Technical Documentation

Web Mapping Application Prototype Development

AAFC - Science and Technology

Winter 2017

Overview

The purpose of this documentation is to outline, record, and support the development of the soil erosion web mapping application prototype undertaken by Andrew Roberts and Anne Hagerman, Algonquin co-op 2016-2017. Research and development of the prototype has been broken down into sections, where functional elements have been explored and documented. Each Feature section of this document provides an outline of requirements, objectives, and procedure concerning a stage of functionality development for the prototype. Titles correspond and are linked to files located on Daedalus server, where Anne or Andrew Workspace serve as the primary directories.

Scope

The purpose of the Soil Erosion Web Map Application is to make current and accurate erosion modeling information easily and readily available to a wide audience, including but not limited to farmers, planners, engineers, and policy makers. The application will do this by integrating current data available through WMS and WCS services, as well as Canadian Land Inventory (C.L.I) soil data in a publicly facing web app. This application will allow for the capture, processing, and serving of data specific to an area of interest (AOI) where users can select or upload an AOI, run a process using server side GeoSpatial models, then see or download that processed information.

Team - HR

Lead Supervision - Xiaoyuan Geng, Head Soil Scientist at Landscape Analysis and Applications
Tech Guru - Tim Martin, Sustainability Metrics
Developer - Andrew Roberts, Algonquin co-op student
Developer - Anne Hagerman, Algonquin co-op student

Abbreviations

OL	**OpenLayers**
OSM	**Open Street Map**
WMS	**Web Mapping Service**
WCS	**Web Coverage Service**
WPS	**Web Processing Service**
AOI	**Area Of Interest**

Standards

Standards - OGC data standards, open sourced software that conforms to use and production under those standards. Download formats meet OGC standards.

Environment

Development of the prototype was done on Ubuntu v16.04 LTS through an OracleVM VirtualBox v5.1.6 virtual machine. Apache Web Server v2.4.18 was the primary web server. Java Web Server Tomcat v9.0.0 was installed and sat on Apache to handle the Java requirements of GeoServer v2.9.1, an open source Java based server designed for GeoSpatial data sharing and editing. PHP v7.0.8 was used to create dynamic HTML pages which were written using the open sourced editor Brackets v1.8. Client side web mapping was achieved using a local download of OpenLayers v3.19.1.

Early features were created locally on the virtual machines; each machine hosting the web servers, all scripts and files, and the spatial data served through Geoserver. Later development stages called for a separation of functionality and the prototype was moved to two servers, Ulysses and Daedalus. Ulysses hosts geospatial data served through GeoServer services, while Daedalus hosts the application data (openlayers, php, html/css).

Architecture

The basic architecture of the application is client -> server -> data-server (see Figure 1 below). The client interacts with the application on the front end - html, css, and javascript handle all interactions. This includes interacting with the map itself, such as drawing objects or pan/zoom. The client can make requests directly to the data-server, Ulysses, to add data directly to the map window so the user can see it. This is done by setting an OL layer source as a feature service from GeoServer using a Ulysses URL. This is light processing and a built in foundation to OL.

The server, Daedalus, is setup to handle all (mostly) data processing. User selected information is sent to Daedalus from the client using PHP. Daedalus can make requests directly to Ulysses, retrieving raw data through GeoServer that can then be included in heavier geoprocessing models. This is the workhorse.

The data-server, Ulysses, houses the data and makes it available through GeoServer. Some light processing can be done on Ulysses through a WPS request via GeoServer, but the main purpose is to provide online access to its geospatial data stores through GeoServer.

