseDocument>
      <wps:Output>
        <ows:Identifier>Buffer</ows:Identifier>
        <ows:Title>Area serviced by playground.</ows:Title>
        <ows:Abstract>Area within which most users of this
playground will live.</ows:Abstract>
      </wps:Output>
    </wps:ResponseDocument>
  </wps:ResponseForm>
</wps:Execute>
```

3.3.3.2 Python Version

As we already defined the utility functions `createGeometryFromWFS` and `outputResult`, the code is as simple as this :

```
def Buffer (conf, inputs, outputs) :
    geometry=extractInputs (inputs["InputPolygon"])
    try:
        bdist=int (inputs ["BufferDistance"] ["value"])
    except:
        bdist=10
    rgeom=geometry.Buffer (bdist)
    outputResult (outputs ["Result"], rgeom)
    geometry.Destroy ()
    rgeom.Destroy ()
    return 3
```

We simply added the use of `inputs["BufferDistance"]["value"]` as arguments of the Geometry instance `Buffer` method. Once you get this code added to your `ogr_ws_service_provider.py` file, simply copy it in the ZOO Kernel directory (or type `make install` from your ZOO Service Provider root directory). Note that you also need the `Buffer.zcfg` file detailed in the next section.

3.3.3.3 The Buffer MetadataFile file

You must add `BufferDistance` to the Service Metadata File to let clients know that this Service supports this parameter. To do this, please copy your original `Boundary.zcfg` file as `Buffer.zcfg` and add the following lines to the `DataInputs` block :

```
[BufferDistance]
Title = Buffer Distance
Abstract = Distance to be used to calculate buffer.
minOccurs = 0
maxOccurs = 1
<LiteralData>
  DataType = float
  <Default>
    uom = degree
    value = 10
  </Default>
  <Supported>
    uom = meter
  </Supported>
</LiteralData>
```

Note that as `minOccurs` is set to 0 which means that the input parameter is optional and don't have to be passed. You must know that ZOO Kernel will pass the default value to the Service function for an optional parameter with a default value set.

You can get a full copy of the `Buffer.zcfg` file here :

<http://zoo-project.org/trac/browser/trunk/zoo-services/ogr/base-vect-ops/cgi-env/Buffer.zcfg>

You can now ask ZOO Kernel for `GetCapabilities`, `DescribeProcess` and `Execute` for the Buffer Service.

4 Building a WPS client using OpenLayers

The next step of our workshop is to connect to the OGR Services we have created from an OpenLayers map. This will allow to apply single or multiple geometries processes on user-selected polygons and to display the new generated geometries.

4.1 Creating a simple map showing the dataset as WMS

OpenLayers is also included in OSGeoLive default distribution, so it is convenient to use it for our needs. Please open your text editor and write the following HTML snippet:

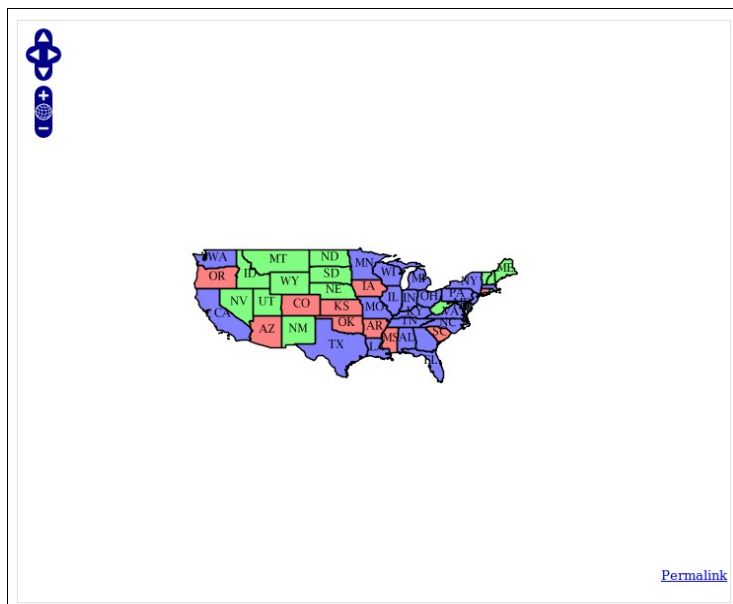
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"><html
xmlns="http://www.w3.org/1999/xhtml" xml:lang="EN" lang="EN">
<head>
  <meta content="text/html; charset=UTF-8" http-equiv="content-type"/>
  <title>ZOO WPS Client example</title>
  <style>
    #map{width:600px;height:600px;}
  </style>
  <link rel="stylesheet" href="/openlayers/theme/default/style.css" type="text/css" />
  <script type="text/javascript" src="/openlayers/lib/OpenLayers.js"></script>
</head>
<body onload="init()">
  <div id="map"></div>
</body>
</html>
```

