ZOO 実践入門:
The Open WPS Platform

FOSS4G 2010 Osaka / Tokyo

Sponsored By

Special thanks to our Knowledge Partners

Laboratorio di Geomatica
Politecnico di Milano
via Valleggio, 11 - 22100 COMO

SUPSI Scuola universitaria professionale della Svizzera italiana

ワークショップ　目次

1. はじめに 3
1.1 ZOO とは？ 3
1.2 ZOO はどのように動作するのか 3
1.3 このワークショップで行うこと 4
2. OSGeoLive の仮想マシンによる ZOO の利用 5
2.1 ZOO カーネルのインストール 5
2.2 GetCapabilities で ZOO のインストレーションをテストする 7
2.3 ZOO Services Provider ディレクトリの準備 9
3. OGR の Web サービス作成 10
3.1 イントロダクション 10
3.2 ZOO メタデータファイルの準備 10
3.3 ジオメトリサービスの実装 14
3.3.1 Boundary 15
3.3.2 その他の関数サービスの作成 (ConvexHull and Centroid) 30
3.3.3 バッファサービス作成 36
4. OpenLayers を使用した WPS クライアントの構築 42
4.1 WMS のデータセットを表示する簡単なマップの作成 42
4.2 WFS のデータレイヤー取得と選択コントロールの追加 44
4.3 JavaScript から呼び出される単一ジオメトリサービス 46
4.4 JavaScript から呼び出される複合ジオメトリサービス 49
5. 演習 53
5.1 C バージョン 53
5.2 Python バージョン 53
5.3 サービスのテスト 54

1. はじめに

1.1 ZOO とは？

ZOO プロジェクトは MIT/X-11 ライセンスのもとに公開された WPS（Web Processing Service) オープンソースプロジェクトです。 ZOO は WPS Web サービスを生成、チェイン（サービスとサービスを連携）し、さらに OGC WPS 準拠の親切なフレームワークを提供します。ZOO は、3つの主要なパートにより構成されています:

- ZOO カーネル: 様々なプログラミング言語によって作成された Web サービスを管理、そしてサービスチェインを可能にする強力なサーバーサイドの C カーネルです。
- ZOO サービス: 多様なオープンソースライブラリを基に開発された Web サービスの一式
- ZOO API: サーバーサイドの JavaScript API は、ZOO サービスを呼びだし、チェインを行う。このことは開発や処理の連携を容易にします。

ZOO は WPS 準拠のクエリ を理解・実行する強力なシステムを提供することで、WPS サービスの開発を、円滑に行うためにデザインされました。ZOO はいくつかのプログラミング言語をサポートすることで、既存のコードから好みの言語で Web サービスを生成することができます。プロジェクトに関する詳しい情報は、ZOO プロジェクトオフィシャルサイトを参照ください。

1.2 ZOO はどのように動作するのか

ZOO は、ZOO の コアシステム (別名 ZOO Kernel)を構成する'WPS サービスカーネル'に基づいています。そのコアシステムは、動的ライブラリを取り込み（ロードし）、 オンデマンド Web サービスとしてそれらを扱うことができます。ZOO カーネルは C 言語で書かれていますが、ZOO サービスを生成するために、いくつかのプログラム言語をサポートします。 ZOO サービスは ZOO メタファイル (.zcfg) と、それと対応する実装についてのコードを構成する一つのリンクです。メタファイルはすべての利用可能な機能を記載しており、そのファイルは任意の入出力 と同様に WPS Execute リクエストによって呼び出すことができます。サービスはアルゴリズムや機能を含み、現在、C/C++, Fortran, Java, Python, PHP そして JavaScript によって実装することができます。 ZOO カーネルは Apache と共に動作し、地図生成エンジンや Web マッピングクライアントと通信が行えます。ZOO カーネルは、あなたの空間情報基盤（SDI）システムや、Web マッピングアプリケーションに、WPS 機能を簡単に追加できます。 GDAL/OGR によってサポートされたすべての形式のデータを、入力データ

を使用でき、あなたの地図生成エンジンや、Web マッピングクライアントに適したベクトル・ラスター形式のデータ出力を生成できます。

1.3 このワークショップで行うこと

このワークショップは、ZOO プロジェクトとその特徴、また OGC WPS 1.0.0 仕様に関する機能についての紹介を目的としています。参加者は 3 時間に及び、ZOO カーネルの使用方法、ZOO サービスとその設定ファイルの作り方、最後に、作成したサービスをクライエント側のウェブマッピングアプリケーションにリンクする方法を学びます。 プリコンパイルされた ZOO 1.0 のバージョンは、OSGeo 公式ライブ DVD の OSGeoLive で提供しています。ワークショップを簡潔にするために、OSGeoLive 仮想マシンイメージディスクは、すでにあなたのコンピュータにインストールされています。そのため、参加者は手動で ZOO カーネルをコンパイルし、インストールする必要はありません。 よって、ワークショップの最初のステップは、この OSGeoLive イメージディスクから ZOO カーネルを動作、及びテストすることであり、全参加者が少なくとも 30 分で ZOO カーネルを動かすことができるでしょう。

GetCapabilities リクエストを利用して、ウェブブラウザから ZOO カーネルがテストされると、参加者はベクトルデータにおける基本的な空間処理を目的とした、OGR ベースの ZOO サービスプロバイダを作成してもらうことになるでしょう。参加者にはまず、C 言語または Python のどちらを使用するか選択して頂きます。ZOO サービスプロバイダー、および関連したサービスのすべてのプログラミングステップは、C 言語と Python で詳記されます。

ZOO サービスが準備され、ZOO カーネルによって起動可能になると、参加者は OpenLayers の簡単なアプリケーションから、様々な機能の使い方を学びます。サンプルデータは、オークニー社から提供されたもので、 OSGeo ライブ DVD の中に含まれています。また、データは MapServer により OGC WMS/WFS 標準 Web サービスとして利用可能であり、簡単な地図上に表示、そして ZOO の入力データとして使用されます。さらに、いくつかの特定の操作・実行コントロール機能は、表示されるポリゴン上で単一及び、複数のジオメトリを実行するために JavaScript コードに追加されます。

すべての手順は段階的にまとめられ、様々なコードの抜粋と、その説明をそれぞれ行ないます。 講師が各マシンの ZOO カーネルが機能しているかなどをチェックし、コードディング中のサポートを行ないます。 ワークショップ中、技術に関するご質問があればいつでも受け付けます。

Let's go !

2. OSGeoLive の仮想マシンによる ZOO の利用

OSGeoLive は Xubuntu を基にしたライブ DVD の一つであり、インストールをすることなく、多くのオープンソース地理空間情報に関連するソフトウェアを利用することができる。これは全てフリーオープンソースソフトウェアから構成され、今年からテストを目的として ZOO1.0 も含んでいます。

2.1 ZOO カーネルのインストール

導入部分での説明の通り、すぐに ZOO カーネルを利用できるように、OSGeoLive 仮想マシンイメージディスクが、あなたのコンピュータにインストールされています。仮想マシンイメージディスクの使用は、ZOO カーネルの使用や、ローカル環境での ZOO サービス開発のための最も簡単な方法であり、C サービスのコンパイルや、Python の動作をするための必要な設定はすべて行われています。ZOO に関連するすべての教材やソースコードは、/home/user/zoows ディレクトリに置かれています。ワークショップ中は、このディレクトリ内で作業します。 ZOO カーネルのバイナリバージョンもまた、/home/user/zoows/sources/zoo-kernel にコンパイルされ保存されているので、ZOO カーネルを使用するためには、/usr/lib/cgi-bin ディレクトリの zoo_loader.cgi と main.cfg の2つの重要なファイルをコピーするだけです。以下のコマンドに従ってください。:

```
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
```

このワークショップ中は、ZOO カーネルと zoo_loader.cgi スクリプトについては、同じものとして扱いますので注意してください。

main.cfg ファイルは、identification や provider に関するメタ情報だけでなく、いくつかの重要な設定情報も含まれています。ファイルは様々なセクション、主に main,identification と provider をデフォルトとして構成されています。もちろん特定のサービスのために、新しいセクションの追加は自由に行ってください。 しかし、env 及び lenv セクションの名前は、特定の方法で扱われることに注意してください。 env は、実行中のあなたのサービスに必要とする環境変数を定義します。例えば、あなたのサービスがフレームバッファ上の X サーバーへのアクセスを要求したとき、この特異性を考慮するために DISPLAY=:1 の1行を env セクション中に追加することができます。env セクションと同様に、main.cfg には lenv セクションがあります。これは、ZOO カーネルや ZOO サービスによって、実行中のサービスのステータス情報が記述されます。例えば、あなたのサ

ービスがうまく動作しなかった際、クライアントに返答される ExecuteResponse の Status ノード中に表示されるように、lenv の中の message の値を設定することができます。 また、もしあなたのサービスが実行に時間がかり、処理ステータスの情報が取得できる場合、完了した実行中のサービスタスクをパーセンテージを表示するために 、lenv に値を 0 から 100 の間で設定できます。