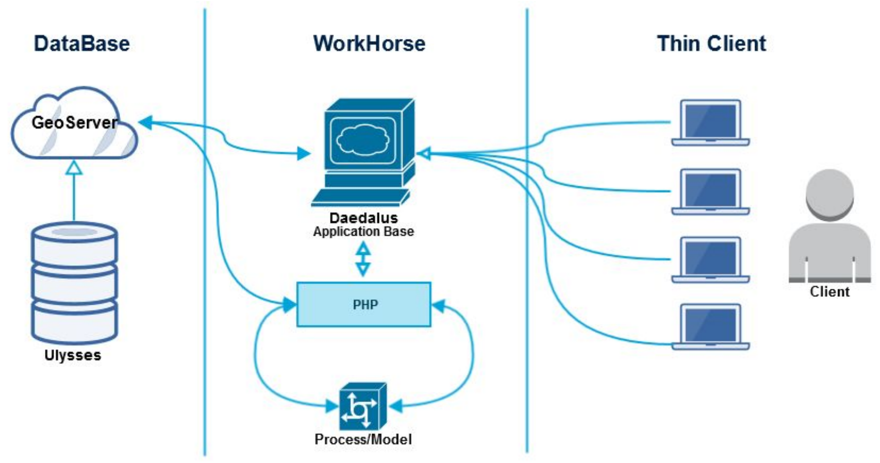


Figure 1. Three tiered platform

Feature Development

GeoServer Structure

GeoServer data management requires the creation of Workspaces, Stores, Layers or Layer Groups, and Styles. A Workspace functions the same way as a namespace, acting as a container that allows for the organization of other features. Different layers can have the same name if they fall under a different workspace. Stores connect to a data source that contain raster or vector data, while layers are either a raster or vector dataset representing a collection of geographic features. For raster datasets stores can only point to one single dataset, served a single layer in that store.

For local prototype development GeoServer sat on the local virtual machine, pointing to the locally stored data. Canadian and provincial raster datasets were served through a WCS, WMS, WPS enabled Canada Workspace. The rasters are WGS84 subsets of the global SoilGrid soil data, found on Ulysses. Unless otherwise stated, this description falls to all GeoServer data used in the local prototype development.

For more information on GeoServer Data Management, follow this link

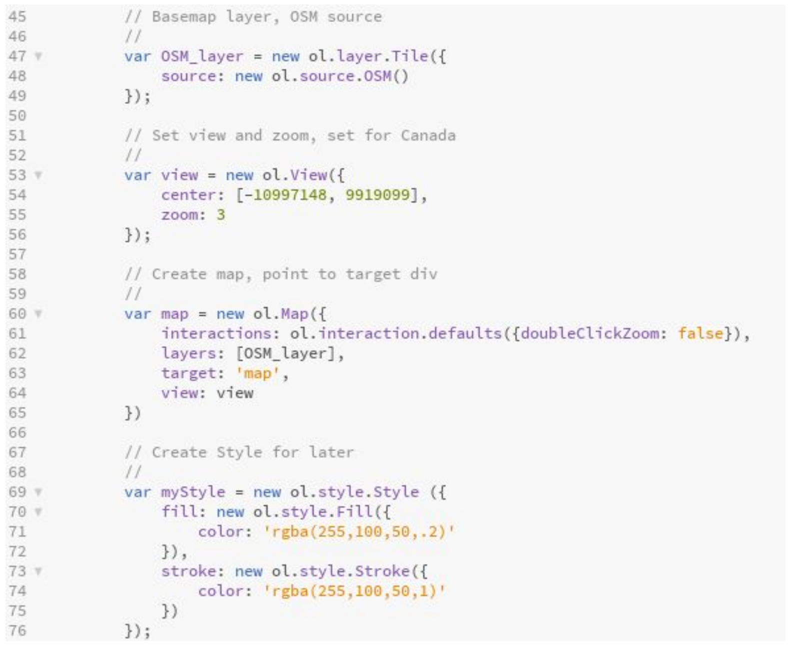
Prototype Data

HTML - JavaScript Structure

Unless otherwise specified, all html files using the OpenLayers (OL) library will contain the same map page configuration. Relative script and stylesheet tags point to the local OL install, the Google api jQuery library, basic javascript template functions, and basic CSS styling. Example specific changes to JavaScript functions are included in the HTML file. Example specific changes to CSS styling are made in a separate, overriding CSS sheet.