The following JavaScript code must then be added in a `<script></script>` section within the `<head>` one. This will setup a map showing the United States data as WMS.

```
var map, layer, select, hover, multi, control;

function init(){
  OpenLayers.ProxyHost= "../cgi-bin/proxy.cgi?url=";
  map = new OpenLayers.Map('map', {
    controls: [
      new OpenLayers.Control.PanZoom(),
      new OpenLayers.Control.Permalink(),
      new OpenLayers.Control.Navigation()
    ]
  });
  layer = new OpenLayers.Layer.WMS (
    "States WMS/WFS",
    "http://localhost:8082/geoserver/ows",
    {layers: 'topp:states', format: 'image/png'},
    {buffer:1, singleTile:true}
  );
  map.addLayers([layer]);
  map.zoomToExtent(new OpenLayers.Bounds(-140.444336,25.115234,-44.438477,50.580078));
}
```

Once done, please save your HTML file as `zoo-ogr.html` in your workshop directory, then copy it in `/var/www` and visualize it with your favorite Web browser using this link : <http://localhost/zoo-ogr.html>. You should obtain a map centered on the USA with the WMS layer activated.



4.2 Fetching the data layer as WFS and adding selection controls

Before accessing the displayed data via WFS, you first have to create new vector layers dedicated to host the several interactions we are going to create. Please add the following lines within the `init()` function, and do not forget to add the newly created layer in the `map.addLayers` method:

```
select = new OpenLayers.Layer.Vector("Selection", {styleMap:
    new OpenLayers.Style(OpenLayers.Feature.Vector.style["select"])
});
hover = new OpenLayers.Layer.Vector("Hover");

multi = new OpenLayers.Layer.Vector("Multi", {styleMap:
    new OpenLayers.Style({
        fillColor:"red",
        fillOpacity:0.4,
        strokeColor:"red",
        strokeOpacity:1,
        strokeWidth:2
    })
});

map.addLayers([layer, select, hover, multi]);
```

Then, you can now access `tdata` by creating new controls to select polygons, as the following. Please note that `OpenLayers.Protocol.WFS.fromWMSLayer(layer)` is used to access geometries and that several state of selection are declared and append to the `control` variable.

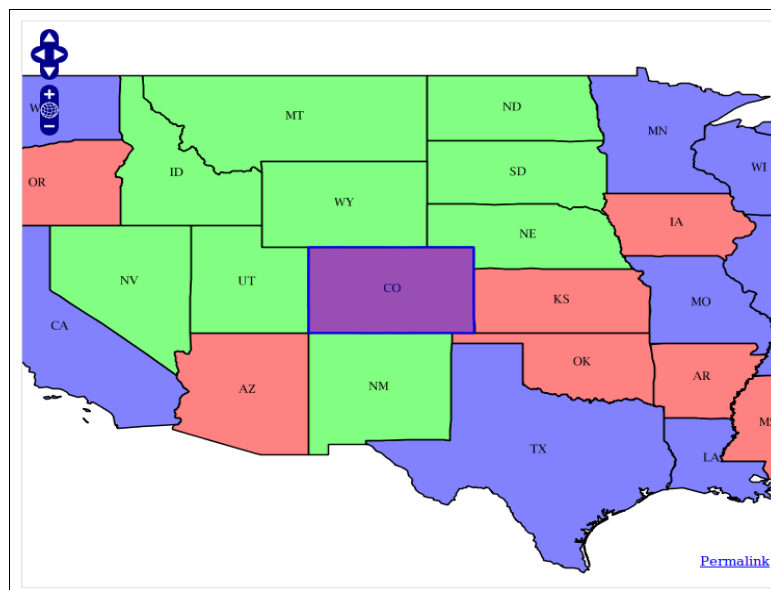
```
control = new OpenLayers.Control.GetFeature({
    protocol: OpenLayers.Protocol.WFS.fromWMSLayer(layer)
});
```

```

control.events.register("featureselected", this, function(e) {
    select.addFeatures([e.feature]);
});
control.events.register("featureunselected", this, function(e) {
    select.removeFeatures([e.feature]);
});
control.events.register("hoverfeature", this, function(e) {
    hover.addFeatures([e.feature]);
});
control.events.register("outfeature", this, function(e) {
    hover.removeFeatures([e.feature]);
});
map.addControl(control);
control.activate();

```

Please save your HTML file again. You should now be able to select a polygon only by clicking on it. The selected polygon should appear in blue color.