このファイルを確認してください。3つの重要なパラメーターが下記に示されています。

- serverAddress : ZOO カーネルへ接続するための URL
- tmpPath : 一時ファイル保存場所へのフルパス
- tmpUrl : serverAddress に関連した一時ディレクトリへアクセスするための URL フルパス

実行中の仮想マシンから使用される main.cfg ファイルの値は次の通りです:

```
serverAddress=http://localhost/zoo
tmpPath=/var/www/temp
tmpUrl=../temp/
```

気がついているかもしれませんが、tmpUrl は serverAddress からの相対 URL であり、ディレクトリでなければなりません。ZOO カーネルが zoo_loader.cgi スクリプトの完全 URL として使用できるとしても、ZOO カーネルの完全な機能や読み易さのため、http://localhost/zoo/ として直接 URL を使用できるように、Apache の標準設定を変更しなければなりません。

最初に、zoo ディレクトリを/var/www/(Apache の DocumentRoot ディレクトリ)の中に作成してください。次に /etc/apache2/sites-available/default の設定ファイルを編集し、/var/www に関する Directory ブロックの後に、下記の数行を追加してください:

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

/var/www/zoo のなかに、.htaccess を新しく作り、下記の数行を記述してください:

```
RewriteEngine on
RewriteRule call/(.*)/(.*)
/cgi-bin/zoo_loader.cgi?request=Execute&service=WPS&version=1.0.0&Iden
tifier=$1&DataInputs=sid=$2&RawDataOutput=Result [L,QSA]
RewriteRule (.*)/(.*) /cgi-bin/zoo_loader.cgi?metapath=$1 [L,QSA]
```

```
RewriteRule (.*) /cgi-bin/zoo_loader.cgi [L,QSA]
```

この最後のファイルを Apache で実行するためには、以下のようにロードファイルをコピーして、Apache モジュールの書き換えを実行しなければなりません。:

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

または、a2enmod ツールを次のように使用してください:

```
a2enmod rewrite
```

Apache ウェブサーバーを再起動することで、ZOO カーネルにアクセスできます。:

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

このワークショップでは OSGeoLive 環境から、その他の2つのソフトウェアを使用します。

まず MapServer は、これから設定していく ZOO サービスのための WFS 入力データとして使用します。セクション3 では、オークニー社（日本のポリゴンデータ）から提供された MapServer データセットを、作成したサービスで使用されます。

OpenLayers ライブラリもまた、OSGeo Live 仮想マシンイメージディスクの中で利用できます。セクション4 で、新しく作られた ZOO サービスの問合せに使う簡単な WPS クライアントアプリケーションに利用されます。

GDAL ライブラリの Python モジュールと OGR C-API を使用するため、対応するヘッダーファイル、ライブラリ、関連ファイルを必要とします。すべてがデフォルトで、OSGeoLive パッケージ上で使用できる状態になっています。

2.2 GetCapabilities で ZOO のインストレーションをテストする

ブラウザからの以下のリクエストに従って簡単に ZOO カーネルのクエリーを実行できます:

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

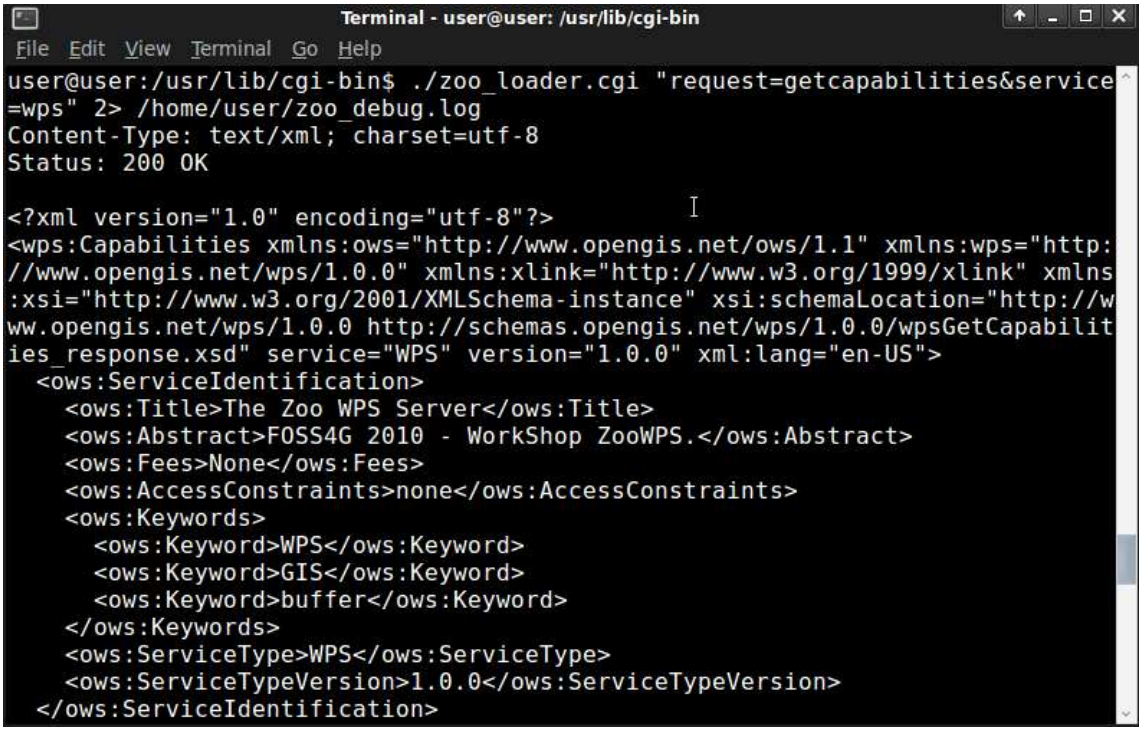
これで、以下の有効な Capabilities XML ドキュメントを得ることができます:

```
−<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>
```

利用可能な ZOO サービスがないと、ProcessOfferings セクションで、no Process ノードが戻ってくることに注意してください。以下のコマンドをつかってコマンドラインからの GetCapabilities リクエストも行うことができます:

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

次のスクリーンショットに表示されているように、ブラウザと同様の結果が返されます:



コマンドラインからの ZOO カーネルの実行は、新しいサービスの開発に便利です。

2.3 ZOO Services Provider ディレクトリの準備

まず作業を簡略化するために、新しい Services Provider を作成する際に使用されるディレクトリ構造について説明します:

- メイン Services Provider ディレクトリはこれらを含みます:
  - すべての zcfg メタファイルと 共有オブジェクト(C Shared Library あるいは Python もモジュール)サービスを含む cgi-env ディレクトリ
  - Services Provider をコンパイルするためには Makefile と *c ファイルが必要です。

上記のことに注意しながら、ws_sp メイン Services Provider ディレクトリを、/home/user/zoows/sources/にある既存の zoo-services ディレクトリに作成してください。

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

次のセクションで Makefile と C 、Python Service Shared Object のコードについて説明していきます。

3. OGR の Web サービス作成

3.1 イントロダクション

このパートでは ZOO のサービス提供機能を作成します。これには OSGeoLive の DVD に含まれている ZOO のインストーラーで組み込まれる OGR の C 言語 API、Python モジュールを使います。目標とするゴールは、OGR と GEOS ベースの簡単な空間関数を WPS サービスとして使うことです。

最初に Boundary（境界）の関数からスタートしましょう。この関数の詳細については後ほど説明いたします。徐々に ZOO サービスとしてコードを書き、テストを行います。 同じ手順で Buffer(バッファ領域の作成)、Centroid(重心)、Convex Hull（凸構造体）の関数についても行います。一旦出来上がると、複数の幾何プロセス、たとえば Intersection（交差），Union（融合），Difference（差異）、そして Symetric Difference（対称性の差異）などがこのワークショップの終わりには組み込まれていることになります。

既に述べたように、このワークショップではサービスを記述する言語は C もしくは Python（もしくは両方！）を選ぶことができます。解説は C 言語に沿って行いますが、この解説は Python を利用する方にも非常に役立つ内容になっています。どちらの言語を使用するか決めたらインストラクターに報告してください。どちらの言語を選択されても、ワークショップの結果は同じです。

3.2 ZOO メタデータファイルの準備

ZOO メタデータの用意

ZOO のサービスは、ZOO メタデータファイル(.zcfg)と ZOO サービスプロバイダーと呼ばれる機能の応じたランタイム実装モジュールとの組み合わせによって成り立ちます。 まずはじめに順を追って .zcfg ファイルを用意しましょう。 使う慣れているテキストエディタで /home/user/zoows/sources/zoo-services/ws_sp directory フォルダにある Boundary.zcfg を開いてください。 最初に、ファイル先頭にある括弧中に以下のようにサービス名を定義します。

```
[Boundary]
```

この名前は非常に重要です。この名前はサービス名になりサービスプロバイダーの機能を表します。タイトル（Title）とこのサービスが何を行うのかについての短い要約（Abstract）は、利用者のために書いておくべきです。

```
Title = Compute boundary.
```

