MapStore

MapStore is an highly modular **Open Source** WebGIS framework developed by GeoSolutions to create, manage and securely share maps and mashups. This simple and intuitive framework is able to mix map contents provided by Google Maps, OpenStreetMap, Bing or other servers compliant to OGC standards like WFS, CSW, WMC, WMS, WMTS and TMS. MapStore is used to find, view and query published geospatial data and to integrate multiple remote sources into a single map; the result is an high quality and user friendly framework that allows different kind of use cases by harmonizing remote data with smart and advanced functionalities (like chart widgets, dashboards, timelines and others). MapStore resources are not only related to Maps but also Dashboards and Stories; in MapStore you can create your own innovative and fascinating Application Context where users can save, manage and share its own resources by also managing access permissions to other groups of users.

MapStore is not only a product but also a WebGIS framework. As a standard geoportal product, it is a web-based product that allows to provide a powerful and interactive geospatial WebGIS, it provides a direct and real-time access to geospatial data warehouses and it supports the most common standards

formats available for geospatial data. MapStore also provides advanced spatial analysis capabilities that can be used to build WebGIS solutions through a powerful, dynamic and open geospatial application. Since MapStore is also a framework, you can use it to build your own WebGIS applications by using its plugins and modules.

Last but not the least, MapStore is map agnostic and ensures the greatest flexibility: its abstraction tier allows to work with different web mapping libraries. The mapping engines currently supported by MapStore are OpenLayers (used by default for desktops), LeafletJS (used by default for mobile devices) and Cesium 3D viewer.

MapStore has been designed from the beginning to provide a coherent and comprehensive experience across different devices types.

MapStore is based on OpenLayers, Leaflet and ReactJS, and is licensed under the Simplified BSD license.

Supported Browsers

The browsers supported by MapStore are *Google Chrome*, *Microsoft Edge*, *Mozilla Firefox* and *Safari*. Ensure to have the latest version installed.

Quick Start

You can either choose to download a standalone binary package or a WAR file to quickly start playing with MapStore. See the Quick Start documentation for more details.

Documentation

- Users Guide
- Developers Guide

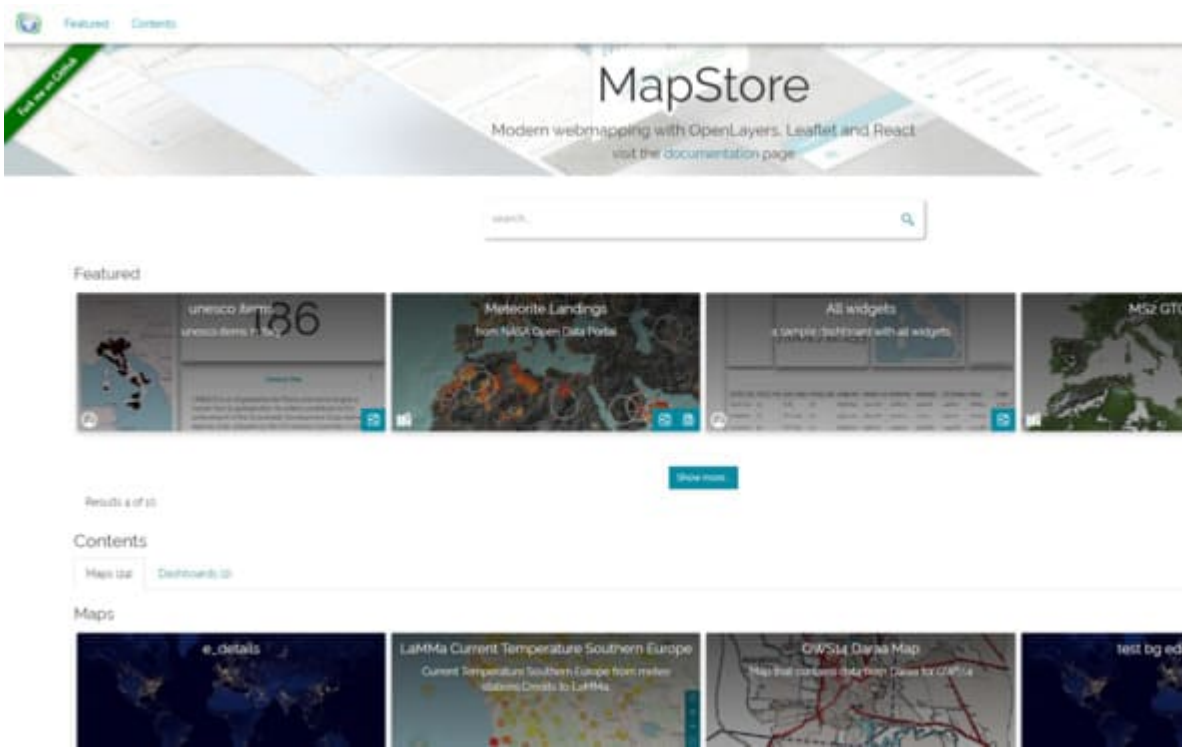
Home Page

In order to get started, let's take a look at the portal interface and get an idea of how to navigate around it. First of all it's necessary to specify that the user can take advantage of different tools and sections according to his authentication in MapStore. In particular, a user can access the MapStore application by:

- Anonymous user
- Normal user
- Administrator user

Anonymous user

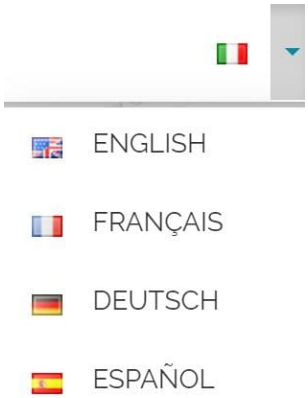
Accessing MapStore as anonymous user, the Homepage shows up as in the figure below:



The anonymous user is allowed to:

- Access GeoSolutions website with a click on the  icon
- Navigate through the *Featured* and *Contents* sections

- Set the application language, with the *Language switcher*:

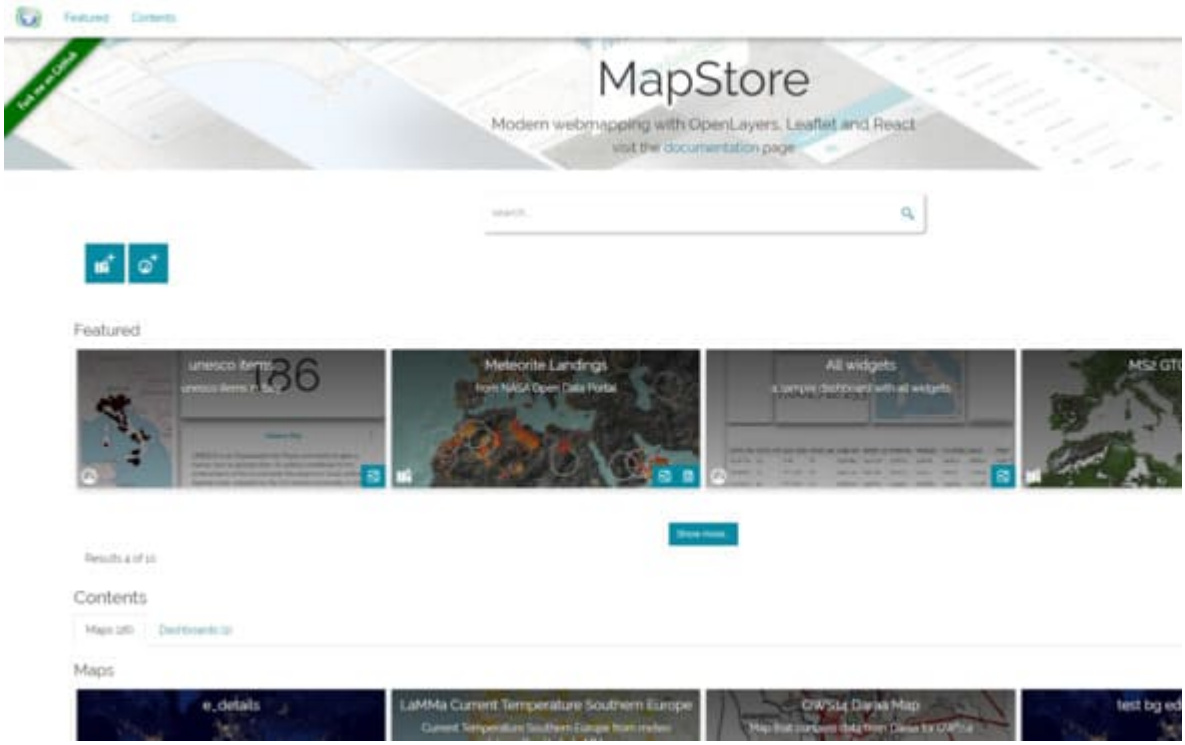


- Login  (more information about Login can be found in Managing Users and Groups section)
- Perform a search for resources, through the *Search bar*:

- Share a resource 
- Take a look at map Details when available 
- Open resources and navigate inside them according to their Permissions

Normal user

With a login as normal user, the Homepage displays as below:



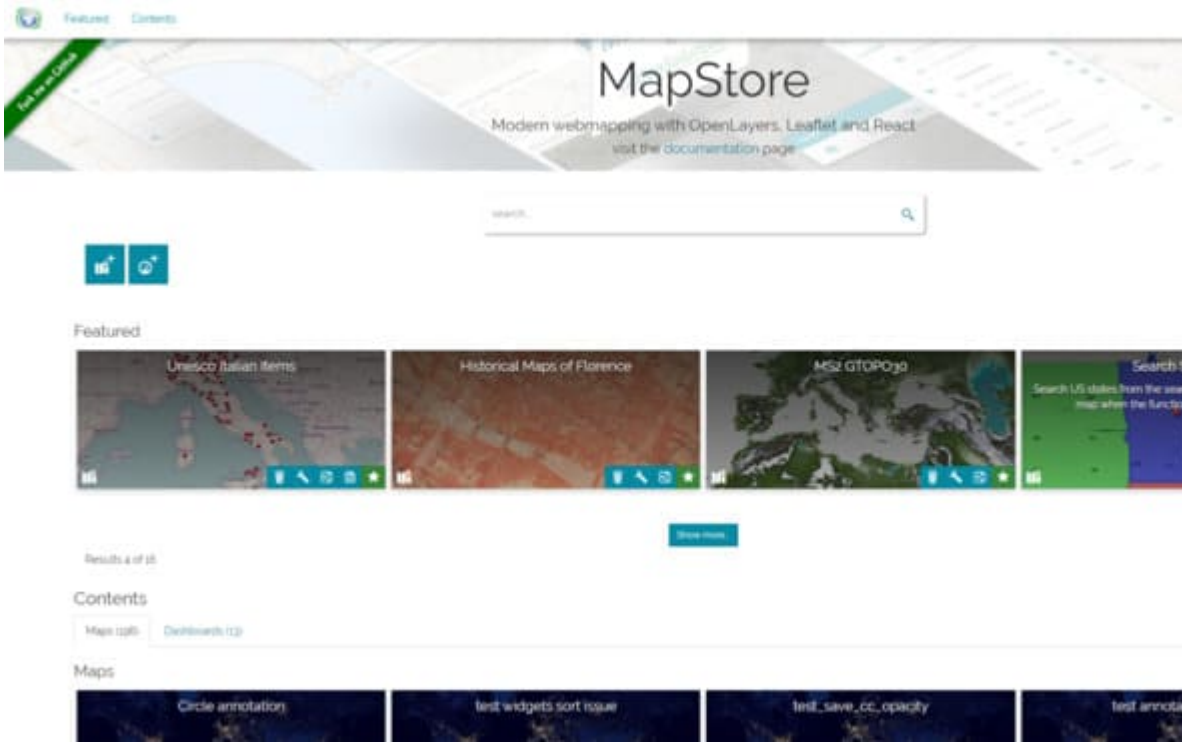
The normal user, in addition to what the anonymous user can do, is allowed to:

- Create new resources:

- View, edit and remove resources according to their Permissions 

Administrator user

Once logged in as *Administrator*, the Homepage it's like the following:



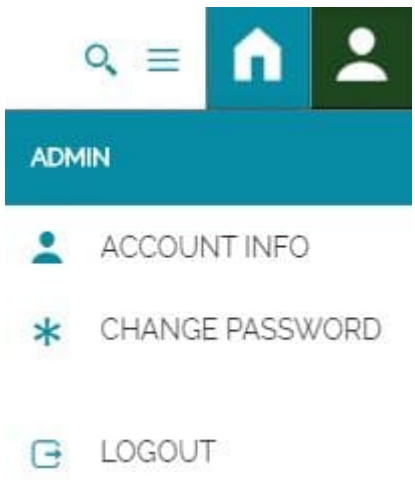
The admin can see and edit everything. In particular, in addition to what normal user can do, an administrator can also:

- Access the **Manager** button for Manage Accounts and Manage Contexts
- Manage the resources by including or excluding them from *Featured* section
- View, edit and remove any resource

Managing Users and Groups

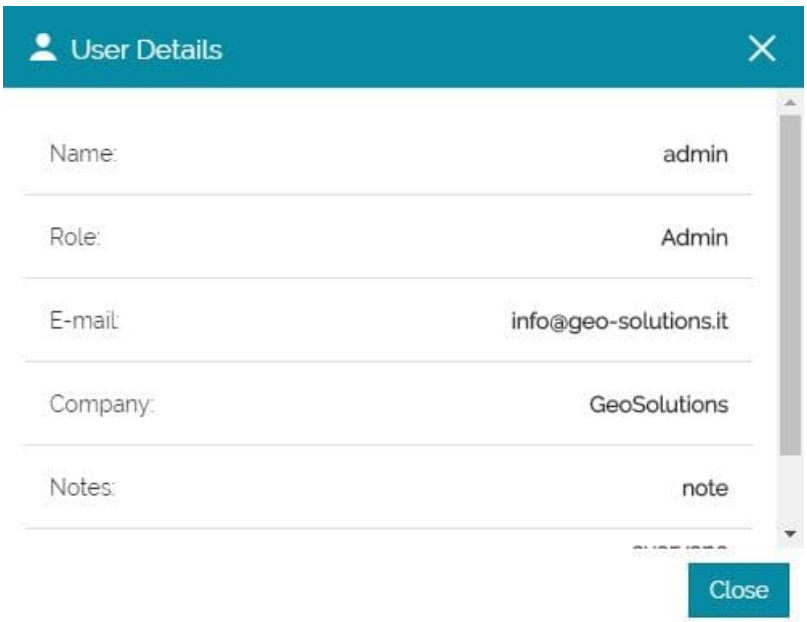
Accessing MapStore as anonymous user the **Login** button in Homepage is blue  . With a click on it, the following window appears:

Once the login is made, the same button displays in green  and a click on it opens a list of options:

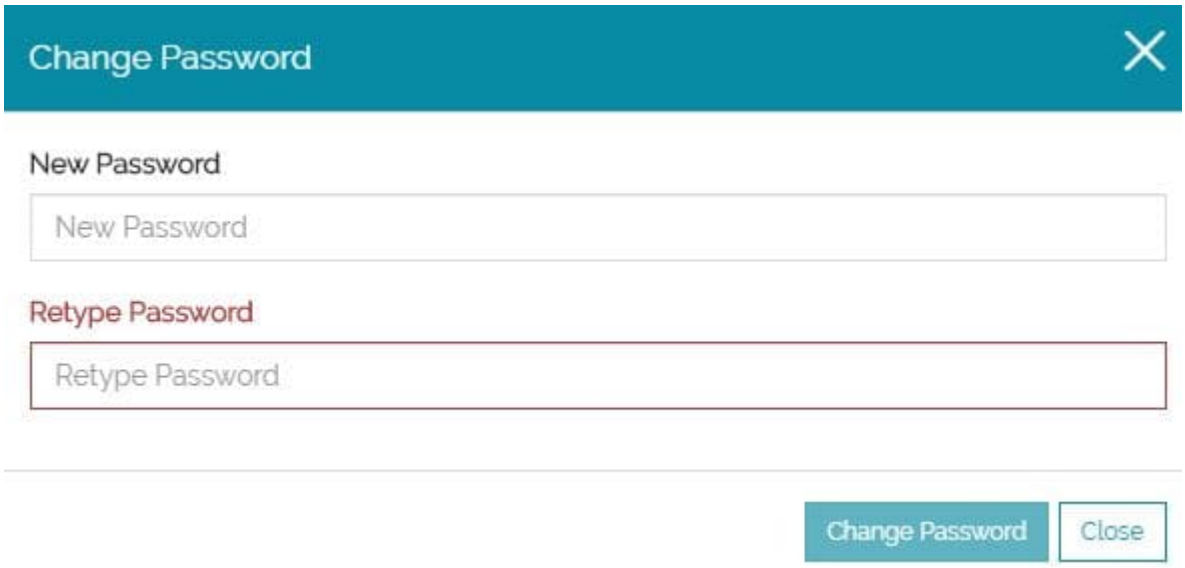


Through these options it is possible to:

- Get the following **Account info**: *Name*, *Role*, *E-mail*, *Company*, *Notes* and *Groups* (in order to understand how these info are set see the Managing Users section)

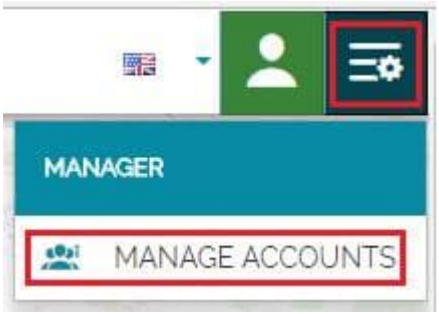


- **Change Password**



- **Logout**

Once logged as Admin, become possible to manage users and groups and the **Manage Accounts** option appears in Homepage:



Selecting **Manage Accounts** options, the *Account Manager* opens:

In this page it is possible to switch between Manage Users or Manage Groups sections:

Managing Users

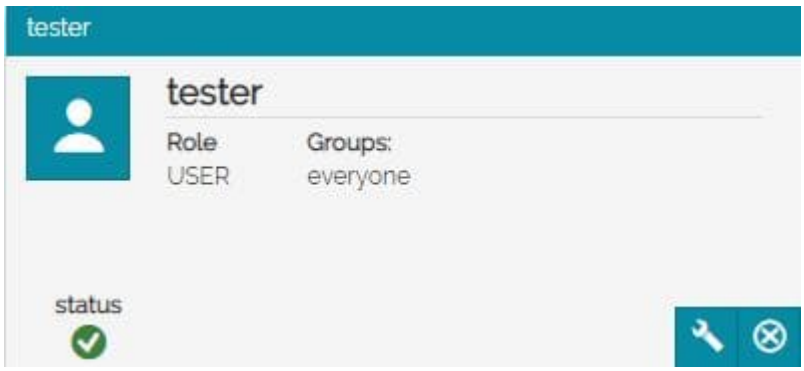
Switching to *Users Manager*, the page displayed is the following:

In this page the Admin can:

- Perform a search among the existing users

- Create a new user with the **New User** button 

- Edit or remove an existing one, through the **Edit user**  and **Delete user**  buttons on each user card:



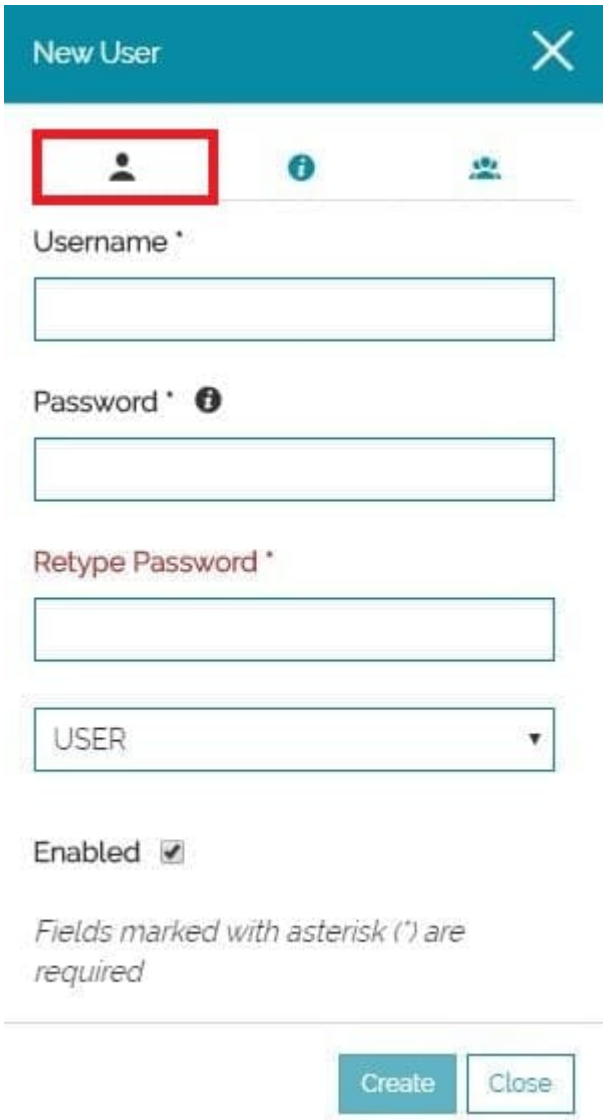
Both the **New User** 

and the **Edit user**  buttons, open the *User editor* window that is composed of three sections:

- *User ID*
- *Other information*
- *Group membership*

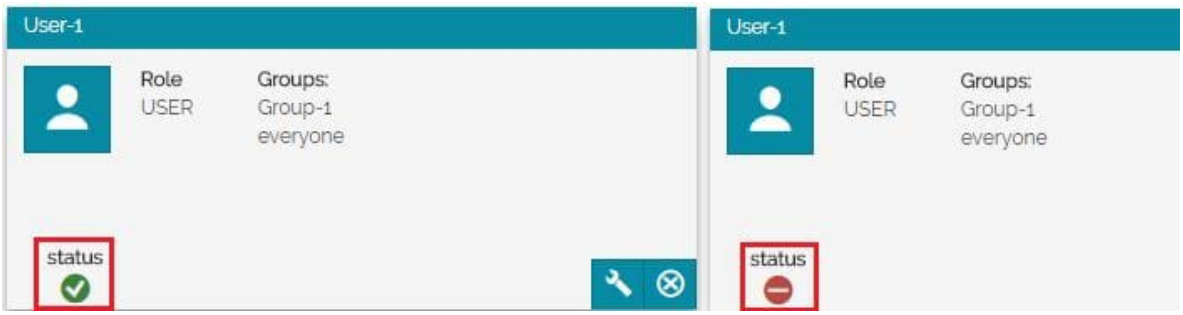
User ID

As soon as the *New User* window opens, the *User ID* section is displayed:



In this section the Admin is allowed to:

- Set the *Username*
- Set the *Password*
- Select the *User role* (Normal user or Admin)
- Choose if an user is *Enabled* or not. Enabled users will have a green status icon under their profile, otherwise disabled users will have a red status and will not be able to log in.



Warning

Username and *Password* are the only mandatory fields. The password must contain at least 6 characters.

Other information

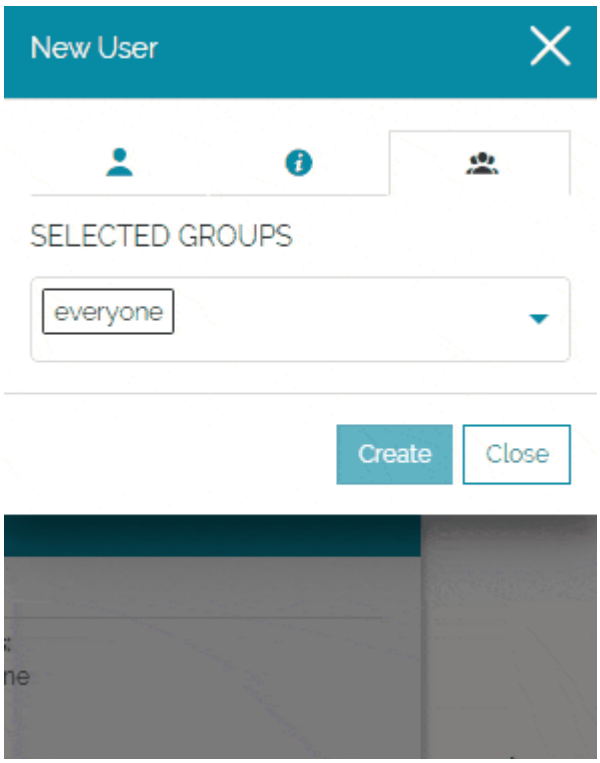
Switching to *Other information* section, it display the following:

Here the Admin can set:

- *Email*
- *Company*
- *Notes*

Group membership

Through the last section of the window it is possible to manage the groups in which the user belongs to:



Note

The *everyone* group, set by default, it is impossible to remove since it must be attributed to all users.

Managing Groups

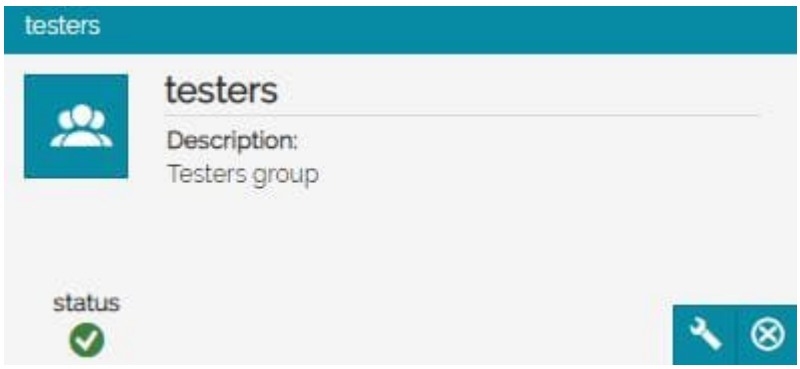
The *Groups Manager* section displays like the following:

Similar to what happens for the Users Manager, in this page the Admin can:

- Perform a search among the existing groups

- Create a new group with the **New Group** button 

- Edit or remove an existing one, through the **Edit group**  and **Delete group**  buttons on each group card:



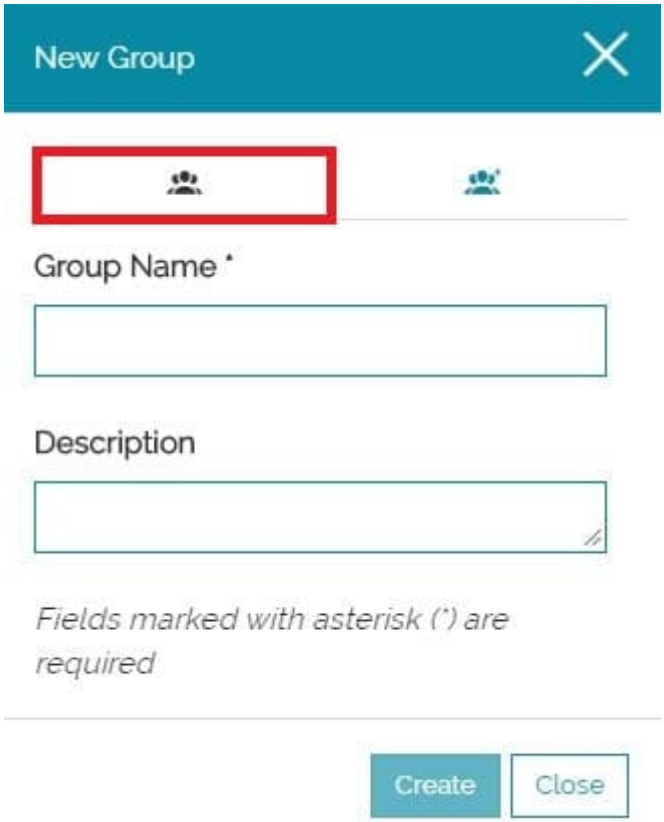
Both the **New Group** 

and the **Edit group**  buttons, open the *Group editor* window that is composed of two sections:

- *Group ID*
- *Members manager*

Group ID

As soon as the *New Group* window opens, the *Group ID* section is displayed:



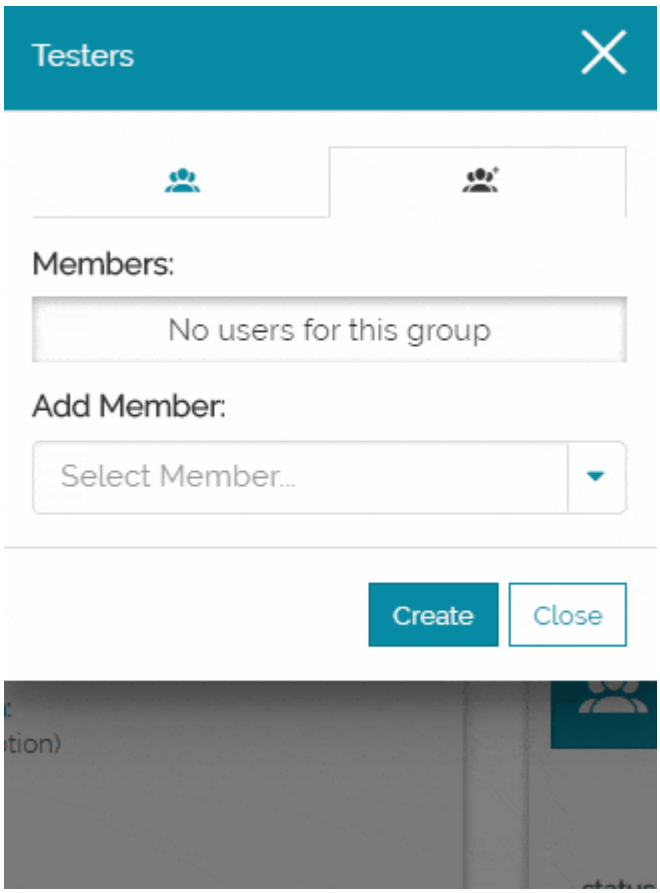
In this section the Admin is allowed to:

- Set the *Group Name*
- Set the group *Description*

> **Warning**
>
> The *Group Name* is the only mandatory field.

Members manager

Through the *Members manager* section it is possible to choose which users are part of the group:

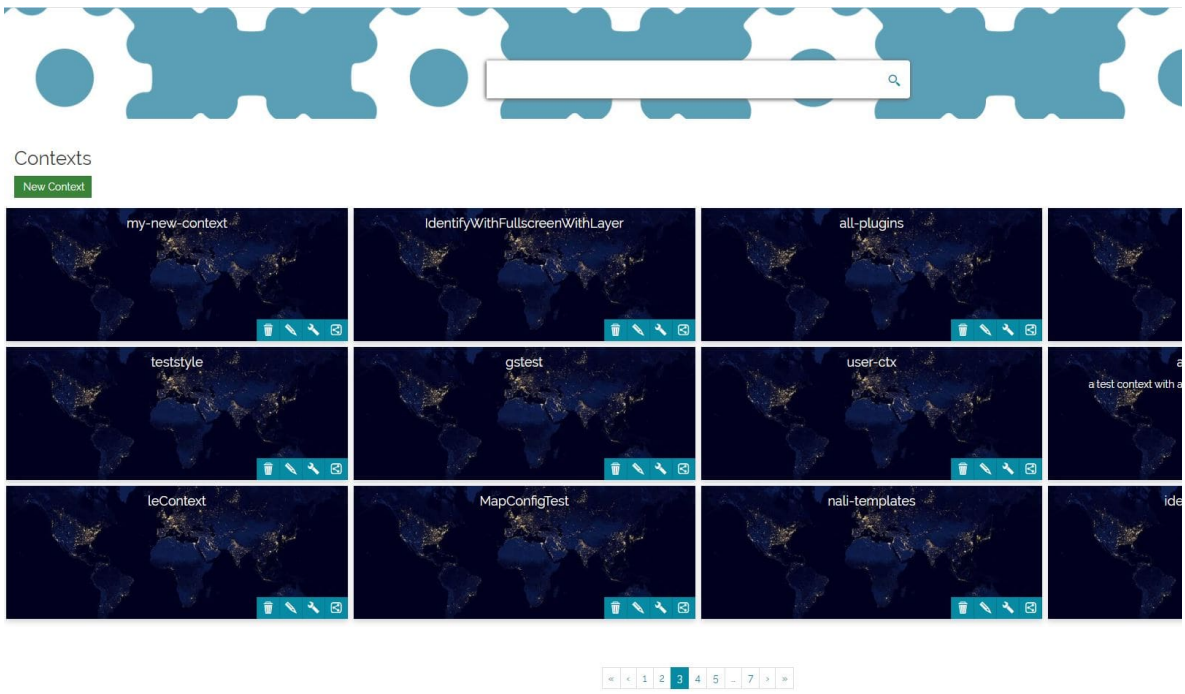


Managing Contexts

In MapStore the **Application Context Manager** is an administrative tool designed to build and configure MapStore's viewers: the administrator is able to configure a custom MapStore viewer by choosing:

- The name of the context (the viewer will have its own specific URL)
- The default map configuration and map contents (like layers, backgrounds, catalogs, CRSs etc)
- The set of plugins available for the viewer

The *Admin* can access the Application Context Manager through the MANAGE CONTEXTS button available in the **Manager** option menu in Homepage.



In this page the administrator can:

- Perform a search among the existing contexts

- Create an Application context through the New Context button

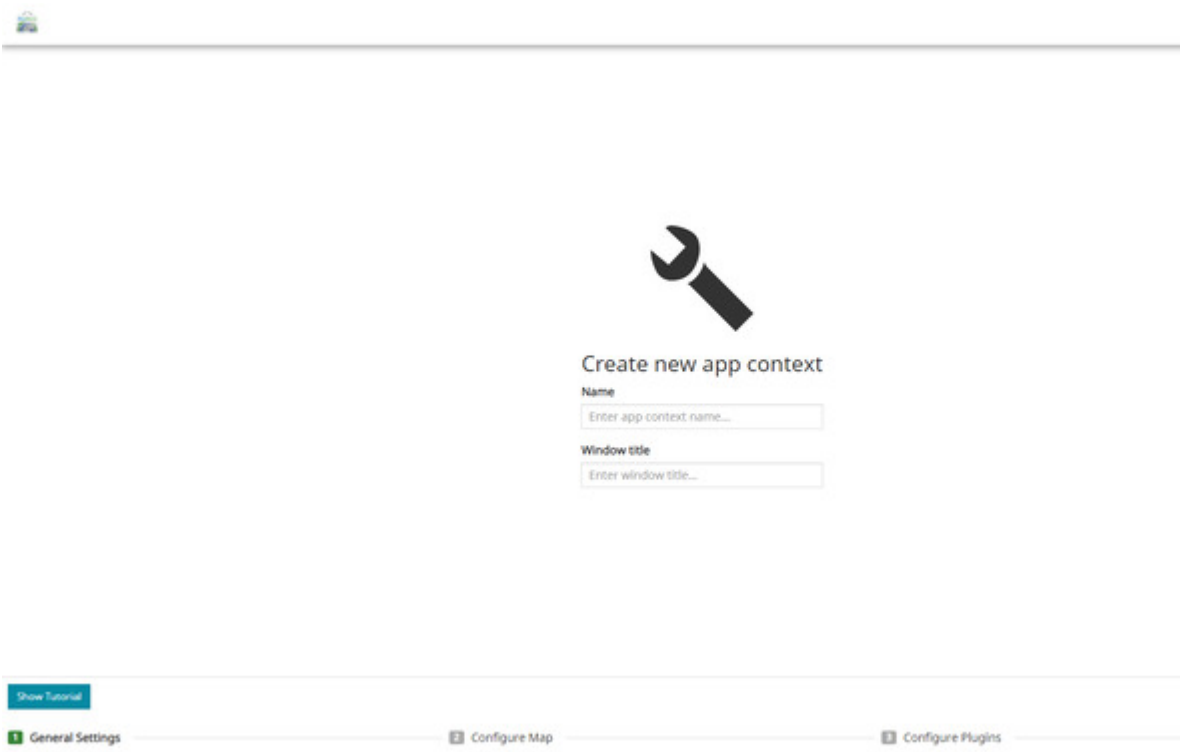
In each context's card the administrator can:

- Remove the context through the **Delete** button
- Edit the context through the **Edit context** button
- Open the Edit properties through the **Edit properties** button
- Sharing the context through the **Share** button

Application Context

In order to create a context, the *Admin* can click on the **New Context** button New Context in the Contexts page and he will be addressed directly to a wizard. The wizard is composed by the following four steps:

You can move through the steps of the wizard with the dedicated buttons located at the bottom right of the page.

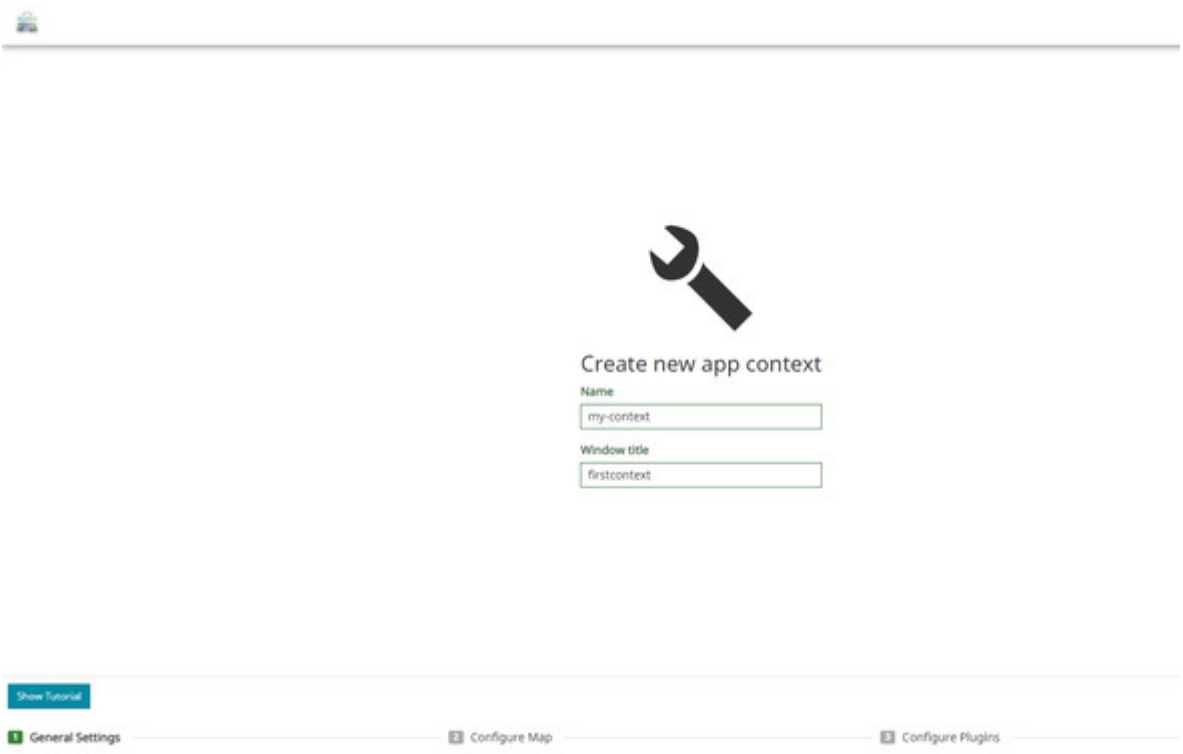


In this way the admin can:

- Move forward on the different steps through the **Next** button Next
- Go back to the previous step through the **Back** button Back
- Closed the context wizard through the **Close** button Close

General Settings

This first step allows to configure the **Name** and the **Window title** of the new context.

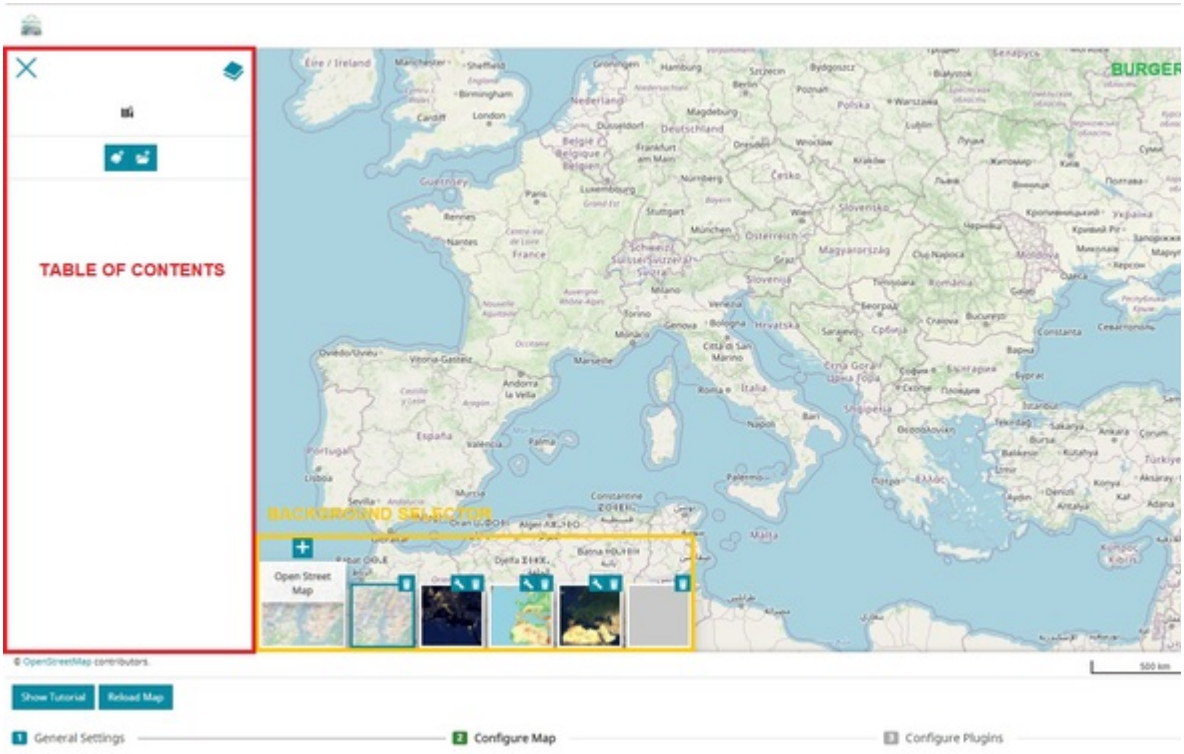


> **Warning**
>
> The name and the window title are both mandatory fields. Note that it is not allowed to choose a name that has already been assigned to another MapStore's resource (like maps, dashboards, stories): a warning message appears in this case to notify the user.

> **Note**
>
> The **Window title** is the name of the browser window.

