ZOO Project Documentation

Release 1.2

Nicolas Bozon, Gérald Fenoy, Jeff McKenna

June 17, 2013

CONTENTS

Authors Nicolas Bozon, Gérald Fenoy, Jeff McKenna

Last Updated $Date: 2012-06-04 15:49:13 +0200 (Lun, 04 jui 2012) $

CHAPTER

ONE

QUICK LINKS

Table 1.1: Quick Links

ZOO Kernel Documentation *Installation* *Documentation Development Guide*	*ZOO Services Documentation* *ZOO Community* *Workshop: Practical Introduction to ZOO: The Open WPS Platform*	*ZOO API Documentation* Search mailing list *workshop-foss4g-cee-2012*

Note: The entire documentation is also available as a single PDF document  and ePub document

CHAPTER

TWO

ABOUT ZOO DOCUMENTATION

ZOO Documentation is a collaborative process; once users contribute documentation through the ZOO Wiki, the Documentation Committee will then review the contributions to add them into the official docs. The committee is currently composed of:

Nicolas Bozon (nicolas.bozon at gmail.com)
Gérald Fenoy (gerald.fenoy at geolabs.fr)
Jeff McKenna (jmckenna at gatewaygeomatics.com)

A special thank to the ZOO-Project sponsors and knwoledge partners for their support.

The committee would like to thank Venkatesh Raghavan for his help on promoting ZOO Workshop in many places.

The committee would like to thank Hirofumi Hayashi and Daisuke Yoshida for their help and japanese translation effort for the ZOO-Project Worshop 2010.

The committee would also like to thank the early contributors to the documentation: Luca Delucchi, René-Luc D'Hont, Angelos Tzotsos and Guillaume Sueur.

CHAPTER

THREE

ZOO WIKI

Users wishing to contribute steps and documents should use the ZOO Wiki.

CHAPTER

FOUR

OFFICIAL DOCUMENTATION

This section contains the official ZOO Project documentation.

4.1 ZOO Kernel Documentation

The following sections will assist you with the ZOO Kernel:

4.1.1 Introduction

Authors Nicolas Bozon, Gérald Fenoy, Jeff McKenna

Last Updated $Date: 2011-12-07 14:19:47 +0100 (Mer, 07 déc 2011) $

ZOO Kernel is the heart of the ZOO. It is a powerful server-side C Kernel which makes it possible to manage and chain Web services, by loading dynamic libraries and handling them as on-demand Web services. The ZOO Kernel is written in the C language, but supports several common programming languages in order to connect to numerous libraries and models.

Using ZOO Kernel as a Web Processing Platform

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!

Note: If you'd like some background on the WPS standard, head to: http://www.opengeospatial.org/standards/wps

---

Supported Languages

ZOO Kernel supports the following programming languages, and let's you use them to create new ZOO Services from new or existing code:

Language	ServiceProvider	DataStructure	Return
C / C++	Shared Library	maps* M	integer
Fortran	Shared Library	CHARACTER*(1024) M(10,30)	integer
Java	Class File	HashMap	integer
Python	Module File	dictionary	integer
PHP	Script File	Array	integer
Perl	Script File		integer
JavaScript	Script file	Object or Array	Array/Object

ZOO Kernel is a WPS Espresso Machine

4.1.2 Installation

Authors Nicolas Bozon, Gérald Fenoy, Jeff McKenna

Last Updated $Date: 2011-12-07 14:44:57 +0100 (Mer, 07 déc 2011) $

This page provides documentation on how to compile then install the ZOO Kernel on Unix, Windows, and Mac OS X platforms.

Prerequisites

Authors Nicolas Bozon, Gérald Fenoy, Jeff McKenna

Last Updated $Date: 2011-12-07 14:44:57 +0100 (Mer, 07 déc 2011) $

Table of Contents

Obtaining the ZOO Kernel Source

Use the following command to get the ZOO Kernel source code through Subversion:

```
svn checkout http://svn.zoo-project.org/svn/trunk zoo-project
```

For users which get a developer account, use the following:

```
sed "s:\[tunnels\]:\[tunnels\]\nzoosvn = /usr/bin/ssh -p 1046:g" -i ~/.subversion/config
svn co svn+zoosvn://svn.zoo-project.org/var/svn/repos/trunk zoo-project
```

The first line of the instruction above defines a specific tunnel to access the svn server through the SSH protocol. Indeed, the ZOO SVN server listens on the 1046 (1024+22) port rather than the default one (22).

Prerequisites

The following libraries are required on your system before you can install the ZOO Kernel:

- autoconf (http://www.gnu.org/software/autoconf/)
- cgic (http://www.boutell.com/cgic)
- cURL (http://curl.haxx.se)
- FastCGI (http://www.fastcgi.com)
- Flex & Bison (http://flex.sourceforge.net/ http://www.gnu.org/software/bison/)
- libxml2 (http://xmlsoft.org)
- OpenSSL (http://www.openssl.org)
- Python (http://www.python.org)

Optional libraries include:

- PHP Embedded (optional) (http://www.php.net)
- Java SDK (optional) (http://java.sun.com)
- SpiderMonkey (optional) (http://www.mozilla.org/js/spidermonkey/)

Compile libcgic

The first step is to compile libcgic from the `zoo-project/thirds` directory. For such a task, please use the following command:

```
cd thirds/cgic206
make
```

Make sure that a `libcgic.a` is created in your `zoo-project/thirds/cgic206` directory. If yes, then you can go to the next step.

On Windows, rather than using the make command, please use:

```
nmake /f makefile.vc.
```

> **Warning:** If you don't compile libcgic first, and try to compile the ZOO Kernel, you will get an error such as *cannot find -lcgic*

Unix

Authors Nicolas Bozon, Gérald Fenoy, Jeff McKenna

Last Updated $Date: 2011-12-07 14:44:57 +0100 (Mer, 07 déc 2011) $

Table of Contents

Note: You must be sure to perform the *prerequisite steps* before following this page.

For the impatient

To build the `zoo_loader.cgi` CGI program with the default options, cd to the directory where you extracted the ZOO Kernel source code package and use the following commands:

```
$ cd zoo-kernel
$ autoconf
$ ./configure
$ make
```

Unless something went wrong, you should have executables in the current directory for the `zoo_loader.cgi` CGI program. You can copy the `zoo_loader.cgi` program and the `main.cfg` file to your HTTP server's CGI directory and start using it.

At this step your ZOO-Kernel should work. Nevertheless, don't forget to correct the main.cfg settings to set `tmpPath` and `tmpUrl` to fit your web server configuration.

Configure Options

Here is the list of available options as returned by *./configure –help*:

```
--with-gdal-config=FILE specify an alternative gdal-config file
--with-xml2config=FILE  specify an alternative xml2-config file
--with-python=PATH      To enable python support or specify an alternative
                        directory for python installation, disabled by
                        default
--with-php=PATH         To enable php support or specify an alternative
                        directory for php installation, disabled by default
--with-perl=PATH        To enable perl support or specify an alternative
                        directory for perl installation, disabled by default
--with-java=PATH        To enable java support, specify a JDK_HOME,
                        disabled by default
--with-js=PATH          specify --with-js=path-to-js to enable js support,
                        specify --with-js on linux debian like, js support
                        is disabled by default
```

All the options are described in more details below.

(Required) GDAL Support If your gdal-config program is not found in your PATH then you can use the `--with-gdal-config` option to specify its location. For instance, let's suppose that your gdal-config was installed in /usr/local/bin and this directory is not in your PATH, then you can use the following command:

```
$ ./configure --with-gdal-config=/usr/local/bin/gdal-config
```

(Required) XML2 Support If your xml2-config program is not found in your PATH then you can use the `--with-xml2config` option to specify its location. For instance, let's suppose that your xml2-config was installed in /usr/local/bin and this directory is not in your PATH, then you can use the following command:

```
$ ./configure --with-xml2config=/usr/local/bin/xml2-config
```

(Optional) Python Support If you want to activate Python support for the ZOO Kernel then you will have to use the `--with-python` option. If your python-config program is found in your PATH then you don't have to specify the path where Python was installed, such as:

```
$ ./configure --with-python
```

This assumes that python-config is found in your PATH.

In the case that your python-config is not found in your PATH, then you can specify the Python installation directory you are using. For instance, let's suppose that you installed Python in /usr/local, then you can use the following command:

```
$ ./configure --with-python=/usr/local
```

This assumes that /usr/local/bin/python-config exists.

(Optional) PHP Support To be able to activate PHP support for the ZOO Kernel you'll need to get a local PHP Embedded installation; for more information about the required configure options when compiling PHP you can refer to this page :

> http://zoo-project.org/trac/wiki/ZooKernel/Embed/PHP

If you want to activate the PHP support for the ZOO Kernel then you will have to use the `--with-php` option. If your php-config program is found in your PATH then you don't have to specify the path where PHP was installed, then you can use the following commnd:

```
$ ./configure --with-php
```

This assumes that php-config is found in your PATH.

In the case that your php-config is not found in your PATH, then you can specify the PHP installation directory you are using. For instance, let's suppose that you installed PHP in /usr/local, then you can use the following command:

```
$ ./configure --with-php=/usr/local
```

This assumes that /usr/local/bin/php-config exists.

(Optional) Perl Support If you want to activate Perl support for the ZOO Kernel then you will have to use the `--with-perl` option. If you do not set any value to this option, then the perl program will be searched in your PATH. So in such a case, you can use the following command:

```
$ ./configure --with-perl
```

This assumes that perl is found in your PATH.

In the other case, for custom Perl installations, you can set the installation directory. For instance, let's suppose that you installed Perl in /usr/local and /usr/local/bin is not in your PATH, then you can use the following command:

```
$ ./configure --with-perl=/usr/local
```

This assumes that /usr/local/bin/perl exists.

(Optional) Java Support If you want to activate Java support for the ZOO Kernel then you will have to use the `--with-java` option and set the installation path of your Java SDK. For instance, let's suppose that your Java SDK was installed in the /usr/lib/jvm/java-6-sun-1.6.0.22/ directory, then you can use the following command:

```
$ ./configure --with-java=/usr/lib/jvm/java-6-sun-1.6.0.22/
```

This assumes that the include/linux and jre/lib/i386/client/ subdirectories exist in /usr/lib/jvm/java-6-sun-1.6.0.22/, include/linux contains the jni.h headers file and jre/lib/i386/client/ contains the libjvm.so file.

Note: With Mac OS X you only have to set `macos` as the value for the `--with-java` option to activate Java support. For example:

```
$ ./configure --with-java=macos
```

(Optional) JavaScript Support If you want to activate JavaScript support for the ZOO Kernel then you will have to use the `--with-js` option. If you are using a "Debian-like" GNU/Linux distribution then dpkg will be used to detect if the required packages are installed and you don't have to specify anything here, so you can use the following command:

```
$ ./configure --with-js
```

This assumes that js_api.h and libmozjs.so are found in default directories.

If you have a custom installation of SpiderMonkey or you are not using a Debian packaging system, then you'll have to specify the directory where you installed it. For instance, let's suppose that you installed your SpiderMonkey in /usr, then you'll have to use the following command:

```
$ ./configure --with-js=/usr
```

This assumes that the /usr/include/js exists and contains the js_api.h headers file and /usr/lib contains libmozjs.so file.

OpenSUSE

Authors Nicolas Bozon, Gérald Fenoy, Jeff McKenna

Last Updated $Date: 2011-12-07 14:44:57 +0100 (Mer, 07 déc 2011) $

Table of Contents

Zoo-Kernel is maintained as a package in OpenSUSE Build Service (OBS). This way, rpm's are provided for all versions of openSUSE Linux (11.2, 11.3, 11.4, Factory).

Stable Releases

For installing Zoo-Kernel in openSUSE there are 3 ways available:

One Click Installer One-click installer that can be found here. For openSUSE 11.4 this is the direct link.

Yast Software Manager using a GUI The Application:Geo repository has to be added to the Software Repositories and then Zoo-kernel can be found in Software Management through search.

Command line (as root for openSUSE 11.4)

```
zypper ar http://download.opensuse.org/repositories/Application:/Geo/openSUSE_11.4/
zypper refresh
zypper install zoo-kernel
```

Unstable Version

The latest development version of ZOO-Kernel can be found in OBS under the project home:tzotsos. ZOO-Kernel packages are maintained and tested there before being released to the Application:Geo repository.

Installation methods are identical as the stable version. Make sure to use this repository instead.

Command line installation (as root for openSUSE 11.4)

```
zypper ar http://download.opensuse.org/repositories/home:/tzotsos/openSUSE_11.4/
zypper refresh
zypper install zoo-kernel
zypper install zoo-kernel-grass-bridge
```

Additionally, there is the option of adding the zoo-wps-grass-bridge package. This option will automatically install grass7 (svn trunk).

Try the Installation

- http://localhost/cgi-bin/zoo_loader.cgi?ServiceProvider=&metapath=&Service=WPS&Request=GetCapabilities&Version=1.0.0
- http://localhost/cgi-bin/zoo_loader.cgi?ServiceProvider=&metapath=&Service=WPS&Request=DescribeProcess&Version=1.0.0
- http://localhost/cgi-bin/zoo_loader.cgi?ServiceProvider=&metapath=&Service=WPS&Request=Execute&Version=1.0.0&Identif

CentOS

Authors Nicolas Bozon, Gérald Fenoy, Jeff McKenna

Last Updated $Date: 2011-12-07 14:44:57 +0100 (Mer, 07 déc 2011) $

Table of Contents

Note: This documentation was created thanks to Guillaume Sueur from Neogeo Technologies which took time to test installing the ZOO-Kernel on a CentOS 5.5 environment.

Requirements

Install some standard tools to be able to run ZOO-Kernel on your platform :

```
yum install apache2
yum install build-essentials
yum install gcc-c++
yum install zlib-devel
yum install libxml2-devel
yum install bison
yum install openssl
yum install python-devel
yum install subversion
```

Compile then install FastCGI library from source

```
wget http://www.fastcgi.com/dist/fcgi.tar.gz
tar xzf fcgi-2.4.0.tar.gz
./configure
make
make install
echo /usr/local/lib >> /etc/ld.so.conf.d/local.conf
ldconfig
```

Compile then install the autoconf tools :

```
wget http://ftp.gnu.org/gnu/autoconf/autoconf-latest.tar.gz
tar xzf autoconf-latest.tar.gz
./configure --prefix=/usr
make
make install
```

Compile then install the flex tool :

```
wget http://downloads.sourceforge.net/project/flex/flex/flex-2.5.35/flex-2.5.35.tar.gz?r=http%3A%2F%2
tar xzf flex-2.5.35.tar.gz
cd flex-2.5.35
./configure --prefix=/usr
make
make install
```

Using the curl provided in the CentOS distribution will produce a ZOO-Kernel unable to run any Service. Indeed, some segmentation faults occur when trying to run `Execute` requests on the ZOO-Kernel, compiling the ZOO-Kernel setting `USE_GDB` flag in the `CFLAGS` of your `Makefile` will let you run ZOO-Kernel from gdb and be able to get more information on what is going wrong with your ZOO-Kernel. Doing this we can figure out that code on line 173 and line 175 have to be commented in the `ulinet.c` file to get a ZOO-Kernel working using the curl available in CentOS (curl version 7.15.5). If you don't apply the modification, you will get an error from a gdb session pointing segfault in `Curl_cookie_clearall`.

You can optionally compile then install curl from source :

```
wget http://curl.haxx.se/download/curl-7.21.3.tar.bz2
tar xjf curl-7.21.3.tar.bz2
```

```
cd curl-7.21.3
./configure --prefix=/usr
make
make install
```

Compile then install Python :

```
wget http://www.python.org/ftp/python/2.6.6/Python-2.6.6.tar.bz2
tar xjf Python-2.6.6.tar.bz2
cd Python-2.6.6
./configure
make
make install
```

Compile then install your own GDAL library :

```
wget http://download.osgeo.org/gdal/gdal-1.7.3.tar.gz
tar xzf gdal-1.7.3.tar.gz
cd gdal-1.7.3
./configure  # add your options here
make
make install
```

Install the Sun JAVA SDK into `/usr/share` then use the following command to ensure that the `libjvm.so` will be found at runtime from any context.

```
echo /usr/share/java-1.6.0-openjdk-1.6.0.0/jre/lib/i386/client/ >> /etc/ld.so.conf.d/jvm.conf
ldconfig
```

Compile ZOO-Kernel and ZOO-Services

Compile then install ZOO-Kernel and your first ZOO-Services.

First of all, compile the cgic library providen in the SVN source tree:

```
svn co http://svn.zoo-project.org/svn/trunk zoo-project
cd zoo-project/thirds/cgic206
make
```

Compile then install ZOO-Kernel.

```
cd ../../zoo-kernel
./configure --with-java=/usr/share/jdk1.6.0_23/ --with-python
make zoo_loader.cgi
cp main.cfg /var/www/cgi-bin/
cp  zoo_loader.cgi /var/www/cgi-bin/
```

Compile then deploy your first ZOO-ServicesProviders (simple HelloPy, line 1 and 2, and the OGR base-vect-ops ServiceProvider, line 3 to 6):

```
cp ../zoo-services/hello-py/cgi-env/*.zcfg /var/www/cgi-bin/
cp ../zoo-services/hello-py/test_service.py /var/www/cgi-bin/
cd ../ogr/base-vect-ops/
make
cp ./cgi-env/* /var/www/cgi-bin/
vi /var/www/cgi-bin/main.cfg --> set your own informations here
```

To ensure that the `libjvm.so` will be found from apache, please restart it :

```
/etc/init.d/httpd restart
```

Testing your ZOO-Kernel

Test your ZOO-Kernel from command line:

```
cd /var/www/cgi-bin
./zoo_loader.cgi "request=Execute&service=WPS&version=1.0.0&Identifier=HelloPy&DataInputs=a=Djay"
./zoo_loader.cgi "request=Execute&service=WPS&version=1.0.0&Identifier=Buffer&DataInputs=BufferDistar
```

Debian / Ubuntu

Authors Nicolas Bozon, Gérald Fenoy, Jeff McKenna, Luca Delucchi

Last Updated $Date: 2011-12-07 14:19:47 +0100 (Mer, 07 déc 2011) $

Table of Contents

Note: An Ubuntu 10.4 with ZOO virtual image is available at http://www.zoo-project.org/Ubuntu10.4_ZOO.zip (root: ZOO.test)

The following instructions were tested on Debian Squeeze, Ubuntu 10.04 and Ubuntu 10.10

Installation Workflow

- install some dependencies

```
sudo apt-get install flex bison libfcgi-dev libxml2 libxml2-dev curl openssl autoconf checkinstall
```

- download ZOO source

```
svn checkout http://svn.zoo-project.org/svn/trunk zoo-project
```

- install cgic from packages

```
cd zoo-project/thirds/cgic206/
```

- compile

```
make
```

- go to kernel path

```
cd ../../zoo-kernel/
```

- create configure file

```
autoconf
```

- run configure

```
./configure --with-java=/path/to/java
./configure --with-python
```

Note: In Ubuntu 10.04 libmozjs-dev does not exist, so to use JavaScript you can compile SpiderMonkey (remember to put /usr/local/lib in a file inside /etc/ld.so.conf.d/). For PHP, you must make sure to compile PHP with –enable-embed.

- to use JavaScript with compiled SpiderMonkey.

```
sudo echo "/usr/local/lib" > /etc/ld.so.conf.d/libmoz185.conf
sudo ldconfig
./configure --with-js=/usr/local
```

- to use JavaScript with Xulrunner .

```
sudo echo "/usr/lib/xulrunner-dev" > /etc/ld.so.conf.d/libmoz185.conf
sudo ldconfig
./configure --with-js=/usr/lib/xulrunner-dev
```

- compile

```
make zoo_loader.cgi
```

- copy necessary files into your cgi-bin

```
sudo cp main.cfg /usr/lib/cgi-bin
sudo cp zoo_loader.cgi /usr/lib/cgi-bin
```

- install ZOO ServiceProvider (in this case we try Python service)

```
sudo cp ../zoo-services/hello-py/cgi-env/*.zcfg /usr/lib/cgi-bin
sudo cp ../zoo-services/hello-py/*.py /usr/lib/cgi-bin/
```

- change some paths in the main.cfg

```
sudo nano /usr/lib/cgi-bin/main.cfg
  - serverAddress = http://127.0.0.1
  - providerSite = http://127.0.0.1
```

- try the installation
  - http://127.0.0.1/cgi-bin/zoo_loader.cgi?ServiceProvider=&metapath=&Service=WPS&Request=GetCapabilities&Version=
  - http://127.0.0.1/cgi-bin/zoo_loader.cgi?ServiceProvider=&metapath=&Service=WPS&Request=DescribeProcess&Version=
  - http://127.0.0.1/cgi-bin/zoo_loader.cgi?ServiceProvider=&metapath=&Service=WPS&Request=Execute&Version=1.0.0&I

Note: If you have some problem in the execute request using Python service, add the following to `main.cfg`:

```
[env]
PYTHONPATH=<YOUR_PYTHONPATH>
```

Windows

Authors Nicolas Bozon, Gérald Fenoy, Jeff McKenna

Last Updated $Date: 2011-12-07 14:44:57 +0100 (Mer, 07 déc 2011) $

Table of Contents

Using OSGeo4W

Install OSGeo4W Download the OSGeo4W installer from http://trac.osgeo.org/osgeo4w/, and install it with all the dependencies needed by your services (GDAL/OGR for example). The following libs are required: FastCGI, libxml, Python, cURL.

Install other tools and libraries After installing OSGeo4W on your platform you'll need more GNU tools and libraries. This package contains full dependencies required to compile on WIN32 platform and this one contains full runtime dependencies to place in your c:OSGeo4Wbin.

Download and Install ZOO Kernel Download the binary version of the ZOO Kernel for WIN32 then place it in the `C:\OSGeo4W\bin` directory. Don't forget to place a `main.cfg` file in the same directory, you can use a modified copy of this file.

Deploy ZOO Services Providers Your can use the binary version of the OGR Services Provider available from here. Then place the two libraries with their respective .zcfg files in your local `C:\OSGeo4W\bin` directory.

Testing Now you should be able to query your local ZOO Kernel.

Compiling Using Your Own Libraries

Note: You must be sure to perform the *prerequisite steps* before compiling the ZOO Kernel.

The following steps are for use with the Microsoft Visual Studio compiler (and tested with MSVC 2008).

1. Make sure the gnuwin32 tools `bison.exe` and `flex.exe` are found in your path. You can download the GNUwin32 tools here.
2. Modify the file `zoo-project\zoo-kernel\nmake.opt` to point to your local libraries. You can find a modified nmake.opt that points to local libs here. You can also find a modified `zoo-project\zoo-kernel\makefile.vc` file here.
3. Execute:

   ```
   nmake /f makefile.vc
   ```

4. A file `zoo_loader.cgi` should be created. Note that if another file named `zoo_loader.cgi.manifest` is also created, you will have to run another command:

   ```
   nmake /f makefile.vc embed-manifest
   ```

5. Copy the files `zoo_loader.cgi` and `main.cfg` into your cgi-bin directory.
6. Using the command prompt, test the zoo-kernel by executing the following command:

```
D:\ms4w\Apache\cgi-bin> zoo_loader.cgi
```

which should display a message such as:

```
Content-Type: text/xml; charset=utf-8
Status: 200 OK

<?xml version="1.0" encoding="utf-8"?>
<ows:ExceptionReport xmlns:ows="http://www.opengis.net/ows/1.1" xmlns:xsi="http://www.w3.org/200
  <ows:Exception exceptionCode="MissingParameterValue">
    <ows:ExceptionText>Parameter &lt;request&gt; was not specified</ows:ExceptionText>
  </ows:Exception>
</ows:ExceptionReport>
```

7. Edit the file `cgi-bin/main.cfg` so that it contains values describing your WPS service. An example of such a file running on Windows is:

```
[main]
encoding = utf-8
version = 1.0.0
serverAddress = http://localhost/
lang = en-CA
tmpPath=/ms4w/tmp/ms_tmp/
tmpUrl = /ms_tmp/

[identification]
title = The Zoo WPS Development Server
abstract = Development version of ZooWPS. See http://www.zoo-project.org
fees = None
accessConstraints = none
keywords = WPS,GIS,buffer

[provider]
providerName=Gateway Geomatics
providerSite=http://www.gatewaygeomatics.com
individualName=Jeff McKenna
positionName=Director
role=Dev
adressDeliveryPoint=1101 Blue Rocks Road
addressCity=Lunenburg
addressAdministrativeArea=False
addressPostalCode=B0J 2C0
addressCountry=ca
addressElectronicMailAddress=info@gatewaygeomatics.com
phoneVoice=False
phoneFacsimile=False
```

8. Open a web browser window, and execute a GetCapababilites request on your WPS service: http://localhost/cgi-bin/zoo_loader.cgi?request=GetCapabilities&service=WPS

   The response should be displayed in your browser, such as:

```
<wps:Capabilities xsi:schemaLocation="http://www.opengis.net/wps/1.0.0 http://schemas.opengis.ne
<ows:ServiceIdentification>
  <ows:Title>The Zoo WPS Development Server</ows:Title>
  <ows:Abstract>
    Development version of ZooWPS. See http://www.zoo-project.org
  </ows:Abstract>
  <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>
  ...
```

Optionally Compile Individual Services An example could be the `OGR base-vect-ops` provider in the `zoo-project\zoo-services\ogr\base-vect-ops` directory.

1. Edit the makefile.vc located in that directory, and execute:

   ```
   nmake /f makefile.vc
   ```

   Inside that same directory, the file `cgi-env\ogr_service.zo` should be created.

2. Copy all of the files inside `zoo-services\ogr\base-vect-ops\cgi-env` into your cgi-bin directory

3. Test this service provider through the following URL:

   http://localhost/cgi-bin/zoo_loader.cgi?request=Execute&service=WPS&version=1.0.0&Identifier=Buffer&DataInputs=BufferDi
   project.org%3A8082%2Fgeoserver%2Fows%3FSERVICE%3DWFS%26REQUEST%3DGetFeature%26VERSION%3D1.0.0%

   The response displayed in your browser should contain:

   ```
   <wps:ProcessSucceeded>Service "Buffer" run successfully.</wps:ProcessSucceeded>
   ```

Mac OS X

Authors Nicolas Bozon, Gérald Fenoy, Jeff McKenna

Last Updated $Date: 2011-12-07 14:44:57 +0100 (Mer, 07 déc 2011) $

Table of Contents

Using the Installer

1. To install a default build of the ZOO-Project on your Mac OS X computer use the installer.

   Note: The installer assumes that you are using the distributed Apache2 version that comes with your Mac. The installer will place ZOO-Kernel and ZOO-Services into your cgi-bin at `/Library/WebServer/CGI-Executables`, and the zoo-demo folder will be placed within your document root at `/Library/WebServer/Documents`

   ---

2. Make sure that your Apache server is running, and then access the ZOO Project Demo at:

   http://localhost/zoo-demo/spatialtools.html

3. To add additional services, please follow the following intructions to compile your own ZOO Project instance.

Compiling from Source

1. Install Xcode.
2. Before you start downloading the ZOO-Project source code, you'll need to install some tools required to compile ZOO-Kernel properly.

