GEOOs-Portal

Introducción

En este documento se describe el diseño del servicio Portal de GEOOs. Está orientado a desarrolladores de software que deseen construir Plugin para agregar nuevas funcionalidades al Portal, o colaborar en mejoras o correcciones de errores dentro del mismo Portal.

El portal utiliza como backend un microservicio Node JS, mientras que el frontend está desarrollado en JavaScript, diseñado y construido en base a una biblioteca de componentes denominada ZVC. La principal ventaja de esta biblioteca es que permite el desarrollo de los componentes de GUI (paneles, editores, diálogos) en forma independiente unos de otros, comunicados mediante propiedades y eventos.

La comunicación entre el frontend y el backend es mediante servicios HTTP / JSON, utilizando un pequeño wrapper (z-server) que actúa de dispatcher en el backend. La comunicación entre el frontend y los otros componentes de GEOOs (GeoServer y ZRepo) se realiza también usando protocolo HTTTP / JSON y se han encapsulado en clases JavaScript cliente, para simplificar el diseño y facilitar su reutilización.

Las principales dependencias del Portal de GEOOs son:

- Backend
  - Node JS – 14
  - MongoDB 4.4.1
- Frontend
  - Bootstrap 4
  - Leaflet-1.7.1
  - Echarts(echarts-gl)[Gráficos 3D, https://github.com/ecomfe/echarts-gl]
  - Highcharts-8.2.2 [https://www.highcharts.com]
  - ZVC-0.95 [https://github.com/otrojota/zvc]

Ambiente de Desarrollo

Debido a la facilidad de uso y rapidez en los ciclos de desarrollo y pruebas, se propone configurar un ambiente de desarrollo basado en Docker Swarm. Mediante el uso del package “nodemon” de NodeJS, es posible reiniciar automáticamente los servicios del backend que se ven afectados al modificarse (guardarse los archivos) que los implementan, es decir no necesita de bajar y subir los servicios manualmente. Por otra parte, gracias al uso de ZVC, los cambios en las interfaces de usuario HTML y sus controladores JS, se ven reflejados automáticamente al cargarse las páginas web o seleccionarse los menús que cargan dinámicamente los componentes (vistas) modificados.

El ambiente de desarrollo del componente portal de GEOOs, debe contener al menos los servicios necesarios para ejecutar este componente (Portal), junto a aquellos que se desee agregar. Se requiere un stack con el servicio de MongoDB, el servicio del GEOOs-Portal y un tercer servicio para implementar un plugin, en este caso será un plugin de prueba que se usará durante el resto de este documento.

A diferencia de la instalación presentada en el documento de "geoos-componentes-instalación", acá se muestra cómo instalar un ambiente de desarrollo, en donde se trabajará sobre los archivos fuentes de los productos. En el caso del documento referenciado, se utilizaban imágenes Docker ya empaquetadas con cada componente (portal, en este caso). Acá se mostrará cómo utilizar una imagen base con Node JS que apunte (usando un volumen Docker tipo binding) al directorio con los archivos fuentes de nuestros proyectos.

Se propone crear un directorio base para alojar a los proyectos que se usarán, por ejemplo "**/geoos**" (bajo la carpeta del usuario activo, dependiendo del sistema operativo)

Los proyectos de ejemplo y utilitarios para crear el ambiente se encuentran compartidos en GitHub. En particular, se utilizará el proyecto con los archivos fuentes del componente Portal, uno con configuraciones de prueba de docker-swarm y otro con el plugin de demostración que se trabajará en este documento.

Desde el directorio "geoos", ejecutar

- git clone https://github.com/geoos/swarm-examples.git
- git clone https://github.com/geoos/portal.git
- git clone https://github.com/geoos/sample-plugin.git

Los dos últimos son proyectos Node, por lo que deben ser inicializados (descarga de las dependencias). Para ello, se debe ingresar a cada uno de sus subdirectorios (por ejemplo, "portal" creado en el paso anterior) y ejecutar

- npm install

Para que el stack de servicios de Docker funcione sin modificar los archivos de ejemplo que se proveen, se debe crear un directorio "d**ev-data**" bajo "**geoos**". Dentro de este nuevo directorio, crear además dos subdirectorios: "**config**" y "**mongo-data**"

Como sólo modificaremos el componente portal, y cómo GEOOs soporta configuraciones distribuidas, acá se usará el Portal apuntando a los componentes GeoServer y zrepo desde la instalación oficial en geoos.org.

Para ello, usamos un archivo de configuración "**geoos/dev-data/config/portal.hjson**" que referencia a esos servicios remotos:

```
{
```

```
webServer:{http:{port:8090}}
#Available base Maps
maps:[{
code:"esri-world-physical", name:"Esri - World Physical Map"
url:'https://server.arcgisonline.com/ArcGIS/rest/services/World_Physical_Map/MapServer/tile/{
z}/{y}/{x}'
options:{
attribution: 'Tiles &copy; Esri &mdash; Source: US National Park Service'
maxZoom: 8
}
}, {
code:"open-topo", name:"OSM - Open Topo"
url:'https://{s}.tile.opentopomap.org/{z}/{x}/{y}.png'
options:{
maxZoom: 17
attribution: 'Map data: &copy; <a
href="https://www.openstreetmap.org/copyright">OpenStreetMap</a> contributors, <a
href="http://viewfinderpanoramas.org">SRTM</a> | Map sty>'
}
}, {
code:"esri-world-imagery", name:"Esri - World Imagery"
url:'https://server.arcgisonline.com/ArcGIS/rest/services/World_Imagery/MapServer/tile/{z}/{y}
/{x}'
options:{
attribution: 'Tiles &copy; Esri &mdash; Source: Esri, i-cubed, USDA, USGS, AEX, GeoEye,
Getmapping, Aerogrid, IGN, IGP, UPR-EGP, and the GIS User Community'
}
}, {
code:"stamen-terrain", name:"Stamen - Terrain"
url:'https://stamen-tiles-{s}.a.ssl.fastly.net/terrain/{z}/{x}/{y}{r}.{ext}'
options:{
attribution: 'Map tiles by <a href="http://stamen.com">Stamen Design</a>, <a
href="http://creativecommons.org/licenses/by/3.0">CC BY 3.0</a> &mdash; Map data
&copy; <a href>'
subdomains: 'abcd',
minZoom: 0,
maxZoom: 18,
ext: 'png'
}
}, {
code:"esri-world-street", name:"Esri - World Street Map",
url:'https://server.arcgisonline.com/ArcGIS/rest/services/World_Street_Map/MapServer/tile/{z}/
{y}/{x}',
options:{
```

```
attribution: 'Tiles &copy; Esri &mdash; Source: Esri, DeLorme, NAVTEQ, USGS, Intermap, iPC, NRCAN, Esri Japan, METI, Esri China (Hong Kong), Esri (Thailand), TomTom, 2012'
}
}, {
code:"esri-world-terrain", name:"Esri - World Terrain",
url:'https://server.arcgisonline.com/ArcGIS/rest/services/World_Terrain_Base/MapServer/tile/{z}/{y}/{x}',
options:{
attribution: 'Tiles &copy; Esri &mdash; Source: USGS, Esri, TANA, DeLorme, and NPS',
maxZoom:13
}
}, {
code:"esri-ocean-basemap", name:"Esri - Ocean Base Map",
url:'https://server.arcgisonline.com/ArcGIS/rest/services/Ocean_Basemap/MapServer/tile/{z}/{y}/{x}',
options:{
attribution: 'Tiles &copy; Esri &mdash; Sources: GEBCO, NOAA, CHS, OSU, UNH, CSUMB, National Geographic, DeLorme, NAVTEQ, and Esri',
maxZoom:13
}
}, {
code:"esri-natgeo", name:"Esri - NatGeo World Map",
url:'https://server.arcgisonline.com/ArcGIS/rest/services/NatGeo_World_Map/MapServer/tile/{z}/{y}/{x}',
options:{
attribution: 'Tiles &copy; Esri &mdash; National Geographic, Esri, DeLorme, NAVTEQ, UNEP-WCMC, USGS, NASA, ESA, METI, NRCAN, GEBCO, NOAA, iPC',
maxZoom:16
}
}, {
code:"wikimedia", name:"Wikimedia",
url:'https://maps.wikimedia.org/osm-intl/{z}/{x}/{y}{r}.png',
options:{
attribution: '<a
href="https://wikimediafoundation.org/wiki/Maps_Terms_of_Use">Wikimedia</a>',
minZoom:1,
maxZoom:19
}
}]
plugins:["base-plugin"]
# Here
hereAPIKey:"eG2NCop0Fa_k1JUQUaiZvkt-pQKlSlYODS_UwKqaD6c"
# Servers
geoServers:["https://geoserver.geoos.org"]
zRepoServers:[{url:"https://zrepo.geoos.org", token:"geoos-public"}]
```

```
# Variables Clasification
varSubjects:[{
code:"meteo", name:"Metereología"
}, {
code:"oce", name:"Oceanografía"
}, {
code:"geo", name:"Geología"
}, {
code:"geopolitica", name:"División Geopolítica"
}, {
code:"senso", name:"Datos desde Senso"
}]
varTypes:[{
code:"sat", name:"Telemetría Satelital"
}]
varRegions:[{
code:"centro", name:"Zona Centro"
}]
}
```

En el proyecto descargado con git llamado "**swarm-examples**" se entrega una configuración de stack de Docker que está preparada para proveer el ambiente que se requiere para el desarrollo de un plugin del portal, o para modificar directamente el proyecto portal. Dentro de este proyecto, se usará el archivo "**devel/geoos-portal.yml**" el que levanta los 3 servicios requeridos:

```
version: '3.6'
services:
db:
  image: mongo:4.4.1-bionic
  environment:
    MONGO_INITDB_ROOT_USERNAME: geoos-admin
    MONGO_INITDB_ROOT_PASSWORD: geoos-password
  volumes:
    - ./../../dev-data/mongo-data:/data/db

portal:
  image: node:14-alpine
  deploy:
  restart_policy:
  condition: on-failure
  ports:
    - 9234:9234/tcp # nodemon debugger
```

```
    - 8091:8090/tcp
  working_dir: /usr/src/app
  environment:
    MONGO_URL: "mongodb://geoos-admin:geoos-password@db:27017"
    MONGO_DATABASE: geoos
  volumes:
    - ./../../portal:/usr/src/app
    - ./../../dev-data/config:/home/config
  command: "npm run-script debug"

plugin:
  image: node:14-alpine
  ports:
    - 9235:9235/tcp # nodemon debugger
    - 8092:8090/tcp
  working_dir: /usr/src/app
  volumes:
    - ./../../sample-plugin:/usr/src/app
        command: "npm run-script debug"
```

Como se observa en el archivo anterior, tanto el servicio del portal como el del plugin se levantan como una imagen Docker vacía de NodeJS-14 apuntando a las fuentes de cada uno de los proyectos, y ejecutando un script de inicio llamado "**debug**". Además de exponerse los puertos web (8091 y 8092) se publican los puertos 9234 y 9235 para proveer de depuración remota hacia los backends de cada uno de estos dos servicios.

