GSKY (OWS) server

- Setup of server, database and datasets -

By

Arapaut V. Sivaprasad

30 Nov – 9 Dec 2018

FOREWORD

This document describes the code base of the GSKY server and the different GSKY services for developers. A User Guide is available for the end users. Animated slideshow at GSKY Developer Guide.ppsx will help to understand the flow of process before reading the details in this document. TerriaMap is a product of CSIRO Data61 and is a client that can display data from various servers including GSKY. A basic understanding of TerriaMap will help to understand how GSKY works and, hence, this document will explain how the modules in both work together. It is not in the scope of this document to detail everything about TerriaMap, but about GSKY it is.

DISCLAIMER

This is an evolving document and may go out of sync with the programs in time. Those who maintain the programs are requested to update this document as well. While every effort has been made to make it as comprehensive as possible, not every line in the code is explained here. The self-explanatory lines and, except when essential, the code from packages are not described.

HOW TO USE THIS DOCUMENT

- Read TL;DR if you just want to know the basics.
- Watch the slideshow, GSKY_User_Guide.ppsx, to use the system.
- Watch GSKY_Developer_Guide.ppsx to know how it works.
- Read the OVERVIEW and SYNOPSIS to get a bit more.
- Read DETAILS of GSKY code and TERRIAMAP-GSKY conversation to modify the code.

TL;DR

- The GSKY server provides three services, *viz.* WMS, WPS and WCS.
  - The WMS service is described in this document.
- Five functions in the main package (ows.go) are involved.
  - Several functions in other packages are called by these three.
- **func init()** sets up the GSKY server and listen on port 80 or 8080
  - Only port 80 is working on the server I tried.
  - Hence, must start the GSKY server as '*/local/gsky/share/gsky/gsky -p 80&*'
- **func main()**
  - Init and Main are only involved in setting up the server.
- **func owsHandler()**
  - Called when a HTTP request is received.
- **func generalHandler ()** handles all requests from the web.
  - e.g. http://130.56.242.15/ows/geoglam?...&request=GetMap&layers=global...cover&bbox=16...327&...
  - There is support for both GET and POST, but only the GET method is described in this document.
- **func serveWMS()** serves the data to the web request.
  - **case "GetMap"** retrieves and serves the data.
  - The data comes from the MAS server (specified in config.json) and worker nodes.
- The GSKY configuration file(s) reside in /usr/local/etc and must be named as 'config.json'
  - Sub-directories are allowed. e.g. /usr/local/etc/geoglam
  - These form separate namespaces.
- TerriaMap, a derivative of TerriaJS, is the client that uses GSKY server to display the data.
  - TerriaMap makes five separate 'requests' to the GSKY server for service=WMS.
  - *GetCapabilities*, *DescribeLayer*, *GetLegendGraphic*, *GetMap* and *GetFeatureInfo*

INTRODUCTION

The GSKY server is what allows a web-based service like 'Terria JS' to access the data and display tiles on the map. The script, 'build_all.sh', will setup the GSKY server, and its documentation, 'build_all.pptx', describes the process as animated PowerPoint presentation.

This document is a detailed description of the code base that creates and runs the GSKY server. The PowerPoint presentation, 'build_all.ppsx', gives animated description of the GSKY server setup process.

To understand the process, the fundamental interactions at code level between TerriaMap and GSKY are explained, but it is by no way a comprehensive description of how TerriaMap works.

HIGH LEVEL PROCESS FLOW

- Animated slide show: GSKY_Developer_Guide.ppsx and GSKY_Developer_Guide.pptx

OVERVIEW

Components

GSKY has three separate services, *viz.*, Web Mapping Service (WMS), Web Coverage Service (WCS) and Web Processing Service (WPS). Of these, only the WMS and WPS have been implemented in GSKY so far. This document will describe the WMS.

There are several "requests" for each service. Given below is a list of services and requests described in this document.

Services

- WMS – Web Mapping Service

Requests

- WMS:GetCapabilities
- WMS:DescribeLayer
- WMS:GetLegendGraphic
- WMS:GetMAP
- WMS:GetFeatureInfo

TerriaMap components

TerriaJS is an innovative open source solution enabling publishers to efficiently get their spatial data on the web, including 3D and 4D data[Ref]. TerriaMap is a complete website starting point, using TerriaJS. Detailed explanation and source code for TerriaJS is available at this Github repository.

Most of the functions relevant to GSKY can be found in the following JS file. Code snippets from it are reproduced in this document and in GSKY_Developer_Guide.pptx to explain the GSKY-to-TerriaMap interactions.

```
/wwwroot/build/TerriaJS/build/TerriaJS-specs.js
```

SYNOPSIS

When rendering the map data through a client like TerriaJS, a service is called, as e.g. service=WMS, and a related request as 'request=GetMAP'. Several different types of requests are made to the GSKY server to get the appropriate information. The 'request=GetMap' is called the most number of times, and each call returns one piece of the map data. The layer and bounding box (bbox) specified in the request determines which area of the map is to be returned.

The different requests are listed below. They are roughly in the order in which they are called, but zooming, changing layer, etc. will call them in different orders. Full descriptions about these at the code level are given later in the document. A Power Point presentation animates the process for a quick overview.

Service: WMS

GetCapabilities

This is the first call before any data is displayed. The details it sends to the web, in XML format, contains all the layers, times, bounding boxes, etc. so that the display can list them to choose from.

http://130.56.242.15/ows/geoglam?service=WMS&request=GetCapabilities&version=1.3.0&tiled=true

DescribeLayer

When a displayed layer is clicked on, it sends a request for 'DescribeLayer' as below. It sends back a textual description of the selected layer.

GetLegendGraphic

When a layer is added to the map, by clicking 'add to map', it calls this request to display an image showing the colors representing various bands such as photosynthetic and non-photosynthetic vegetation and bare soil.

http://130.56.242.15/ows/geoglam?service=WMS&version=1.1.0&request=GetLegendGraphic&format=image%2Fpng&transparent=True&layer=global%3Ac6%3Amonthly_anom_frac_cover&srs=EPSG%3A3857

GetMap

This is the call that sends the actual map data. There will be many such calls to display a picture. At low zoom level, when a large part of the globe is shown, there will be more such calls than when the zoom level is higher. This call is very time and CPU-intensive, but it is displayed in a very short time, thanks to the concurrency in execution of the code. It is the most brilliant aspect of the GSKY service, which has speeds comparable to a Google search.

http://130.56.242.15/ows/geoglam?time=2...&service=WMS&request=GetMap&layers=global...&request=GetMap&bbox=15...7,-26...,15...,-26..&

GetFeatureInfo

When an are on the map is clicked, it calls this request to display details such as x,y co-ordinates, time and the band name.

http://130.56.242.15/ows/geoglam?time=&service=WMS&request=GetFeatureInfo&layers=&query_layers=&...