   First of all install PROJ, GEOS and GDAL frameworks from here.

   At this step, you should get the following directories on your local hard drive :

   ```
   /Library/Frameworks/PROJ.framework
   /Library/Frameworks/GEOS.framework
   /Library/Frameworks/GDAL.framework
   ```

3. Then, create a `src` directory and inside that directory download the gettext source code and uncompress it.

   now, compile gettext with the following commands to produce a universal binary :

   ```
   cd gettext-0.18.1.1
   CFLAGS="-O -g -arch i386 -arch ppc -arch x86_64"  \
     LDFLAGS="-arch i386 -arch ppc -arch x86_64"   ./configure
   make
   sudo make install
   ```

4. Compile and install your ZOO-Kernel
   - Download source from SVN, and use the following command to compile libcgic :

     ```
     svn co http://svn.zoo-project.org/svn/trunk zoo
     cd zoo/thirds/cgic206
     make
     ```

   - If you produced the `libcgic.a` file, you can run `autoconf` and then `configure` from zoo-kernel directory.

     ```
     cd zoo/zoo-kernel
     autoconf
     ./configure --with-python --with-java=macos \
        --with-gdal-config=/Library/Frameworks/GDAL.framework/Versions/1.8/Programs/gdal-config
     ```

     Obviously, if you don't need Python or Java support then you should remove the corresponding configure option.

     Note: Note that we used the –with-java=macos configure option. Due to the generic location of the JDK on all Mac OS X platforms, you don't have to provide its full path.

   - Now, run the following commands to compile and deploy your ZOO-Kernel on your Apache server :

     ```
     make
     cp zoo_loader.cgi main.cfg /Library/WebServer/CGI-Executables
     ```

     You should be ready to request your ZOO-Kernel installation using the following link : http://localhost/cgi-bin/zoo_loader.cgi?request=GetCapabilities&service=WPS .

     If everything is ok, you can follow the next steps to deploy new Services Providers.

Note: If you are using your own libs (not the default libs on your system) then you must take care to create universal versions of those libs, as the ZOO-Kernel will try to create a universal binary. If you are not following this advice, you might receive compile errors of `symbol(s) not found for architecture ppc` or `file was built for unsupported file format which is not the architecture being linked (ppc).`

Deploy the OGR Services Provider

Requirements Before your try to use any service, please set the correct path in the `main.cfg` for tmpPath and tmpUrl.

You can use the following setup :

```
tmpPath = /Library/WebServer/Documents/tmp
tmpUrl = ../../tmp
```

Obviously you'll then need to create this directory, using the following command :

```
mkdir /Library/WebServer/Documents/tmp
```

C Version To compile the base-vect-ops ServicesProvider you'll need to edit the Makefile in *zoo/zoo-services/ogr/base-vect-ops/* directory. Add "-I/Library//Frameworks/GEOS.framework/Versions/3/Headers/" to the CFLAGS value on the first line. To compile, add GDAL framework to the PATH environmenet variable, to ensure that gdal-config tool will be found, run make and then copy cgi-env files in the /Library/WebServer/CGI-Executables directory.

```
cd zoo/zoo-services/ogr/base-vect-ops/
export PATH=$PATH:/Library/Frameworks/GDAL.framework/Versions/1.7/Programs/
make
cp cgi-env/* /Library/WebServer/CGI-Executables
```

You can test using this url if everything is ok with your setup.

Python Version Requirements

First of all run python from a Terminal.app and try the following import from the python interpreter :

```
import osgeo.ogr
import libxml2
```

If you get an issue when importing the libxml2 module from your python interpreter then that means you need to install the Python support for the libxml2 library which is already installed on your Mac OS X environment. To accomplish this, you have first to determine what version of libxml2 is installed on your platform, using the following command:

xml2-config –version

Download the source corresponding to your version (i.e. on 10.6.6 you get 2.7.3) from the libxml2 download page into your `src` directory then uncompress it.

Use the following command to install the python support :

```
cd src/libxml2-2.7.3/python/
python setup.py install
```

Deploy OGR Python Services Provider

- Now copy the `zoo-services/ogt/base-vect-ops/cgi-env` files into `/Library/WebServer/CGI-Executables`.

You can test using this url if everything is ok with your setup.

Test using Local Demo Page

- Download the OpenLayers library and uncompress it in your personal Sites directory (located in your home directory).
- Rename the OpenLayers directory as openlayers.
- Download this zip archive and then uncompress it in your personal Sites directory.
- Load your local demo pages using urls similar to the following (replacing MyUserName by your MacOS user name) :
  - http://localhost/~MyUserName/zoo-demo/spatialtools.html
  - http://localhost/~MyUserName/zoo-demo/spatialtools-py.html

4.2 ZOO Services Documentation

The following sections will assist you with ZOO Services:

4.2.1 Introduction

ZOO Services are example Web services which work with the ZOO *Kernel*. They are based on various existing Open Source libraries and tend to provide simple Web processing functions such as GIS format conversion, GIS file reprojection, basic spatial operations, basic raster operations...

The available ZOO Services are under development and come without any warranty. They are based on existing code and prove that the *ZOO Kernel* works with many different codes and languages (please have a look to the ZOO demos!). The ZOO Project team wants to encourage people to use the ZOO Services as a functional basis for Web processing, but above all to provide a source of inspiration to the community for creating new ZOO Services.

What is a ZOO Service?

A ZOO Service is a couple composed of:

- The code you want to turn into a standardized Web service
- A configuration file (.zcfg) which describes this Web service

Learn more on configuring ZOO services by reading the *.zcfg Reference*.

Example ZOO Services

See the ZOO Services *examples page*.

4.2.2 ZCFG : the ZOO Service Configuration File

Authors Nicolas Bozon, Gérald Fenoy, Jeff McKenna

Last Updated $Date: 2011-12-07 14:44:57 +0100 (Mer, 07 déc 2011) $

Table of Contents

The ZOO Service configuration file (.zcfg) describes the service and will be parsed by the ZOO Kernel. We will describe here what such a file contains. You can also take a look at the existing examples of ZCFG files in the `cgi-env` directory of each services available in the ZOO-Project SVN source tree.

A ZOO Configuration file is divided into three distinct sections :

1. Main Metadata information
2. List of Inputs metadata information
3. List of Outputs metadata information

Note: The ZOO Service Configuration File is case sensitive.

Main Metadata Information

The fist part in a ZOO Configuration file contains the metadata information relative to the service. Note that the "name of your service" between brackets on the first line has to be the exact same name as the function you defined in your services provider code. In most cases, this name is also the name of the ZCFG file without the "`.zcfg`" extension.

You can see below a description of the main metadata information:

```
[Name of your service]
Title = Title of your service
Abstract = Description of your service
processVersion = Version number of your service
storeSupported = true/false
statusSupported = true/false
serviceType = the programming language used to implement the service (C/Fortran/Python/Java/PHP/Javas
serviceProvider = name of your services provider (shared library/Python Module/Java Class/PHP Script/
<MetaData>
  title = Metadata title of your service
</MetaData>
```

List of Inputs

The list of inputs contains metadata information of each supported input, and they are grouped using a `<DataInputs>` node.

Each input is defined as :

- a name (between brackets as for the name of the service before)
- various medata properties (`Title`, `Abstract`, `minOccurs` and `maxOccurs`)
- a Type Of Data node (*description*)

A typical list of inputs (`<DataInputs>`) look like the following:

```
<DataInputs>
  [Name of the first input]
    Title = Title of the first input
    Abstract = Abstract describing the first input
    minOccurs = Minimum occurence of the first input
    maxOccurs = Maximum occurence of the first input
    <Type Of Data Node />
  [Name of the second input]
    Title = Title of the second input
    Abstract = Abstract describing the second input
    minOccurs = Minimum occurence of the second input
    maxOccurs = Maximum occurence of the second input
    <Type Of Data Node />
</DataInputs>
```

Note: you can add `<MetaData>` node as in the main metadata information.

List of Outputs

The list of outputs is very similar to a list of inputs except it is specified as a `<DataOutputs>` node.

A typical `<DataOutputs>` node looks like the following:

```
<DataOutputs>
  [Name of the output]
    Title = Title of the output
    Abstract = Description of the output
    <Type Of Data Node />
</DataOutputs>
```

Type Of Data Nodes

In the beginning of this ZCFG introduction, we spoke about "Type Of Data Nodes" to describe the data type of inputs and outputs.

You can define your data as:

- *LiteralData*
- *BoundingBoxData*
- *ComplexData*

Except for `LiteralData`, each *Type Of Data* node must have at least one `<Default>` and one `<Supported>` node. Even if one of those are empty, it **has to be present** with an opening and closing tag on two different lines. So, something like the following:

```
<Default>
</Default>
```

Otherwise, ZOO-Kernel won't be able to parse your ZCFG and will fail to process requests.

LiteralData node

A <LiteralData> node contains:

- one <Default> node,
- zero or more <Supported> nodes depending on the existence or the number of supported Units Of Measure (UOM), and
- a dataType property. The dataType property defines the type of literal data, such as a string, an interger and so on (consult the complete list of supported data types).

<Default> and <Supported> nodes can contain the uom property to define which UOM has to be used for this input value.

For input <LiteralData> nodes, you can add the value property to the <Default> node to define a default value for this input. This means that, when your Service will be run, even if the input wasn't defined, this default value will be set as the current value for this input.

A typical <LiteralData> node, defining a float data type using meters or degrees for its UOM, looks like the following:

```
<LiteralData>
  dataType = float
  <Default>
    uom = meters
  </Default>
  <Supported>
    uom = feet
  </Supported>
</LiteralData>
```

BoundingBoxData node

A <BoundingBoxData> node contains:

- one <Default> node with a CRS property defining the default Coordinate Reference Systems (CRS), and
- one or more <Supported> nodes depending on the number of CRS your service supports (note that you can alternatively use a single <Supported> node with a comma-separated list of supported CRS).

A typical <BoundingBoxData> node, for two supported CRS (EPSG:4326 and EPSG:3785), looks like the following:

```
<BoundingBoxData>
  <Default>
    CRS = urn:ogc:def:crs:EPSG:6.6:4326
  </Default>
  <Supported>
    CRS = urn:ogc:def:crs:EPSG:6.6:4326
  </Supported>
  <Supported>
    CRS = urn:ogc:def:crs:EPSG:6.6:3785
```

```
  </Supported>
</BoundingBoxData>
```

ComplexData node

A ComplexData node contains:

- a `<Default>` node and
- one or more `<Supported>` nodes depending on the number of supported formats. A format is made up of this set of properties : `mimeType`, `encoding` and optionaly `schema`.

For output ComplexData nodes, you can add the `extension` property to define what extension to use to name the file when storing the result is required. Obviously, you'll have to add the `extension` property to each supported format (for the `<Default>` and `<Supported>` nodes).

You can also add the `asReference` property to the `<Default>` node to define if the output should be stored on server side per default.

Note: the client can always modify this behavior by setting `asReference` attribute to `true` or `false` for this output in the request `ResponseDocument` parameter.

You can see below a sample ComplexData node for default `application/json` and `text/xml` (encoded in UTF-8 or base64) mimeTypes support:

```
<ComplexData>
  <Default>
    mimeType = application/json
    encoding = UTF-8
  </Default>
  <Supported>
    mimeType = text/xml
    encoding = base64
    schema = http://fooa/gml/3.1.0/polygon.xsd
  </Supported>
  <Supported>
    mimeType = text/xml
    encoding = UTF-8
    schema = http://fooa/gml/3.1.0/polygon.xsd
  </Supported>
</ComplexData>
```

4.2.3 How To Setup ZOO Services

Authors Nicolas Bozon, Gérald Fenoy, Jeff McKenna

Last Updated $Date: 2011-12-07 14:19:47 +0100 (Mer, 07 déc 2011) $

ZOO Services are quite easy to create once you have installed the ZOO Kernel and have chosen code (in the language of your choice) to turn into a ZOO service. Here are some HelloWorlds in Python, PHP, Java and JavaScript with links to their corresponding `.zcfg` files.

Table of Contents

Python

You'll find here information needed to deploy your own Python Services Provider.

Python ZCFG requirements

Note: For each Service provided by your ZOO Python Services Provider, the ZCFG File must be named the same as the Python module function name.

The ZCFG file should contain the following :

serviceType Python

serviceProvider The name of the Python module to use as a ZOO Service Provider. For instance, if your script, located in the same directory as your ZOO Kernel, was named `my_module.py` then you should use `my_module` (the Python module name) for the serviceProvider value in ZCFG file.

Python Data Structure used

The Python module's function to be used as a service must:

- take three arguments : the main configuration, inputs and outputs maps are passed to the Python module as dictionaries.
- return an integer : corresponding to the service status code.

Sample Data Structure

In the following you'll find a sample argument passed to the Python module's function for the two first main configuration file' sections.

```
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"}
}
```

Sample ZOO Python Services Provider

The following code represents a simple ZOO Python Services Provider which provides only one Service, the HelloPy one.

```
import sys
def HelloPy(conf,inputs,outputs):
   outputs["Result"]["value"]="Hello "+inputs["a"]["value"]+" from Python World !"
   return 3
```

PHP

```
<?
function HelloPHP(&$main_conf,&$inputs,&$outputs){
   $outputs["Result"]["value"]="Hello ".$inputs[S][value]." from PHP world !";
   return 3;
}
?>
```

Java

```
import java.util.*;
public class HelloJava {
  public static int HelloWorldJava(HashMap conf,HashMap inputs, HashMap outputs) {
     HashMap hm1 = new HashMap();
     hm1.put("dataType","string");
     HashMap tmp=(HashMap)(inputs.get("S"));
     java.lang.String v=tmp.get("value").toString();
     hm1.put("value","Hello "+v+" from JAVA WOrld !");
     outputs.put("Result",hm1);
     System.err.println("Hello from JAVA WOrld !");
     return 3;
  }
}
```

Javascript

```
function hellojs(conf,inputs,outputs){
   outputs=new Array();
   outputs={};
   outputs["result"]["value"]="Hello "+inputs["S"]["value"]+" from JS World !";
   return Array(3,outputs);
}
```

4.2.4 Service Examples

Authors Nicolas Bozon, Gérald Fenoy, Jeff McKenna

Last Updated $Date: 2011-12-07 14:44:57 +0100 (Mer, 07 déc 2011) $

ZOO Services are quite easy to create once you have installed the ZOO Kernel and have chosen code (in the language of your choice) to turn into a ZOO service. Here are some HelloWorlds with links to their corresponding `.zcfg` files.

Table of Contents

GDAL ZOO Service

The GDAL ZOO Service is based on GDAL source code and copyright. This ZOO Service aims to provide some basic raster processing operations to your ZOO Kernel installation. Learn more and read documentation on the official GDAL website.

Implemented functions

Gdal_Grid creates regular grid from the scattered data

Gdal_Translate converts raster data between different formats

OGR ZOO Service

The OGR ZOO Service is based on OGR source code and copyright. This ZOO Service aims to provide some basic vector spatial operations to your ZOO Kernel installation. Learn more and read documentation on the official OGR website.

Implemented functions

Base_Vect_Ops

4.3 ZOO API Documentation

The following sections will assist you with the ZOO API:

4.3.1 Description

ZOO API is a Javascript library designed to make the WPS Process creation and chaining easier. It works on the server-side using the Mozilla foundation JavaScript engine, SpiderMonkey. It uses a Proj4js adaptation for server-side reprojection. It also allows to easily convert vector formats (such as GML, KML, GeoJSON, etc). The API lets you orchestrate WPS services simply and offers the ability to add logic and controls in the WPS chaining.

Download ZOO API

- zoo-api.js
- zoo-proj4js.js

ZOO API MIT/X-11 License

4.3.2 Classes

Authors Nicolas Bozon, Gérald Fenoy, Jeff McKenna

Last Updated $Date: 2011-12-07 14:44:57 +0100 (Mer, 07 déc 2011) $

The following classes are available in the ZOO API:

ZOO

The following constants and functions are available for the ZOO class:

Constants

NAME	DESCRIPTION
SERVICE_ACCEPTED	{Integer} used for
SERVICE_STARTED	{Integer} used for
SERVICE_PAUSED	{Integer} used for
SERVICE_SUCCEEDED	{Integer} used for
SERVICE_FAILED	{Integer} used for

Functions

NAME	DESCRIPTION
removeItem	Remove an object from an array.
indexOf	
extend	Copy all properties of a source object to a destination object.
rad	
distVincenty	Given two objects representing points with geographic coordinates, this calculates the distance between those points on the surface of an ellipsoid.
Class	Method used to create ZOO classes.
UpdateStatus	Method used to update the status of the process

Constants

SERVICE_ACCEPTED `{Integer}` used for

SERVICE_STARTED `{Integer}` used for

SERVICE_PAUSED `{Integer}` used for

SERVICE_SUCCEEDED `{Integer}` used for

SERVICE_FAILED `{Integer}` used for

Functions

removeItem

```
removeItem: function(array,item)
```

Remove an object from an array. Iterates through the array to find the item, then removes it.

Parameters

```
array {Array}
item {Object}
```

Returns

`{Array}` A reference to the array

indexOf

```
indexOf: function(array,obj)
```

Parameters

```
array {Array}
obj {Object}
```

Returns

`{Integer}` The index at, which the first object was found in the array. If not found, returns -1.

extend

`extend: function(destination,source)`

Copy all properties of a source object to a destination object. Modifies the passed in destination object. Any properties on the source object that are set to undefined will not be (re)set on the destination object.

Parameters

`destination {Object}` The object that will be modified
`source {Object}` The object with properties to be set on the destination

Returns

`{Object}` The destination object.

rad

`rad: function(x)`

Parameters

`x {Float}`

Returns

`{Float}`

distVincenty

`distVincenty: function(p1,p2)`

Given two objects representing points with geographic coordinates, this calculates the distance between those points on the surface of an ellipsoid.

Parameters:

`p1` *{ZOO.Geometry.Point}* (or any object with both .x, .y properties)
`p2` *{ZOO.Geometry.Point}* (or any object with both .x, .y properties)

Class

`Class: function()`

Method used to create ZOO classes. Includes support for multiple inheritance.

UpdateStatus

`UpdateStatus: function(env,value)`

Method used to update the status of the process

Parameters

`env {Object}` The environment object

`value {Float}` The status value between 0 to 100

ZOO.Bounds

Instances of this class represent bounding boxes.

Properties

NAME	DESCRIPTION
left	{Number} Minimum horizontal coordinate.
bottom	{Number} Minimum vertical coordinate.
right	{Number} Maximum horizontal coordinate.
top	{Number} Maximum vertical coordinate.

Functions

NAME	DESCRIPTION
ZOO.Bounds	Construct a new bounds object.
clone	Create a cloned instance of this bounds.
equals	Test a two bounds for equivalence.
toString	{String} String representation of bounds object.
toBBOX	
toGeometry	Create a new polygon geometry based on this bounds.
getWidth	{Float} The width of the bounds
getHeight	{Float} The height of the bounds (top minus bottom)
add	
extend	Extend the bounds to include the point, lonlat, or bounds specified.
intersectsBounds	Determine whether the target bounds intersects this bounds.
containsBounds	Determine whether the target bounds is contained within this bounds.

Properties

left `{Number}` Minimum horizontal coordinate.

bottom `{Number}` Minimum vertical coordinate.

right `{Number}` Maximum horizontal coordinate.

top `{Number}` Maximum vertical coordinate.

Functions

ZOO.Bounds Construct a new bounds object.

Parameters

`left {Number}` The left bounds of the box. Note that for width calculations, this is assumed to be less than the right value.

`bottom {Number}` The bottom bounds of the box. Note that for height calculations, this is assumed to be more than the top value.

`right {Number}` The right bounds.

`top {Number}` The top bounds.

clone

`clone:function()`

Create a cloned instance of this bounds.

Returns

{ZOO.Bounds} A fresh copy of the bounds

equals

`equals:function(bounds)`

Test a two bounds for equivalence.

Parameters

`bounds {ZOO.Bounds}`

Returns

`{Boolean}` The passed-in bounds object has the same left, right, top, bottom components as this. Note that if bounds passed in is null, returns false.

toString

`toString:function()`

Returns

`{String}` String representation of bounds object. (ex. *"left-bottom=(5,42) right-top=(10,45)"*)

toBBOX

`toBBOX:function(decimal)`

Parameters

`decimal {Integer}` How many significant digits in the bbox coords? Default is 6

Returns

`{String}` Simple String representation of bounds object. (ex. *"5,42,10,45"*)

toGeometry

`toGeometry: function()`

Create a new polygon geometry based on this bounds.

Returns

{ZOO.Geometry.Polygon} A new polygon with the coordinates of this bounds.

getWidth

`getWidth:function()`

Returns

`{Float}` The width of the bounds

getHeight

`getHeight:function()`

Returns

`{Float}` The height of the bounds (top minus bottom).

add

`add:function(x,y)`

Parameters

`x {Float}`
`y {Float}`

Returns

{ZOO.Bounds} A new bounds whose coordinates are the same as this, but shifted by the passed-in x and y values.

extend

`extend:function(object)`

Extend the bounds to include the point, lonlat, or bounds specified. Note, this function assumes that left < right and bottom < top.

Parameters

`object {Object}` Can be Point, or Bounds.

intersectsBounds

`intersectsBounds:function(bounds,inclusive)`

Determine whether the target bounds intersects this bounds. Bounds are considered intersecting if any of their edges intersect or if one bounds contains the other.

Parameters

`bounds` *{ZOO.Bounds}* The target bounds.
`inclusive {Boolean}` Treat coincident borders as intersecting. Default is true. If false, bounds that do not overlap but only touch at the border will not be considered as intersecting.

Returns

`{Boolean}` The passed-in bounds object intersects this bounds.

containsBounds

`containsBounds:function(bounds,partial,inclusive)`

Determine whether the target bounds is contained within this bounds.

Parameters

`bounds` *{ZOO.Bounds}* The target bounds.

`partial {Boolean}` If any of the target corners is within this bounds consider the bounds contained. Default is false. If true, the entire target bounds must be contained within this bounds.

`inclusive {Boolean}` Treat shared edges as contained. Default is true.

Returns

`{Boolean}` The passed-in bounds object is contained within this bounds.

ZOO.Feature

Vector features use the ZOO.Geometry classes as geometry description.

Properties

NAME	DESCRIPTION
fid	{String}
geometry	{ZOO.Geometry}
attributes	{Object} This object holds arbitrary properties that describe the feature.
bounds	{ZOO.Bounds} The box bounding that feature's geometry, that property can be set by an ZOO.Format object when deserializing the feature, so in most cases it represents an information set by the server.

Functions

NAME	DESCRIPTION
ZOO.Feature	Create a vector feature.
destroy	nullify references to prevent circular references and memory leaks
clone	Create a clone of this vector feature.
move	Moves the feature and redraws it at its new location

Properties

fid `{String}`

geometry *{ZOO.Geometry}*

attributes `{Object}` This object holds arbitrary properties that describe the feature.

bounds *{ZOO.Bounds}* The box bounding that feature's geometry, that property can be set by an ZOO.Format object when deserializing the feature, so in most cases it represents an information set by the server.

Functions

ZOO.Feature Create a vector feature.

Parameters

`geometry` *{ZOO.Geometry}* The geometry that this feature represents.

`attributes {Object}` An optional object that will be mapped to the attributes property.

destroy

```
destroy: function()
```

nullify references to prevent circular references and memory leaks

clone

```
clone: function ()
```

Create a clone of this vector feature. Does not set any non-standard properties.

Returns

{ZOO.Feature} An exact clone of this vector feature.

move Moves the feature and redraws it at its new location

ZOO.Format

Base class for format reading/writing a variety of formats.

Properties

NAME	DESCRIPTION
options	{Object} A reference to options passed to the constructor.
externalProjection	{ZOO.Projection} When passed a externalProjection and internalProjection, the format will reproject the geometries it reads or writes.
internalProjection	{ZOO.Projection} When passed a externalProjection and internalProjection, the format will reproject the geometries it reads or writes.
data	{Object} When keepData is true, this is the parsed string sent to read.
keepData	{Object} Maintain a reference (data) to the most recently read data.

Functions

NAME	DESCRIPTION
ZOO.Format	Instances of this class are not useful.
destroy	Clean up.
read	Read data from a string, and return an object whose type depends on the subclass.
data	{Object} When keepData is true, this is the parsed string sent to read.
write	Accept an object, and return a string.

Properties

options `{Object}` A reference to options passed to the constructor.

externalProjection *{ZOO.Projection}* When passed a externalProjection and internalProjection, the format will reproject the geometries it reads or writes. The externalProjection is the projection used by the content which is passed into read or which comes out of write. In order to reproject, a projection transformation function for the specified projections must be available. This support is provided via zoo-proj4js.

internalProjection *{ZOO.Projection}* When passed a externalProjection and internalProjection, the format will reproject the geometries it reads or writes. The internalProjection is the projection used by the geometries which are returned by read or which are passed into write. In order to reproject, a projection transformation function for the specified projections must be available. This support is provided via zoo-proj4js.

data `{Object}` When *keepData* is true, this is the parsed string sent to *read*.

keepData `{Object}` Maintain a reference (*data*) to the most recently read data. Default is false.

Functions

ZOO.Format Instances of this class are not useful. See one of the subclasses.

Parameters

`options {Object}` An optional object with properties to set on the format

Valid options

`keepData {Boolean}` If true, upon read, the data property will be set to the parsed object (e.g. the json or xml object).

Returns

An instance of ZOO.Format

destroy

```
destroy: function()
```

Clean up.

read

```
read: function(data)
```

Read data from a string, and return an object whose type depends on the subclass.

Parameters

`data {string}` Data to read/parse.

Returns

Depends on the subclass

write

```
write: function(data)
```

Accept an object, and return a string.

Parameters

`object {Object}` Object to be serialized

Returns

`{String}` A string representation of the object.

ZOO.Format.GeoJSON

Read and write GeoJSON.