Los scripts "**debug**" y "**start**" (este último para producción) se encuentran definidos en el archivo "**package.json**" dentro de estos dos proyectos. El inicio en modo "debug" ejecuta el comando "nodemon" indicando los directorios que se desea monitorear. Se debe recordar que este producto permite reiniciar los servicios al encontrarse cambios en los archivos con el código fuente, lo que nos permite modificar y probar rápidamente los cambios dentro del contenedor Docker sin reiniciarlos, ya que éstos ejecutan las aplicaciones apuntando directamente al código fuente que se modifica (mediante un volumen tipo binding mapeado a **/usr/src/app** dentro de cada contenedor Docker en ejecución).

Para ejecutar el stack de servicios, desde el directorio **geoos/swarm-examples/devel** se debe ejecutar el comando:

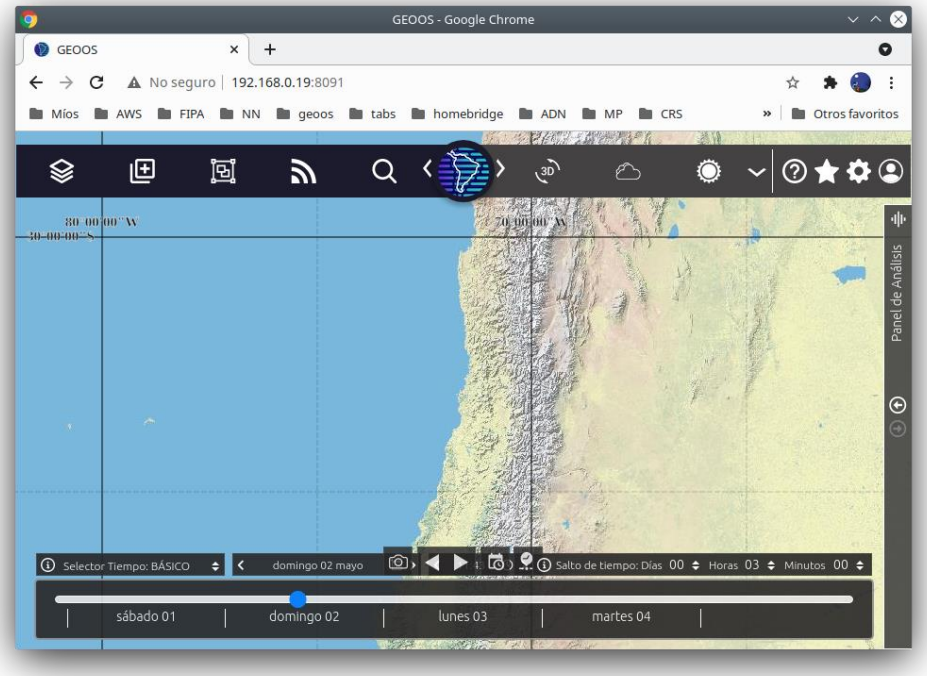
- Docker stack deploy -c geoos-portal.yml geoos

Es muy importante que el comando se ejecute desde dentro del directorio indicado, ya que las rutas definidas en el archivo YML son relativas a la ubicación del mismo archivo.

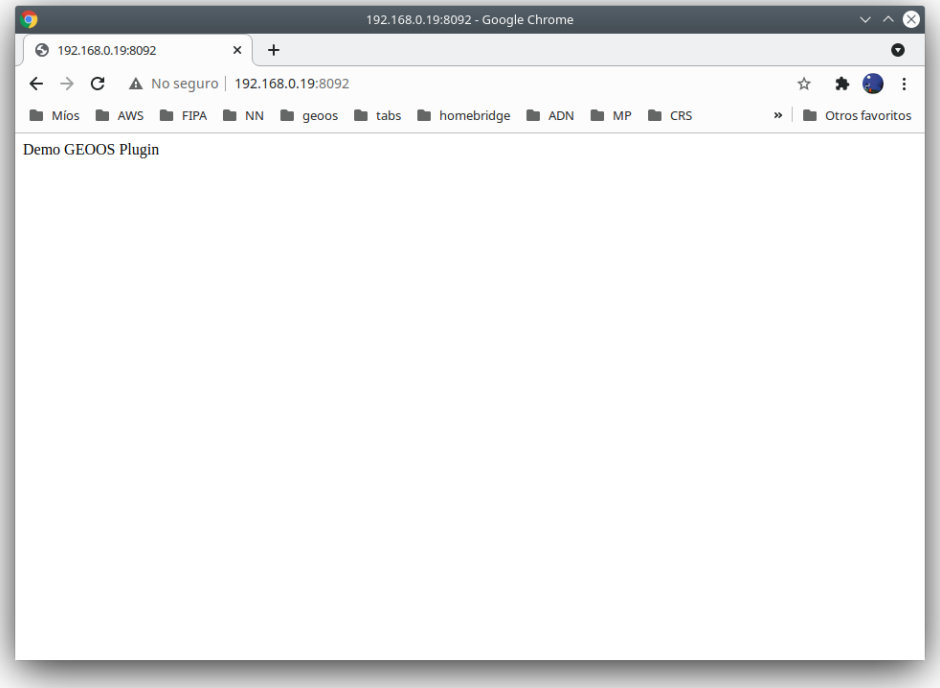
Es posible determinar que los servicios se han levantado correctamente, mediante el comando:

- docker service ls

Una vez que se han levantado, es posible verificarlo apuntando el browser a la dirección del servidor (o máquina de desarrollo) y cada puerto mapeado, por ejemplo:



De la misma forma, el componente plugin se levanta en el puerto 8092:



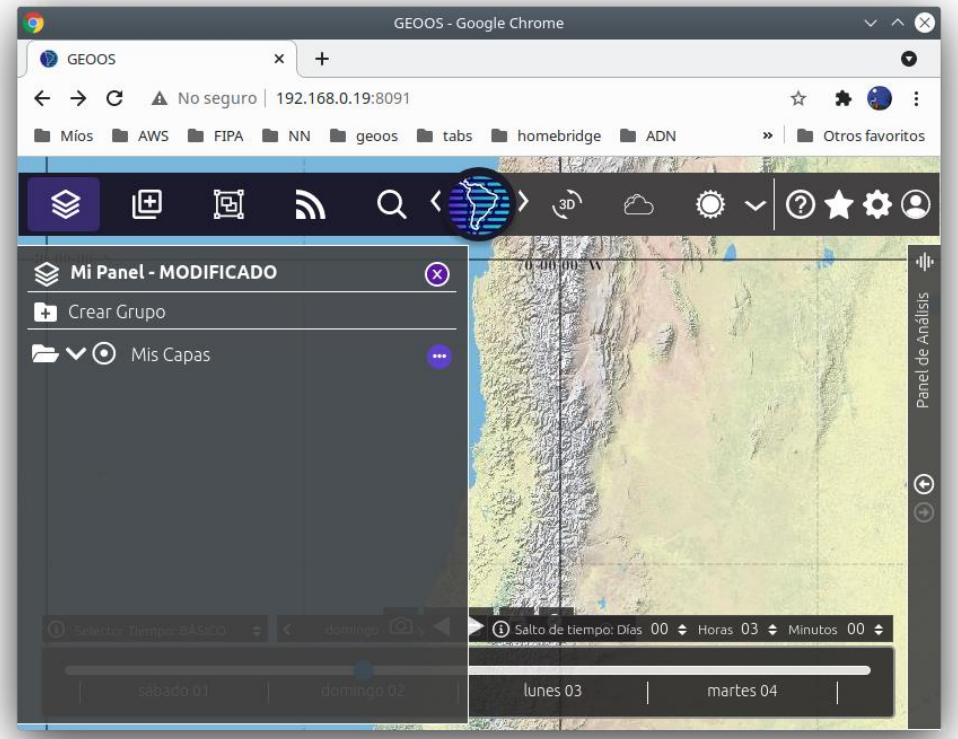
Con los pasos anteriores completados, ya se encuentra habilitado el ambiente de desarrollo del portal y un plugin de ejemplo. El resto de este documento trabajará sobre los componentes acá instalados.

Diseño del Portal

El Portal GEOOs se basa en la biblioteca ZVC, la que provee de un mecanismo de diseño y construcción de interfaces web basadas en componentes. Una de las principales ventajas de esta biblioteca es que permite el diseño de componentes de GUI en forma separada, como una Vista (HTML) y un Controlador (JS) que representan una sección de la página web (aplicación de una página). ZVC se incluye como una dependencia en el archivo index.html y desde ahí se continúan cargando dinámicamente los componentes (vistas y controladores) a medida que se vayan necesitando. No se requiere de una etapa previa de compilación o armado de “bundles” de javascript con la aplicación (a diferencia de otras bibliotecas de GUI como React o Vue). En particular, este mecanismo de carga de los HTML y JS a medida que se van necesitando desde la aplicación, permite que los componentes de interfaz de usuario puedan ser provistos por otros servicios diferentes al portal inicial desde donde se carga la aplicación (de acuerdo con todas las políticas de seguridad tipo CORS y mezcla de HTTP con HTTPS que provean los navegadores actuales).