OpenLayers Structure

BaseMap, View, Style, and Map element are all declared at the top of each <body> -> <script> tag. OL map element is given the target div 'map'. This base template can be assumed for all future html files and can be found under Templates directory on Daedalus.. View will be configured to suit each feature. (See figure 1.)
Detailed comments are included in source code.



```
        // Basemap layer, OSM source
        //
        var OSM_layer = new ol.layer.Tile({
            source: new ol.source.OSM()
        });

        // Set view and zoom, set for Canada
        //
        var view = new ol.View({
            center: [-10997148, 9919099],
            zoom: 3
        });

        // Create map, point to target div
        //
        var map = new ol.Map({
            interactions: ol.interaction.defaults({doubleClickZoom: false}),
            layers: [OSM_layer],
            target: 'map',
            view: view
        })

        // Create Style for later
        //
        var myStyle = new ol.style.Style ({
            fill: new ol.style.Fill({
                color: 'rgba(255,100,50,.2)'
            }),
            stroke: new ol.style.Stroke({
                color: 'rgba(255,100,50,1)'
            })
        });
```

Figure 1. OL code template

PHP Structure

PHP 7.0 was installed as per the setup and configuration documentation. Supplemental modules included json, xml, curl, and zip. All directories requiring php access or execution must be assigned www:data as user. Permissions for those directories were set at 777, permitting read, write, and execution rights to all users. There are obvious security concerns with this level of privilege, but within the scope of this project those concerns were acceptable.

AddLayer_POST

Objective

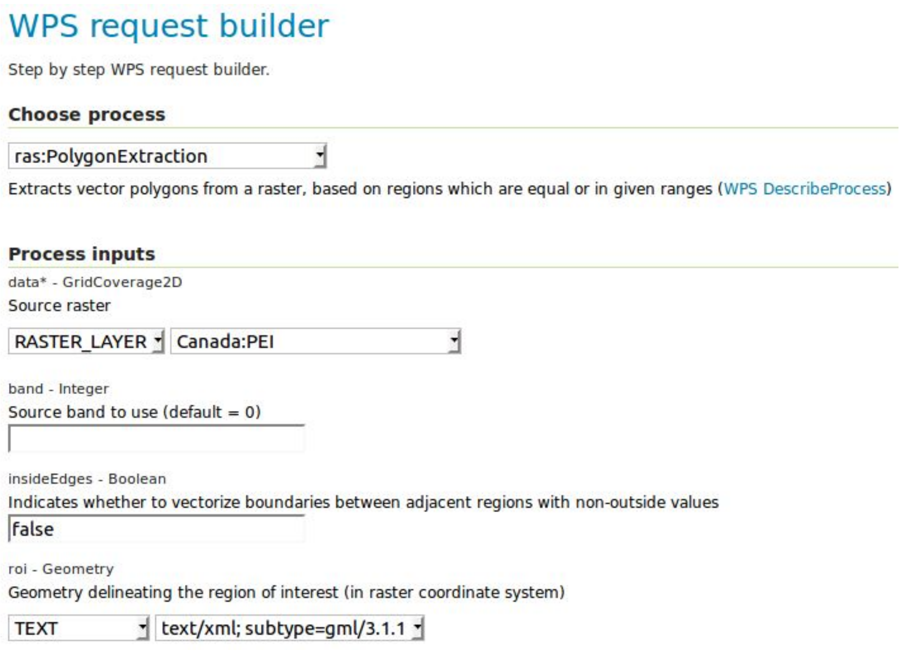
The goal of this exercise was to utilize one of Geoservers built in web processing services (WPS), PolygonExtraction, to extract a polygon of PEI's boundary extents and draw this polygon to the OL map.

Requirements

WPS extension must be installed on GeoServer to run this feature. Detailed instructions to do so can be found in the Setup and Configuration documents, section 7.9 Enabling Web Processing Service.

Procedure

A map onclick function is added to code, triggering a POST XMLHttpRequest() to Geoserver containing a WPS XML request. The PolygonExtraction request is not dynamic, responding only to the doubleclick and sending a predefined XML string. This string is acquired from the WPS Request Builder on the GeoServer Web Admin interface, found through GeoServer Welcome -> Demos -> WPS Request Builder. See Figure 2.



WPS request builder

Step by step WPS request builder.