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

これらのメタデータ情報は GetCapabilities リクエストによって返されます。

他にも processVersion のような特定情報を追加することができます。 ZOO サービスが結果を保存する機能を持つかどうかを storeSupported パラメータの値を true もしくは false にすることで指定できます。また処理をバックグラウンドで行い、結果をステータスで通知するかどうかを statusSupported の値で指定します。

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

ZOO サービス・メタデータファイルのメインのセクションでは、二つの重要事項を指定します。 In the main section of the ZOO Service metadata file, you must also specify two important things:

- serviceProvider　サービスの関数を含む C の共用ライブラリの名前、または Python モジュール名
- serviceType　サービスに使用するプログラミング言語の指定 (値は C または Python となります。あなたが使用する言語とあわせてください。)

C !サービスプロバイダーの例 ：

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

この例では Boundary 関数を含む ogr_ws_service_provider.zo 共用ライブラリを使うことになります。ライブラリファイルは ZOO カーネルと同じディレクトリに置いてください。

Python !サービスプロバイダーの例 ：

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

この例では Boundary 関数の Python コードを含む ogr_ws_service_provider.py ファイルを使うことになります。

メインセクションでは、下記の例のように他の任意の情報を追加することもできます。

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

```
</MetaData>
```

これで主要なメタデータ情報が定義できました。 次は ZOO サービスで使われるデータの入力方法を定義します。サービスに必要な任意の入力を定義できます。 ZOO のカーネルは.zcfg ファイルに定義したデータよりも多くのデータを処理するリクエストを受け付けることに注意してください。これらのデータはフィルタリングされることなくサービスへと渡されます。 この例の Boundary サービスでは、一つのポリゴンが入力として使用され、それに対して Boundary 関数が実行されます。

入力データ宣言は DataInputs ブロックで行われます。 書式はサービスを定義するときと同じで、名前は括弧でくくられています。同様にタイトルや要約を入力の MetaData セクションに追加することができます。minOccurs と maxOccurs のパラメータは必ず設定してください。これらは、どのパラメータがサービス関数で実行するのに必要なのかを ZOO カーネルに通知するのに使われます。

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

メタデータは、どのようなタイプのデータがサービスでサポートされるのかを定義します。この Boundary 関数の例では、入力するポリゴンは GML ファイル形式または JSON 文字列形式で提供することができます。 次のステップではデフォルトのデータ形式とサポートする入力フォーマットを定義しましょう。それぞれのフォーマットはタイプに応じて LitteralData または ComplexData のブロックで定義する必要があります。ここでは最初の例として ComplexData ブロックのみを使います。

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

```
 </Supported>
</ComplexData>
```

次に、同様のメタデータ情報をサービスの出力についても定義します。DataOutputs のブロック内に次のように定義します。

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

この .zcfg ファイルのコピーは以下の URL からも入手できます ：

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

ZOO メタデータファイルの修正が終わりましたら、ZOO のカーネルと同じディレクトリ (この例では /usr/lib/cgi-bin)へコピーしてください。これで以下の URL のリクエストが実行できるはずです:

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

返って来る ProcessDescriptions XML ドキュメントは、以下のようになっているはずです ：

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

GetCapabilities と DescribeProcess を実行には .zfg ファイルさえあれば完了します。簡単ですね？ このステップでは、ZOO カーネルに Execute をリクエストすれば、 以下のような ExceptionReport ドキュメントのレスポンスが返ってきます:

```
−<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>
```

もし Python のサービスで実行すれば、同じようなエラーメッセージが返ってきます ：

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

3.3 ジオメトリサービスの実装

サービスプロバイダー作成と開発をステップバイステップで学習するため、はじめに Boundary 関数専用の非常に簡単なものの作成に着目しましょう。 同様の手段を Buffer、 Centroid と ConvexHull の実装に用います。

メタデータは既にOKです。それでは、サービスのコード作成に入りましょう。 ZOOサービスのコーディングの際注意すべき、もっとも重要なことは、あなたの作成する対応サービスは3つの引数（内部の maps データ型または Python dictionaries）、整数型の実行ステータスを示す返り値(SERVICE_FAILED or SERVICE_SUCCEEDED)を必要とするということです:

- conf : 主な実行環境設定(main.cfg の記載に対応)
- inputs : 入力に関する リクエスト / 初期値
- outputs : 出力に関する リクエスト / 初期値

3.3.1 Boundary

C バージョン

先に述べたとおり、ZOO カーネルは、maps と呼ばれる特別なデータ型の中で、あなたの作成したサービス機能の引数を受け渡します。 サービスを C 言語で記述するために、あなたはこの read/write モードのデータ型にどのようにアクセスするかを学ぶ必要があります。maps は、map と呼ばれる name を格納するリストへのリンクで、content map と次の map へのポインタ（またはそれ以上 map がない場合は NULL ）からなります。ここに、あなたが確認可能な zoo-kernel/service.h ファイル内で定義されたデータ型を示します:

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

maps に含まれる map も単純なリンクされたリストで、キー値のペアを格納するために用いられます。 したがって、map は name と value そして次の map へのポインタからなります。 ここに、あなたが確認可能な zoo-kernel/service.h ファイル内で定義されたデータ型を示します:

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

部分的あるいは全部を埋めたデータ構造対が ZOO カーネルからあなたのサービスへ受け渡されたとして、この意味は、各々の maps 中の既存の map に関わる以外には、maps の生成に関わらな

くてよいという事です。 始めに知っておくべき関数は、 maps 内の map の詳細にアクセスするための getMapFromMaps (zoo-kernel/service.h ファイルに定義) です。 この関数は次の3つの引数からなります:

- m : 任意の map を検索するのに用いる maps ポインタの代表値（※先頭ポインタ）
- name : 検索する map の名称を示す char* の代表値（※文字列型の先頭ポインタ）
- key : name という名称の map 内の固有キー

例えば、inputs という名の maps 内の InputPolygon の値の map にアクセスする構文は次の通りです。C コードではこのようになります:

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

map を得た時点で、次のような構文で値フィールドにアクセスすることが可能です:

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

maps map のフィールドへのアクセスと読み込みはご存じのとおりです、今からこのようなデータ構造への書き込みはどのように行うかを学習しましょう。 これもまた zoo-kernel/service.h ファイルに定義された、単純な setMapInMaps 関数を用います。setMapInMaps 関数は4つの引数を必要とします:

- m : 更新を行う maps ポンタ,
- ns : 更新を行う maps の名称,
- n : 値の更新を行う map の名称,
- v : この map にセットする値.

これは、どのようにして outputs の Result maps 内のいくつかの map の値を追加または編集するかという例です:

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

既存の map の生成と更新には setMapInMaps 関数が使用されるということに注意してください。確かに、もし « value » と呼ばれる map が既存の場合（※NULL ではないという意味）、その値は自動的に更新されます。 このワークショップでは、maps の map を使う場

合、zoo-kernel/service.h に定義された addToMap 関数を直接使用し、map 内の値の追加と更新を行うこともできます:

- 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.

このデータ型はすべての C 言語ベースの ZOO サービスにおいて使用されるほど、本当に重要です。そしてそれは、個々のデータ型の利用を除いて、他の言語でも同様にであることを示しています。たとえば Python の場合、ディクショナリー型が用いられ、さらに簡単に操作することができます。

ここに、Python 言語による maps データ型の対応例があります(これは maps の main 設定内の要約です):

```
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"
  }
}
```

maps と map にどのように対応するかを知るために、OGR Boundary 関数を使用した最初の ZOO サービスを書く準備をしましょう。

すでにはじめにで述べたように、OSGeoLive に準備された MapServer WFS server を使用します、それで完全な WFS レスポンスを入力値に利用することが出来るでしょう。私たちは、完全な WFS レスポンスというよりはむしろ、ジオメトリオブジェクトだけを簡単に利用する OGR_G_GetBoundary のように OGR Geometry 関数を使用しましょう。まずはじめにすることは、完全な WFS レスポンスに定義されたジオメトリを抽出する関数を書くことです。私たちはそれを createGeometryFromWFS と呼びましょう。

これはそのようなコードです:

```
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;
}
```

関数の本体に使用された errorException の呼び出しに注目してください。この関数は zoo-kernel/service_internal.h に定義されており、zoo-kernel/service_internal.c に記述されています。それは下記のように3つの引数を必要とします:

- 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).