El portal GEOOs cuenta como cualquier aplicación web estándar con una página (única) de entrada llamada "**index.html**" desde la cual se incluyen sus dependencias y se crea el componente ZVC inicial Main, desde donde se comienza a cargar el resto.

La estructura de directorios bajo "**www/main**" en el proyecto Portal representa los principales paneles que interactúan en GEOOs. Por ejemplo, podemos abrir el archivo "**www/main/myPanel/MyPanel.html**" y modificar el título del panel. Basta grabar la modificación y recargar la página de Geoos para ver el cambio:



En la aplicación web existe un objeto global accesible desde todos los paneles, llamado "**geoos**". Usando este objeto global, es posible acceder a propiedades del portal, registrar eventos, consultar la configuración, etc. El código de este objeto está en "**www/js/geoos.js**". La única instancia de este objeto queda disponible usando el objeto Windows, de la forma "**window.geoos**".

Los paneles principales del portal se registran en este objeto global al momento de inicializarse. Por ejemplo, el objeto "**window.geoos.analysisPanel"** referencia al controlador del panel de análisis (gráficos, de la zona inferior). El código de este controlador está en "**www/main/analysisPanel/AnalysisPanel.js**". Como se observa en el evento "onThis_Init" que se ejecuta al inicializarse este componente, él se registra en el objeto window.geoos usando:

“**window.geoos.analysisPanel = this;**”. De esta forma, los paneles principales de GEOOs están disponibles como variables globales desde cualquier otro elemento del frontend.

Usando las herramientas para el desarrollador de un browser como Chrome, es posible, con el portal GEOOs en ejecución, utilizar la consola javascript y escribir “**window.geoos**”, lo que nos despliega todos los atributos y funciones declaradas en el objeto global.

Diseño de un nuevo Plugin para el Portal GEOOs

El Portal GEOOs posee varios puntos de extensión que pueden ser usados desde plugins. Esto permite tener instalaciones del portal a medida, para resolver requerimientos particulares de cada situación. Un Plugin de Portal GEOOs es microservicio que provee al menos la publicación por HTTP GET de un archivo “**plugin.js**”.

Dentro del archivo **plugin.js** se debe definir una clase que extienda a **GEOOSPlugin** y retorne un objeto (única instancia) de esta clase. Cada Plugin de GEOOs se identifica por un código único. En este caso, usaremos “**demo**”. El plugin es además responsable de definir (y seguramente proveer) los archivos (js y css) que deben cargarse en el portal al iniciarse éste.

Para agregar el nuevo plugin al portal, creamos el archivo “**www/plugin.js**” dentro del proyecto (microservicio) “**sample-plugin**”, sobrescribiendo las propiedades “getter” que definen su código y la lista de archivos a importar en el portal (inicialmente vacía).

```
class GEOOSDemoPlugin extends GEOOSPlugIn {
  get code() {return "demo"}
  get includeFiles() {return []}
}
(new GEOOSDemoPlugin())
```

Para verificar que el nuevo archivo está bien publicado, podemos entrar por el browser a la IP del servidor con la siguiente ruta: http://IP-Servidor:8092/plugin.js y verificar que se descarga el código del plugin.

La lista de plugins que debe cargarse al iniciarse el portal, se configura en el archivo "**portal.hjson**" dentro de la carpeta de configuraciones de GEOOs: **"geoos/dev-data/config/portal.hjson"**

El Portal GEOOs incluye un plugin base con los paneles, herramientas de análisis y visualizadores básicos disponibles en cualquier instalación e incluidos en el proyecto "**portal**". En el archivo de configuración indicado, se debe buscar la línea que declara el atributo "plugins" como un arreglo JSON de strings, y agregar la ruta al nuevo plugin que deseamos cargar. La línea final dentro del archivo "**portal.hjson**" debería quedar como sigue:

- plugins:["base-plugin", "http://IP-Servidor:8092/"]

La aplicación web del portal realizará un request "GET" hacia la url configurada como plugin, añadiendo "**/plugin.js**" al final de ésta. Es necesario que el microservicio que publica los contenidos del plugin soporte publicación mediante los protocolos definidos para CORS, ya que corresponde a dominios diferentes. En el ejemplo descargado se utiliza el package Node llamado CORS, el que implementa estas funcionalidades (encabezados HTTP) como parte de la cadena de acciones de express.

Análisis - Calculador de superficie de un área

El primer punto de extensión del portal es el área de análisis (zona inferior, al seleccionarse un objeto en el mapa), en donde se muestran las series de tiempo y el panel de propiedades de un objeto. Esta área se abre automáticamente al seleccionarse un objeto vectorial en el mapa, el que puede ser un objeto de usuario (punto o área) o un polígono u otro objeto vectorial contenido en una capa (vectorial) agregada al mapa.

Para efectos de ejemplificar la creación de paneles de análisis dentro de los plugins del portal GEOOs, agregaremos un panel que desplegará en área (en metros cuadrados) de un objeto "área rectangular" agregado por el usuario al panel.

Dentro del proyecto “**sample-plugin**” creamos la carpeta “**www/analysis**” y dentro de ella el archivo: “**calculador-superficie.js**”

Editamos el archivo “**plugin.js**” con la lista de archivos que requiere nuestro plugin y agregamos “**analysis/calculador-superficie.js**”. Las urls de los archivos de la lista son relativas al archivo inicial “**plugins.js**” que los declara.

La clase del plugin queda como sigue:

```
class GEOOSDemoPlugin extends GEOOSPlugIn {
    get code() {return "demo"}
    get includeFiles() {return ["analysis/calculador-superficie.js"]}
}
(new GEOOSDemoPlugin())
```

Los archivos declarados en cada plugin son descargados y ejecutados (si son “js”) en el browser. Para probar que la declaración del plugin está funcionando correctamente, podemos agregar como único contenido del archivo “**calculador-superficie.js**” el siguiente contenido:

```
console.log(“Plugin DEMO cargado”);
```

El texto ingresado debería aparecer en la Consola JavaScript (herramientas del desarrollador) del browser.

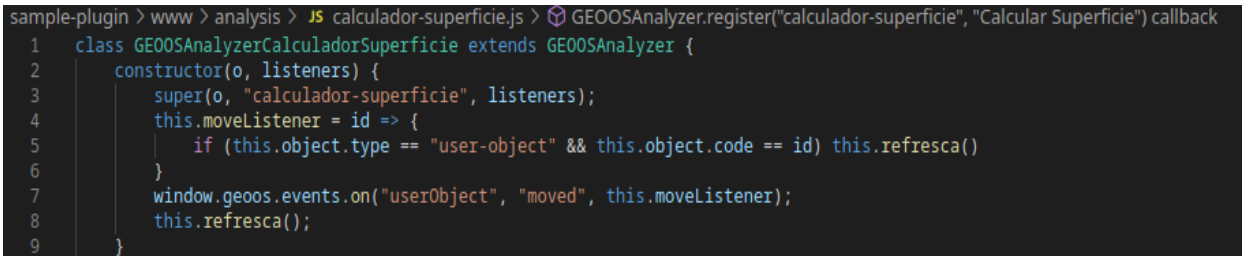
El área de paneles de análisis dentro del portal GEOOs se divide en un selector del análisis, una sección de propiedades y el panel principal.

El selector de análisis permite al usuario escoger desde la lista de análisis que aplican al objeto seleccionado en el mapa, de acuerdo con el tipo de este objeto o alguna otra lógica que cada “analizador” determine. Es responsabilidad de cada “analizador” de GEOOs decidir si es aplicable a un determinado objeto o no.

El área de propiedades corresponde a una sección que el portal construye a partir de una serie de paneles que el analizador declara. Estos paneles (componentes ZVC con su vista y controlador) tienen acceso al analizador para consultar y modificar sus propiedades.

Los analizadores son objetos de una clase que extiende a **GEOOSAnalyzer**. Normalmente se sobrescriben sus métodos constructores, destructor (destroy), el que retorna los paneles de propiedades que aplican, el que retorna el panel principal y la condición que determina si el analizador aplica o no a un objeto determinado, la que se establece al momento de registrarse el analizador.

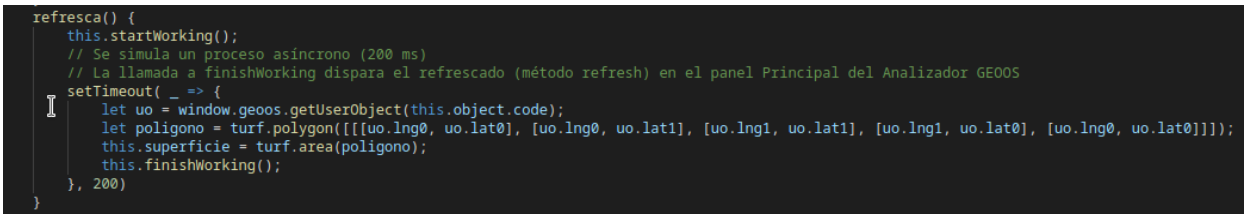
Acá se muestra primero por partes cada sección importante del analizador y luego el contenido completo de los archivos involucrados: en este caso, el analizador con la lógica, junto a la vista (HTML) y controlador (js) con la interfaz ZVC del panel principal.



El constructor de la clase para el nuevo analizador comienza por invocar al constructor de la clase padre (abstracta: GEOOSAnalyzer) con el código del nuevo analizador, el objeto seleccionado en el mapa por el usuario, y la lista de "listeners" configurados para comunicarse con los paneles de la interfaz de usuario.