4.3 Calling the single geometrie processes from JavaScript

Now that everything is setup, we can go on and call our OGR ZOO services with JavaScript. Please add the following lines after the `init()` function, which will call the single geometry processes.

```

function simpleProcessing(aProcess) {
    if (select.features.length == 0)
        return alert("No feature selected!");
    var url = '/zoo/?request=Execute&service=WPS&version=1.0.0&';
    if (aProcess == 'Buffer') {
        var dist = document.getElementById('bufferDist').value;
        if (isNaN(dist))
            return alert("Distance is not a Number!");
        url+='Identifier=Buffer&DataInputs=BufferDistance='+dist+'@datatype=interger;InputPolygon=Reference@xlink:href=';
    } else
        url += 'Identifier='+aProcess+'&DataInputs=InputPolygon=Reference@xlink:href=';
    var xlink = control.protocol.url + "?SERVICE=WFS&REQUEST=GetFeature&VERSION=1.0.0";
    xlink += '&typename='+control.protocol.featurePrefix;
    xlink += ':'+control.protocol.featureType;
    xlink += '&SRS='+control.protocol.srsName;
    xlink += '&FeatureID='+select.features[0].fid;
}

```

```

url += encodeURIComponent(xlink);
url += '&RawDataOutput=Result@mimeType=application/json';
var request = new OpenLayers.Request.XMLHttpRequest();
request.open('GET',url,true);
request.onreadystatechange = function() {
  if(request.readyState == OpenLayers.Request.XMLHttpRequest.DONE) {
    var GeoJSON = new OpenLayers.Format.GeoJSON();
    var features = GeoJSON.read(request.responseText);
    hover.removeFeatures(hover.features);
    hover.addFeatures(features);
  }
}
request.send();
}

```

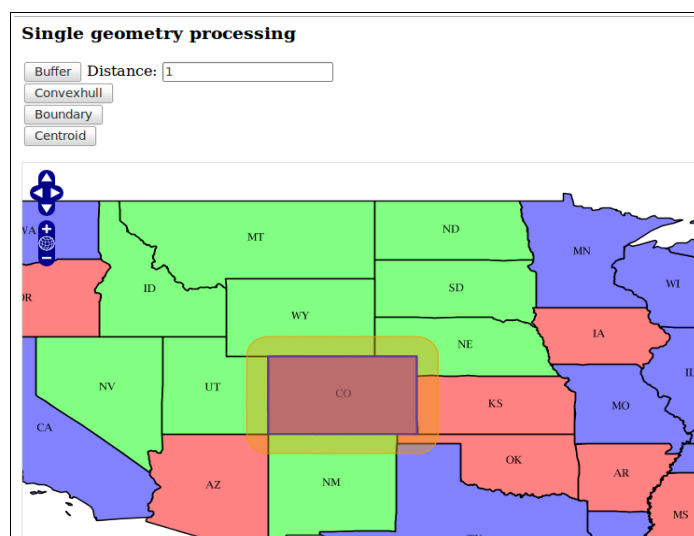
Then, some specific buttons must be added in the HTML, in order to be able to call the different processes we just declared. You can add them on top of the map by writing the following lines before the `<div id="map"></div>`.

```

<h3>Single geometry processing</h3>
<ul>
  <li>
    <input type="button" onclick="simpleProcessing(this.value);" value="Buffer" />
    <input id="bufferDist" value="1" />
  </li>
  <li>
    <input type="button" onclick="simpleProcessing(this.value);" value="ConvexHull" />
  </li>
  <li>
    <input type="button" onclick="simpleProcessing(this.value);" value="Boundary" />
  </li>
  <li>
    <input type="button" onclick="simpleProcessing(this.value);" value="Centroid" />
  </li>
</ul>

```

Save your HTML file again. You should now be able to select a polygon and to launch a Buffer, ConvexHull, Boundary or Centroid on it by clicking one of the button. The result of the process should appear as GeoJSON layer on the map, in orange color.



4.4 Calling the multiples geometries processes from JavaScript

Using the same technique, you can now write a function dedicated to the multiple geometries processes. Please add the following lines after the `simpleProcessing()` function, we will guide you during the exercise in section 5 on how to create such a function.