Configure Map

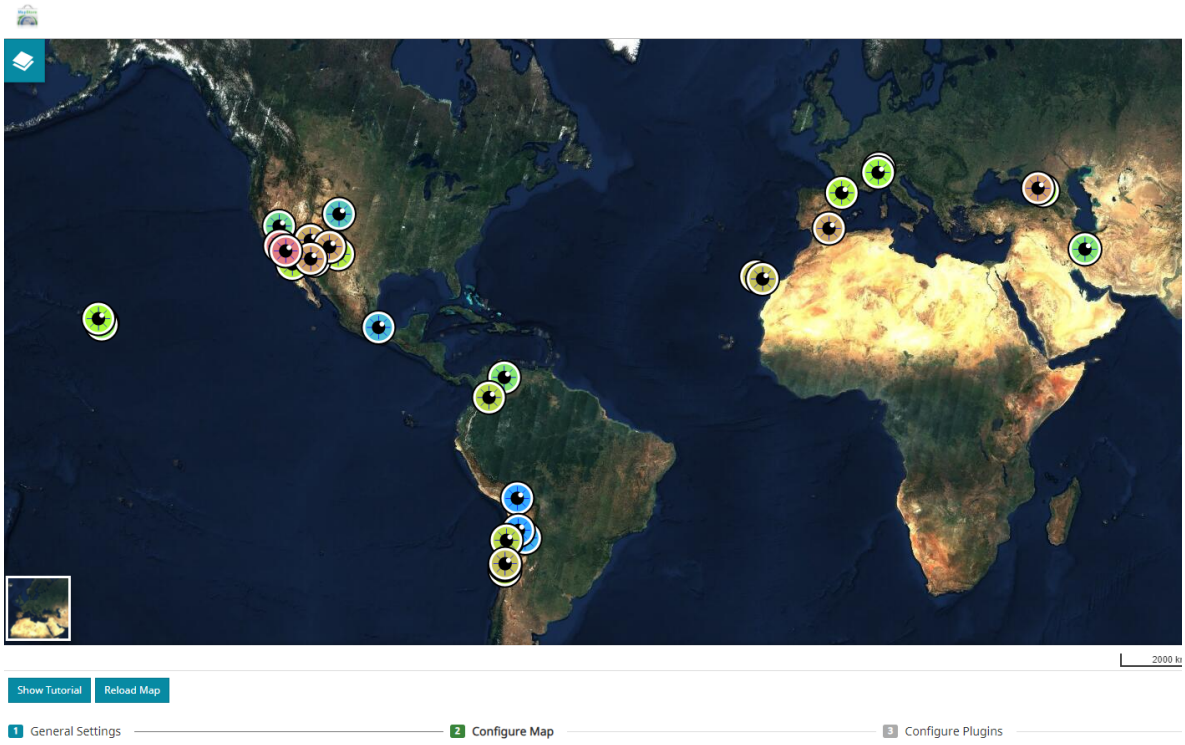
To create the context viewer, the map configuration (like the one described here opens so that the admin can set the initial state of the context map.



In particular the admin can configure the context map using the following MapStore tools:

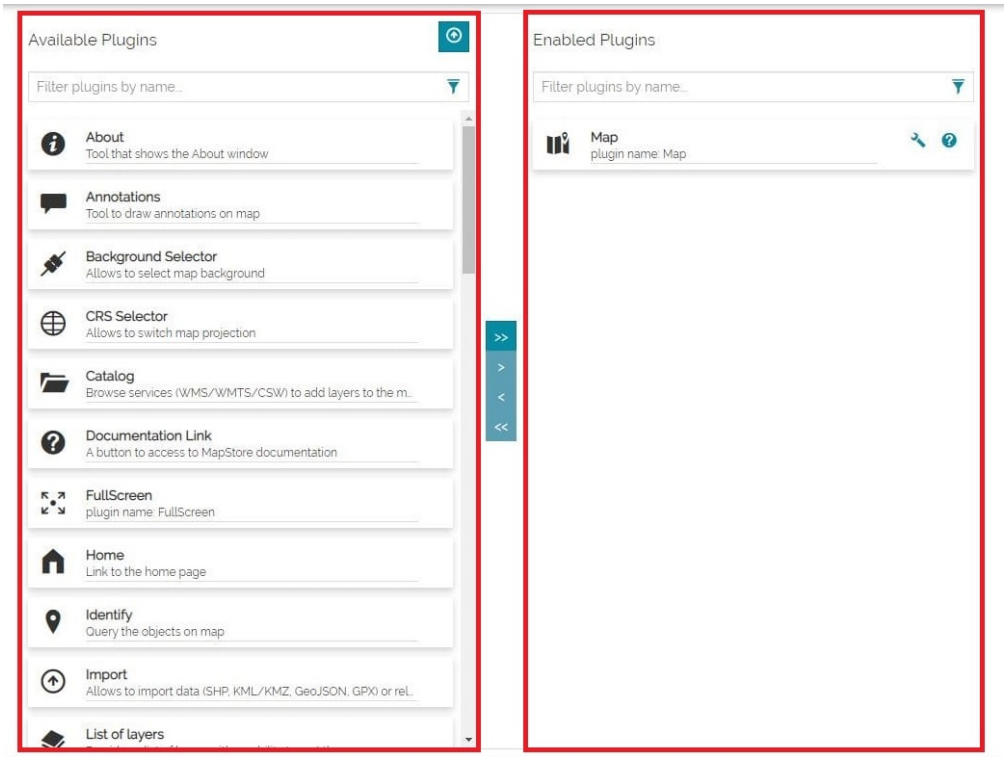
- Catalog, present in *Burger Menu* , to configure the supported remote services (like CSW, TMS, WMS and WMTS) and add layers to the map.
- Import, present in *Burger Menu* , to import map files and import vector file.
- Annotations, present in *Burger Menu* button, to add annotations to the map.
- Table of Contents, through the button where the admin can use all the available functionalities to manage context layers.
- Background Selector, at the bottom left of the viewer, allows the user to add, manage and remove map backgrounds
- CRS Selector, through the button at the bottom right of the *Footer*, to switch the Coordinate Reference System of the map
- The Side Bar, at the bottom right of the viewer, is useful to the admin to explore the map.

An example of a context viewer with a new background and a layer, added to the map, can be the following:

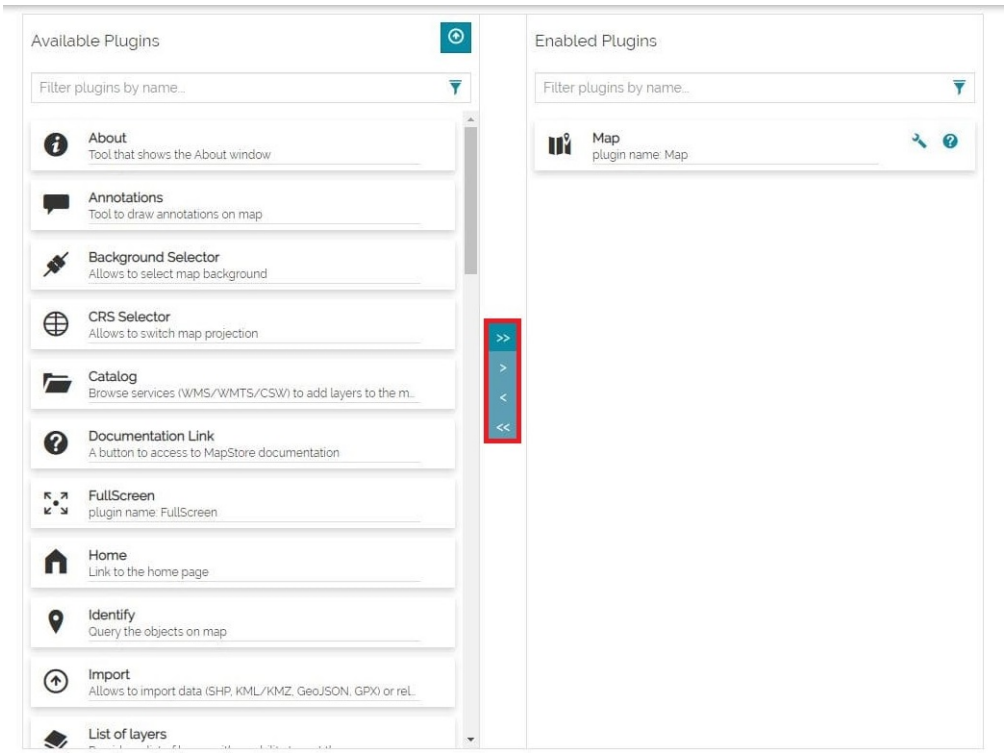


Configure Plugins

This wizard step allows to select the extensions that will be available in the context viewer: the user of a context will use only the plugins enabled by the administrator. Within this wizard step, all the available plugins in MapStore are present in the left side list ready to be selected for the context . The right side list contains the list of plugins selected by the administrator for the context.

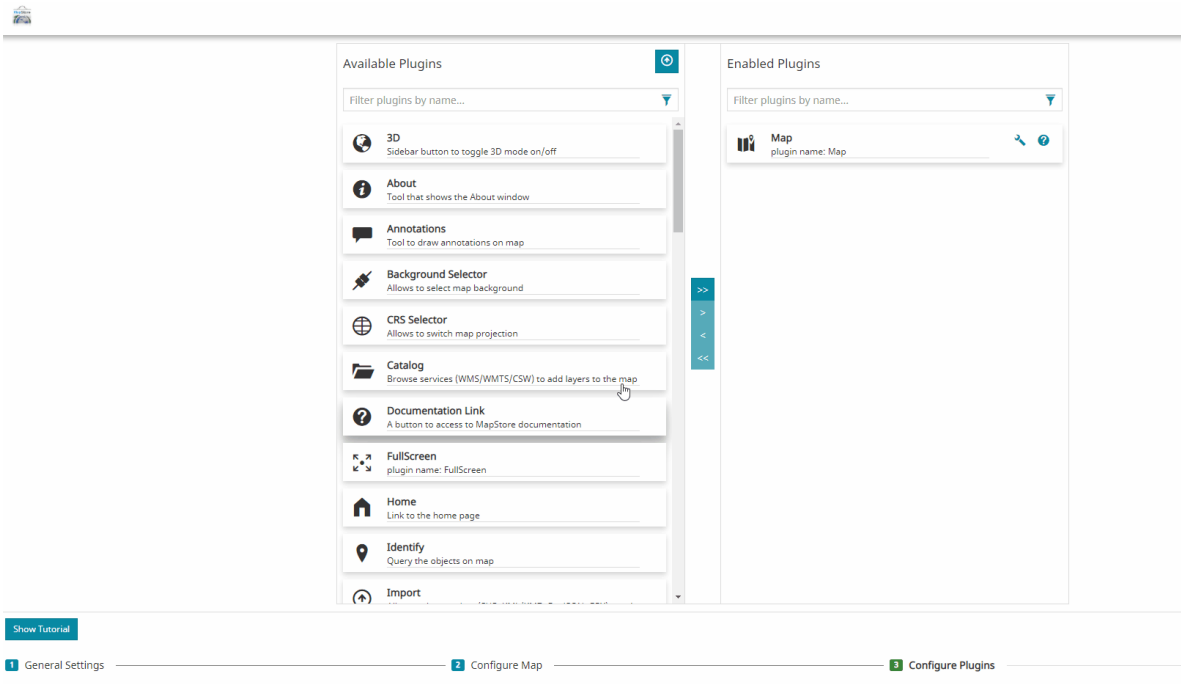


Through the central vertical bar the administrator can select the plugins to include in the context viewer by moving them from the **Available Plugins** list to the **Enabled Plugins** list.

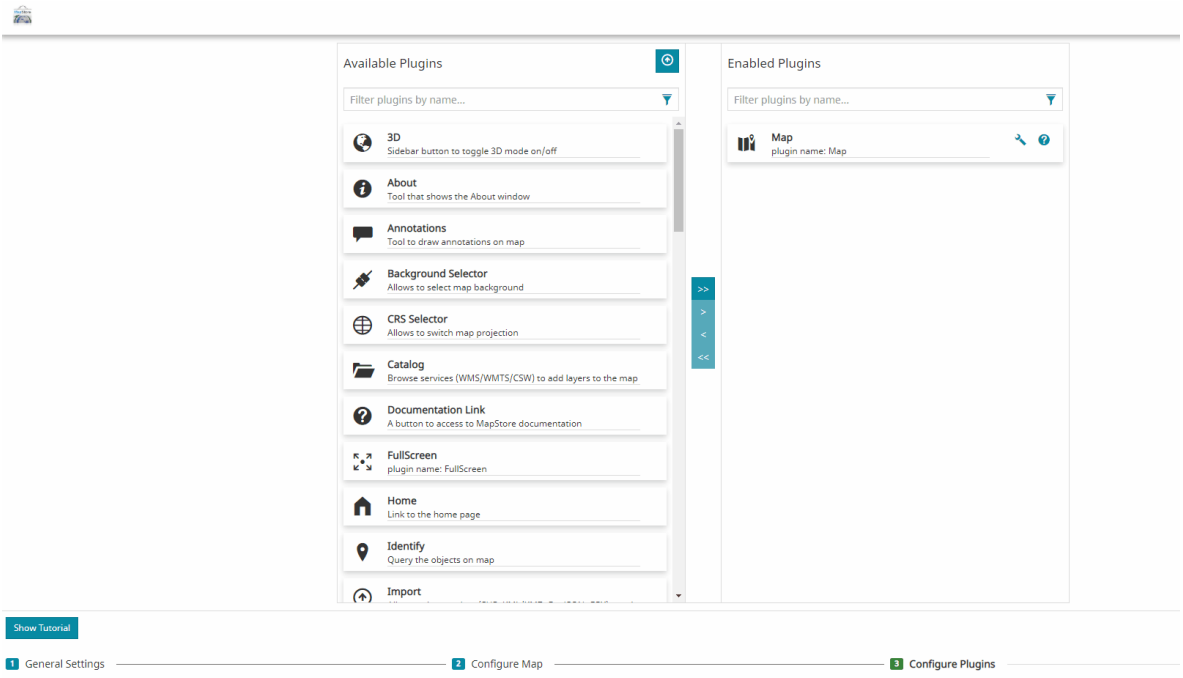


In particular, the admin can:

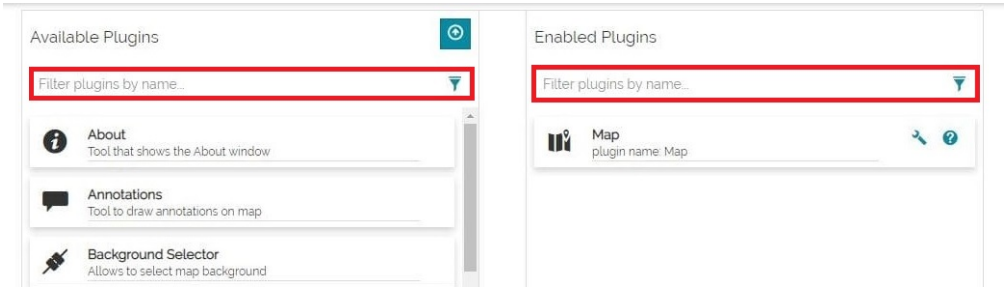
- Add an extension from the *Available Plugins* list to the *Enabled Plugins* list, using the **Add Extension** button `>`. Instead, remove an extension from the *Enabled Plugins* list using the **Remove Extension** button `<`, as follows:



- Bring all extensions from one list to another using the **Add all extensions** button `>>` or remove all extensions using the **Remove all extensions** button `<<`, as follows:



To search for an extension listed, the admin can use the **Search bar**.



Add extensions to MapStore

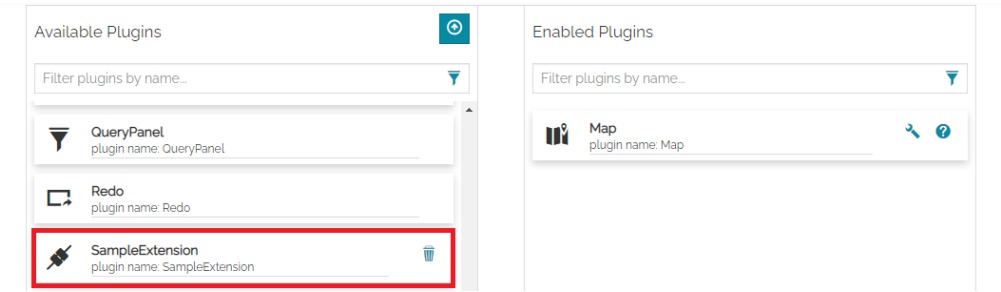
The MapStore administrator can also install a custom plugin by using the **Add extension to MapStore** button  , at the top right of the *Available Plugins* list.

Here the admin, in order to upload the plugin's package, can drag and drop it inside the import screen or select it from the folders of the local machine through the 

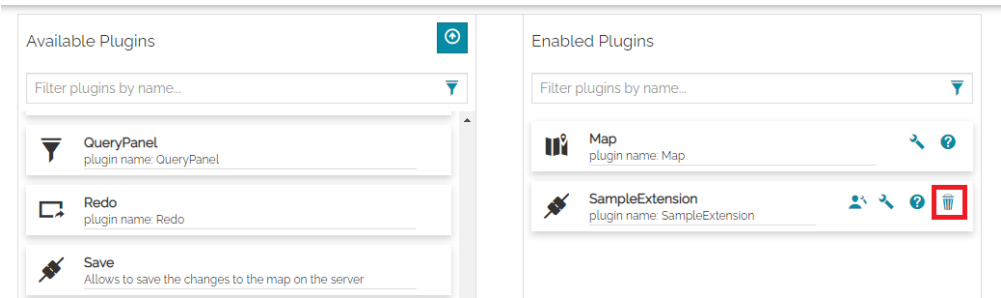
button.

> **Warning**
>
> A plugins package must be provided as .zip archives that contains:
>
> - An `index.json` file with a plugin definition
> - A plugin file with the extension code in `JavaScript`
> - All mandatory translations files in MapStore.
>
> A sample extension for testing purposes is available here. More extensions will be available in the future versions of MapStore.

Through the **Add** button Add the plugin is inserted in the *Available Plugins* list.

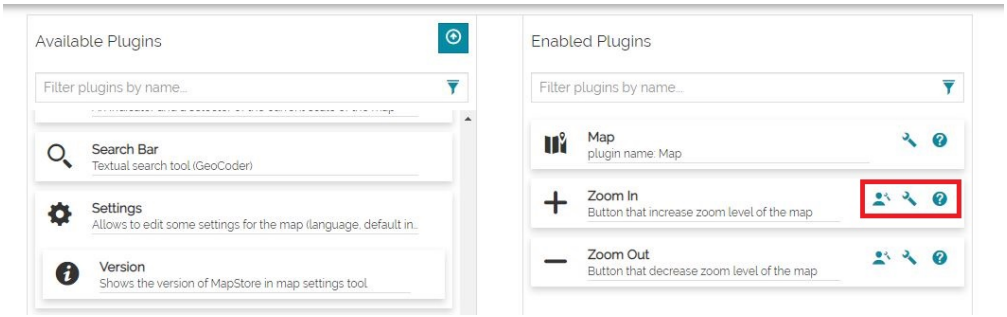


A plugin so installed can be included in the context viewer by moving it in the *Enabled Plugins* list or uninstalled through the **Delete** button .



Optional tools for enabled plugins

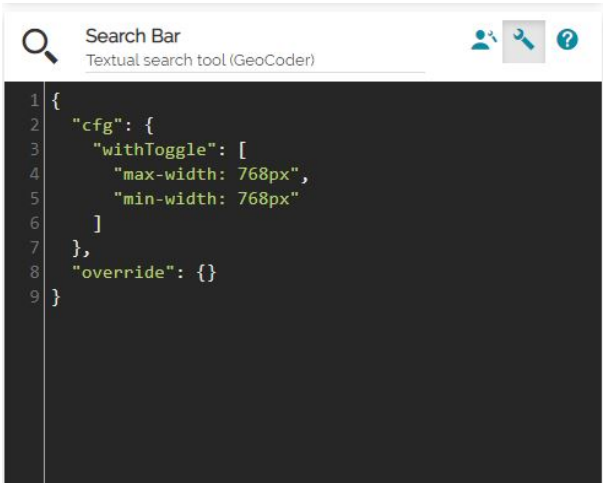
In the *Enabled Plugins* list, the following buttons are displayed for each extension:



- The **Enable selection of current plugin for user** button allows the admin to configure which extensions will be present in the Extension Library and not activated by default.

> **Note**
>
> Once a plugin has been included in a context, it is active by default and available inside the viewer. The administrator can click on **Enable loading this plugin on startup** button to make that plugin not active by default: clicking on this button the plugin will not be available in the context viewer until explicitly activated by the end user through the *Extension Library*.

- The **Edit Plugin Configuration** button allows the admin to interact with a text area to specify the plugin configuration and to override the default one.



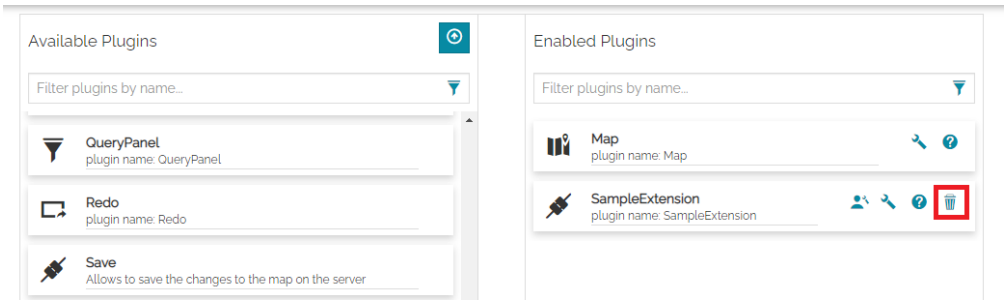
- The **Open plugin configuration documentation** button opens the Plugins Documentation in another page.

How to update extensions

Extension can be updated using two steps:

- Old extension removal.
- Uploading and installation of the new version of extension.

As previously stated, extension can be removed on "Configure Plugins" step of wizard using **Delete** button .



At this point extension will be removed from application completely. Save context after extension removal only if you want to be sure that extension will not be activated for the context if it's reinstalled at some point.

Do not save context and upload new version of extension right away after old version removal. Context don't need to be saved after new version installation.

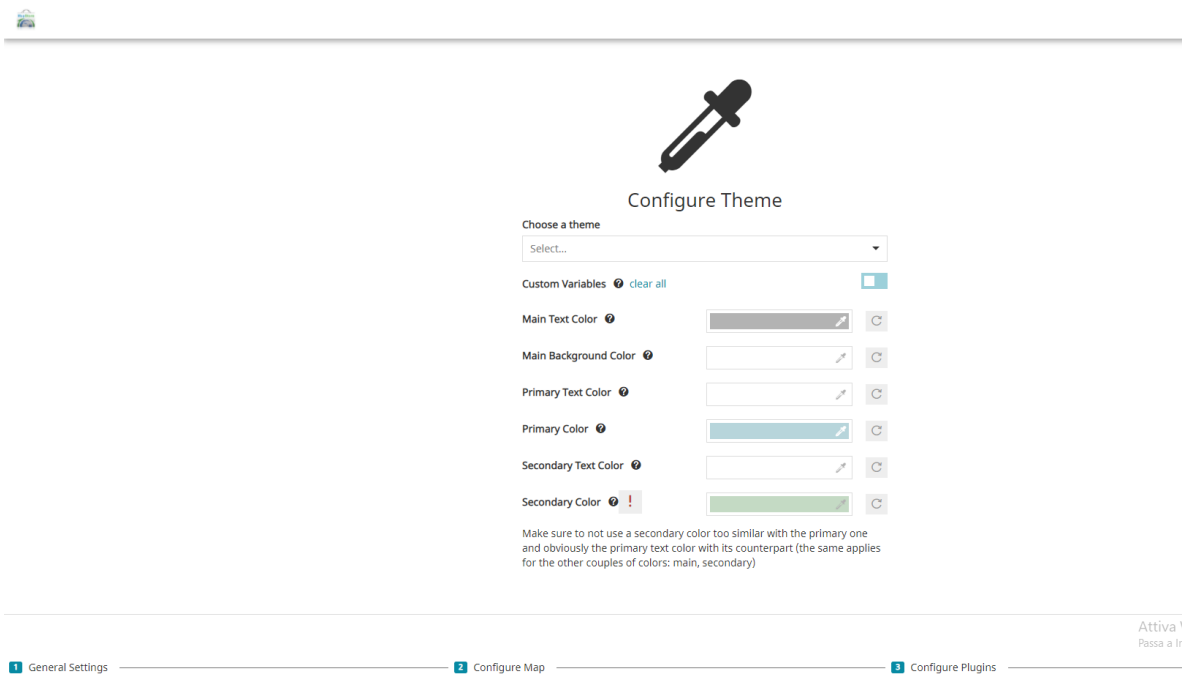
With all stated above, complete workflow is:

- Open context editing and jump to the "Configure Plugins" step of the wizard.
- Delete old version of extension using **Delete** button .
- Upload and install new version of extension using the **Add extension to MapStore** button
- Do not save context, close wizard.

Existing configuration of extension (default or customized) will be preserved for all the contexts using extension.

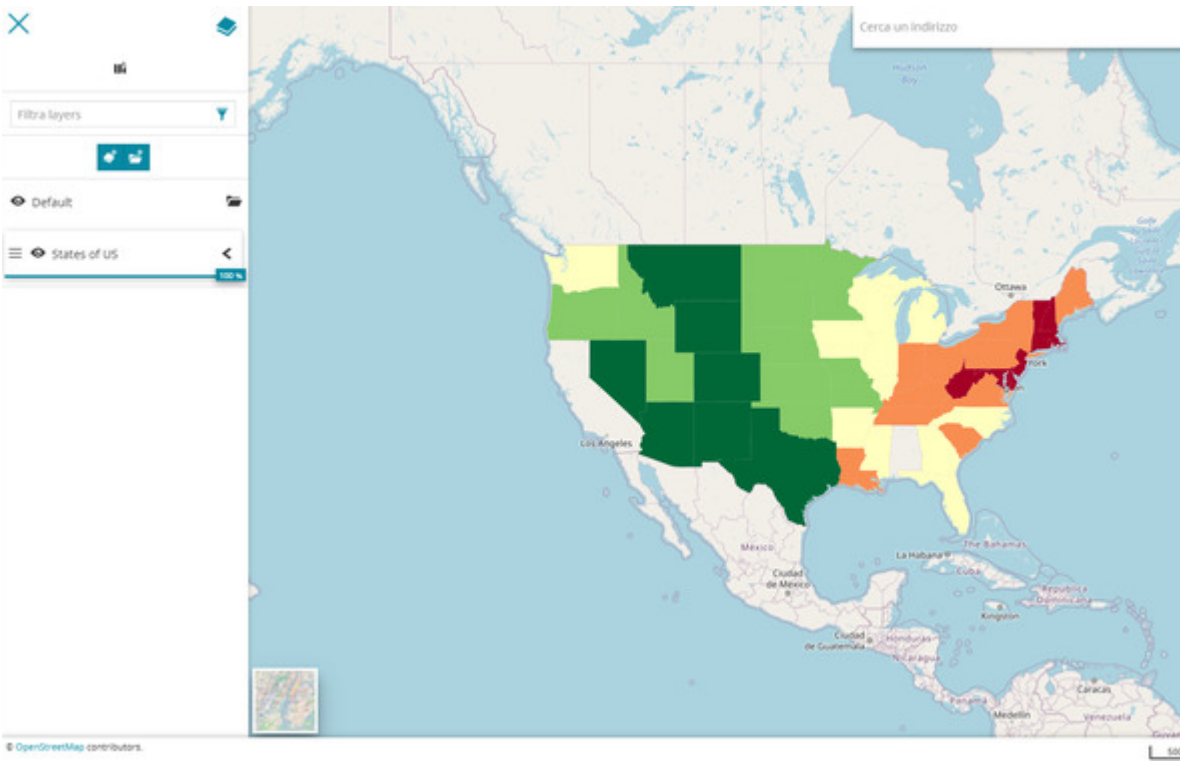
Configure Theme

The last wizard steps allows to configure the theme to use for a context. A dropdown allows to select one of the available themes (see the Styling and Theming section of the online documentation to know how to create and include additional themes to MapStore). By default in MapStore a **default** and a **dark** themes are available.



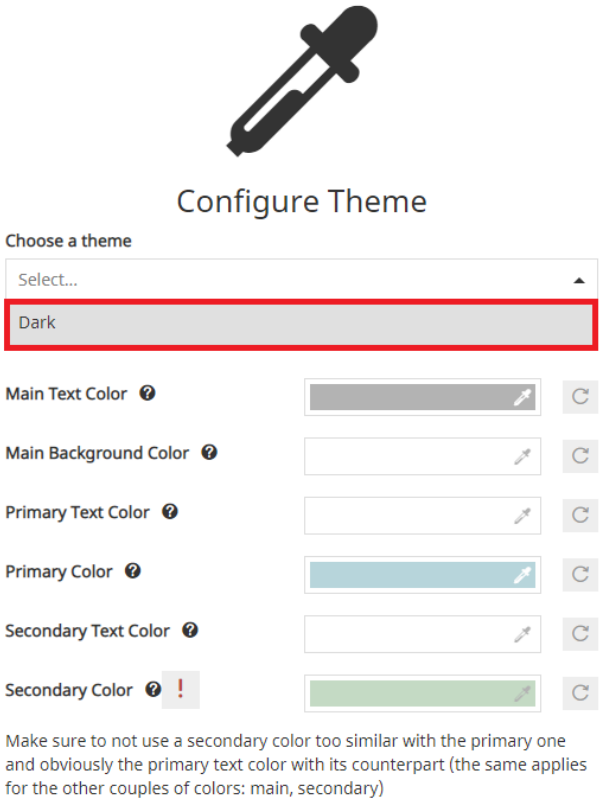
Default Theme

The **default** theme is always available for a context and it is the MapStore default one. This theme is automatically applied to the context if the *Configure Theme* wizard step is skipped during the context creation or when the theme selection drop-down is cleared. An example of a default context can be the following:

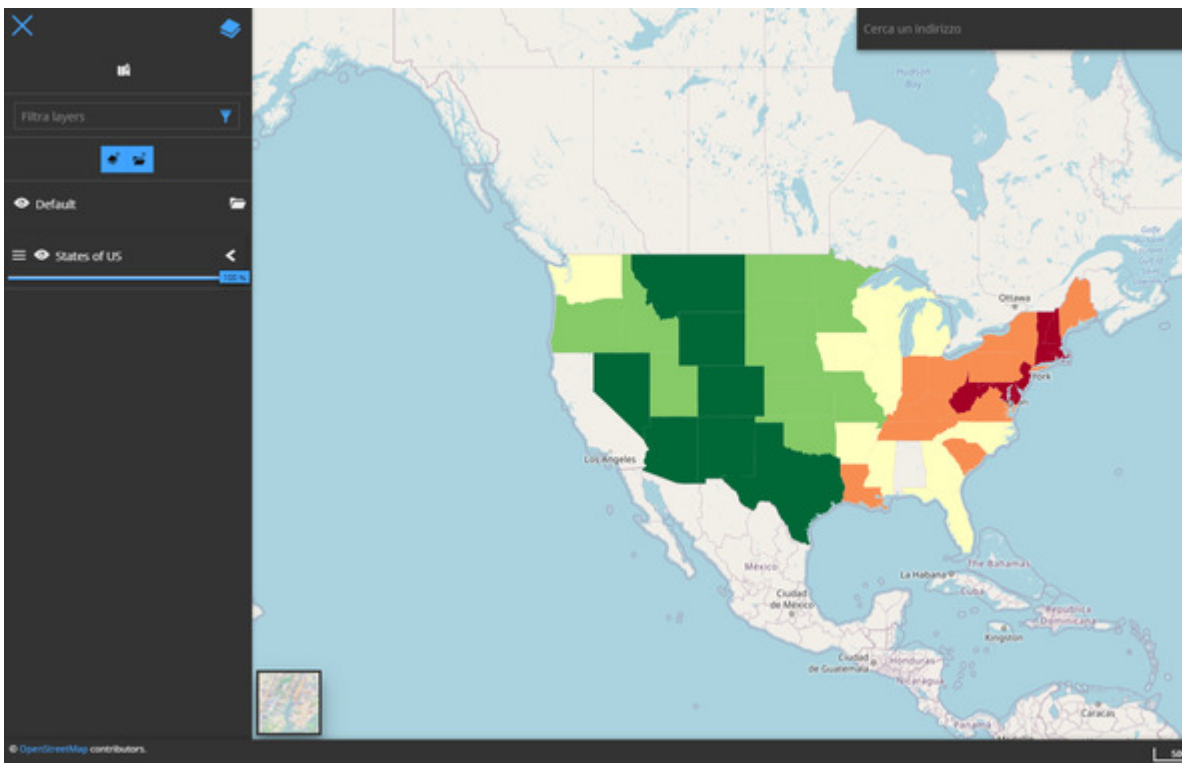


Dark Theme

MapStore also provides by default an additional theme, the dark one, that can be selected from the drop-down menu to be used as an alternative theme for application contexts.

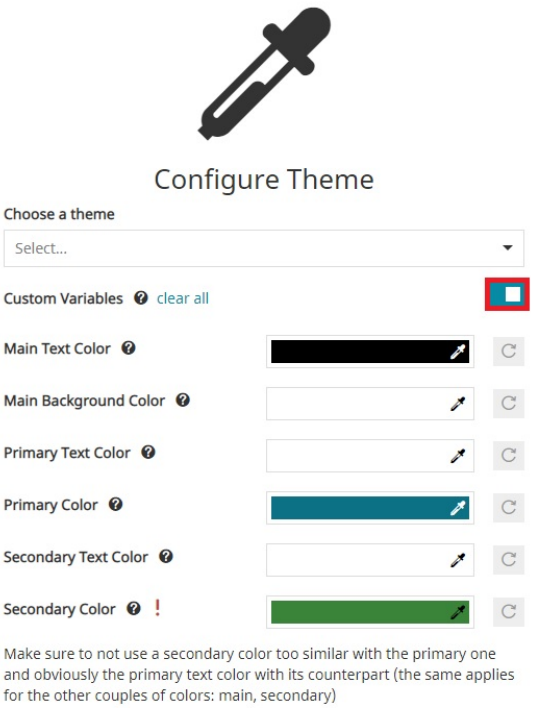


An example of the **dark** theme applied to a context is the following one:

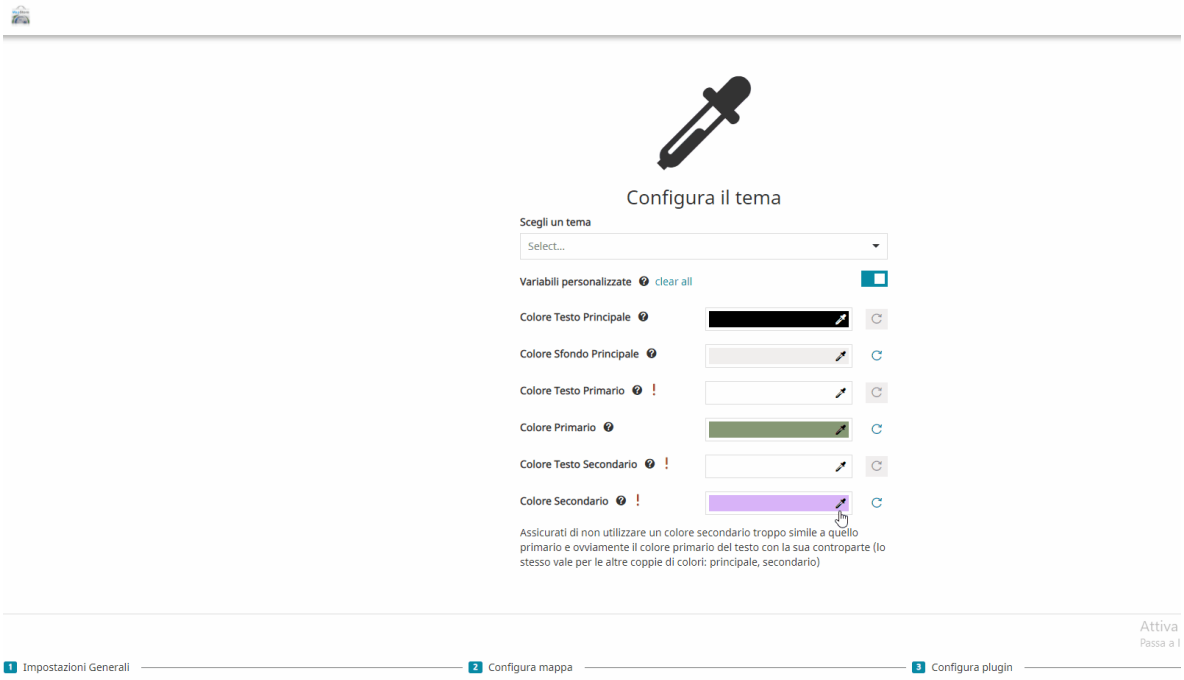


Custom Theme

After selecting a theme from the drop-down, it is also possible to customize it from UI by enabling **Custom Variables**.



Once **Custom Variables** is enabled, the context editor can modify main, primary and secondary colors for both backgrounds and texts (an helper clarifies the UI elements involved for each field in the form). Clicking on the *Change Color* button a color picker is displayed to allow the selection of the desire color, as follows:

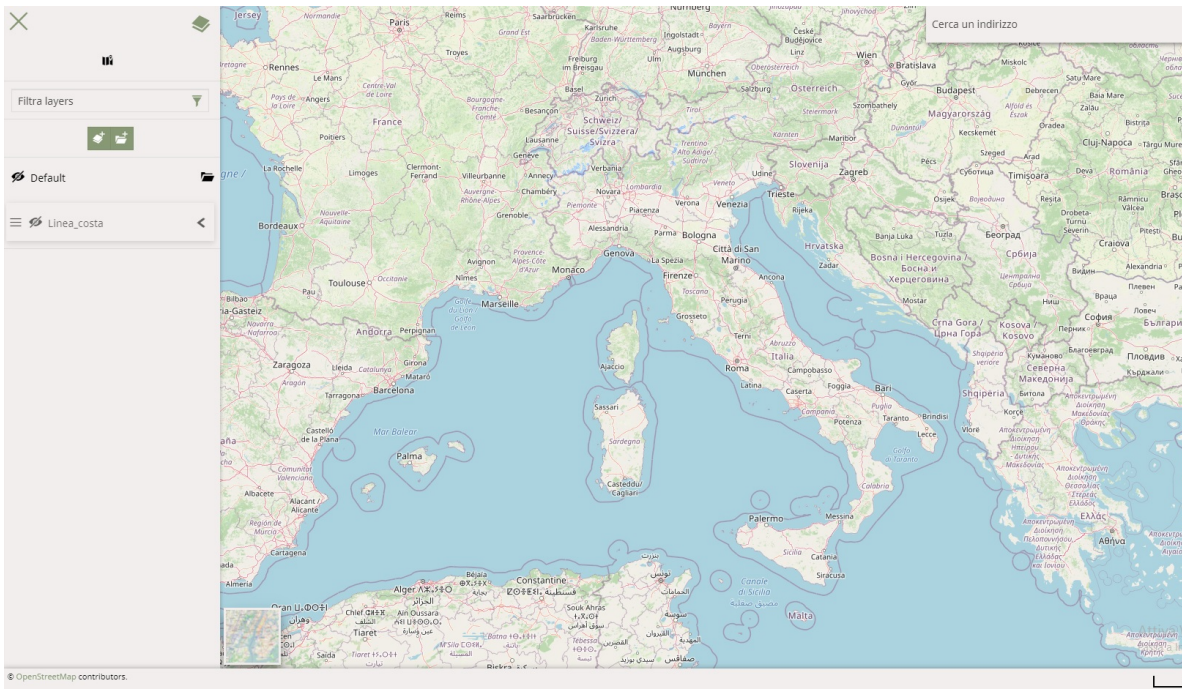


The colors that can be customized are the following ones:

- **Main Text Color** to choose the color used in panel or dialog texts
- **Main Background Color** to choose the color used in panel or dialog backgrounds
- **Primary Text Color** to choose the color used for icons inside toolbar, header and button texts
- **Primary Color** to choose the color used for icons inside toolbar, header and button backgrounds
- **Secondary Text Color** to choose the color used as button text when a button is active or selected
- **Secondary Color** to choose the color used as button background when a button is active or selected

> **Warning**
>
> To ensure a good and well readable color contrast between each UI component, make sure to not use a secondary color too similar to the primary one and obviously the primary text color with its counterpart (the same applies for the other couples of colors: main, secondary).

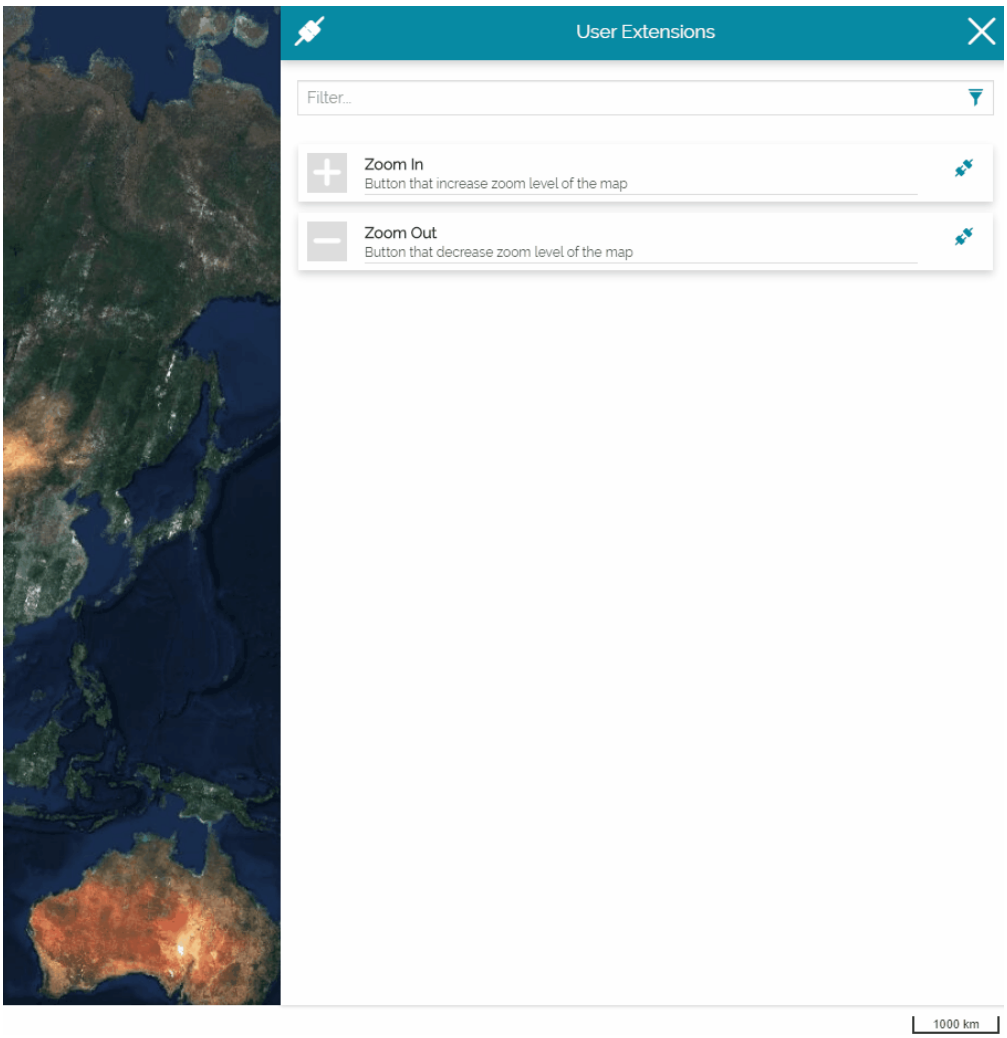
An example of a custom context can be the following:


Filtra layers
Default
Linea_costa
Cerca un indirizzo
© OpenStreetMap contributors.