Inherits from

- *ZOO.Format.JSON*

Properties

NAME	DESCRIPTION
ZOO.Format.GeoJSON	Create a new parser for GeoJSON.
read	Deserialize a GeoJSON string.
isValidType	Check if a GeoJSON object is a valid representative of the given type.
parseFeature	Convert a feature object from GeoJSON into an ZOO.Feature.
parseGeometry	Convert a geometry object from GeoJSON into an ZOO.Geometry.

parseCoords Properties

NAME	DESCRIPTION
parseCoords	Object with properties corresponding to the GeoJSON geometry types.

parseCoords Functions

NAME	DESCRIPTION
parseCoords.point	Convert a coordinate array from GeoJSON into an ZOO.Geometry.Point.
parseCoords.multipoint	Convert a coordinate array from GeoJSON into an ZOO.Geometry.MultiPoint.
parseCoords.linestring	Convert a coordinate array from GeoJSON into an ZOO.Geometry.LineString.
parseCoords.multilinestring	Convert a coordinate array from GeoJSON into an ZOO.Geometry.MultiLineString.
parseCoords.polygon	Convert a coordinate array from GeoJSON into an ZOO.Geometry.Polygon.
parseCoords.multipolygon	Convert a coordinate array from GeoJSON into an ZOO.Geometry.MultiPolygon.
parseCoords.box	Convert a coordinate array from GeoJSON into an ZOO.Geometry.Polygon.
write	Serialize a feature, geometry, array of features into a GeoJSON string.
createCRSObject	Create the CRS object for an object.

extract Properties

NAME	DESCRIPTION
extract	Object with properties corresponding to the GeoJSON types.

extract Functions

NAME	DESCRIPTION
extract.feature	Return a partial GeoJSON object representing a single feature.
extract.geometry	Return a GeoJSON object representing a single geometry.
extract.point	Return an array of coordinates from a point.
extract.multipoint	Return an array of coordinates from a multipoint.
extract.linestring	Return an array of coordinate arrays from a linestring.
extract.multilinestring	Return an array of linestring arrays from a linestring.
extract.polygon	Return an array of linear ring arrays from a polygon.
extract.multipolygon	Return an array of polygon arrays from a multipolygon.
extract.collection	Return an array of geometries from a geometry collection.

Functions

ZOO.Format.GeoJSON Create a new parser for GeoJSON.

Parameters

`options {Object}` An optional object whose properties will be set on this instance.

read

`read: function(json,type,filter)`

Deserialize a GeoJSON string.

Parameters

`json {String}` A GeoJSON string

`type {String}` Optional string that determines the structure of the output. Supported values are "Geometry", "Feature", and "FeatureCollection". If absent or null, a default of "FeatureCollection" is assumed.

`filter {Function}` A function which will be called for every key and value at every level of the final result. Each value will be replaced by the result of the filter function. This can be used to reform generic objects into instances of classes, or to transform date strings into Date objects.

Returns

`{Object}` The return depends on the value of the type argument. If type is "FeatureCollection" (the default), the return will be an array of *ZOO.Feature*. If type is "Geometry", the input json must represent a single geometry, and the return will be an *ZOO.Geometry*. If type is "Feature", the input json must represent a single feature, and the return will be an *ZOO.Feature*.

isValidType

`isValidType: function(obj,type)`

Check if a GeoJSON object is a valid representative of the given type.

Returns

`{Boolean}` The object is valid GeoJSON object of the given type.

parseFeature

`parseFeature: function(obj)`

Convert a feature object from GeoJSON into a *ZOO.Feature*.

Parameters

`obj {Object}` An object created from a GeoJSON object

Returns

{ZOO.Feature} A feature.

parseGeometry

`parseGeometry: function(obj)`

Convert a geometry object from GeoJSON into a *ZOO.Geometry*.

Parameters

`obj {Object}` An object created from a GeoJSON object

Returns

{ZOO.Geometry} A geometry.

parseCoords Properties

parseCoords Object with properties corresponding to the GeoJSON geometry types. Property values are functions that do the actual parsing.

parseCoords Functions

parseCoords.point Convert a coordinate array from GeoJSON into a *ZOO.Geometry.Point*.

Parameters

`array {Object}` The coordinates array from the GeoJSON fragment.

Returns

{ZOO.Geometry.Point} A geometry.

parseCoords.multipoint Convert a coordinate array from GeoJSON into a *ZOO.Geometry.MultiPoint*.

Parameters

`array {Object}` The coordinates array from the GeoJSON fragment.

Returns

{ZOO.Geometry.MultiPoint} A geometry.

parseCoords.linestring Convert a coordinate array from GeoJSON into a *ZOO.Geometry.LineString*.

Parameters

`array {Object}` The coordinates array from the GeoJSON fragment.

Returns

{ZOO.Geometry.LineString} A geometry.

parseCoords.multilinestring Convert a coordinate array from GeoJSON into a *ZOO.Geometry.MultiLineString*.

Parameters

`array {Object}` The coordinates array from the GeoJSON fragment.

Returns

{ZOO.Geometry.MultiLineString} A geometry.

parseCoords.polygon Convert a coordinate array from GeoJSON into a *ZOO.Geometry.Polygon*.

Parameters

`array {Object}` The coordinates array from the GeoJSON fragment.

Returns

{ZOO.Geometry.Polygon} A geometry.

parseCoords.multipolygon Convert a coordinate array from GeoJSON into a *ZOO.Geometry.MultiPolygon*.

Parameters

`array {Object}` The coordinates array from the GeoJSON fragment.

Returns

{ZOO.Geometry.MultiPolygon} A geometry.

parseCoords.box Convert a coordinate array from GeoJSON into a *ZOO.Geometry.Polygon*.

Parameters

`array {Object}` The coordinates array from the GeoJSON fragment.

Returns

{ZOO.Geometry.Polygon} A geometry.

write

```
write: function(obj,pretty)
```

Serialize a feature, geometry, array of features into a GeoJSON string.

Parameters

`obj {Object}` A *{ZOO.Feature}*, *{ZOO.Geometry}*, or an array of features.
`pretty {Boolean}` Structure the output with newlines and indentation. Default is false.

Returns

`{String}` The GeoJSON string representation of the input geometry, features, or array of features.

createCRSObject

```
createCRSObject: function(object)
```

Create the CRS object for an object.

Parameters

`object` *{ZOO.Feature}*

Returns

`{Object}` An object which can be assigned to the crs property of a GeoJSON object.

extract Properties

extract Object with properties corresponding to the GeoJSON types. Property values are functions that do the actual value extraction.

extract Functions

extract.feature Return a partial GeoJSON object representing a single feature.

Parameters

`feature` *{ZOO.Feature}*

Returns

`{Object}` An object representing the point.

extract.geometry Return a GeoJSON object representing a single geometry.

Parameters

`geometry` *{ZOO.Geometry}*

Returns

`{Object}` An object representing the geometry.

extract.point Return an array of coordinates from a point.

Parameters

`point` *{ZOO.Geometry.Point}*

Returns

`{Array}` An array of coordinates representing the point.

extract.multipoint Return an array of coordinates from a multipoint.

Parameters

`multipoint` *{ZOO.Geometry.MultiPoint}*

Returns

`{Array}` An array of point coordinate arrays representing the multipoint.

extract.linestring Return an array of coordinate arrays from a linestring.

Parameters

`linestring` *{ZOO.Geometry.LineString}*

Returns

`{Array}` An array of coordinate arrays representing the linestring.

extract.multilinestring Return an array of linestring arrays from a linestring.

Parameters

`multilinestring` *{ZOO.Geometry.MultiLineString}*

Returns

`{Array}` An array of linestring arrays representing the multilinestring.

extract.polygon Return an array of linear ring arrays from a polygon.

Parameters

`polygon` *{ZOO.Geometry.Polygon}*

Returns

`{Array}` An array of linear ring arrays representing the polygon.

extract.multipolygon Return an array of polygon arrays from a multipolygon.

Parameters

`multipolygon` *{ZOO.Geometry.MultiPolygon}*

Returns

`{Array}` An array of polygon arrays representing the multipolygon

extract.collection Return an array of geometries from a geometry collection.

Parameters

`collection` *{ZOO.Geometry.Collection}*

Returns

`{Array}` An array of geometry objects representing the geometry collection.

ZOO.Format.GML

Read/Write GML.

Inherits from

- *ZOO.Format*

Properties and Functions

NAME	DESCRIPTION
schemaLocation	{String} Schema location for a particular minor version.
namespaces	{Object} Mapping of namespace aliases to namespace URIs.
defaultPrefix	
collectionName	{String} Name of featureCollection element.
featureName	{String} Element name for features.
geometryName	{String} Name of geometry element.
xy	{Boolean} Order of the GML coordinate true:(x,y) or false:(y,x) Changing is not recommended, a ne
extractAttributes	{Boolean} Could we extract attributes
ZOO.Format.GML	Create a new parser for GML.
read	Read data from a string, and return a list of features.
parseFeature	This function is the core of the GML parsing code in ZOO.
parseGeometry	Properties of this object are the functions that parse geometries based on their type.
parseGeometry.point	Given a GML node representing a point geometry, create a ZOO point geometry.
parseGeometry.multipoint	Given a GML node representing a point geometry, create a ZOO multipoint geometry.
parseGeometry.linestring	Given a GML node representing a linestring geometry, create a ZOO linestring geometry.
parseGeometry.polygon	Given a GML node representing a polygon geometry, create a ZOO polygon geometry.
parseGeometry.multigeometry	Given a GML node representing a multigeometry, create a ZOO geometry collection.
parseAttributes	
parseExtendedData	Parse ExtendedData from GML.
write	Accept Feature Collection, and return a string.
createFeature	Accept an ZOO.Feature, and build a GML node for it.
buildGeometryNode	Builds and returns a GML geometry node with the given geometry.
buildGeometry	Object containing methods to do the actual geometry node building based on geometry type.
buildGeometry.point	Given a ZOO point geometry, create a GML point.
buildGeometry.multipoint	Given a ZOO multipoint geometry, create a GML GeometryCollection.
buildGeometry.linestring	Given a ZOO linestring geometry, create a GML linestring.
buildGeometry.multilinestring	Given a ZOO multilinestring geometry, create a GML GeometryCollection.
buildGeometry.linearring	Given a ZOO linearring geometry, create a GML linearring.
buildGeometry.polygon	Given a ZOO polygon geometry, create a GML polygon.
buildGeometry.multipolygon	Given a ZOO multipolygon geometry, create a GML GeometryCollection.
buildGeometry.collection	Given a ZOO geometry collection, create a GML MultiGeometry.
buildCoordinatesNode	builds the coordinates XmlNode

schemaLocation `{String}` Schema location for a particular minor version.

namespaces `{Object}` Mapping of namespace aliases to namespace URIs.

defaultPrefix

collectionName `{String}` Name of featureCollection element.

featureName `{String}` Element name for features. Default is “sql_statement”.

geometryName `{String}` Name of geometry element. Defaults to “geometryProperty”.

xy `{Boolean}` Order of the GML coordinate true:(x,y) or false:(y,x) Changing is not recommended, a new Format should be instantiated.

extractAttributes `{Boolean}` Could we extract attributes

ZOO.Format.GML Create a new parser for GML.

Parameters

`options {Object}` An optional object whose properties will be set on this instance.

read

```
read: function(data)
```

Read data from a string, and return a list of features.

Parameters

`data {String}` data to read/parse.

Returns

{Array(ZOO.Feature)} An array of features.

parseFeature

```
parseFeature: function(node)
```

This function is the core of the GML parsing code in ZOO. It creates the geometries that are then attached to the returned feature, and calls parseAttributes() to get attribute data out.

Parameters

`node {E4XElement}`

Returns

{ZOO.Feature} A vector feature.

parseGeometry Properties of this object are the functions that parse geometries based on their type.

parseGeometry.point Given a GML node representing a point geometry, create a ZOO point geometry.

Parameters

`node {E4XElement}` A GML node.

Returns

{ZOO.Geometry.Point} A point geometry.

parseGeometry.multipoint Given a GML node representing a multipoint geometry, create a ZOO multipoint geometry.

Parameters

`node {E4XElement}` A GML node.

Returns

{ZOO.Geometry.Point} A multipoint geometry.

parseGeometry.linestring Given a GML node representing a linestring geometry, create a ZOO linestring geometry.

Parameters

`node {E4XElement}` A GML node.

Returns

{ZOO.Geometry.LineString} A linestring geometry.

parseGeometry.multilinestring Given a GML node representing a multilinestring geometry, create a ZOO multilinestring geometry.

Parameters

`node {E4XElement}` A GML node.

Returns

{ZOO.Geometry.MultiLineString} A multilinestring geometry.

parseGeometry.polygon Given a GML node representing a polygon geometry, create a ZOO polygon geometry.

Parameters

`node {E4XElement}` A GML node.

Returns

{ZOO.Geometry.Polygon} A polygon geometry.

parseGeometry.multipolygon Given a GML node representing a multipolygon geometry, create a ZOO multipolygon geometry.

Parameters

`node {E4XElement}` A GML node.

Returns

{ZOO.Geometry.MultiPolygon} A multipolygon geometry.

parseGeometry.envelope Given a GML node representing an envelope, create a ZOO polygon geometry.

Parameters

`node {E4XElement}` A GML node.

Returns

{ZOO.Geometry.Polygon} A polygon geometry.

parseAttributes

```
parseAttributes: function(node)
```

Parameters

`node {E4XElement}`

Returns

`{Object}` An attributes object.

write

```
write: function(features)
```

Generate a GML document string given a list of features.

Parameters

`features` *{Array(ZOO.Feature)}* List of features to serialize into a string.

Returns

`{String}` A string representing the GML document.

createFeature

`createFeature: function(feature)`

Accept an ZOO.Feature, and build a GML node for it.

Parameters

`feature` *{ZOO.Feature}* The feature to be built as GML.

Returns

`{E4XElement}` A node reprensting the feature in GML.

buildGeometryNode

`buildGeometryNode: function(geometry)`

Parameters

`geometry` *{ZOO.Geometry}* The geometry to be built as GML.

Returns

`{E4XElement}` A node reprensting the geometry in GML.

buildGeometry Object containing methods to do the actual geometry node building based on geometry type.

buildGeometry.point Given a ZOO point geometry, create a GML point.

Parameters

`geometry` *{ZOO.Geometry.Point}* A point geometry.

Returns

`{E4XElement}` A GML point node.

buildGeometry.multipoint Given a ZOO multipoint geometry, create a GML multipoint.

Parameters

`geometry` *{ZOO.Geometry.MultiPoint}* A multipoint geometry.

Returns

`{E4XElement}` A GML multipoint node.

buildGeometry.linestring Given a ZOO linestring geometry, create a GML linestring.

Parameters

`geometry` *{ZOO.Geometry.LineString}* A linestring geometry.

Returns

`{E4XElement}` A GML linestring node.

buildGeometry.multilinestring Given a ZOO multilinestring geometry, create a GML multilinestring.

Parameters

`geometry` *{ZOO.Geometry.MultiLineString}* A multilinestring geometry.

Returns

`{E4XElement}` A GML multilinestring node.

buildGeometry.linearring Given a ZOO linearring geometry, create a GML linearring.

Parameters

`geometry` *{ZOO.Geometry.LinearRing}* A linearring geometry.

Returns

`{E4XElement}` A GML linearring node.

buildGeometry.polygon Given an ZOO polygon geometry, create a GML polygon.

Parameters

`geometry` *{ZOO.Geometry.Polygon}* A polygon geometry.

Returns

`{E4XElement}` A GML polygon node.

buildGeometry.multipolygon Given a ZOO multipolygon geometry, create a GML multipolygon.

Parameters

`geometry` *{ZOO.Geometry.MultiPolygon}* A multipolygon geometry.

Returns

`{E4XElement}` A GML multipolygon node.

buildCoordinatesNode

`buildCoordinatesNode: function(geometry)`

builds the coordinates XmlNode

```
<gml:coordinates decimal="." cs="," ts=" ">...</gml:coordinates>
```

Parameters

`geometry` *{ZOO.Geometry}*

Returns

`{E4XElement}` created E4XElement

ZOO.Format.JSON

A parser to read/write JSON safely.

Inherits from

- *ZOO.Format*

Properties

NAME	DESCRIPTION
indent	{String} For "pretty" printing, the indent string will be used once for each indentation level.
space	{String} For "pretty" printing, the space string will be used after the ":" separating a name/value pair.
newline	{String} For "pretty" printing, the newline string will be used at the end of each name/value pair or array item.
level	{Integer} For "pretty" printing, this is incremented/decremented during serialization.
pretty	{Boolean} Serialize with extra whitespace for structure.

Functions

NAME	DESCRIPTION
ZOO.Format.JSON	Create a new parser for JSON.
read	Deserialize a json string.
write	Serialize an object into a JSON string.
writeIndent	Output an indentation string depending on the indentation level.
writeNewline	Output a string representing a newline if in pretty printing mode.
writeSpace	Output a string representing a space if in pretty printing mode.

Serialize Properties

NAME	DESCRIPTION
serialize	Object with properties corresponding to the serializable data types.

Serialize Functions

NAME	DESCRIPTION
serialize.object	Transform an object into a JSON string.
serialize.array	Transform an array into a JSON string.
serialize.string	Transform a string into a JSON string.
serialize.number	Transform a number into a JSON string.
serialize.boolean	Transform a boolean into a JSON string.
serialize.date	Transform a date into a JSON string.

Properties

indent `{String}` For "pretty" printing, the indent string will be used once for each indentation level.

space `{String}` For "pretty" printing, the space string will be used after the ":" separating a name/value pair.

newline `{String}` For "pretty" printing, the newline string will be used at the end of each name/value pair or array item.

level `{Integer}` For "pretty" printing, this is incremented/decremented during serialization.

pretty `{Boolean}` Serialize with extra whitespace for structure. This is set by the *write* method.

Functions

ZOO.Format.JSON Create a new parser for JSON.

Parameters

`options {Object}` An optional object whose properties will be set on this instance.

read

```
read: function(json,filter)
```

Deserialize a json string.

Parameters

`json {String}` A JSON string

`filter {Function}` A function which will be called for every key and value at every level of the final result. Each value will be replaced by the result of the filter function. This can be used to reform generic objects into instances of classes, or to transform date strings into Date objects.

Returns

`{Object}` An object, array, string, or number.

write

```
write: function(value,pretty)
```

Serialize an object into a JSON string.

Parameters

`value {String}` The object, array, string, number, boolean or date to be serialized.
`pretty {Boolean}` Structure the output with newlines and indentation. Default is false.

Returns

`{String}` The JSON string representation of the input value.

writeIndent

```
writeIndent: function()
```

Output an indentation string depending on the indentation level.

Returns

`{String}` An appropriate indentation string.

writeNewline

```
writeNewline: function()
```

Output a string representing a newline if in pretty printing mode.

Returns

`{String}` A string representing a new line.

writeSpace

```
writeSpace: function()
```

Output a string representing a space if in pretty printing mode.

Returns

`{String}` A space.

Serialize Properties

serialize Object with properties corresponding to the serializable data types. Property values are functions that do the actual serializing.

Serialize Functions

serialize.object Transform an object into a JSON string.

Parameters

`object {Object}` The object to be serialized.

Returns

`{String}` A JSON string representing the object.

serialize.array Transform an array into a JSON string.

Parameters

`array {Array}` The array to be serialized

Returns

`{String}` A JSON string representing the array.

serialize.string Transform a string into a JSON string.

Parameters

`string {String}` The string to be serialized

Returns

`{String}` A JSON string representing the string.

serialize.number Transform a number into a JSON string.

Parameters

`number {Number}` The number to be serialized.

Returns

`{String}` A JSON string representing the number.

serialize.boolean Transform a boolean into a JSON string.

Parameters

`bool {Boolean}` The boolean to be serialized.

Returns

`{String}` A JSON string representing the boolean.

serialize.date Transform a date into a JSON string.

Parameters

`date {Date}` The date to be serialized.

Returns

`{String}` A JSON string representing the date.

ZOO.Format.KML

Read/Write KML.

Inherits from

- *ZOO.Format*

Properties and Functions

NAME	DESCRIPTION
kmlns	{String} KML Namespace to use.
foldersName	{String} Name of the folders.
foldersDesc	{String} Description of the folders.
placemarksDesc	{String} Name of the placemarks.
extractAttributes	{Boolean} Extract attributes from KML.
ZOO.Format.KML	Create a new parser for KML.
parseFeatures	Loop through all Placemark nodes and parse them.
parseFeature	This function is the core of the KML parsing code in ZOO.
parseGeometry	Properties of this object are the functions that parse geometries based on their type.
parseGeometry.point	Given a KML node representing a point geometry, create a ZOO point geometry.
parseGeometry.linestring	Given a KML node representing a linestring geometry, create a ZOO linestring geometry.
parseGeometry.polygon	Given a KML node representing a polygon geometry, create a ZOO polygon geometry.
parseGeometry.multigeometry	Given a KML node representing a multigeometry, create a ZOO geometry collection.
parseAttributes	
parseExtendedData	Parse ExtendedData from KML.
write	Accept Feature Collection, and return a string.
createPlacemark	Creates and returns a KML placemark node representing the given feature.
buildGeometryNode	Builds and returns a KML geometry node with the given geometry.
buildGeometry	Object containing methods to do the actual geometry node building based on geometry type.
buildGeometry.point	Given a ZOO point geometry, create a KML point.
buildGeometry.multipoint	Given a ZOO multipoint geometry, create a KML GeometryCollection.
buildGeometry.linestring	Given a ZOO linestring geometry, create a KML linestring.
buildGeometry.multilinestring	Given a ZOO multilinestring geometry, create a KML GeometryCollection.
buildGeometry.linearring	Given a ZOO linearring geometry, create a KML linearring.
buildGeometry.polygon	Given a ZOO polygon geometry, create a KML polygon.
buildGeometry.multipolygon	Given a ZOO multipolygon geometry, create a KML GeometryCollection.
buildGeometry.collection	Given a ZOO geometry collection, create a KML MultiGeometry.
buildCoordinatesNode	Builds and returns the KML coordinates node with the given geometry <coordinates>...</coordinates>

kmlns `{String}` KML Namespace to use. Defaults to 2.2 namespace.

foldersName `{String}` Name of the folders. Default is "ZOO export". If set to null, no name element will be created.

foldersDesc `{String}` Description of the folders. Default is "Exported on [date]." If set to null, no description element will be created.

placemarksDesc `{String}` Name of the placemarks. Default is "No description available".

extractAttributes `{Boolean}` Extract attributes from KML. Default is true. Extracting styleUrls requires this to be set to true

ZOO.Format.KML Create a new parser for KML.

Parameters

`options {Object}` An optional object whose properties will be set on this instance.

parseFeatures

`parseFeatures: function(nodes)`

Loop through all Placemark nodes and parse them. Will create a list of features

Parameters

`nodes {Array}` of {E4XElement} data to read/parse.
`options {Object}` Hash of options

parseFeature

`parseFeature: function(node)`

This function is the core of the KML parsing code in ZOO. It creates the geometries that are then attached to the returned feature, and calls parseAttributes() to get attribute data out.

Parameters

`node {E4XElement}`

Returns

{ZOO.Feature} A vector feature.

parseGeometry Properties of this object are the functions that parse geometries based on their type.

parseGeometry.point Given a KML node representing a point geometry, create a ZOO point geometry.

Parameters

`node {E4XElement}` A KML Point node.

Returns

{ZOO.Geometry.Point} A point geometry.

parseGeometry.linestring Given a KML node representing a linestring geometry, create a ZOO linestring geometry.

Parameters

`node {E4XElement}` A KML LineString node.

Returns

{ZOO.Geometry.LineString} A linestring geometry.

parseGeometry.polygon Given a KML node representing a polygon geometry, create a ZOO polygon geometry.

Parameters

`node {E4XElement}` A KML Polygon node.

Returns

{ZOO.Geometry.Polygon} A polygon geometry.

parseGeometry.multigeometry Given a KML node representing a multigeometry, create a ZOO geometry collection.

Parameters

`node {E4XElement}` A KML MultiGeometry node.

Returns

{ZOO.Geometry.Collection} A geometry collection.

parseAttributes

```
parseAttributes: function(node)
```

Parameters

`node {E4XElement}`

Returns

`{Object}` An attributes object.

parseExtendedData

```
parseExtendedData: function(node)
```

Parse ExtendedData from KML. Limited support for schemas/datatypes. See http://code.google.com/apis/kml/documentation/kmlreference.html#extendeddata for more information on extendeddata.

Parameters

`node {E4XElement}`

Returns

`{Object}` An attributes object.

write

```
write: function(features)
```

Accept Feature Collection, and return a string.

Parameters

`features` *{Array(ZOO.Feature)}* An array of features.

Returns

`{String}` A KML string.

createPlacemark

```
createPlacemark: function(feature)
```

Creates and returns a KML placemark node representing the given feature.

Parameters

`feature` *{ZOO.Feature}*

Returns

`{E4XElement}`

buildGeometryNode

```
buildGeometryNode: function(geometry)
```

Builds and returns a KML geometry node with the given geometry.

Parameters

`geometry` *{ZOO.Geometry}*

Returns

`{E4XElement}`

buildGeometry Object containing methods to do the actual geometry node building based on geometry type.

buildGeometry.point Given a ZOO point geometry, create a KML point.

Parameters

`geometry` *{ZOO.Geometry.Point}* A point geometry.

Returns

`{E4XElement}` A KML point node.

buildGeometry.multipoint Given a ZOO multipoint geometry, create a KML GeometryCollection.

Parameters

`geometry` *{ZOO.Geometry.MultiPoint}* A multipoint geometry.

Returns

`{E4XElement}` A KML GeometryCollection node.

buildGeometry.linestring Given a ZOO linestring geometry, create a KML linestring.

Parameters

`geometry` *{ZOO.Geometry.LineString}* A linestring geometry.

Returns

`{E4XElement}` A KML linestring node.

buildGeometry.multilinestring Given a ZOO multilinestring geometry, create a KML GeometryCollection.

Parameters

`geometry` *{ZOO.Geometry.MultiLineString}* A multilinestring geometry.

Returns

`{E4XElement}` A KML GeometryCollection node.

buildGeometry.linearring Given a ZOO linearring geometry, create a KML linearring.

Parameters

`geometry` *{ZOO.Geometry.LinearRing}* A linearring geometry.

Returns

`{E4XElement}` A KML linearring node.

buildGeometry.polygon Given a ZOO polygon geometry, create a KML polygon.

Parameters

`geometry` *{ZOO.Geometry.Polygon}* A polygon geometry.

Returns

`{E4XElement}` A KML polygon node.

buildGeometry.multipolygon Given a ZOO multipolygon geometry, create a KML GeometryCollection.

Parameters

`geometry` *{ZOO.Geometry.Point}* A multipolygon geometry.

Returns

`{E4XElement}` A KML GeometryCollection node.

buildGeometry.collection Given a ZOO geometry collection, create a KML MultiGeometry.

Parameters

`geometry` *{ZOO.Geometry.Collection}* A geometry collection.

Returns

`{E4XElement}` A KML MultiGeometry node.

buildCoordinatesNode

```
buildCoordinatesNode: function(geometry)
```

Builds and returns the KML coordinates node with the given geometry <coordinates>...</coordinates>

Parameters

`geometry` *{ZOO.Geometry}*

Return

`{E4XElement}`

ZOO.Format.WKT

Class for reading and writing Well-Known Text.