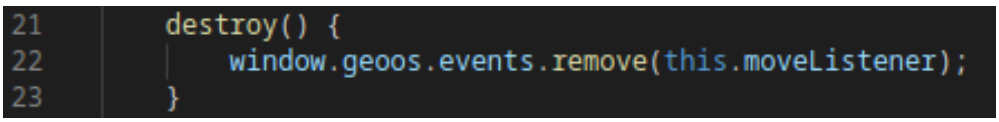
Usando el manejador de eventos globales de GEOOs (**geoos.events**) se registra un escuchador al evento "**moved**" del tipo "**userObject**" y se asocia a un manejador (**moveListener**) definido en el objeto como una función "**arrow**". Es muy importante que, al destruirse el objeto analizador, este escuchador de evento sea removido; de otra forma se podrían provocar errores. Esto se hace luego en el método "**destroy**".

El constructor termina por invocar un refrescado inicial en el objeto para actualizar el área calculada.



El cálculo del área del objeto se simula como una operación asíncrona que toma 200 milisegundos. Esto es para demostrar el uso de **startWorking** y **finishWorking**, los que se sincronizan con el panel principal que muestra los resultados.

El método refresca utiliza la biblioteca "**turf**" que ya está precargada (no es necesario declararla en los archivos que se cargan con el plugin). Se crea un nuevo polígono a partir de las coordenadas del objeto de usuario seleccionado y se almacena el resultado del cálculo de su área como un atributo del mismo analizador. Al finalizarse (**finishWorking**) se refrescará el panel principal, el que consultará ese atributo.



El destructor de este panel se encarga de remover el escuchador registrado en el administrador de eventos de geoos.

```
24      getPropertyPanels() {return []}
25      getMainPanel() {
26          let basePath = window.geoos.getPlugin("demo").basePath;
27          return basePath + "/analysis/calculadorSuperficie/CalculadorSuperficieMain"
28      }
```

Este analizador no declara paneles de propiedades. Su panel principal está dentro del mismo plugin, como el componente ZVC: "**CalculadorSuperficieMain**".

```
31  GEOOSAnalyzer.register("calculador-superficie", "Calcular Superficie",
32      o => {
33          // El analizador aplica sólo a user-objects de tipo "area" (no Puntos)
34          if (o.type != "user-object") return false;
35          let uo = window.geoos.getUserObject(o.code);
36          return uo && uo.type == "area";
37      },
38      (o, listeners) => (new GEOOSAnalyzerCalculadorSuperficie(o, listeners)),
39      250
40  )
```

Al final del código del archivo del analizador, se registra este nuevo componente (GEOOsAnalyzer) en la plataforma. Para ellos se utiliza un método estático de GEOOsAnalyzer llamado "**register**", el que recibe el código único del analizador, un nombre para desplegar (selector de la zona inferior del portal) y una función (**arrow**, en este caso) que determina para qué objetos seleccionados aplica el analizador que se registra. Esta función debe retornar "**true**" o "**false**" a partir del objeto entregado.

Finalmente, se debe pasar como argumento al registrar el analizador una función "**factory**" que cree el nuevo analizador.

Para ver ejemplos más complejos, se recomienda revisar el código en el proyecto portal, bajo "**www/base-plugin/analysis**", en particular el caso de la serie de tiempo.

El código completo del analizador de ejemplo es:

```
class GEOOSAnalyzerCalculadorSuperficie extends GEOOSAnalyzer {
        constructor(o, listeners) {
                super(o, "calculador-superficie", listeners);
                this.moveListener = id => {
                    if (this.object.type == "user-object" && this.object.code == id)
        this.refresca()
                }
                window.geoos.events.on("userObject", "moved", this.moveListener);
                this.refresca();
        }
```

```
        refresca() {
            this.startWorking();
            // Se simula un proceso asíncrono (200 ms)
            // La llamada a finishWorking dispara el refrescado (método refresh) en el
panel Principal del Analizador GEOOS
            setTimeout( _ => {
                let uo = window.geoos.getUserObject(this.object.code);
                let poligono = turf.polygon([[[uo.lng0, uo.lat0], [uo.lng0, uo.lat1],
        [uo.lng1, uo.lat1], [uo.lng1, uo.lat0], [uo.lng0, uo.lat0]]]);
                this.superficie = turf.area(poligono);
                this.finishWorking();
            }, 200)
        }
        destroy() {
            window.geoos.events.remove(this.moveListener);
        }
        getPropertyPanels() {return []}
        getMainPanel() {
            let basePath = window.geoos.getPlugin("demo").basePath;
            return basePath +
        "/analysis/calculadorSuperficie/CalculadorSuperficieMain"
        }
}

GEOOSAnalyzer.register("calculador-superficie", "Calcular Superficie",
        o => {
            // El analizador aplica sólo a user-objects de tipo "area" (no Puntos)
            if (o.type != "user-object") return false;
            let uo = window.geoos.getUserObject(o.code);
            return uo && uo.type == "area";
        },
        (o, listeners) => (new GEOOSAnalyzerCalculadorSuperficie(o, listeners)),
        250
)
```

El componente gráfico ZVC que se usa debe mostrar el área calculada por el analizador. Según se observa en el código anterior, se debe crear en **www/analysis/calculadorSuperficie/CalculadorSuperficieMain.html** y su equivalente (igual nombre) ".js" para el controlador.

El archivo HTML con la vista contiene la definición de una etiqueta y un área para mostrar el resultado:

```
<div class="p-4">
        <div>Superficie Calculada:</div>
```

```
        <h5 id="lblResultado"></h5>
</div>
```

El controlador asociado al panel principal es creado desde el panel con el objeto analizador ya instanciado. El controlador debe definir un método “refresh” que será invocado al finalizarse los cálculos en el analizador (finishWorking).

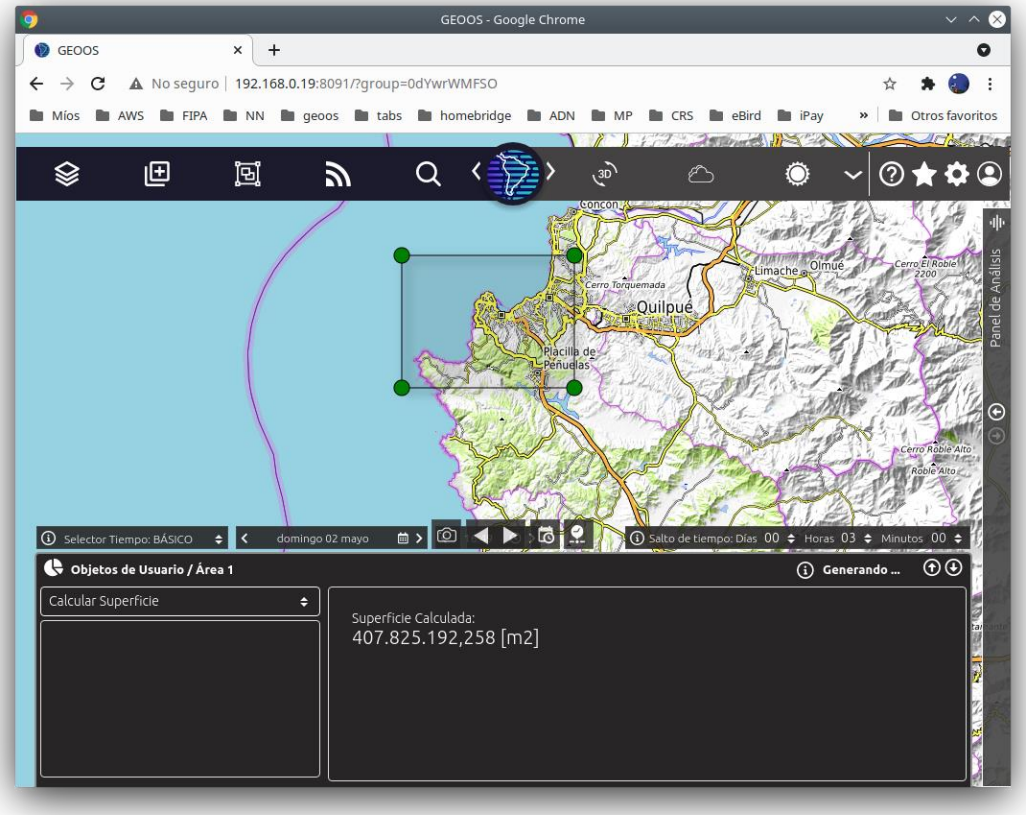
```
class CalculadorSuperficieMain extends ZCustomController {
        onThis_init(options) {
            this.analyzer = options.analyzer;
            this.refresh();
        }

        doResize() {
            // Ejemplo de captura de tamaño
            let size = this.size;
        }

        refresh() {
            if (!isNaN(this.analyzer.superficie)) {
            this.lblResultado.text = this.analyzer.superficie.toLocaleString() + "        [m2]";
            }
            this.triggerEvent("setCaption", "Superficie Calculada");
        }
}
ZVC.export(CalculadorSuperficieMain);
```

Como se observa en el código anterior, el controlador del panel accede a la propiedad “superficie” actualizada en el cálculo del objeto analizador. Debido a que se registró el evento “moved” de los objetos de usuario, cada vez que se mueva o cambie el tamaño del área seleccionada, se recalculará la superficie, y se terminará por refrescar el panel principal (en el finishWorking).

El resultado final debería verse como muestra la siguiente imagen:



Panel de Herramientas – Histograma de una Variable en un Área

Un panel de herramientas GEOOs es similar a un panel de análisis, con la diferencia que no está necesariamente asociado a un objeto seleccionado en el mapa por el usuario. La otra diferencia es que usa un área de la página mucho mayor y se puede agregar a la lista de botones de favoritos de la zona superior.

El plugin base del portal de GEOOs provee las herramientas de análisis de gráfico 3D para una variable y visualizador 3D de terreno y nubosidad por capas. Usando plugins adicionales, se pueden agregar nuevas herramientas de análisis para casos particulares.

A continuación, a modo de ejemplo, se diseñará un panel de análisis que mostrará un histograma con el comportamiento de una variable de capa ráster para todos los puntos contenidos en un área seleccionada en el mapa.