Extension Library

The USER EXTENSIONS button, present in *Burger Menu* , provide to the user the list of extensions ready to be activated for the viewer: that list of available extensions for the user has been defined by the the administrator during the Application Context creation.

The *User Extensions* panel opens allows the user to choose which extension to add to the viewer through the **Add Extension** button , as follows:

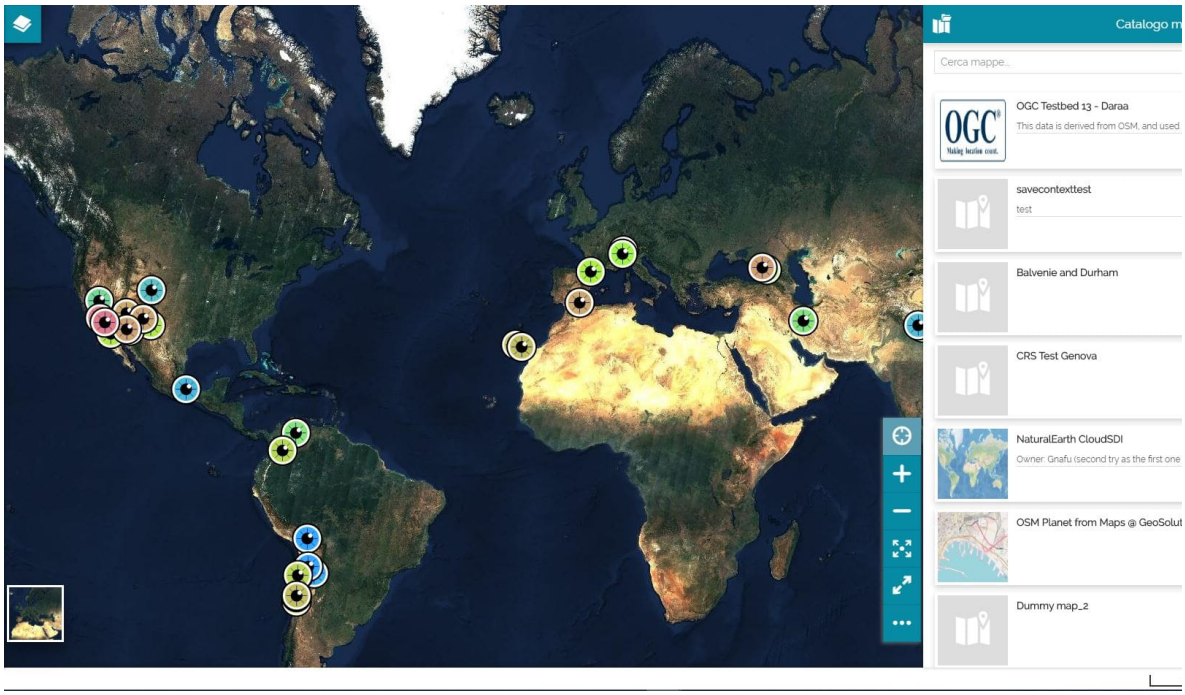


Note

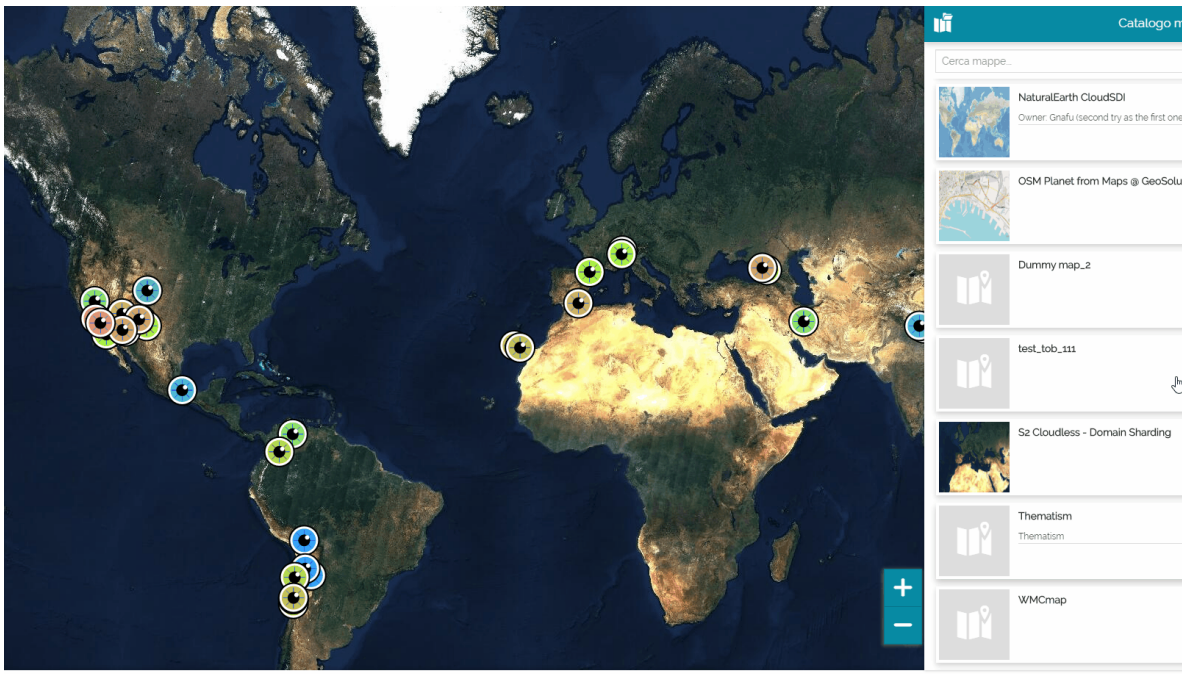
The *User Extensions* is enabled by the admin in the Application Context wizard.

Map Catalog

The **Map Catalog** is an extension that can be included in the step #3 of the application context wizard to allow the end user to browse the existing MapStore maps directly inside the viewer itself. The MAP CATALOG button, present in *Burger Menu*, provides to the user the list of the MapStore maps that are also available in Homepage.



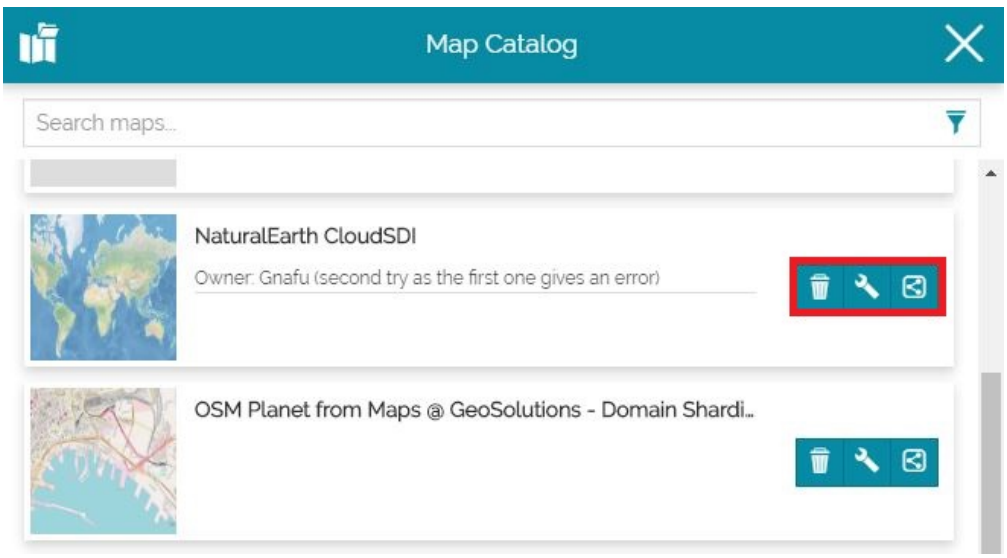
The *Map Catalog* panel allows the user to select a map and loaded it in the same browser page, as follows:



Note

Selecting a map card in the Map Catalog list, the map currently in use will be replaced. In addition, if the selected map has been created in a context, also the viewer will be replaced with the one of the related map context.

For each map in the *Map Catalog* list the following buttons are displayed:

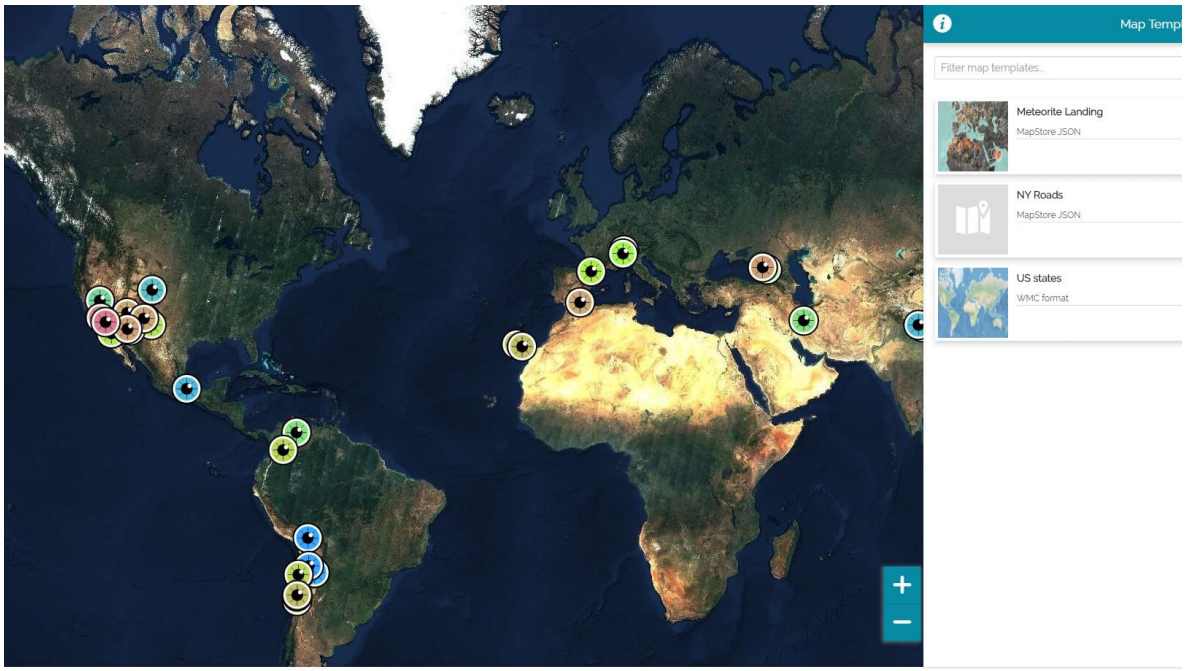


- The **Delete** button allows the user to remove the map
- The **Edit Properties** button allows the user to Edit Properties of the map

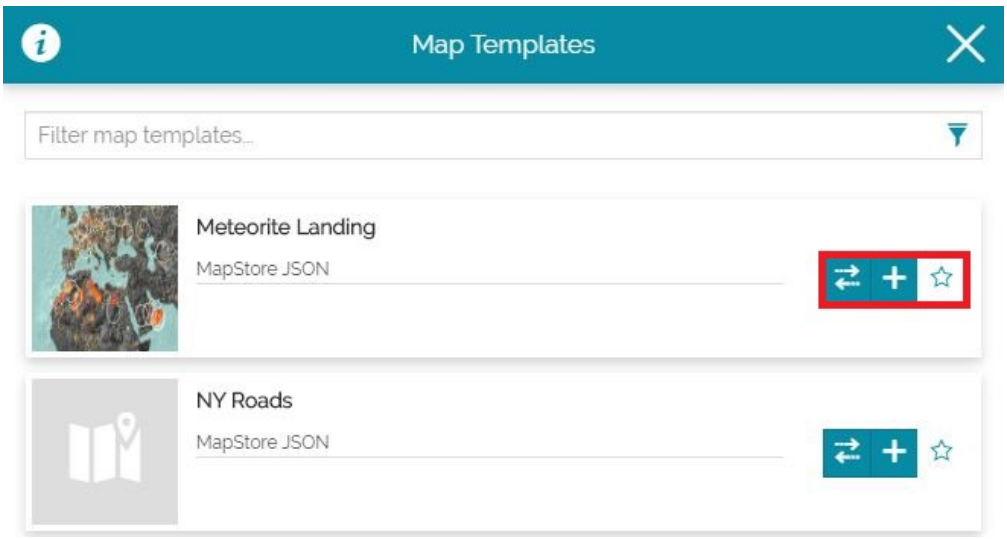
- The **Share** button  allows the user to Share the map

Map Templates

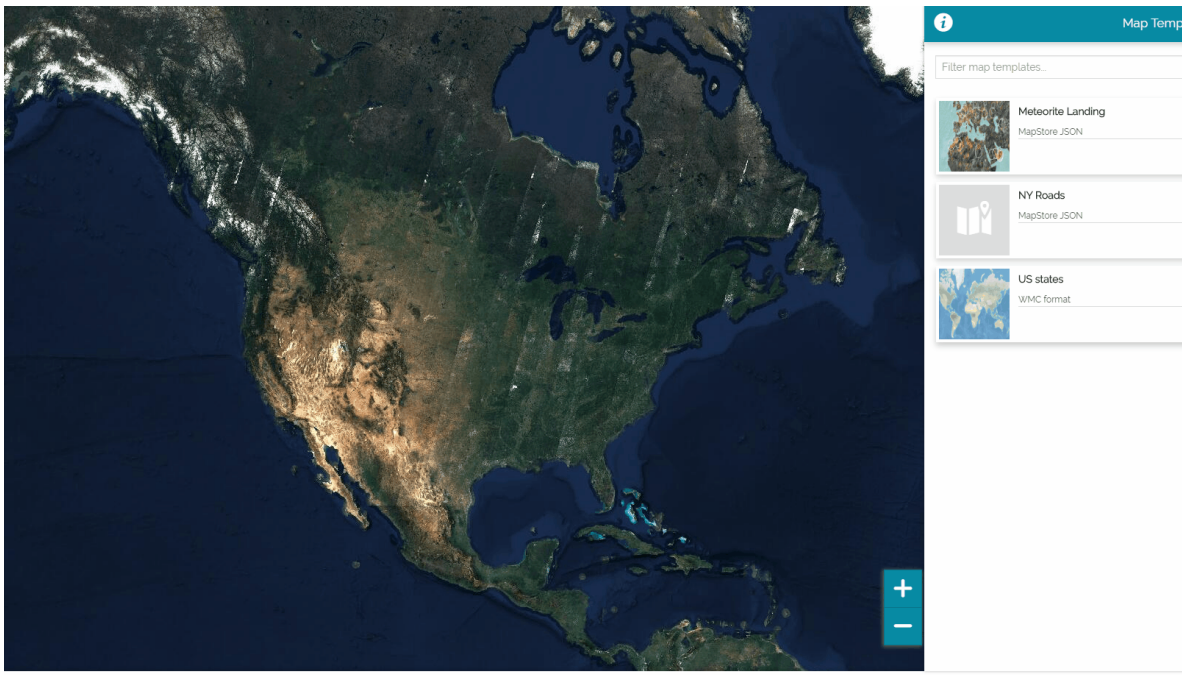
This extension allows to browse **Map Templates** in a MapStore's viewer. Supported Map Templates formats in MapStore are WMC and MapStore's native JSON. The MAP TEMPLATES button, present in *Burger Menu*, provides to the user the list of the available templates.



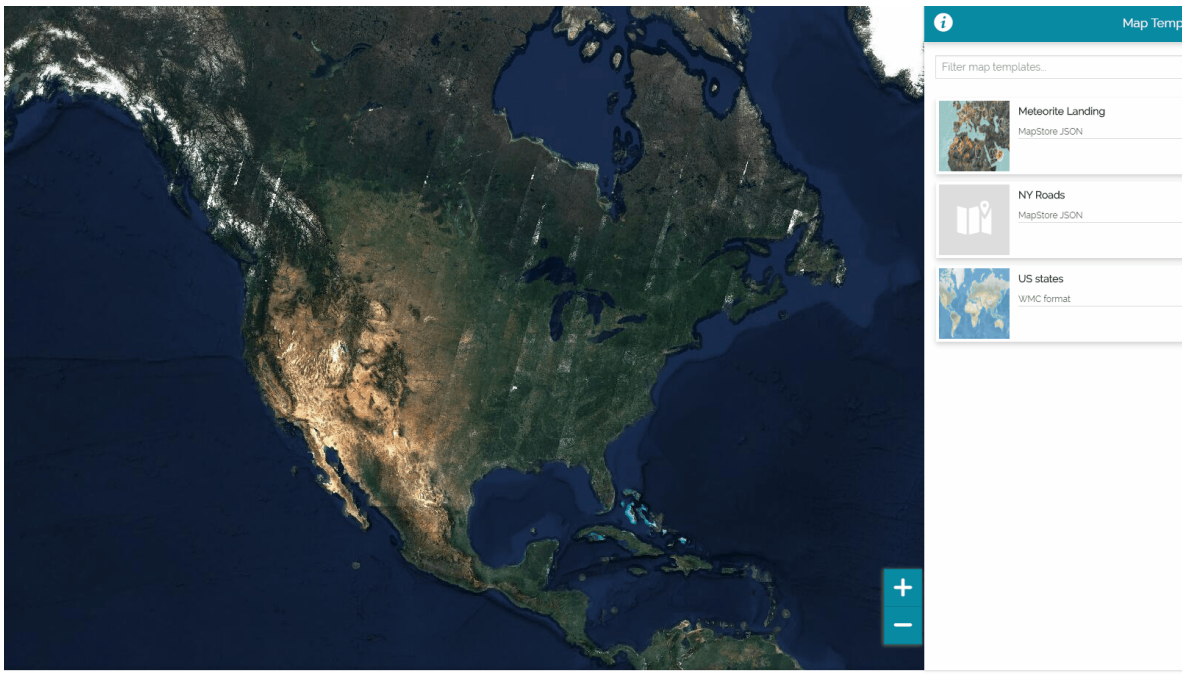
For each template in the *Map Templates* list the following buttons are displayed:



- The **Replace** button allows the user to entirely replace the current map with the one defined in the template, as follows:



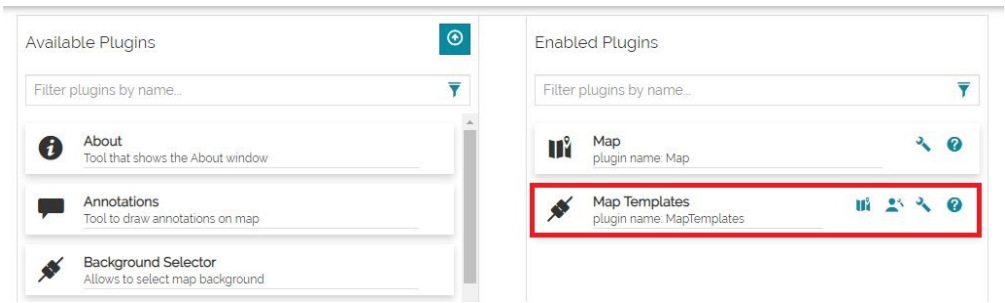
- The **Add Template** button allows the user to add the map template contents (layers) to current map without replacing it (by default a new group is created in that case in TOC, on top of the other ones, to contains layers coming from the template to better identify them), as follows:



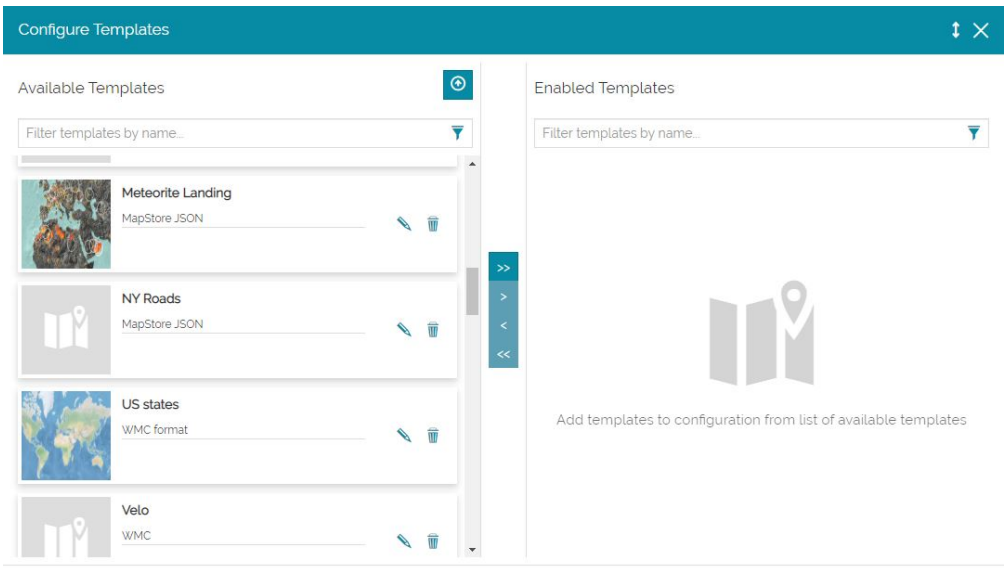
- The **Add to favorites** button ☆ allows the user to add the template to favorites on top of the list

Enabling the Map templates in a context

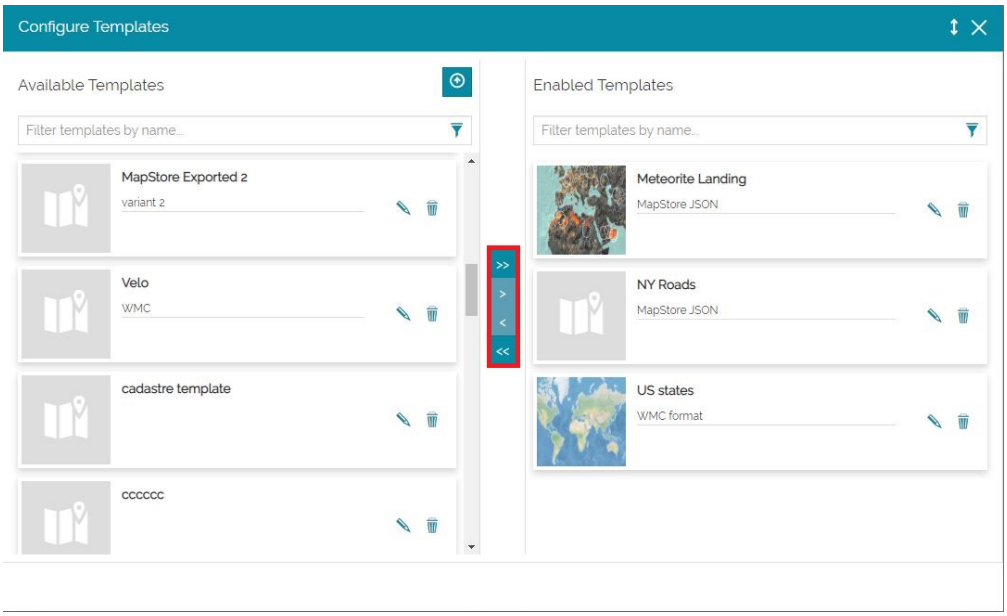
The *Map templates* extension is enabled by the admin in the Application Context wizard. In particular, this is possible in the third step of the wizard and after the extension is added to the *Enabled Plugins* list.



As soon as the **Configure templates** button is selected the *Configure templates* modal window opens, it allows the admin to manage the map templates.

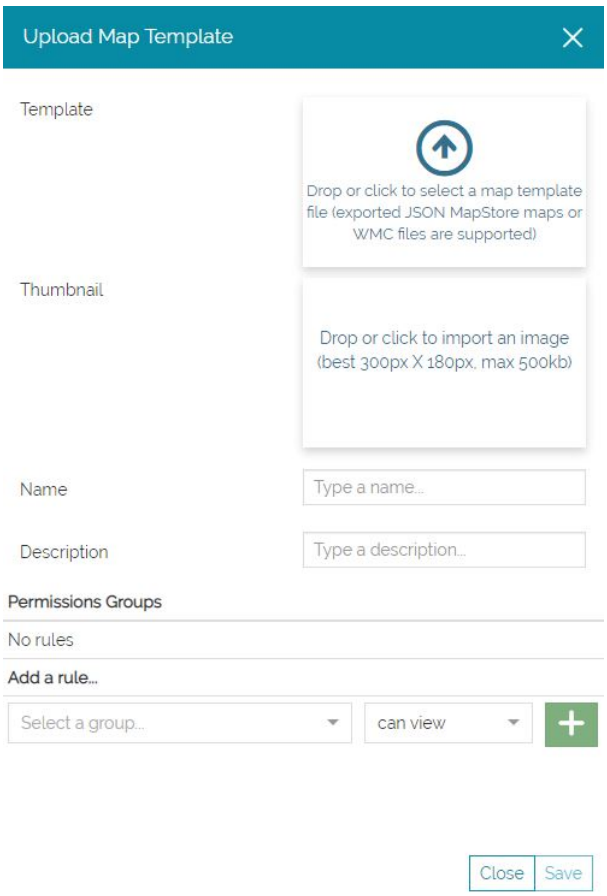


Through the *Configure Template* tool, the administrator can browse existing templates in MapStore and enable them for the context simply by moving the desired ones from the *Available Templates* list to the *Enabled Templates* list: this is possible with the central bar, as follows:



Uploading the template

It is possible for the administrator to create new Map Templates in MapStore by uploading new template files. In order to upload a new template the admin can select the **Upload new template** button  to open the **Upload new template** window:



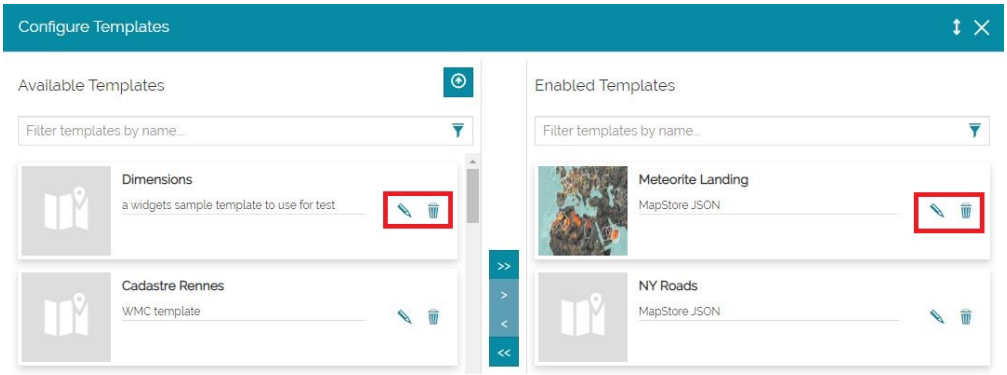
Here the admin, in order to import a template file, can drag and drop it inside the import area or simply click on that area to select it from the folders of the local machine.

> **Warning**
>
> The file that the admin can upload are:
>
> - The MapStore native map definition `json` format
> - The WMC (Web Map Context) file in `xml` format

The admin can also add **Thumbnail**, **Name**, **Description** and **Groups permissions** as describe here

Customize the template

The admin can also delete or modify an existing template through the buttons that are available on the left side of each templates item inside the *Configure templates* UI.

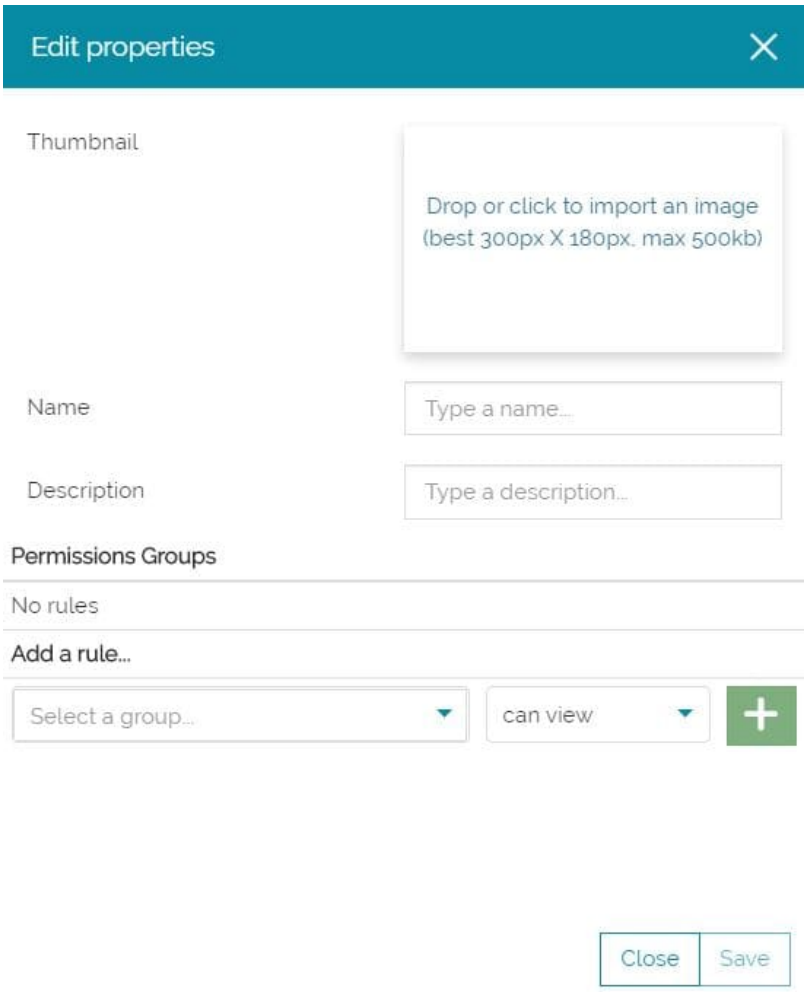


In particular, the admin can:

- Modify the template using the Edit properties that opens by clicking on the **Edit properties** button
- Delete the template through the **Delete** button

Resource Properties

In order to customize the properties of a resource, the Admin or a normal user with permission can access the *Edit properties* window from the **Edit properties** button in Homepage or from the **Save** and the **Save as** buttons inside the resource viewer.



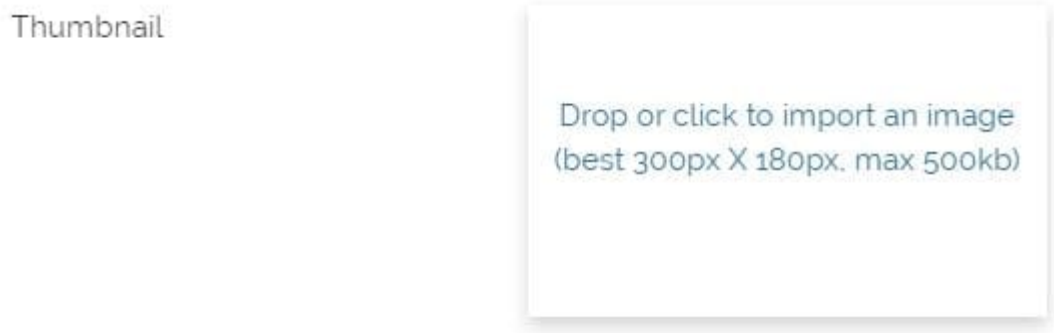
Through the *Edit properties* window the user can perform the following operations:

- Add a *Thumbnail*
- Add a *Name* and a *Description*
- Add a *Permission* rule

> **Warning**
>
> The name of a resource is the only mandatory field. Note that is not allowed to choose a name that has already been assigned to another resource.

Thumbnail

It is possible to add an image as thumbnail dropping it or clicking inside the following box:



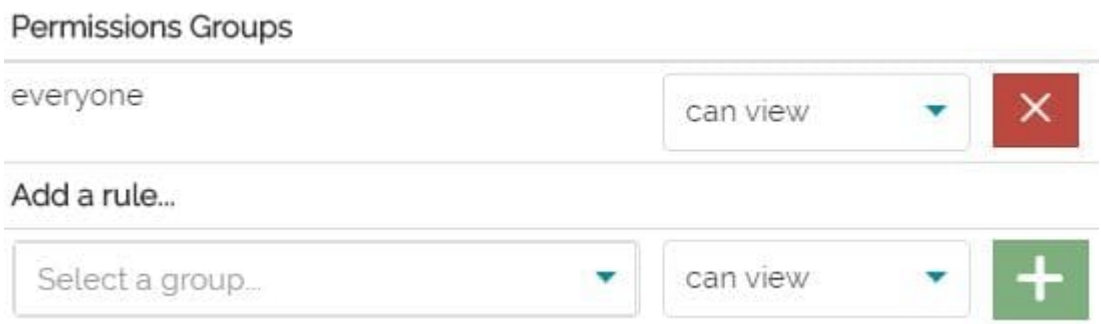
> **Warning**
>
> The image to be added must not be larger than 500 kb and its best dimensions are 300x180 px. The supported formats are `jpg` (or `jpeg`) and `png`.

Permission rules

In the *Add a rule...* section you can set one ore more permission rules in order to allow a group to access the resource. In particular it is possible to choose between a particular group of authenticated users or the *everyone* group that includes all authenticated users but also anonymous users (more information about different user types can be found in Homepage section).
Moreover it is possible to choose between two different ways with which the selected group can approach the resource:

- *View* the map and save a copy
- *Edit* the map and re-save it

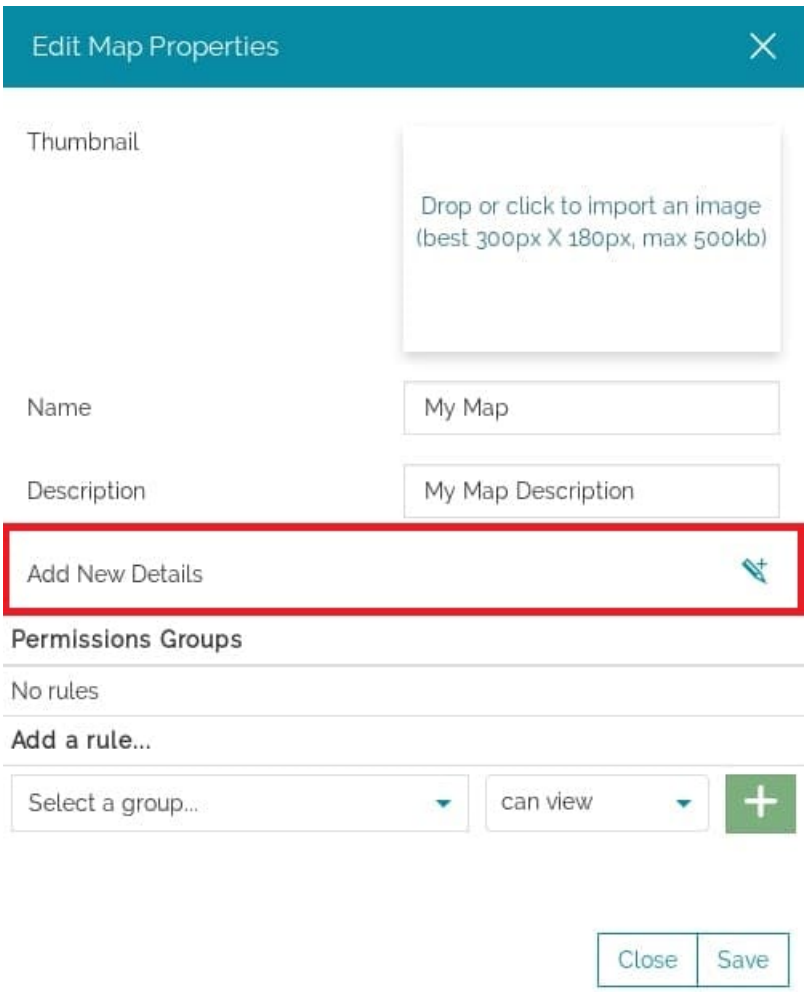
In order to add a rule, the user can select the group and set permissions inside the *Add a rule...* section. Once the rule is set, with the **Add** button  it is possible to add it to the *Permissions Groups* list.
For example, a resource that can be seen by *everyone*, should have a rule like the following:



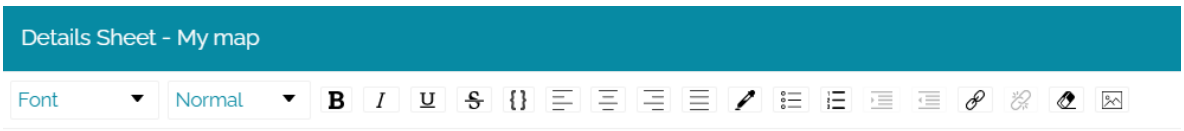
Once a rule is set, the user can always remove it through the **Remove** button  .
How to manage users and groups is a topic present in the Managing Users and Managing Groups sections.

Details

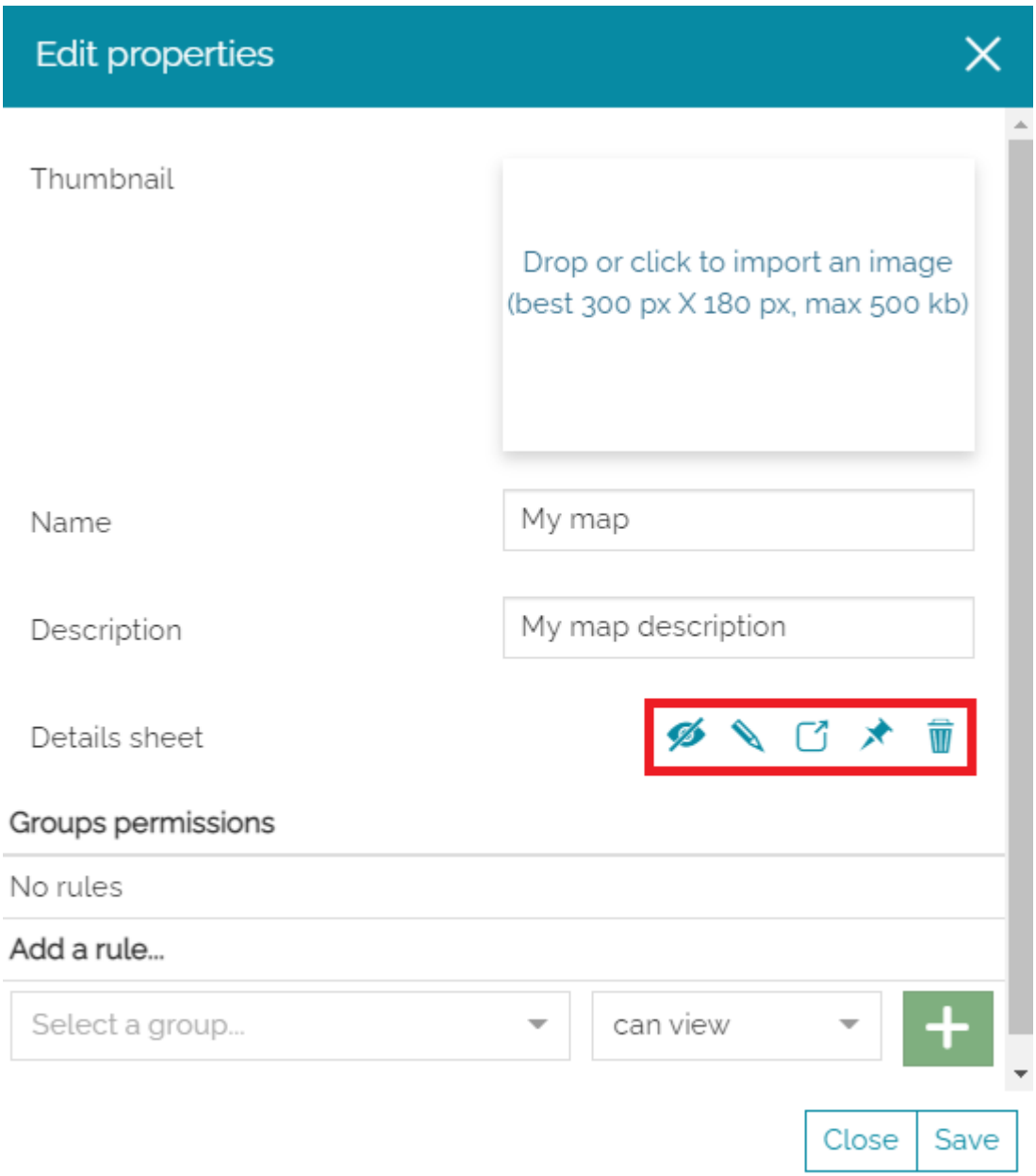
Only for resources of type map, it is possible to add details to the map. This is useful to associate some information to the map or an overview description of its content. In this case the *Edit properties* window is the following:



With a click on the **Add new details** button it opens a panel where the user can write the details of the map.

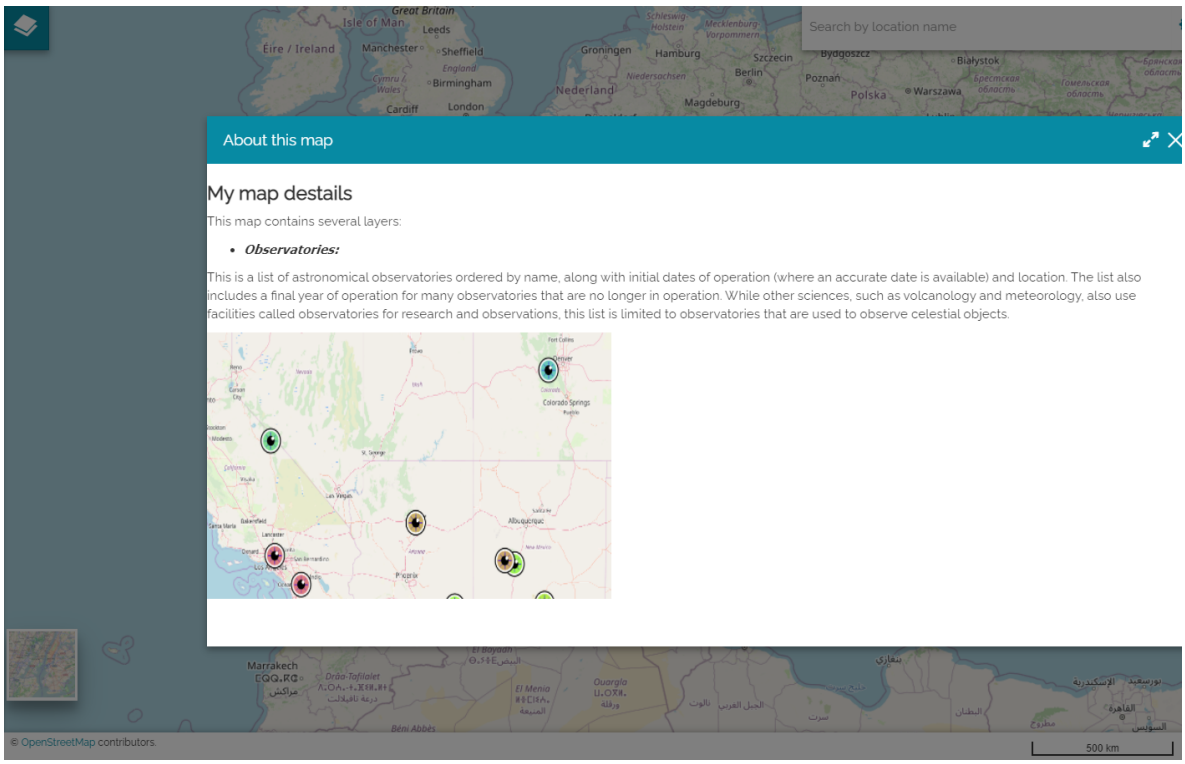


The text can be edited and some links and images can be added through the Text Editor Toolbar. Once the editing is done, the map details can be saved with the **Save** button Save and other buttons appear on the *Edit properties* panel.