Choose process

ras:PolygonExtraction

Extracts vector polygons from a raster, based on regions which are equal or in given ranges (WPS DescribeProcess)

Process inputs

data* - GridCoverage2D
Source raster
RASTER_LAYER | Canada:PEI

band - Integer
Source band to use (default = 0)

insideEdges - Boolean
Indicates whether to vectorize boundaries between adjacent regions with non-outside values
false

roi - Geometry
Geometry delineating the region of interest (in raster coordinate system)
TEXT | text/xml; subtype=gml/3.1.1

Figure 2. GeoServer WPS Request Builder

WPSs can handle a variety of responses. In this feature we request GeoJSON format, which is easily parsed using javascript's JSON.parse() function ->
var xmlDoc = JSON.parse(xhttp.responseText);

It's worth noting here that the xhttp response property must match the format of the response being received. If requesting an XML doc, *xhttp.responseXML* will be used. If GeoJSON response is requested, as in this example, *xhttp.responseText* will be used. In order for OL to read the received features into the appropriate format, a new GeoJSON format object is created -> *var format = new ol.format.GeoJSON();*

Features are read from the parsed JSON object into a new OL Features object following the defined format. Although redundant in this example, specifying the transformation from data to feature projections is a key step in rendering the map object. If data and map projection don't match, errors will be thrown which cannot be viewed in the returned XML OR in the browser debug console. These features are added to a new layer, and the layer is added to the map.

```
var features = format.readFeatures(xmlDoc,{
          featureProjection: 'EPSG:4326',
         dataProjection: 'EPSG: 4326'
 });
 var vector = new ol.layer.Vector({
         source: new ol.source.Vector({
         format: format
         }),
                  style: myStyle
 });
 vector.getSource().addFeatures(features);
 map.addLayer(vector);
```

DragBox Popup

Objective

This feature contains two main purposes. First is to implement a dragbox interaction on the map object and use the selected AOI to generate an image with a dynamic WMS request. The second is to implement a pixel value fetch with a single click event.

Requirements

WMS enabled data access for Canadian soils data was required.

Procedure

- Image Generation With WPS Request

A DragBox interaction is added to the code and the interaction is added to the map object -> *map.addInteraction(dragBox)*. The dragbox interaction has a *boxend* property which is triggered when the dragbox drawing is finished (control key released), on which we retrieve the geometry, then the extent (the bounding box), from the drawn dragbox.

```
dragBox.on('boxend', function(e) {
        var extent = dragBox.getGeometry().getExtent();
        var bbox = extent.toString();
```

After converting the extent array to a string and assigning that to the variable *bbox*, we simply include the variable in the location of a WMS GetMap URL the defines the bounding box of the requested map. This URL defines the layer, style, height and width, coordinate system, and the return format of the requested map. Here, we request a PNG as it will be included in a simple image tag in our application page. All the output formats can be found here in the GeoServer documentation. For our purposes, PNG was sufficient enough to demonstrate the capabilities.

* It is important to note that the requested coordinate system, srs or EPSG: code, must match that of your map object. Errors are difficult to identify when resulting from mismatched projections.

- Pixel Value Fetch With Single Click

This example uses the *GetFeatureInfo()* operation to retrieve, on a single click event *map.on('click', function {})* the feature info of layers at the click location, in this case the Canadian soil data pixel value. View projection and resolution are calculated and included in a getFeatureInfoURL request, defining the format as text/javascript.

```
map.on('click', function(evt) {
        window.url = wms_layer.getSource().getGetFeatureInfoUrl(
                evt.coordinate,
                viewResolution,
                viewProjection,
                {'INFO_FORMAT': 'text/javascript'}
```

If the url returned true, an AJAX request was sent using this URL and defining the dataType as jsonp. This step was a workaround for the Same Origin Policy, a concept in the web application security model, which threw a 'Cross Origin Request Blocked' error when attempting to parse and insert the URL feature data. The URL can be freely inserted into images or iframes, but AJAXing the request data into info divs is not permitted.

JSONP dataType is not enabled by default. Steps to enable the data type can be found in the Setup and Configuration Documents, under section 7.10 Enabling JSONP.

Once enabled, the response is parsed and AJAXed into the pixelValue div

```
if (url) {
    var parser = new ol.format.GeoJSON();
    $.ajax({
        url: url,
        dataType: 'jsonp',
        jsonpCallback: 'parseResponse'
    }).then(function(response) {
        var result = parser.readFeatures(response);
        container.innerHTML = result[0].T.GRAY_INDEX;
   });
 }
```

* Note that section 7.7 of the Setup and Configuration Documents outlines the procedure for allowing Cross Origin Resource Sharing from GeoServer, negating the use of JSONP. This is a more sustainable solution, particularly on a public (theoretically) server.