Inherits from

- *ZOO.Format*

Functions and Properties

NAME	DESCRIPTION
ZOO.Format.WKT	Create a new parser for WKT
read	Deserialize a WKT string and return a vector feature or an array of vector features.
write	Serialize a feature or array of features into a WKT string.
extract	Object with properties corresponding to the geometry types.
parse	Object with properties corresponding to the geometry types.
parse.point	Return point feature given a point WKT fragment.
parse.multipoint	Return a multipoint feature given a multipoint WKT fragment.
parse.linestring	Return a linestring feature given a linestring WKT fragment.
parse.multilinestring	Return a multilinestring feature given a multilinestring WKT fragment.
parse.polygon	Return a polygon feature given a polygon WKT fragment.
parse.multipolygon	Return a multipolygon feature given a multipolygon WKT fragment.
parse.geometrycollection	Return an array of features given a geometrycollection WKT fragment.

ZOO.Format.WKT Create a new parser for WKT

Parameters

`options {Object}` An optional object whose properties will be set on this instance

Returns

{ZOO.Format.WKT} A new WKT parser.

read

`read: function(wkt)`

Deserialize a WKT string and return a vector feature or an array of vector features. Supports WKT for POINT, MULTIPOINT, LINESTRING, MULTILINESTRING, POLYGON, MULTIPOLYGON, and GEOMETRYCOLLECTION.

Parameters

`wkt {String}` A WKT string

Returns

{<ZOO.Feature.Vector>|Array} A feature or array of features for GEOMETRYCOLLECTION WKT.

write

`write: function(features)`

Serialize a feature or array of features into a WKT string.

Parameters

`features {<ZOO.Feature.Vector>|Array}` A feature or array of features

Returns

`{String}` The WKT string representation of the input geometries

extract Object with properties corresponding to the geometry types. Property values are functions that do the actual data extraction.

parse Object with properties corresponding to the geometry types. Property values are functions that do the actual parsing.

parse.point Return point feature given a point WKT fragment.

Parameters

`str {String}` A WKT fragment representing the point

Returns

{ZOO.Feature} A point feature

parse.multipoint Return a multipoint feature given a multipoint WKT fragment.

Parameters

`str {String}` A WKT fragment representing the multipoint

Returns

{ZOO.Feature} A multipoint feature

parse.linestring Return a linestring feature given a linestring WKT fragment.

Parameters

`str {String}` A WKT fragment representing the linestring

Returns

{ZOO.Feature} A linestring feature

parse.multilinestring Return a multilinestring feature given a multilinestring WKT fragment.

Parameters

`str {String}` A WKT fragment representing the multilinestring

Returns

{ZOO.Feature} A multilinestring feature

parse.polygon Return a polygon feature given a polygon WKT fragment.

Parameters

`str {String}` A WKT fragment representing the polygon

Returns

{ZOO.Feature} A polygon feature

parse.multipolygon Return a multipolygon feature given a multipolygon WKT fragment.

Parameters

`str {String}` A WKT fragment representing the multipolygon

Returns

{ZOO.Feature} A multipolygon feature

parse.geometrycollection Return an array of features given a geometrycollection WKT fragment.

Parameters

`str {String}` A WKT fragment representing the geometrycollection

Returns

`{Array}` An array of ZOO.Feature

ZOO.Format.WPS

Read/Write WPS.

Inherits from

- *ZOO.Format*

Functions and Properties

NAME	DESCRIPTION
schemaLocation	{String} Schema location for a particular minor version.
namespaces	{Object} Mapping of namespace aliases to namespace URIs.
read	
parseExecuteResponse	
parseData	Object containing methods to analyse data response.
parseData.complexdata	Given an Object representing the WPS complex data response.
parseData.literaldata	Given an Object representing the WPS literal data response.
parseData.reference	Given an Object representing the WPS reference response.

schemaLocation `{String}` Schema location for a particular minor version.

namespaces `{Object}` Mapping of namespace aliases to namespace URIs.

read

`read:function(data)`

Parameters

`data {String}` A WPS xml document

Returns

`{Object}` Execute response.

parseExecuteResponse

`parseExecuteResponse: function(node)`

Parameters

`node {E4XElement}` A WPS ExecuteResponse document

Returns

`{Object}` Execute response.

parseData Object containing methods to analyse data response.

parseData.complexdata Given an Object representing the WPS complex data response.

Parameters

`node {E4XElement}` A WPS node.

Returns

`{Object}` A WPS complex data response.

parseData.literaldata Given an Object representing the WPS literal data response.

Parameters

`node {E4XElement}` A WPS node.

Returns

`{Object}` A WPS literal data response.

parseData.reference Given an Object representing the WPS reference response.

Parameters

`node {E4XElement}` A WPS node.

Returns

`{Object}` A WPS reference response.

ZOO.Geometry

A Geometry is a description of a geographic object.

Properties

NAME	DESCRIPTION
id	{String} A unique identifier for this geometry.
parent	{ZOO.Geometry}This is set when a Geometry is added as component of another geometry
bounds	{ZOO.Bounds} The bounds of this geometry

Functions

NAME	DESCRIPTION
ZOO.Geometry	Creates a geometry object.
destroy	nullify references to prevent circular references and memory leaks
clone	Create a clone of this vector feature.
extendBounds	Moves the feature and redraws it at its new location
clearBounds	Nullify this components bounds and that of its parent as well.
getBounds	Get the bounds for this Geometry.
calculateBounds	Recalculate the bounds for the geometry.
toString	Returns the Well-Known Text representation of a geometry
ZOO.Geometry.fromWKT	Generate a geometry given a Well-Known Text string.

Properties

id `{String}` A unique identifier for this geometry.

parent *{ZOO.Geometry}* This is set when a Geometry is added as component of another geometry

bounds *{ZOO.Bounds}* The bounds of this geometry

Functions

ZOO.Geometry Creates a geometry object.

destroy

```
destroy: function()
```

Destroy this geometry.

clone

```
clone: function()
```

Create a clone of this geometry. Does not set any non-standard properties of the cloned geometry.

Returns

{ZOO.Geometry} An exact clone of this geometry.

extendBounds

```
extendBounds: function(newBounds)
```

Extend the existing bounds to include the new bounds. If geometry's bounds is not yet set, then set a new Bounds.

Parameters

`newBounds` *{ZOO.Bounds}*

clearBounds

`clearBounds: function()`

Nullify this components bounds and that of its parent as well.

getBounds

`getBounds: function()`

Get the bounds for this Geometry. If bounds is not set, it is calculated again, this makes queries faster.

Returns

`newBounds` *{ZOO.Bounds}*

calculateBounds

`calculateBounds: function()`

Recalculate the bounds for the geometry.

toString

`toString: function()`

Returns the Well-Known Text representation of a geometry

Returns

`{String}` Well-Known Text

ZOO.Geometry.fromWKT

`ZOO.Geometry.fromWKT = function(wkt)`

Generate a geometry given a Well-Known Text string.

Parameters

`wkt {String}` A string representing the geometry in Well-Known Text.

Returns

{ZOO.Geometry} A geometry of the appropriate class.

ZOO.Geometry.Collection

A Collection is exactly what it sounds like: A collection of different Geometries. These are stored in the local parameter *components* (which can be passed as a parameter to the constructor).

As new geometries are added to the collection, they are NOT cloned. When removing geometries, they need to be specified by reference (ie you have to pass in the exact geometry to be removed).

The <getArea> and *getLength* functions here merely iterate through the components, summing their respective areas and lengths.

Create a new instance with the *ZOO.Geometry.Collection* constructor.

Inherits from

- *ZOO.Geometry*

Properties

NAME	DESCRIPTION
components	{Array(ZOO.Geometry)} The component parts of this geometry
componentTypes	{Array(String)} An array of class names representing the types of components that the collection can include.

Functions

NAME	DESCRIPTION
ZOO.Geometry.Collection	Creates a Geometry Collection – a list of geoms.
destroy	Destroy this geometry.
clone	Clone this geometry.
getComponentsString	Get a string representing the components for this collection.
calculateBounds	Recalculate the bounds by iterating through the components and calling extendBounds() on each item.
addComponent	Add a new component (geometry) to the collection.
removeComponents	Remove components from this geometry.
removeComponent	Remove a component from this geometry.
getLength	Calculate the length of this geometry.
getCentroid	{ZOO.Geometry.Point} The centroid of the collection.
getGeodesicLength	Calculate the approximate length of the geometry were it projected onto the earth.
move	Moves a geometry by the given displacement along positive x and y axes.
rotate	Rotate a geometry around some origin.
resize	Resize a geometry relative to some origin.
equals	Determine whether another geometry is equivalent to this one.
transform	Reproject the components geometry from source to dest.
intersects	Determine if the input geometry intersects this one.
getVertices	Return a list of all points in this geometry.

Properties

components {Array(*ZOO.Geometry*)} The component parts of this geometry

componentTypes `{Array(String)}` An array of class names representing the types of components that the collection can include. A null value means the component types are not restricted.

Functions

ZOO.Geometry.Collection Creates a Geometry Collection – a list of geoms.

Parameters

`components` {Array(*ZOO.Geometry*)} Optional array of geometries

destroy

```
destroy: function ()
```

Destroy this geometry.

clone

```
clone: function()
```

Clone this geometry.

Returns

{ZOO.Geometry.Collection} An exact clone of this collection.

getComponentsString

```
getComponentsString: function()
```

Get a string representing the components for this collection.

Returns

`{String}` A string representation of the components of this geometry.

calculateBounds

```
calculateBounds: function()
```

Recalculate the bounds by iterating through the components and calling extendBounds() on each item.

addComponent

```
addComponent: function(component,index)
```

Add a new component (geometry) to the collection. If this.componentTypes is set, then the component class name must be in the componentTypes array. The bounds cache is reset.

Parameters

`component` *{ZOO.Geometry}* A geometry to add
`index {int}` Optional index into the array to insert the component

Returns

`{Boolean}` The component geometry was successfully added

removeComponents

```
removeComponents: function(components)
```

Remove components from this geometry.

Parameters

`components` {Array(*ZOO.Geometry*)} The components to be removed

removeComponent

```
removeComponent: function(component)
```

Remove a component from this geometry.

Parameters

`component` *{ZOO.Geometry}*

getLength

```
getLength: function()
```

Calculate the length of this geometry

Returns

`{Float}` The length of the geometry

getCentroid

`getCentroid: function()`

Returns

{ZOO.Geometry.Point} The centroid of the collection

getGeodesicLength

`getGeodesicLength: function(projection)`

Calculate the approximate length of the geometry were it projected onto the earth.

Parameters

`projection` *{ZOO.Projection}* The spatial reference system for the geometry coordinates. If not provided, Geographic/WGS84 is assumed.

Returns

`{Float}` The appoximate geodesic length of the geometry in meters.

move

`move: function(x,y)`

Moves a geometry by the given displacement along positive x and y axes. This modifies the position of the geometry and clears the cached bounds.

Parameters

`x {Float}` Distance to move geometry in positive x direction.
`y {Float}` Distance to move geometry in positive y direction.

rotate

`rotate: function(angle,origin)`

Rotate a geometry around some origin

Parameters

`angle {Float}` Rotation angle in degrees (measured counterclockwise from the positive x-axis)
`origin` *{ZOO.Geometry.Point}* Center point for the rotation

resize

`resize: function(scale,origin,ratio)`

Resize a geometry relative to some origin. Use this method to apply a uniform scaling to a geometry.

Parameters

`scale {Float}` Factor by which to scale the geometry. A scale of 2 doubles the size of the geometry in each dimension (lines, for example, will be twice as long, and polygons will have four times the area).

`origin` *{ZOO.Geometry.Point}* Point of origin for resizing
`ratio` `{Float}` Optional x:y ratio for resizing. Default ratio is 1.

Returns

{ZOO.Geometry} The current geometry.

equals

```
equals: function(geometry)
```

Determine whether another geometry is equivalent to this one. Geometries are considered equivalent if all components have the same coordinates.

Parameters

`geom` *{ZOO.Geometry}* The geometry to test.

Returns

`{Boolean}` The supplied geometry is equivalent to this geometry.

transform

```
transform: function(source,dest)
```

Reproject the components geometry from source to dest.

Parameters

`source` *{ZOO.Projection}*
`dest` *{ZOO.Projection}*

Returns

{ZOO.Geometry}

intersects

```
intersects: function(geometry)
```

Determine if the input geometry intersects this one.

Parameters

`geometry` *{ZOO.Geometry}* Any type of geometry.

Returns

`{Boolean}` The input geometry intersects this one.

getVertices

```
getVertices: function(nodes)
```

Return a list of all points in this geometry.

Parameters

`nodes` `{Boolean}` For lines, only return vertices that are endpoints. If false, for lines, only vertices that are not endpoints will be returned. If not provided, all vertices will be returned.

Returns

`{Array}` A list of all vertices in the geometry.

ZOO.Geometry.Curve

A Curve is a MultiPoint, whose points are assumed to be connected. To this end, we provide a "getLength()" function, which iterates through the points, summing the distances between them.

Inherits from

- *ZOO.Geometry.MultiPoint*

Properties

NAME	DESCRIPTION
componentTypes	{Array(String)} An array of class names representing the types of components that the collection can include.

Functions

NAME	DESCRIPTION
ZOO.Geometry.Curve getLength	{Float} The length of the curve

Properties

componentTypes `{Array(String)}` An array of class names representing the types of components that the collection can include. A null value means the component types are not restricted.

Functions ZOO.Geometry.Curve

Parameters

`point` {Array(*ZOO.Geometry.Point*)}

getLength

`getLength: function()`

Returns

`{Float}` The length of the curve

ZOO.Geometry.LinearRing

A Linear Ring is a special LineString which is closed. It closes itself automatically on every addPoint/removePoint by adding a copy of the first point as the last point.

Also, as it is the first in the line family to close itself, a getArea() function is defined to calculate the enclosed area of the linearRing

Inherits from

- *ZOO.Geometry.LineString*

Properties

NAME	DESCRIPTION
componentTypes	{Array(String)} An array of class names representing the types of components that the collection can include.

Functions

NAME	DESCRIPTION
ZOO.Geometry.LinearRing	Linear rings are constructed with an array of points.
addComponent	Adds a point to geometry components.
move	Moves a geometry by the given displacement along positive x and y axes.
rotate	Rotate a geometry around some origin
resize	Resize a geometry relative to some origin.
transform	Reproject the components geometry from source to dest.
getCentroid	{ZOO.Geometry.Point} The centroid of the ring
getArea	
getGeodesicArea	Calculate the approximate area of the polygon were it projected onto the earth.
containsPoint	Test if a point is inside a linear ring.

Properties

componentTypes `{Array(String)}` An array of class names representing the types of components that the collection can include. A null value means the component types are not restricted.

Functions

ZOO.Geometry.LinearRing Linear rings are constructed with an array of points. This array can represent a closed or open ring. If the ring is open (the last point does not equal the first point), the constructor will close the ring. If the ring is already closed (the last point does equal the first point), it will be left closed.

Parameters

`points` {Array(*ZOO.Geometry.Point*)} points

addComponent

```
addComponent: function(point,index)
```

Adds a point to geometry components. If the point is to be added to the end of the components array and it is the same as the last point already in that array, the duplicate point is not added. This has the effect of closing the ring if it is not already closed, and doing the right thing if it is already closed. This behavior can be overridden by calling the method with a non-null index as the second argument.

Parameter

`point` *{ZOO.Geometry.Point}*
`index` `{Integer}` Index into the array to insert the component

Returns

`{Boolean}` Was the Point successfully added?

move

`move: function(x,y)`

Moves a geometry by the given displacement along positive x and y axes. This modifies the position of the geometry and clears the cached bounds.

Parameters

`x {Float}` Distance to move geometry in positive x direction.
`y {Float}` Distance to move geometry in positive y direction.

rotate

`rotate: function(angle,origin)`

Rotate a geometry around some origin

Parameters

`angle {Float}` Rotation angle in degrees (measured counterclockwise from the positive x-axis)
`origin` *{ZOO.Geometry.Point}* Center point for the rotation

resize

`resize: function(scale,origin,ratio)`

Resize a geometry relative to some origin. Use this method to apply a uniform scaling to a geometry.

Parameters

`scale {Float}` Factor by which to scale the geometry. A scale of 2 doubles the size of the geometry in each dimension (lines, for example, will be twice as long, and polygons will have four times the area).
`origin` *{ZOO.Geometry.Point}* Point of origin for resizing
`ratio {Float}` Optional x:y ratio for resizing. Default ratio is 1.

Returns

`{ZOO.Geometry}` The current geometry.

transform

`transform: function(source,dest)`

Reproject the components geometry from source to dest.

Parameters

`source` *{ZOO.Projection}*
`dest` *{ZOO.Projection}*

Returns

{ZOO.Geometry}

getCentroid

```
getCentroid: function()
```

Returns

{ZOO.Geometry.Point} The centroid of the ring

getArea

```
getArea: function()
```

Note: The area is positive if the ring is oriented CW, otherwise it will be negative.

Returns

`{Float}` The signed area for a ring.

getGeodesicArea

```
getGeodesicArea: function(projection)
```

Calculate the approximate area of the polygon were it projected onto the earth. Note that this area will be positive if ring is oriented clockwise, otherwise it will be negative.

Parameters

`projection` *{ZOO.Projection}* The spatial reference system for the geometry coordinates. If not provided, Geographic/WGS84 is assumed.

Reference

Robert. G. Chamberlain and William H. Duquette, "Some Algorithms for Polygons on a Sphere", JPL Publication 07-03, Jet Propulsion Laboratory, Pasadena, CA, June 2007 http://trs-new.jpl.nasa.gov/dspace/handle/2014/40409

Returns

`{float}` The approximate signed geodesic area of the polygon in square meters.

containsPoint

```
containsPoint: function(point)
```

Test if a point is inside a linear ring. For the case where a point is coincident with a linear ring edge, returns 1. Otherwise, returns boolean.

Parameters

`point` *{ZOO.Geometry.Point}*

Returns

`{Boolean | Number}` The point is inside the linear ring. Returns 1 if the point is coincident with an edge. Returns boolean otherwise.

ZOO.Geometry.LineString

A LineString is a Curve which, once two points have been added to it, can never be less than two points long.

Inherits from

- *ZOO.Geometry.Curve*

Functions

NAME	DESCRIPTION
ZOO.Geometry.LineString	Create a new LineString geometry
removeComponent	Only allows removal of a point if there are three or more points in the linestring.
intersects	Test for instersection between two geometries.
getSortedSegments	{Array} An array of segment objects.
splitWithSegment	Split this geometry with the given segment.
split	Use this geometry (the source) to attempt to split a target geometry.
splitWith	Split this geometry (the target) with the given geometry (the source).
getVertices	Return a list of all points in this geometry.

ZOO.Geometry.LineString Create a new LineString geometry

Parameters

`points` {Array(*ZOO.Geometry.Point*)} An array of points used to generate the linestring

removeComponent

```
removeComponent: function(point)
```

Only allows removal of a point if there are three or more points in the linestring. (otherwise the result would be just a single point)

Parameters

`point` *{ZOO.Geometry.Point}* The point to be removed

intersects

```
intersects: function(geometry)
```

Test for instersection between two geometries. This is a cheapo implementation of the Bently-Ottmann algorigithm. It doesn't really keep track of a sweep line data structure. It is closer to the brute force method, except that segments are sorted and potential intersections are only calculated when bounding boxes intersect.

Parameters

`geometry` *{ZOO.Geometry}*

Returns

`{Boolean}` The input geometry intersects this geometry.

getSortedSegments

```
getSortedSegments: function()
```

Returns

`{Array}` An array of segment objects. Segment objects have properties x1, y1, x2, and y2. The start point is represented by x1 and y1. The end point is represented by x2 and y2. Start and end are ordered so that x1 < x2.

splitWithSegment

`splitWithSegment: function(seg,options)`

Split this geometry with the given segment.

Parameters

`seg {Object}` An object with x1, y1, x2, and y2 properties referencing segment endpoint coordinates.
`options {Object}` Properties of this object will be used to determine how the split is conducted.

Valid options

`edge {Boolean}` Allow splitting when only edges intersect. Default is true. If false, a vertex on the source segment must be within the tolerance distance of the intersection to be considered a split.
`tolerance {Number}` If a non-null value is provided, intersections within the tolerance distance of one of the source segment's endpoints will be assumed to occur at the endpoint.

Returns

`{Object}` An object with lines and points properties. If the given segment intersects this linestring, the lines array will reference geometries that result from the split. The points array will contain all intersection points. Intersection points are sorted along the segment (in order from x1,y1 to x2,y2).

split

`split: function(target,options)`

Use this geometry (the source) to attempt to split a target geometry.

Parameters

`target` *{ZOO.Geometry}* The target geometry.
`options {Object}` Properties of this object will be used to determine how the split is conducted.

Valid options

`mutual {Boolean}` Split the source geometry in addition to the target geometry. Default is false.
`edge {Boolean}` Allow splitting when only edges intersect. Default is true. If false, a vertex on the source must be within the tolerance distance of the intersection to be considered a split.
`tolerance {Number}` If a non-null value is provided, intersections within the tolerance distance of an existing vertex on the source will be assumed to occur at the vertex.

Returns

`{Array}` A list of geometries (of this same type as the target) that result from splitting the target with the source geometry. The source and target geometry will remain unmodified. If no split results, null will be returned. If mutual is true and a split results, return will be an array of two arrays - the first will be all geometries that result

from splitting the source geometry and the second will be all geometries that result from splitting the target geometry.

splitWith

```
splitWith: function(geometry,options)
```

Split this geometry (the target) with the given geometry (the source).

Parameters

`geometry` *{ZOO.Geometry}* A geometry used to split this geometry (the source).
`options {Object}` Properties of this object will be used to determine how the split is conducted.

Valid options

`mutual {Boolean}` Split the source geometry in addition to the target geometry. Default is false.
`edge {Boolean}` Allow splitting when only edges intersect. Default is true. If false, a vertex on the source must be within the tolerance distance of the intersection to be considered a split.
`tolerance {Number}` If a non-null value is provided, intersections within the tolerance distance of an existing vertex on the source will be assumed to occur at the vertex.

Returns

`{Array}` A list of geometries (of this same type as the target) that result from splitting the target with the source geometry. The source and target geometry will remain unmodified. If no split results, null will be returned. If mutual is true and a split results, return will be an array of two arrays - the first will be all geometries that result from splitting the source geometry and the second will be all geometries that result from splitting the target geometry.

getVertices

```
getVertices: function(nodes)
```

Return a list of all points in this geometry.

Parameters

`nodes {Boolean}` For lines, only return vertices that are endpoints. If false, for lines, only vertices that are not endpoints will be returned. If not provided, all vertices will be returned.

Returns

`{Array}` A list of all vertices in the geometry.

ZOO.Geometry.MultiLineString

A MultiLineString is a geometry with multiple ZOO.Geometry.LineString components.

Inherits from

- *ZOO.Geometry.Collection*

Functions

NAME	DESCRIPTION
ZOO.Geometry.MultiLineString	Constructor for a MultiLineString Geometry.

ZOO.Geometry.MultiLineString Constructor for a MultiLineString Geometry.

Parameters

`components` {Array(*ZOO.Geometry.LineString*)}

ZOO.Geometry.MultiPoint

MultiPoint is a collection of Points. Create a new instance with the *ZOO.Geometry.MultiPoint* constructor.

Inherits from

- *ZOO.Geometry.Collection*

Properties

NAME	DESCRIPTION
componentTypes	{Array(String)} An array of class names representing the types of components that the collection can include.

Functions

NAME	DESCRIPTION
ZOO.Geometry.MultiPoint	Create a new MultiPoint Geometry
addPoint	rapper for ZOO.Geometry.Collection.addComponent
removePoint	Wrapper for ZOO.Geometry.Collection.removeComponent

Properties

componentTypes `{Array(String)}` An array of class names representing the types of components that the collection can include. A null value means the component types are not restricted.

Functions

ZOO.Geometry.MultiPoint Create a new MultiPoint Geometry

Parameters

`components` {Array(*ZOO.Geometry.Point*)}

Returns

{ZOO.Geometry.MultPoint}

addPoint

```
addPoint: function(point,index)
```

Wrapper for *ZOO.Geometry.Collection.addComponent*

Parameters

`point` *{ZOO.Geometry.Point}* Point to be added

`index {Integer}` Optional index

removePoint

`removePoint: function(point)`

Wrapper for *ZOO.Geometry.Collection.removeComponent*

Parameters

`point` *{ZOO.Geometry.Point}* Point to be removed

ZOO.Geometry.MultiPolygon

MultiPolygon is a geometry with multiple *ZOO.Geometry.Polygon* components. Create a new instance with the *ZOO.Geometry.MultiPolygon* constructor.

Inherits from

- *ZOO.Geometry.Collection*

Functions

NAME	DESCRIPTION
ZOO.Geometry.MultiPolygon	Create a new MultiPolygon geometry

ZOO.Geometry.MultiPolygon Create a new MultiPolygon geometry

Parameters

`components` {Array(*ZOO.Geometry.Polygon*)} An array of polygons used to generate the MultiPolygon

ZOO.Geometry.Point

Point geometry class.

Inherits from

- *ZOO.Geometry*

Properties

NAME	DESCRIPTION
x	{float}
y	{float}

Functions

NAME	DESCRIPTION
ZOO.Geometry.Point	Construct a point geometry.
clone	{ZOO.Geometry.Point} An exact clone of this ZOO.Geometry.Point
calculateBounds	Create a new Bounds based on the x/y
equals	Determine whether another geometry is equivalent to this one.
toShortString	{String} Shortened String representation of Point object.
move	Moves a geometry by the given displacement along positive x and y axes.
rotate	Rotate a point around another.
getCentroid	{ZOO.Geometry.Point} The centroid of the collection
resize	Resize a point relative to some origin.
intersects	Determine if the input geometry intersects this one.
transform	Translate the x,y properties of the point from source to dest.
getVertices	Return a list of all points in this geometry.

Properties

x `{float}`

y `{float}`

Functions

ZOO.Geometry.Point Construct a point geometry.

Parameters

```
x {float}
y {float}
```

clone

```
clone: function(obj)
```

Returns

{ZOO.Geometry.Point} An exact clone of this ZOO.Geometry.Point

calculateBounds

```
calculateBounds: function ()
```

Create a new Bounds based on the x/y

equals

```
equals: function(geom)
```

Determine whether another geometry is equivalent to this one. Geometries are considered equivalent if all components have the same coordinates.

Parameters

`geom` *{ZOO.Geometry.Point}* The geometry to test.