La nueva Herramienta se llamará “**HistogramaVariable**”. Para crearla, agregamos al mismo plugin de ejemplo un archivo nuevo:

```
sample-plugin > www > JS plugin.js > ...
1   class GEOOSDemoPlugin extends GEOOSPlugIn {
2       get code() {return "demo"}
3       get includeFiles() {return ["analysis/calculador-superficie.js", "tools/histograma-variable.js"]}
4   }
5
6   (new GEOOSDemoPlugin())
```

Las Herramientas deben extender a la clase **GEOOSTool**. Al igual que el caso del analizador, las herramientas deben declarar algunas propiedades y sobrescribir algunos métodos para responder a los cambios solicitados por el usuario.

Todas las herramientas deben tener un código único dentro de la plataforma. En este caso usaremos "**histograma-variable**". Esta herramienta funcionará sobre un área definida por un objeto de usuario ya existente en el mapa o que se agregará en el momento.

Se utilizará un objeto histograma ofrecido por Highcharts 8, que ya se encuentra cargado en el portal dentro de las dependencias base.

Las herramientas de análisis deben indicar un panel inicial que permita seleccionar o configurar los elementos necesarios para su funcionamiento (si aplican). En este caso, se usará un panel ya existente en el portal que permite seleccionar un objeto existente o crear uno nuevo y usarlo para el análisis.

Además del panel inicial, se usan dos paneles de propiedades también existentes: uno que permite renombrar la herramienta y otro que permite seleccionar una variable desde una capa ráster.

A continuación, se muestran las secciones del código más importantes de la clase **ToolHistogramaVariable**, contenida en el archivo **www/tools/histograma-variable.js** que provee el proyecto plugin de demostración que se ha desarrollado.

```
sample-plugin > www > tools > JS histograma-variable.js > ToolHistogramaVariable > createDefaultConfig
1   class ToolHistogramaVariable extends GEOOSTool {
2       constructor(id, name, createOptions) {
3           let config = {
4               layerId:createOptions.layerId, object:createOptions.object
5           }
6           super("histograma-variable", id, name, config);
7           this.data = {aborter:null, grid:null}
8           this.timeChangeListener = async _ => await this.timeChanged();
9           this.objectMoveListener = async objectId => await this.objectMoved(objectId);
10      }
11
```

La clase con la lógica y configuración de la nueva herramienta debe extender a **GEOOSTool**. El panel que se usará de creación de la herramienta, y que permite seleccionar un objeto existente o crear uno nuevo, retorna un objeto como el que se recibe en el argumento "**createOptions**" del constructor. Desde ahí se rescata el identificador de la capa y el objeto seleccionado para almacenarse en la configuración, la que es traspasada al constructor de la clase padre.

El objeto “**data**” definido en el constructor permite almacenar los resultados de la consulta que se ejecuta hacia el componente “**GeoServer**” que entrega esa capa, junto a un objeto que permite abortar la consulta en caso de cancelarse la operación.

Se declaran además dos manejadores de eventos para refrescar los valores en función de los cambios en el tiempo y la posición del área de consulta, re ejecutando la consulta a GeoServer.

```
12      get object() {return this.config.object}
13      get layerId() {return this.config.layerId}
14      get layer() {return window.geoos.getActiveGroup().getLayer(this.layerId)}
15      get mainPanelPath() {
16          let basePath = window.geoos.getPlugin("demo").basePath;
17          return basePath + "/tools/histogramaVariable/HistogramaVariableMain"
18      }
```

Se declaran algunas propiedades (getter) que simplifican el acceso a atributos de la configuración. Se indica además la ruta del componente ZVC que actuará como panel principal de la herramienta.

```
20      async activate() {
21          super.activate();
22          window.geoos.events.on("portal", "timeChange", this.timeChangeListener);
23          window.geoos.events.on("userObject", "moved", this.objectMoveListener);
24      }
25      async deactivate() {
26          window.geoos.events.remove(this.timeChangeListener);
27          window.geoos.events.remove(this.objectMoveListener);
28          super.deactivate();
29      }
```

Los eventos son registrados durante la activación y des registrados al momento de destruirse el panel. Esto se maneja utilizando los eventos “**activate**” y “**deactivate**” de ZVC, como se muestra en la figura anterior.

```
31      getPropertyPanels() {
32          return [{
33              code:"tool-props", name:"Nombre del Análisis", path:"./propertyPanels/PropToolName"
34          }, {
35              code:"raster-var", name:"Selección de Variable", path:"./propertyPanels/SelectRasterVariable"
36          }]
37      }
38
39      get caption() {
40          if (this.variable) return this.name + " / " + this.variable.name;
41          return this.name;
42      }
```

La ruta a los paneles de propiedades que usa la herramienta puede ser absoluta (usando la ruta base del plugin como la declaración del panel principal en la línea 15 del código mostrado) o relativa, si esta ruta comienza con un punto. En este caso, para rutas relativas a paneles que provee el portal, la ruta base de estos paneles corresponde al proyecto portal, bajo **www/main/toolsPanel**.

```
44      get variable() {
45          if (this._variable) return this._variable;
46          if (this.config.variable) {
47              this._variable = GEOOSQuery.deserialize(this.config.variable)
48              return this._variable;
49          } else {
50              return null;
51          }
52      }
53      set variable(v) {
54          this._variable = v;
55          this.config.variable = v?v.serialize():null;
56          window.geoos.events.trigger("tools", "renamed", this);
57          window.geoos.events.trigger("tools", "propertyChange", this);
58          this.refresh();
59      }
60
61      createDefaultConfig() {
62          this.config.variable = {type:"raster", id:"default", format:"grid", geoServer:"geoos-main", dataSet:"noaa-gfs4", variable:"TMP_2"}
63      }
```

El panel de propiedades que permite seleccionar una variable de capa ráster necesita que el objeto tool implemente una propiedad llamada “variable”, con un getter y un setter. El objeto variable corresponde a una instancia de la clase GEOOSVariable, la que es utilizada como un generalizador de los diferentes tipos de variables que se utilizan en GEOOs, ya sea desde GeoServer o ZRepo. Para efectos de este ejemplo, basta con saber que encapsula la información necesaria para la invocación al componente (GeoServer) que resolverá la consulta.

Se sobrescribe además el método que permite crear la configuración default o inicial de la herramienta, y que es invocado justo después de finalizar el panel de creación. En este método le indicamos a nuestro objeto que comience inicialmente con la variable “Temperatura” del modelo GFS4 de la NOAA.

```
65      get bounds() {
66          if (this.object.type == "user-object/area") {
67              let area = window.geoos.getUserObject(this.object.code);
68              return {n:area.lat0, s:area.lat1, w:area.lng0, e:area.lng1};
69          }
70          throw "Object type " + this.object.type + " not handled";
71      }
```

La propiedad “bounds” es utilitaria para simplificar la consulta al GeoServer, el que requiere que los límites del área de consulta se entreguen como n,s,w,e

```
    refresh() {
        this.startWorking();
        if (this.data.aborter) {
            this.data.aborter.abort();
            this.data.aborter = null;
        }
        if (!this.variable) this.createDefaultConfig();
        let b = this.bounds;
        let {promise, controller} = this.variable.query({
            format:"grid", n:b.n, s:b.s, w:b.w, e:b.e
        });
        this.data.aborter = controller;
        promise
            .then(res => {
                this.data.aborter = null;
                this.data.grid = res;
                if (this.mainPanel) this.mainPanel.refresh();
                this.finishWorking();
            }).catch(err => {
                this.finishWorking();
                this.data.aborter = null;
                console.error(err)
            })

    }
```

Al igual que los paneles de análisis, las herramientas poseen un par de métodos “startWorking” y “finishWorking” que permiten sincronizar la interfaz de usuario con las consultas y cálculos que se realicen en esta clase. En particular, este método comienza por abortar alguna consulta pendiente (si existe) y luego usar el método “query” del objeto GEOOsQuery (la variable seleccionada). Al finalizarse se refresca el panel principal.

```
    async timeChanged() {
        if (this.variable && this.variable.dependsOnTime) {
            if (this.mainPanel) {
                this.refresh();
            } else {
                this.data.grid = null;
            }
        }
    }

    async objectMoved(objectId) {
        if (this.object.type.startsWith("user-object") && objectId == this.object.code) {
            if (this.mainPanel) {
                this.refresh();
            } else {
                this.data.grid = null;
            }
        }
    }

    async isValid() {
        if (this.object.type.startsWith("user-object/")) return window.geoos.getUserObject(this.object.code)?true:false;
        return true;
    }
}
```

Los manejadores de los eventos de interés (tiempo del mapa cambiado y objeto movido) se encargan de refrescar los datos al detectarse estos eventos.