GeoTIFF_Save

Objective

The purpose of this feature is to demonstrate the flow of client/server/data-server processing using PHP. To do this, two main functions are performed. First, a map interaction extent object's coordinates are sent to php, processed, and AJAXed back into the client page. Second, the extent coordinates are dynamically added to an XML POST using PHP to GeoServers WCS requesting a GeoTIFF of Canada, cookie cut using the extent coordinates. The returned data is validated in PHP and either an error message or a download option is sent to the client.

Requirements

- WCS enabled data access for Canadian soils data was required, enabled via GeoServer admin login.
- PHP must be installed and configured on the workhorse server; Daedalus, or for local setup, the local machine.
- PHP www-data user must have read/write/execute permissions to all folders requiring access.

Procedure

- Processing The Extent Coordinates

As with the DragBox feature, an Extent interaction is created in code then added to the map object, under the condition that it is only triggered while using a platformModifierKey, or the control key ->

```
var extent = new ol.interaction.Extent({
        condition: ol.events.condition.platformModifierKeyOnly
});
map.addInteraction(extent);
extent.setActive(false);
```

The inactive extent interaction is added to the map, and on a *"keydown"* event listener setActive is set to *true*. Control + left click drag now creates an interactive extent object in the map window. Reprojection of the extent coordinates is performed when the *"keyup"* event listener 'hears' the control key release. At this point, the projection code for the layer the extent is drawn on (in this case the OSM basemap) is retrieved and used in an extent transformation function. The product of this function is transformed coordinates for the extent object, from Web Mercator to WGS84, assigned to a new variable, *newCoord*. See code ->

```
var newCoord;
var count = 0;
this.addEventListener('keyup', function(event) {
        if (event.keyCode == 16) {
                extent.setActive(false);
                var oldCode = OSM_layer.getSource().getProjection().getCode();
                 newCoord = ol.proj.transformExtent([extent.j[0], extent.j[1], extent.j[2],
                extent.j[3]], oldCode, 'EPSG:4326' );
                if (count >= 1) {
                        openSelectionTab();
                }
                count += 1;
        }
});
```

From here, the new coordinates are individually assigned to variables and sent to PHP via an AJAX POST request using jQuery. The POST request will be detailed later in the AJAX section. Once passed into the PHP file, the coords are checked for security. This is done with a simple custom *secure_input()* function, removing any characters and formatting that could harbor malicious script ->

```
function secure_input($data) {
        $data = trim($data);
        $data = stripslashes($data);
        $data = htmlspecialchars($data);
        return $data;
}
```

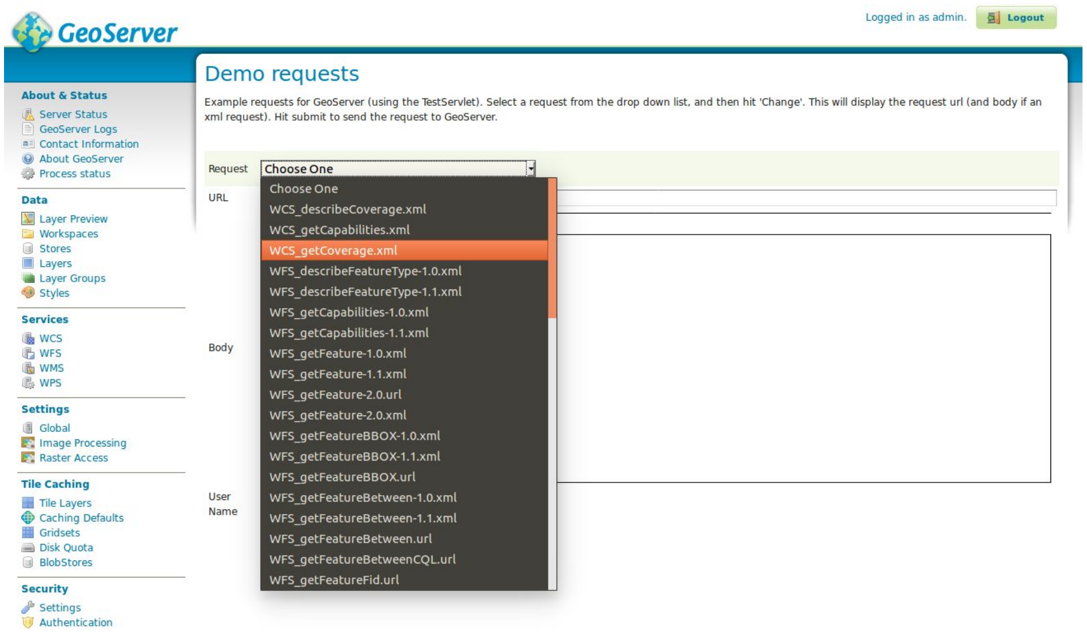
From here we simply add the coordinates to an array, trim the decimal places to 4, and pass the array to function *writeToFile()* the will write them to file using a date/time-stamp file name. The function returns either 'success' for 'fail', depending on the f*write()* status. Below we see the securing, formatting, and writing to file. *$_REQUEST([""])* is a PHP constant for capturing data sent to PHP script. The *writeToFIle()* output is passed to a variable which is then sent to the client to indicate the status of the file save ->

```
$minX = secure_input($_REQUEST["minX"]);
$minY = secure_input($_REQUEST["minY"]);
$maxX = secure_input($_REQUEST["maxX"]);
$maxY = secure_input($_REQUEST["maxY"]);

$coords = array($minX, $minY, $maxX, $maxY);
$length = count($coords);
for ($x = 0; $x < $length; $x++) {
$coords[$x] = number_format((float)$coords[$x], 4, '.', '');
}
$fileStatus = writeToFile($coords, $errMsg);
```