Returns

`{Boolean}` The supplied geometry is equivalent to this geometry.

toShortString

`toShortString: function()`

Returns

`{String}` Shortened String representation of Point object. (ex. "5, 42")

move

`move: function(x,y)`

Moves a geometry by the given displacement along positive x and y axes. This modifies the position of the geometry and clears the cached bounds.

Parameters

`x {Float}` Distance to move geometry in positive x direction.
`y {Float}` Distance to move geometry in positive y direction.

rotate

`rotate: function(angle,origin)`

Rotate a point around another.

Parameters

`angle {Float}` Rotation angle in degrees (measured counterclockwise from the positive x-axis)
`origin` *{ZOO.Geometry.Point}* Center point for the rotation

getCentroid

`getCentroid: function()`

Returns

{ZOO.Geometry.Point} The centroid of the collection

resize

`resize: function(scale,origin,ratio)`

Resize a point relative to some origin. For points, this has the effect of scaling a vector (from the origin to the point). This method is more useful on geometry collection subclasses.

Parameters

`scale {Float}` Ratio of the new distance from the origin to the old distance from the origin. A scale of 2 doubles the distance between the point and origin.
`origin` *{ZOO.Geometry.Point}* Point of origin for resizing
`ratio {Float}` Optional x:y ratio for resizing. Default ratio is 1.

Returns

{ZOO.Geometry} The current geometry.

intersects

```
intersects: function(geometry)
```

Determine if the input geometry intersects this one.

Parameters

`geometry` *{ZOO.Geometry}* Any type of geometry.

Returns

`{Boolean}` The input geometry intersects this one.

transform

```
transform: function(source,dest)
```

Translate the x,y properties of the point from source to dest.

Parameters

`source` *{ZOO.Projection}*
`dest` *{ZOO.Projection}*

Returns

{ZOO.Geometry}

getVertices

```
getVertices: function(nodes)
```

Return a list of all points in this geometry.

Parameters

`nodes` `{Boolean}` For lines, only return vertices that are endpoints. If false, for lines, only vertices that are not endpoints will be returned. If not provided, all vertices will be returned.

Returns

`{Array}` A list of all vertices in the geometry.

ZOO.Geometry.Polygon

Polygon is a collection of *ZOO.Geometry.LinearRing*.

Inherits from

- *ZOO.Geometry.Collection*

Functions

NAME	DESCRIPTION
ZOO.Geometry.Polygon	Constructor for a Polygon geometry.
getArea	Calculated by subtracting the areas of the internal holes from the area of the outer hole.
containsPoint	Test if a point is inside a polygon.
createRegularPolygon	Create a regular polygon around a radius.

ZOO.Geometry.Polygon Constructor for a Polygon geometry. The first ring (*this.component[0]*) is the outer bounds of the polygon and all subsequent rings (*this.component[1-n]*) are internal holes.

Parameters

`components` {Array(*ZOO.Geometry.LinearRing*)}

getArea

```
getArea: function()
```

Calculated by subtracting the areas of the internal holes from the area of the outer hole.

Returns

`{float}` The area of the geometry

containsPoint

```
containsPoint: function(point)
```

Test if a point is inside a polygon. Points on a polygon edge are considered inside.

Parameters

`point` *{ZOO.Geometry.Point}*

Returns

`{Boolean | Number}` The point is inside the polygon. Returns 1 if the point is on an edge. Returns boolean otherwise.

createRegularPolygon

```
ZOO.Geometry.Polygon.createRegularPolygon = function( origin,
                                                      radius,
                                                      sides,
                                                      rotation      )
```

Create a regular polygon around a radius. Useful for creating circles and the like.

Parameters

`origin` *{ZOO.Geometry.Point}* center of polygon.
`radius` `{Float}` distance to vertex, in map units.
`sides` `{Integer}` Number of sides. 20 approximates a circle.
`rotation` `{Float}` original angle of rotation, in degrees.

ZOO.Geometry.Surface

Surface geometry class.

Inherits from

- *ZOO.Geometry*

ZOO.Process

Used to query OGC WPS process defined by its URL and its identifier. Usefull for chaining localhost process.

Properties and Functions

NAME	DESCRIPTION
schemaLocation	{String} Schema location for a particular minor version.
namespaces	{Object} Mapping of namespace aliases to namespace URIs.
url	{String} The OGC's Web PRocessing Service URL, default is http://localhost/zoo.
identifier	{String} Process identifier in the OGC's Web Processing Service.
ZOO.Process	Create a new Process
Execute	Query the OGC's Web PRocessing Servcie to Execute the process.
buildInput	Object containing methods to build WPS inputs.
buildInput.complex	Given an E4XElement representing the WPS complex data input.
buildInput.reference	Given an E4XElement representing the WPS reference input.
buildInput.literal	Given an E4XElement representing the WPS literal data input.
buildDataInputsNode	Method to build the WPS DataInputs element.

schemaLocation `{String}` Schema location for a particular minor version.

namespaces `{Object}` Mapping of namespace aliases to namespace URIs.

url `{String}` The OGC's Web PRocessing Service URL, default is http://localhost/zoo.

identifier `{String}` Process identifier in the OGC's Web Processing Service.

ZOO.Process Create a new Process

Parameters

`url {String}` The OGC's Web Processing Service URL.
`identifier {String}` The process identifier in the OGC's Web Processing Service.

Execute

```
Execute: function(inputs)
```

Query the OGC's Web PRocessing Servcie to Execute the process.

Parameters

`inputs {Object}`

Returns

`{String}` The OGC's Web processing Service XML response. The result needs to be interpreted.

buildInput Object containing methods to build WPS inputs.

buildInput.complex Given an E4XElement representing the WPS complex data input.

Parameters

`identifier {String}` the input indetifier
`data {Object}` A WPS complex data input.

Returns

`{E4XElement}` A WPS Input node.

buildInput.reference Given an E4XElement representing the WPS reference input.

Parameters

`identifier {String}` the input indetifier
`data {Object}` A WPS reference input.

Returns

`{E4XElement}` A WPS Input node.

buildInput.literal Given an E4XElement representing the WPS literal data input.

Parameters

`identifier {String}` the input indetifier
`data {Object}` A WPS literal data input.

Returns

`{E4XElement}` The WPS Input node.

buildDataInputsNode

`buildDataInputsNode:function(inputs)`

Method to build the WPS DataInputs element.

Parameters

`inputs {Object}`

Returns

`{E4XElement}` The WPS DataInputs node for Execute query.

ZOO.Projection

Class for coordinate transforms between coordinate systems.

Properties

NAME	DESCRIPTION
proj	{Number} Proj4js.Proj instance.
projCode	{String}

Functions

NAME	DESCRIPTION
ZOO.Projection	This class offers several methods for interacting with a wrapped zoo-pro4js projection object.
getCode	Get the string SRS code.
getUnits	Get the units string for the projection – returns null if zoo-proj4js is not available.
toString	Convert projection to string (getCode wrapper).
equals	Test equality of two projection instances.
destroy	Destroy projection object.
transform	Transform a point coordinate from one projection to another.

Properties

proj `{Object}` Proj4js.Proj instance.

projCode `{String}`

Functions

ZOO.Projection This class offers several methods for interacting with a wrapped zoo-pro4js projection object.

Parameters

`projCode {String}` A string identifying the Well Known Identifier for the projection.
`options {Object}` An optional object to set additional properties.

Returns

{ZOO.Projection} A projection object.

getCode

```
getCode: function()
```

Get the string SRS code.

Returns

`{String}` The SRS code.

getUnits

```
getUnits: function()
```

Get the units string for the projection – returns null if zoo-proj4js is not available.

Returns

`{String}` The units abbreviation.

toString

```
toString: function()
```

Convert projection to string (getCode wrapper).

Returns

`{String}` The projection code.

equals

`equals: function(projection)`

Test equality of two projection instances. Determines equality based soley on the projection code.

Returns

`{Boolean}` The two projections are equivalent.

destroy

`destroy: function()`

Destroy projection object.

transform

`ZOO.Projection.transform = function(point,source,dest)`

Transform a point coordinate from one projection to another. Note that the input point is transformed in place.

Parameters

`point {{ZOO.Geometry.Point> | Object}` An object with x and y properties representing coordinates in those dimensions.
`sourceProj {ZOO.Projection}` Source map coordinate system
`destProj {ZOO.Projection}` Destination map coordinate system

Returns

`point {object}` A transformed coordinate. The original point is modified.

ZOO.Request

Contains convenience methods for working with ZOORequest which replace XMLHttpRequest.

Functions

NAME	DESCRIPTION
GET	Send an HTTP GET request.
POST	Send an HTTP POST request.

GET Send an HTTP GET request.

Parameters

`url {String}` The URL to request.
`params {Object}` Params to add to the url

Returns

`{String}` Request result.

POST Send an HTTP POST request.

Parameters

`url {String}` The URL to request.
`body {String}` The request's body to send.
`headers {Object}` A key-value object of headers to push to the request's head

Returns

`{String}` Request result.

ZOO.String

Contains convenience methods for string manipulation:

Functions and Properties

NAME	DESCRIPTION
startsWith	Test whether a string starts with another string.
contains	Test whether a string contains another string.
trim	Removes leading and trailing whitespace characters from a string.
camelize	Camel-case a hyphenated string.
tokenRegEx	Used to find tokens in a string.
numberRegEx	Used to test strings as numbers.
isNumeric	Determine whether a string contains only a numeric value.
numericIf	Converts a string that appears to be a numeric value into a number.

startsWith

```
startsWith: function(str,sub)
```

Test whether a string starts with another string.

Parameters

`str {String}` The string to test.
`sub {Sring}` The substring to look for.

Returns

`{Boolean}` The first string starts with the second.

contains

```
contains: function(str,sub)
```

Test whether a string contains another string.

Parameters

`str {String}` The string to test.

`sub {String}` The substring to look for.

Returns

`{Boolean}` The first string contains the second.

trim

```
trim: function(str)
```

Removes leading and trailing whitespace characters from a string.

Parameters

`str {String}` The (potentially) space-padded string. This string is not modified.

Returns

`{String}` A trimmed version of the string with all leading and trailing spaces removed.

camelize

```
camelize: function(str)
```

Camel-case a hyphenated string. Ex. "chicken-head" becomes "chickenHead", and "-chicken-head" becomes "ChickenHead".

Parameters

`str {String}` The string to be camelized. The original is not modified.

Returns

`{String}` The string, camelized

tokenRegEx Used to find tokens in a string. Examples: *${a}, ${a.b.c}, ${a-b}, ${5}*

numberRegEx Used to test strings as numbers.

isNumeric

```
isNumeric: function(value)
```

Determine whether a string contains only a numeric value.

Examples

```
ZOO.String.isNumeric("6.02e23") // true
ZOO.String.isNumeric("12 dozen") // false
ZOO.String.isNumeric("4") // true
ZOO.String.isNumeric(" 4 ") // false
```

Returns

`{Boolean}` String contains only a number.

numericIf

```
numericIf: function(value)
```

Converts a string that appears to be a numeric value into a number.

Returns

`{Number|String}` a Number if the passed value is a number, a String otherwise.

4.3.3 Examples

In this page you can find some small examples on how to use the JavaScript ZOO-API on the server side.

ZOO-API contains many classes and functions. You can find the description list *here*.

ZOO.Process example of use

```
function SampleService(conf,inputs,outputs){
   var myProcess = new ZOO.Process('http://localhost/cgi-bin-new1/zoo_loader_new1.cgi','Boundary');
   var myInputs = {InputPolygon: { type: 'complex', value: '{"type":"Polygon","coordinates":[[[-106.9
   var myExecuteResult=myProcess.Execute(myInputs);
   return {result: ZOO.SERVICE_SUCCEEDED, outputs: [ {name:"Result", value: myExecuteResult} ] };
}
```

In this really short example you can see how to create `ZOO.Process` class instance and call the Execute method on such an instance. Then you'll just need to return a JavaScript object containing the attributes result and outputs, which I'm sure you already know what is about. The first is about the status of the process (can be `ZOO.SERVICE_SUCEEDED`, `ZOO.SERVICE_FAILED` and so on), the last is obviously the resulting maps (take a look at the maps internal data structure used by ZOO Kernel in `service.h`).

ZOO.UpdateStatus

```
function SampleLongService(conf,inputs,outputs){
   var my_i=0;
   while(my_i<100){
       try{
         conf["lenv"]["status"]=my_i;
       }
       catch(e){
       }
       ZOOUpdateStatus(conf,my_i);
       SampleService(conf,inputs,outputs);
       my_i+=10;
   }
   return SampleService(conf,inputs,outputs);
}
```

You can see in this sample code how to use the `ZOOUpdateStatus` function to update the current status of your running process. This information will be really helpfull when the ZOO Kernel will run your JavaScript Service in background mode (if the user set to `true` the `storeExecuteResponse` parameter in his request).

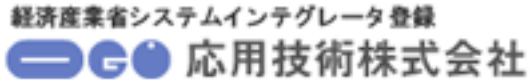
4.4 Workshop: Practical Introduction to ZOO: The Open WPS Platform

Author Nicolas Bozon, Gérald Fenoy

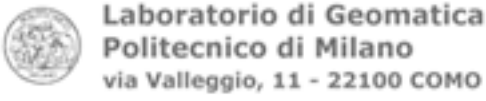
Last Updated $Date: 2011-12-07 14:19:47 +0100 (Mer, 07 déc 2011) $

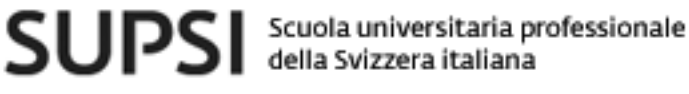
4.4.1 FOSS4G 2010 Osaka / Tokyo / Beijing

4.4.2 Sponsored By



4.4.3 Special Thanks To Our Knowledge Partners





4.4.4 Workshop Table of Contents

Introduction

Table of Contents

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 service 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 one and from an already existing code. Further information on the project is available on the ZOO Project official website .

How does ZOO works ?

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, Perl, 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.

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 OGC 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 sack 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 was providen by Orkney and included in the OSGeoLiveDVD, data are available trough OGC WMS/WFS WebServices using MapServer and will be displayed on a simple map 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 :

```
this is a code block
```

Warning: This is a warning message.

Note: This is an important note.

Let's go !

Using ZOO from an OSGeoLive virtual machine

Table of Contents

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.

ZOO Kernel Installation

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 compiling C Services and running Python ones 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 file if you need them for a specific Service. Nevertheless, you have to know that the env and lenv sections name are used in a specific way.

The env section 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, then you will have to set the DISPLAY environnement variably, in this case you would add DISPLAY=:1 line in your env section.

As for the env section, there is the section lenv where specific informations about status informations of a running Service will be written by the ZOO Kernel or the ZOO Services. For instance, when your service failed, you can set the value for message in lenv to see it displayed in the Status node of the ExecuteResponse returned back to the client. If your service will take long time and can get informations about processing status, you can then set a value between 0 and 100 to status in lenv to represent the percentage completed of the running Service task, we will talk deeper about this later in this workshop.

Please take a look to your local file main.cfg 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 DocumentRoot. 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 call/(.*)/(.*) /cgi-bin/zoo_loader.cgi?request=Execute&service=WPS&version=1.0.0&Identif
```

```
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/
```

Or using the `a2enmod` tool this way :

```
a2enmod rewrite
```

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 form the OSGeoLive environment will be used during this workshop. MapServer will first be used to provide WFS input data for the ZOO Services we are going to develop. The MapServer dataset providen by Orkney (japanese regions polygons) will be passed to our service during section 4.

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.

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

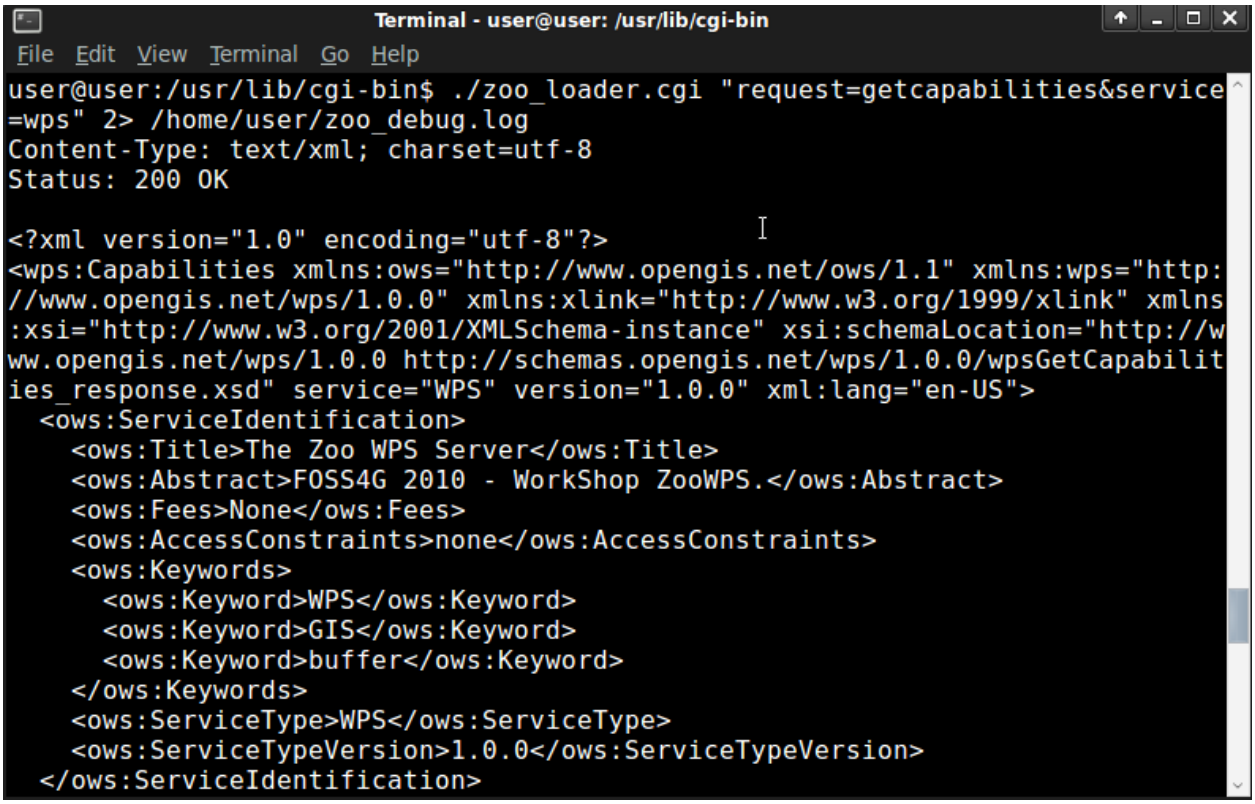
You should then get a valid Capabilities XML document, as the following :

```
−<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:

```
cd /usr/lib/cgi-bin
./zoo_loader.cgi "request=GetCapabilities&service=WPS"
```

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



Invoking ZOO Kernel from command line can be helpful during development process of new Services.

Preparing your ZOO Services Provider 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.

Creating OGR based Web Services

Table of Contents

Introduction

In this part, we are going to create a ZOO ServicesProvider 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, codded 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.

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 ServicesProvider 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 ServicesProvider 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>
      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 LitteralData 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>
   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 :

```
−<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>
```

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>
       Python module ogr_ws_service_provider cannot be loaded.
     </ows:ExceptionText>
   </ows:Exception>
 </ows:ExceptionReport>
```

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

```
−<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>
```

Implementing single geometry services

In order to learn the Services Provider creation and deployement 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

Boundary

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 map 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 setMapInMaps function once again defined in zoo-kernel/service.h. The setMapInMaps function takes four parameters :

- `m` : a maps pointer you want to update,
- `ns` : the name of the maps you want 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 values of some map in the Result maps from outputs :

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

Please note that the setMapInMaps 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.

Even if you will mainly use map from maps during this workshop, you can also add or update values in a map directly using the addToMap function defined in zoo-kernel/service.h. The addToMap function take three paramters :

- `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 in this map.

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 the correspoding maps datatype used in Python language (this is a summarized version of the main configaration 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 MapServer 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->nodesetval){
    errorException(conf, "Unable to parse Input Polygon","InvalidParameterValue");
    exit(0);
  }
  int size = (xpathObj->nodesetval) ? xpathObj->nodesetval->nodeNr : 0;
  xmlDocPtr ndoc = xmlNewDoc(BAD_CAST "1.0");
  for(int k=size-1;k>=0;k--){
    xmlDocSetRootElement(ndoc, xpathObj->nodesetval->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){
    errorException(conf, "Unable to call OGR_G_CreatFromGML","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 occured.

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){
    setMapInMaps(m,"lenv","message","Unable to parse InputPolygon");
    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);
  if(geometry==NULL){
    setMapInMaps(m,"lenv","message","Unable to parse InputPolygon");
    return SERVICE_FAILED;
  }
  res=OGR_G_GetBoundary(geometry);
  tmp1=getMapFromMaps(outputs,"Result","mimeType");
  if(strncmp(tmp1->value,"application/json",16)==0){
    char *tmp=OGR_G_ExportToJson(res);
    setMapInMaps(outputs,"Result","value",tmp);
    setMapInMaps(outputs,"Result","mimeType","text/plain");
    free(tmp);
  }
  else{
    char *tmp=OGR_G_ExportToGML(res);
    setMapInMaps(outputs,"Result","value",tmp);
    free(tmp);
  }
  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=createGeometryFromWFS(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 `createGeometryFromWFS` 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){
  char *tmp=OGR_G_ExportToJson(res);
  setMapInMaps(outputs,"Result","value",tmp);
  setMapInMaps(outputs,"Result","mimeType","text/plain");
  free(tmp);
}
else{
  char *tmp=OGR_G_ExportToGML(res);
  setMapInMaps(outputs,"Result","value",tmp);
  free(tmp);
}
```

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, setMapInMaps 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/!ServicesProvider.zo ./service.c $LDFLAGS
```

Please note that the CFLAGS 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 --clfags` -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` environment 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 ServicesProvider, 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 meannings 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.

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 bellow. Indeed, as Python is an interpreted language, you do not have to compile anything before deploying your service which makes the deployement 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
    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 dicuss the functions body here as we already gave all the details before and the code was volontary 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.

Testing the Service using Execute Request 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:

link

```
http://localhost/cgi-bin/zoo_loader.cgi?request=Execute&service=WPS&version=1.0.0&Identifier=Boundary
```

As you can see in the url above, we use an URLEncoded WFS request to the MapServer 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/cgi-bin/mapserv?map=/var/www/wfs.map&SERVICE=WFS&REQUEST=GetFeature&VERSION=1.0.0&ty
```

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 would look like in case `storeExecuteResponse` was set to true in the request:

```
−<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 you 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 :

link

```
http://localhost/cgi-bin/zoo_loader.cgi?request=Execute&service=WPS&version=1.0.0&Identifier=Boundary
```

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 :

link

```
http://localhost/cgi-bin/zoo_loader.cgi?request=Execute&service=WPS&version=1.0.0&Identifier=Boundary
```

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.

Now, you know how to ask ZOO Kernel to run service in background, ask for `RawDataOutput` specifying `mimeType` or any specific format to be returned by the Kernel. When you ask for `ResponseDocument`, you can also specify to the ZOO Kernel that you want the result to be stored on the server side.

To do such a thing, you have to set the attribute `asReference` as true and then the resulting ExecuteResponse will contain a Reference node including the href attribute to let you access the produced file. To be able to handle this, you have to add the extension parameter in your `DataOutputs` node in the corresponding ZCFG file.

Here is a sample url which provide such a result:

link

```
http://localhost/cgi-bin/zoo_loader.cgi?request=Execute&service=WPS&version=1.0.0&Identifier=Boundary
```

You can see bellow what kind of result can be expected :

```
- <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-US" serviceInstance="http://localhost/zoo/">
  - <wps:Process wps:processVersion="2">
      <ows:Identifier>Buffer</ows:Identifier>
      <ows:Title>Create a buffer around a polygon. </ows:Title>
    - <ows:Abstract>
        Create a buffer around a single polygon. Accepts the polygon as GML and provides GML output for the buffered feature.
      </ows:Abstract>
    </wps:Process>
  - <wps:Status creationTime="2010-11-29T11:19:24Z">
      <wps:ProcessSucceeded>Service "Buffer" run successfully.</wps:ProcessSucceeded>
    </wps:Status>
  - <wps:ProcessOutputs>
    - <wps:Output>
        <ows:Identifier>Result</ows:Identifier>
        <ows:Title>Buffered Polygon</ows:Title>
      - <ows:Abstract>
          GML stream describing the buffered polygon feature.
        </ows:Abstract>
        <wps:Reference href="http://localhost/zoo/../temp/Buffer_102765.js" mimeType="text/plain" encoding="UTF-8"/>
      </wps:Output>
    </wps:ProcessOutputs>
  </wps:ExecuteResponse>
```

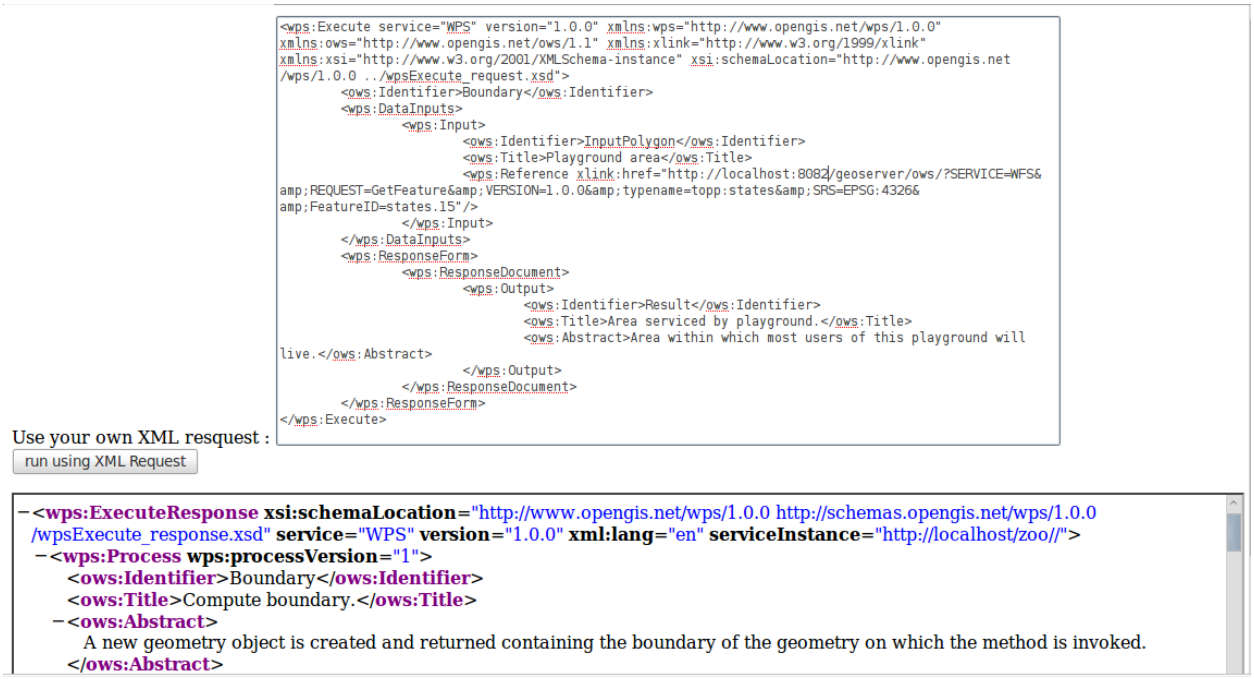
Simplification and readability of request

As you can see in the simple example we used since the begining of this workshop, it is sometimes hard to write the Execute requests using the GET method as it makes really long and complexe 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="ht
 <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/cgi-bin/mapserv?map=/var/www/wfs.map&amp;SERVICE=WFS&a
  </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:



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="ht
 <ows:Identifier>Boundary</ows:Identifier>
 <wps:DataInputs>
  <wps:Input>
   <ows:Identifier>InputPolygon</ows:Identifier>
   <wps:Data>
    <wps:ComplexData mimeType="application/json">
{ "type": "MultiPolygon", "coordinates": [ [ [ [ -105.998360, 31.393818 ], [ -106.212753, 31.478128 ]
    </wps: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.

storeExecuteResponse parameter and GetStatus Service

If you go in your local `/home/user/zoows/sources/zoo-services/utils/status`, you'll find the code for a ServiceProvider which will provide the GetStatus service and the longProcess one. The last is a simple example

to learn how to use the status variable from lenv section of the main configuration maps and the updateStatus function you have to call to take your status value into account. The main service provider is the GetStatus one, it is able to give you information about the current status value from a service running in background mode.

You have to know that the ZOO Kernel will detect the presence of the GetStatus service and if it is available it will then return the link the corresponding Execute request.

So now you will deploy the GetStatus and longProcess service on your local environment. As for each services, you shall be able to deploy the services simply by copying the cgi-env directory into your Apache `cgi-bin` directory. You can use the following command :