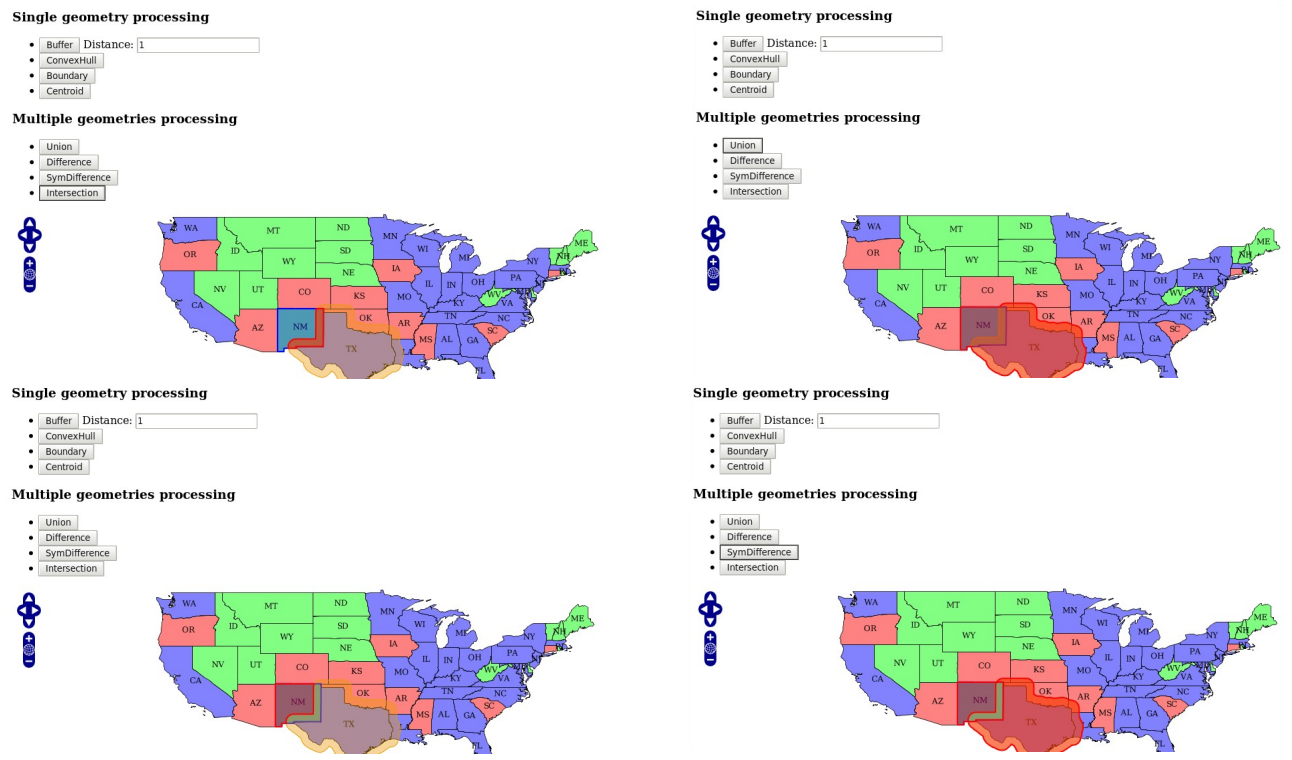
```
function multiProcessing(aProcess) {
  if (select.features.length == 0 || hover.features.length == 0)
    return alert("No feature created!");
  var url = '/zoo/';
  var xlink = control.protocol.url + "?SERVICE=WFS&REQUEST=GetFeature&VERSION=1.0.0";
  xlink += '&typename=' + control.protocol.featurePrefix;
  xlink += ':' + control.protocol.featureType;
  xlink += '&SRS=' + control.protocol.srsName;
  xlink += '&FeatureID=' + select.features[0].fid;
  var GeoJSON = new OpenLayers.Format.GeoJSON();
  try {
    var params = '<wps:Execute service="WPS" version="1.0.0"
xmlns:wps="http://www.opengis.net/wps/1.0.0" xmlns:ows="http://www.opengis.net/ows/1.1"
xmlns:xlink="http://www.w3.org/1999/xlink" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://www.opengis.net/wps/1.0.0/./wpsExecute_request.xsd">';
    params += ' <ows:Identifier>' + aProcess + '</ows:Identifier>';
    params += ' <wps>DataInputs>';
    params += ' <wps:Input>';
    params += ' <ows:Identifier>InputEntity1</ows:Identifier>';
    params += ' <wps:Reference xlink:href="' + xlink.replace(/&/gi, '&amp;') + '"/>';
    params += ' </wps:Input>';
    params += ' <wps:Input>';
    params += ' <ows:Identifier>InputEntity2</ows:Identifier>';
    params += ' <wps>Data>';
    params += ' <wps:ComplexData mimeType="application/json">
'+GeoJSON.write(hover.features[0].geometry)+' </wps:ComplexData>';
    params += ' </wps>Data>';
    params += ' </wps:Input>';
    params += ' </wps>DataInputs>';
    params += ' <wps:ResponseForm>';
    params += ' <wps:RawDataOutput>';
    params += ' <ows:Identifier>Result</ows:Identifier>';
    params += ' </wps:RawDataOutput>';
    params += ' </wps:ResponseForm>';
    params += ' </wps:Execute>';
  } catch (e) {
    alert(e);
    return false;
  }
  var request = new OpenLayers.Request.XMLHttpRequest();
  request.open('POST', url, true);
  request.setRequestHeader('Content-Type', 'text/xml');
  request.onreadystatechange = function() {
    if (request.readyState == OpenLayers.Request.XMLHttpRequest.DONE) {
      var GeoJSON = new OpenLayers.Format.GeoJSON();
      var features = GeoJSON.read(request.responseText);
      multi.removeFeatures(multi.features);
      multi.addFeatures(features);
    }
  }
  request.send(params);
}
```