TerriaMap to GSKY conversation

Add data->My Data->Add web data

This is the point at which any server, including GSKY, will be selected by Terria. Upon giving the URL of the GSKY server (e.g. http://130.56.242.15/ows/geoglam) the first thing Terria does is to get the capabilities of the server. The GSKY request=GetCapabilities is called as …

*http://130.56.242.15/ows/geoglam?service=WMS&request=**GetCapabilities**&version=1.3.0&tiled=true*

GSKY sends back, in response to the above call, the full details of all datasets and their layer names. Terria "gulps" the details and the layer names are listed as further navigation links.

DescribeLayer

Clicking a layer name sends the following request to the GSKY server.

*http://130.56.242.15/ows/geoglam?service=WMS&version=1.1.1&sld_version=1.1.0&request=**DescribeLayer**&layers=global%3Ac6%3Amonthly_anom_frac_cover*

GSKY sends back the details of the layer such as abstract, title, etc. Terria displays the layer title and description. Also shown on the page are the info received through the 'request=GetCapabilities' and a map overview with an 'Add to the map' link.

Add to the map

Clicking this link displays the map with the layers overlaid on it. Terria sends two requests, *viz.*, **GetLegendGraphic** and **GetMap** as below. Of these, the GetMap is requested multiple times to get all tiles that are to be displayed based on the zoom level of the displayed map.

```
http://130.56.242.15/ows/geoglam?service=WMS&version=1.1.0&request=GetLegendGraphic&format=image%2Fpng&transparent=True&layer=global%3Ac6%3Amonthly_anom_frac_cover&srs=EPSG%3A3857
```

```
http://130.56.242.15/ows/geoglam?time=2018...00Z&srs=EPSG:3857&...&service=WMS&request=GetMap&layers=global...&bbox=15...&width=256&height=256
```

Terria displays a legend graphic image to show the intensity of colors for the various bands, and the bands themselves are layered over the map.

GetFeatureInfo

Clicking any point on the map sends a request= GetFeatureInfo as below. GSKY sends back the details such as X,Y co-ordinates, time and band name.

```
http://130.56.242.15/ows/geoglam?time=2018...&srs=EPSG:3857&...&service=WMS&...&request=GetFeatureInfo&layers=global...&bbox=15...&...&query_layers=global...
```

Terria displays an info box over the map to show the feature details in text form.

DETAILS of GSKY code

Functions

There are three main functions that call several sub-functions to process and deliver the data. The main functions, in 'ows.go', are described below.

func init()

This is called only once when a GSKY server is started by the following command:

```
/local/gsky/share/gsky/gsky -p 80&
```

The server so started will remain as a background process and waits for connection on port 80 like any webserver.

The '**func init()**' does the following:

- Define error and info output methods. These are used to show messages on the console. They are text messages displayed on the interactive shell session from where the server was started. If the shell window has been closed, then these messages will not be saved anywhere.

```
Error = log.New(os.Stderr, "OWS: ", log.Ldate|log.Ltime|log.Lshortfile)
Info = log.New(os.Stdout, "OWS: ", log.Ldate|log.Ltime|log.Lshortfile)
```

- Define other required variables and do sanity checks on them. Exit if any error in any one of the components.

- DataDir
- EtcDir
- filePaths

```
utils.EtcDir = *serverConfigDir
utils.DataDir = *serverDataDir
```

```
var EtcDir = /usr/local/etc/
var DataDir = /usr/local/share/gsky@
```

```
- confMap
- reWMSMap
- reWCSMap
- reWPSMap
```

- Define filePaths to find the required files.

```
filePaths := []string{
        utils.DataDir + "/static/index.html",
        utils.DataDir + "/templates/WMS_GetCapabilities.tpl",
        utils.DataDir + "/templates/WMS_DescribeLayer.tpl",
        utils.DataDir + "/templates/WMS_ServiceException.tpl",
        utils.DataDir + "/templates/WPS_DescribeProcess.tpl",
        utils.DataDir + "/templates/WPS_Execute.tpl",
        utils.DataDir + "/templates/WPS_GetCapabilities.tpl",
        utils.DataDir + "/templates/WCS_GetCapabilities.tpl",
        utils.DataDir + "/templates/WCS_DescribeCoverage.tpl"}
```

   - Check that the above files are present. Error exit if any is not found.

```
        for _, filePath := range filePaths {
                if _, err := os.Stat(filePath); os.IsNotExist(err) {
                        panic(err)
                }
        }
```

   - Define **confMap**

```
        confMap, err := utils.LoadAllConfigFiles(utils.EtcDir, *verbose)
```

This reads the 'config.json' to create a "GO map type" which is like a hash in Perl or Dictionary in Python. In addition to the keys specified in the config.json, the resulting map 'type Config' contains several new keys generated by the program (see below; marked in red are the keys from config.json). The server exits if the config.json does not exist or is unreadable.