```
sudo cp ~user/zoows/sources/zoo-services/utils/status/cgi-env/*{zcfg,zo} /usr/lib/cgi-bin
```

For simple Services it is the right way to deploy Service Providers. But in this specific case you'll have also to add some special parameter in the main section of you main configuration file and to copy an xsl file used to replace on the fly in the ResponseDocument the percentCompleted attribute of the ProcessStarted node returned by the GetStatus service.

So first edit you `main.cfg` file to add the following lines in your main section :

```
rewriteUrl=call
dataPath=/var/www/data
```

Here you define the path where the service is able to find the xsl file, specified in the dataPath parameter. You also tell the ZOO Kernel that you want to use the rewriteUrl we defined in the previous section.

To finish your deployment, you'll have now to copy the xsl file in the defined dataPath directory. You can use the following command :

```
cp ~/zoows/sources/zoo-services/utils/status/cgi-env/*xsl /var/www/data
```

Now, if you run the following request to run the service longProcess :

http://localhost/zoo/?request=Execute&service=WPS&version=1.0.0&Identifier=longProcess&DataInputs=&storeExecuteResponse=tr

You shall get the a XML document looking like the following:

```
- <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-US" serviceInstance="http://localhost/zoo/" statusLocation="http://localhost/zoo/call/GetStatus/4432">
  - <wps:Process wps:processVersion="1">
      <ows:Identifier>longProcess</ows:Identifier>
      <ows:Title>Demo long process. </ows:Title>
    - <ows:Abstract>
        This service doesn't do anything except taking its time, it demonstrates how to use the updateStatus function from your ZOO Service.
      </ows:Abstract>
    </wps:Process>
  - <wps:Status creationTime="2010-10-30T11:16:59Z">
    - <wps:ProcessAccepted>
        Service "longProcess" was accepted by the ZOO Kernel and it run as a background task. Please consult the statusLocation attribtue providen in this document to get the
        up-to-date document.
      </wps:ProcessAccepted>
    </wps:Status>
  </wps:ExecuteResponse>
```

If you poll the statusLocation url provider in the answer you'll then be able to view the evolution of the percentCompleted attribute value growing, like you can see in the following screenshot.

```
- <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-US" serviceInstance="http://localhost/zoo/" statusLocation="http://localhost/zoo/call/GetStatus/4432">
  - <wps:Process wps:processVersion="1">
      <ows:Identifier>longProcess</ows:Identifier>
      <ows:Title>Demo long process. </ows:Title>
    - <ows:Abstract>
        This service doesn't do anything except taking its time, it demonstrates how to use the updateStatus function from your ZOO Service.
      </ows:Abstract>
    </wps:Process>
  - <wps:Status creationTime="2010-10-30T11:16:59Z">
    - <wps:ProcessStarted percentCompleted="65">
        ZOO Service "longProcess" is currently running. Please, reload this document to get the up-to-date status of the Service.
      </wps:ProcessStarted>
    </wps:Status>
  </wps:ExecuteResponse>
```

This won't be used during this workshop but can be useful for really time consuming services.

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.

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){
    setMapInMaps(conf,"lenv","message","Unable to fetch InputPolygon value.");
    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);
  if(geometry==NULL){
    setMapInMaps(conf,"lenv","message","Unable to parse InputPolygon value.");
    return SERVICE_FAILED;
  }
  res=OGR_G_ConvexHull(geometry);
  tmp1=getMapFromMaps(outputs,"Result","mimeType");
  if(strncmp(tmp1->value,"application/json",16)==0){
    char* tmp=OGR_G_ExportToJson(res);
    setMapInMaps(outputs,"Result","value",tmp);
    setMapInMaps(outputs,"Result","mimeType","text/plain");
    free(tmp);
  }
  else{
    char* tmp=OGR_G_ExportToGML(res);
    setMapInMaps(outputs,"Result","value",tmp);
    free(tmp);
  }
  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){
    setMapInMaps(conf,"lenv","message","Unable to fetch InputPolygon value.");
    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);
  if(geometry==NULL){
    setMapInMaps(conf,"lenv","message","Unable to parse InputPolygon value.");
    return SERVICE_FAILED;
  }
  res=(*myFunc)(geometry);
  tmp1=getMapFromMaps(outputs,"Result","mimeType");
  if(strncmp(tmp1->value,"application/json",16)==0){
    char *tmp=OGR_G_ExportToJson(res);
    setMapInMaps(outputs,"Result","value",tmp);
    setMapInMaps(outputs,"Result","mimeType","text/plain");
    free(tmp);
  }
  else{
    char *tmp=OGR_G_ExportToGML(res);
    setMapInMaps(outputs,"Result","value",tmp);
    free(tmp);
  }
  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_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.

Python Version

```
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 automaticaly. 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 simplier 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.

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.

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 whenthe OGR_G_Buffer must be called (instead of OGR_G_GetBoundary). Here is the ful lcode :

```
int Buffer(maps*& conf,maps*& inputs,maps*& outputs){
  OGRGeometryH geometry,res;
  map* tmp1=getMapFromMaps(inputs,"InputPolygon","value");
  if(tmp==NULL){
    setMapInMaps(conf,"lenv","message","Unable to fetch InputPolygon value.");
    return SERVICE_FAILED;
  }
  map* tmp1=getMapFromMaps(inputs,"InputPolygon","mimeType");
  if(strncmp(tmp->value,"application/json",16)==0)
```

```
    geometry=OGR_G_CreateGeometryFromJson(tmp->value);
  else
    geometry=createGeometryFromWFS(conf,tmp->value);
  double bufferDistance=1;
  tmp=getMapFromMaps(inputs,"BufferDistance","value");
  if(tmp!=NULL)
    bufferDistance=atof(tmp->value);
  res=OGR_G_Buffer(geometry,bufferDistance,30);
  tmp1=getMapFromMaps(outputs,"Result","mimeType");
  if(strncmp(tmp1->value,"application/json",16)==0){
    char *tmp=OGR_G_ExportToJson(res);
    setMapInMaps(outputs,"Result","value",tmp);
    setMapInMaps(outputs,"Result","mimeType","text/plain");
    free(tmp);
  }
  else{
    char *tmp=OGR_G_ExportToGML(res);
    setMapInMaps(outputs,"Result","value",tmp);
    free(tmp);
  }
  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%2Fcgi-bin%2Fmapserv%3FSERVICE%3DW
```

Can now be rewritten this way :

```
DataInputs=InputPolygon=Reference@xlink:href=http%3A%2F%2Flocalhost%2Fcgi-bin%2Fmapserv%3FSERVICE%3DW
```

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="ht
 <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/cgi-bin/mapserv?map=/var/www/wfs.map&amp;SERVICE=WFS&a
   </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>
```

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 detailled in the next section.

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 orginal `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.

Building a WPS client using OpenLayers

Table of Contents

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.

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 `/var/www/zoo-ogr.html` using your favorite text editor and paste the following HTML snippet:

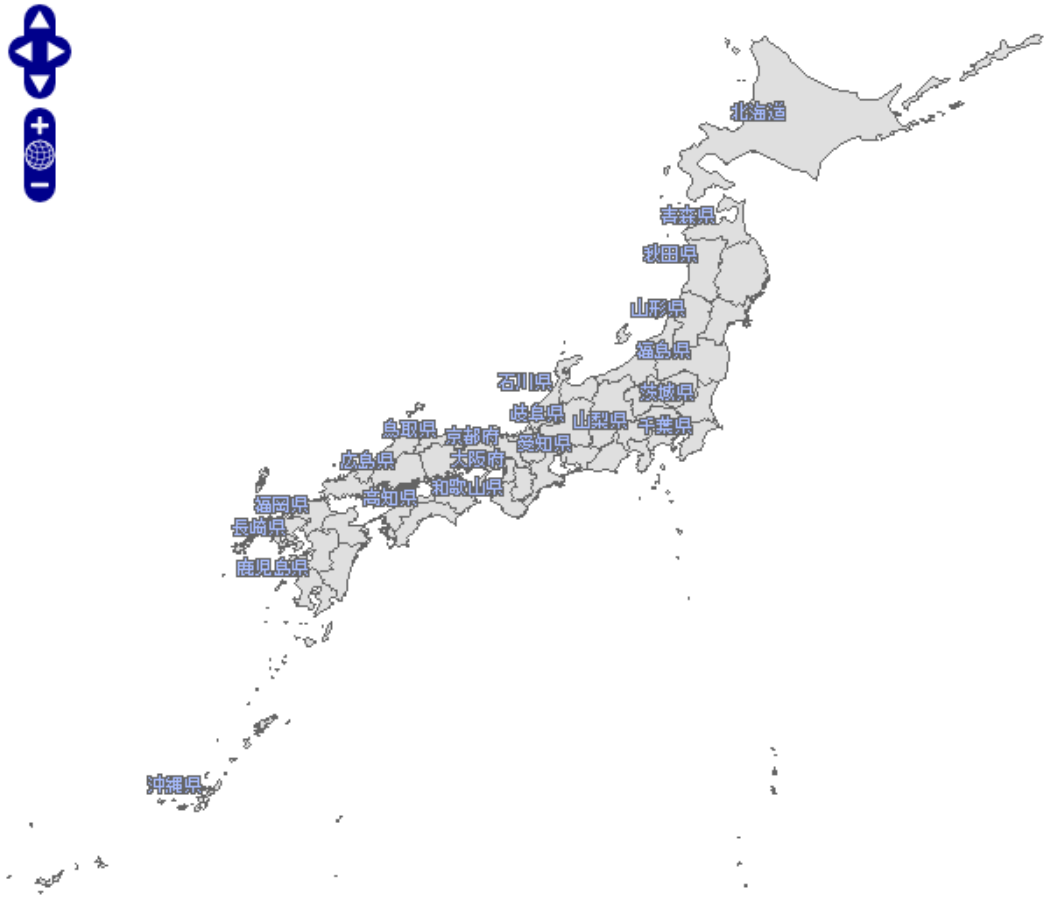
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-str
 <head>
  <meta content="text/html; charset=UTF-8" http-equiv="content-type"/>
  <title>ZOO WPS Client example</title>
  <style>
  #map{width:700px;height:600px;}
  </style>
  <link rel="stylesheet" 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 japanese regions data as WMS.

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

var typename="regions";
var main_url="http://localhost/cgi-bin/mapserv?map=/var/www/wfs.map";
```

```
function init(){
  map = new OpenLayers.Map('map', {
    controls: [
      new OpenLayers.Control.PanZoom(),
      new OpenLayers.Control.Permalink(),
      new OpenLayers.Control.Navigation()
    ]
  });
  layer = new OpenLayers.Layer.WMS(typename,main_url, {
        layers: 'regions',
        transparent: 'true',
        format: 'image/png'
      },
      {
        isBaseLayer: true,
        visibility: true,
        buffer: 0,
        singleTile: true
      }
);
  map.addLayers([layer]);
  map.setCenter(new OpenLayers.LonLat(138,33.5),5);
}
```

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 Japan with the WMS layer activated.



Fetching the data layer as WFS and adding selection controls

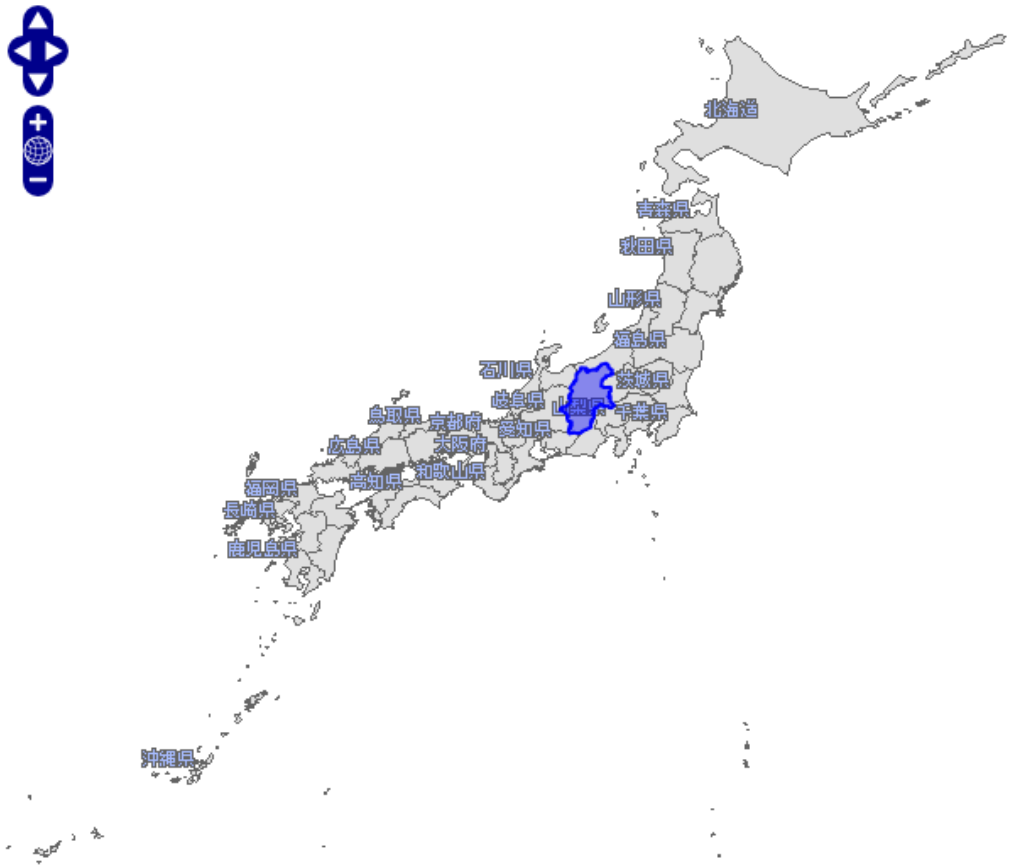
Before accessing the displayed data via WFS, you first have to create new vector layers dedicated to host the several interactions we are going to create. Please add the following lines within the `init()` function, and do not forget to add the newly created layer in the map.addLayers method:

```
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 the data by creating new controls to select polygons, as the following. Please note that `OpenLayers.Protocol.WFS.fromWMSLayer` is used to access geometries and that several state of selection are declared and append to the control variable.

```
var protocol = OpenLayers.Protocol.WFS.fromWMSLayer(layer, {
        featurePrefix: 'ms',
        geometryName: 'msGeometry',
        featureType: typename
});

control = new OpenLayers.Control.GetFeature({
        protocol: protocol,
        box: false,
        hover: false,
        multipleKey: "shiftKey",
        toggleKey: "ctrlKey"
});
control.events.register("featureselected", this, function(e) {
                select.addFeatures([e.feature]);
});
control.events.register("featureunselected", this, function(e) {
                select.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.



Calling the single geometry services 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. You can notice that we use a specific `parseMapServerId` function which let us remove the unecessary white space returned as fid value.

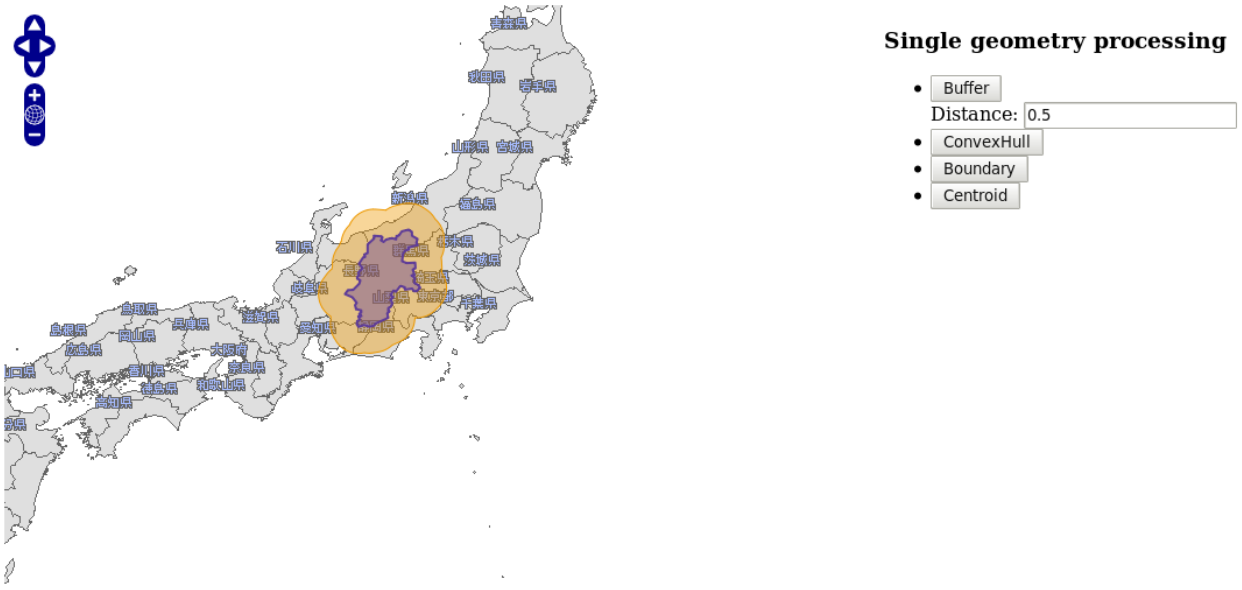
```
function parseMapServerId(){
  var sf=arguments[0].split(".");
  return sf[0]+"."+sf[1].replace(/ /g,'');
}

function simpleProcessing(aProcess) {
  if (select.features.length == 0)
    return alert("No feature selected!");
  if(multi.features.length>=1)
    multi.removeFeatures(multi.features);
  var url = '/cgi-bin/zoo_loader.cgi?request=Execute&service=WPS&version=1.0.0&';
  if (aProcess == 'Buffer') {
    var dist = document.getElementById('bufferDist')?document.getElementById('bufferDist').value:'1';
    if (isNaN(dist)) return alert("Distance is not a Number!");
    url+='Identifier=Buffer&DataInputs=BufferDistance='+dist+'@datatype=interger;InputPolygon=Referer
  } 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='+parseMapServerId(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>`.

```
<div style="float: right;padding-left: 5px;">
 <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>
</div>
```

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.



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 = '/cgi-bin/zoo_loader.cgi';
  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='+parseMapServerId(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
    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].geomet
    params += '</wps:Data>';
    params += '</wps:Input>';
    params += '</wps:DataInputs>';
    params += '<wps:ResponseForm>';
    params += '<wps:RawDataOutput>';
    params += '<ows:Identifier>Result</ows:Identifier>';
    params += '</wps:RawDataOutput>';
```

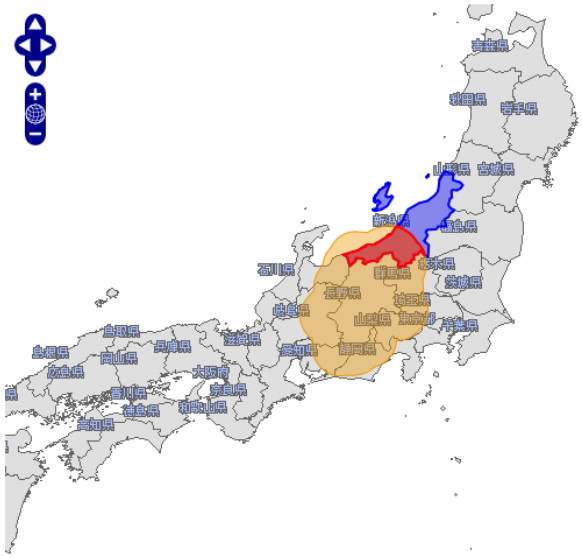
```
    params += '</wps:ResponseForm>';
    params += '</wps:Execute>';
  } catch(e) {
    alert(e);
    return false;
  }
  var request = new OpenLayers.Request.XMLHttpRequest();
  request.open('POST',url,true);
  request.setRequestHeader('Content-Type','text/xml');
  request.onreadystatechange = function() {
    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.value);" value="Union"/>
  </li>
  <li>
    <input type="button" onclick="multiProcessing(this.value);" value="Difference"/>
  </li>
  <li>
    <input type="button" onclick="multiProcessing(this.value);" value="SymDifference"/>
  </li>
  <li>
    <input type="button" onclick="multiProcessing(this.value);" 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 :

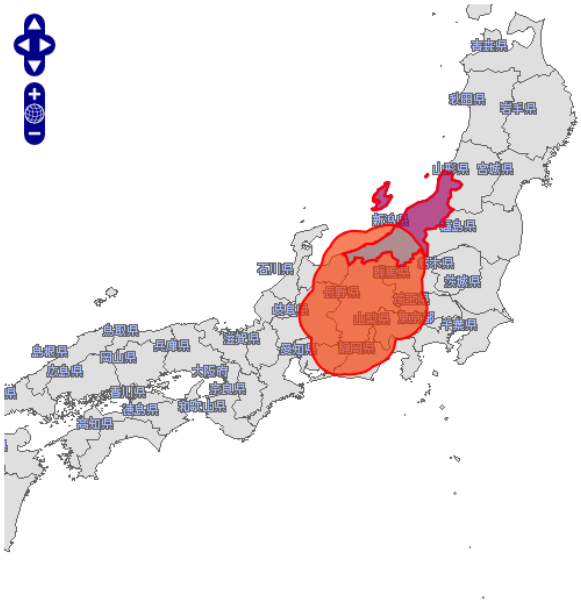


Single geometry processing

- Buffer
  Distance: 0.5
- ConvexHull
- Boundary
- Centroid

Multiple geometries processing

- Union
- Difference
- SymDifference
- Intersection

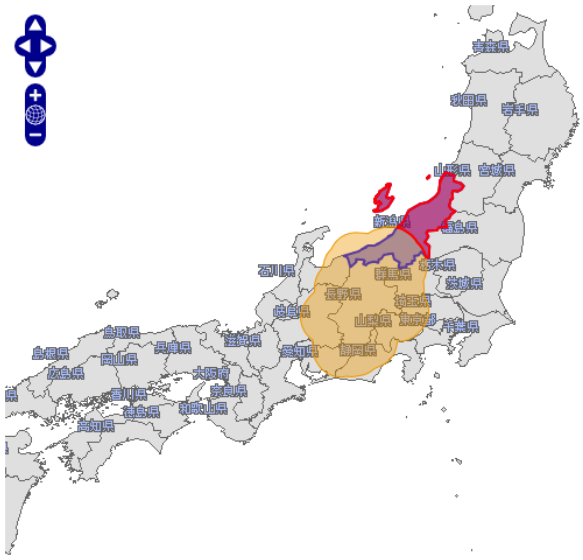


Single geometry processing

- Buffer
  Distance: 0.5
- ConvexHull
- Boundary
- Centroid

Multiple geometries processing

- Union
- Difference
- SymDifference
- Intersection

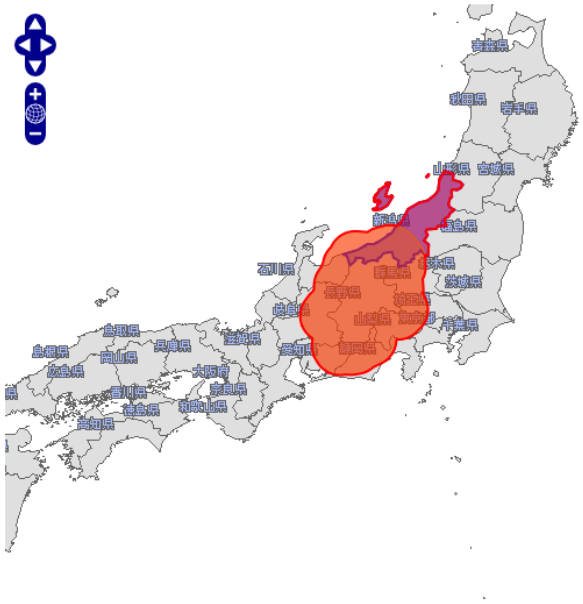


Single geometry processing

- Buffer
  Distance: 0.5
- ConvexHull
- Boundary
- Centroid

Multiple geometries processing

- Union
- Difference
- SymDifference
- Intersection



Single geometry processing

- Buffer
  Distance: 0.5
- ConvexHull
- Boundary
- Centroid

Multiple geometries processing

- Union
- Difference
- SymDifference
- Intersection

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.

Exercise

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

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

- Intersection
- Union
- Difference
- SymDifference

C version

Your 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 `IntputEntity2`. You are invited to update other values in the ZOO Metadata File to set the proper metadata informations.

Python Version

Your 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 `IntputEntity2`. You are invited to update other values in the ZOO metadata file to set the proper metadata informations.

Testing your services

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

4.5 Practical Introduction to ZOO-Project by playing with building blocks

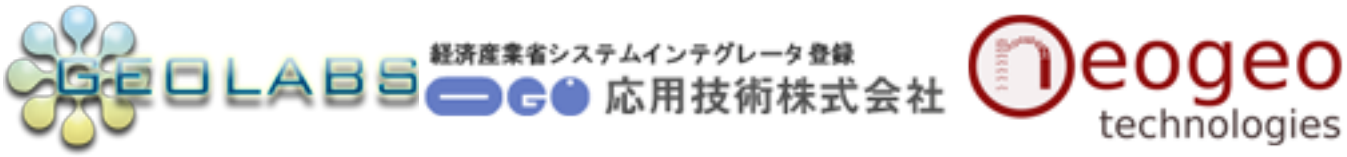
Author Gérald Fenoy, Nicolas Bozon

Contact gerald.fenoy at geolabs.fr, nicolas.bozon at gmail.com

Last Updated $Date$

4.5.1 FOSS4G 2012 Prague

4.5.2 Sponsored By



4.5.3 Special thanks to our Knowledge Partners

4.5.4 WorkShop table of content

Introduction

Table of Contents

What is ZOO ?

ZOO-Project is a WPS (Web Processing Service) open source project 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 and faster.

ZOO was designed to make the service creation and deployment easy, by providing a powerful system able to understand and execute WPS compliant queries. It supports seven programming languages, thus allowing you to create Web Services using the one you prefer. It also lets you use an existing code and to turn it as a WPS Service. The current supported programming languages are the following:

- C/C++
- Python
- Perl
- Java
- Fortran
- PHP
- JavaScript

More information on the project is available on the ZOO Project official website .

How does ZOO works ?

ZOO is based on a C Kernel which is the ZOO-Project core system (aka ZOO-Kernel). The latter is able to dynamically load libraries and to handle them as on-demand Web 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 that can be called using a WPS Execute Request, as well as the desired input/output. Services contain the algorithms and functions, and can be implemented using any of the supported languages.

ZOO-Kernel works as CGI through Apache and can communicate with cartographic engines and Web mapping clients. It simply adds the WPS support to your spatial data infrastructure and your webmapping applications. 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.

What are we going to do in this workshop?

Participants will then learn how to use ZOO Kernel and how to create ZOO Services using the OSGeo Live 5.5 A pre-compiled ZOO 1.3-dev package is provided inside OSGeoLive, so participants won't have to compile and install it manually. Configuration file and basic ways to run ZOO Kernel and ZOO Service will be presented. Participants will be then invited to start programming their own service using Python language. Some ZOO Services will be presented and individually tested inside a ready-to-use OpenLayers application. Finally, this services will be chained using the server-side Javascript ZOO API.

The whole workshop is organized step-by-step and numerous code snippets are available along with 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

```
this is a code block
```

Warning: This is a warning message.

Note: This is an important note.

Let's go !

Configuration and ZOO-Kernel use

Table of Contents

ZOO-Kernel Configuration

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. Every ZOO-Project related material and source code have been placed in `/home/user/zoo-ws-2012` directory. We will work with file included in this directory during this workshop.

Note: we will use ZOO-Kernel or `zoo_loader.cgi` script without any distinction in this document.

As explained later, the ZOO-Kernel may require to store temporary files in `/var/www/temp`. Depending on parameters set in the `main.cfg`, cache files would be located in the same directory.