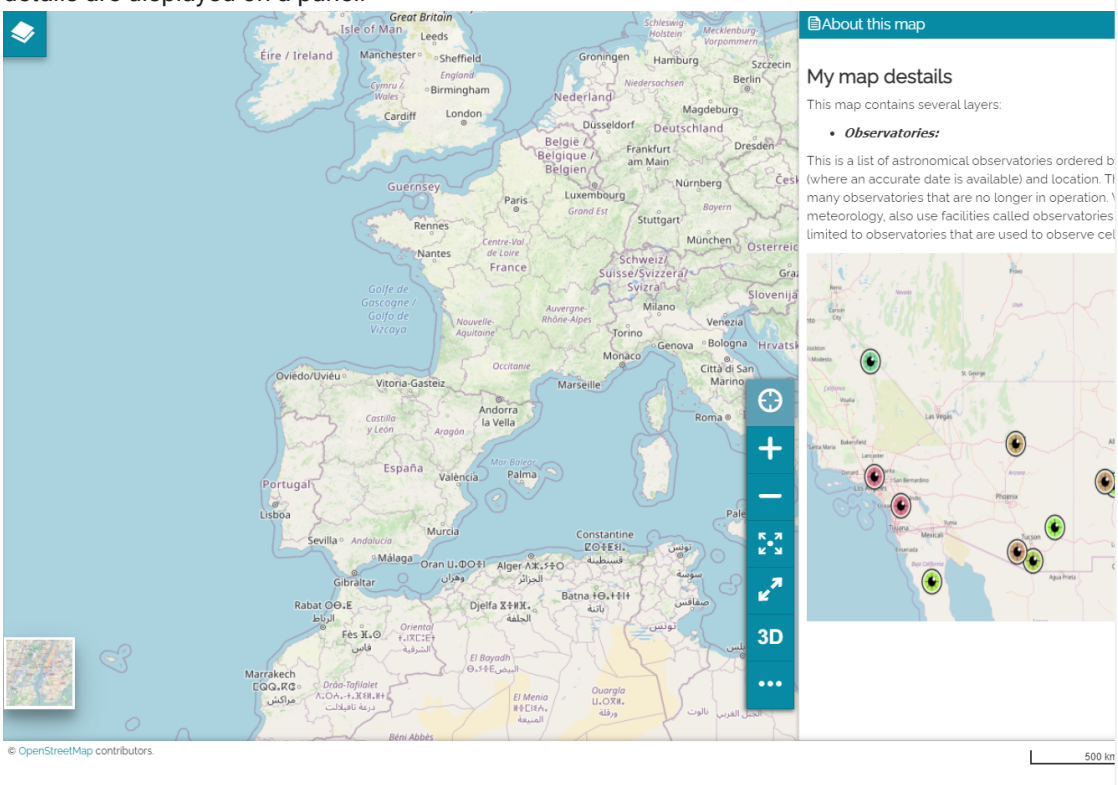
Here, the user is allowed to:

- Show the details **preview**
- **Edit** the details
- Enable the **Show as modal** button, to show the details on a modal when the user clicks on ABOUT THIS MAP button, which is listed in the Side Toolbar options



Note

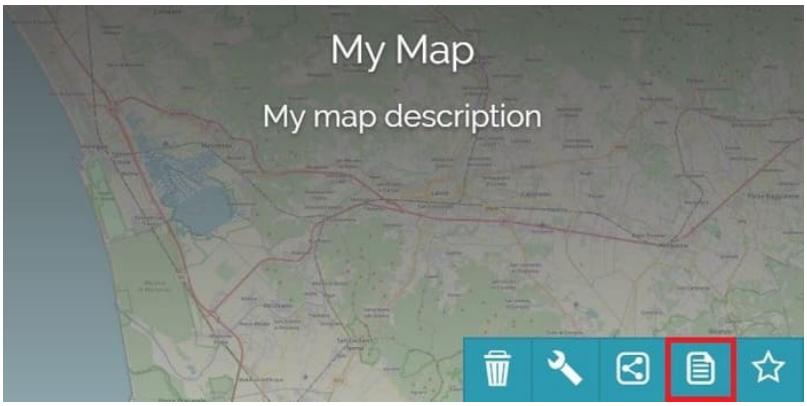
If the **Show as modal** button is not activated once the user opens the *About this map* button, the details are displayed on a panel.



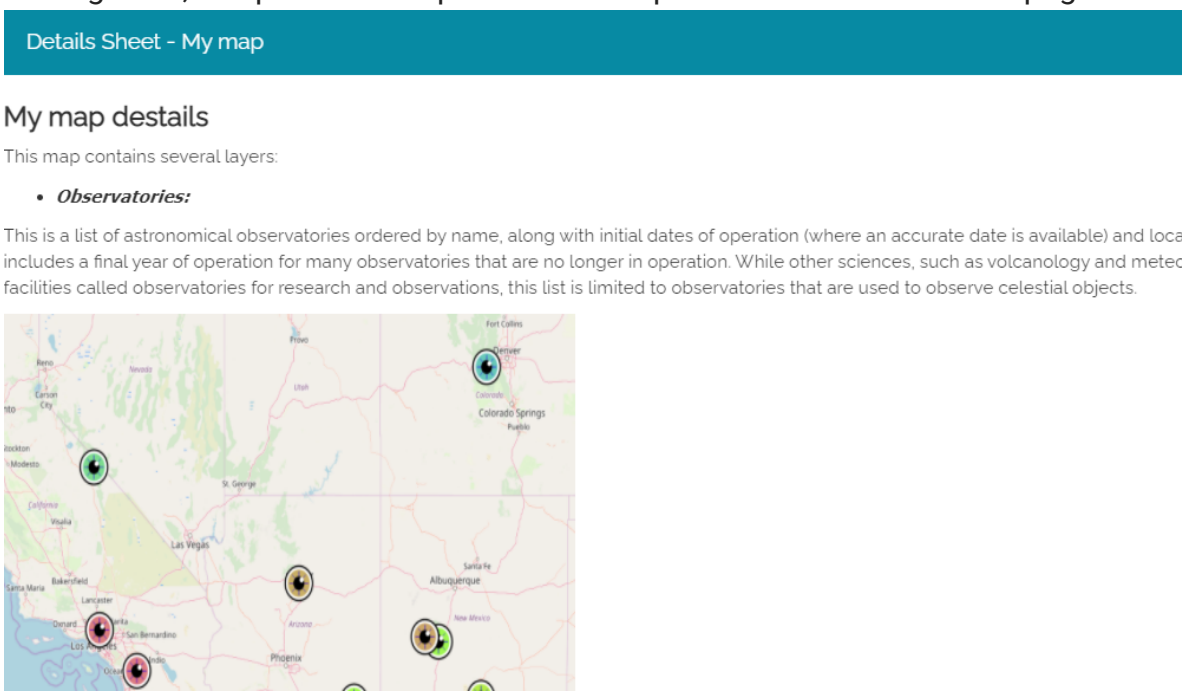
> **Warning**
>
> The *About this map* button is visible in the Side Toolbar only when the details are present on the map.

- Enable the **Show at startup** button. If active, as soon as the user opens the map, the details panel is visualized.
- **Delete** the details sheet

Once the details are saved, the **Show details** button appears also on the map card in Homepage



Through this, it is possible to open the details panel also from the home page.



Sharing Resources

MapStore provides the possibility to share resources (*maps*, *dashboards* and *geostories*) through two different ways:

- Directly from the MapStore Homepage by clicking on the Share button present in the toolbar of each resource card

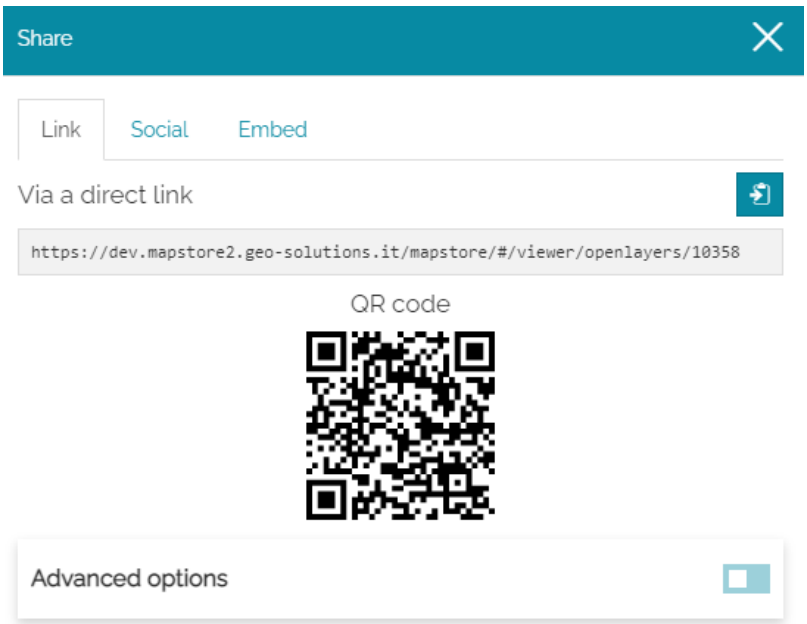
- Inside the resource by selecting the SHARE option from the Side Toolbar

From the *Share panel* the user is allowed to share a resource in different ways:

- With a **direct link**
- Through a **social network**
- With **embedded code** or **APIs** (only available for *maps*)

Link

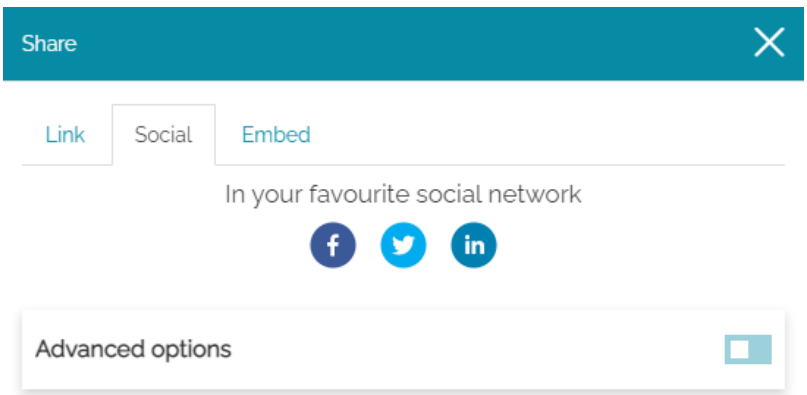
As soon as the *Share panel* opens, the **Link** section is the one visible by default:



Here, the user can copy the resource **URL link** or share it through the **QR code**.

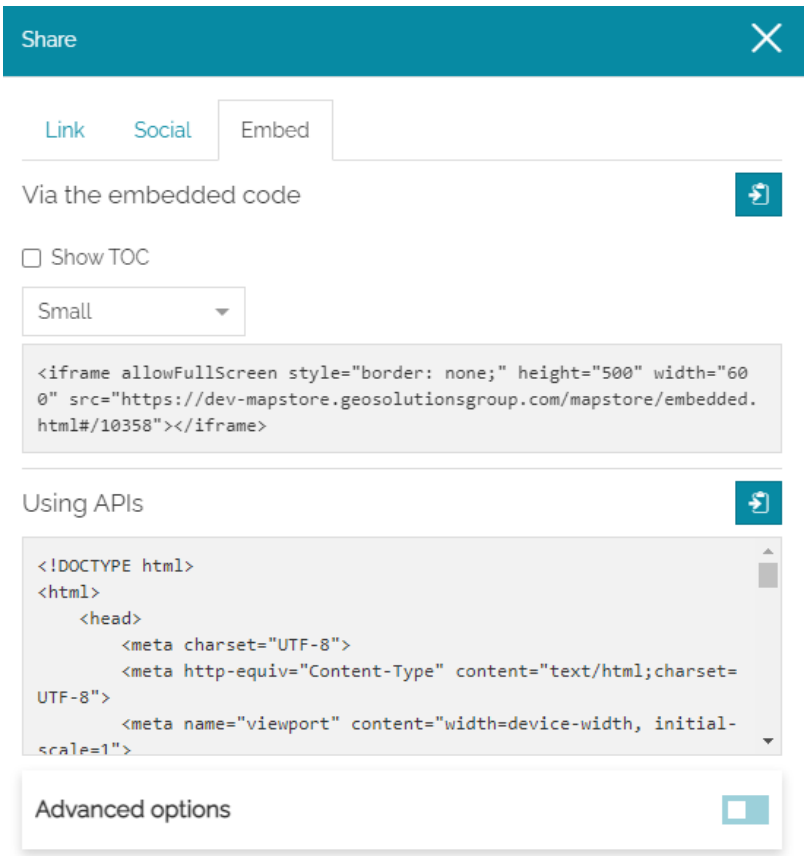
Social

The **Social** section allows the user to share the resource on the most common social networks like **Facebook**, **Twitter** and **LinkedIn** simply by clicking on the social icon.



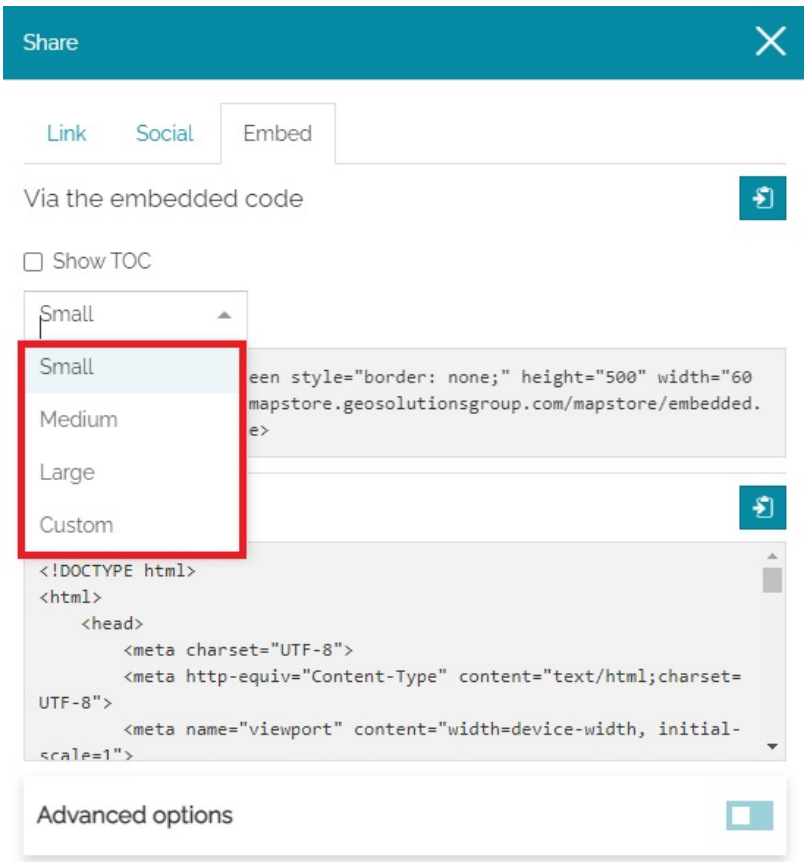
Embed

The **Embed** section provides to the user the needed snippets, **embedded code** or the **MS APIs** (only available for *maps*) to embed MapStore in a third party web page.

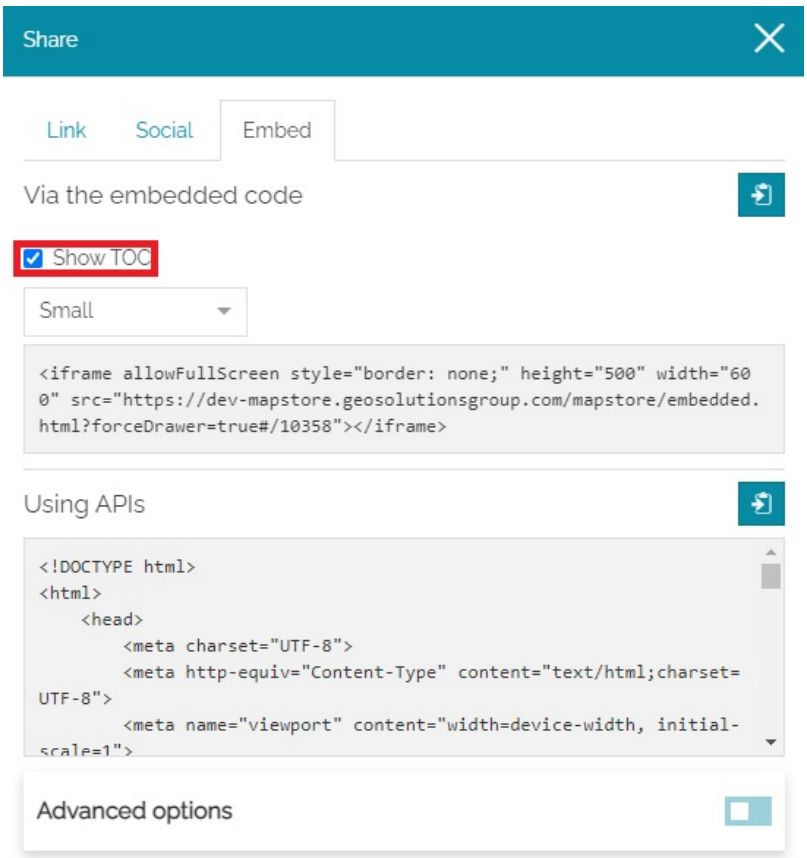


In addition, MapStore provides options to customize a bit the *embedded code*:

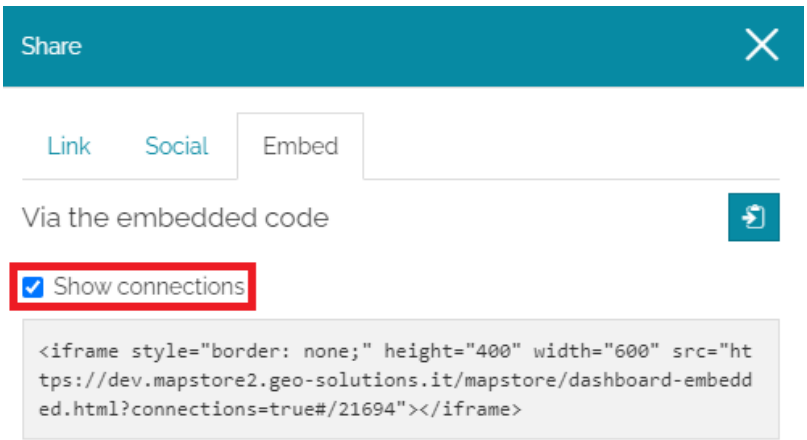
- The user can configure **height** and **width** of the embedded resource by choosing `Small` (*600x500*), `Medium` (*800x600*), `Large` (*1000x800*) and `Custom` (it is possible to choose the desired size).



- For maps, the user can choose to show the TOC in the embedded map by enabling the **Shown TOC** option



- For dashboards, the user can show the connections between widgets on the embedded dashboard by enabling the **Show connections**



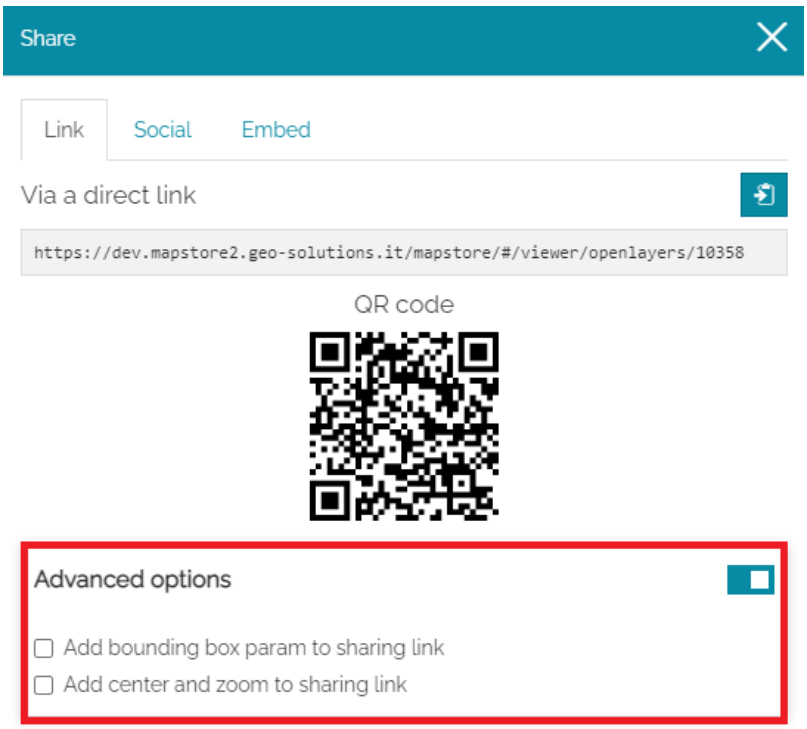
Advanced options

Some **Advanced options** are available for maps and geostories inside the **Share** tool.

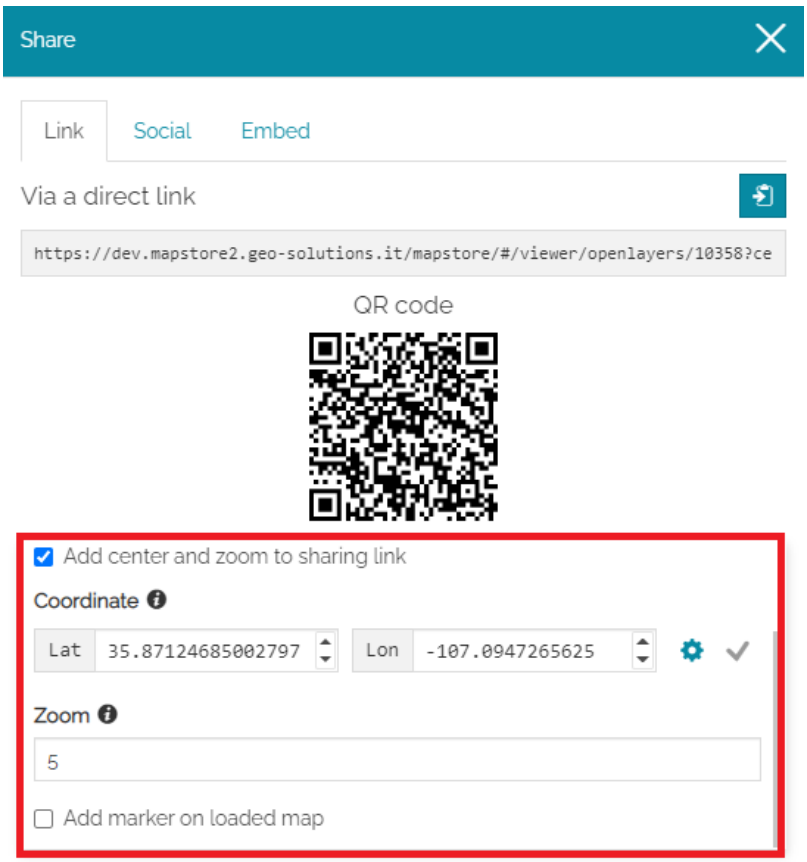
> **Note**
>
> Some **Advanced options** are available only opening *Share tool* from the Side Toolbar and not from the MapStore home page.

Advanced options for sharing maps

In case of maps, enabling the **Advanced options** in the *Share tool* the user can include the following to the share URL:



- The **bounding box** parameter to share the current viewport of the map visualized by the user
- The desired center and zoom of the map by enabling the **Add center and zoom**

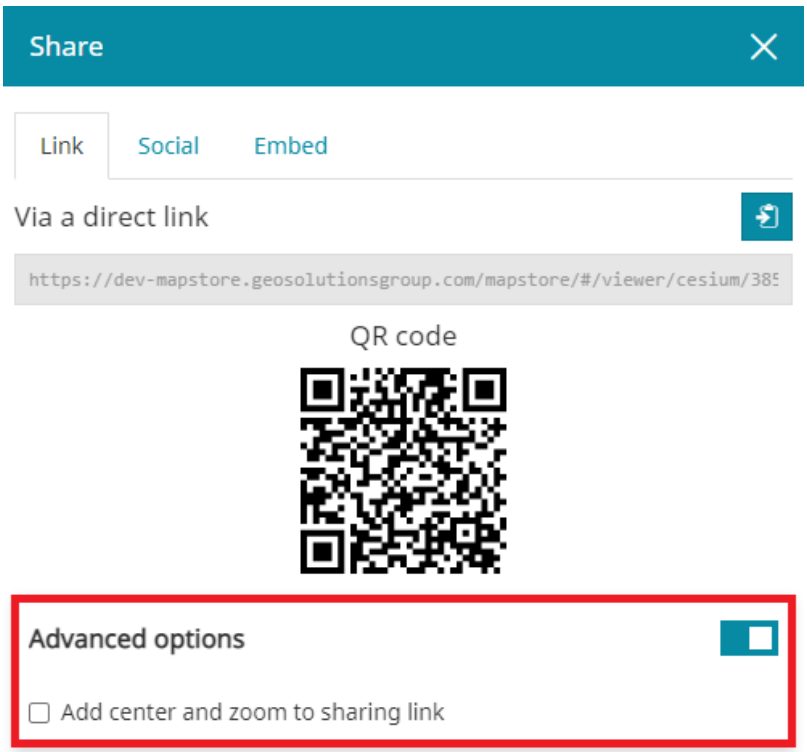


The related available options allow the user to:

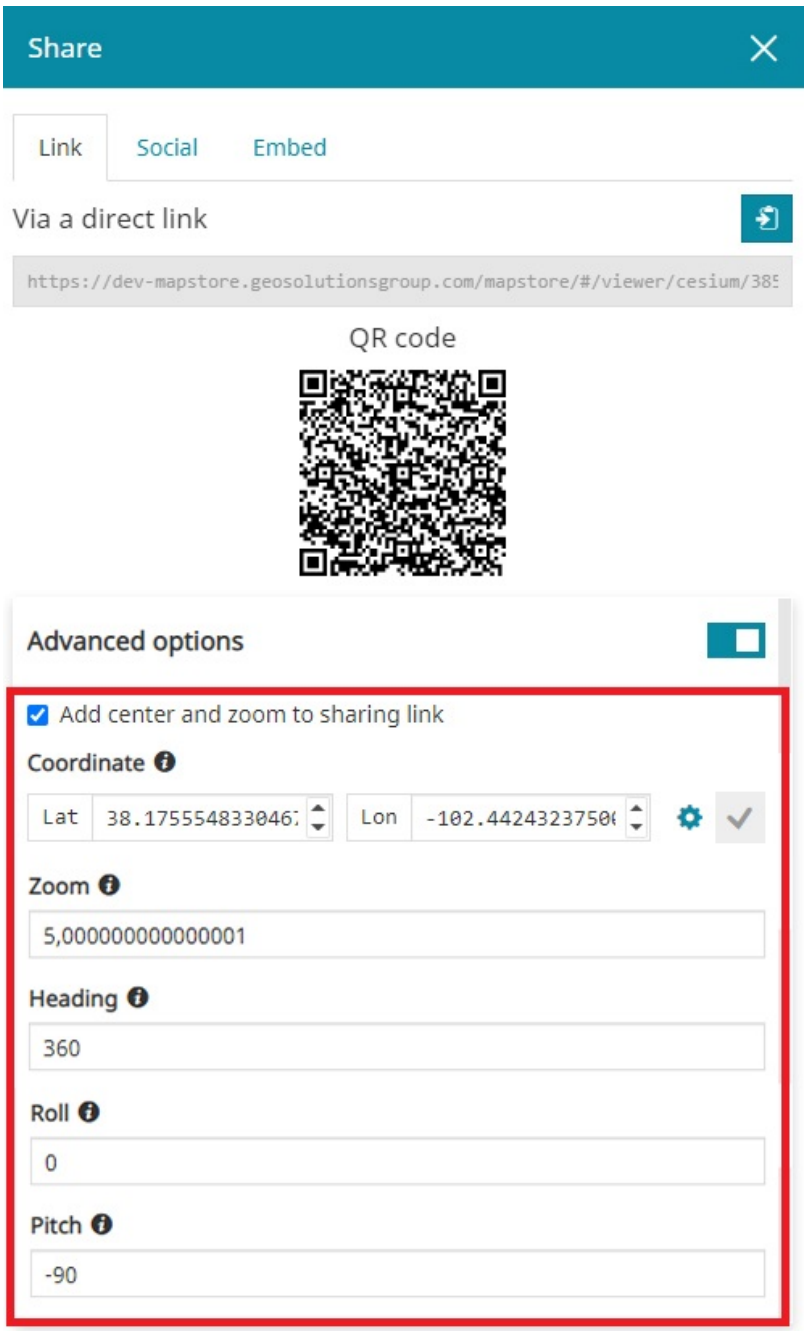
- Center the shared map to specific coordinates by typing them in two different formats (*Decimal* or *Aeronautical* that can be chosen through the ⚙ button) or by clicking on the map to set automatically the coordinate fields.
- Share the map at a specific **Zoom level** (`Min:1` and `Max:35`)
- **Add marker on loaded map** to show the center point in the shared map

Advanced options for sharing 3D maps

Once the 3D Navigation is active on map, the user can include the following to the share URL by enabling the **Advanced options** in the *Share tool*:



- The desired center and zoom of the map by enabling the **Add center and zoom to sharing link**

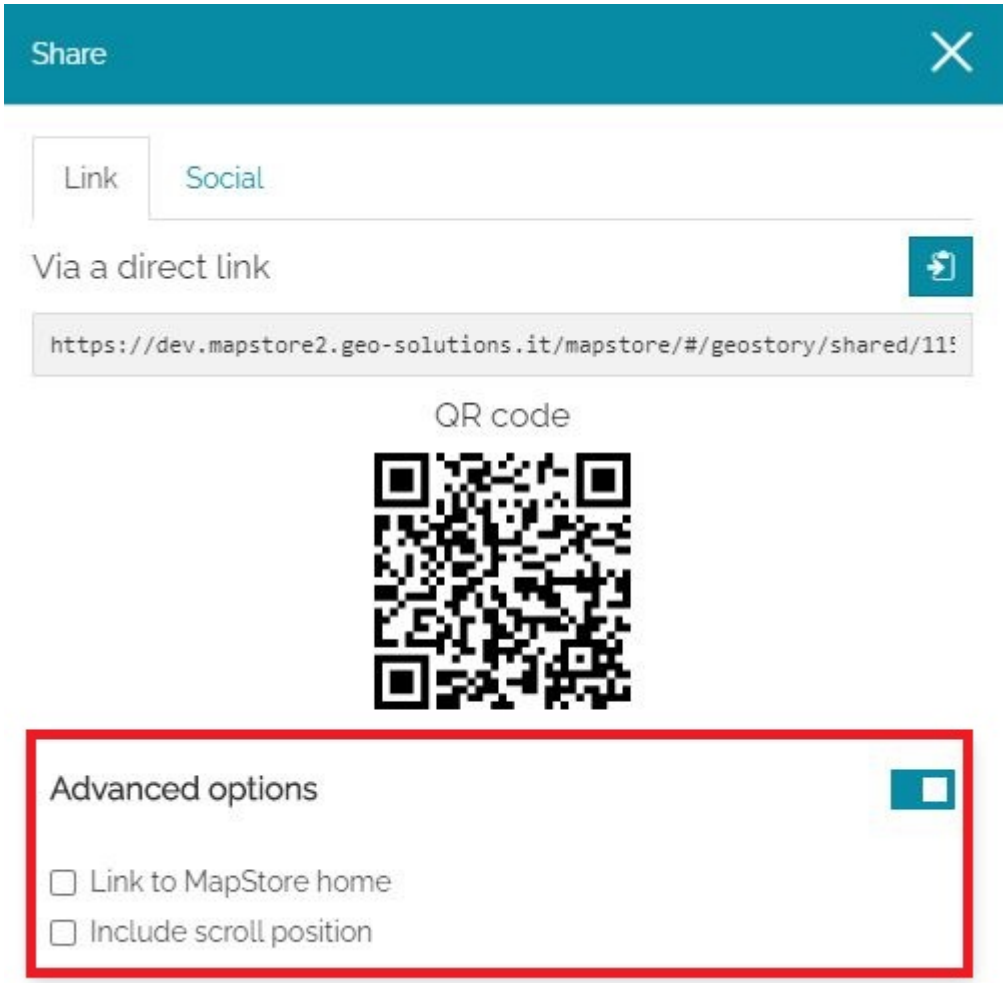


The related available options allow the user to:

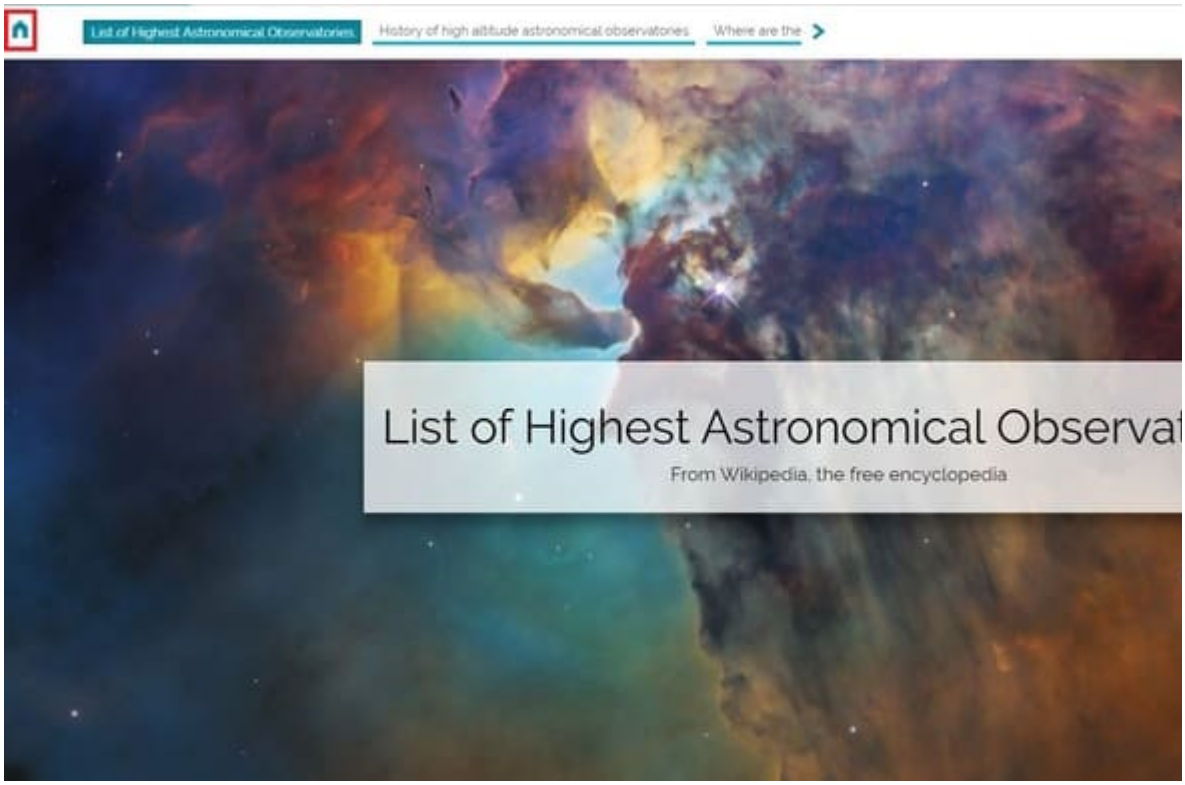
- Center the shared map to specific coordinates by typing them in two different formats (*Decimal* or *Aeronautical* that can be chosen through the ⚙ button) or by clicking on the map to set automatically the coordinate fields.
- Share the map at a specific **Zoom level** (`Min:1` and `Max:35`), **Heading** (`Min:0°` and `Max:360°`), **Roll** (`Min:-90°` and `Max:90°`) and **Pitch** (`Min:-90°` and `Max:90°`)

Advanced options for sharing GeoStories

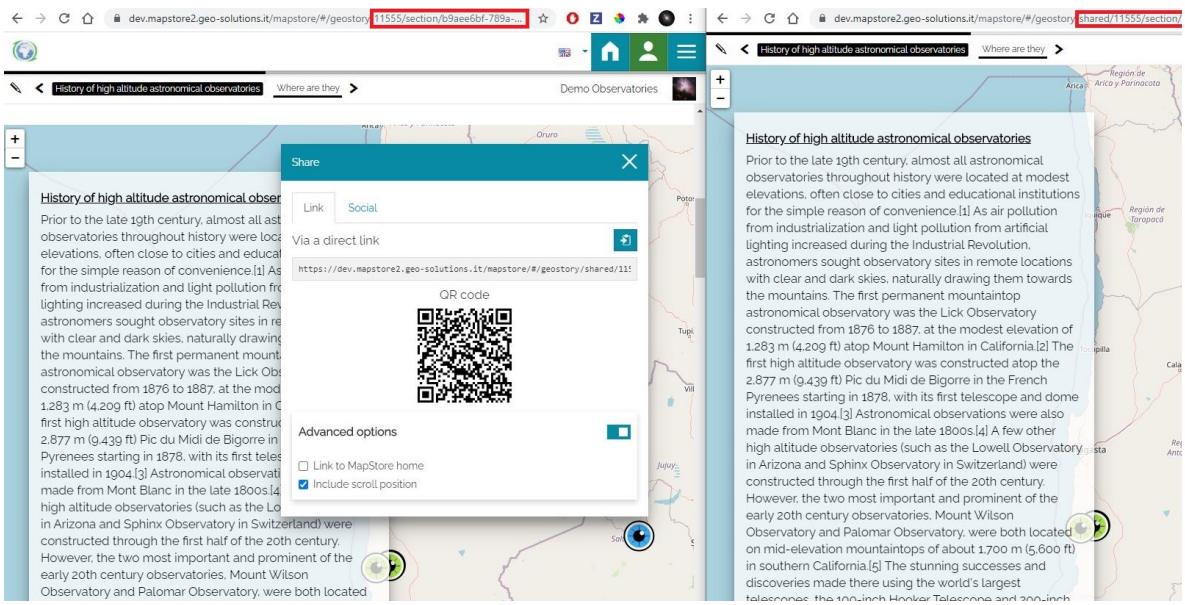
In case of GeoStories, enabling the **Advanced options** in the *Share tool* the user can include the following to the share URL:



- The **Home button** to allow the possibility to bring the user to the MapStore Home Page if needed: that button will be automatically included in view mode inside the story toolbar just beside the navigation bar.



- The **scroll position** allows to share the URL of the current section of the story visualized by the user



Exploring Maps

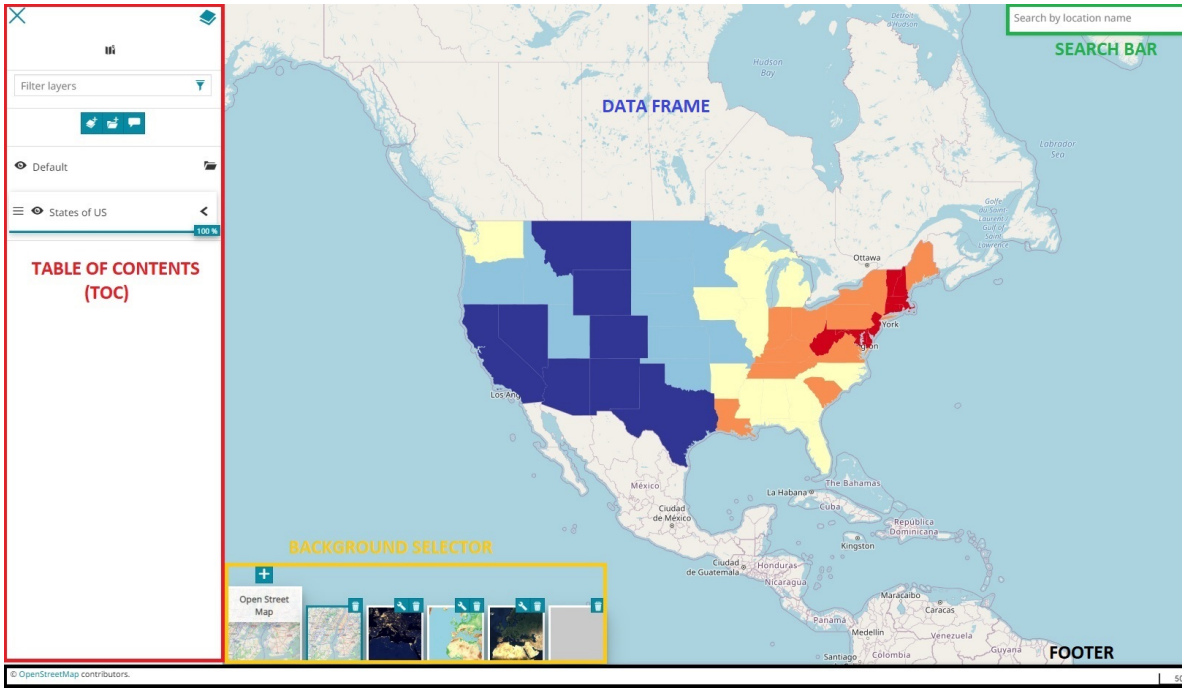
In cartography, a map is any two-dimensional graphic representation of the spatial relationships of the whole or a part of the earth. In digital cartography as in MapStore, a map consists in overlaying various layers of geographic data and their styles in data frames, and it contains various map elements such as a legend and a scale bar.

In order to create a map, the user can click on the **New Map** button  in Homepage and will be addressed directly to the map viewer (by default only Administrators and Normal Users can create a new map, as explained before in Homepage section):

Once a map is created and saved, it will be available in Homepage content section.