configMap:

```
{
  ".": {
    "service_config": {
      "ows_hostname": "130.56.242.15",
      "NameSpace": "",
      "mas_address": "10.0.1.210:8888",
      "worker_nodes": [
        "10.0.1.190:6000",
        "10.0.1.192:6000"
      ],
      "ows_cluster_nodes": null,
      "temp_dir": "",
      "max_grpc_buffer_size": 0
    },
    "layers": [
      {
        "ows_hostname": "130.56.242.15",
        "NameSpace": "",
        "name": "LS8:NBAR:TRUE",
        "title": "DEA Landsat 8 surface reflectance true colour",
        "abstract": "This product has been corrected … ",
        "metadata_url": "",
        "data_url": "",
        "data_source": "/g/data2/rs0/datacube/002/LS8_OLI_NBAR",
        "start_isodate": "2013-03-01T00:00:00.000Z",
        "end_isodate": "mas",
        "EffectiveStartDate": "2013-03-01T00:00:00.000Z",
        "EffectiveEndDate": "2018-09-07T00:00:00.000Z",
        "TimestampToken": "1540944639.747149",
        "step_days": 16,
        "step_hours": 0,
        "step_minutes": 0,
        "accum": true,
        "time_generator": "regular",
        "resolution_filter": null,
        "dates": [
          "2013-03-01T00:00:00.000Z",
          "2013-03-17T00:00:00.000Z",
          "...",
          "2018-08-22T00:00:00.000Z",
          "2018-09-07T00:00:00.000Z"
        ],
        "rgb_products": [
          "red",
          "green",
          "blue"
        ],
        "RGBExpressions": {
          "ExprText": [
            "red",
            "green",
            "blue"
          ],
          "Expressions": null,
          "VarList": [
            "red",
            "green",
            "blue"
          ],
          "ExprNames": [
            "red",
            "green",
            "blue"
          ],
          "ExprVarRef": [
            [
              "red"
            ],
            [
              "green"
            ],
                  [
                     "blue"
                  ]
               ]
            },
            "mask": null,
            "offset_value": 0,
            "clip_value": 2500,
            "scale_value": 0.1016,
            "palette": null,
            "legend_path": "",
            "legend_height": 0,
            "legend_width": 0,
            "styles": null,
            "zoom_limit": 500,
            "max_grpc_recv_msg_size": 10485760,
            "wms_polygon_segments": 2,
            "wcs_polygon_segments": 10,
            "wms_timeout": 20,
            "wcs_timeout": 30,
            "grpc_wms_conc_per_node": 16,
            "grpc_wcs_conc_per_node": 16,
            "wms_polygon_shard_conc_limit": 2,
            "wcs_polygon_shard_conc_limit": 2,
            "band_strides": 0,
            "wms_max_width": 512,
            "wms_max_height": 512,
            "wcs_max_width": 50000,
            "wcs_max_height": 30000,
            "wcs_max_tile_width": 1024,
            "wcs_max_tile_height": 1024,
            "feature_info_max_data_links": 0,
            "feature_info_data_link_url": "",
            "feature_info_bands": null,
            "FeatureInfoExpressions": {
               "ExprText": null,
               "Expressions": null,
               "VarList": null,
               "ExprNames": null,
               "ExprVarRef": null
            },
            "nodata_legend_path": ""
         }
      ],
      "processes": null
   }
}
```

config.json:

```
{
 "service_config": {
  "mas_address": "10.0.1.210:8888",
  "worker_nodes": [
    "10.0.1.190:6000",
    "10.0.1.192:6000"
  ],
  "ows_hostname": "130.56.242.15"
 },
 "layers": [
  {
   "step_days": 16,
   "abstract":"This product has been corrected...and may include clouds.  ",
   "start_isodate": "2013-03-01T00:00:00.000Z",
   "clip_value": 2500,
   "data_source": "/g/data2/rs0/datacube/002/LS8_OLI_NBAR",
   "offset_value": 0,
   "rgb_products": [
    "red",
    "green",
    "blue"
   ],
   "name": "LS8:NBAR:TRUE",
   "title": "DEA Landsat 8 surface reflectance true colour",
   "scale_value": 0.1016,
   "time_generator": "regular",
   "accum": true,
   "end_isodate": "mas",
   "zoom_limit": 500
  }
 ]
}
```

The 'func init()' does a check for 'validateConfig' and exits if true. Its value is hard coded in the program (see below) and is unclear why it is needed.

```
if *validateConfig {
		os.Exit(0)
}

var (
		validateConfig  = flag.Bool("check_conf", false, "Validate server config files.")
)

dumpConfig      = flag.Bool("dump_conf", true, "Dump server config files.")
```

By setting the above value to true, it will dump the 'confMap' constructed above and exits. This function can be used for debugging purposes.

In addition to the above variable, dumpConfig, the following is a full list. Their functions are self-explanatory. In particular, setting verbose=true will display info such as the URL params, ConfigMAP, etc.

```
var (
		port           = flag.Int("p", 8080, "Server listening port.")
```

```
        serverDataDir   = flag.String("data_dir", utils.DataDir, "Server data directory.")
        serverConfigDir = flag.String("conf_dir", utils.EtcDir, "Server config directory.")
        validateConfig  = flag.Bool("check_conf", false, "Validate server config files.")
        dumpConfig      = flag.Bool("dump_conf", false, "Dump server config files.")
        verbose         = flag.Bool("v", false, "Verbose mode for more server outputs.")
)


utils.WatchConfig(Info, Error, &configMap, *verbose)
```

Upon receiving a SIGHUP, for example sending the process to the background, the above unction reloads the config.json(s).

```
        case <-sighup:
                infoLog.Println("Caught SIGHUP, reloading config...")
                confMap, err := LoadAllConfigFiles(EtcDir, verbose)
```

```
○               reWMSMap = utils.CompileWMSRegexMap()
○               reWCSMap = utils.CompileWCSRegexMap()
○               reWPSMap = utils.CompileWPSRegexMap()
```

```
reWMSMap = {"bbox":{},"crs":{},"height":{},"request":{},"service":{},"time":{},"width":{},"x":{},"y":{}}
```

The above functions blank out certain keys. Unsure as yet its purpose.

func owsHandler()

This function is called when any HTTP request arrives from TerriaMap. The r.URL.Path received from the call will be like '/ows' or '/ows/namespace'. Based on it the variable, 'namespace', is assigned value of '.' or 'namespace' (e.g. geoglam). Return with an error if the namespace is not in 'configMap[namespace]'. This can happen if the GSKY URL is typed in wrongly.

```
        Invalid dataset namespace: geoglam1 for url: /ows/noname
```

If namespace is OK, then call the 'func generalHandler()'

```
        config.ServiceConfig.NameSpace = namespace
        generalHandler(config, w, r)
```

func generalHandler ()

This function handles all requests coming from the web as GET params.

