**GSKY (OWS) server**

* Setup of server, database and datasets -

By

Arapaut V. Sivaprasad

30 Nov – 4 Dec 2018

**FOREWORD**

This document describes the code base of the GSKY server.

**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.

# TL;DR

* The GSKY server accepts http requests for WMS, WPS and WCS.
  + The WMS service is described in this document.
* Three 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 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.

# INTRODUCTION

The GSKY server is what allows a web0based 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, ‘gsky\_server.pptx’, gives animated description of the steps. This document is an extension of it with more details and explanations for reference.

# HIGH LEVEL PROCESS FLOW

* Animated display of process flow: [GSKY\_OWS\_Server.ppsx](https://github.com/asivapra/gsky/blob/master/Documents/ows/GSKY_OWS_Server.ppsx) and [GSKY\_OWS\_Server.pptx](https://github.com/asivapra/gsky/blob/master/Documents/ows/GSKY_OWS_Server.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.

## 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  
 - confMap  
 - reWMSMap - reWCSMap  
 - reWPSMap

utils.**EtcDir** = \*serverConfigDir

utils.**DataDir** = \*serverDataDir

var EtcDir = /usr/local/etc/

var DataDir = /usr/local/share/gsky@

* 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 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.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

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](https://golang.org/pkg/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](https://golang.org/pkg/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] 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)

# 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.