言い換えると、WFS レスポンスがプロパティを解析されなかった場合、問題が起こったことをクライアントに知らせる ExceptionReport 文が返されます。 WFS レスポンスからジオメトリオブジェクトを抽出する関数が書かれ、そしていまから Boundary サービスの定義を始めることができます。これが Boundary サービスの全体コードです:

```
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;
}
```

上記のコードに見られるように、私たちのサービスに受け渡されたデータインプットの mimeType が初めにチェックされます:

```
  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);
```

基本的に、application/json に設定された mimeType を持つ入力を受け取った場合、ローカル関数の OGR_G_CreateGeometryFromJson が、そうでない場合は createGeometryFromWFS を用います。 入力値はいつも同様の種類ではないことに注意してください。実際には、OGR_G_CreateGeometryFromJson を直接使用することは、JSON 文字列は完全な GeoJSON のものではなくてジオメトリオブジェクトだけを含むものであることを意味します。とはいえ、あなたはこのコードを GeoJSON 文字列からジオメトリオブジェクトを抽出する関数を作り出すよう、簡単に完全な GeoJSON 文字列を使えるように書き換えることができます（OGR GeoJSON Driver にも利用されている json-c ライブラリのインスタンスを使うことにより）。

入力されたジオメトリオブジェクトにアクセスできると、OGR_G_GetBoundary 関数を用いて、 res のジオメトリオブジェクトに結果を格納することができます。その時、あなたはサポートされた出力フォーマットとして宣言した初期値の GeoJSON または GML として正しいフォーマットの値だけを格納しなければなりません。（※zcfg にそのように記載したので） ZOO カーネルは、事前にフィルされた出力値を提供し、たとえもしわれわれの例が、application/json よりもむしろ

text/plain を使用する mimeType を使用して書き換えた場合は、あなたはキー名に対応する値を埋めることだけを行うように注意してください。実際には、クライアントからリクエストされたフォーマットにより(あるいは初期値のもの)、わたしたちは JSON または GML 表現のジオメトリを準備ます。

```
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);
}
```

Boundary ZOO サービスは実装されたので、あなたは共有ライブラリを作るためそれをコンパイルする必要があります。 service.h に定義された関数 (getMapFromMaps, setMapInMaps と addToMap)を使用したので、あなたはそれらのファイルをあなたの C コードに含める必要があります。同様の必要性は zoo-kernel/service_internal.h に宣言された errorException 関数にもあります。また、あなたのサービスオブジェクトファイルをランタイムの際の errorException に使用される zoo-kernel/service_internal.o をリンクする必要があります。そして、libxml2 と OGR C-API にアクセスするために必要となるファイルもまた含めなければなりません。

シェアードライブラリに必要なものとして、extern "C"を宣言するブロックコード内にあなたのコードを格納しなければなりません。サービスプロバイダディレクトリ(/home/zoows/sources/zoo-services/ws_sp 内の)ルートに置く service.c ファイルの中に最終的なコードを格納されなければなりません。それはこのように見えます:

```
#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>
}
```

サービスの完全なソースは準備が出来たので、次にシェアードライブラリとしてコードをコンパイルした結果のシェアードオブジェクトに対応するサービスを生成する必要があります。これは次のコマンドを用いて行うことができます:

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

CFLAGS and LDFLAGS 環境変数は事前にセットする必要があることに注意してください。

CFLAGS はインクルードされたヘッダを探すために求められるパスおよび ogr_api.h, libxml2 directory, service.h と service_internal.h ファイルのディレクトリへのパスのすべてを含む必要があります。OSGeoLive 環境のおかげで、いくつかの添付されたツールを用いてそのような値を取り出すことができます:xml2-config と gdal-config どちらも --cflags 条件に用いられます。それらは要求通りのパスを生成します。

もし zoo-services 内のメインディレクトリに ZOO サービスのプロバイダーを作成する説明を続けるならば、あなたのカレントパス(/home/user/zoows/sources/zoo-services/ws_sp)に対して相対的な../../zoo-kernel ディレクトリにある ZOO カーネルとソースツリーを見ることができます。あなたのソースツリーを別の場所に移動させる場合でも全く同様のコマンドラインを使ったコードコンパイルを保持するために相対パスなのですが、ZOO カーネルのフルパスを使用することもできることに注意してください。コンパイラーが service.h と service_internal.h を見つけることが出来るようにするために CFLAGS に -I../../zoo-kernel を加える必要があります。

完全な CFLAGS 記述はこの通りです:

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

CFLAGS に正しくセットするインクルードパスを得たので、リンクに対応するライブラリ(LDFLAGS 環境変数として定義)に取り組みましょう。gdal と libxml2 ライブラリに対するリンクの場合、 上記と同じツールに --cflags のかわりに --libs を用いることができます。完全な LDFLAGS の記述はこの通りです:

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

次にコードのコンパイル時に手助けをする Makefile を作成します。ZOO サービスプロバイダディレクトリのルートに短い Makefile を書きましょう、下記の行を含みます:

```
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
```

この Makefile を使用して、ZOO サービスプロバイダディレクトリから make を実行すると、cgi-env に結果として ogr_ws_service_provider.zo というファイルが得られます。

メタデータファイルと ZOO サービスシェアードオブジェクトの両方が cgi-env ディレクトリにできました。新しい ServicesProvider を配置するために、ZOO カーネルがあるディレクトリ /usr/lib/cgi-bin に ZOO サービスシェアードオブジェクトと対応するメタファイルの両方をコピーしましょう。このタスクを実行するためには sudo コマンドを用いなければなりません:

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

これで、ZOO サービスのソースツリーの意味するところが解りましたね。cgi-env ディレクトリはあなたの新しいサービスまたはサービスプロバイダを、cgi-env のコンテンツを cgi-bin ディレクトリにコピーするという簡単な方法で 配置をするためのものです。

make install と直接タイプすることで、新しいサービスプロバイダを ZOO カーネルで利用可能にするために、Makefile に下記の行を追加することが出来ることに注意してください:

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

これで、 ZOO サービスプロバイダは、ZOO カーネルを通しての実行リクエストから利用される準備が出来ました。

Python バージョン

ZOO のサービスするプロバイダの実装に Python を使用するため、cgi-env ディレクトリにある ogr_ws_service_provider.py のすべてのコピーするコードは下記のとおりです。実際、Python のようなインタプリタ言語では、極めて簡単な配置ステップで作られるサービスの配置以外にはなにもコンパイルする必要がありません:

```
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"]["valu
e"])
    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
```

私たちは既に詳細を伝えていて、コードは同様な方法で作られているため、ここでは関数本体については述べません。

実行の前にすることは、cgi-env のファイルを cgi-bin にコピーすることだけです:

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

インストールセクションに対応した簡単な Makefile 次のように書かれます:

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

最後に、簡単に ZOO サービスプロバイダのメインディレクトリから make install を実行すると、ZOO プロバイダーが配置されます。

Execute リクエストを使用したテスト

単純かつ読みづらい方法

皆さんは、ご自分の ogr_ws_service_provider と呼ばれる、ZOO カーネルツリーに配置された ZOO サービスプロバイダとして格納された OGR Boundary サービスのコピーがあり、サービスをテストするために Execute リクエストを次の通りに使うことができます:

```
http://localhost/cgi-bin/zoo_loader.cgi?request=Execute&service=WPS&version=1.0.0&Identifier=Boundary&DataInputs=InputPolygon=Reference@xlink:href=http%3A%2F%2Flocalhost%2Fcgi-bin%2Fmapserv%3Fmap%3D%2Fvar%2Fwww%2Fwfs.map%26SERVICE%3DWFS%26REQUEST%3DGetFeature%26VERSION%3D1.0.0%26typename%3Dregions%26SRS%3DEPSG%3A4326%26FeatureID%3Dregions.3192
```

上記の URL に見られるように、DataInputs KVP 値内の xlink:href キーにある、OSGeoLive 上の MapServer WFS サーバーや、Reference にセットされる InputPolygon の値を URL エンコードされた WFS リクエストとして使用します。エンコードされていない WFS リクエストに対応したものは次の通りです:

http://localhost/cgi-bin/mapserv?map=/var/www/wfs.map&SERVICE=WFS&REQUEST=GetFeature&VERSION=1.0.0&typename=regions&SRS=EPSG:4326&featureid=regions.3192

サービスの実行に利用される入力値についての情報を得たい場合、lineage=true を付け加えることができることに注意してください。さらに、後で再利用するために ZOO サービスの ExecuteResponse ドキュメントを格納する必要があるかもしれません。このような場合、前のリクエストに storeExecuteResponse=true を追加してください。このパラメータ指定を true にしていない実行の時と比べて、ZOO カーネルの挙動は正確には同じでないということに注意してください。実際、このようなリクエストでは、ZOO カーネルは statusLocation の属性に対する ExecuteResponse を返し、それは実行中の結果あるいは最後に ExecuteResponse に置かれた情報をクライアントに与えます。

リクエスト中で storeExecuteResponse が true に設定された場合、ExecuteResponse がどのようになるかの例は次の通りです:

statusLocation にしたがって、以前にリクエストした時に得られたものと同じ ExecuteResponse が得られました。クライアントアプリケーションのキャッシングシステムを備えているということで大変便利になるということに注意してください。