The URL string typically looks like the following. The flow control options are marked in red.

```
/ows/geoglam?time=2018-10-
01T00%3A00%3A00.000Z&srs=EPSG%3A3857&transparent=true&format=image%2Fpng&exceptions=application%2Fvnd.o
gc.se_xml&styles=&tiled=true&feature_count=101&service=WMS&version=1.1.1&request=GetMap&layers=global%3Ac6
%3Amonthly_anom_frac_cover&bbox=16280475.528516259%2C-2504688.542848654%2C17532819.79994059%2C-
1252344.271424327&width=256&height=256
```

Though both GET and POST methods are supported, the above URL uses the GET method and it is the one described here.

```
case "GET":
        query = utils.NormaliseKeys(r.URL.Query())
}
```

The query string is put into a map object (hash) as below.

```
version: [1.1.1]
transparent: [true]
tiled: [true]
service: [WMS]
height: [256]
srs: [EPSG:3857]
feature_count: [101]
styles: []
bbox: [16280475.528516259,-2504688.542848654,17532819.79994059,-1252344.271424327]
width: [256]
format: [image/png]
exceptions: [application/vnd.ogc.se_xml]
layers: [global:c6:monthly_anom_frac_cover]
time: [2018-10-01T00:00:00.000Z]
request: [GetMap]
```

- serveWMS(ctx, params, conf, r.URL.String(), w)

ctx: This is returned from 'http.Request' in the package "net/http"

```
r *http.Request
ctx := r.Context()
```

params: Retrieved from the query string shown above. Only just the params pertaining to the WMS service are taken.

```
{"service":"WMS","request":"GetMap","crs":"EPSG:3857","bbox":[16280475.528516259,-2504688.542848654,17532819.79994059,-1252344.271424327],"height":256,"width":256,"time":"2018-10-01T00:00:00Z","layers":["global:c6:monthly_anom_frac_cover"],"styles":[""],"version":"1.1.1"}
```

conf: This is a struct object derived from the config.json. There may be several layers, but only one 'service_config' and 'processes' in the config.json

```
type Config struct {
		ServiceConfig ServiceConfig `json:"service_config"`
		Layers       []Layer      `json:"layers"`
		Processes     []Process    `json:"processes"`
}
```

ServiceConfig: Partially comes from config.json. Those that come from config.json are marked in green. Others come from elsewhere (must find out). There will only be 1 ServiceConfig.

```
"service_config":
{
		"ows_hostname":"130.56.242.15",
		"NameSpace":"geoglam",
		"mas_address":"10.0.1.210:8888",
		"worker_nodes":["10.0.1.190:6000","10.0.1.192:6000"],
		"ows_cluster_nodes":null,
		"temp_dir":"",
		"max_grpc_buffer_size":0
}
```

Layers: Partially comes from config.json. There could be several layers within this. Those that come from config.json are marked in green. Others come from elsewhere (must find out).

```
"layers":[
{
		"ows_hostname":"130.56.242.15",
"NameSpace":"geoglam",
"name":"global:c5:frac_cover",
"title":"GEOGLAM Fractional Cover C5",
"abstract":"Fractional Cover - MODIS,...",
"metadata_url":"",
"data_url":"",
"data_source":"/g/data2/u39/public/prep/modis-fc/FC.v302.MCD43A4",
"start_isodate":"2000-02-18T00:00:00.000Z",
"end_isodate":"2017-03-14T00:00:00.000Z",
"EffectiveStartDate":"2000-02-18T00:00:00.000Z",
"EffectiveEndDate":"2017-03-14T00:00:00.000Z",
"TimestampToken":"",
"step_days":8,
"step_hours":0,
"step_minutes":0,
"accum":false,
"time_generator":"mcd43",
"resolution_filter":null,
"dates":["2000-02-18T00:00:00.000Z","2000-02-26T00:00:00.000Z","...","2017-03-14T00:00:00.000Z"],
"rgb_products":["bare_soil","phot_veg","nphot_veg"],
"RGBExpressions":
{
            "ExprText":["bare_soil","phot_veg","nphot_veg"],
            "Expressions":null,
            "VarList":["bare_soil","phot_veg","nphot_veg"],
            "ExprNames":["bare_soil","phot_veg","nphot_veg"],
            "ExprVarRef":[["bare_soil"],["phot_veg"],["nphot_veg"]]
}
"mask":null,
"offset_value":0,
"clip_value":100,
"scale_value":2.54,
"palette":null,
"legend_path":"/local/gsky/share/gsky/static/legend/MODIS_FC_Triangle.png",
"legend_height":0,
"legend_width":0,
"styles":null,
"zoom_limit":10000,
"max_grpc_recv_msg_size":10485760,
"wms_polygon_segments":2,
"wcs_polygon_segments":10,
"wms_timeout":20,
"wcs_timeout":30,
"grpc_wms_conc_per_node":16,
"grpc_wcs_conc_per_node":16,
"wms_polygon_shard_conc_limit":2,
"wcs_polygon_shard_conc_limit":2,
"band_strides":0,
"wms_max_width":512,
"wms_max_height":512,
"wcs_max_width":50000,
"wcs_max_height":30000,
"wcs_max_tile_width":1024,
"wcs_max_tile_height":1024,
"feature_info_max_data_links":0,
"feature_info_data_link_url":"",
"feature_info_bands":null,
"FeatureInfoExpressions":
{
            "ExprText":null,
            "Expressions":null,
            "VarList":null,
            "ExprNames":null,
            "ExprVarRef":null
}
```

```
		"nodata_legend_path":""
}
```

Processes: Partially **c**omes from config.json. There will be only one set.

```
"processes":
[
		{
		"data_sources":null,
		"identifier":"geometryDrill",
		"title":"Geometry Drill",
		"abstract":"",
		"max_area":400,
		"literal_data":null,
		"complex_data":
		[
				{
				"identifier":"geometry",
				"title":"Geometry",
				"abstract":"",
				"mime_type":"application/vnd.geo+json",
				"encoding":"",
				"schema":"http://geojson.org/geojson-spec.html",
				"min_occurs":0
				}
		],
		"identity_tol":-1,
		"dp_tol":-1,
		"approx":true
		}
]
```

r.URL.String(): The http URL to GET the tile info.

```
/ows/geoglam?time=2018-10-01T00%3A00%3A00.000Z&srs=EPSG%3A3857&transparent=true&format=image%2Fpng&exceptions=application%2Fvnd.ogc.se_xml&styles=&tiled=true&feature_count=101&service=WMS&version=1.1.1&request=GetMap&layers=global%3Ac6%3Amonthly_anom_frac_cover&bbox=16280475.528516259%2C-2504688.542848654%2C17532819.79994059%2C-1252344.271424327&width=256&height=256
```

w: This is returned from 'http.ResponseWriter' in the package "`net/http`"

```
&{0xc42075ee60 0xc420756700 {} 0x4d1610 false false false false 0xc420074100 {0xc42077e000 map[] false false} map[Access-Control-Allow-Origin:[*]] true 0 -1 0 false false [] 0 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0 0] [0 0 0] 0xc420076000 0}
```

func serveWMS()

This is the function that delivers the data to the web request. There are several 'request=XXX' types, but in this document only the 'GetMAP' is described.

```
switch *params.Request {
		case "GetCapabilities":
		case "GetFeatureInfo":
		case "DescribeLayer":
		case "GetLegendGraphic":
		case "GetMap":
```

case "GetMap":

This goes through a series of actions as listed below.