MapStore WebGIS Portal Interface

The Mapstore WebGIS Portal interface is composed by the following main blocks:



In particular:

- The Table of Contents (TOC) shows the layers and the layers groups on the map and allows to remove or edit them, and add some new ones
- The MapStore Toolbars includes the Search Bar and the Side Toolbar, an important list of options that contains several functions and information
- The Sidebar that is mainly a navigation panel
- The Background Selector allows to add, remove or edit map's background
- The Footer includes the CRS selector, the coordinates, the scale and the credits of the layer
- The *Data Frame* is the space where the layers are displayed

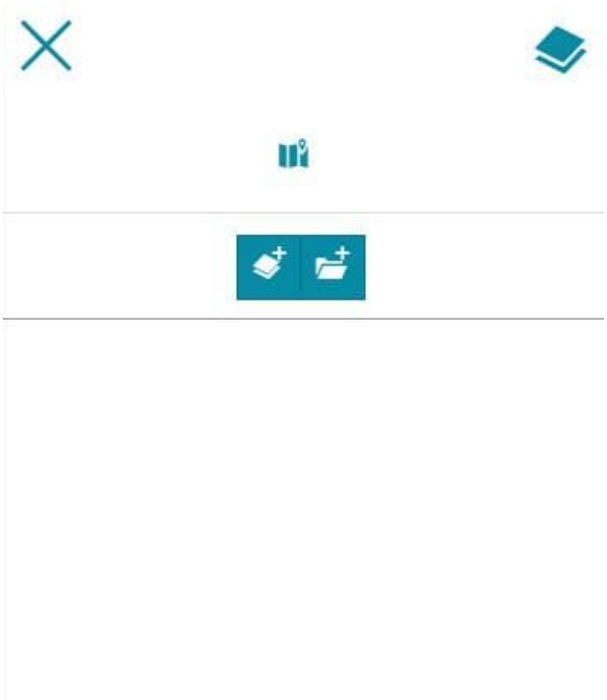
Table of Contents

The Table of Contents, briefly TOC from now on, is a space where all the layers and the layers groups are listed. Through this panel it is also possible to carry out the following operations:

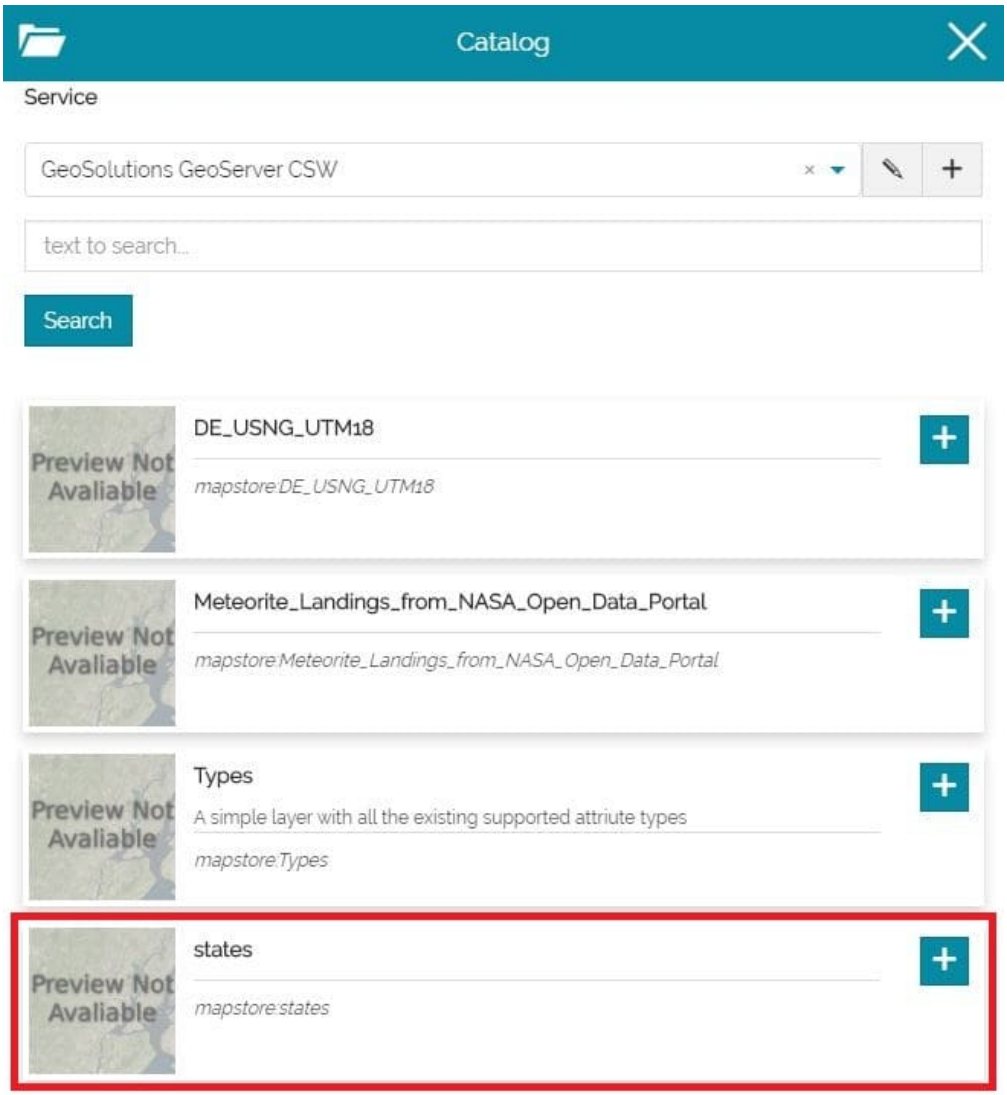
- Add and remove layers and groups
- Perform a search between layers
- Change the position (and consequently the display order in map) of layers and groups
- Set some display options directly from the panel
- Manage layers and groups and query layers through the toolbar actions

Add and remove layers and groups

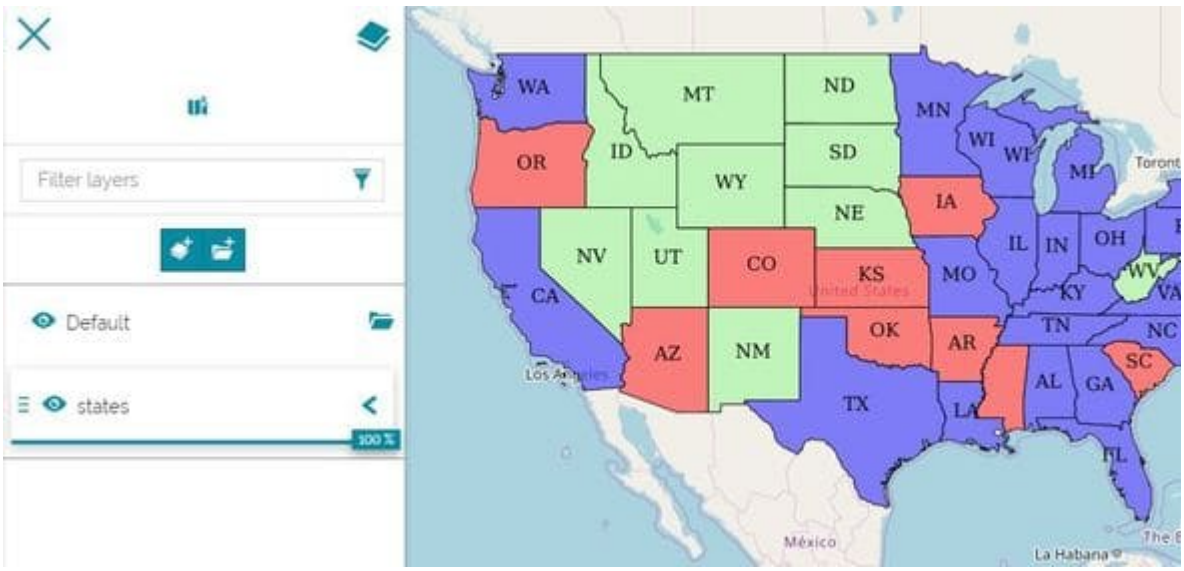
The user can access the TOC with the **Layers** button  on the top-left corner of the map viewer. For example, in a new map, the following panel appears:



The **Add Layer** button opens the Catalog, a panel where it is possible to choose the desired layer and add it to the map with the **Add to Map** button :



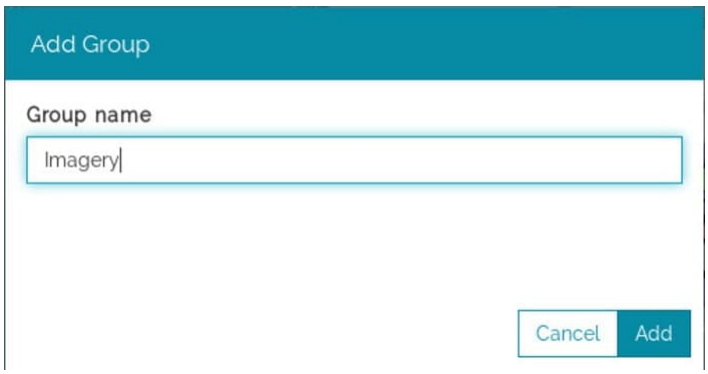
Once the layer is added to the map, the result should be like the following:



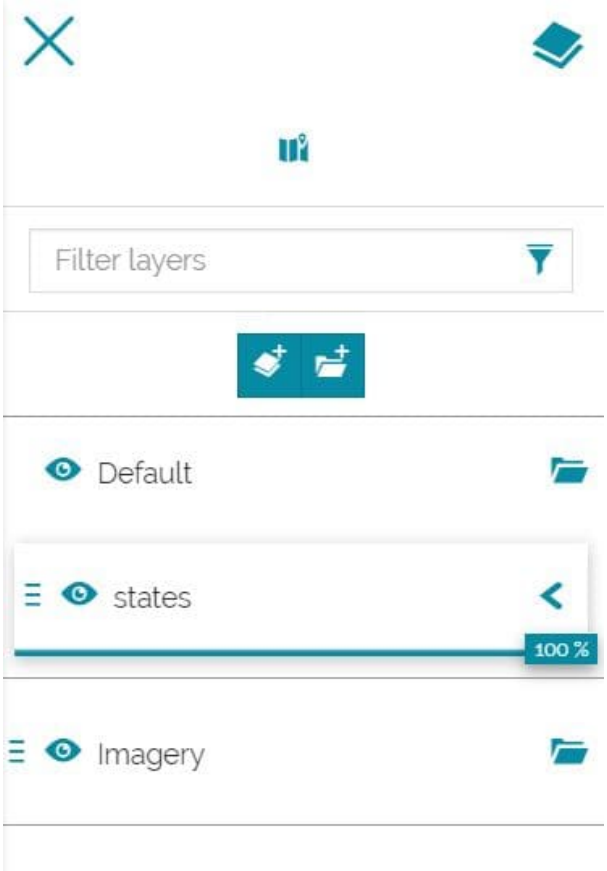
Note

When a layer is added for the first time to the TOC, without any group present, the *Default* group is created. This group host all the layers that don't belong to a specific group and can also host sub-groups within it.

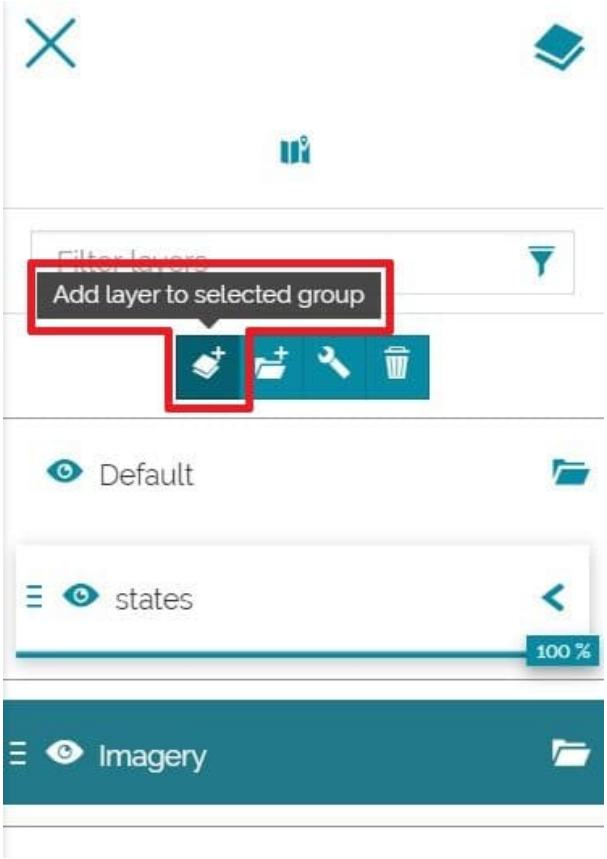
In order to add a new group, clicking on the **Add Group** button the following window opens:



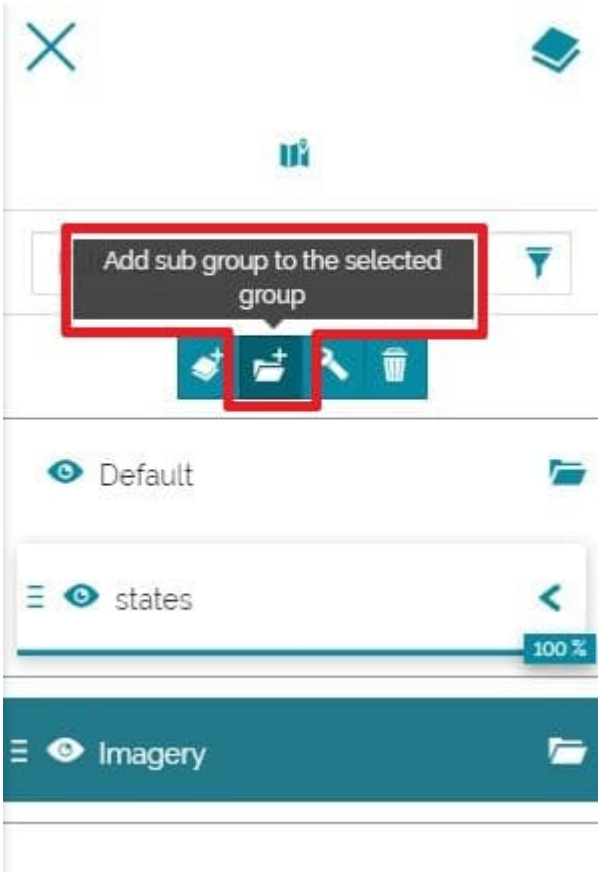
Once the name of the group is typed, with the Add button the new group is added to the TOC.



In order to add a new layer to a specific group, it is possible to select that group and click on **Add layer to selected group** :



In order to add a subgroup inside a specific group selected, the user can click on the **Add sub group to the selected group** button  (maximum 4 subgroup levels are allowed):

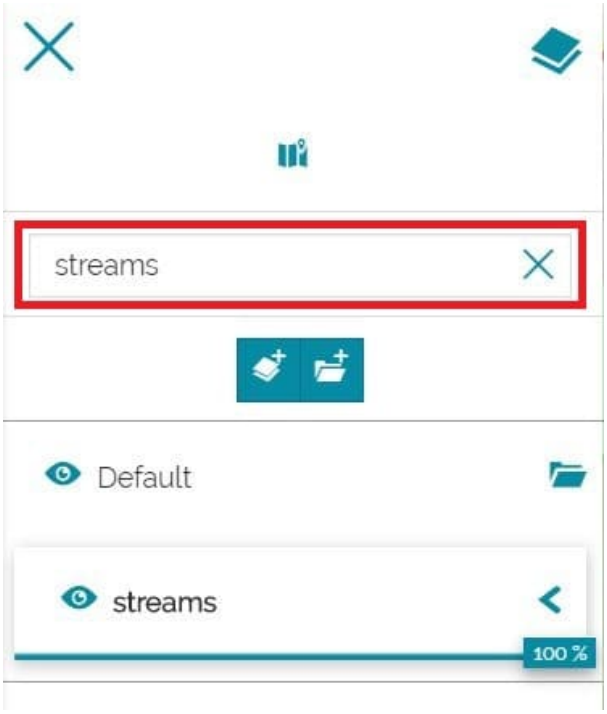


Layers and groups can be removed selecting them and clicking on the **Remove** button  present in the toolbar of each selected layer and group.

> **Warning**
>
> When a group is removed, also all the layers and subgroups associated with it will be removed.

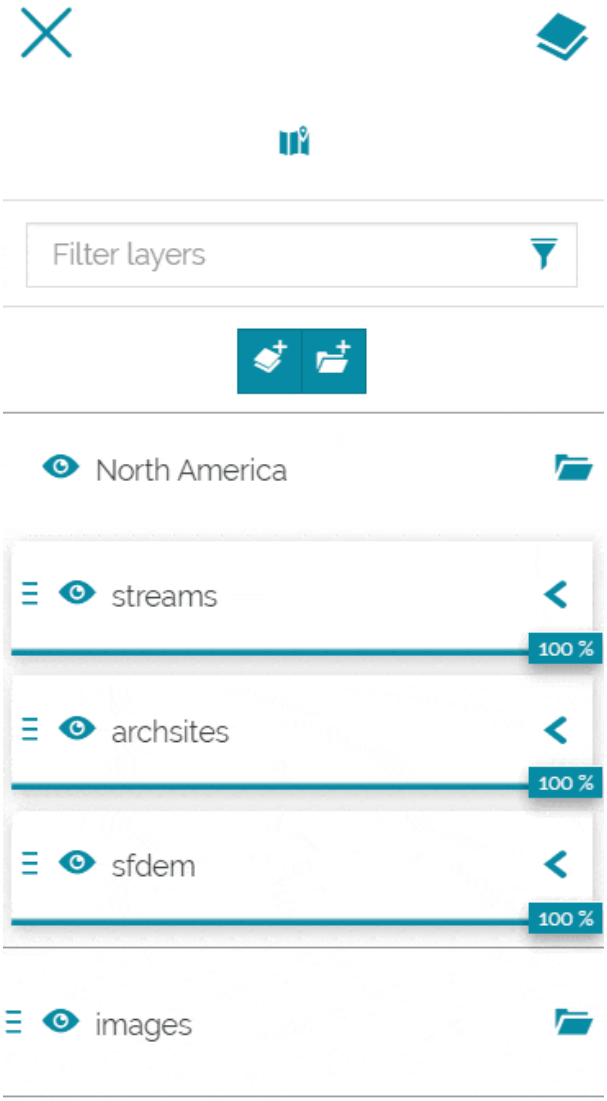
Search for layers

With the TOC it is also possible to perform a search between the added layers. This operation can be done simply by typing the name (o part of it) of the layer in the search bar:

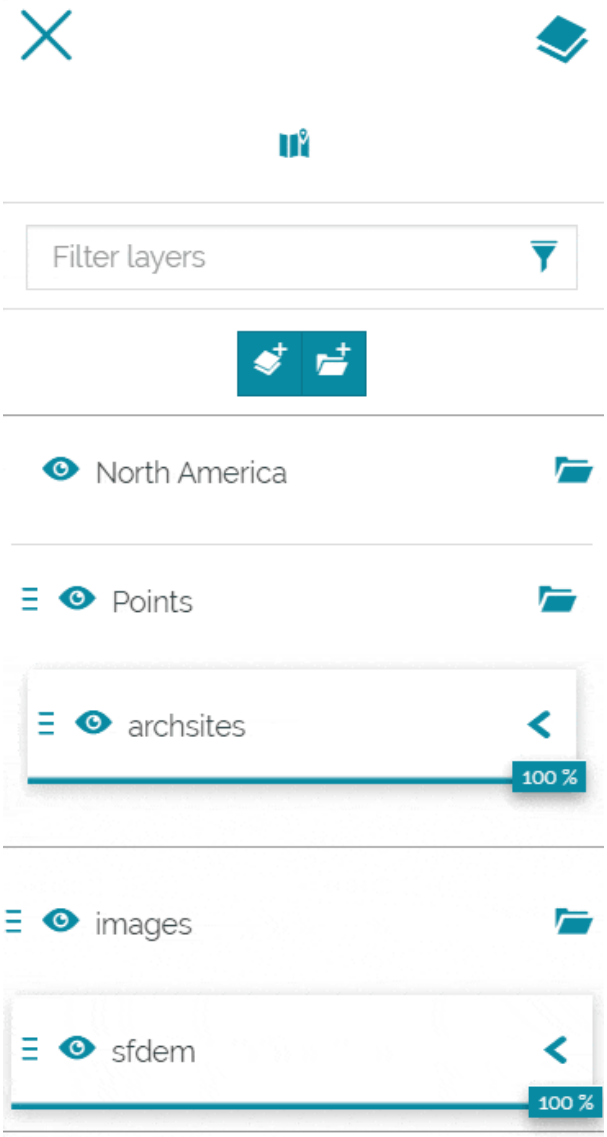


Choose layers and groups position

With the drag and drop it is possible to change layers position inside the same group, but also moving them between different groups. Once the *Default* group is created, all the layers without a specific group are automatically added to this one. Changing layers position with the drag and drop, for example, it can display like the following:



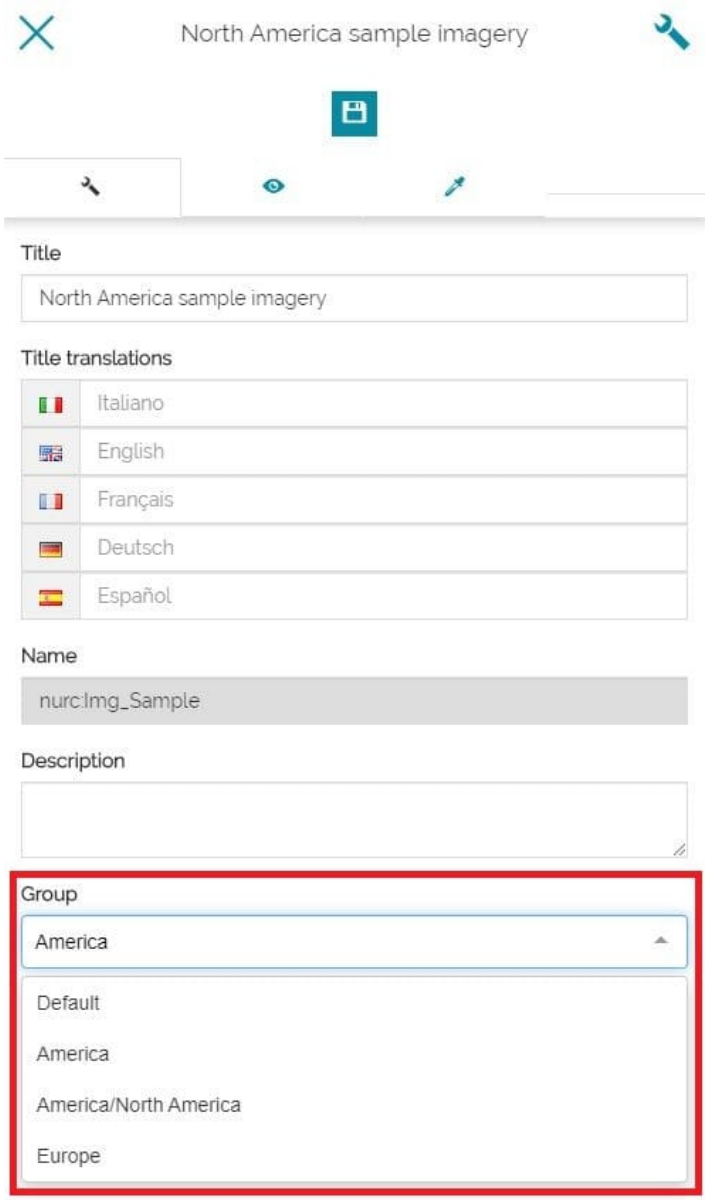
Groups and sub-groups, no matter their level, can be nested inside other groups and sub-groups, or can be separated from their parent-level to create new main groups. These operation can be performed, again, with the drag and drop function.



Warning

The only constraints applied to the groups manager refer to the *Default* group (each layer added to the map the first time is included in that group). Drag and Drop operations are not allowed for the *Default*, but it's allowed to rename it or to nest groups or sub-groups inside it.

Layers position can also be determined through the **Selected layer settings** button available in the toolbar that appears once a layer is selected. This button opens a panel where the user can choose the destination group (or subgroup):



Display options in panel

Directly from the TOC panel, it is possible to set different types of display options. In particular, for layers, it is possible to:

- Toggle layers visibility by switching on and off the "eye" icon to the left of the layer name
- Expand or collapse the legend by clicking on the icon. The width and height property of the legend can be overridden via Legend options under Display tab.
- Control the transparency in map by scrolling the opacity slider

> **Note**
>
> When the user switch off the visibility of a layer, also the group where that layer is nested change the "eye" icon in (no matter if other visible layers are present in that group)

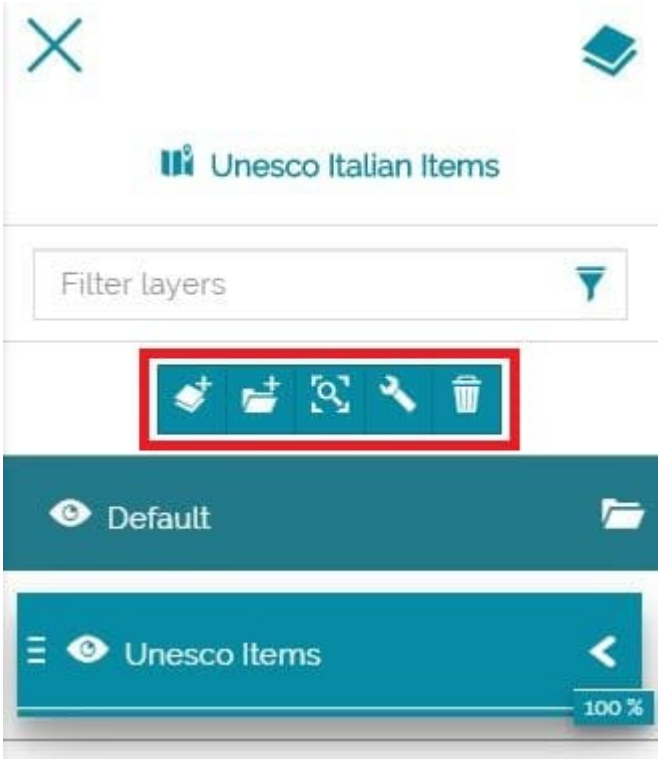
With groups there's the possibility to:

- Expand or collapse the list of layers or subgroups nested inside it
- Toggle groups visibility by switching on and off the "eye" icon to the left of the group name

> **Note**
>
> When the user switch off the visibility of a group, also the visibility of all the layers and subgroups nested inside it will be automatically switched off.

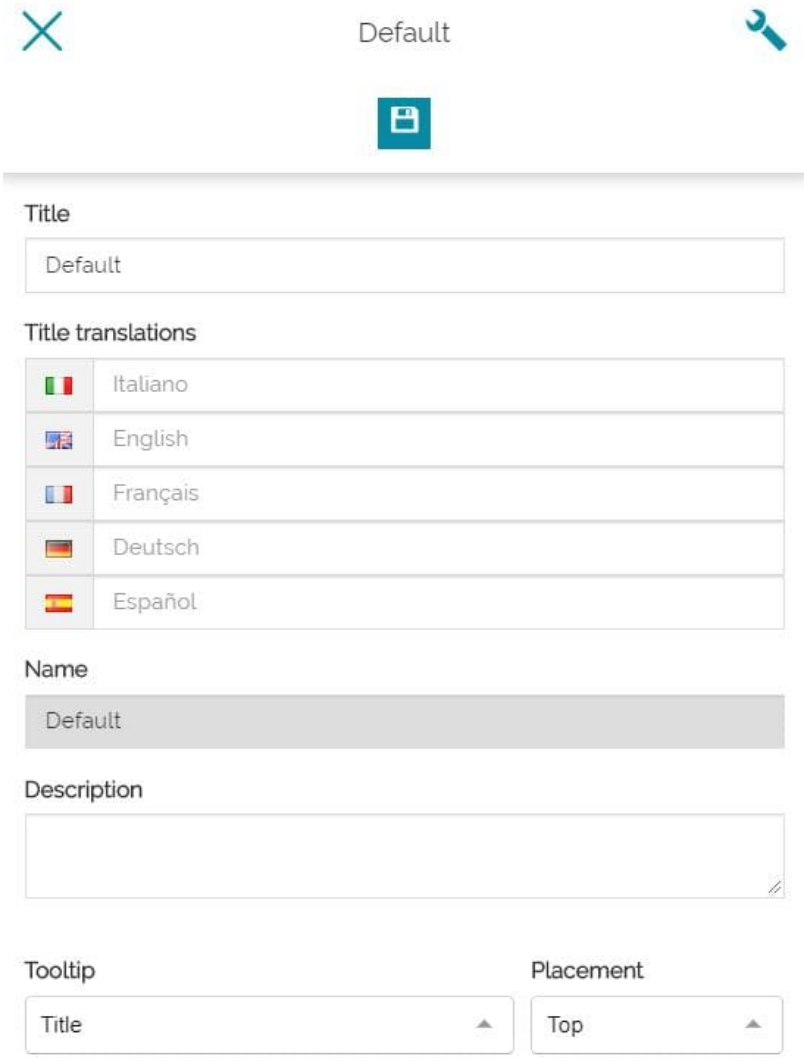
Toolbar options

Once a group is selected the following toolbar appears:



Through this toolbar it is possible to:

- **Add layer to selected group** : it is possible to add one or more layers to the group
- **Add sub group to the selected group** : it is possible to add one or more sub-groups to the selected group
- **Zoom to selected layers extent** : in order to zoom the map to all layers belonging to the group
- Open the **Selected group settings** where it is possible to change the group's title, the title translations and see the group name (its ID). It is also possible to add/customize the description of the group and configure the tooltips placement in the UI (more information can be found in Layer Settings section)

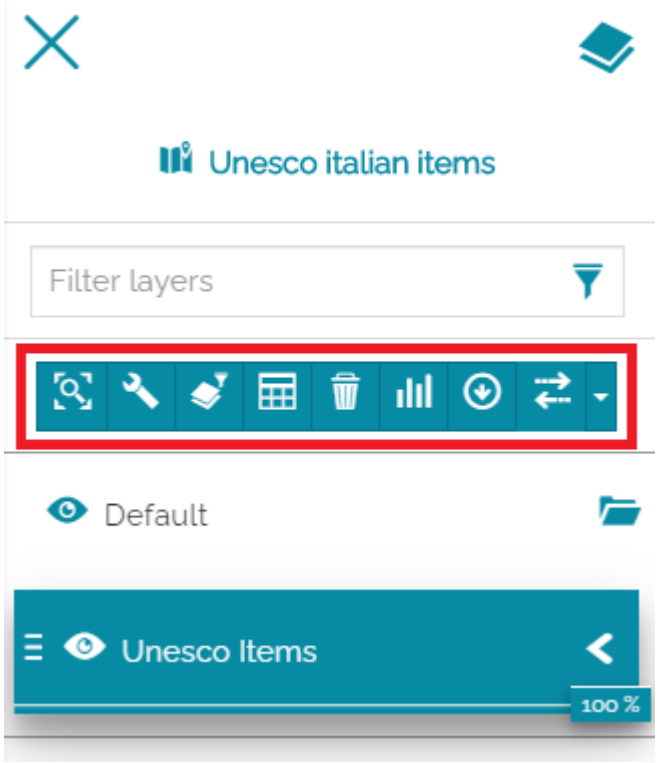


- **Remove selected group** and its content

Once the changes have been made, it's possible to save them through the **Save** button .

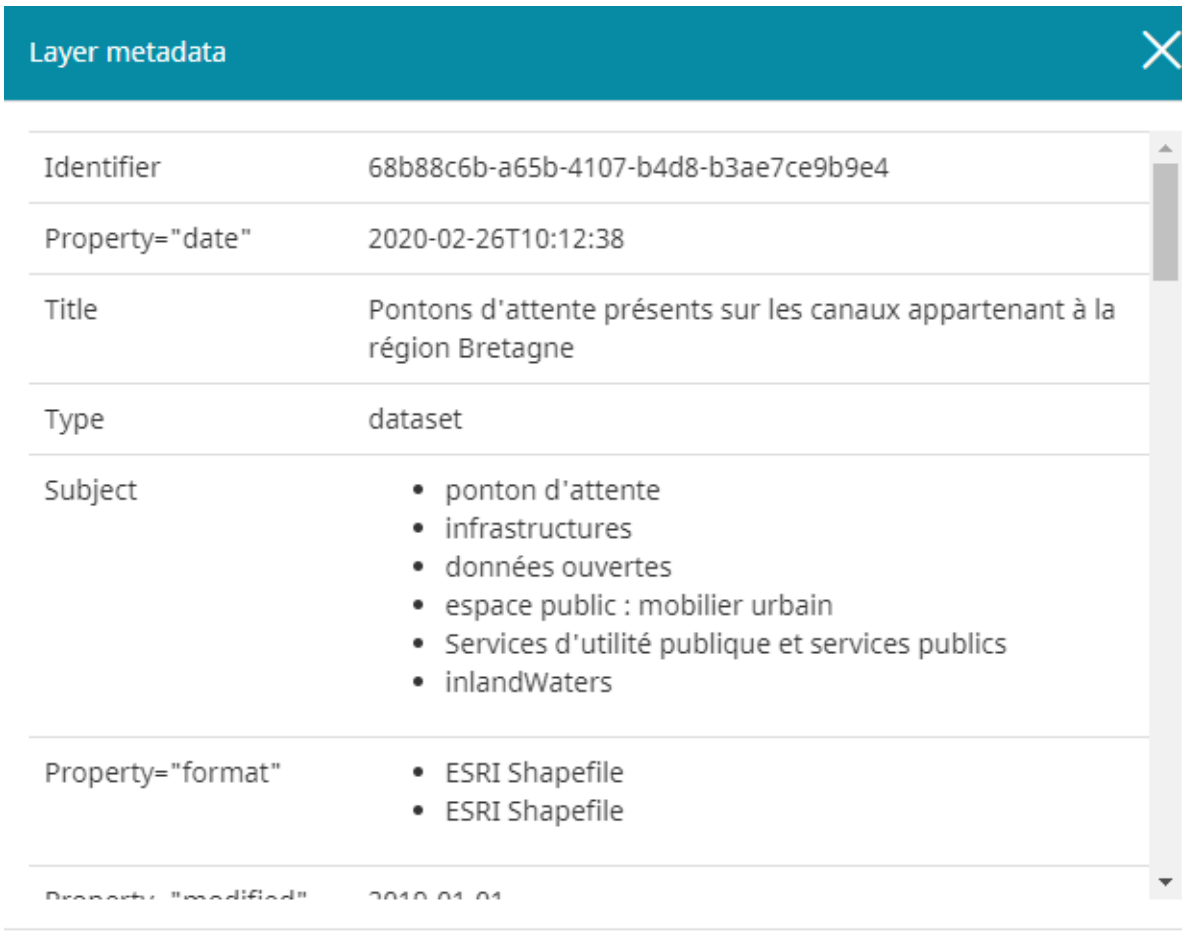
> **Note**
>
> The information thus modified will be kept only within the current user session. In order to make these kinds of changes persistent across different user session, the map needs to be saved.

Selecting a layer, the toolbar is the following one:



In this case the user is allowed to:

- **Zoom to selected layer extent** : in order to zoom the map to the layer's extent
- Access the selected Layer Settings
- Set a Filter for that layer
- Access the Attribute Table
- **Remove** the selected layer
- Create Widgets for the selected layer
- Export the data of the selected layer
- Open the **Layer Metadata** (if configured), to retrieve layer metadata from the remote catalog source.

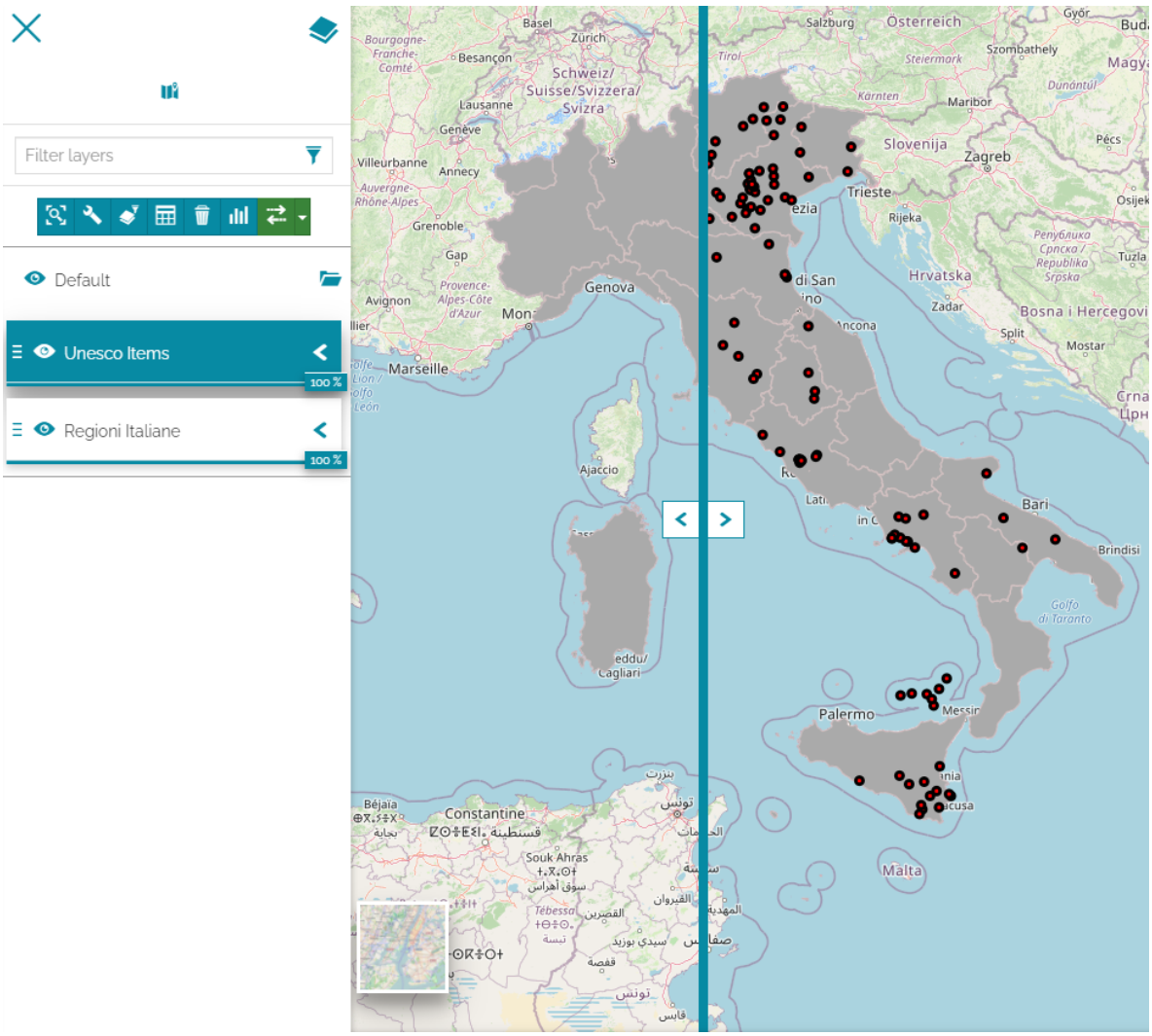


Note

The **Metadata Tool** is not configured by default in MapStore. A complete documentation to configure it is available as part of the TOC Plugins documentation (see *metadataOptions*). Once the **Metadata Tool** has been configured, MapStore is able to load the layer metadata from the remote CSW service and parse it to be presented to the user according to the provided plugin configuration. This functionality automatically works in case of WMS layers coming from a CSW catalog source, while for layers coming directly from a WMS catalog source the Metadata Link must be present in the WMS Layer GetCapabilities.

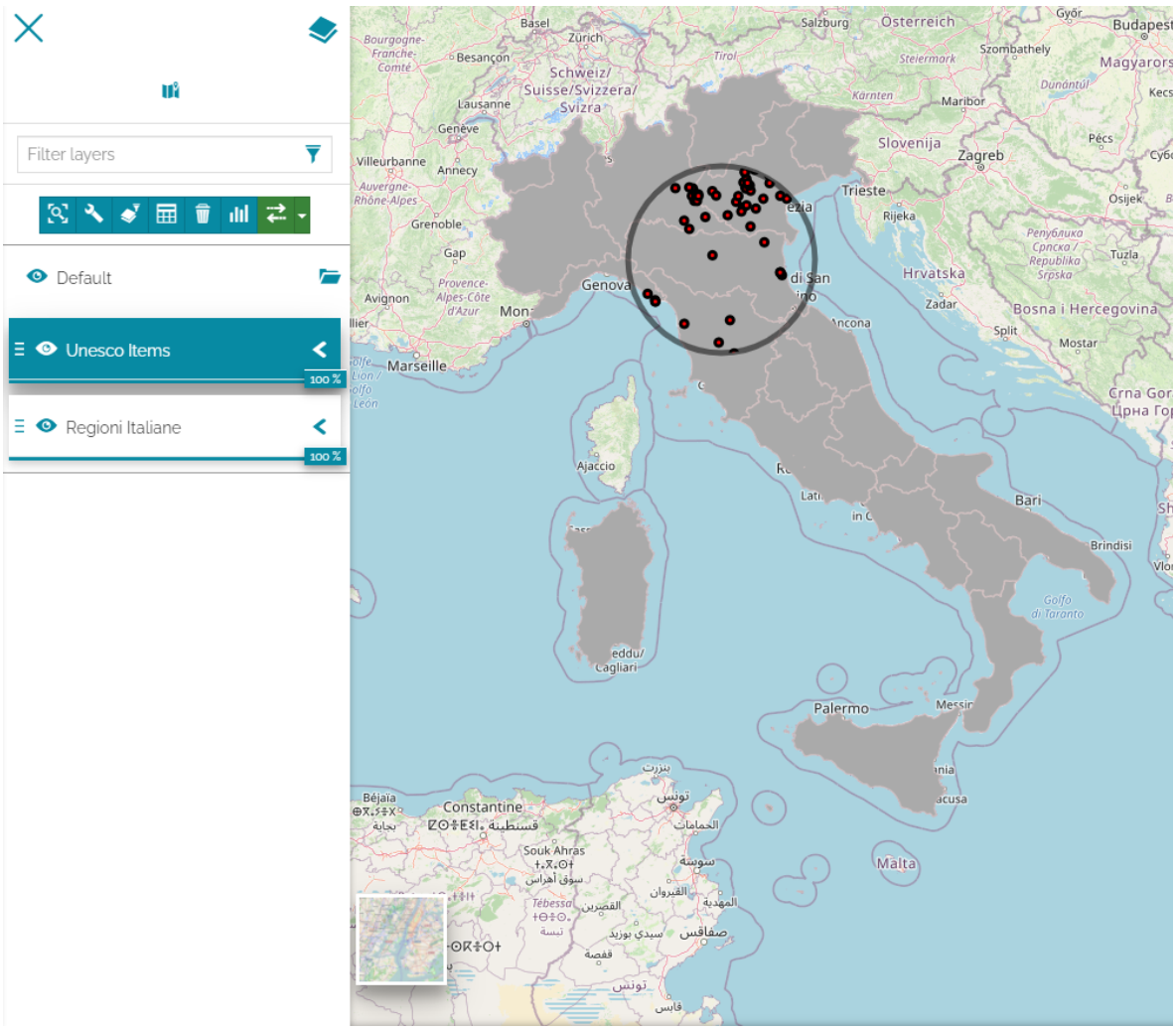
- Open the **Compare tool** where it is possible to *Swipe* or *Spy* the selected layer .