- Requesting GeoTIFF and Saving To File

Using the captured and secured min/max variables seen above, we dynamically add them to a pre-formatted XML WCS request. This XML request is generated from Demo Requests through the GeoServer Web Interface, found through GeoServer Welcome -> Demos -> Demo Requests -> WCS GetCoverage



Demo requests

Example requests for GeoServer (using the TestServlet). Select a request from the drop down list, and then hit 'Change'. This will display the request url (and body if an xml request). Hit submit to send the request to GeoServer.

This can be used as a basic WCS template, changing values like EPSG code or layer where necessary. This XML string is passed to the variable *$post* and cleaned for illegal characters. Next, a cURL session is initialized with *curl_init()*. Once headers and options are set and *$post* data is included, we perform the cURL session and capture the result in a variable -> *$ch_result = curl_exec()*. See [PHP - cURL Documentation] for details on cURL.

If the data sent to GeoServer is well formed and valid, the return from *curl_exec()* will be a GeoTIFF, as requested. If there is an error, an XML error report is returned. If we write the cURL response data to file without checking it we risk sending the client a download link to a corrupted file. To prevent this, we open a *new XMLReader()* to see if we can parse the response. If we can, we have received an error message and we send of an error report. If we can't, we have received a GeoTIFF and it's safe to write this to file ->

```
$reader = new XMLReader();
$reader->xml($ch_result);

if($reader->read()){
        $errMsg = "error";
        file_put_contents("error/errorlog.txt", $errMsg, FILE_APPEND);
        return("*GeoTiff save failed - Contact Server Administrator");
}else{
        $fileName = "../PHP/temp/geotiff". date("Y-m-dThisa") . ".tif";
        file_put_contents($fileName, $ch_result);
        return($fileName);
}
```

** Note, no error check is done on the cURL connection to GeoServer since this example is hosted locally and both the web app and GeoServer are dependant on the same server being alive. Error check on cURL must be done when data-server and application server are separate.

- AJAX

With both of this examples functions (requesting GeoTIFF and processing extent coordinates) the PHP script is setup to perform a function, and return either an error or a success message with a bit of data. These products are echoed as a json element back to the client via an AJAX response to the initial extent POST. Back on the client side, we simply parse this json response and insert the elements into the client html. Using jQuery's AJAX function allows us to utilize the *.fail()* callback and protect against server failure. In this example the POST method uses 3 callbacks to nicely handle the request; *.post()*, *.done()*, and *.fail()*. All of this is include in a jQuery button onclick function seen here (callback content is omitted for brevity. See source code for the full code) ->

```
$("#button-submit").click(function(){
        if (newCoord == undefined){
                <handle extent error>
        }else{
                <do something before you post>
                $.post("../PHP/GeoTIFF_Save_v1.4.php",
                {
                        <data>
                })
                .done(function(data, status){
                        <do stuff here>
                })
                .fail(function(xhr){
                        <handle error here>
                })
        }
});
```