```
sudo mkdir /var/www/tmp
sudo chown www-data /var/www/tmp
```

General ZOO-Kernel settings are made in the `main.cfg` file located in the same directory as the ZOO-Kernel, so in `/usr/lib/cgi-bin/`. You can see a typical `main.cfg` content in the following:

```
[main]
lang=en-US,fr-FR,ja-JP
version=1.0.0
encoding=utf-8
serverAddress=http://localhost/zoo/
dataPath=/var/www/data
tmpPath=/var/www/tmp
tmpUrl=../tmp
cacheDir=/var/www/cache/
mapserverAddress=http://localhost/cgi-bin/mapserv
msOgcVersion=1.0.0

[identification]
title=The ZOO-Project FOSS4G 2012 Prague Workshop
```

```
keywords=WPS,GIS,buffer
abstract=ZOO-Project platform 2012 .See http://www.zoo-project.org for more informations
accessConstraints=none
fees=None

[provider]
positionName=Developer
providerName=ZOO-Project
addressAdministrativeArea=Lattes
addressDeliveryPoint=1280 Av. des Platanes
addressCountry=fr
phoneVoice=False
addressPostalCode=34970
role=Dev
providerSite=http://www.zoo-project.org
phoneFacsimile=False
addressElectronicMailAddress=gerald.fenoy@geolabs.fr
addressCity=Lattes
individualName=Gérald FENOY
```

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.

From the `[main]` section settings are as follow:

- `lang`: the supported languages separated by a coma (the first is the default one),
- `version`: the supported WPS version,
- `encoding`: the default encoding of WPS Responses,
- `serverAddress`: the url to access your ZOO-Kernel instance,
- `dataPath`: the path to store data files (when MapServer support was activated, this directory is used to store mapfiles and data).
- `tmpPath`: the path to store temporary files (such as ExecuteResponse when storeExecuteResponse was set to true),
- `tmpUrl`: a url relative to `serverAddress` to access the temporary file,
- `cacheDir`: the path to store cached request files [1] (optional),
- `mapservAddress`: your local MapServer address (optional),
- `msOgcVersion`: the version for all supported OGC Web Services output [2] (optional).

The `[identification]` and `[provider]` section are specific to OGC metadata and should be set [3].

Obviously, you are free to add new sections to this file if you need more. Nevertheless, you have to know that there is some specific names you should use only for specific needs: `[env]`, `[lenv]` and `[senv]`.

> **Warning:** `[senv]` and `[lenv]` are used / produced on runtime internaly by the ZOO-Kernel and should be defined only from the Service code.

[1] when you use GET requests passed through `xlink:href` the ZOO-Kernel will execute the request only once, the first time you will ask for this ressource and it will store on disk the result. The next time you will need the same feature, the cached file will be used which make your process running faster. If `cachedir` was not specified in the `main.cfg` then `tmpPath` value will be used.

[2] since version 1.3dev, when MapServer is activated, your service can automatically return a WMS, WFS or WCS request to expose your data. Your can set here the specific version number you want to use to request your local MapServer setup. It depends mostly on the client capability to deal with specific OGC Web Services version.

[3] since version 1.3dev, when MapServer is activated, the same metadata will be used for setting metadata for OGC Web Services.

The `env` section is used to store specific environment variables you want to be set prior to load your Services Provider and run your Service. A typical example, is when your Service requires to access to a X server running on framebuffer, then you will have to set the `DISPLAY` environnement variable, in this case you would add `DISPLAY=:1` line in your `[env]` section.

The `lenv` is used to store runtime informations automatically set by the ZOO-Kernel before running your service and can be accesses / updated from it:

- `sid`: the service unique identifier,
- `status`: the current progress value (value between 0 and 100, percent),
- `cwd`: the current working directory of the ZOO-Kernel,
- `message`: an error message when returning `SERVICE_FAILED` (optional),
- `cookie`: the cookie your service want to return to the client (for authentication purpose or tracking).

The `senv` is used to store session informations on the server side. You can then access them automatically from service if the ser

- **`XXX`: the session unique identifier where `XXX` is the name included in the** returned cookie.

For instance, if you get the following in your Service source code [4] :

```
conf["lenv"]["cookie"]="XXX=XXX1000000; path=/"
conf["senv"]={"XXX": "XXX1000000","login": "demoUser"}
```

That means that the ZOO-Kernel will create a file `sess_XXX1000000.cfg` in the `cacheDir` and return the specified cookie to the client. Each time the client will request the ZOO-Kernel using the Cookie, it will automatically load the value stored before running your service. You can then easilly access this informations from your service source code. This functionality won't be used in the following presentation.

Testing the ZOO installation with GetCapabilities

You can request ZOO-Kernel using the following link from your Internet browser:

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

You should get a valid Capabilities XML document, looking like 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" xml:lang="en-US" version="1.0.0">
 −<ows:ServiceIdentification>
    <ows:Title>The ZOO-Project OSGeoLiveDVD Server 2012</ows:Title>
   −<ows:Abstract>
      Demo version of Zoo-Project for OSGeoLiveDVD 2012. See http://www.zoo-project.org
    </ows:Abstract>
   −<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:Fees>None</ows:Fees>
    <ows:AccessConstraints>none</ows:AccessConstraints>
  </ows:ServiceIdentification>
```

Please note that some Process node are returned in the ProcessOfferings section, as somes are available already on OSGeoLive DVD. You can also run a GetCapabilities request from the command line, using the following command:

[4] If you're not familiar with ZOO-Project, you can pass this part and come to it after the next section.

```
cd /usr/lib/cgi-bin
./zoo_loader.cgi "request=GetCapabilities&service=WPS"
```

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

```
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://w
ww.opengis.net/wps/1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xlink="http://www.w3.org/1999/xlink" xsi:schemaLocation="http://www.opengi
s.net/wps/1.0.0 http://schemas.opengis.net/wps/1.0.0/wpsGetCapabilities_response
.xsd" service="WPS" xml:lang="en-US" version="1.0.0">
  <ows:ServiceIdentification>
    <ows:Title>The ZOO-Project OSGeoLiveDVD Server 2012</ows:Title>
    <ows:Abstract>Demo version of Zoo-Project for OSGeoLiveDVD 2012. See http://
www.zoo-project.org</ows:Abstract>
    <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:Fees>None</ows:Fees>
    <ows:AccessConstraints>none</ows:AccessConstraints>
  </ows:ServiceIdentification>
```

Invoking ZOO Kernel from command line can be helpful during development process of new Services.

Creating your first ZOO Service

Table of Contents

Introduction

In this part, you will create and publish from a simple ZOO Service named `Hello` which will simply return a hello message containing the input value provided. It will be usefull to present in deeper details general concept on how

ZOO-Kernel works and handles request.

Service and publication process overview

Before starting developing a ZOO Service, you should remember that in ZOO-Project, a Service is a couple made of:

- a metadata file: a ZOO Service Configuration File (ZCFG) containing metadata informations about a Service (providing informations about default / supported inputs and outputs for a Service)
- a Services Provider: it depends on the programming language used, but for Python it is a module and for JavaScript a script file.

To publish your Service, which means make your ZOO Kernel aware of its presence, you should copy a ZCFG file in the directory where `zoo_loader.cgi` is located (in this workshop, `/usr/lib/cgi-bin`).

> **Warning:** only the ZCFG file is required for the Service to be considerate as available. So if you don't get the Service Provider, obviously your Execute request will fail as we will discuss later.

Before publication, you should store your ongoing work, so you'll start by creating a directory to store the files of your Services Provider:

```
mkdir -p /home/user/zoo-ws-2012/src/zoo-project/zoo-services/ws_sp/cgi-env
```

Once both the ZCFG and the Python module are both ready, you can publish simply by copying the corresponding files in the same directory as the ZOO-Kernel.

Creating your first ZCFG file

You will start by creating the ZCFG file for the `Hello` Service. Edit the `/home/user/zoo-ws-2012/src/zoo-project/zoo-services/ws_sp/Hello.zcfg` file and add the following content:

```
[Hello]
 Title = Return a hello message.
 Abstract = Create a welcome string.
 processVersion = 2
 storeSupported = true
 statusSupported = true
 serviceProvider = test_service
 serviceType = Python
 <DataInputs>
  [name]
   Title = Input string
   Abstract = The string to insert in the hello message.
   minOccurs = 1
   maxOccurs = 1
   <LiteralData>
       dataType = string
       <Default />
   </LiteralData>
 </DataInputs>
 <DataOutputs>
  [Result]
   Title = The resulting string
   Abstract = The hello message containing the input string
   <ComplexData>
```

```
25     <LiteralData>
26         dataType = string
27         <Default />
28     </LiteralData>
29     </ComplexData>
30  </DataOutputs>
```

Note: the name of the ZCFG file and the name between braket (here `[Hello]`) should be the same and correspond to the function name you will define in your Services provider.

As you can see in the ZOO Service Configuration File presented above it is divided into three distinct sections:

1. Main Metadata information (from line 2 to 8)
2. List of Inputs metadata information (from 9 line to 20)
3. List of Outputs metadata information (from line 21 to 36)

You can get more informations about ZCFG from the reference documentation.

If you copy the `Hello.zcfg` file in the same directory as your ZOO Kernel then you will be able to request for DescribeProcess using the `Hello Identifier`. The `Hello` service should also be listed from Capabilities document.

Test requests

In this section you will tests each WPS requests : GetCapabilities, DescribeProcess and Execute. Note that only GetCapabilities and DescribeProcess should work at this step.

Test the GetCapabilities request If you run the `GetCapabilities` request:

```
http://localhost/cgi-bin/zoo_loader.cgi?request=GetCapabilities&service=WPS
```

Now, you should find your Hello Service in a `Process` node in `ProcessOfferings`:

```
<wps:Process wps:processVersion="2">
 <ows:Identifier>Hello</ows:Identifier>
 <ows:Title>Return a hello message.</ows:Title>
 <ows:Abstract>Create a welcome string.</ows:Abstract>
</wps:Process>
```

Test the DescribeProcess request You can access the `ProcessDescription` of the `Hello` service using the following `DescribeProcess` request:

```
http://localhost/cgi-bin/zoo_loader.cgi?request=DescribeProcess&service=WPS&version=1.0.0&Identifier=
```

You should get the following response:

```
<wps:ProcessDescriptions xmlns:ows="http://www.opengis.net/ows/1.1" xmlns:wps="http://www.opengis.net
  <ProcessDescription wps:processVersion="2" storeSupported="true" statusSupported="true">
    <ows:Identifier>Hello</ows:Identifier>
    <ows:Title>Return a hello message.</ows:Title>
    <ows:Abstract>Create a welcome string.</ows:Abstract>
    <DataInputs>
      <Input minOccurs="1" maxOccurs="1">
        <ows:Identifier>name</ows:Identifier>
        <ows:Title>Input string</ows:Title>
```

```
        <ows:Abstract>The string to insert in the hello message.</ows:Abstract>
        <LiteralData>
          <ows:DataType ows:reference="http://www.w3.org/TR/xmlschema-2/#string">string</ows:DataType
          <ows:AnyValue/>
        </LiteralData>
      </Input>
    </DataInputs>
    <ProcessOutputs>
      <Output>
        <ows:Identifier>Result</ows:Identifier>
        <ows:Title>The resulting string</ows:Title>
        <ows:Abstract>The hello message containing the input string</ows:Abstract>
        <LiteralOutput>
          <ows:DataType ows:reference="http://www.w3.org/TR/xmlschema-2/#string">string</ows:DataType
        </LiteralOutput>
      </Output>
    </ProcessOutputs>
  </ProcessDescription>
</wps:ProcessDescriptions>
```

Test the Execute request Obviously, you cannot run your Service because the Python file was not published yet. If you try the following `Execute` request:

```
http://localhost/cgi-bin/zoo_loader.cgi?request=Execute&service=WPS&version=1.0.0&Identifier=Hello&Da
```

You should get an ExceptionReport similar to the one provided in the following, which is normal behavior:

```
<ows:ExceptionReport xmlns:ows="http://www.opengis.net/ows/1.1" xmlns:xsi="http://www.w3.org/2001/XMI
  <ows:Exception exceptionCode="NoApplicableCode">
    <ows:ExceptionText>Python module test_service cannot be loaded.</ows:ExceptionText>
  </ows:Exception>
</ows:ExceptionReport>
```

Implementing the Python Service

General Principles The most important thing you must know when implementing a new ZOO-Services using the Python language is that the function corresponding to your Service returns an integer value representing the status of execution (`SERVICE_FAILED` [5] or `SERVICE_SUCCEEDED` [6]) and takes three arguments (Python dictionaries):

- `conf` : the main environment configuration (corresponding to the main.cfg content)
- `inputs` : the requested / default inputs (used to access input values)
- `outputs` : the requested / default outputs (used to store computation result)

Note: when your service return `SERVICE_FAILED` you can set `conf["lenv"]["message"]` to add a personalized message in the ExceptionReport returned by the ZOO Kernel in such case.

You get in the following a sample `conf` value based on the `main.cfg` file you saw before.

```
{
  "main": {
    language: "en-US",
```

[5] `SERVICE_FAILED=4`
[6] `SERVICE_SUCCEEDED=3`

```
    lang: "fr-FR,ja-JP",
    version: "1.0.0",
    encoding: "utf-8",
    serverAddress: "http://localhost/cgi-bin/zoo_loader.cgi",
    dataPath: "/var/www/zoows-demo/map/data",
    tmpPath: "/var/www/temp",
    tmpUrl: "../temp",
    cacheDir: "/var/www/temp/"
  },
  "identification": {
    title: "The ZOO-Project OSGeoLiveDVD Server 2012",
    keywords: "WPS,GIS,buffer",
    abstract: "Demo version of Zoo-Project for OSGeoLiveDVD 2012. See http://www.zoo-project.org",
    accessConstraints: "none",
    fees: "None"
  },
  "provider": {
    positionName: "Developer",
    providerName: "ZOO-Project",
    addressAdministrativeArea: "Lattes",
    addressCountry: "fr",
    phoneVoice: "False",
    addressPostalCode: "34970",
    role: "Dev",
    providerSite: "http://www.zoo-project.org",
    phoneFacsimile: "False",
    addressElectronicMailAddress: "gerald.fenoy@geolabs.fr",
    addressCity: "Denver",
    individualName: "Gérald FENOY"
  }
```

In the following you get a sample outputs value passed to a Python or a JavaScript Service:

```
{
  'Result': {
    'mimeType': 'application/json',
    'inRequest': 'true',
    'encoding': 'UTF-8'
  }
}
```

Note: the `inRequest` value is set internally by the ZOO-Kernel and can be used to determine from the Service if the key was provided in the request.

The Hello Service You can copy and paste the following into the `/home/user/zoo-ws-2012/src/zoo-project/zoo-serv` file.

```
def Hello(conf,inputs,outputs):
    outputs["Result"]["value"]=\
            "Hello "+inputs["name"]["value"]+" from the ZOO-Project Python world !"
    return 3
```

Once you finish editing the file, you should copy it in the `/usr/lib/cgi-bin` directory:

```
sudo cp /home/user/zoo-ws-2012/src/zoo-project/zoo-services/ws_sp/cgi-env/* /usr/lib/cgi-bin
```

Interracting with your service using Execute requests

Now, you can request for Execute using the following basic url:

```
http://localhost/cgi-bin/zoo_loader.cgi?request=Execute&service=WPS&version=1.0.0&Identifier=Hello&Da
```

You can request the WPS Server to return a XML WPS Response containing the result of your computation, requesting for ResponseDocument or you can access the data directly requesting for RawDataOutput.

- Sample request using the RawDataOutput parameter:

```
http://localhost/cgi-bin/zoo_loader.cgi?request=Execute&service=WPS&version=1.0.0&Identifier=Hello&Da
```

- Sample request using the default ResponseDocument parameter:

```
http://localhost/cgi-bin/zoo_loader.cgi?request=Execute&service=WPS&version=1.0.0&Identifier=Hello&Da
```

When you are using ResponseDocument there is specific attribut you can use to ask the ZOO Kernel to store the result: `asReference`. You can use the following example:

```
http://localhost/cgi-bin/zoo_loader.cgi?request=Execute&service=WPS&version=1.0.0&Identifier=Hello&Da
```

When computation take long time, the client should request setting both `storeExecuteResponse` and `status` parameter to true. This will make the ZOO Kernel directly return a response containing a `statusLocation` attribut which can be used to access the status of an ongoing service or the result when the process ended.

```
http://localhost/cgi-bin/zoo_loader.cgi?request=Execute&service=WPS&version=1.0.0&Identifier=Hello&Da
```

Conclusion

Even if this first service was really simple it was useful to illustrate how the ZOO-Kernel fill `conf`, `inputs` and `outputs` parameter prior to load and run your function service, how to write a ZCFG file, how to publish a Services Provider by placing the ZCFG and Python files in the same directory as the ZOO-Kernel, then how to interract with your service using both `GetCapabilities`, `DescribeProcess` and `Execute` requesr. We will see in the next section how to write similar requests using the XML syntax.

Presenting building blocks - Using OGR based Web Services

Table of Contents

Introduction

In this section, you will use basic ZOO-Services : `BufferPy`, `IntersectionPy` and `DifferencePy` which use OGR Python module. The intended goal of this section is to present and interact with your new building blocks before chaining them in the next section.

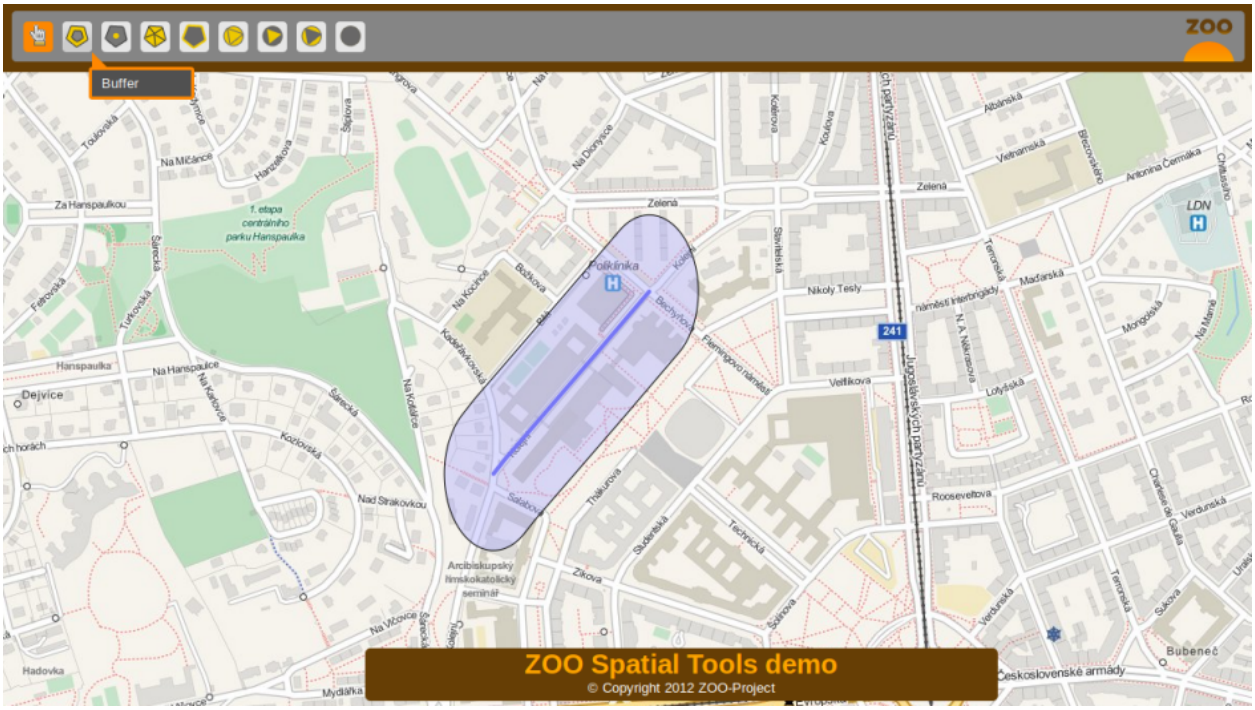
First of all, you should use the following link to access a user interface to interact with your service : http://localhost/zoows-demo/spatialtools-py.html

Services Provider and configuration files

First you may verify if the ZOO-Services are available from your current setup. You can take a look at the `Buffer.zcfg`, `Intersection.zcfg` and `DifferencePy.zcfg` to get details about parameters. As you can see from the ZCFG files, you will use ZOO-Services provided by the `foss4g_ws` Python service provider. So if you want to modify the Python code you will have to edit this file. You are invited to use similar requests as you seen in previous sections to learn about each service.

The Buffer Service

First click on a street then once the street is displayed in blue, click the 'Buffer' button on top, you should get similar result as displayed in the following.



Since version ZOO-Project 1.2.0, you can run automatically some basic tests to make sure that you wrote a correct ZCFG file and your service is validating.

Note: the current testing is really simple and should be adapted to each Services Provider, mainly to define input names.

You can use the following command:

```
cd /home/user/zoo/testing
./run.sh http://localhost/cgi-bin/zoo_loader.cgi Buffer
```

Note: During or after the test run, you can take a look inside the `tmp` directory which contains both the XML requests send to the ZOO Kernel (`*1.xml`) and the responses it gave (`output*.xml`).

The Intersection Service

Using the same client interface as before, once you get a Buffer, you can then select a street intersecting the Buffer geometry to compute intersection by clicking on the Intersection button.

The Difference Service

Using the same instructions as for Intersetion, you can get the following result.

Conclusion

Now you know this three services, and you get a simple interface to interact with your MapServer WFS and your ZOO-Project WPS Servers, you are ready to use the Services in a different way by chaining them using the ZOO-API to build more complexe and powerfull services.

Playing with buildign blocks - Creating JavaScript Web Services

Table of Contents

Introduction

This section illustrate how you can use JavaScript on the server-side to chain services together to build new ones. You will create a ZOO Services Provider using the services you seen before and the WFS server using the ZOO-API. The final goal is to query all POIs included in a buffer around a feature and to highlight them using a mask around this buffer. The following screenshot show you the expected result:

You can decompose the result above in two different ones: the mask around the buffer and the points included in the buffer. So you will create two different Services: one called `BufferMask` and another one called `BufferRequest`.

But before implementing any JavaScript Service, you will get an overview of how to use ZOO-API from your ZOO-Project installation in the next section.

As before, you first create a new directory to store files for your new Services Provider named `ws2012js`:

```
mkdir -p ~/zoo-ws-2012/zoo-project/zoo-services/ws2012js/cgi-env/
```

ZOO-API Overview

ZOO-API and ZOO-Kernel JavaScript support make you able to run services implemented in JavaScript on the server side. JavaScript is a popular programing language but mostly used on the client side. Let say from a browser, but here it is a bit different.

To support JavaScript language ZOO-Kernel use the SpiderMonkey API to create a javascript runtime environment from which it will load your JS file then extract the function corresponding to the service to run it using the prefilled parameters. The JavaScript runtime environment created by the ZOO-Kernel depend on your setup. If you placed the `ZOO-api.js` and `ZOO-proj4js.js` in the same directory as your ZOO-Kernel it means that your environment will contains ZOO-API and Proj4js will be loaded before your service. In such case you can access to the Classes defined in the JavaScript ZOO-API to manipulate geographic data, for more informations please refer to the ZOO-API Documentation.

Even if it can be useful to run JavaScript on the server side, you should remember that some basic JavaScript functions you are familiar with does not exist or get a different behavior. For instance the simple `alert` function will display

messages in apache error logs rather than in a window when used from a browser. The `alert` function can be used as follow:

```
alert("My alert message");
```

There is no XMLHttpRequest available in the JavaScript evironement your service will run into. Hopefully, the ZOO-Kernel expose a C function to the JavaScript world named: `JSRequest`. This function make you able from your JavaScript services to call other WPS services (locally or remotelly) or other kind OGC services such as WFS. When you are using the ZOO-API it is possible to call Services using a `ZOO.Process` instance [7], to parse WPS Responses using `ZOO.Format.WPS` (cf. ref).