From the dropdown menu of the **Compare tool** it is possible to click on Swipe button so that the Swipe tool is enabled on the map for the selected layer: to activate the Swipe it is also possible to simply click on the **Compare tool** button.



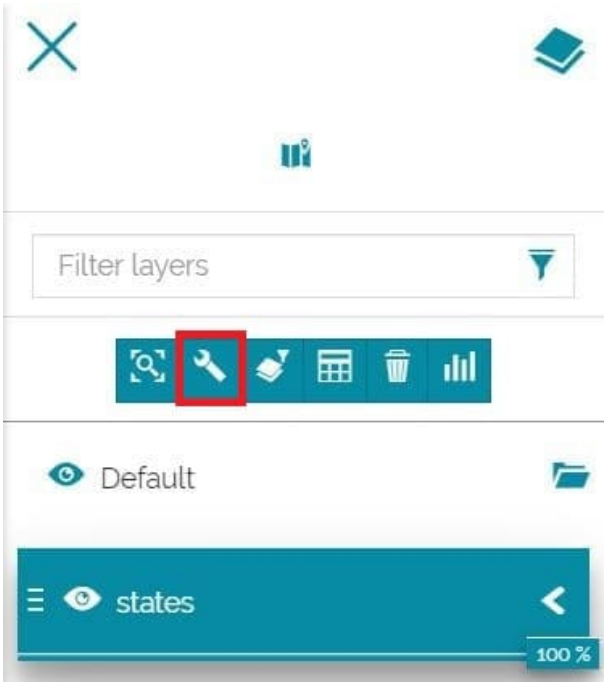
From the Compare tool dropdown , it is also possible to click on Configure . Doing this a configuration modal opens for the selected Compare tool (*Swipe* or *Spy glass*) so that, in case of Swipe, the user can change the orientation of the swipe from *Vertical* to *Horizontal*.

The user can also activate the Spy glass from the same dropdown menu in order to switch the Compare tool in **Spy glass** mode. If the *Spy glass* is active, clicking on the Configure option, the configuration modal opens so that it is possible to change the size of the spy glass (the `radius`).


Filter layers
Default
Unesco Items
100 %
Regioni Italiane
100 %
Basel
Zürich
Besançon
Schweiz/
Suisse/Svizzera/
Svizra
Lausanne
Genève
Villeurbanne
Annecy
Grenoble
Gap
Avignon
Marseille
Genova
Trieste
Rijeka
Slovenija
Zagreb
Hrvatska
Zadar
Bosna i Hercegovina
Split
Mostar
Ancona
Ajaccio
Bari
Brindisi
Palermo
Malta
Constantine
Souk Ahras
Tébessa
Béjaia
تونس
سوسة
المهدية
صفاقس
القيروان
قفصة
سيدي بوزيد
القصرين
الحمامات
Salzburg
Österreich
Szombathely
Maribor
Osijek
Split
Mostar
Bari
Brindisi
Palermo
Malta
Constantine
تونس

Layer Settings

In this section, you will learn how to manage the layer settings in terms of general information, display mode, style and feature Info.
Since a layer is added to the TOC it is possible to access its settings with the dedicated button  that appears selecting a layer:



The layer settings panel is composed of four sections:

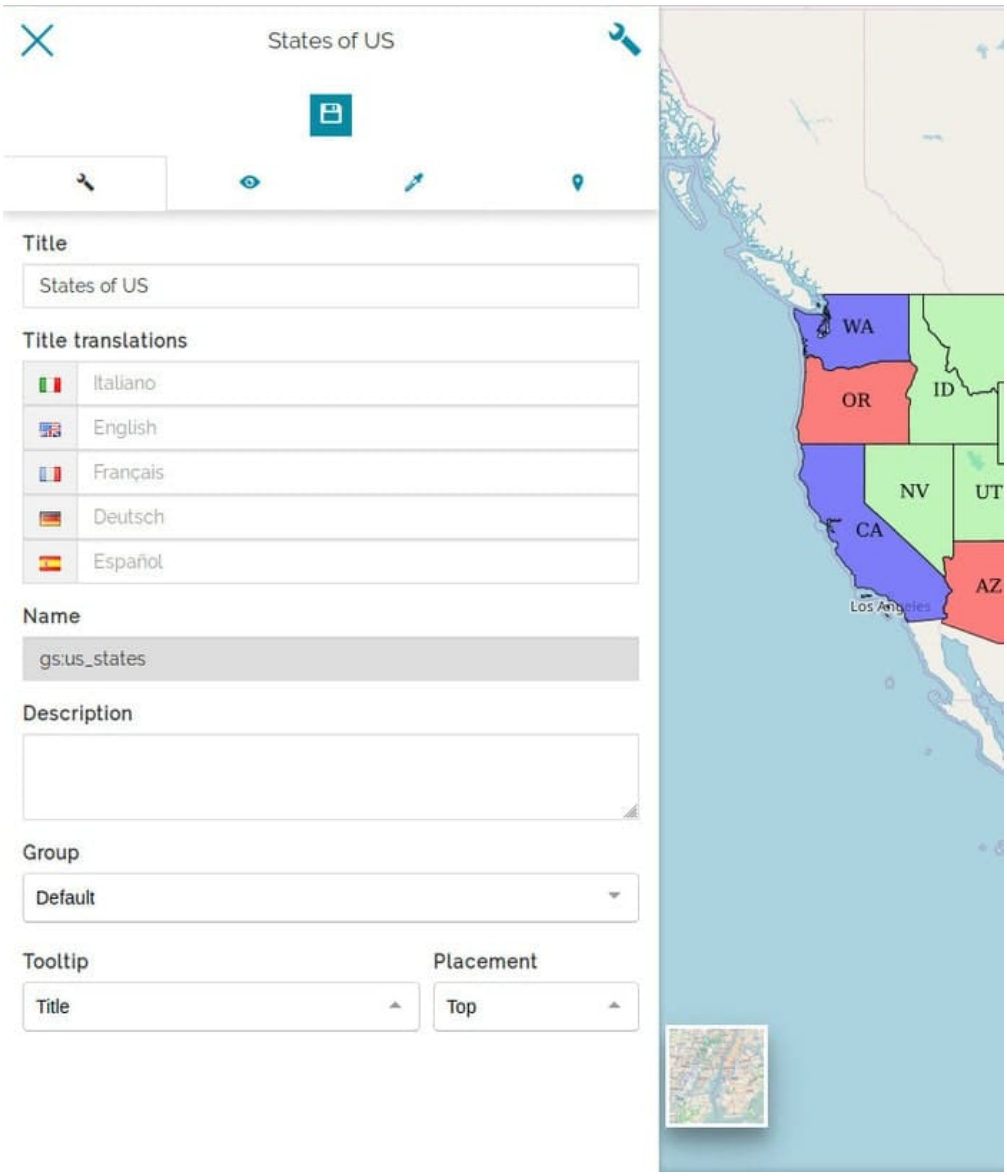
- General information
- Display
- Style

- Feature Info

> **Warning**
>
> For WMTS layers the Style and the Feature Info sections are not implemented. Moreover the Display section is limited to the Transparency level parameter.

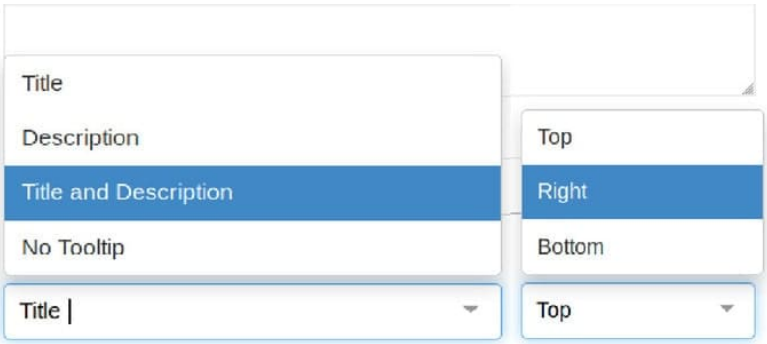
General information

By default, as soon as the user opens the layer settings panel the General information section appears:

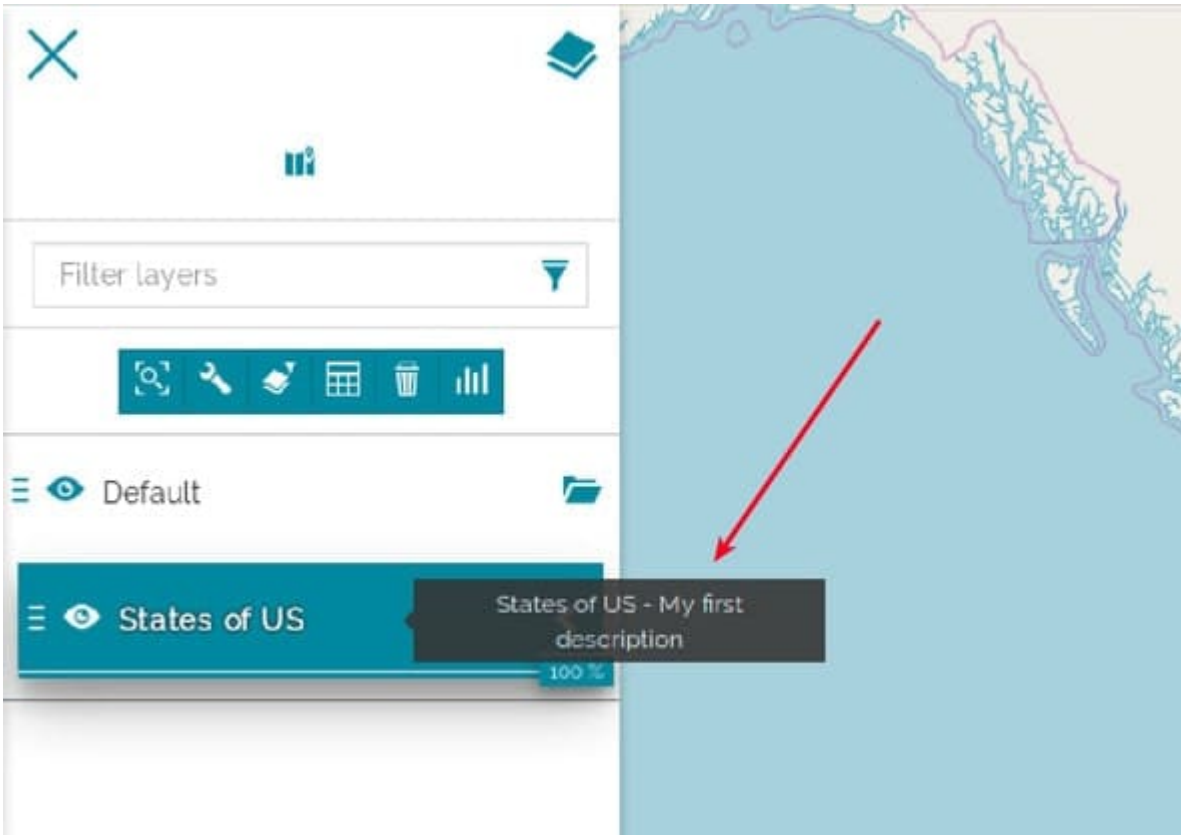


In this page it is possible to:

- Change the *Title*
- Set the *Title translations*, that will be switched by changing the language
- Take a look at the *Name* of the layer
- Edit the layer's *Description*
- Set the layer *Group*
- Configure the *Tooltip* that appears moving the cursor over the layer's item in TOC. In this case the user can decide that the *Title*, the *Description*, both or nothing will be displayed. Moreover you can set the *Placement* of the tooltip, choosing between *Top*, *Right* or *Bottom*:

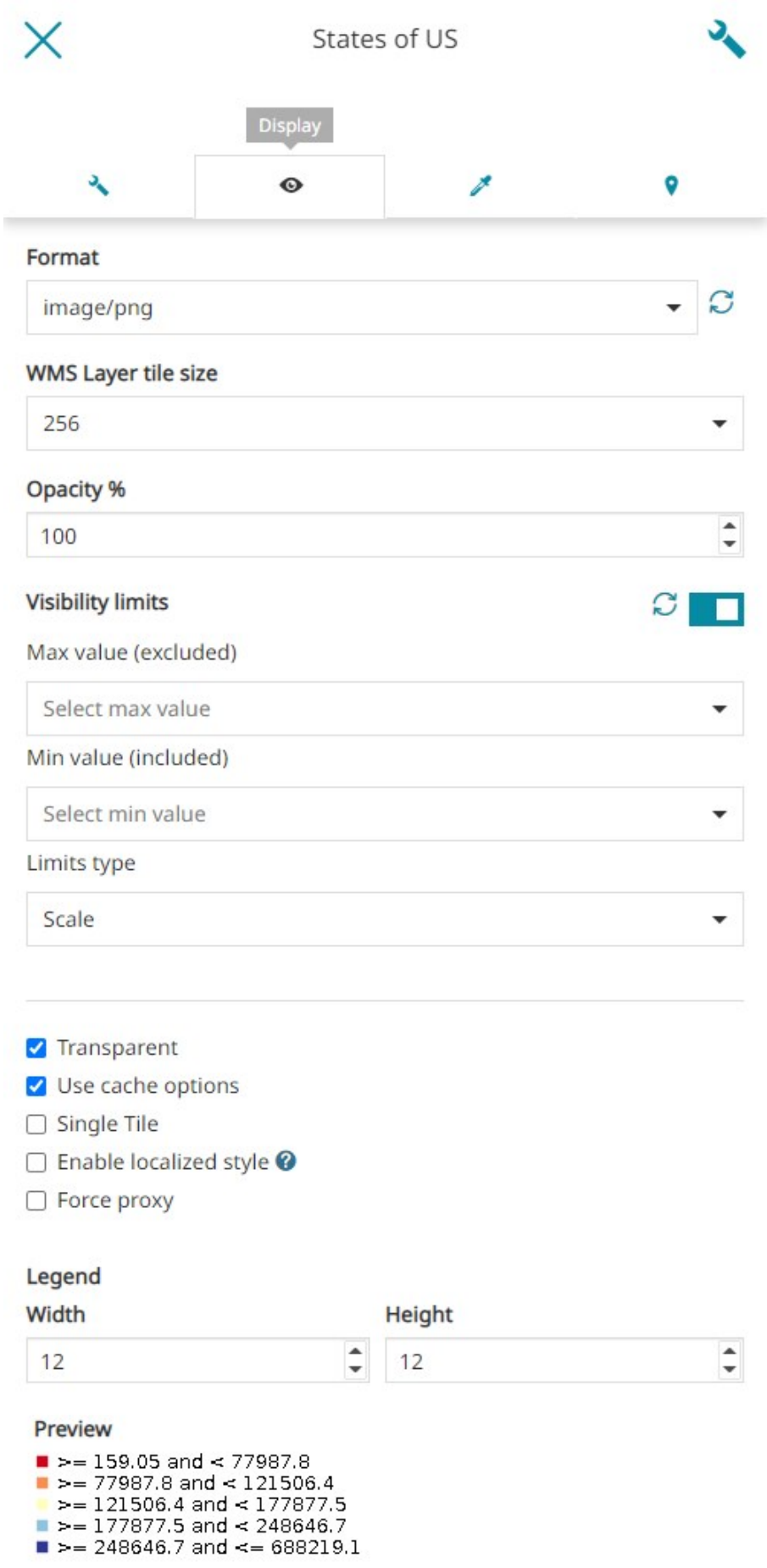


Setting a tooltip that shows the Title and the Description on the Right, for example, it can be similar to the following:



Display

Through the second section of the layer settings panel it is possible to change the display settings:


States of US
Display
Format
image/png
WMS Layer tile size
256
Opacity %
100
Visibility limits
Max value (excluded)
Select max value
Min value (included)
Select min value
Limits type
Scale
Transparent
Use cache options
Single Tile
Enable localized style
Force proxy
Legend
Width
Height
12
12
Preview
>= 159.05 and < 77987.8
>= 77987.8 and < 121506.4
>= 121506.4 and < 177877.5
>= 177877.5 and < 248646.7
>= 248646.7 and <= 688219.1

In particular, the user is allowed to:

- Set the image format: choosing between `png`, `png8`, `jpeg`, `vnd.jpeg-png`, `vnd.jpeg-png8` and `gif`

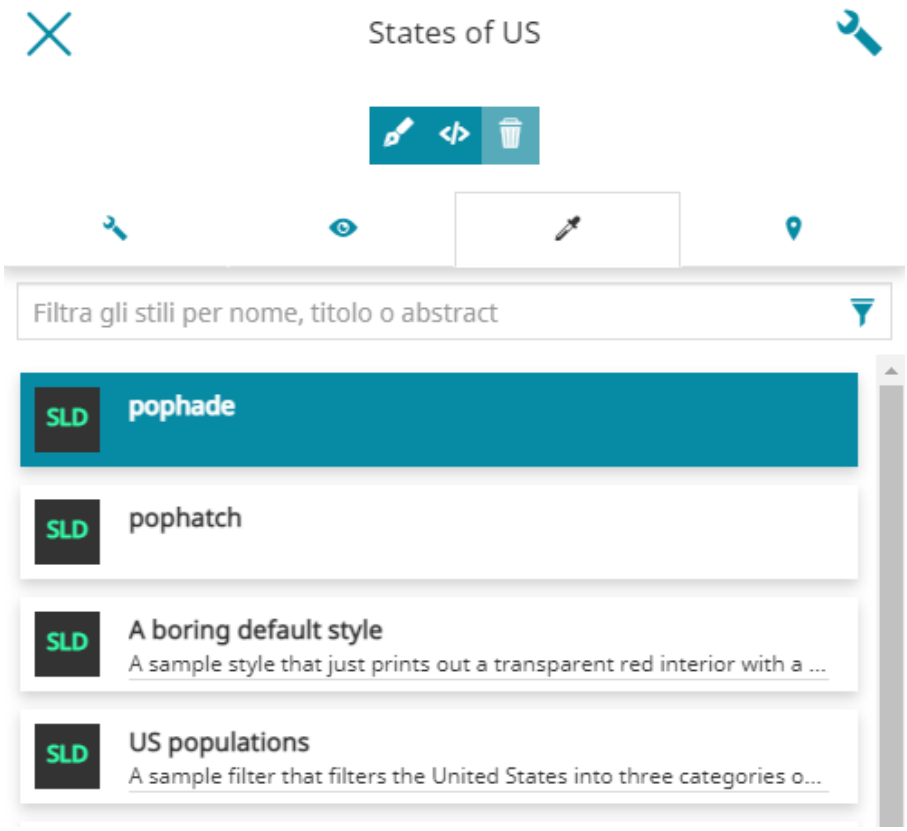
> **Note**
>
> The list of available format is the same of the related catalog source. Therefore, for WMS services, the updated list of formats supported by the WMS server is used.

- Set the size of layer tiles: choosing between `256` or `512`
- Set the opacity value of the layer (in %)
- Enable/disable the **Visibility limits** to display the layer only within certain scale limits. The user is allowed to request the `MinScaleDenominator` and `MaxScaleDenominator` value present on the *WMS GetCapabilities* of the layer though the  button or set the *Max value* and the *Min value* and select the *Limits type* choosing between `Scale` or `Resolution`.
- Enable/disable the transparency for that layer
- Enable/disable the use of the layer cached tiles (if checked, the *Tiled=true* URL parameter will be added to the WMS request and to use tiles cached with GeoWebCache)
- Decide to display the image as a single tile or as multiple tiles
- Enable/disable the localized style. If enabled allows to include the MapStore's locale in each **GetMap**, **GetLegendGraphic** and **GetFeatureInfo** requests to the server, as explained in the WMS Catalog Settings
- Enable/disable the *Force proxy* layer option. If enabled, forces the application to check the source and applies proxy if needed.
- Set the layer *Legend* with custom *Width* and *Height* options. Both of these field values if greater than the default legend's size of 12, then the custom values gets applied on the legend width and height display property
- A preview of the legend is shown with the applied custom values from Legend fields above.

> **Warning**
>
> The *Format* and *Layer tile size* options are available only for the layers added from CSW and WMS catalog sources.

Style

The third section, dedicated to the layer style, displays like the following:



In this case the user is allowed to:

- Search through the available layer styles and select the desired one
- Create a new style
- Edit an existing style
- Delete an existing style

Note

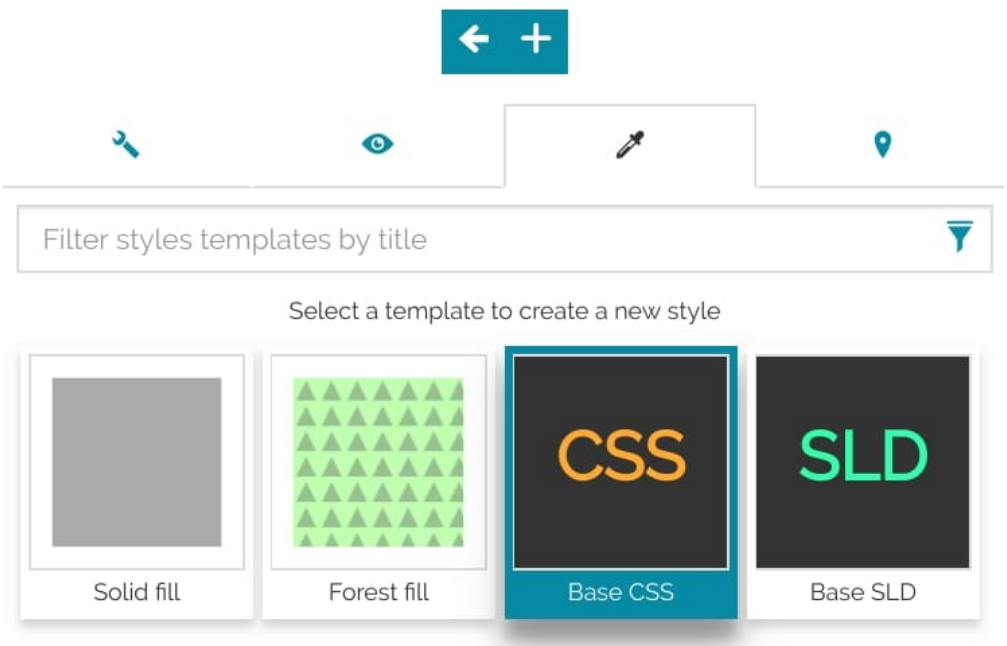
By the default service security rules the GeoServer's REST APIs are available only for the GeoServer administrators, so a basic authentication form will appears in MapStore to enter the *Admin* credentials. Without Admin rights, the editing capabilities on styles are not available and only the list of available styles will appear to allow the user to select one of them to the layer.

Take a look at the User Integration with GeoServer section of Developer Guide in order to understand how to configure the way MapStore and GeoServer share users, groups and roles. If the users integration between GeoServer and MapStore is configured, the editing functionalities of the styles will be available according to the role of the authenticated user in MapStore in a more transparent way.

Create a new style

It is possible to create a new style with a click on the button. At this stage the user can choose between different types of template from which the customization will start:

- *CSS - Cascading Style Sheet* (a language used for describing the presentation of a document written in a markup language like the HTML)
- *SLD - Styled Layer Descriptor* (an XML schema specified by the Open Geospatial Consortium OGC for describing the appearance of map layers)



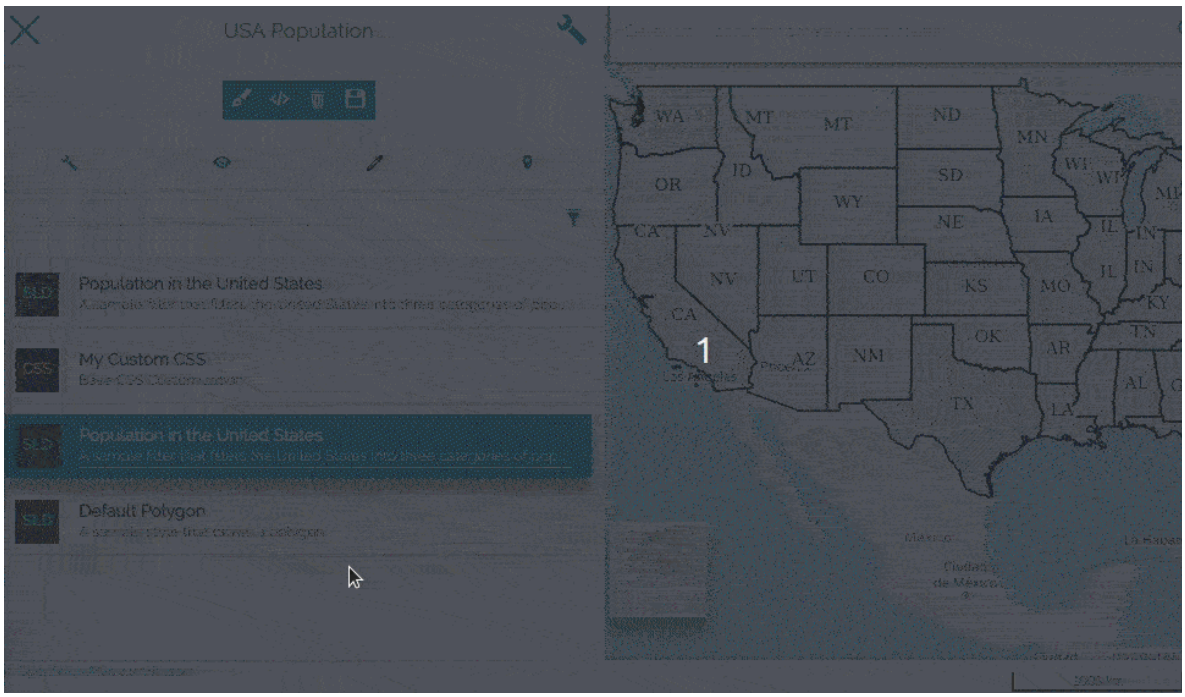
> **Note**
>
> The availability of the style formats depends, firstly, from the GeoServer. MapStore, by default, will add all the supported format that the server provides. To edit or create styles using the CSS format the CSS extension must be installed in GeoServer

Once the new style is chosen, with a click on the + button the following window opens:

Here the user can set the *Title* and the *Abstract* (optional), and through the **Save** button the new style will be automatically added to the styles list.

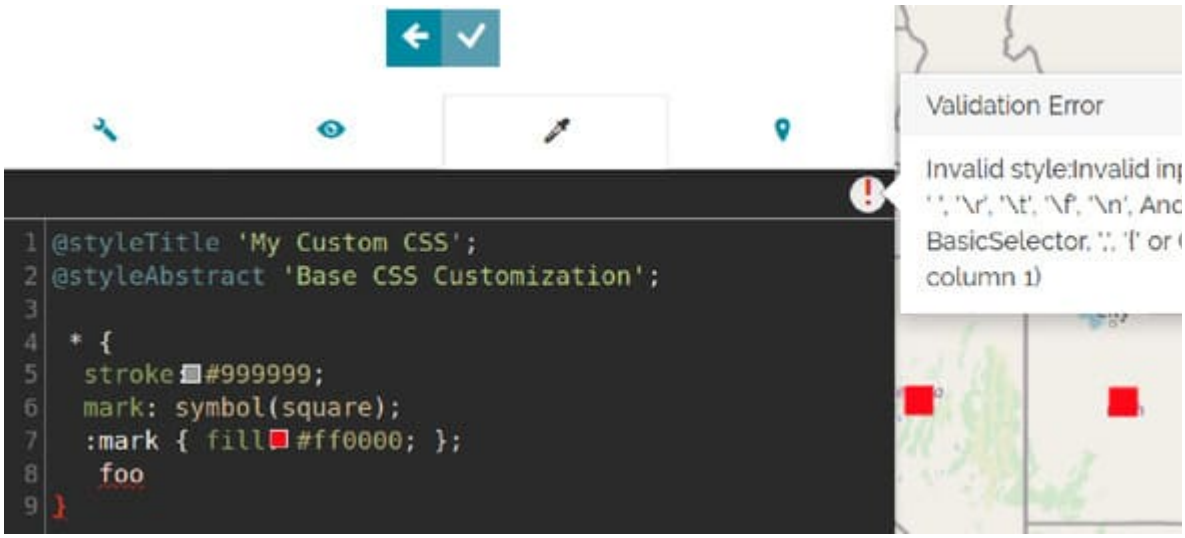
Edit an existing style

Existing styles can be edited clicking on the </> button. The page that opens allows the user to customize the style in the related format:

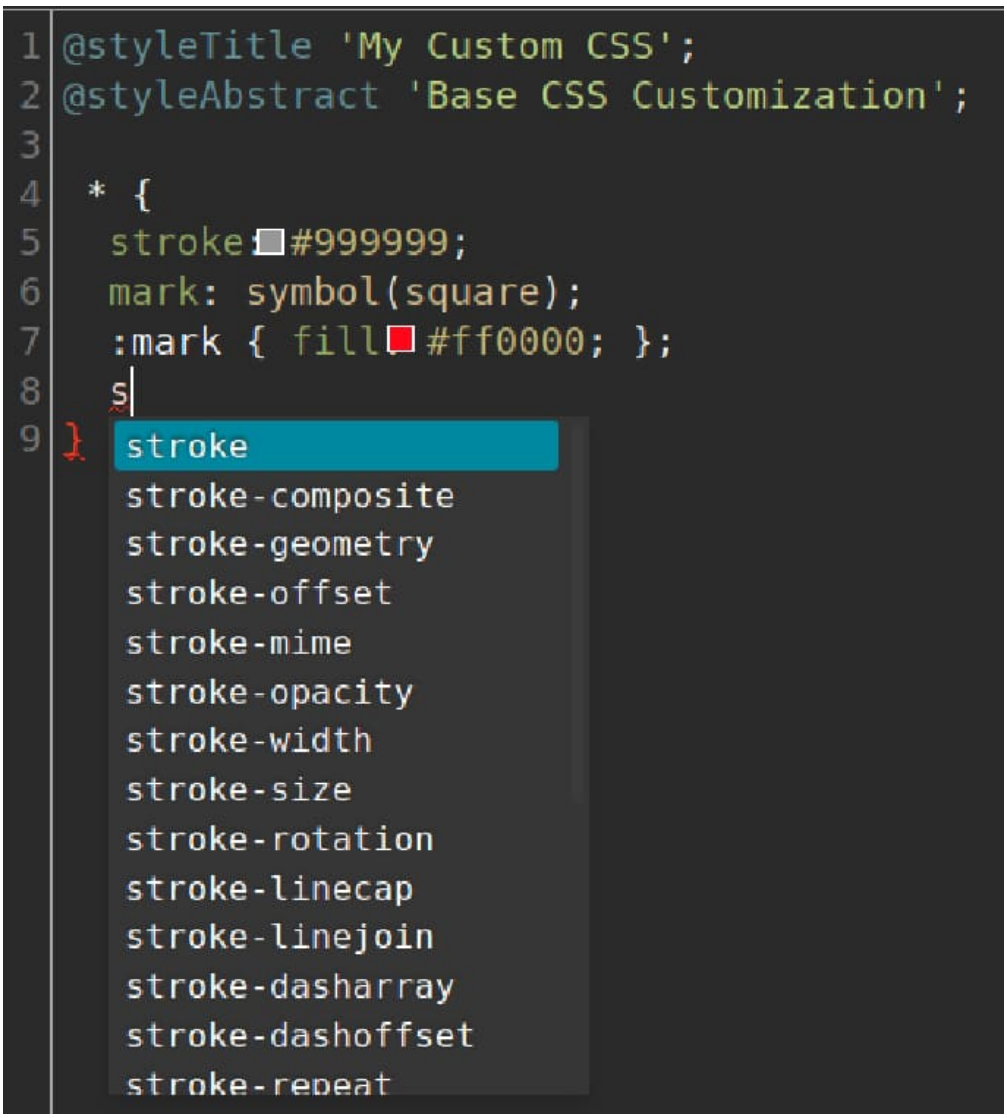


The editor is easy to approach thanks also to the following functions:

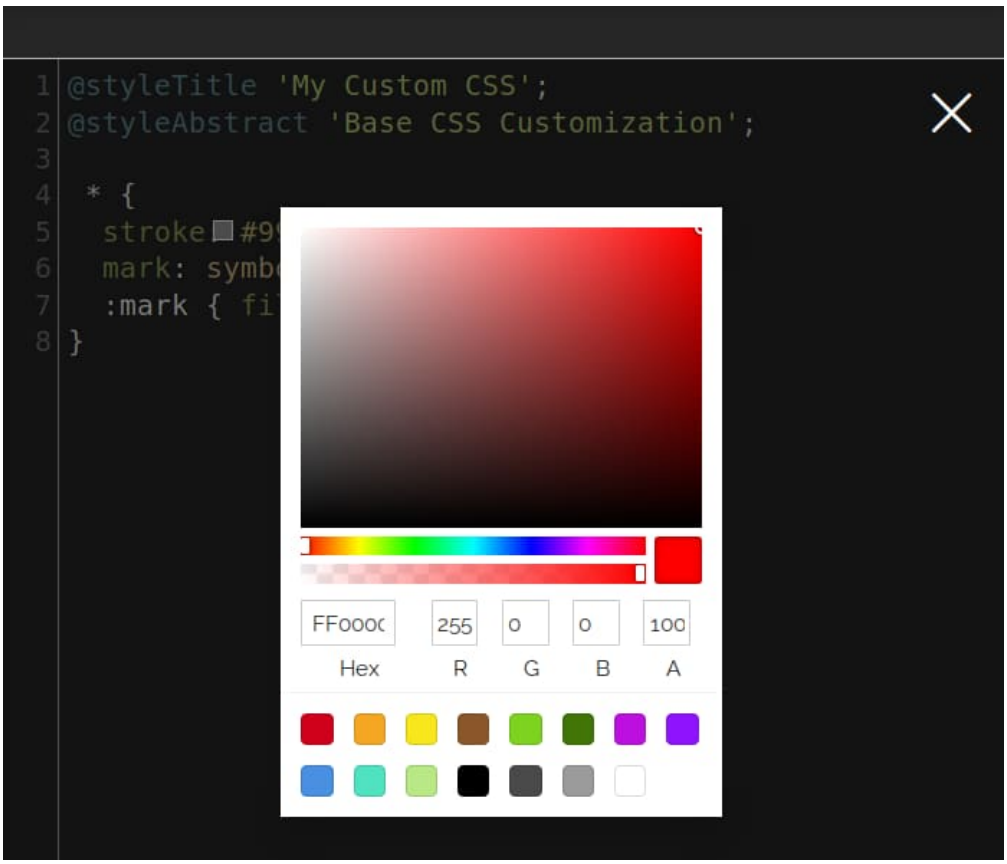
- The *sintax control* highlights any possible error with a red underline (if error are detected an icon with a red exclamation point ⓘ will be shown in the top-right side of the editor)



- The *autocomplete* function suggests the possible style's properties in order to prevents syntax errors:



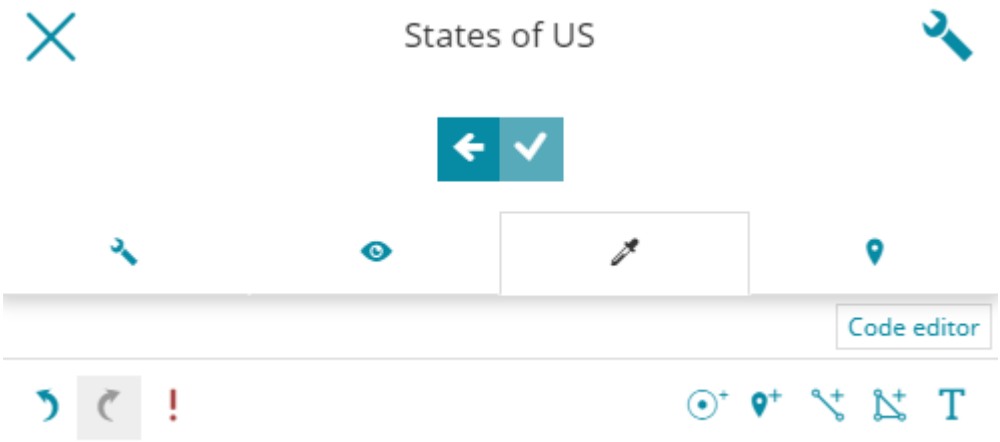
- The *color picker*, that can be activated through the square filled icon (▀) near the color code, helps in choosing colors directly from the editor, showing an interface like the following:



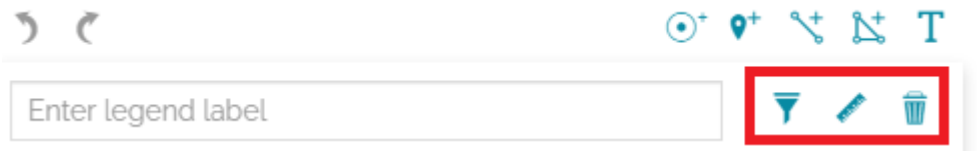
> **Warning**
>
> The *autocomplete* and the *color picker* functions are available only in the CSS editor.

Visual Editor Style

MapStore also allows to edit the layers style using a *Visual editor* with a most user friendly UI.Clicking on the Visual editor button a section opens so that the user can customize the style through with a visual style editor by adding/editing symbolizers, which can be: *Mark*, *Icon*, *Line*, *Fill* and *Text*. It is anyway possible to switch to the text editor mode if necessary for a more complex styling.



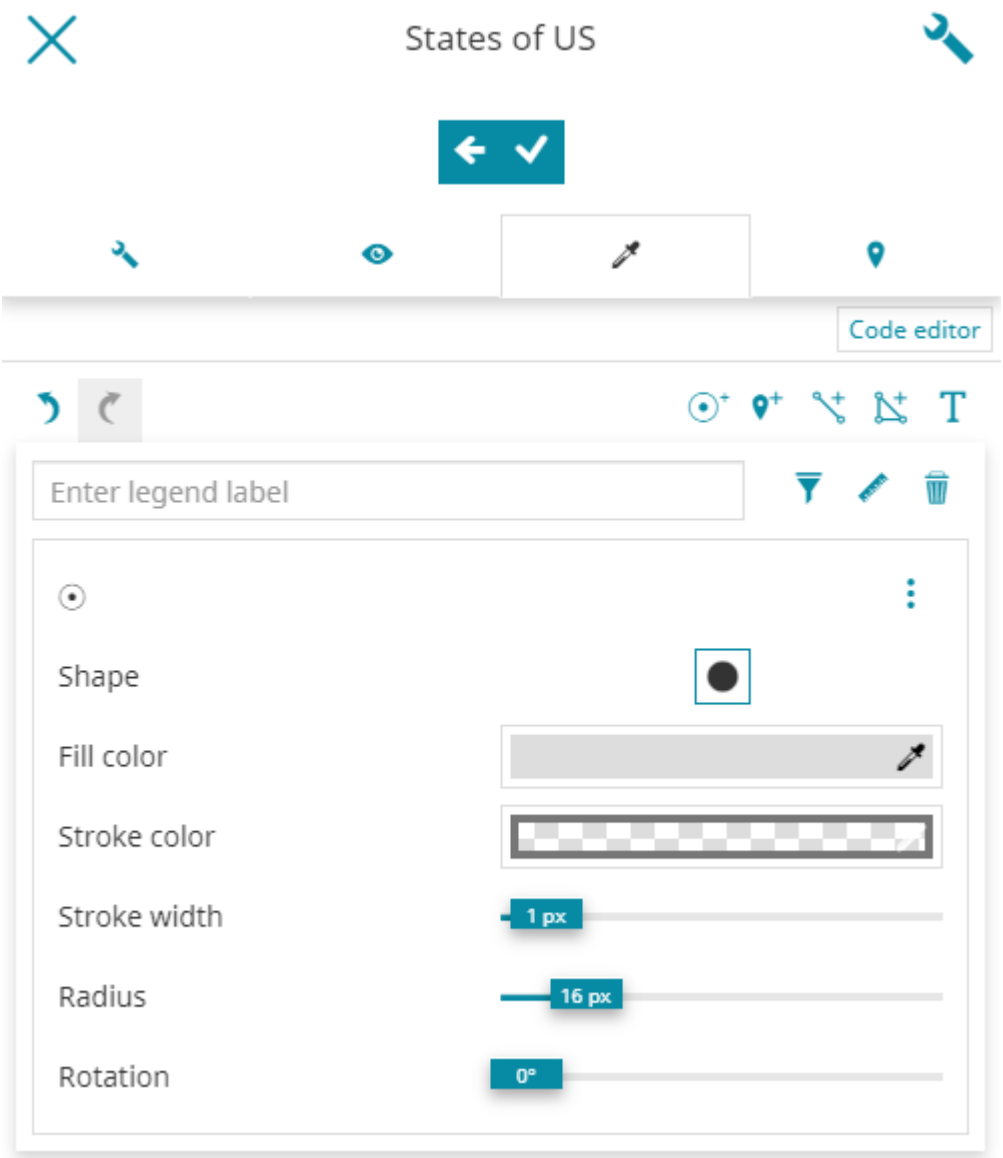
Once a symbolizer has been added and customized, you can:



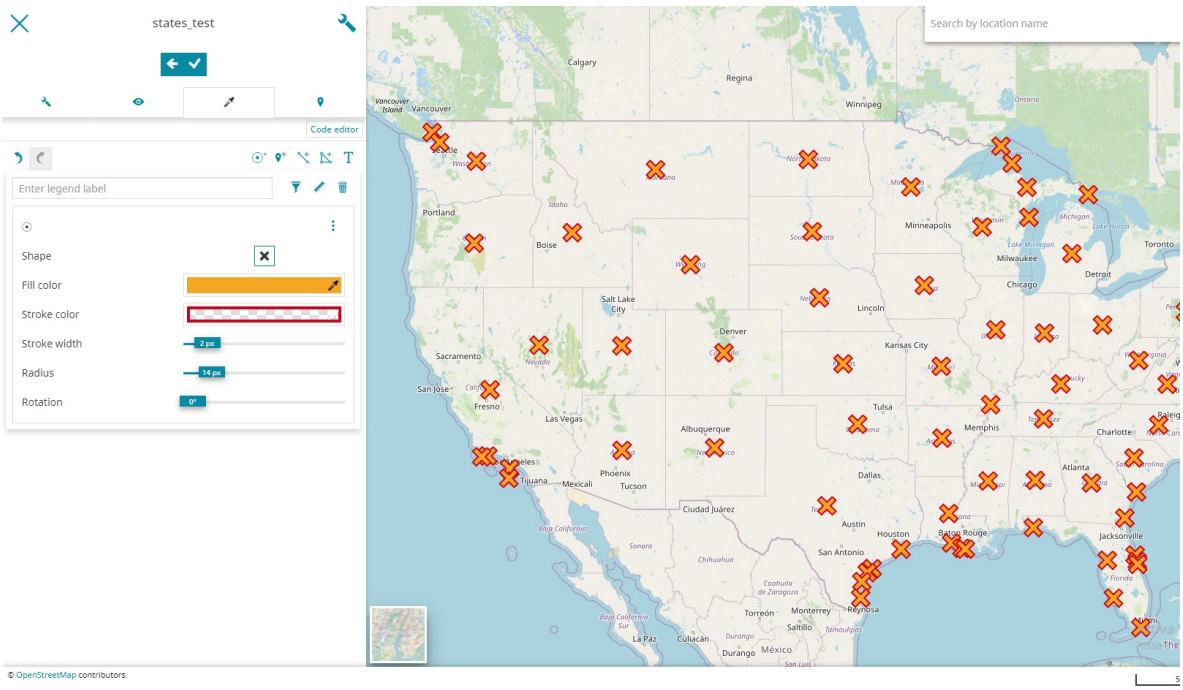
- **Filter** the style rule, as explained here, in order to apply the style only to certain layer features. It is possible clicking on the button.
- Add a **Scale denominator filter** (`max` and `min` scale) to visualize the style rule only within certain scale limits. This is possible by clicking the button.
- **Remove** the symbolizer by clicking the button.