- Sanity checks:
  - Exit if the software version is not specified in the URL as e.g. *version=1.1.1*
  - If the 'time=' is not in the URL, e.g *time=2018-10-01T00%3A00%3A00.000Z*, use the current time.

    - Exit if the CRS is not specified as e.g. *srs=EPSG%3A3857*
    - Exit if bounding box is not specified as e.g. *bbox=16280475.528516259%2C-2504688.542848654%2C17532819.79994059%2C-1252344.271424327*
    - Exit if Height and Width are not specified as e.g. *width=256&height=256*
    - Exit if Height and Width are too large.
    - Change BBox to float64 if CRS=EPSG:4326 and GSKY version is 1.3.0 (unsure why)
    - Change CRS to EPSG:4326 if it is defined as CRS:84 and GSKY version is 1.3.0 (unsure why)

- Get the index of the requested layer: e.g. *layers=global%3Ac6%3Amonthly_anom_frac_cover*

```
idx, err := utils.GetLayerIndex(params, conf)
```

utils/wms.go:GetLayerIndex()

```
product := params.Layers[0]
for i := range config.Layers {
        if config.Layers[i].Name == product {
                return i, nil
        }
}
```

The 'params' hold all layers specified in config.json. If the requested 'product' matches one of them, then send its index. Otherwise, send an error. The product name is case-sensitive.

```
currentTime, err := utils.GetCurrentTimeStamp(conf.Layers[idx].Dates)
```

if the 'time=' is not in the URL, then use the current time. The implication is that users will get a blank image in the HTTP response instead of the 500 internal server error.

utils/wms.go: GetCurrentTimeStamp()

```
if len(timestamps) == 0 { currentTime = time.Now().UTC() }
```

- Construct the "endTime"

  var endTime *time.Time

```
        if conf.Layers[idx].Accum == true {
                step := time.Minute *
time.Duration(60*24*conf.Layers[idx].StepDays+60*conf.Layers[idx].StepHours+conf.Layers[idx].StepMinutes)
                eT := params.Time.Add(step)
                endTime = &eT
        }
```

- styleIdx, err := utils.GetLayerStyleIndex(params, conf, idx)

Take the styles index from the URL (styles=&). If it is empty, the index is taken from config.json as the layer which is specified as layers=global%3Ac6%3Amonthly_anom_frac_cover. In this case, idx=7.

- geoReq := &proc.GeoTileRequest{ConfigPayLoad: proc.ConfigPayLoad{NameSpaces: styleLayer.RGBExpressions.VarList,…

This returns the params required to request the tile info from the MAS database.

geoReq:

```
&{{[bare_soil] 0xc42015ac80 {25 5 50} 0xc4201983e0 <nil> 10000 2 16 0 -1} /g/data2/tc43/modis-
fc/v310/tiles/monthly/anomalies EPSG:3857 [1.6280475528516259e+07 -2.504688542848654e+06
1.753281979994059e+07 -1.252344271424327e+06] 256 256 0 0 2018-10-01 00:00:00 +0000 UTC 2018-10-09
00:00:00 +0000 UTC}
```

Determine the resolution or 'zoom level'. Larger between X and Y resolution is taken as the zoom level.

```
xRes := (params.BBox[2] - params.BBox[0]) / float64(*params.Width)
yRes := (params.BBox[3] - params.BBox[1]) / float64(*params.Height)
reqRes := xRes
if yRes > reqRes {
   reqRes = yRes
}
```

- if conf.Layers[idx].ZoomLimit != 0.0 && reqRes > conf.Layers[idx].ZoomLimit

The above says that if the zoom level is low a tile is not shown, but instead an image that says "zoom in to view this layer" is displayed. The image data is retrieved by the following code.

  - Create an object, 'indexer', of type '*processor.TileIndexer'. This holds the HTTP request params to be sent to the MAS server.

indexer:

```
&{context.Background.WithValue(&http.contextKey{name:"http-server"}, &http.Server{Addr:"0.0.0.0:80",
Handler:http.Handler(nil), TLSConfig:(*tls.Config)(0xc4201e0f00), ReadTimeout:0, ReadHeaderTimeout:0, WriteTimeout:0,
IdleTimeout:0, MaxHeaderBytes:0, TLSNextProto:map[string]func(*http.Server, *tls.Conn,
http.Handler){"h2":(func(*http.Server, *tls.Conn, http.Handler))(0x6b2e40)}, ConnState:(func(net.Conn, http.ConnState))(nil),
ErrorLog:(*log.Logger)(nil), disableKeepAlives:0, inShutdown:0, nextProtoOnce:sync.Once{m:sync.Mutex{state:0, sema:0x0},
done:0x1}, nextProtoErr:error(nil), mu:sync.Mutex{state:0, sema:0x0}, listeners:map[net.Listener]struct
{}{http.tcpKeepAliveListener{TCPListener:(*net.TCPListener)(0xc4201a60d8)}:struct {}{}}, activeConn:map[*http.conn]struct
{}{(*http.conn)(0xc42077edc0):struct {}{}, (*http.conn)(0xc42077ee60):struct {}{}}, doneChan:(chan struct {})(nil),
onShutdown:[]func(){(func())(0x6bb440)}}).WithValue(&http.contextKey{name:"local-addr"}, &net.TCPAddr{IP:net.IP{0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0xff, 0xff, 0xa, 0x0, 0x1, 0xef}, Port:80, Zone:""}).WithCancel.WithCancel.WithCancel
0xc4201fede0 0xc4201fee40 0xc4201fed80 10.0.1.210:8888 0}
```