以前のリクエストのすべての ResponseForm を指定しておらず、リクエストされなかったなら、作成した zcfg ファイルに定義された初期値で用いられる application/json mimeType に基づく ResponseDocument が返されます。しかしながら、クエリーの ResponseDocument パラメータに mimeType=text/xml 属性を加えることで、ZOO カーネルにどのような種類の結果が欲しいかを伝えることができます。以前のリクエストに対して、このパラメータを追加することで GML 表記の結果を得られます:

```
http://localhost/cgi-bin/zoo_loader.cgi?request=Execute&service=WPS&ve
rsion=1.0.0&Identifier=Boundary&DataInputs=InputPolygon=Reference@xlin
k:href=http%3A%2F%2Flocalhost%2Fcgi-bin%2Fmapserv%3Fmap%3D%2Fvar%2Fwww
%2Fwfs.map%26SERVICE%3DWFS%26REQUEST%3DGetFeature%26VERSION%3D1.0.0%26
```

```
typename%3Dregions%26SRS%3DEPSG%3A4326%26FeatureID%3Dregions.3192&Respo
nseDocument=Result@mimeType=text/xml
```

WPS 仕様による定義では、完全な ResponseDocument ではなくデータだけの RawDataOutput を問い合わせることができます。そうするためには、次に示されるリクエストのように、リクエストの ResponseDocument を RawDataOutput に置き換えるだけです。:

```
http://localhost/cgi-bin/zoo_loader.cgi?request=Execute&service=WPS&ve
rsion=1.0.0&Identifier=Boundary&DataInputs=InputPolygon=Reference@xlin
k:href=http%3A%2F%2Flocalhost%2Fcgi-bin%2Fmapserv%3Fmap%3D%2Fvar%2Fwww
%2Fwfs.map%26SERVICE%3DWFS%26REQUEST%3DGetFeature%26VERSION%3D1.0.0%26
typename%3Dregions%26SRS%3DEPSG%3A4326%26FeatureID%3Dregions.3192&RawD
ataOutput=Result@mimeType=application/json
```

最後に、このワークショップの次のセクションにあるクライアントアプリケーションの開発に使用するために、にこの種のリクエストを JSON 文字列形式で取得するよう mimeType の初期値を戻しておくことに注意してください。(※次のセクションでは mimeType=application/json を使用します)

リクエストの単純化と可読性

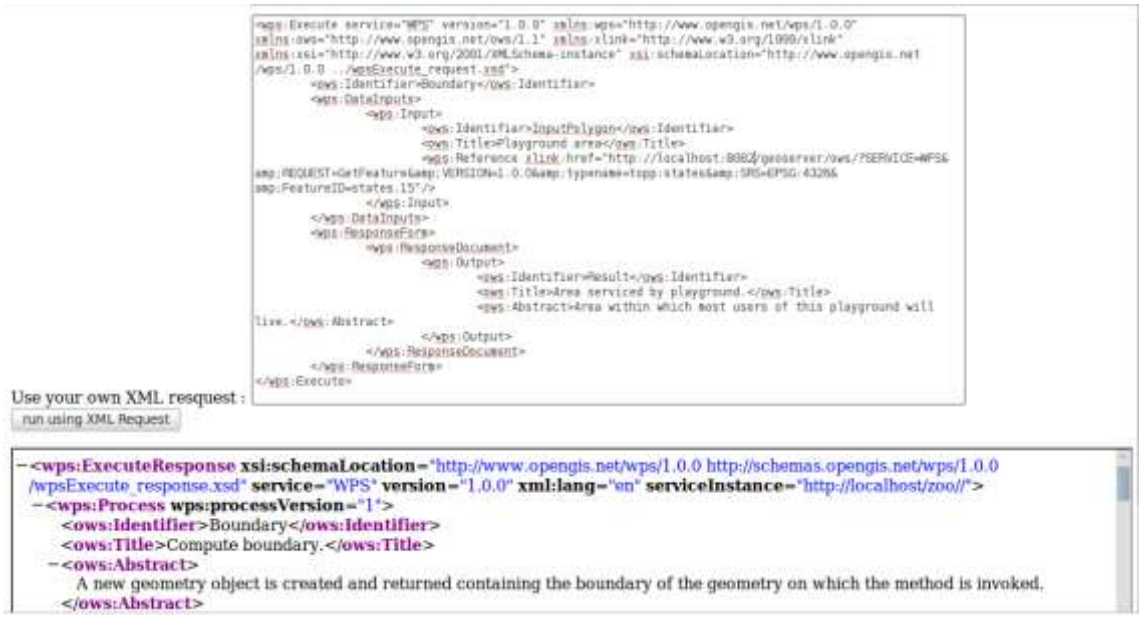
このワークショップのはじめから用いられた例をみると、複雑で長い URL を作成して GET に用いられる Execute リクエストを書くことは時により困難です。 次のリクエスト例では、そういうことですので、POST XML リクエストを用いましょう。初めに、以前の Execute で使用したリクエストに対応する XML は次のとおりです:

```
<wps:Execute service="WPS" version="1.0.0"
xmlns:wps="http://www.opengis.net/wps/1.0.0"
xmlns:ows="http://www.opengis.net/ows/1.1"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.opengis.net/wps/1.0.0 ../wpsExecute_req
uest.xsd">
 <ows:Identifier>Boundary</ows:Identifier>
 <wps:DataInputs>
  <wps:Input>
   <ows:Identifier>InputPolygon</ows:Identifier>
   <ows:Title>Playground area</ows:Title>
```

```
    <wps:Reference
xlink:href="http://localhost/cgi-bin/mapserv?map=/var/www/wfs.map&amp;
SERVICE=WFS&amp;REQUEST=GetFeature&amp;VERSION=1.0.0&amp;typename=regi
ons&amp;SRS=EPSG:4326&amp;featureid=regions.3192"/>
  </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>
```

XML リクエストを簡単に実行するために、test_services.html と呼ばれる HTML フォームを /var/www に準備しました。次に示すリンクを使用してそれにアクセスすることができます: http://localhost/test_services.html.

ブラウザでこのページを開いてください、テキストエリアフィールド内に簡単に XML リクエストを埋めて、« run using XML Request » 送信ボタンをクリックします。サービスが GET リクエストを使用して実行したときと同様に結果を得られます。スクリーンショットは、上記に示されたリクエストを含む HTML フォームとページの下部にある iframe に表示された ExecuteResponse を示します:



xlink:href の値はこのようなデータ入力を取り扱うもっとも簡単な方法です。もちろん、ジオメトリの完全な JSON 文字列を使う事もでき、それは次に示す XML リクエスト例のとおりです:

```
<wps:Execute service="WPS" version="1.0.0"
xmlns:wps="http://www.opengis.net/wps/1.0.0"
xmlns:ows="http://www.opengis.net/ows/1.1"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.opengis.net/wps/1.0.0 ../wpsExecute_req
uest.xsda">
 <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 ], [ -106.383041, 31.733763 ], [ -106.538971,
31.786198 ], [ -106.614441, 31.817728 ], [ -105.769730, 31.170780 ],
[ -105.998360, 31.393818 ] ] ], [ [ [ -94.913429, 29.257572 ], [ -94.767380,
```

```
29.342451 ], [ -94.748405, 29.319490 ], [ -95.105415, 29.096958 ],
[ -94.913429, 29.257572 ] ] ] ] }
    </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>
```

もし、すべてがうまくいったなら、引数として JSON ジオメトリの Boundary が受け渡され、サービスは GML と JSON の両方を入力データとして取り扱えるということです。先のリクエストで、私たちは、ComplexData ノードの属性に追加した mimeType に、入力データが text/xml mimeType ではなくて application/json 文字列を指定して受け渡される指定をしました。それは、以前に述べたように @mimeType=application/json を追加したのと同じことです。

3.3.2 その他の関数サービスの作成 (ConvexHull and Centroid)

Boudary の単純なサービスコードがありますので、全く同じ数の引数:ジオメトリをもつ ConvexHull と Centroid を簡単に追加することができます。実装の詳細と ConvexHull サービスの配置については、下記に示すたとおりです。 Centroid についても同様です。

C バージョン

まず次に示すコードを service.c ソースコードに追加しましょう:

```
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;
}
```

この新しいコードは、Boundary サービスと全く同じです。ただ一つ我々が変更するのは、OGR_G_ConvexHull 関数が呼ばれている行の部分です（前に使用した OGR_G_GetBoundary ではなく)。関数をまったくコピーペーストすることは良くありませんので、す

べての場合において、新しいサービスの関数本体の記述を行うもっと一般的な方法があります。次に示す常用関数は単純化のためのものです:

```
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;
}
```

次に、完全な関数名の代わりに myFunc と呼ばれる関数ポインタを使用します。Boundary サービスをこの方法で再実装する方法は次のとおりです:

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

service.c 内に定義された、この applyOne ローカル関数を用いて，その他のサービスを次のように定義できます:

```
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);
}
```

applyOne 関数の一般化により2つの新しいサービス : ConvexHull and Centroid が ZOO プロバイダに追加されます。

Centroid の前に、OGR_G_Centroid にあるように、返されるジオメトリオブジェクト以外にも既存のポリゴンをセットするジオメトリを入力値とする MY_OGR_Centroid 関数を定義する必要があります。マルチポリゴンの ConvexHull を確保するためでもあります。コードは次のとおりです:

```
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;
}
```