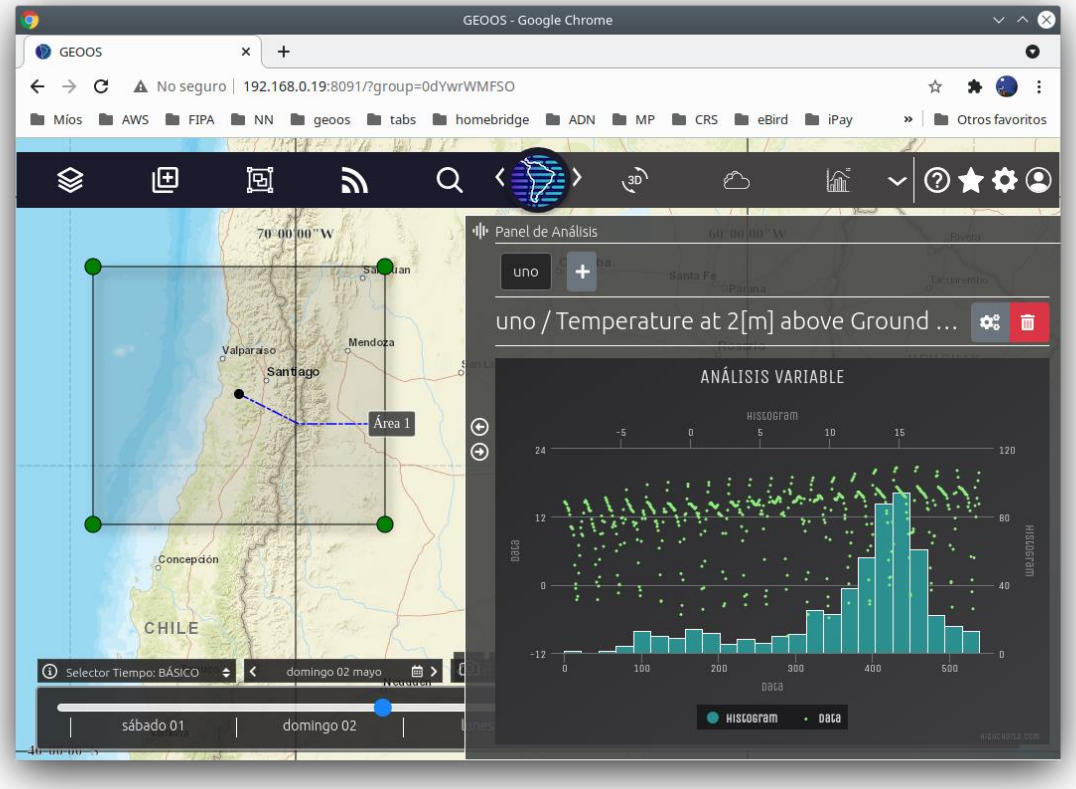
El método "**isValid**" es usado por el portal para determinar si debe mostrar contenido o no para una cierta configuración.

```
124  GEOO (property) creationPanelPath: string 'Histograma Variable en Área", {
125      creationPanelPath:"./creationPanels/ToolObjectSelector",
126      creationPanelOptions:{
127          allowedObjectTypes:["user-object/area"],
128          caption:"Seleccione el Área para el Análisis"
129      },
130      icon:window.geoos.getPlugin("demo").basePath + "/tools/img/histograma.png",
131      menuIcon:window.geoos.getPlugin("demo").basePath + "/tools/img/histograma.svg",
132      menuIconStyles:{filter:"invert(1)"},
133      menuLabel:"Histograma",
134      factory:(name, creationPanelResult) => (new ToolHistogramaVariable(null, name, creationPanelResult)),
135      deserialize:(id, name, config) => {
136          let tool = new ToolHistogramaVariable(id, name, {layerId:config.layerId, object:config.object})
137          tool.config = config;
138          return tool;
139      }
140  })
```

Finalmente, la herramienta debe ser registrada usando un método estático de la clase GEOOsTool. Dentro de los parámetros de registro se debe indicar la ruta del panel creador y su configuración. En este caso, se le indica que se usará el panel de selección de objeto, configurado para objetos de usuario del tipo "area". Una herramienta puede proveer su propio panel de creación si éste está incluido dentro del mismo plugin.

La imagen .svg corresponde al ícono de la barra superior, mientras que la "png" (144x144) es el ícono que se muestra durante la creación de la herramienta.

La siguiente figura muestra un ejemplo de ejecución del panel de análisis creado:



El código completo de la clase ToolHistogramaVariable es:

```
class ToolHistogramaVariable extends GEOOSTool {
  constructor(id, name, createOptions) {
    let config = {
      layerId:createOptions.layerId, object:createOptions.object
    }
    super("histograma-variable", id, name, config);
    this.data = {aborter:null, grid:null}
    this.timeChangeListener = async _ => await this.timeChanged();
    this.objectMoveListener = async objectId =>      await this.objectMoved(objectId);
  }

  get object() {return this.config.object}
  get layerId() {return this.config.layerId}
  get layer() {return window.geoos.getActiveGroup().getLayer(this.layerId)}
  get mainPanelPath() {
    let basePath = window.geoos.getPlugin("demo").basePath;
    return basePath + "/tools/histogramaVariable/HistogramaVariableMain"
  }
```

```
  async activate() {
    console.log("activate hist")
    super.activate();
    window.geoos.events.on("portal", "timeChange", this.timeChangeListener);
    window.geoos.events.on("userObject", "moved", this.objectMoveListener);
  }

  async deactivate() {
    console.log("deactivate hist")
    window.geoos.events.remove(this.timeChangeListener);
    window.geoos.events.remove(this.objectMoveListener);
    super.deactivate();
  }

          getPropertyPanels() {
          return [{
      code:"tool-props", name:"Nombre del Análisis",      path:"./propertyPanels/PropToolName"
}, {
code:"raster-var", name:"Selección de Variable", path:"./propertyPanels/SelectRasterVariable"
}]
}

get caption() {
if (this.variable) return this.name + " / " + this.variable.name;
return this.name;
}

get variable() {
if (this._variable) return this._variable;
if (this.config.variable) {
this._variable = GEOOSQuery.deserialize(this.config.variable)
return this._variable;
} else {
return null;
}
}
set variable(v) {
this._variable = v;
this.config.variable = v?v.serialize():null;
window.geoos.events.trigger("tools", "renamed", this);
window.geoos.events.trigger("tools", "propertyChange", this);
this.refresh();
}
```

```
createDefaultConfig() {
this.config.variable = {type:"raster", id:"default", format:"grid", geoServer:"geoos-main",
dataSet:"noaa-gfs4", variable:"TMP_2"}
}

get bounds() {
if (this.object.type == "user-object/area") {
let area = window.geoos.getUserObject(this.object.code);
return {n:area.lat0, s:area.lat1, w:area.lng0, e:area.lng1};
}
throw "Object type " + this.object.type + " not handled";
}
refresh() {
this.startWorking();
if (this.data.aborter) {
this.data.aborter.abort();
this.data.aborter = null;
}
if (!this.variable) this.createDefaultConfig();
let b = this.bounds;
let {promise, controller} = this.variable.query({
format:"grid", n:b.n, s:b.s, w:b.w, e:b.e
});
this.data.aborter = controller;
promise
.then(res => {
this.data.aborter = null;
this.data.grid = res;
if (this.mainPanel) this.mainPanel.refresh();
this.finishWorking();
}).catch(err => {
this.finishWorking();
this.data.aborter = null;
console.error(err)
})

}

async timeChanged() {
if (this.variable && this.variable.dependsOnTime) {
if (this.mainPanel) {
this.refresh();
} else {
this.data.grid = null;
}
```

```
}
}

async objectMoved(objectId) {
if (this.object.type.startsWith("user-object") && objectId == this.object.code) {
if (this.mainPanel) {
this.refresh();
} else {
this.data.grid = null;
}
}
}

async isValid() {
if (this.object.type.startsWith("user-object/")) return
window.geoos.getUserObject(this.object.code)?true:false;
return true;
}
}

GEOOSTool.register("histograma-variable", "Histograma Variable en Área", {
        creationPanelPath:"./creationPanels/ToolObjectSelector",
        creationPanelOptions:{
                allowedObjectTypes:["user-object/area"],
                caption:"Seleccione el Área para el Análisis"
        },
        icon:window.geoos.getPlugin("demo").basePath +
        "/tools/img/histograma.png",
        menuIcon:window.geoos.getPlugin("demo").basePath +   "/tools/img/histograma.svg",
        menuIconStyles:{filter:"invert(1)"},
        menuLabel:"Histograma",
        factory:(name, creationPanelResult) => (new ToolHistogramaVariable(null,          name,
creationPanelResult)),
        deserialize:(id, name, config) => {
                let tool = new ToolHistogramaVariable(id, name,
        {layerId:config.layerId, object:config.object})
                tool.config = config;
                return tool;
        }
})
```

Como antes se mencionó, las herramientas de GEOOs declaran un Panel como “Main”, que corresponde al área principal de la página que usarán. El área de análisis tiene 3 estados: Minimizada, en donde sólo se muestra el título en el borde derecho del portal, Abierta, en

dónde se muestra sólo el panel “Main” del Tool activo y Maximizada, en donde se agrega el área de configuración del Tool, la que incluye sus paneles de propiedades, declarados en la clase anterior.

La clase **ToolHistogramaVariable** es la encargada de recuperar los datos de la variable para el área y tiempo seleccionados, utilizando un objeto **GEOOsQuery** retornado por el panel de propiedades que se declaró en la misma clase, y que ya existe dentro de los que provee el Portal. El despliegue de la información es responsabilidad del componente ZVC **HistogramaVariableMain**, con su vista (HTML) y controlador (js)

Este componente utiliza la biblioteca Highcharts (ya incluida en el Portal) para desplegar un histograma a partir de los datos originales retornados por GeoServer. La vista del componente declara un “div” que será el contenedor del gráfico del histograma:

sample-plugin > www > tools > histogramaVariable > HistogramaVariableMain.html > ...

```
<div id="mainPanelContainer">
    <div id="histogramaContainer" style="position: absolute;"></div>
</div>
```

El controlador de la clase se encarga de tomar los datos recuperados por la clase Tool y crear o actualizar el gráfico del histograma. El objeto **Tool** asociado le es entregado a este componente dentro de su inicialización (**onThis_init**). Usando los eventos **activate** y **deactivate** de ZVC, este componente debería indicarle a su “Tool” asociado que él (como panel principal) está activo o visible. De esta forma, el objeto **Tool** puede refrescar al componente ZVC cuando tiene disponible sus datos. Esta sección del código del componente es la siguiente:

sample-plugin > www > tools > histogramaVariable > HistogramaVariableMain.js > HistogramaVariableMain

```
class HistogramaVariableMain extends ZCustomController {
    async onThis_init(options) {
        this.tool = options.tool;
    }

    onThis_activated() {this.tool.mainPanel = this;}
    onThis_deactivated() {this.tool.mainPanel = null;}

    doResize() {
        let size = this.size;
        this.mainPanelContainer.size = size;
        this.histogramaContainer.size = {width:size.width, height:size.height - 10};
        if (this.chart) this.chart.setSize(this.histogramaContainer.width, this.histogramaContainer.height);
    }
    async refresh() {
        let grid = this.tool.data.grid;
```

Como se observa en el código anterior, el componente puede además sobrescribir el método **doResize** para ajustar su contenido de acuerdo con cambios en el tamaño del Portal. En este caso, se actualiza el tamaño del gráfico si éste ya está desplegado.

El resto del código del controlador del componente se encarga de crear y actualizar los datos de acuerdo con los resultados ya obtenidos por la clase “**Tool**” asociada.