  - Send the value of 'geoReq' (see above) into the channel, 'indexer.In'.
  - Run the 'indexer' as: *go indexer.Run(*verbose)*. It returns the data as shown below for each Namespace (e.g. "bare_soil").

```
&{{[bare_soil] <nil> {25 5 50} 0xc420110920 <nil> 0 0 16 0 0} NETCDF:"/g/data2/tc43/modis-
fc/v310/tiles/monthly/anomalies/FC_Mean_Diff.v310.MCD43A4.h31v11.2018.006.nc":bare_soil EPSG:3857
[1.6280475528516259e+07 -2.504688542848654e+06 1.753281979994059e+07 -1.252344271424327e+06] 256 256 0 0
bare_soil [2018-01-01 00:00:00 +0000 UTC 2018-02-02 00:00:00 +0000 UTC 2018-03-02 00:00:00 +0000 UTC 2018-04-03
00:00:00 +0000 UTC 2018-05-01 00:00:00 +0000 UTC 2018-06-02 00:00:00 +0000 UTC 2018-07-04 00:00:00 +0000 UTC 2018-
08-01 00:00:00 +0000 UTC 2018-09-01 00:00:00 +0000 UTC 2018-10-01 00:00:00 +0000 UTC 2018-11-01 00:00:00 +0000 UTC
2018-12-01 00:00:00 +0000 UTC] 2018-10-01 00:00:00 +0000 UTC POLYGON ((14454893.153252 -
2223437.436882,14454893.153252 -3336315.161445,15567770.877815 -3336315.161445,15567770.877815 -
2223437.436882,14454893.153252 -2223437.436882)) Float32}
```

  - Unless the Namespace is not "EmptyTile", it means there is data.

```
if hasData {out, err := utils.GetEmptyTile(utils.DataDir+"/zoom.png", *params.Height, *params.Width)}
```

The 'out' from the above call has the content of 'zoom.png' which is overlaid on the map.

```
/usr/local/share/gsky/zoom.png:
[137 80 78 71 13 10 26 10 0 0 0 … 66 96 130]
```

If the zoom level is at or higher than the required level, retrieve the tile and display it as per the code below. It will finish the output from 'func serveWMS'

- Create an object, ‘tp’, of type, ‘*processor.TilePipeline’, which is similar to the ‘indexer’ object (see above) created for displaying the zoom.png.
    - Send the value of ‘geoReq’ to ‘func Process’ (in tile_pipieline.go) to get the data as a raster image (type []utils.Raster)
    - Send the query to the MAS server and get the data. (this section needs more investigation)

```
http://10.0.1.210:8888/g/data2/tc43/modis-fc/v310/tiles/monthly/anomalies?intersects&metadata=gdal&time=2018-10-01T00:00:00.000Z&until=2018-10-09T00:00:00.000Z&srs=EPSG:3857&wkt=POLYGON%20((16280475.528516%20-2504688.542849,%2017532819.799941%20-2504688.542849,%2017532819.799941%20-1252344.271424,%2016280475.528516%20-1252344.271424,%2016280475.528516%20-2504688.542849))&namespace=bare_soil&nseg=2&limit=-1
```

```
		case res := <-tp.Process(geoReq, *verbose):
```

res:

```
&{bare_soil [255 255 255 255 … -11.833333 255 255 … 256 255}
```

Scale the output as per the zoom factor.

```
norm, err := utils.Scale(res, scaleParams)
```

Encode the data into a PNG image.

```
out, err := utils.EncodePNG(norm, styleLayer.Palette)
```

Send the PNG image to the web.

```
w.Write(out)
```

TERRIAMAP-GSKY conversation

Displaying the data as layers over the global map (provided by Microsoft and Bing) requires the real-time communication between the two programs. In this case, TerriaMap, the client, makes a series of requests to the GSKY server. There are five separate ‘requests’ under the ‘service=WMS’. They are described below with code snippets and example outputs.

1. GetCapabilities

The first step is to get the capabilities of the GSKY server. Terria calls the server through the function below. Line numbers in the code (TerriaJS-specs.js) are given for reference. Only the relevant steps are shown. There are many steps in between.

In TerriaJS-specs.js:

```
45158: function loadFromCapabilities(wmsItem) {

45212:    updateInfoSection(wmsItem, overwrite, 'GetCapabilities URL', wmsItem.getCapabilitiesUrl);

44663:    overrideProperty(this, 'getCapabilitiesUrl', {

44677:    return cleanUrl(this.url) + '?service=WMS&version=1.3.0&request=GetCapabilities';
```

To GSKY: *http://130.56.242.15/ows/geoglam?service=WMS&request=**GetCapabilities**&version=1.3.0&tiled=true*

In ows.go:

```
func serveWMS(ctx context.Context, params utils.WMSParams, conf *utils.Config, reqURL string, w http.ResponseWriter) {
		case "GetCapabilities":
```

```
                    utils.ExecuteWriteTemplateFile(w, conf,
utils.DataDir+"/templates/WMS_GetCapabilities.tpl")
```

The above code returns an XML created from the template, */local/gsky/share/gsky/templates/WMS_GetCapabilities.tpl*, by replacing several tags with their values (e.g. .Name, .Title, .Abstract, etc.).

In TerriaJS-specs.js:

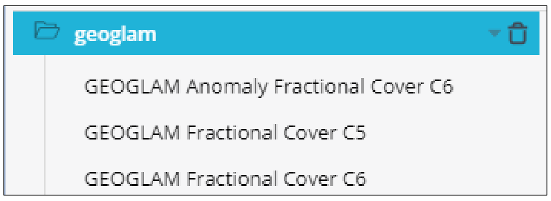
```
45807: function updateInfoSection(item, overwrite, sectionName, sectionValue) {
119431: function updateInfoSection(item, overwrite, sectionName, sectionValue) {
```

(This function is apparently duplicated. See (2) under 'OBSERVATIONS')

```
45252:    updateValue(wmsItem, overwrite, 'tilingScheme', tilingScheme);

45257:    updateValue(wmsItem.parameters, overwrite, 'srs', srs);

45260:       updateValue(wmsItem, overwrite, 'colorScaleMinimum', -50);

45261:       updateValue(wmsItem, overwrite, 'colorScaleMaximum', 50);

45823:   function updateValue(item, overwrite, propertyName, propertyValue) {
119447: function updateValue(item, overwrite, propertyName, propertyValue) {
```

(This function is apparently duplicated. See (2) under 'OBSERVATIONS')

From the XML returned from GSKY, the layer names alone are displayed as below. The rest of the data is kept in memory for later use.



GEOGLAM Anomaly Fractional Cover C6

GEOGLAM Fractional Cover C5

GEOGLAM Fractional Cover C6

2. DescribeLayer

This request gets certain values for the selected layers and display them.