Note that this time we didn't use the GET method to request the ZOO Kernel but a XML POST. We did that because if you use the GET method you will get error due to the HTTP GET method limitation based on the length of your request. Using JSON string representing the geometry will make your request longer.

Once you get the functions to call your multiple geometries processes, you' must add some buttons to fire the request call. Here is the HTML code to add to your current zoo-ogr.html file :

```
<h3>Multiple geometries processing</h3>
<ul>
  <li>
    <input type="button" onclick="multiProcessing(this.name);" value="Union"/>
  </li>
  <li>
    <input type="button" onclick="multiProcessing(this.name);" value="Difference"/>
  </li>
  <li>
    <input type="button" onclick="multiProcessing(this.value);" value="SymDifference"/>
  </li>
  <li>
    <input type="button" onclick="multiProcessing(this.name);" value="Intersection"/>
  </li>
</ul>
```

Please reload the page. You should then be able to run your multiple geometries services and you should get results displayed in red as shown by the following screenshots :



It seems that something is missing in your Services Provider to get the same results ... The multiple geometries Services ! This is what we are going to do together in the next section.

5 Exercise

You know everything now about writing zcfg metadata files and get short pieces of code in `service.c` or `ogr_service_provider.py` depending if you choose C or Python programming language respectively.

The goal of this exercise is to implement the following multiple geometries services :

- ✓ Intersection
- ✓ Union
- ✓ Difference
- ✓ SymDifference

5.1 C version

You are now invited to edit the `source.c` file you have created during this workshop to add the multiple geometries, using the following OGR C-API functions :

- ✓ `OGR_G_Intersection(OGRGeometryH, OGRGeometryH)`
- ✓ `OGR_G_Union(OGRGeometryH, OGRGeometryH)`
- ✓ `OGR_G_Difference(OGRGeometryH, OGRGeometryH)`
- ✓ `OGR_G_SymmetricDifference(OGRGeometryH, OGRGeometryH)`

You can use the `Boundary.zcfg` file as example, rename the `InputPolygon` input to `InputEntity1` and add a similar input named `InputEntity2`. You are invited to update other values in the ZOO Metadata File to set the proper metadata informations.

5.2 Python Version

You are invited to edit the `ogr_ws_service_provider.py` file you created during this workshop to add the multiple geometries using the following `osgeo.ogr` Geometry methods applied on the first Geometry instance :

- ✓ `Intersection(Geometry)`

- ✓ `Union(Geometry)`

- ✓ `Difference(Geometry)`

- ✓ `SymmetricDifference(Geometry)`

You can once again use the `Boundary.zcfg` file as example, rename the `InputPolygon` input to `InputEntity1` and add a similar input named `InputEntity2`. You are invited to update other values in the ZOO metadata file to set the proper metadata informations.

5.3 Testing your services

Once the multiple geometries Services are deployed on your local environment, please reload the `zoo-ogr.html` created during the previous section from your browser and test your brand new ZOO Services.

Stay tuned !

ZOO Official website: <http://www.zoo-project.org>

ZOO Discuss mailing list: zoo-discuss@gisws.media.osaka-cu.ac.jp

ZOO IRC Chanel: [#zoo_project@irc.freenode.net](irc://irc.freenode.net/#zoo_project)



ZOO Twitter:

http://www.twitter.com/ZOO_Project



ZOO Linkedin Group:

<http://www.linkedin.com/groups?home=&gid=2532284>