El código completo del controlador es el siguiente:

```
class HistogramaVariableMain extends ZCustomController {
	async onThis_init(options) {
		this.tool = options.tool;
	}

	onThis_activated() {this.tool.mainPanel = this;}
	onThis_deactivated() {this.tool.mainPanel = null;}

	doResize() {
	  let size = this.size;
	  this.mainPanelContainer.size = size;
	  this.histogramaContainer.size = {width:size.width, height:size.height - 10};
	  if (this.chart) this.chart.setSize(this.histogramaContainer.width,
	this.histogramaContainer.height);
	}

	async refresh() {
	  let grid = this.tool.data.grid;
	  console.log("grid", grid);
	  if (!grid || grid.min == grid.max) return;
	// Recorrer filas y columnas en la matriz retornada
	  let data = [];
	  grid.rows.forEach(row => {
	    row.forEach(v => {
	      if (v !== null && v !== undefined) data.push(v);
	    })
	  });
	  this.chart = Highcharts.chart('histogramaContainer', {
		title: {
		  text: 'Análisis Variable'
		},
		xAxis: [{
		  title: { text: 'Data' },
		  alignTicks: false
		}, {
		  title: { text: 'Histogram' },
		  alignTicks: false,
		  opposite: true
		}],
		yAxis: [{
		  title: { text: 'Data' }
		}, {
		  title: { text: 'Histogram' },
```

```
                    opposite: true
                }],
                plotOptions: {
                    histogram: {
                        accessibility: {
                            pointDescriptionFormatter: point => {
                                let ix = point.index + 1,
                                x1 = point.x.toFixed(2),
                                x2 = point.x2.toFixed(2),
                                val = point.y;
                                return ix + '. ' + x1 + ' to ' + x2 + ', ' + val + '.';
                            }
                        }
                    }
                },
                series: [{
                    name: 'Histogram',
                    type: 'histogram',
                    xAxis: 1,
                    yAxis: 1,
                    baseSeries: 's1',
                    zIndex: -1
                }, {
                    name: 'Data',
                    type: 'scatter',
                    data: data,
                    id: 's1',
                    marker: {
                        radius: 1.5
                    }
                }]
            });
        }
}
ZVC.export(HistogramaVariableMain);
```

Visualizadores Raster

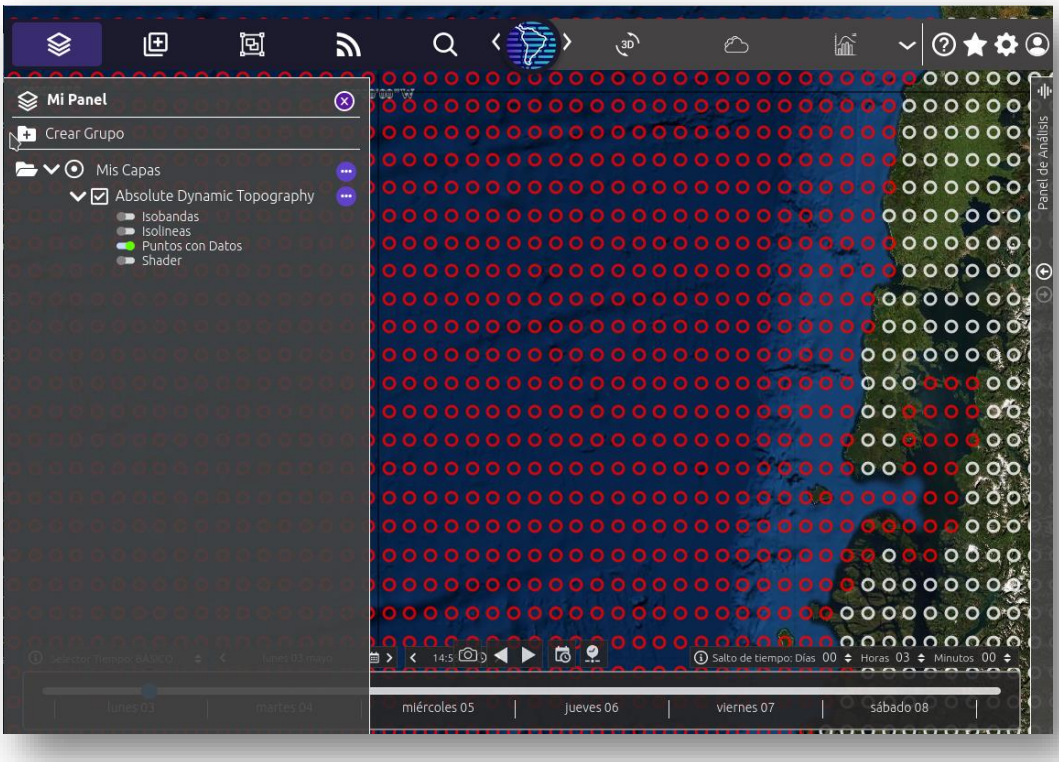
El componente GeoServer del GEOOs maneja datos de tipo “raster”. Estos corresponden a información de valores de variables dentro de una matriz de (al menos) dos dimensiones: latitud y longitud. Los valores pueden ser temporales o valores sin temporalidad.

El portal GEOOs provee algunos visualizadores para los datos de este tipo: Isolíneas, isobandas, raster y, en caso de que los datos sean vectoriales, un visualizador de vectores.

El portal utiliza una biblioteca llamada Konva.js para dibujar sobre el mapa. Específicamente, dentro del portal se defina una Capa Leaflet que permite realizar dibujos a medida en el mapa, en la forma de “**Visualizadores**”. Los visualizadores raster se registran en el portal dependiendo de los tipos de dato (tipo de query) que provee cada capa. En particular, todas las capas raster proveen el formato “grid” de datos, que corresponde a consultar por la matriz de datos de una variable para un momento y límites de coordenadas (rectángulo).

Para crear un nuevo visualizador raster es necesario definir dos clases nuevas: el “helper” que se encarga de consultar los datos a GeoServer y coordinarse con el portal (atender a los eventos de cambio de tiempo o área visible del mapa) y otra clase que tiene la responsabilidad de dibujar los resultados en el mapa.

Como ejemplo, acá se desarrollará un visualizador raster que mostrará los puntos con datos de una capa. Esto es especialmente útil para las variables que aplican sólo a algunas áreas del mapa, como por ejemplo las zonas oceánicas y no terrestres. La siguiente imagen muestra el resultado del visualizador que se desarrollará para la capa de “Topografía Dinámica Absoluta”.



Los puntos con datos se muestran con borde rojo, mientras que aquellos sin datos, con borde blanco.

Para agregar un nuevo visualizador de datos raster en el plugin que tenemos de ejemplo, agregamos dos nuevos archivos a la lista de dependencias que deseamos cargar junto al plugin al iniciarse el portal:

```
sample-plugin > www > JS plugin.js > ...
1   class GEOOSDemoPlugin extends GEOOSPlugIn {
2       get code() {return "demo"}
3       get includeFiles() {return [
4           "analysis/calculador-superficie.js",
5           "tools/histograma-variable.js",
6           "raster-visualizers/dataPoints/data-points.js", "raster-visualizers/dataPoints/DataPointsVisualizer.js"
7       ]}
8   }
```

El archivo “**data-points.js**” ubicado bajo la dirección “**www/raster-visualizers/dataPoints/**” contiene la clase “**DataPointsVisualizerHelper**”, que es la encargada de obtener los datos y coordinarse con el portal, mientras que el archivo “**DataPointsVisualizer**” extiende a **KonvaLeafletVisualizer** y se encarga del pintado de los círculos, según los datos recuperados desde GeoServer.

La clase helper del visualizador extiende a "**RasterVisualizer**" del Portal. Debe ofrecer un método estático que determine si ese visualizador aplica a una cierta capa (de GeoServer) o no. Usando el resultado de esta invocación, el Portal muestra el selector en "Mi Panel"

A continuación, se explican las secciones más importantes de la clase **DataPointsVisualizerHelper**, encarga de la consulta de los datos y coordinación entre la capa gráfica de visualización y el portal.

La función estática "**applyToLayer**" retornará verdadero si la variable asociada a la capa implementa el formato de consulta "grid". Este formato es soportado por todas las variables del tipo raster.

El constructor de la clase recibe el objeto "layer" (de Mi Panel) y una configuración (json) inicial, ya sea preguardada, abierta de un enlace, default o de favoritos. Se almacenan los datos de la configuración inicial y se crea un nuevo objeto "**RasterQuery**" (ver La función estática "applyToLayer" retornará verdadero si la variable asociada a la capa implementa el formato de consulta "grid". Este formato es soportado por todas las variables tipo raster.

El constructor de la clase recibe el objeto "layer" (de Mi Panel) y una configuración (json) inicial, ya sea preguardada, abierta de un enlace, default o de favoritos. Se almacenan los datos de la configuración inicial y se crea un nuevo objeto "RasterQuery" (ver **js/geoos-query.js** y **js/geoserver-client.js**).

Una instancia del Portal de GEOOs puede conectarse simultáneamente a varios GeoServers distribuidos. Por ello, como parte de los datos de la capa seleccionada por el usuario, se tiene el objeto "**layer.geoserver**", el que se utiliza en la consulta para saber hacia quién direccionar el request (HTTP / Rest).

Dentro de un GeoServer, las variables se agrupan en "Datasets", los que presentan una agrupación por origen o temporalidad. Por ejemplo, un Dataset podría ser "noaa-gfs4", el que agrupa las variables del modelo GFS4 generadas por la NOAA.