As for Python services you already seen in previous sections, the functions corresponding to a Service should take three arguments: `conf`, `inputs` and `outputs` [8]. Nevertheless, as the ZOO-Kernel is not able to access the values modified [9] by the Service code, rather than returning an integer as in Python, here you'll need to return both the integer value representing the Status of your Service in a JavaScript Object and the resulting `outputs` values as an Object. You can see in the following an example of a JavaScript Service code:

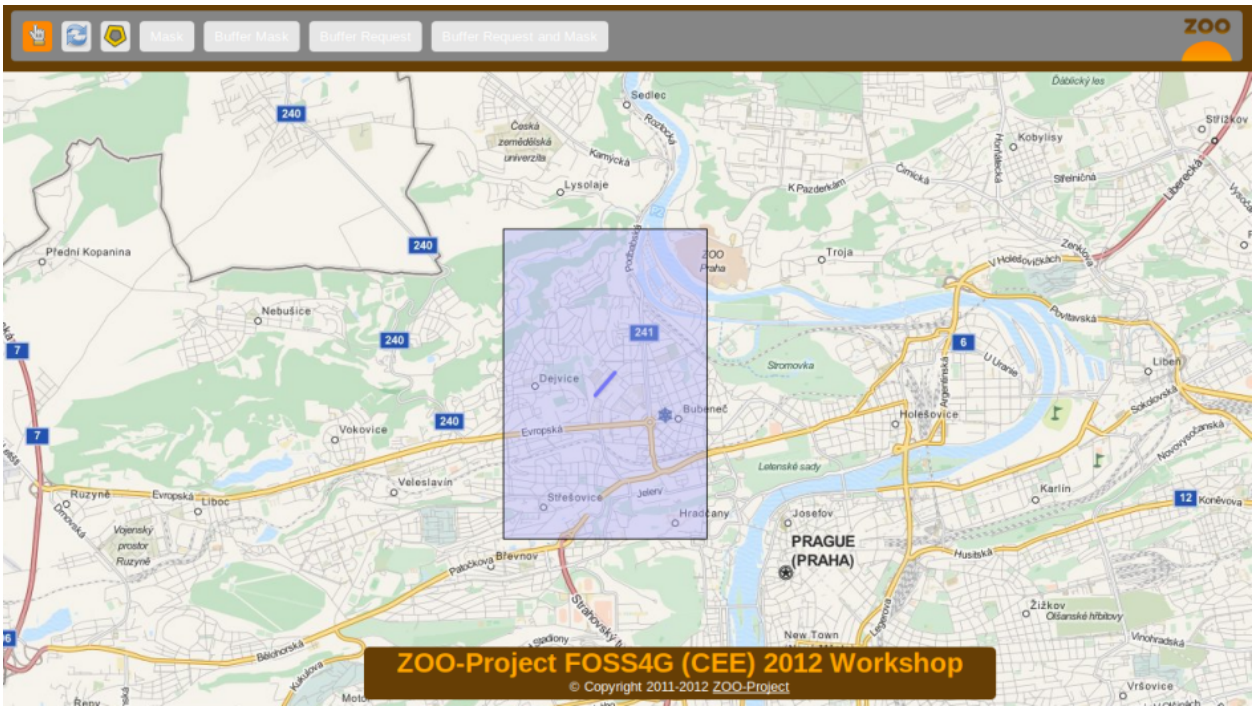
```
function SampleService(conf,inputs,outputs){
  var resultValue=someComputation(inputs);
  return
    {
        result: ZOO.SERVICE_SUCCEEDED,
        outputs: { "Result": { "mimeType": "application/json", "value": resultValue } }
    };
}
```

Before starting to implement the Services we will need to get our final BufferRequest service, let start with a simplier one.

The Mask Service

In this section you will learn how to create your first JavaScript service which will simply return a rectangular mask around a selected feature. To build this mask you will use the Buffer service to create a buffer big enough around a selected geometry to cover a significant part of your map. You can see the expected result in the following screenshot:

[7] The `ZOO.Process` class uses `JSRequest` (cf. ref). You will get example of use later.
[8] So `conf`, `inputs` and `outputs` are simple JavaScript objects, similar to the Python dictionaries used in the previous section.
[9] Such as `conf`, `inputs` and `outputs`.



As before, you will first start by writting the ZCFG, then you will write the JavaScript source code and finish by publishing your Services Provider.

The ZCFG Open the file named `~/zoo-ws-2012/src/zoo-project/zoo-services/ws2012js/cgi-env/Mask.zcf` with your favorite text editor and add the following content:

```
[Mask]
 Title = Compute mask
 Abstract = Compute mask around a geometry
 processVersion = 1
 storeSupported = true
 statusSupported = true
 serviceProvider = foss4gws.js
 serviceType = JS
 <DataInputs>
  [InputData]
   Title = The feature
   Abstract = The feature to run the service with
   minOccurs = 1
   maxOccurs = 1
   <ComplexData>
    <Default>
    mimeType = text/xml
    encoding = utf-8
    </Default>
   </ComplexData>
 </DataInputs>
 <DataOutputs>
  [Result]
   Title = The resulting feature
   Abstract = The feature created by the service.
   <ComplexOutput>
     <Default>
```

```
      mimeType = application/json
      </Default>
    </ComplexOutput>
  </DataOutputs>
```

Here you simply define one default `ComplexData` for both `inputData` and `Result`: a GML and a GeoJSON respectively [10].

The JavaScript service As you will have to request the Buffer service many times from your service, you will first define a `Buffer` function as follow. It uses the `ZOO.Process` to request the Buffer service you seen in the previous section.

Open a file named `~/zoo-ws-2012/src/zoo-project/zoo-services/ws2012js/cgi-env/foss4gws.js` and add the following content:

```
var zoo_url='http://localhost/cgi-bin/zoo_loader.cgi';
var mapserv_url='http://localhost/cgi-bin/mapserv?'+
    'map=/var/www/zoows-demo/map/w2011.map&amp;SERVICE=WFS';

function Buffer(inputData,bDist){

  // Create all required ZOO.formats
  var fJ=new ZOO.Format.JSON();
  var fGJ=new ZOO.Format.GeoJSON();
  var fWPS=new ZOO.Format.WPS();

  // Pass the value as json
  var myInputs = {
      InputPolygon: { type: 'complex', value: fGJ.write(inputData), mimeType: "application/json"},
      BufferDistance: {type: 'float', "value": bDist }
  };
  var myOutputs= { Result: { type: 'ResponseDocument', "mimeType": "application/json" } };
  var myProcess = new ZOO.Process(zoo_url,'BufferPy');
  var myExecuteResult=myProcess.Execute(myInputs,myOutputs);

  // Parse the result and extract JSON geometry
  var bufferResult=fWPS.read(myExecuteResult);
  var bufferResultAsGeoJSON=fJ.read(bufferResult.value);
  return fGJ.read(bufferResultAsGeoJSON);

}
```

From line 12 to 15, you give a GeoJSON string (created from `inputData`) for InputPolygon and, on line 14, you set the BufferDistance value to `bDist`. On line 16, you define Result as a ResponseDocument, so you'll have to parse the WPS response using the ZOO.Format.WPS, on line 21.

On line 17, you create a ZOO.Process instance providing the ZOO-Kernel url and the Service name. Then, on line 18, you run the request passing inputs and outputs previously defined (from line 12 to 15).

Now, you get your `Buffer` function, it is time to create your first JavaScript service. So, edit your `foss4gws.js` file you created before and add the following content:

```
function Mask(conf,inputs,outputs){

  // Create all required ZOO.formats
  var fGML=new ZOO.Format.GML();
```

[10] Using one of the available `ZOO.formats` you are also able to support various `ComplexData` for both input and output of the service. To simplify the presentation here, you will use only this default ones.

```
  var fGJ=new ZOO.Format.GeoJSON();

  // Read the input GML
  var inputData=fGML.read(inputs["InputData"]["value"]);

  // Compute Buffer
  var bufferResultAsJSON=Buffer(inputData,0.015);

  // Create the Buffer result BBOX and store its geometry in a ZOO.Feature
  var bbox = new ZOO.Bounds();
  var bounds=bufferResultAsJSON[0].geometry.getVertices();
  for(var t in bounds){
    bbox.extend(bounds[t]);
  }
  var finalG=bbox.toGeometry();
  var result=new ZOO.Feature(finalG,{"name": "Result1000"});

  // Return the created feature
  return {
      result: ZOO.SERVICE_SUCCEEDED,
      outputs: { "Result": { mimeType: "application/json", value: fGJ.write(result) } }
  };

}
```

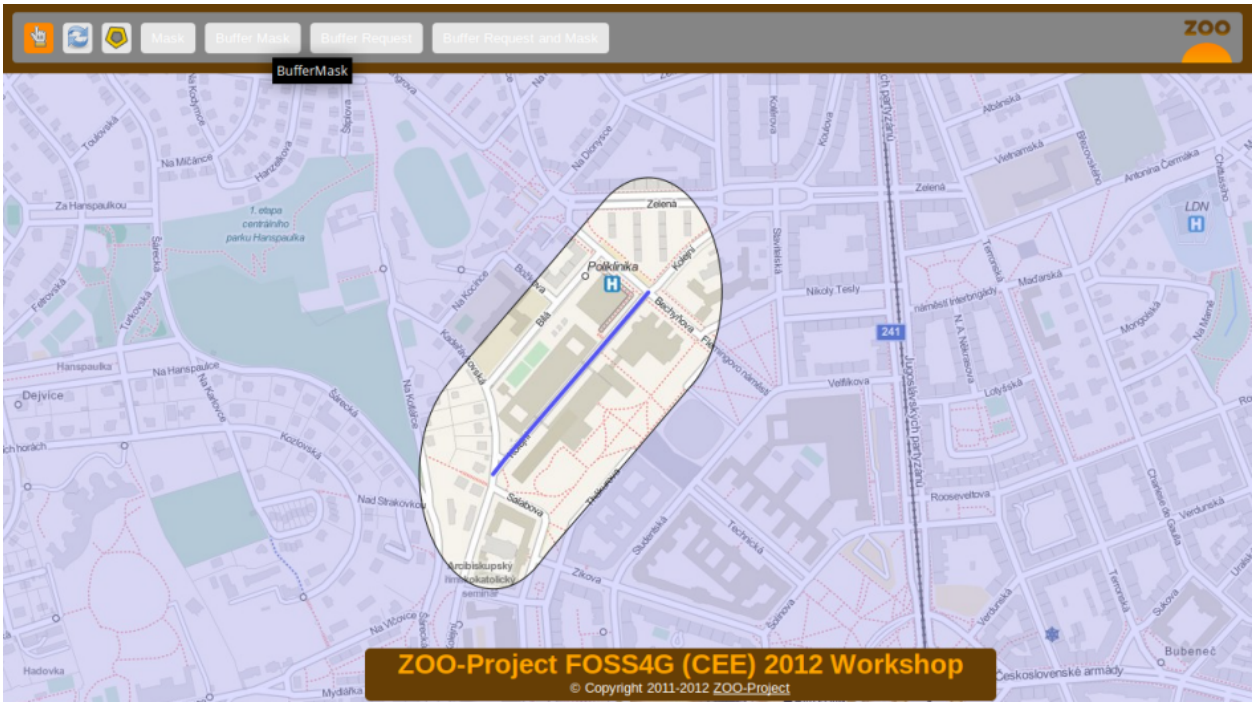
Publish and use your Service Now you get both your ZCFG and your service code ready, you need to deploy your new Services Provider using the following command:

```
cp ~/zoo-ws-2012/src/zoo-project/zoo-services/ws2012js/cgi-env/* /usr/lib/cgi-bin
```

Now you are ready to use your JavaScript service by loading the the following url, click on a street then click on the "Mask" button.

BufferMask Service

In this section you will implement a simple JavaScript service which will be able create a hole in the mask you created in previous section. This service will be used to highlight the buffer zone around a selected fature. You get a preview of the expected result in the following screenshot:



The ZCFG Open the file named `~/zoo-ws-2012/src/zoo-project/zoo-services/ws2012js/cgi-env/BufferMa` with your favorite text editor and copy / paste the following content:

```
[BufferMask]
 Title = Compute buffer mask
 Abstract = Compute buffer mask around a geometry
 processVersion = 1
 storeSupported = true
 statusSupported = true
 serviceProvider = foss4gws.js
 serviceType = JS
 <DataInputs>
  [InputData]
   Title = The feature
   Abstract = The feature to run the service with
   minOccurs = 1
   maxOccurs = 1
   <ComplexData>
    <Default>
    mimeType = text/xml
    encoding = utf-8
    </Default>
   </ComplexData>
 </DataInputs>
 <DataOutputs>
  [Result]
   Title = The resulting feature
   Abstract = The feature created by the service.
   <ComplexOutput>
     <Default>
     mimeType = application/json
     </Default>
   </ComplexOutput>
```

```
</DataOutputs>
```

This ZCFG is similar to the previous one. Please, refer to comments in the previous section for more informations.

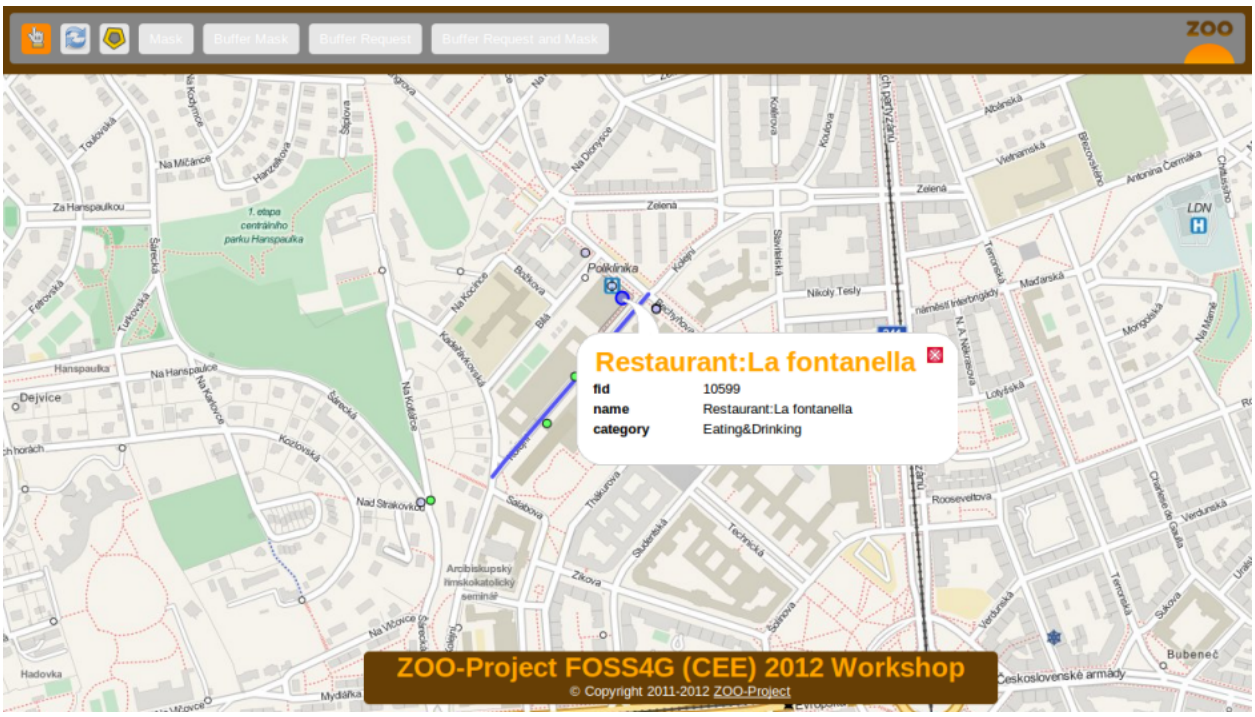
The JavaScript service In this Service you will use same source code (until line 19) you used in the previous section. Indeed, you should compute the Mask as you did before then compute Buffer for creating a hole in the mask (on line 22) to run the Difference service (from line 25 to 40).

```
function BufferMask(conf,inputs,outputs){

  // Create all required ZOO.formats
  var fGML=new ZOO.Format.GML();
  var fGJ=new ZOO.Format.GeoJSON();

  // Read the input GML
  var inputData=fGML.read(inputs["InputData"]["value"]);

  // Compute Buffer
  var bufferResultAsJSON=Buffer(inputData,0.015);

  // Create the Buffer result BBOX
  var bbox = new ZOO.Bounds();
  var bounds=bufferResultAsJSON[0].geometry.getVertices();
  for(var t in bounds){
    bbox.extend(bounds[t]);
  }
  var finalG=bbox.toGeometry();

 // Compute Buffer standard buffer
 var bufferResultAsJSON=Buffer(inputData,0.0015);

 // Request Difference service using Buffer result and features in the BBOX
 var result=new ZOO.Feature(finalG,{"name": "Result1000"});
 var myProcess2 = new ZOO.Process(zoo_url,'DifferencePy');
 var myInputs2 = {
     InputEntity1: {
         type: 'complex',
         value: fGJ.write(finalG),
         mimeType: "application/json"
     },
     InputEntity2: {
         type: 'complex',
         value: fGJ.write(bufferResultAsJSON),
         mimeType: "application/json"
     }
 };
 var myOutputs2= {Result: {type: 'RawDataOutput',  mimeType: "application/json" } };
 var myExecuteResult4=myProcess2.Execute(myInputs2,myOutputs2);

  // Return the bbox
  var result=new ZOO.Feature(finalG,{"name": "Result1000"});
  return {
      result: ZOO.SERVICE_SUCCEEDED,
      outputs: { "Result": {mimeType: "application/json", value: myExecuteResult4 } }
  };

}
```

Publish and use your Service Now, you can publish your service as you did before. To use your service, please use the following url.

BufferRequest service

In this section, you will create a new Service: `BufferRequest` which will request POIs included in the Buffer around a selected feature [11]. You will use the `poi` layer served as WFS through your local mapserver installation. You can see in the following screenshot the expected result:



The ZCFG Open the file named `~/zoo-ws-2012/zoo-project/zoo-services/ws2011js/cgi-env/BufferReques` with your favorite text editor and copy / paste the following content:

```
[BufferRequest]
 Title = Compute buffer request
 Abstract = Compute buffer request around a geometry
 processVersion = 1
 storeSupported = true
 statusSupported = true
 serviceProvider = foss4gws.js
 serviceType = JS
 <DataInputs>
  [InputData]
   Title = The feature
   Abstract = The feature to run the service with
   minOccurs = 1
   maxOccurs = 1
   <ComplexData>
    <Default>
    mimeType = text/xml
    encoding = utf-8
```

[11] So in the hole you created in the previous section.

```
    </Default>
   </ComplexData>
 </DataInputs>
 <DataOutputs>
  [Result]
   Title = The resulting feature
   Abstract = The feature created by the service.
   <ComplexOutput>
     <Default>
     mimeType = application/json
     </Default>
   </ComplexOutput>
 </DataOutputs>
```

The JavaScript code As in the previous Service, you will compute a buffer around the input feature. But then you will request POIs available in the Buffer extent using a WFS request to use them to run `Intersection` service using the initial Buffer. The WFS request is useful to limit the number of points to use when requesting the `Intersection` Service.

```
function BufferRequest(conf,inputs,outputs){

  // Create all required ZOO.formats
  var fGJ=new ZOO.Format.GeoJSON();
  var fGML=new ZOO.Format.GML();

  // Read the input GML
  var inputData=fGML.read(inputs["InputData"]["value"]);

  // Compute Buffer
  var bufferResultAsJSON=Buffer(inputData,0.0015);

  // Create the Buffer result BBOX
  var bbox = new ZOO.Bounds();
  var bounds=bufferResultAsJSON[0].geometry.getVertices();
  for(var t in bounds){
    bbox.extend(bounds[t]);
  }

  // Request Intersection service using Buffer result and WFS request using the
  // BBOX
  var myProcess2 = new ZOO.Process(zoo_url,'IntersectionPy');
  var req="&amp;version=1.0.0&amp;request=GetFeature&amp;typename=poi1";
  req+="&amp;SRS=EPSG:4326&amp;BBOX=";
  var myInputs2 = {
    InputEntity1: {
      type: 'complex',
      value: fGJ.write(bufferResultAsJSON),
      mimeType: "application/json"
    },
    InputEntity2: {
      type: 'complex',
      xlink: mapserv_url+req+bbox.left+","+bbox.bottom+","+bbox.right+","+bbox.top,
      mimeType: "text/xml"
    }
  };
  var myOutputs2= {Result: { type: 'RawDataOutput', "mimeType": "application/json" } };
  var myExecuteResult4=myProcess2.Execute(myInputs2,myOutputs2);
```

```
39
40    return {
41      result: ZOO.SERVICE_SUCCEEDED,
42      outputs: {name:"Result", mimeType: "application/json", value: myExecuteResult4}
43    };
44
45  }
```

Warning: to take advantage of the ZOO-Kernel cache system, you directly use the WFS request as `xlink:href` rather than value for `InputEntity2` (from line 31 to 34) and use `text/xml` `mimeType` (on line 40). Indeed, the ZOO-API doesn't use the internal cache mechanisms.

Publish and use your Service Now, you can publish your service as you did before. To use your service, please use the following url.

Note: You can click on "Buffer Request and Mask" to get the same result as presented in the initial screenshot.

Conclusion

After understanding how basic Geometric Operation Services works, here you built step by step new JavaScript services which reuse the previous ones and combine them in different ways. This was achieved using the ZOO-API, composed by C functions exposed by the ZOO-Kernel to the JavaScript services runtime environement and the JS files which can be optionally installed.

4.6 ZOO Community

The following sections will help you interact with the ZOO community.

4.6.1 Mailing Lists

ZOO-Discuss

The `zoo-discuss` list is your main source of support for the ZOO-Project.

- **Subscribing to zoo-discuss**

  To subscribe to the zoo-discuss listserv visit http://gisws.media.osaka-cu.ac.jp/mailman/listinfo/zoo-discuss. You can later change your subscription information or leave the list at this website.

- **Submitting Questions to zoo-discuss**

  To submit questions to the zoo-discuss listserv, first join the list by following the subscription procedure above. Then post questions to the list by sending an email message to zoo-discuss@gisws.media.osaka-cu.ac.jp.

- **Searching the Archives**

  All zoo-discuss archives are located in http://gisws.media.osaka-cu.ac.jp/pipermail/zoo-discuss/. Searching the archives is best done with Nabble.

4.6.2 IRC Chat

Some of the development of the ZOO-Project is coordinated through IRC. This page describes how you log on to chat, ask questions, and hack around with the developers.

Server and Channel Information

```
Server: irc.freenode.net
Channel: #zoo-project
```

Why IRC?

IRC is a primary medium where Open Source GIS hackers congregate, collaborate, and hack. It makes it easy to communicate things like compilation issues, where immediate, iterative feedback allows folks to make a lot of progress. Something that might take days of heavily-quoted emails through a mailinglist might only take fifteen minutes on IRC.

IRC is a great way to coordinate on-line meetings. The ZOO PSC conducts their monthly meetings through IRC.

How do I join?

- Chatzilla is probably the easiest way to get going. Chatzilla works with Mozilla or Firefox, and once you have it installed, you can log on to the channel by pointing your browser at:

  `irc://irc.freenode.net/#zoo-project`

- There are many other IRC clients available. This page provides a good listing for many different platforms.
- You can also connect to IRC directly in your Web browser without a client:
  - goto http://webchat.freenode.net/
  - enter a nickname (usually your first initial plus your last name)
  - in "Channels" enter: zoo-project
  - Connect!

4.7 Documentation Development Guide

Authors Nicolas Bozon, Gérald Fenoy, Jeff McKenna

Last Updated $Date: 2011-12-07 14:19:47 +0100 (Mer, 07 déc 2011) $

Table of Contents

4.7.1 Background

The current structure of the ZOO Project documentation process is for developers with *SVN* commit access to maintain their documents in reStructuredText format, and therefore all documents live in the /docs directory in SVN. The Sphinx documentation generator is used to convert the reStructuredText files to html, and the live website is then updated on an hourly basis.

4.7.2 reStructuredText Reference Guides

- Docutils Quick reStructuredText
- Docutils reStructuredText Directives
- Sphinx's reStructuredText Primer
- search Sphinx's mailing list

4.7.3 reStructuredText Formatting

- All text should be hard breaks at or around the 80 column mark, just as the source code.

4.7.4 Installing and Using Sphinx for rst-html Generation

Note: You can browse the versions of the Sphinx packages here, and then install the exact version such as:

```
easy_install Sphinx==1.0.7
```

On Windows:

1. install Python 2.X
2. download setuptools
3. make sure that the `C:/Python2X/Scripts` directory is your path
4. execute the following at commandline:

   ```
   easy_install Sphinx
   ```

   ...you should see message: "Finished processing dependencies for Sphinx"

   Note: Make sure you install Sphinx 1.0 or more recent. See note above.

   ---

5. install MiKTeX if you want to build pdfs
6. checkout the /docs directory from SVN, such as:

   ```
   svn checkout http://svn.zoo-project.org/svn/trunk zoo-project
   ```

7. inside the /docs directory, execute:

   ```
   make html
   ```

   or

```
make latex
```

the HTML output will be written to the _build/html sub-directory.

On Linux:

1. make sure you have the Python dev and setuptools packages installed. On Ubuntu:

   ```
   sudo apt-get install python-dev
   sudo apt-get install python-setuptools
   ```

2. install sphinx using easy_install:

   ```
   sudo easy_install Sphinx
   ```

   Note: Make sure you install Sphinx 1.0 or more recent. See note above.

   ---

3. checkout the /docs directory from SVN, such as:

   ```
   svn checkout http://svn.zoo-project.org/svn/trunk zoo-project
   ```

4. to process the docs, from the ZOO /docs directory, run:

   ```
   make html
   ```

   or

   ```
   make latex
   ```

   the HTML output will be written to the build/html sub-directory.

Note: If there are more than one translation, the above commands will automatically build all translations.

On Mac OS X:

1. install sphinx using easy_install:

   ```
   sudo easy_install Sphinx
   ```

   Note: Make sure you install Sphinx 1.0 or more recent. See note above.

   ---

2. install MacTex if you want to build pdfs
3. checkout the /docs directory from SVN, such as:

   ```
   svn checkout http://svn.zoo-project.org/svn/trunk zoo-project
   ```

4. to process the docs, from the ZOO /docs directory, run:

   ```
   make html
   ```

   or

   ```
   make latex
   ```

   the HTML output will be written to the build/html sub-directory.