サービスを配置するため、 cgi-env ディレクトリから ConvexHull.zcfg と Centroid.zcfg として、Boundary.zcfg をコピーします。 次に、前にも行ったとおりの手順で、Execute リクエストの実行とテストのために、最初の行のサービス名を書き換える必要があります。Idenitifier の値を、実行するサービスのリクエストに対応するように、 ConvexHull または Centroid に書き換えます。

ここでは、GetCapabilities と DescribeProcess リクエストはメタデータを書き換えなかったので、中途半端な(Boundary の)情報を返すこと、.zcfg に修正した値をセットできることに注意してください。ところで、テストの目的で使用されるので、入力と出力の名前と、default/supported フォーマットも同じものが使われています。

Python バージョン

```
def ConvexHull(conf,inputs,outputs):
    if inputs["InputPolygon"]["mimeType"]=="application/json":

geometry=osgeo.ogr.CreateGeometryFromJson(inputs["InputPolygon"]["valu
e"])
    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
```

再度、Boundary の関数を簡単にコピーペーストして、Geometry 関数が呼ばれている行を変更します。やはり、C 言語で行ったように、もっと一般化して単純にする方法を教えましょう。

まずはじめに、各々のサービス関数で同様に用いられる InputPolygon ジオメトリの抽出にある最初のステップですが、まずはじめにその関数を作りましょう。 同様に出力値を埋め、他の関数に対

しても自動的に行われるよう定義しましょう。2つの関数 (extractInputs and outputResult) はつぎのとおりです:

```
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()
```

Boundary 関数のコードを、下記に示す関数定義により、最小にすることができます:

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

次に、ConvexHull and Centroid サービスの定義については下記コードになります:

```
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
```

Python においても。マルチポリゴンを取り扱う ConvexHull を使う必要があるということに注意してください。

C バージョンで説明したように、Boundary.zcfg を ConvexHull.zcfg と Centroid.zcfg の各々にコピーしてください。 そして、make install コマンドを使い、サービスプロバイダを再配置します。

3.3.3 バッファサービス作成

我々はバッファサービスを使用することができますが、他のサービスに比べ多くの引数を必要とします。実際コードは、境界や凸包、そして中心点生成サービスを実装する際に使用されるものと少々異なっているからです。

バッファサービスもまた、入力引数としてジオメトリを必要としますが、使用するのは BufferDistance のパラメータです。これにより、単純な整数値として BufferDistance の LitteralData ブロックを定義できるでしょう。そのような種類の入力値の読込処理は、以前使用したのと同じ機能が使用されます。

C バージョン

あなたが、最初の境界サービスのソースコードへ戻るならば、以下について非常に複雑になったとは気づかれないでしょう。実際、あなたは BufferDistance 引数のアクセスに加えて、OGR_G_Buffer が(OGR_G_GetBoundary の代わりに)呼び出されるべように変更するだけです。ここに、全ての lcode を示します:

```
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;
  }
```

新しいコードは、あなたの service.c ファイルに挿入し、再コンパイルの後に /usr/lib/cgi-bin/ ディレクトリにある ZOO サービスプロバイダーの旧バージョンと置き換えなければなりません。もちろん、対応する ZOO のメタデータファイルを同じディレクトリに置かなければなりません。

以前の説明にあったように、ZOO カーネルは、zcfg ファイルで定義されるよりも多くの引数を渡すことができます。ですから、Boundary.zcfg ファイルを Buffer.zcfg と改名したコピーを使用して、バッファ識別子を記載してみましょう。その際、以前も行ったように Execute リクエストを用いて

サービスをテストしてください。そうすると、ResponseDocument でバッファの結果を得られるでしょう。

BufferDistance 入力値がサービスをパスするかどうか上のコードをチェックすることに注意しましょう。 もし、そうでなければデフォルト値として 1 を使用するようにして下さい。前の入力値を使わないようにするためです。

リクエストに DataInputs 値を加えることによって、ジオメトリのバッファを計算するのにあなたのサービスで使用された BufferDistance 値を変更することができます。KVP シンタックスを使用することで、各 DataInputs が、セミコロンによって分割されることに注意してください。

以前のリクエスト:

```
DataInputs=InputPolygon=Reference@xlink:href=http%3A%2F%2Flocalhost%2Fcgi-bin%2Fmapserv%3FSERVICE%3DWFS%26REQUEST%3DGetFeature%26VERSION%3D1.0.0%26typename%3Dregions%26SRS%3DEPSG%3A4326%26FeatureID%3Dregions.3192
```

以下のように置き換えます:

```
DataInputs=InputPolygon=Reference@xlink:href=http%3A%2F%2Flocalhost%2Fcgi-bin%2Fmapserv%3FSERVICE%3DWFS%26REQUEST%3DGetFeature%26VERSION%3D1.0.0%26typename%3Dregions%26SRS%3DEPSG%3A4326%26FeatureID%3Dregions.3192;BufferDistance=2
```

BufferDistance 値を 2 に設定する場合、異なる結果が与えられるでしょう。なぜなら、ソースコード上で 1 をデフォルト値と定めたため、いかなるパラメータも渡されないからです。

ここで、 http://localhost/test_services.html の HTML フォームから使用する XML 形式における同じクエリを見つけられるでしょう:

```
<wps:Execute service="WPS" version="1.0.0"
xmlns:wps="http://www.opengis.net/wps/1.0.0"
xmlns:ows="http://www.opengis.net/ows/1.1"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.opengis.net/wps/1.0.0 ../wpsExecute_request.xsda">
 <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&amp;REQUEST=GetFeature&amp;VERSION=1.0.0&amp;typename=regi
ons&amp;SRS=EPSG:4326&amp;featureid=regions.3192"/>
  </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 バージョン

すでにユーティリティ機能の createGeometryFromWFS と outputResult を定義しているので、コードはこれと同じくらい簡単です:

```
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
```

ここでは、["BufferDistance"]["value"] のインプットを、ジオメトリインスタンスにおけるバッファ方法を引数として加えたのみです。一旦このコードを、ogr_ws_service_provider.py ファイルに追加した後で、ZOO カーネルのディレクトリの中に(もしくは、ZOO サービスプロバイダのルートディレクトリからインストールする形で)それをコピーしてください。また、次のセクションで Buffer.zcfg ファイルを詳述する必要があることに注意してください。

バッファ メタファイル

クライアントにサービスがこのパラメータをサポートすることを知らせるようにするため、 サービス･メタデータ･ファイルに BufferDistance を加えなければなりません。 そうした上で、あなたのオリジナルの Boundary.zcfg ファイルを Buffer.zcfg ファイルとしてコピーして、DataInputs のブロックに以下のラインを加えてください。

```
[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>
```

minOccurs は、パラメータの入力が任意であることを意味するために 0 が設定されているので、パスを通す必要はありません。 むしろ、ZOO カーネルは、任意のパラメータのためにデフォルト設定値とサービス機能にデフォルト値を通すことを知る必要があります。

下記のサイトで、Buffer.zcfg ファイルのフルコピーを取得できます。

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

以上で、あなたは、バッファサービスついて GetCapabilities や DescribeProcess そして Execute を ZOO カーネルにリクエストすることができます。

4. OpenLayers を使用した WPS クライアントの構築

次のステップでは OpenLayers map から作成した OGR Services に関連付けます。こうすることで、選択したポリゴン上で単一または複数の幾何処理を行い、新規作成したジオメトリを表示することができます。

4.1 WMS のデータセットを表示する簡単なマップの作成

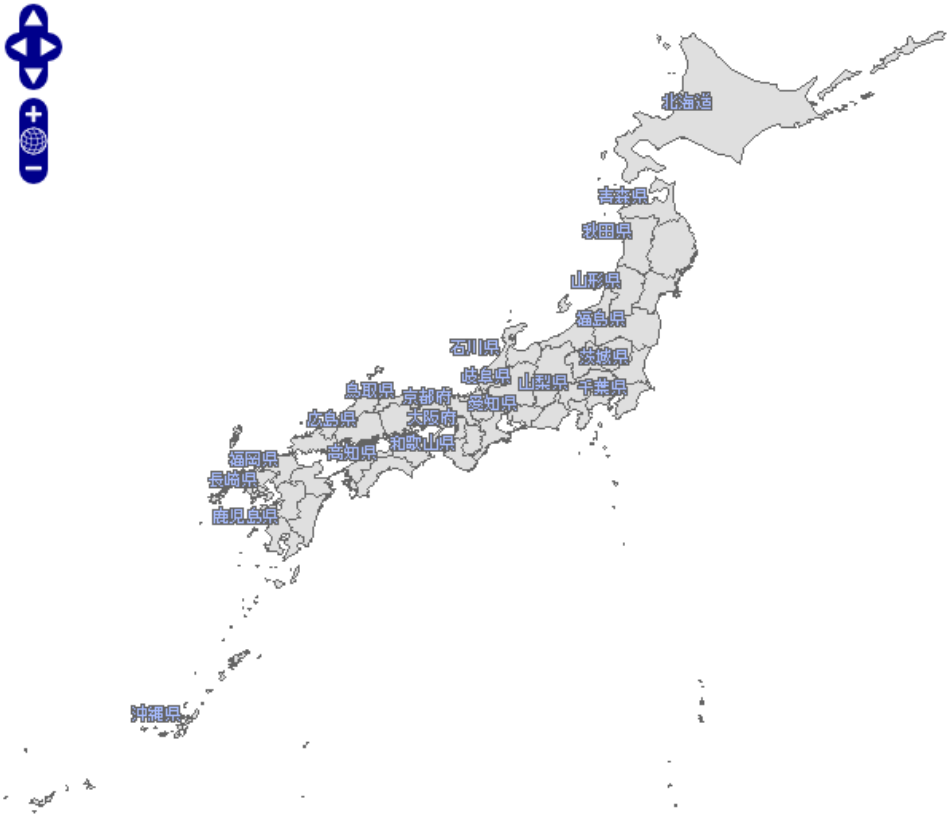
OpenLayers は OSGeoLive のデフォルトのディストリビューションにも含まれているため、便宜上これを使用します。好きなテキストエディタで /var/www/zoo-ogr.html を開き、次の HTML をペーストしてください:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"><html
xmlns="http://www.w3.org/1999/xhtml" xml:lang="EN" lang="EN">
 <head>
  <meta content="text/html; charset=UTF-8" http-equiv="content-type"/>
  <title>ZOO WPS Client example</title>
  <style>
  #map{width: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>
```