Además de inicializar la consulta y almacenar su configuración, la clase provee propiedades para exponer su nombre y código.

```
16      async create() {
17          this.visualizer = this.layer.konvaLeafletLayer.addVisualizer(this.code, new DataPointsVisualizer({
18              zIndex:2,
19              onBeforeUpdate: _ => {this.startQuery(); return false}
20          }));
21          this.timeChangeListener = _ => {
22              if (this.layer.config.dataSet.temporality != "none" && this.active) this.startQuery()
23          }
24          window.geoos.events.on("portal", "timeChange", this.timeChangeListener);
25          this.startQuery();
26      }
27      async destroy() {
28          window.geoos.events.remove(this.timeChangeListener);
29          if (this.aborter) {
30              this.aborter.abort();
31              this.aborter = null;
32          }
33          if (this.layer.konvaLeafletLayer) this.layer.konvaLeafletLayer.removeVisualizer(this.code);
34          this.visualizer = null;
35      }
36      update() {
37          if (this.active && this.layer.active && this.layer.group.active) {
38              this.layer.konvaLeafletLayer.getVisualizer(this.code).update();
39          }
40      }
```

Esta clase Helper es la responsable de crear el visualizador gráfico que se acoplará como capa Leaflet al mapa del portal. En este caso, se crea una nueva instancia de un **DataPointsVisualizer**, el que extiende a **KonvaLeafletVisualizer** y se explica más adelante.

El **zIndex** indica al visualizador en Konva el orden de sus capas. Por cada capa de variable raster agregada al Portal, se crea una capa leaflet que contiene múltiples capas Konva. El **zIndex** indica el orden de la capa Konva que junta a todos los visualizadores raster.

El visualizador indica que antes de actualizarse debe invocar a "**startQuery**", método que se encarga de hacer una consulta asíncrona a geoserver (a través de RasterQuery")

Al finalizar la creación, se registra un listener de evento en el portal para actualizar los datos (Re ejecutar la consulta) si la variable que se muestra tiene temporalidad asociada. Este listener debe des registrarse al destruirse el visualizador.

El destructor además cancela cualquier consulta pendiente de resultados desde el GeoServer asociado.

La actualización del Helper es traspasada a la actualización del visualizador gráfico, el que determinará en qué momento y en qué condiciones se actualizará. Para ello, llamará de vuelta (a través del callback "**onBeforeUpdate**") al Helper para que inicie la consulta asíncrona.

```
41      startQuery(cb) {
42          if (this.aborter) {
43              this.aborter.abort();
44              this.finishWorking();
45          }
46          this.startWorking();
47          let {promise, controller} = this.query.query({margin:1, level:this.layer.level});
48          this.aborter = controller;
49          let visualizer = this.layer.konvaLeafletLayer.getVisualizer(this.code)
50          promise
51              .then(ret => {
52                  this.aborter = null;
53                  this.finishWorking();
54                  window.geoos.events.trigger("visualizer", "results", this);
55                  visualizer.setGridData(ret.foundBox, ret.rows, ret.nrows, ret.ncols);
56                  if (cb) cb();
57              })
58              .catch(err => {
59                  this.aborter = null;
60                  if (err != "aborted" && err.toString().indexOf("abort") < 0) {
61                      console.error(err);
62                      this.finishWorking();
63                  }
64                  visualizer.setGridData(null, null, null, null);
65                  if (cb) cb(err);
66              })
67      }
```

Finalmente, el método "**startQuery**" se encarga de invocar la consulta hacia geoserver e informar al portal cuando se tienen los resultados, de tal forma que al obtenerse los resultados se actualiza el visualizador **KonvaLeaflet** y se informa al manejador global de eventos, por si hay algún otro objeto interesado en los resultados.

El código completo de esta clase es el siguiente:

```
class DataPointsVisualizerHelper extends RasterVisualizer {
        static applyToLayer(layer) {
            return layer.variable.queries.includes("grid");
        }
        constructor(layer, config) {
            super(layer);
            config = config || {};
            this.config = config;
            this.query = new RasterQuery(this.layer.geoServer, this.layer.dataSet,
this.layer.variable, "grid");
            this.aborter = null;
        }
        get code() {return "data-points"}
        get name() {return "Puntos con Datos"}
        async create() {
            this.visualizer = this.layer.konvaLeafletLayer.addVisualizer(this.code, new
DataPointsVisualizer({
```

```
            zIndex:2,
onBeforeUpdate: _ => {this.startQuery(); return false}
        }));
        this.timeChangeListener = _ => {
            if (this.layer.config.dataSet.temporality != "none" && this.active) this.startQuery()
        }
        window.geoos.events.on("portal", "timeChange", this.timeChangeListener);
        this.startQuery();
}

        async destroy() {
            window.geoos.events.remove(this.timeChangeListener);
            if (this.aborter) {
                    this.aborter.abort();
                    this.aborter = null;
            }
            if (this.layer.konvaLeafletLayer)
this.layer.konvaLeafletLayer.removeVisualizer(this.code);
            this.visualizer = null;
        }
        update() {
            if (this.active && this.layer.active && this.layer.group.active) {
                    this.layer.konvaLeafletLayer.getVisualizer(this.code).update();
            }
        }

        startQuery(cb) {
            if (this.aborter) {
                    this.aborter.abort();
                    this.finishWorking();
            }
            this.startWorking();
            let {promise, controller} = this.query.query({margin:1, level:this.layer.level});
            this.aborter = controller;
            let visualizer = this.layer.konvaLeafletLayer.getVisualizer(this.code)
promise
            .then(ret => {
                    this.aborter = null;
                    this.finishWorking();
                    window.geoos.events.trigger("visualizer", "results", this);
                    visualizer.setGridData(ret.foundBox, ret.rows, ret.nrows, ret.ncols);
                    if (cb) cb();
            })
            .catch(err => {
                    this.aborter = null;
```

```
                if (err != "aborted" && err.toString().indexOf("abort") < 0) {
                console.error(err);
                this.finishWorking();
            }
            visualizer.setGridData(null, null, null, null);
            if (cb) cb(err);
        })
        }
        getPropertyPanels() {
            return []
        }
}
RasterVisualizer.registerVisualizerClass("data-points", DataPointsVisualizerHelper);
```

La clase **DataPointsVisualizer** se encarga de la representación de los datos capturados por la clase Helper en el mapa. Se utilizan dos bibliotecas externas para este propósito. Leaflet, que es la base de los mapas de GEOOs y KonvaJS, la que ofrece capacidades gráficas de tipo vectorial y de más alto nivel que las nativas del browser (HTML Canvas).

```
1   class DataPointsVisualizer extends KonvaLeafletVisualizer {
2       constructor(options) {
3           super(options);
4       }
5
6       setGridData(box, rows, nrows, ncols) {
7           this.box = box;
8           this.rows = rows;
9           this.nrows = nrows;
10          this.ncols = ncols;
11          this.update();
12      }
```

Los datos retornados por el GeoServer (y consultados por la clase Helper) son almacenados como atributos del objeto para ser usados durante el pintado. La consulta hacia el GeoServer se hace sobre el rectángulo que define el área visible del mapa en cada momento, sin embargo, dado el carácter discreto de la matriz de datos, geoserver extiende esos límites a los “más cercanos con datos” para cada variable. Esos límites (coordenadas) usados efectivamente en la resolución de la consulta se retornan en el objeto “**box**”.

El objeto “**rows**” contiene para cada elemento (fila que representa una longitud) la lista de valores para las diferentes latitudes. Los atributos **“nrows”** y **“ncols”** contienen la cantidad de filas y columnas en la matriz retornada.

```
13      update() {
14          this.konvaLayer.destroyChildren();
15          if (!this.box) {
16              this.konvaLayer.draw();
17              return;
18          }
19          let lng = this.box.lng0, lat = this.box.lat0;
20          for (let iRow=0; iRow<this.nrows; iRow++) {
21              lng=this.box.lng0;
22              for (let iCol=0; iCol<this.ncols; iCol++) {
23                  let v = this.rows[iRow][iCol];
24                  let point = this.toCanvas({lng, lat})
25                  let circle = new Konva.Circle({
26                      x: point.x,
27                      y: point.y,
28                      radius: 6,
29                      stroke: (v !== null && v !== undefined)?"red":"white",
30                      strokeWidth: 3,
31                  });
32                  this.konvaLayer.add(circle)
33                  lng += this.box.dLng;
34              }
35              lat += this.box.dLat;
36          }
37          this.konvaLayer.draw()
38          super.update();
39      }
```

El método **“update”** es el encargado de la actualización gráfica. Se comienza por eliminar todos los objetos contenidos (children) en la capa **Konva** asociada al visualizador.

A continuación, se recorre la matriz de resultado y se agrega un objeto círculo a la capa konva por cada elemento. El color se selecciona dependiendo de si el elemento recorrido tiene o no datos asociados (es “**null**” si no hay datos).

La clase padre “**KonvaLeafletVisualizer**” incluye los métodos de conversión desde las coordenadas de la proyección del mapa (lat, lng) hacia la capa gráfica (Konva Layer) que se usa para el dibujado de los elementos. De igual forma, así como ofrece el método “**toCanvas**”, hay un equivalente “**toMap**” que realiza el mapeo desde la capa gráfica Konva hacia las coordenadas espaciales (lat,lng).

El código completo de la clase visualizer es:

```
class DataPointsVisualizer extends KonvaLeafletVisualizer {
        constructor(options) {
          super(options);
        }
```

```
        setGridData(box, rows, nrows, ncols) {
            this.box = box;
            this.rows = rows;
            this.nrows = nrows;
            this.ncols = ncols;
            this.update();
        }
        update() {
            this.konvaLayer.destroyChildren();
            if (!this.box) {
                    this.konvaLayer.draw();
                    return;
            }
            let lng = this.box.lng0, lat = this.box.lat0;
            for (let iRow=0; iRow<this.nrows; iRow++) {
                    lng=this.box.lng0;
                    for (let iCol=0; iCol<this.ncols; iCol++) {
                        let v = this.rows[iRow][iCol];
                        let point = this.toCanvas({lng, lat})
                        let circle = new Konva.Circle({
                                x: point.x,
                                y: point.y,
                                radius: 6,
                                stroke: (v !== null && v !== undefined)?"red":"white",
                                strokeWidth: 3,
                        });
                        this.konvaLayer.add(circle)
                        lng += this.box.dLng;
                    }
                    lat += this.box.dLat;
            }
            this.konvaLayer.draw()
            super.update();
        }
}
```