In TerriaJS-specs.js:

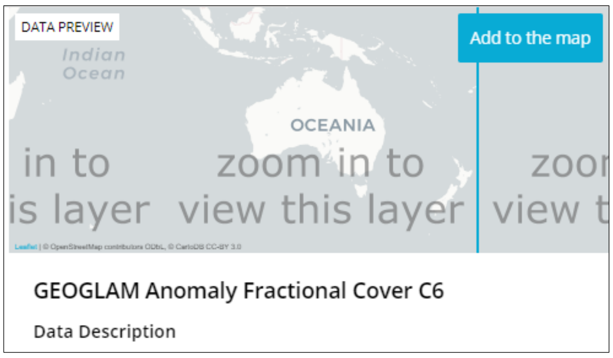
```
45273: WebMapServiceCatalogItem.prototype._load = function () {

45298:  var describeLayersURL = cleanUrl(that.url) +
'?service=WMS&version=1.1.1&sld_version=1.1.0&request=DescribeLayer&layers=' +
encodeURIComponent(that.layers);
```

In ows.go:

```
func serveWMS(ctx context.Context, params utils.WMSParams, conf *utils.Config, reqURL string, w
http.ResponseWriter) {
case "DescribeLayer":

utils.ExecuteWriteTemplateFile(w, conf.Layers[idx], utils.DataDir+"/templates/WMS_DescribeLayer.tpl")
```

The above code returns an XML created from the template, */local/gsky/share/gsky/templates/WMS_DescribeLayer.tpl*, by replacing three tags with their values (e.g. .Name, .OWSHostname and .NameSpace).

The layer name and abstract from the XML returned by the above call is displayed, alongwith the previous data received from the 'GetCapabilities', for further action.



GEOGLAM Anomaly Fractional Cover C6

Data Description

3. GetLegendGraphic

Clicking the ‘Add to the map’ above will first call this request to GSKY.

In TerriaJS-specs.js:

45945: function computeLegendForLayer(catalogItem, thisLayer, styleName) {

45977: legendUri = new URI(cleanUrl(catalogItem.url) + '?service=WMS&version=1.1.0&request=GetLegendGraphic&format=image/png&transparent=True&layer=' + encodeURIComponent(thisLayer.Name));

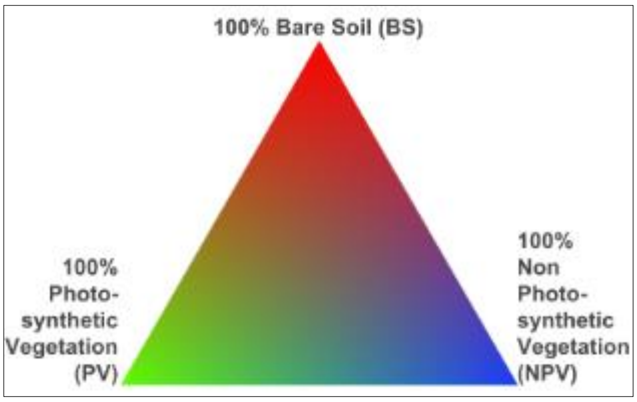
46010: legendUri.setQuery('colorscalerange', [catalogItem.colorScaleMinimum, catalogItem.colorScaleMaximum].join(','));

In ows.go:

func serveWMS(ctx context.Context, params utils.WMSParams, conf *utils.Config, reqURL string, w http.ResponseWriter) {
utils.ExecuteWriteTemplateFile(w, params.Layers[0], utils.DataDir+"/templates/WMS_ServiceException.tpl")

The above code returns an XML created from the template, */local/gsky/share/gsky/templates/WMS_ServiceException.tpl*. It simply tells Terria “no such layer on this server”, if an attempt is made to access a non-existing layer. (check! unsure when such situation arises)

In addition to using the info from XML to handle a call to non-existing layer, this call displays a legend image for the color scheme to represent the bands.


100% Bare Soil (BS)
100% Photo-synthetic Vegetation (PV)
100% Non Photo-synthetic Vegetation (NPV)

4. GetMap

This is what gets and displays the tiles over the map. Depending on the zoom level, there will be several requests to GetMap in one call. To create these request URLs, a template URL is produced first. In the template, the ‘bbox’ values are replaced for each tile.

In TerriaJS-specs.js:

58026: function UrlTemplateImageryProvider(options) {

57999: url : 'this.url?tiled=true&' + 'transparent=true&format=image%2Fpng&exceptions=application%2Fvnd.ogc.se_xml&' +

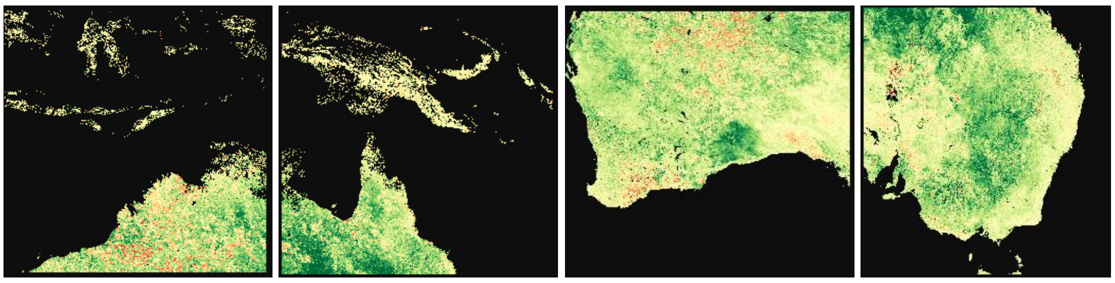
```
'styles=&service=WMS&version=1.1.1&request=GetMap&' +      'layers=
global%3Ac6%3Amonthly_anom_frac_cover &srs=EPSG%3A3857&' +
'bbox={westProjected}%2C{southProjected}%2C{eastProjected}%2C{northProjected}&' +
'width=256&height=256'
58631:    function westProjectedTag(imageryProvider, x, y, level) {
58632:        computeProjected(imageryProvider, x, y, level);
58633:        return projectedScratch.west;
58636:    function southProjectedTag(imageryProvider, x, y, level) {
58641:    function eastProjectedTag(imageryProvider, x, y, level) {
58646:    function northProjectedTag(imageryProvider, x, y, level) {

To GSKY:
http://130.56.242.15/ows/geoglam?time=2018...&srs=EPSG:3857&...&format=image/png&...&service=WMS&...
&request=GetMap&layers=global...&bbox=12...,,,&width=256&height=256
```