次の JavaScript コードを<head>内の <script></script>} セクションに追加します。すると WMS として日本の領域を表示するデータが設定されます。

```
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);
}
```

終了したら HTML ファイルを zoo-ogr.html としてワークショップディレクトリに保存し、/var/www にコピーします。好きなウェブブラウザでリンク/var/www http://localhost/zoo-ogr.html. を開き、これを使って可視化してください。WMS レイヤーが選択されている、USA を中心とするマップを手に入れることができます。



4.2 WFS のデータレイヤー取得と選択コントロールの追加

WFS を通じて表示されているデータにアクセスする前に、先ずはこれから作成するいくつもの相互作用をすることを目的とするに新規ベクトルレイヤーを作成しなければなりません。init() 関数内に次のラインを追加します。map.addLayers メソッドに新規作成したレイヤーを忘れず追加して下さい:

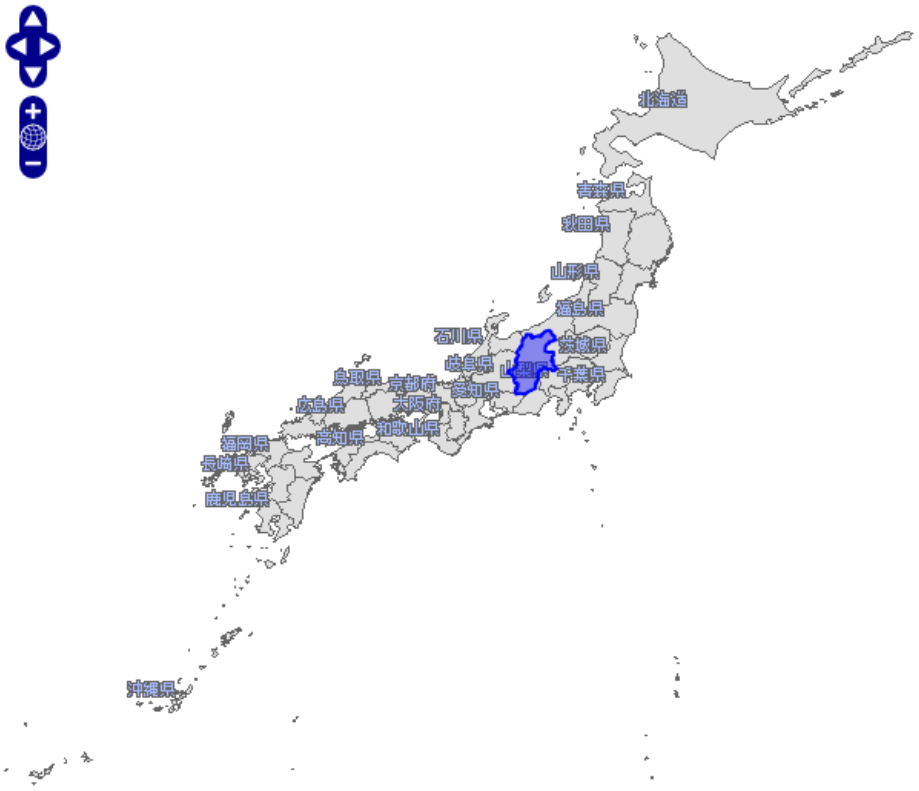
```
select = new OpenLayers.Layer.Vector("Selection", {
  styleMap: new
OpenLayers.Style(OpenLayers.Feature.Vector.style["select"])
});
```

```
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]);
```

次に、下記ようにポリゴンを選択する新規コントロールを作成することによって t データにアクセスすることが可能になります。OpenLayers.Protocol.WFS.fromWMSLayer(layer)はジオメトリへのアクセスに使用され、そして選択の状況は表示され制御変数に追加されるということにご注意ください。

```
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();
```

HTM ファイルを再度保存して下さい。ポリゴン上でクリックすると選択することができます。選択されたポリゴンは青色で表示されます。



4.3 JavaScript から呼び出される単一ジオメトリサービス

これで全て設定できたので、JavaScript で OGR ZOO サービスを呼び出してみましょう。init() 関数のあとに次の行を追加してください。一つのジオメトリ処理を呼び出します。fid 値として返された不要な空白を削除する特定の parseMapServerId 関数を使うことに気づくでしょう。

```
function parseMapServerId(){
  var sf=arguments[0].split(".");
  return sf[0]+"."+sf[1].replace(/ /g,'');
}

function simpleProcessing(aProcess) {
```

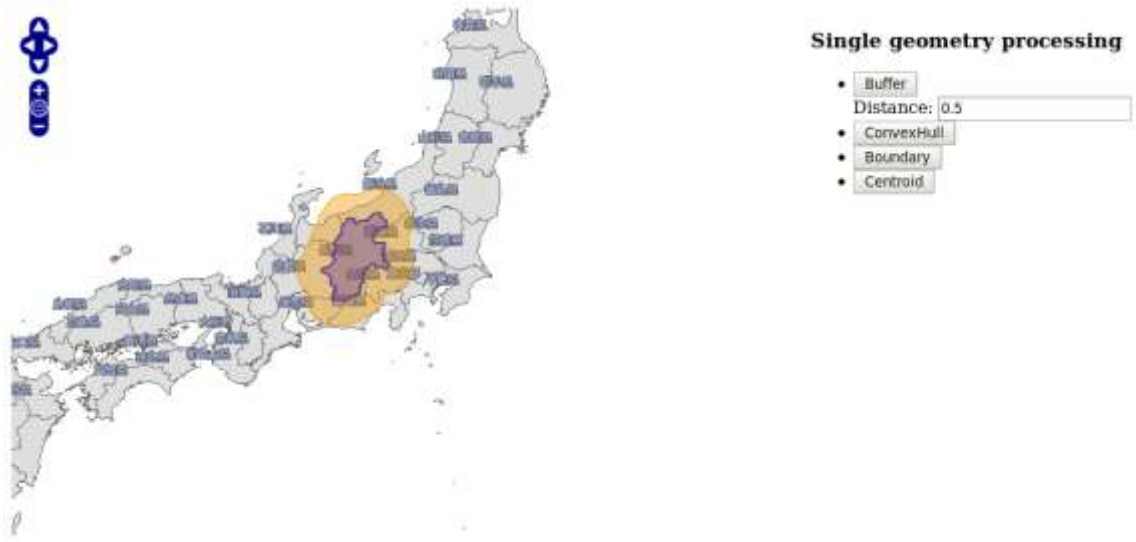
```
  if (select.features.length == 0)
    return alert("No feature selected!");
  if(multi.features.length>=1)
        multi.removeFeatures(multi.features);
  var url =
'/cgi-bin/zoo_loader.cgi?request=Execute&service=WPS&version=1.0.0&';
  if (aProcess == 'Buffer') {
    var dist =
document.getElementById('bufferDist')?document.getElementById('bufferD
ist').value:'1';
    if (isNaN(dist)) return alert("Distance is not a Number!");

url+='Identifier=Buffer&DataInputs=BufferDistance='+dist+'@datatype=in
terger;InputPolygon=Reference@xlink:href=';
  } else
    url +=
'Identifier='+aProcess+'&DataInputs=InputPolygon=Reference@xlink:href=
';
  var xlink = control.protocol.url
+"&SERVICE=WFS&REQUEST=GetFeature&VERSION=1.0.0";
  xlink += '&typename='+control.protocol.featurePrefix;
  xlink += ':'+control.protocol.featureType;
  xlink += '&SRS='+control.protocol.srsName;
  xlink += '&FeatureID='+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();
}
```