Mark

The mark type allows you to add a mark to the layer: clicking on the button a mark panel appears:

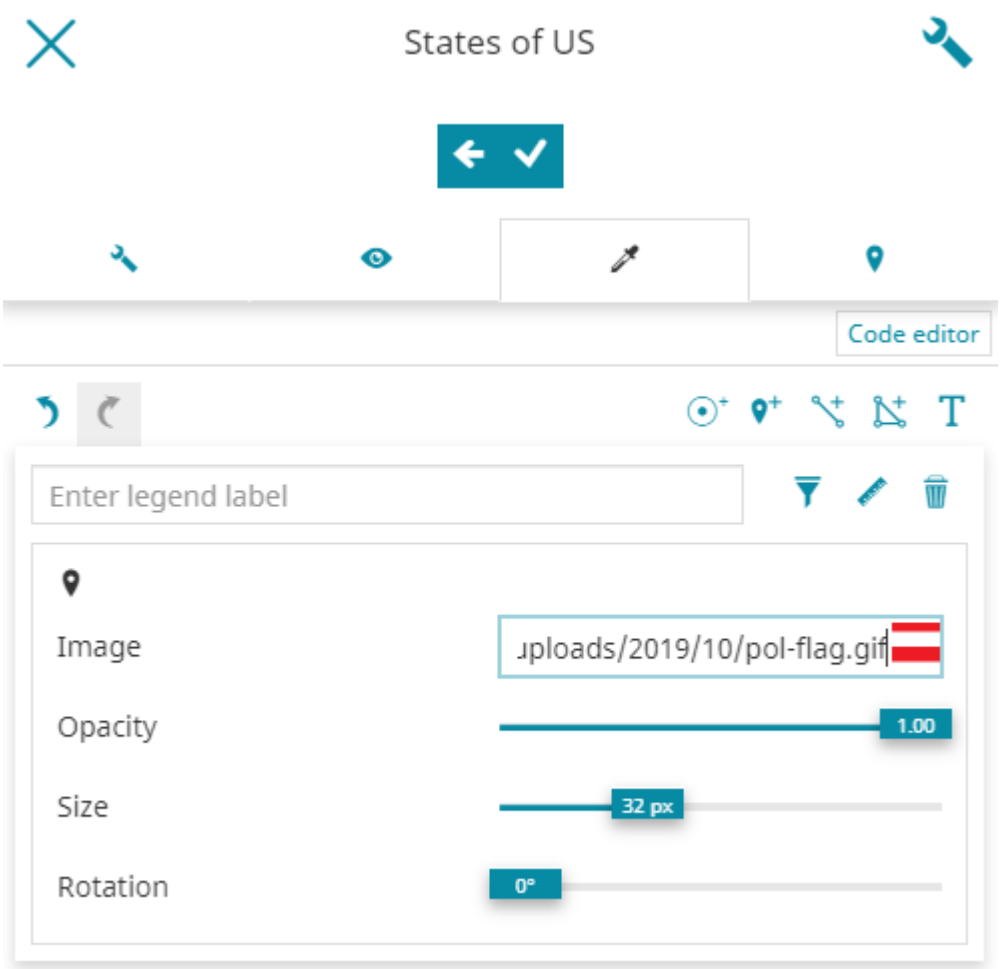


The mark can have different `Shape`, `Color`, `Stroke` with different `Color` and `Width` and customizable `Radius` and `Rotation`. Take a look at the following example.



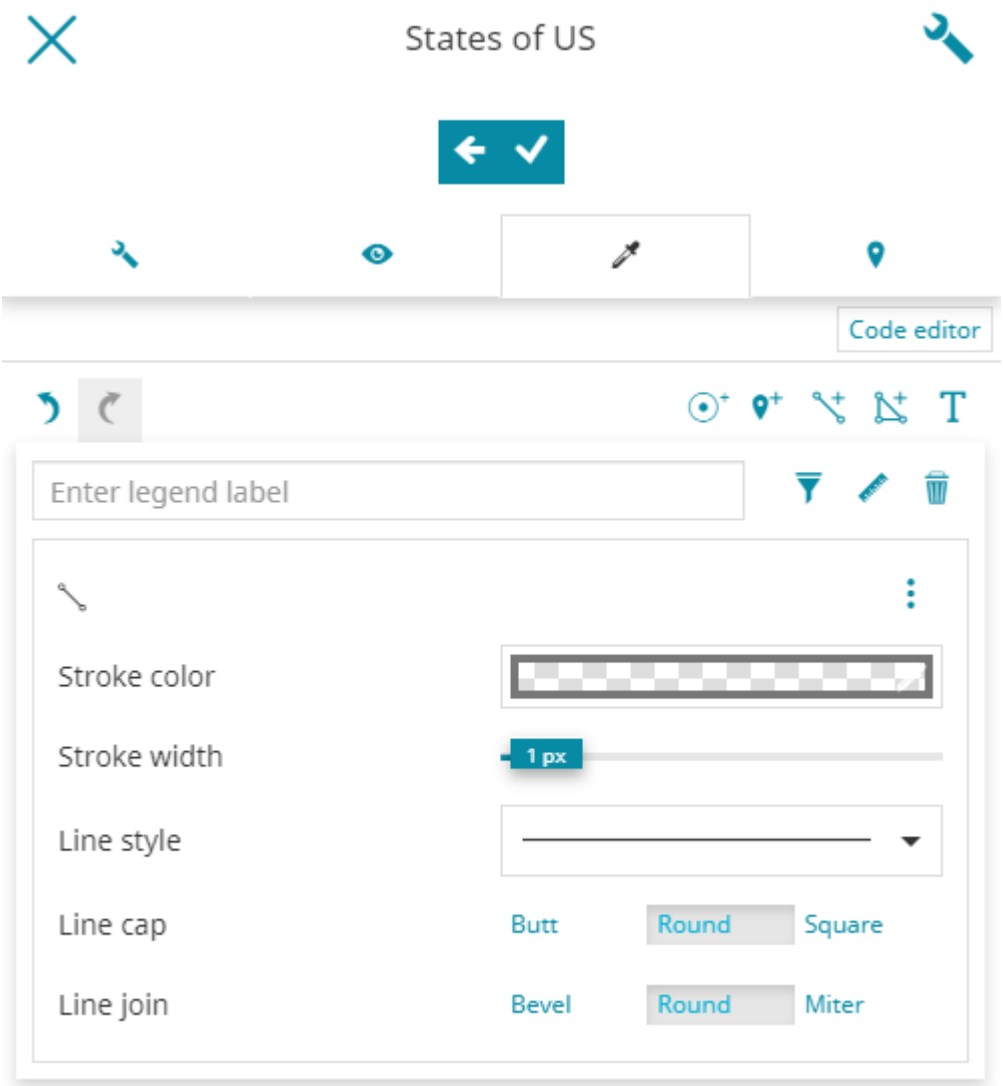
Icon

With the icon panel, which opens by clicking on button, the style editor is allowed to add an image as an icon (by specifying its *URL*) and customize the icon `Opacity`, `Size` and `Rotation` angle:

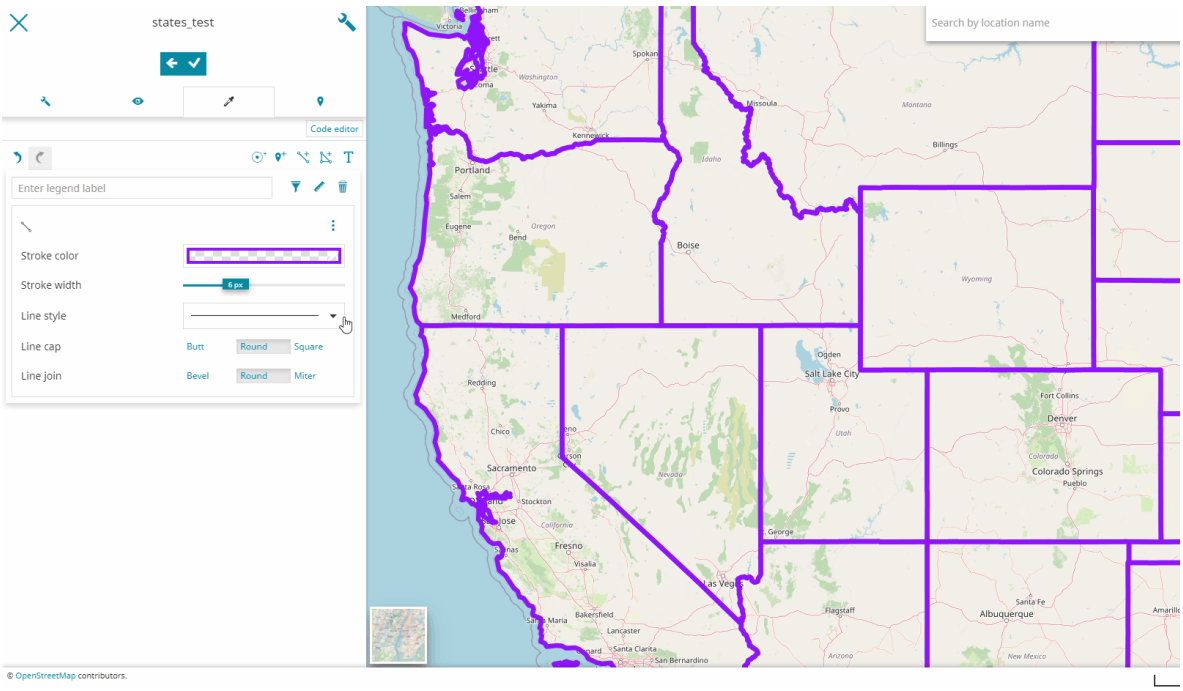


Line

The line rule is used to style linear features of the layer: clicking on the button a panel allows the user to edit the corresponding properties.

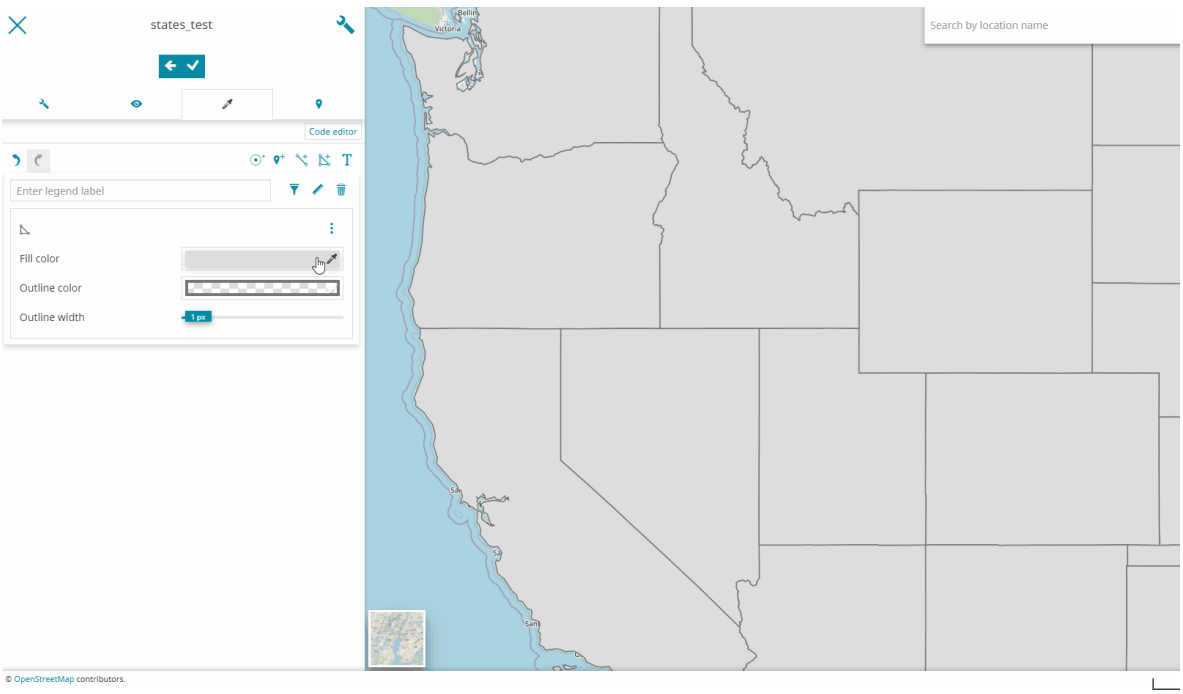


The editor can change the `Stroke color`, the `Stroke width`, the `Line style` (*continuous*, *dashed*, etc), the `Line cap` (*Butt*, *Round*, *Square*) and the `Line join` (*Bevel*, *Round*, *Miter*). An example can be the following one:



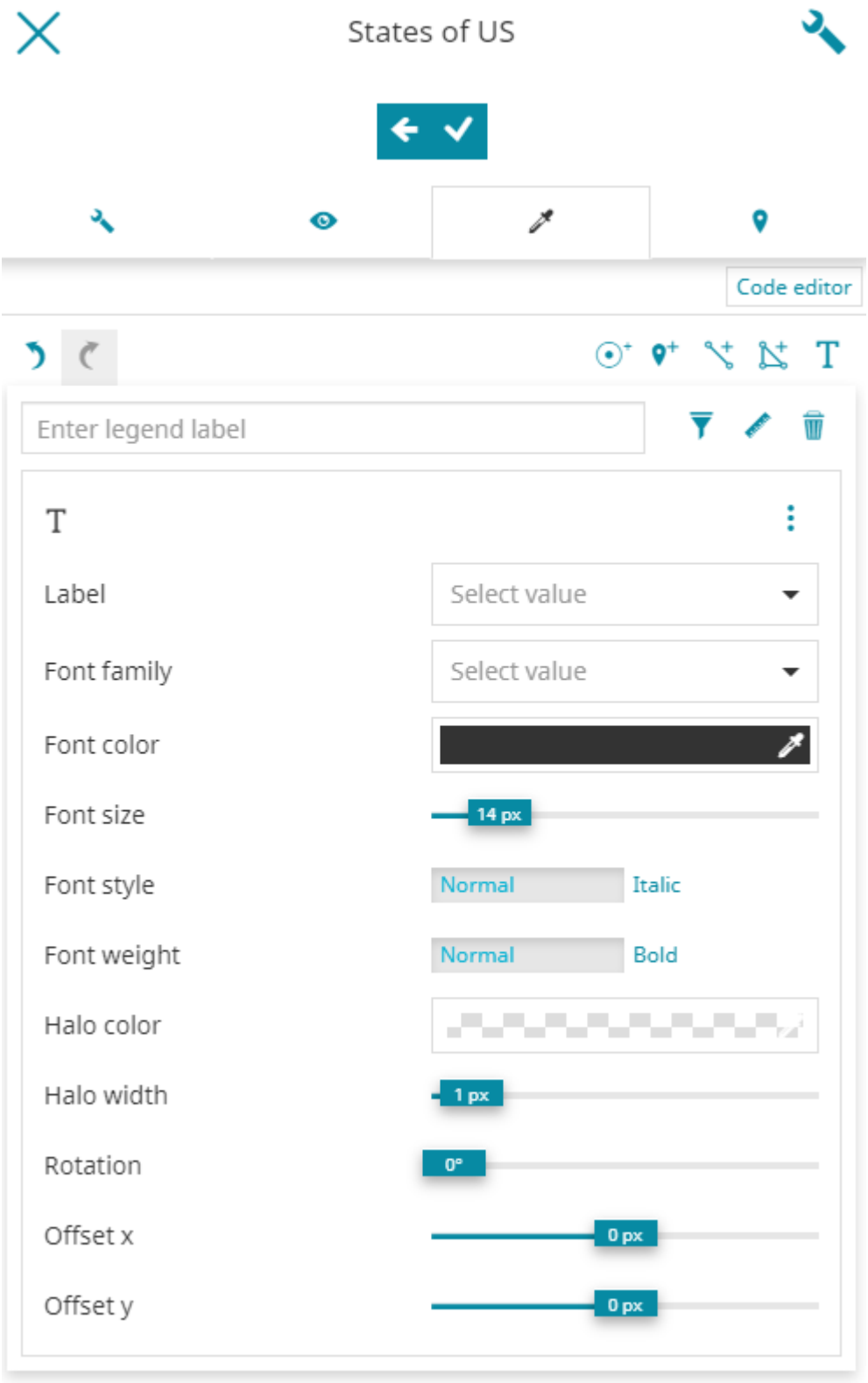
Fill

The Fill rule is used to style polygon features. Clicking on  button, the editor is allowed to customize the `Fill color`, the `Outline color` and the `Outline width`:



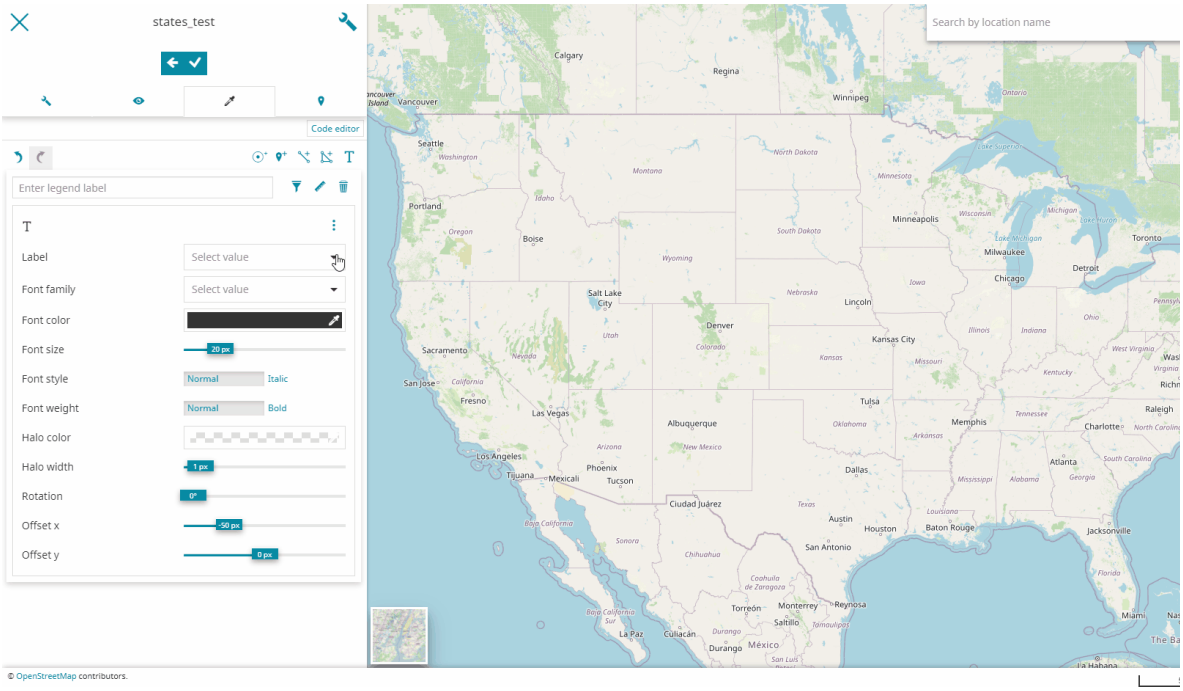
Text

The Text rule is used to style features as text labels. Text labels are positioned either at points or along linear paths derived from the geometry being labelled. Clicking on the  button a specific panel opens:



The editor is allowed to type the name of the layer attribute to use for the `Label` and the dropdown list is filtered accordingly to show the existing attributes that are matching the entered text (the user can anyway directly select an attribute from the list). Moreover, the style editor can customize the `Font Family` (*DejaVu*

Sans, *Serif*, etc), choose the font `Color`, `Size`, `Style` (*Normal* or *Italic*) and `Halo weight` (*Normal* or *Bold*) and select the desired `Halo color` and `Halo weight`. It is also possible to choose the text `Rotation` and `Offset` (*x* and *y*). En example can be the following one



Style Methods

Different styles methods can be used for each style rule. Clicking on the ⋮ button, available on top of the panel of each symbolizer, the editor can choose one of the following depending on the rule type:

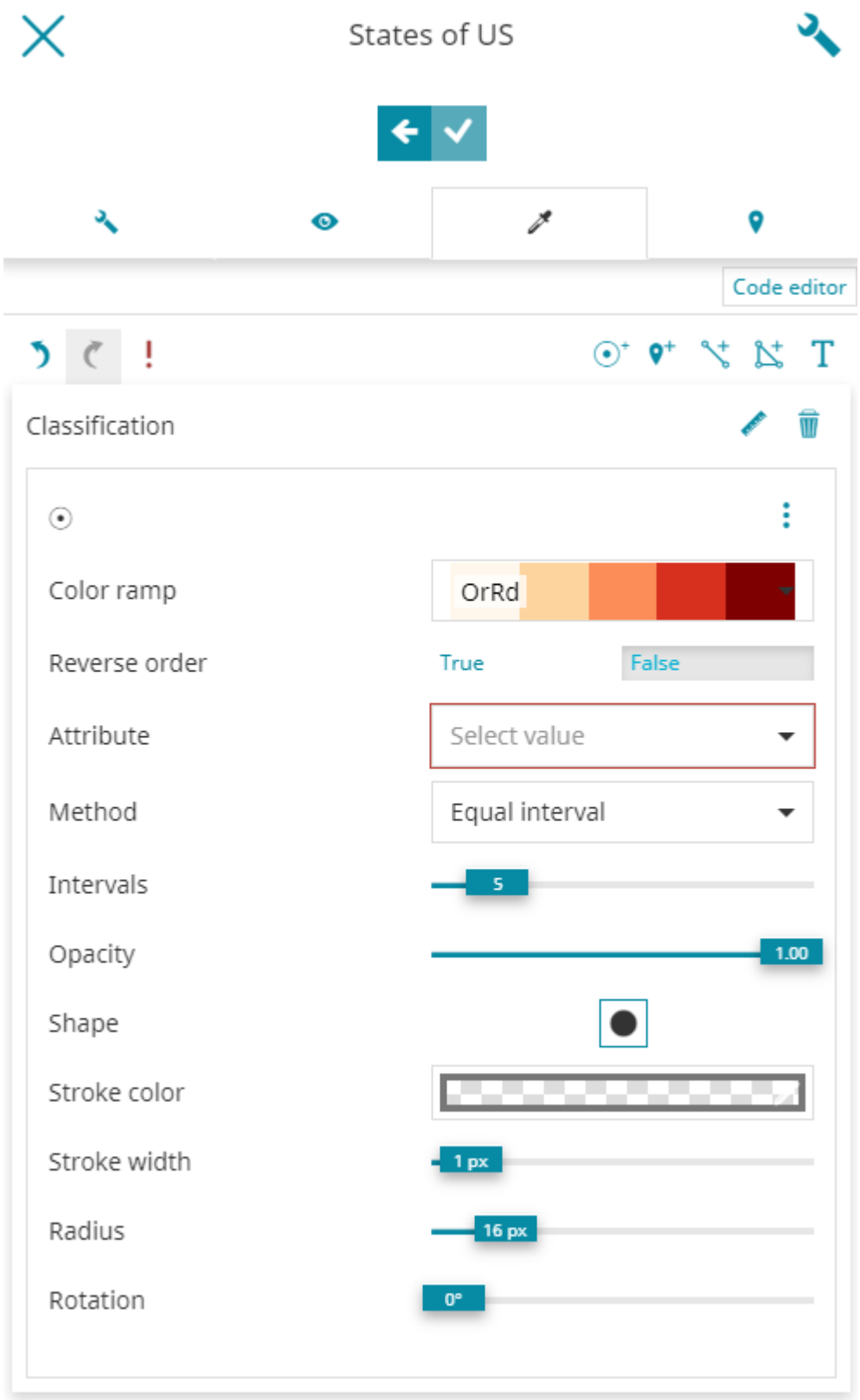
- *Simple style*
- *Classification style*
- *Pattern mark style* (available only for rules of type Line and Fill)
- *Patter icon style* (available only for rules of type Line and Fill)

Simple style

The Simple style is the default style described above for each symbolizer.

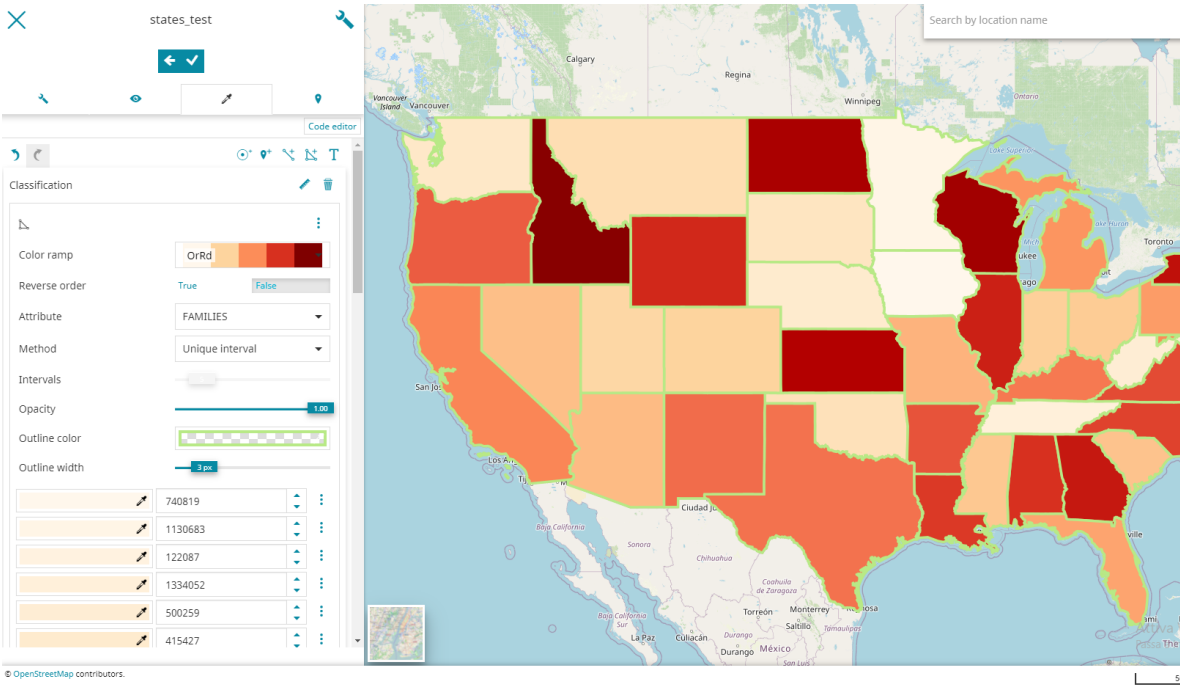
Classification style

MapStore allows you to classify the style based on the attributes of the layer. The *Classification style* is available for *Marker*, *Line*, *Fill* and *Text* by clicking on the ⋮ button and choosing the **Classification style** options from the dropdown menu.



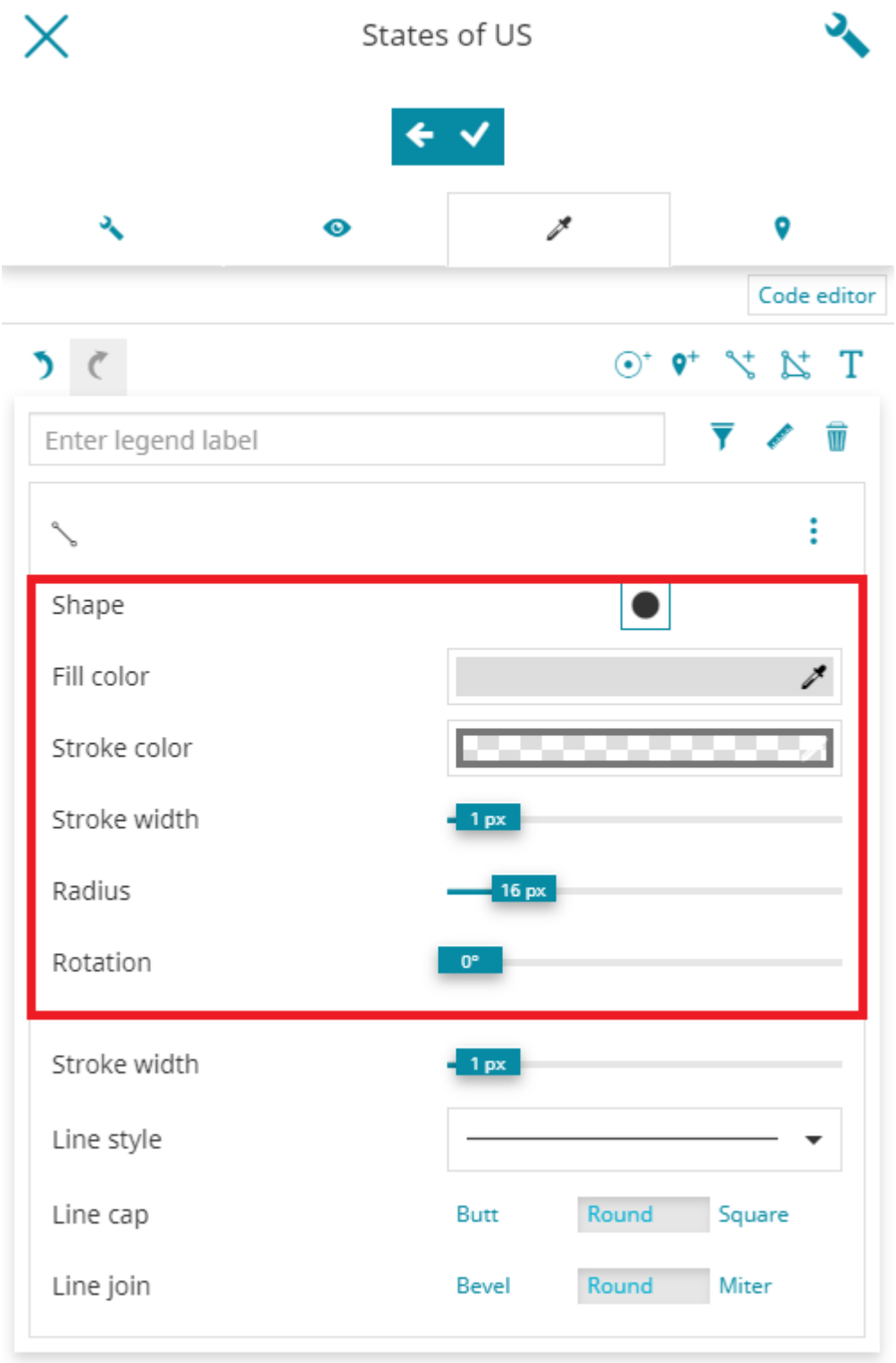
It this case the editor is allowed to choose a `Color ramp` and the order (with `Reverse order`) of the classification intervals colors. It is obviously possible to select the layer `Attribute` to use for the classification along with the

classification `Method` (*Quantile*, *Equal interval*, *Natural breaks* and *Standard deviation*), the number of classification `Intervals` and the `Opacity` (%) of each interval range. An example of the *Classification style* for a *Fill* rule type can be the following one:

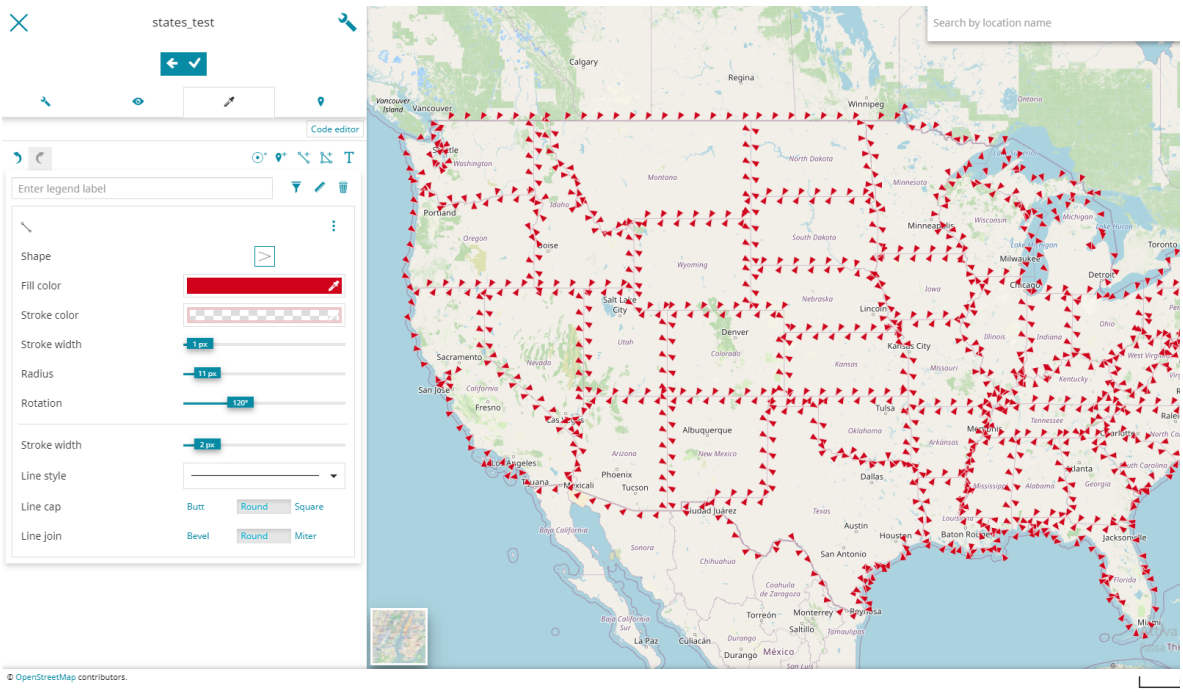


Pattern mark style

With the *Pattern mark style* it is possible to represent *Line* or *Fill* style rules with a mark by clicking on the ⋮ button and choosing the **Pattern mark style** options from the dropdown menu.

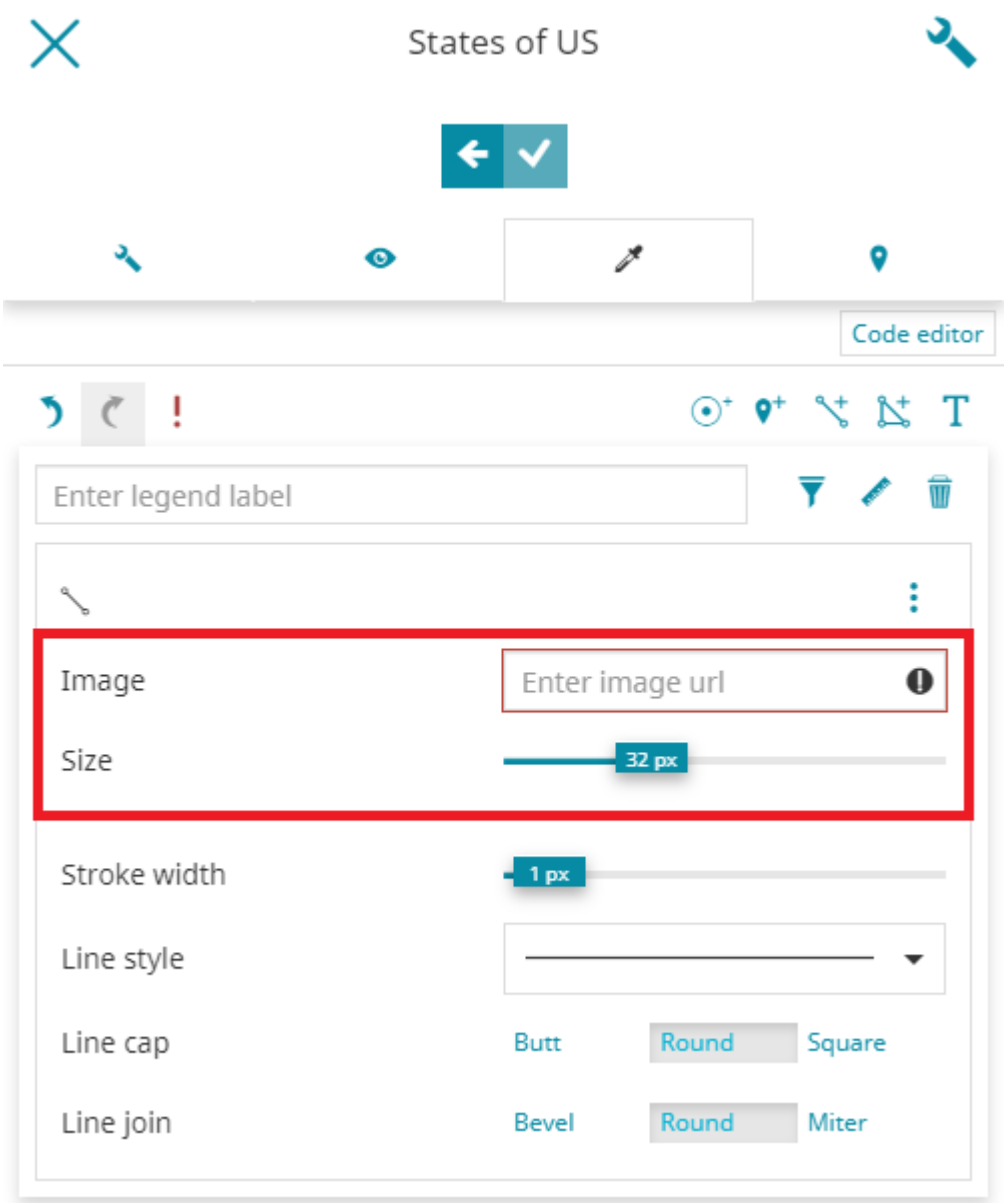


The style editor can configure a *Mark* as explained here along with the usual options available for rules of type line or fill depending on the selected symbolizer. Take a look at the following example of the *Pattern mark style* for the *Line* rule sample.

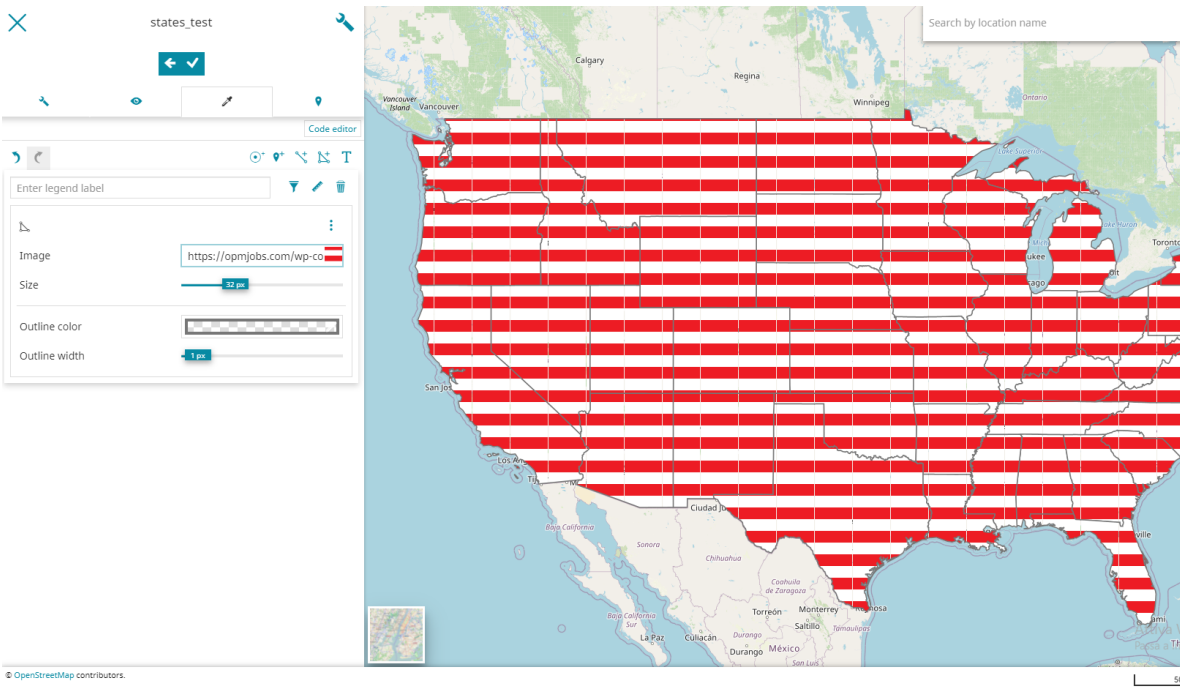


Patter icon style

With the *Pattern icon style* it is possible to represent *Line* or *Fill* style rules with an icon by clicking on the ⋮ button and choosing the **Pattern icon style** options from the dropdown menu.



The style editor can configure the *Icon* as explained here along with the usual options available for rules of type line or fill depending on the selected symbolizer. Take a look at the following example of *Pattern icon style* for a *Fill* rule sample.