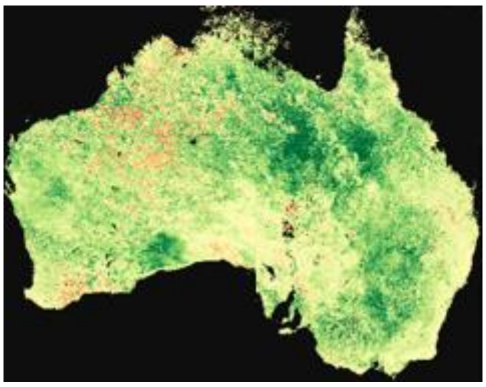
In ows.go:

```
func serveWMS(ctx context.Context, params utils.WMSParams, conf *utils.Config, reqURL string, w
http.ResponseWriter) {
case "GetMap":
 geoReq := &proc.GeoTileRequest{ConfigPayLoad: proc.ConfigPayLoad{NameSpaces:
styleLayer.RGBExpressions.VarList,
 indexer := proc.NewTileIndexer(ctx, conf.ServiceConfig.MASAddress, errChan)
 go indexer.Run(*verbose)
 tp := proc.InitTilePipeline(ctx, conf.ServiceConfig.MASAddress, conf.ServiceConfig.WorkerNodes,
 conf.Layers[idx].MaxGrpcRecvMsgSize, conf.Layers[idx].WmsPolygonShardConcLimit,
 conf.ServiceConfig.MaxGrpcBufferSize, errChan)
 case res := <-tp.Process(geoReq, *verbose):
 out, err := utils.EncodePNG(norm, styleLayer.Palette)
 w.Write(out)
```

The calls like above results in tiles returned as below, which are put together to form layers on the map.

http://130.56.242.15/ows/geoglam?time=20...&service=WMS&request=GetMap&layers=global&bbox=15&
http://130.56.242.15/ows/geoglam?time=20...&service=WMS&request=GetMap&layers=global&bbox=15&
http://130.56.242.15/ows/geoglam?time=20...&service=WMS&request=GetMap&layers=global&bbox=15&
http://130.56.242.15/ows/geoglam?time=20...&service=WMS&request=GetMap&layers=global&bbox=15&





5. GetFeatureInfo

When clicked anywhere on the map, Terria displays the feature info such as X and Y co-ordinates, time, band name and the Latitude/Longitude values. This done by sending a request= GetFeatureInfo to the GSKY server as below.

In TerriaJS-specs.js:
Terria code is yet to be studied and understood!

To GSKY:

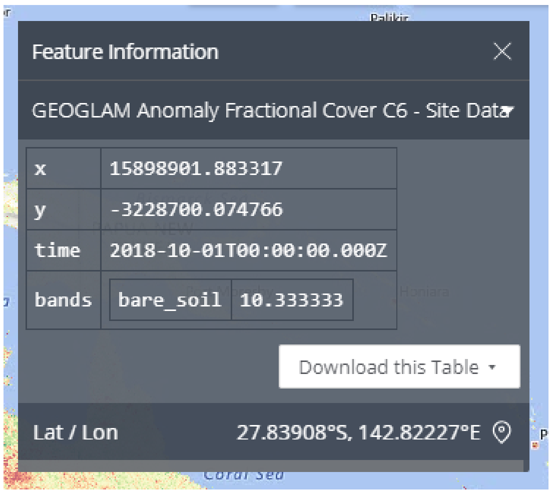
http://130.56.242.15/ows/geoglam?time=2018service=WMS&request=GetFeatureInfo&layers=global&bbox=&query_layers=global&x=89&y=74&Fjson

In ows.go:

```
func serveWMS(ctx context.Context, params utils.WMSParams, conf *utils.Config, reqURL string, w
http.ResponseWriter) {
case "GetFeatureInfo":
        x, y, err := utils.GetCoordinates(params)
        feat_info, err := proc.GetFeatureInfo(ctx, params, conf, *verbose)
        resp := fmt.Sprintf(`{"type":"FeatureCollection","features":[{"type":"Feature","properties":{"x":%f,
"y":%f, %s, %s}}]}`, x, y, timeStr, feat_info)
        w.Write([]byte(resp))
```

The info is sent back and displayed as given below:

```
{"type":"FeatureCollection","features":[{"type":"Feature","properties":{"x":15898901.883317, "y":-
3228700.074766, "time": "2018-10-01T00:00:00.000Z", "bands": {"bare_soil": 10.333333}}}]}
```


Feature Information

GEOGLAM Anomaly Fractional Cover C6 - Site Data

x	15898901.883317
y	-3228700.074766
time	2018-10-01T00:00:00.000Z
bands	bare_soil 10.333333

Download this Table

Lat / Lon 27.83908°S, 142.82227°E

OBSERVATIONS

1. During the compilation of the GSKY server, by 'build_all.sh', the Git repository is cloned into the local server. If using a forked repo instead of the main repo at nci/gsky, it still clones and uses the main repo for the packages (see below). It can confuse the development process.

GSKY compilation uses these files:
/local/gsky/gopath/src/github.com/asivapra/gsky/ows.go
/local/gsky/gopath/src/github.com/nci/gsky/utils/*.go

It can, presumably, be changed in gsky/Makefile.in, but must remember to change it back before pushing to the Github repo.

```
gsky/Makefile.in:
BASEPATH=github.com/nci/gsky
```

Alternatively, if debugging any of the packages other than the main (ows.go), change them in the src directory, */local/gsky/gopath/src/github.com/nci/gsky/utils*. Then, before pushing to the forked repo, be sure to copy the files across to your repo location (e.g. */home/900/avs900/gsky/utils*). The changed files in *gsky/utils* will not be available for the next GSKY build unless they have been merged with the main repo. For the sake of sanity, it is better to change the Makefile.in to use the specified repo, instead of hard coding it as *nci/gsky*.

2. In the Terria code there are apparently duplicated functions as shown below. Cannot determine whether it is an error or on purpose. Perhaps we could ask the developers!

```
45807: function updateInfoSection(item, overwrite, sectionName, sectionValue) {
119431: function updateInfoSection(item, overwrite, sectionName, sectionValue) {

45823:   function updateValue(item, overwrite, propertyName, propertyValue) {
119447: function updateValue(item, overwrite, propertyName, propertyValue) {

There are more!
```

CONCLUSION

The above description is what happens with each 'GetWMS' request. There will be hundreds of such requests from one layer being displayed. Each sends one tile to the web.

Though the process is accurately described to the best of my current understanding of the program, there may be gaps. In particular, I do not yet have access to the MAS server that makes the call to the database. Its code and process will be added later.