次に、私たちがちょうど示した異なる処理を呼び出すには、いくつかの特定のボタンを HTML に追加しなければなりません。 の前に次の行を書き込んでマップの上にそれらを追加します。

```
<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>
```

再び HTML ファイルを保存してください。これでボタンの一つをクリックして、ポリゴンを選択し、Buffer、ConvexHull、Boundary あるいは Centroid を起動できます。処理の結果はマップ上に GeoJSON レイヤーとしてオレンジ色で表示されます。



4.4 JavaScript から呼び出される複合ジオメトリサービス

同じテクニックを使って、複数のジオメトリ処理のために作られている関数を書き込むことができます。simpleProcessing() 関数のあとに次の行を追加してください。このような関数を作成する方法はセクション 5 のエクササイズで説明します。

```
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/1.0.0"
xmlns:ows="http://www.opengis.net/ows/1.1"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.opengis.net/wps/1.0.0/../wpsExecute_req
uest.xsd">';
    params += '<ows:Identifier>'+aProcess+'</ows:Identifier>';
    params += '<wps:DataInputs>';
    params += '<wps:Input>';
    params += '<ows:Identifier>InputEntity1</ows:Identifier>';
    params += '<wps:Reference
xlink:href="'+xlink.replace(/&/gi,'&amp;')+'"/>';
    params += '</wps:Input>';
    params += '<wps:Input>';
    params += '<ows:Identifier>InputEntity2</ows:Identifier>';
    params += '<wps:Data>';
    params += '<wps:ComplexData mimeType="application/json">
'+GeoJSON.write(hover.features[0].geometry)+' </wps:ComplexData>';
    params += '</wps:Data>';
    params += '</wps:Input>';
    params += '</wps:DataInputs>';
    params += '<wps:ResponseForm>';
    params += '<wps:RawDataOutput>';
    params += '<ows:Identifier>Result</ows:Identifier>';
    params += '</wps:RawDataOutput>';
    params += '</wps:ResponseForm>';
    params += '</wps:Execute>';
  } catch(e) {
    alert(e);
    return false;
  }
  var request = new OpenLayers.Request.XMLHttpRequest();
  request.open('POST',url,true);
  request.setRequestHeader('Content-Type','text/xml');
  request.onreadystatechange = function() {
```

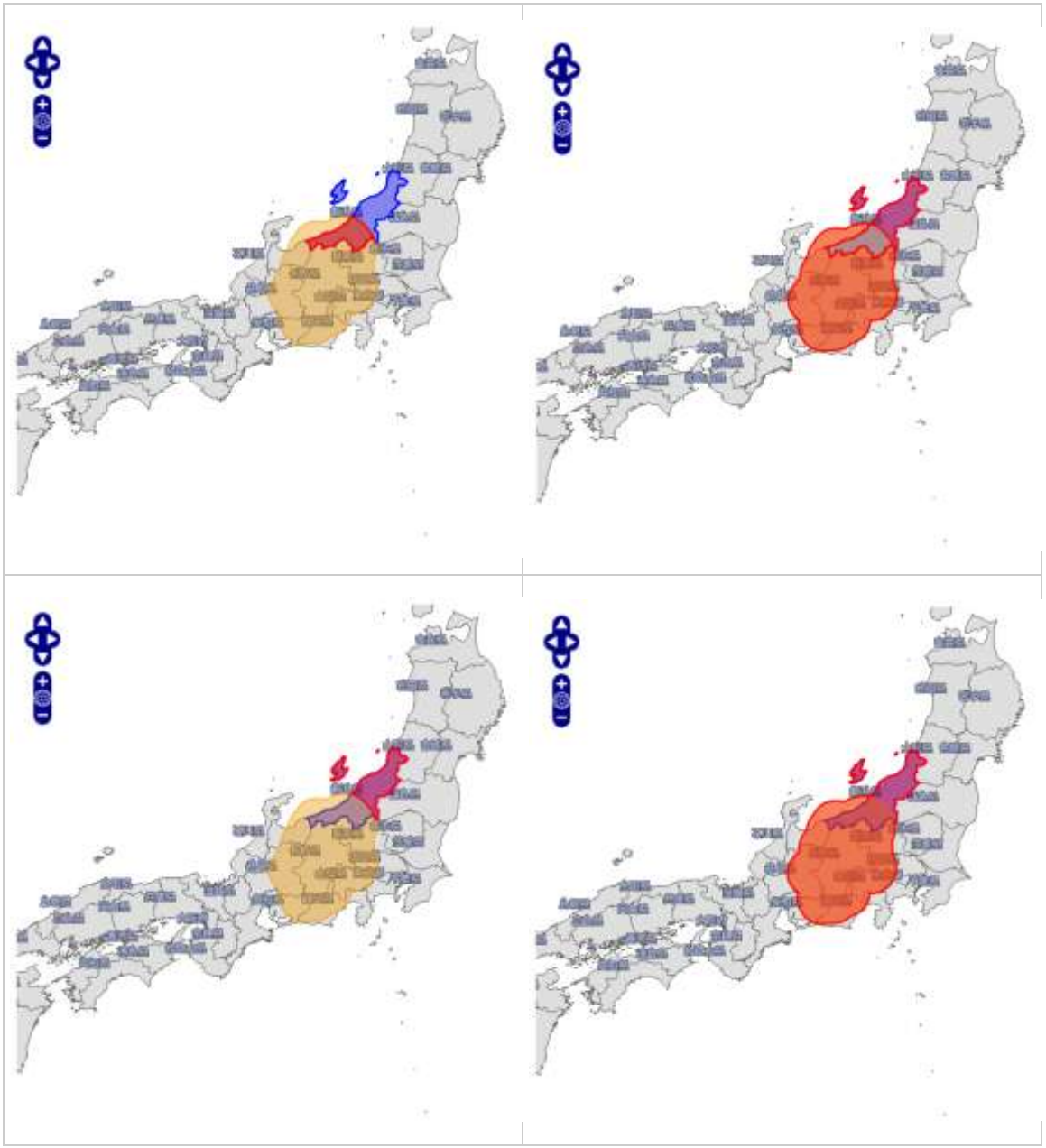
```
    if(request.readyState == OpenLayers.Request.XMLHttpRequest.DONE) {
      var GeoJSON = new OpenLayers.Format.GeoJSON();
      var features = GeoJSON.read(request.responseText);
      multi.removeFeatures(multi.features);
      multi.addFeatures(features);
    }
  }
  request.send(params);
}
```

今回は、ZOO Kernel を呼び出すのに GET メソッドではなく XML POST を使用していることに注意してください。GET メソッドを使用すると、リクエストの長さに基づく HTTP GET メソッド制限のためにエラーが表示されます。ジオメトリを表す JSON ストリングの使用すると、リクエストは長くなります。

一度複数のジオメトリ処理を呼び出す関数を得ると、リクエストの呼び出しを消すためにいくつかボタンを追加しなければなりません。これは現在の zoo-ogr.html ファイルに追加するための HTML コードです:

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

ページをリロードしてください。すると、複数のジオメトリサービスを実行でき、次のスクリーンショットのように赤色で表示された結果を得ることができます:



同じ結果を得るには、Services Provider に何かが足りないように思われます ... 複数のジオメトリサービスです! これは次のセクションで行います。

5. 演習

以上で、zcfg メタデータファイルの記述方法と、 C 言語または Python 言語の選択によって、service.c または ogr_service_provider.py の短いコードが得られました。 この演習のゴールは、次のような複合ジオメトリサービスを実装することです ：

- Intersection
- Union
- Difference
- SymDifference

5.1 C バージョン

source.c ファイルの編集を選択された方は、このワークショップ演習を通して、追加した複合ジオメトリは、次の OGR C-API 関数を使用しています:

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

Boundary.zcfg ファイルを例に、InputPolygon を InputEntity1 にして、同様に IntputEntity2 を追加することができます。ZOO メタデータファイルのそのほかの値についても、適当なメタデータ情報を設定してください。

5.2 Python バージョン

ogr_ws_service_provider.py ファイルの編集を選択された方は、このワークショップ演習を通して、追加した複合ジオメトリは、次の Geometry インスタンスに適用された osgeo.ogrGeometry 関数を使用しています:

- Intersection(Geometry)
- Union(Geometry)

- Difference(Geometry)
- SymmetricDifference(Geometry)

Boundary.zcfg ファイルを例に、InputPolygon を InputEntity1 にして、同様に IntputEntity2 を追加することができます。ZOO メタデータファイルのそのほかの値についても、適当なメタデータ情報を設定してください。

5.3 サービスのテスト

ここで、複合ジオメトリサービスは、ローカル環境の上に配置されました。ブラウザ上から、前のセクションで作成した zoo-ogr.html をリロードしてください。そして新しい ZOO サービスをテストしてください。