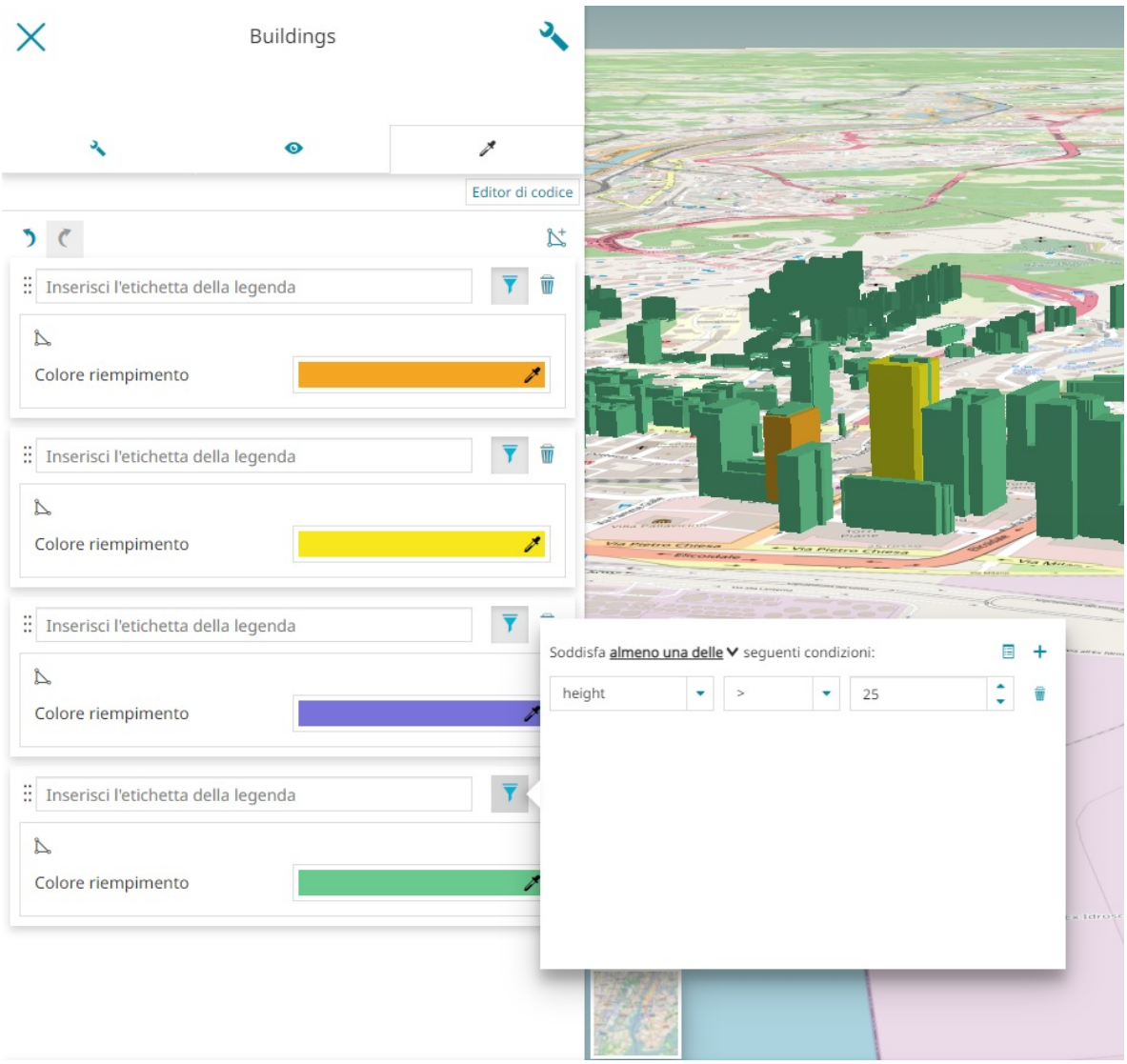


Styling on the 3D navigation

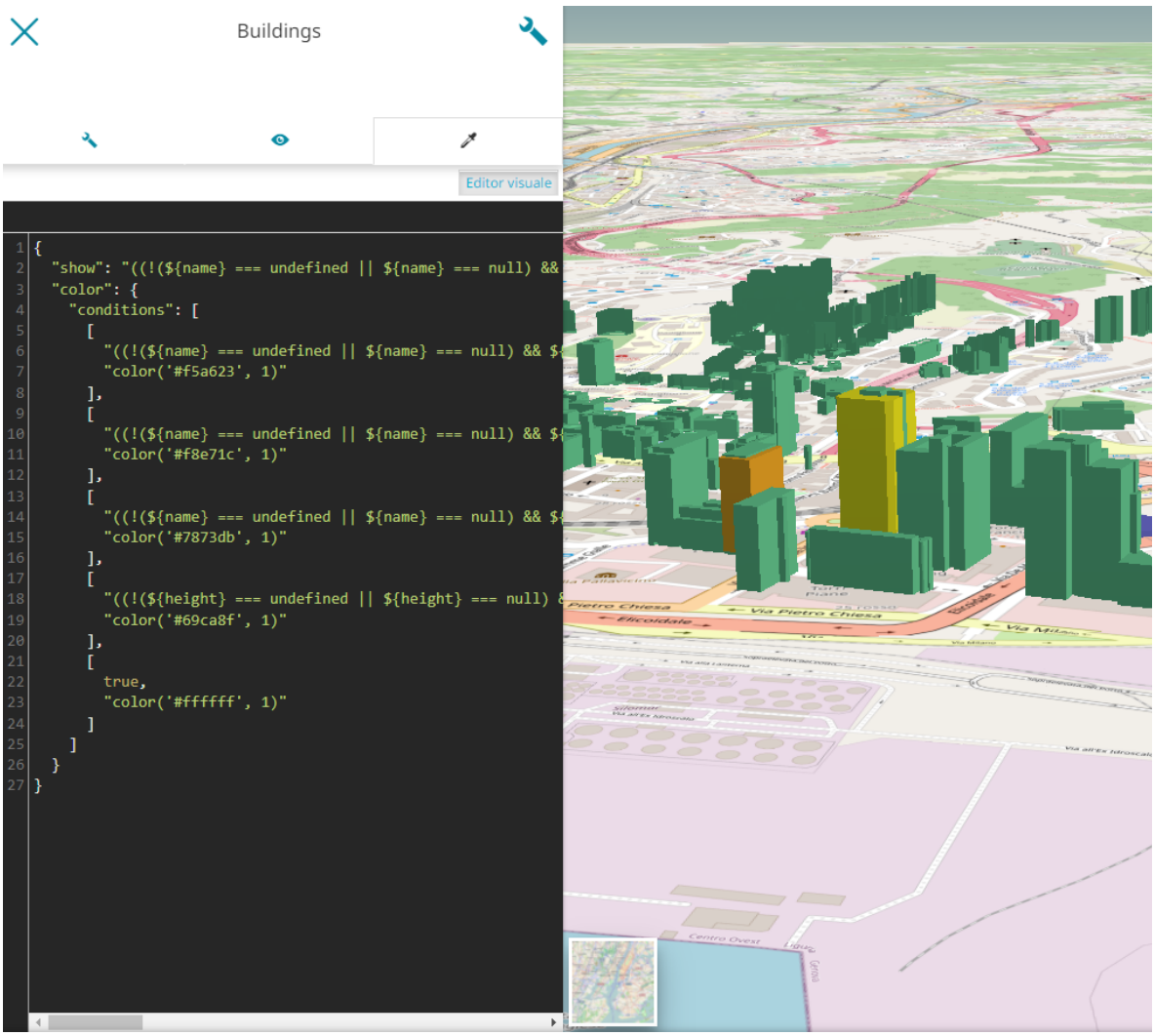
Thanks to the new improvements made to the *Visual Style Editor* editor, when 3D Navigation is enabled, the editor has the ability to customize the style of **3D Tiles** and **vector layers**.

Styling of 3D Tiles layer

With MapStore it is possible to customize the style of a 3D Tiles layer client side. The MapStore support is working in respect of the 3D Tiles Specification 1.0 and on top of the Cesium Styling capabilities. Below is an example of how the Style Editor of a 3D Tiles layer is appearing in the MapStore UI.

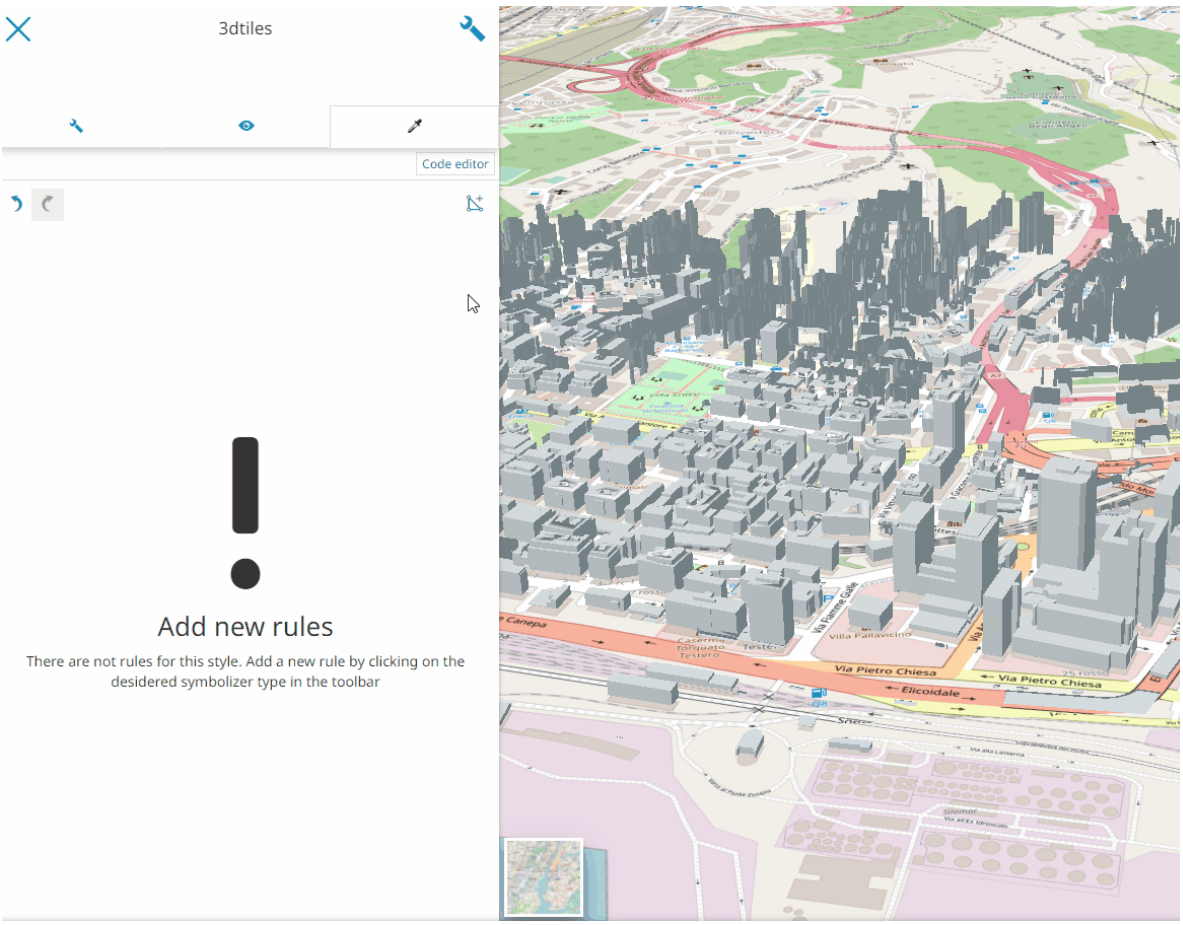


For the 3D Tiles styling, while with the **Code Text Editor** it is possible to leverage completely on the styling specifications:

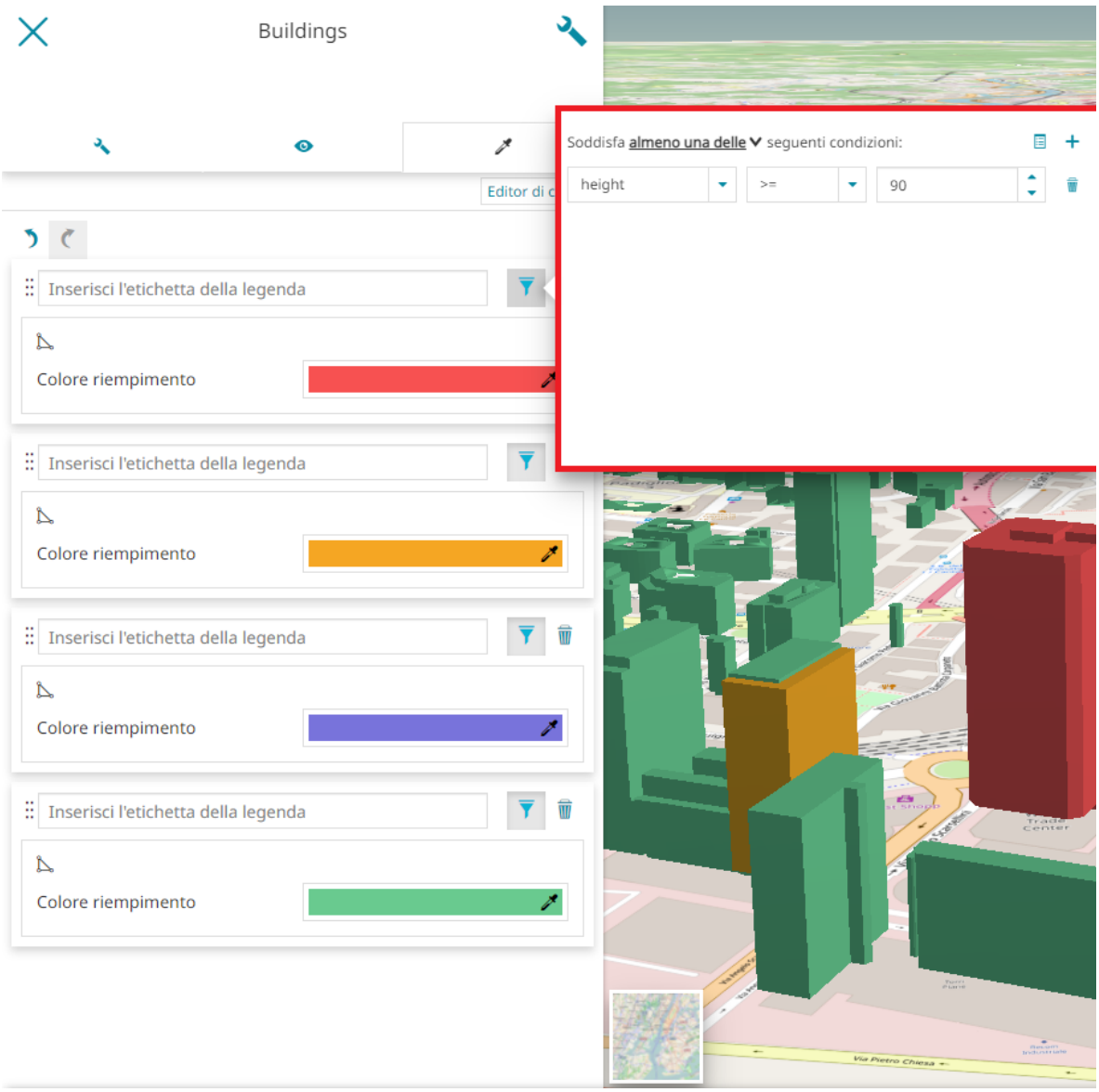


The MapStore **Visual Style Editor** supports for now only a limited set of capabilities:

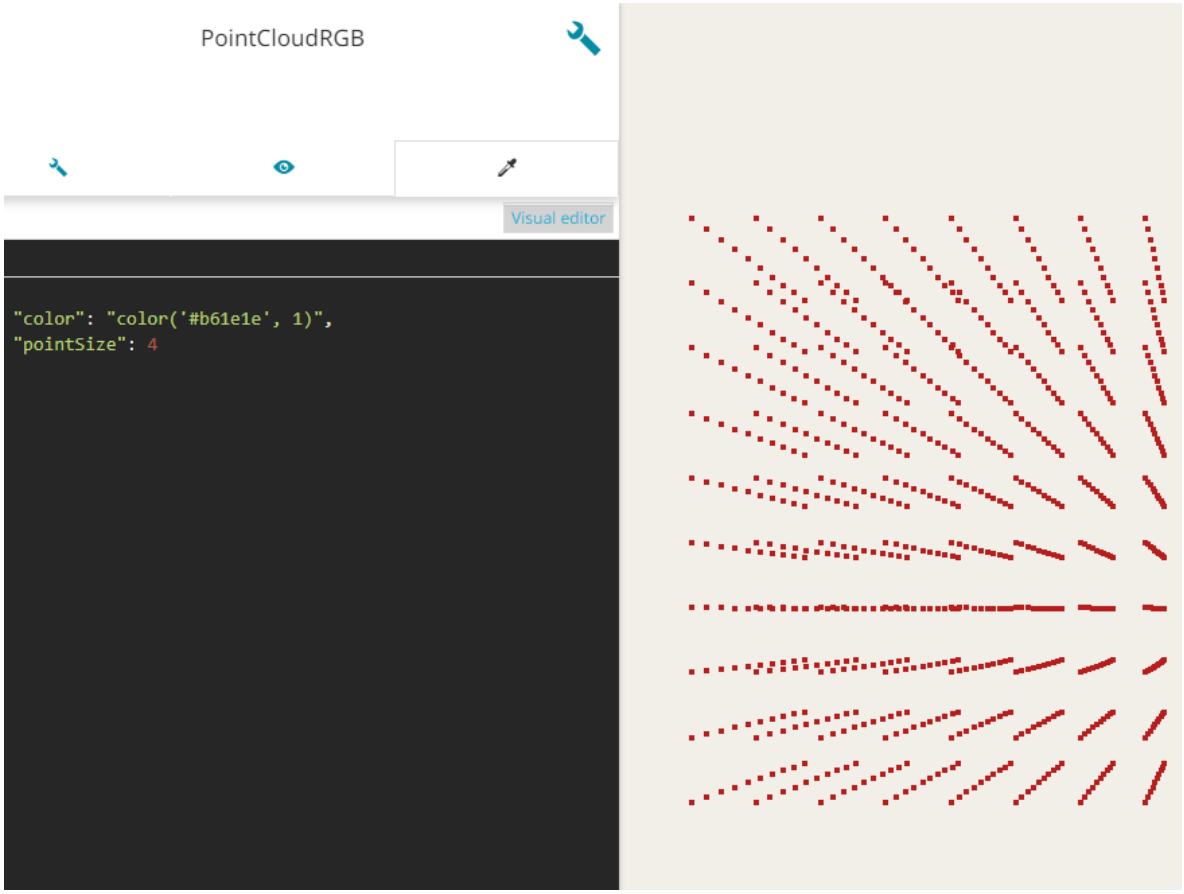
- Customization of the **Fill color**

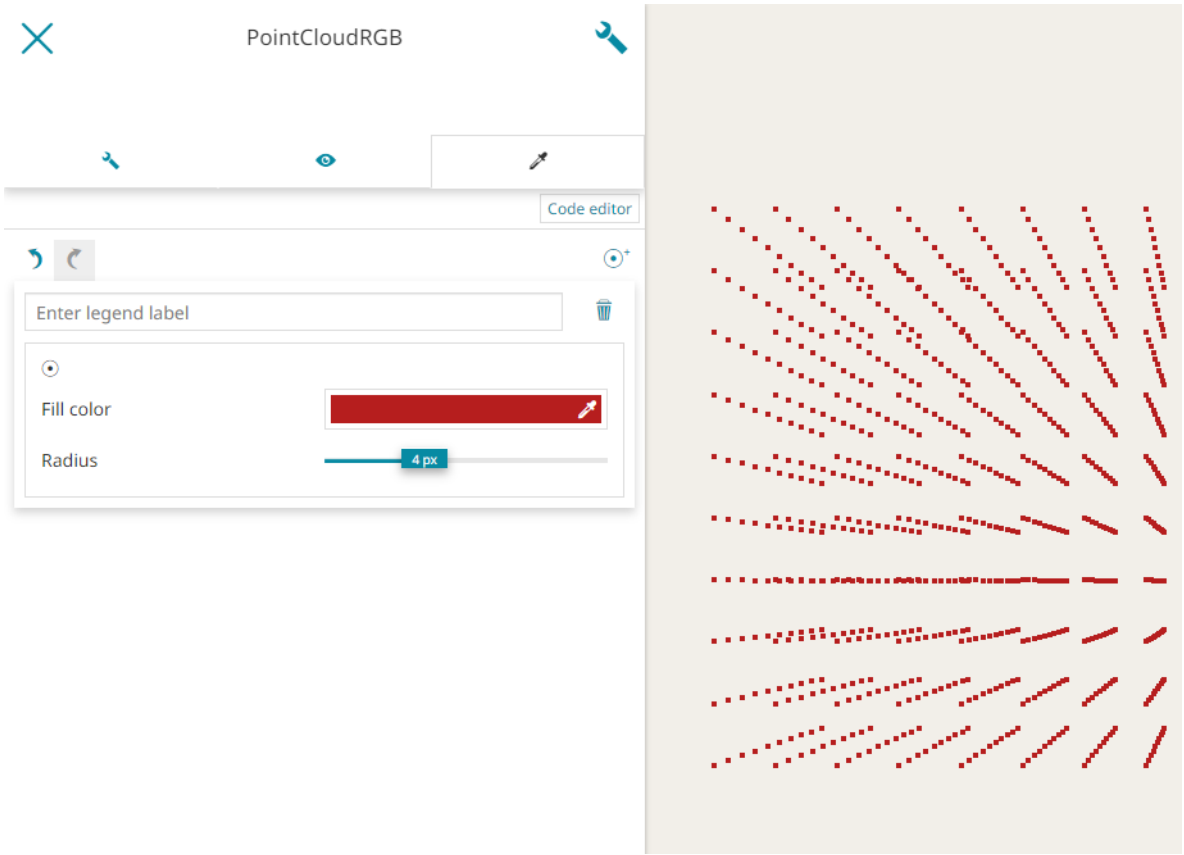


- Style Rule filtering based on the available properties dictionary defined in the tileset.json



- Possibility to customize the radius in case of point cloud features


PointCloudRGB
Visual editor
"color": "color('#b61e1e', 1)",
"pointSize": 4


PointCloudRGB
Code editor
Enter legend label
Fill color
Radius
4 px

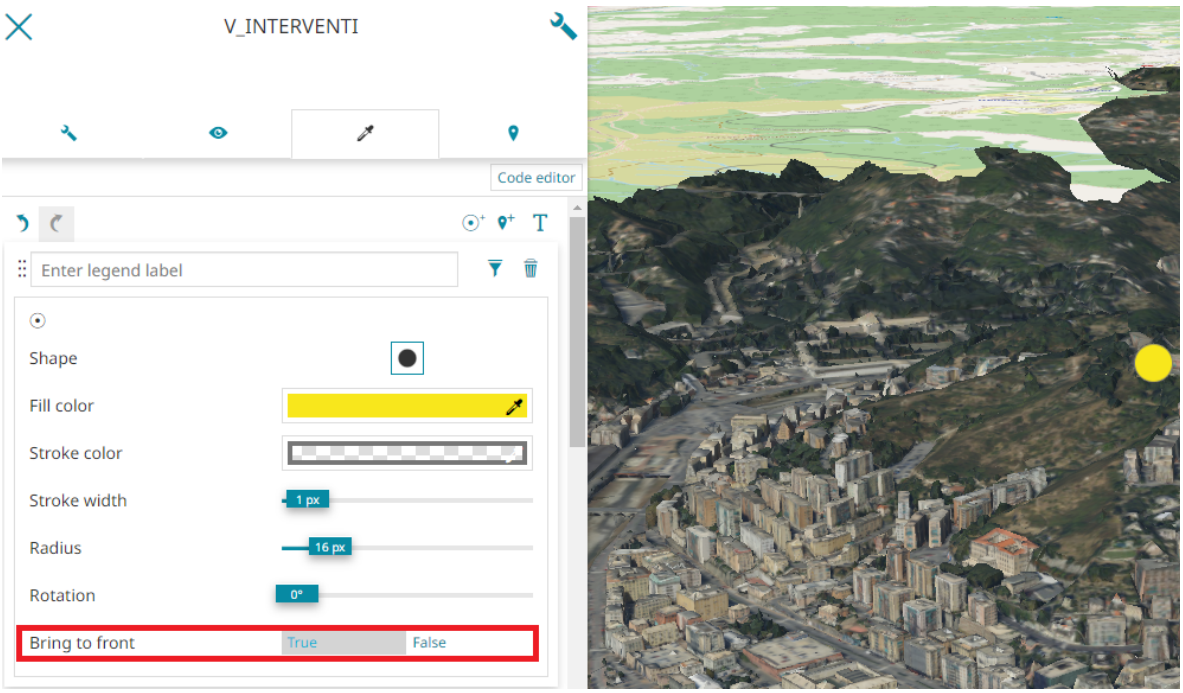
Styling of Vector layer

In 3D Navigation, MapStore allows to customize the style of the *Vector layer* with the same characteristics of the Visual Style Editor as described in the previous chapter.

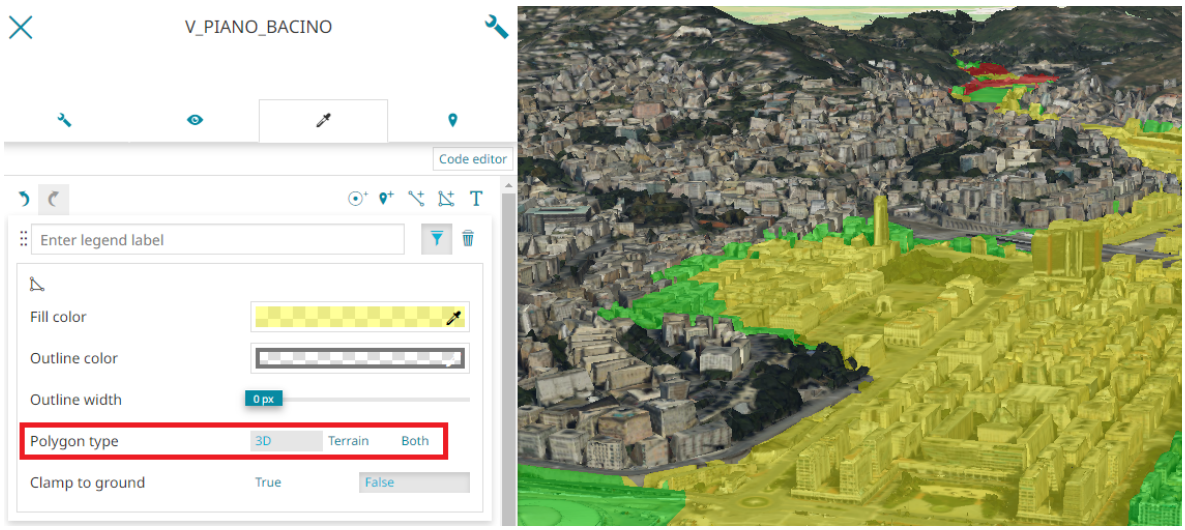
> **Warning**
>
> For the Vector layer, the *Cesium Style Editor* have some limitations:
>
> - It's possible to apply only one type of symbolizer at the time, so if the rule editor shows multiple rule with the same filter, only the first one is used.
> - For the *Line symbolizers*: the *Line cap* and *Line join* options are not available as properties in Cesium

Furthermore, MapStore adds some customization options, for **WFS layers**, in the *Cesium Style Editor* which are:

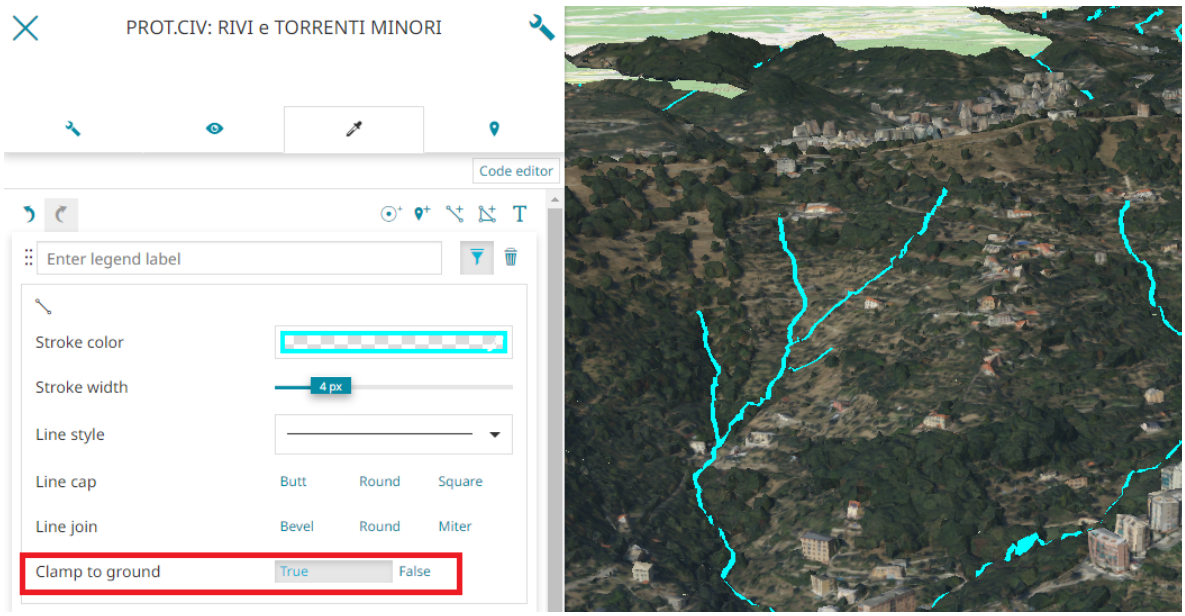
- **Bring to front** to bring the Mark or the Icon in front of the **3d Tiles** layer (This option is available for Icon and Mark symbolizers).



- **Polygon type** to choose whether the classification, drape effect, should affect `3D`, `Terrain` or `Both`. (This option is available for Fill symbolizers)

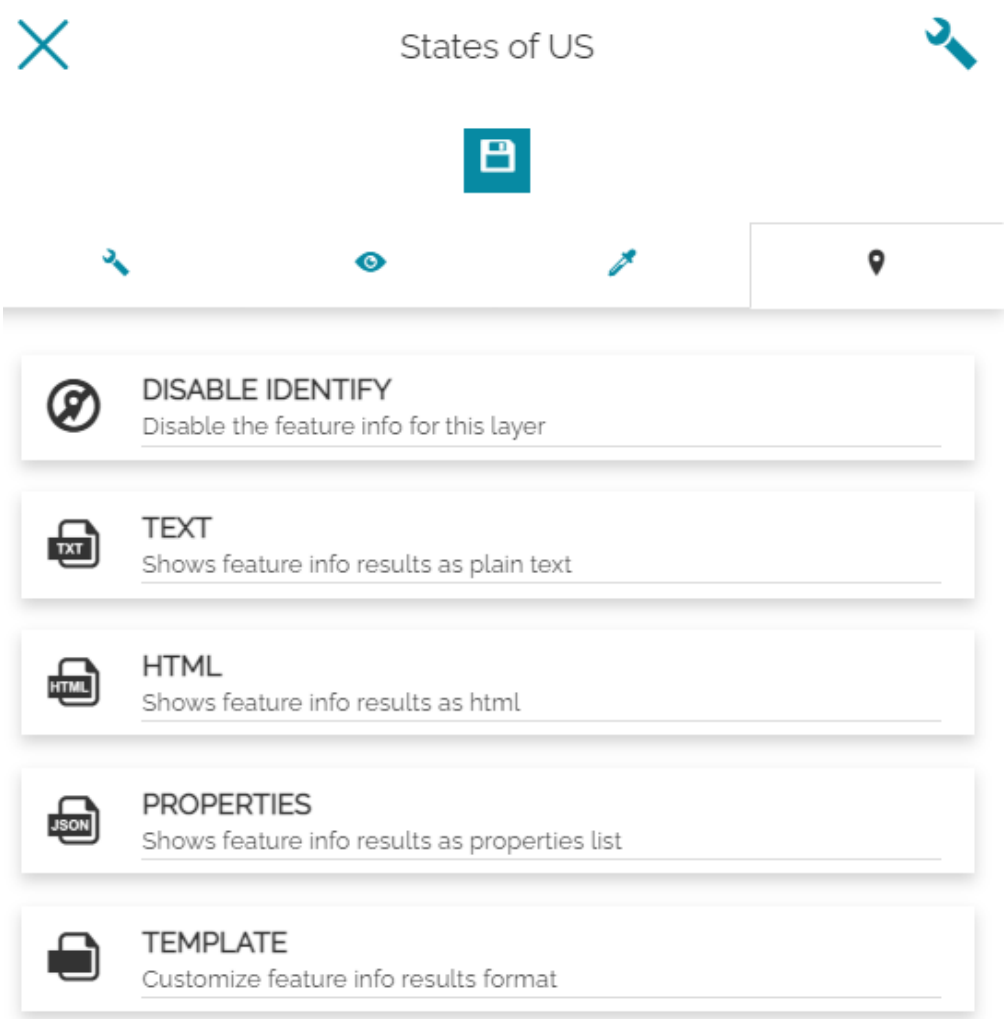


- **Clap to ground** to enable/disable the boolean Property specifying whether the polyline should be clamped to the ground (This option is available for Line and Fill symbolizers).



Feature Info Form

Through the last section of the layer settings panel, it is possible to decide the information format that appears querying a layer with the Identify Tool:



In particular, the user can choose between:

- **Disable Identify** to disable the Identify for the layer
- **Text**
- **HTML**
- **Properties**
- **Template**

> **Note**
>
> Without selecting any format here, the Identify Tool will return the layers information with the format chosen in Map Settings (in the Side Toolbar). Once a user specifies the information format in layers settings, instead, that format will take precedence over the map settings only for that specific layer.

Text

An example of layer information in text format can be:

```
Results for FeatureType 'https://gs-stable.geo-solutions.it/geoserver/geoserver:us_st
ates':
------------------------------------------
the_geom = [GEOMETRY (Polygon) with 297 points]
STATE_NAME = Montana
STATE_FIPS = 30
SUB_REGION = Mtn
STATE_ABBR = MT
LAND_KM = 376990.894
WATER_KM = 3858.589
PERSONS = 799065.0
FAMILIES = 211666.0
HOUSHOLD = 306163.0
MALE = 395769.0
FEMALE = 403296.0
WORKERS = 293243.0
DRVALONE = 250373.0
CARPOOL = 41442.0
PUBTRANS = 2050.0
EMPLOYED = 350723.0
UNEMPLOY = 26217.0
SERVICE = 123090.0
MANUAL = 43619.0
P_MALE = 0.495
P_FEMALE = 0.505
SAMP_POP = 150091.0
------------------------------------------
```

HTML

An example of layer information in HTML format can be:

us_states

fid	STATE_NAME	STATE_FIPS	SUB_REGION	STATE_ABBR	LAND_KM	WATER_KM	PERSON
us_states.28	South Dakota	46	W N Cen	SD	196575.21	3169.429	696004

Properties

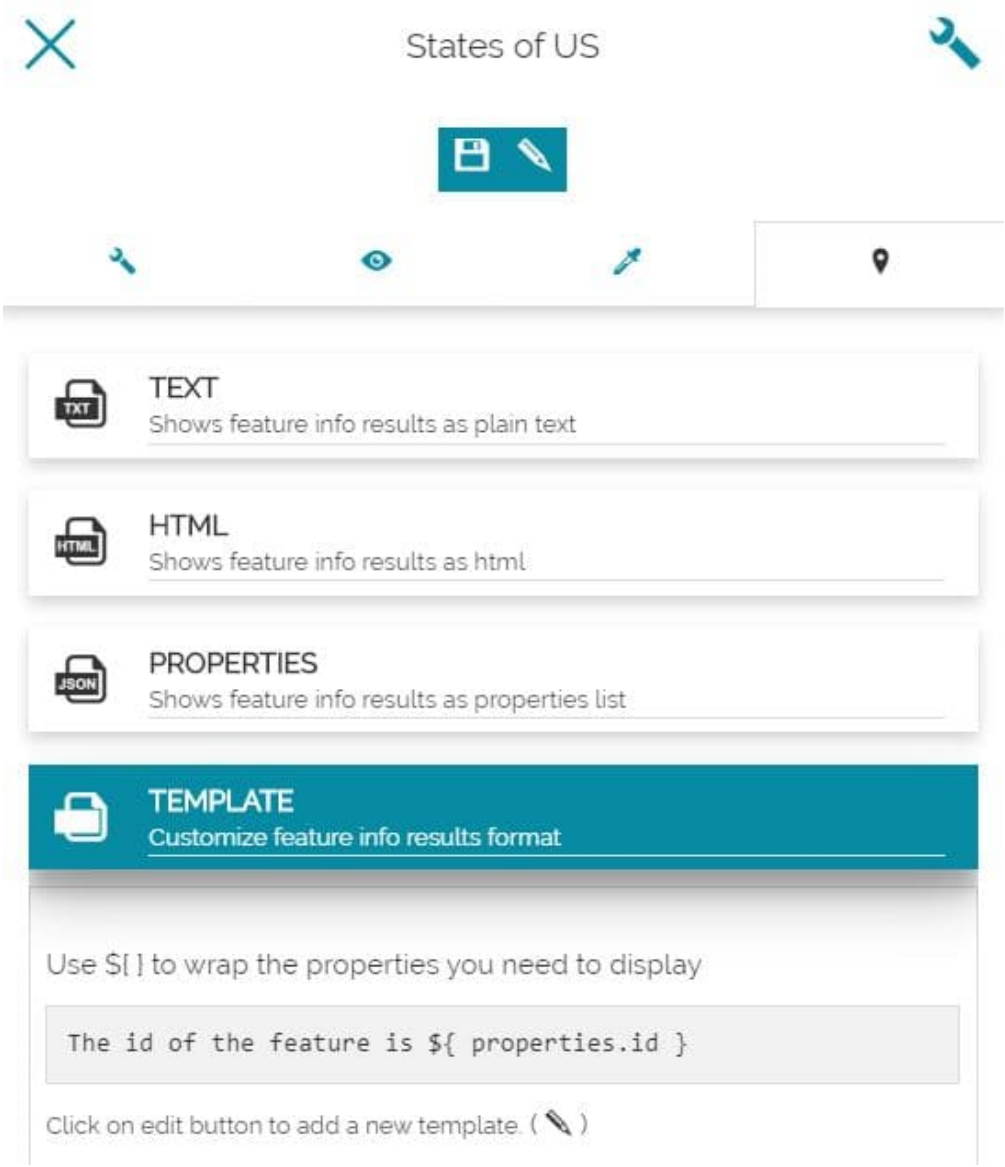
An example of layer information in properties format can be:

us_states.28

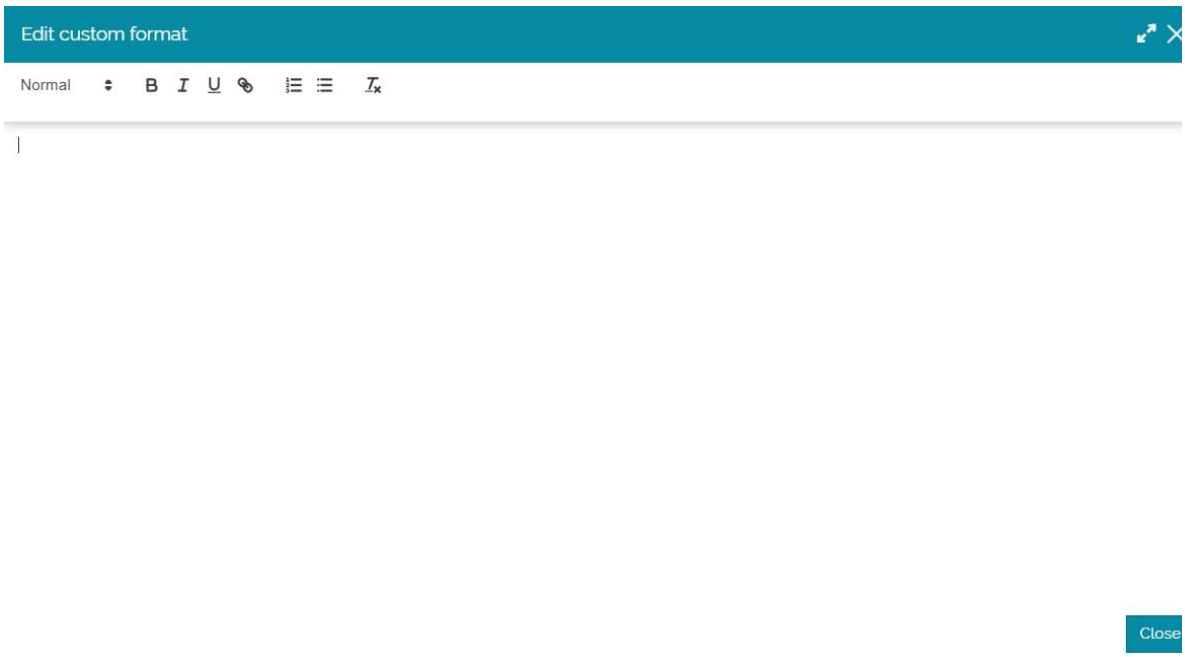
STATE_NAME South Dakota
STATE_FIPS 46
SUB_REGION W N Cen
STATE_ABBR SD
LAND_KM 196575.21
WATER_KM 3169.429
PERSONS 696004
FAMILIES 180306
HOUSHOLD 259034
MALE 342498
FEMALE 353506
WORKERS 250825
DRVALONE 233478
CARPOOL 32610
PUBTRANS 971
EMPLOYED 321891
UNEMPLOY 13983
SERVICE 119594
MANUAL 41921
P_MALE 0.492
P_FEMALE 0.508
SAMP_POP 162746

Templates

In this case the user can customize the information format:

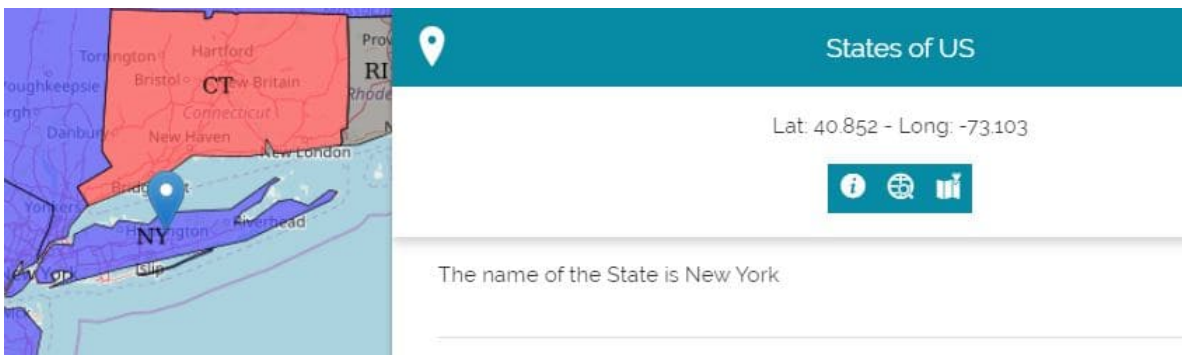


In particular, by clicking on the button, the following text editor appears:



Here it is possible to insert the text to be displayed through the Identify Tool, with the possibility to wrap the desired properties.
Let's make an example: we assume to have a layer where each record corresponds to a USA State geometry in the map. In the Attribute Table of this layer there's the `STATE_NAME` field that, for each record, contains a text value with the name of the State.
If the goal is to show, performing the Identify Tool, only the State name, an option could be to insert the following text on the Template text editor:

In this case, by clicking on the map, the Identify Tool returns:



Using the `${properties.NAME_OF_THE_FIELD}` syntax, MapStore is able to parse the response to the Identify Tool request by matching the configured placeholder.

Filtering Layers

When using vector layers it might be useful to work with a subset of features. About that, MapStore let the user set up a **Layer Filter** that acts directly on a layer with WFS available and filter its content upfront. The map will immediately update when a filter is applied.

> **Warning**
>
> The MapStore's filtering capabilities are working on top of the WFS specifications so that service must be enabled if you want to filter a layer using the tools described in this section.

Filter types

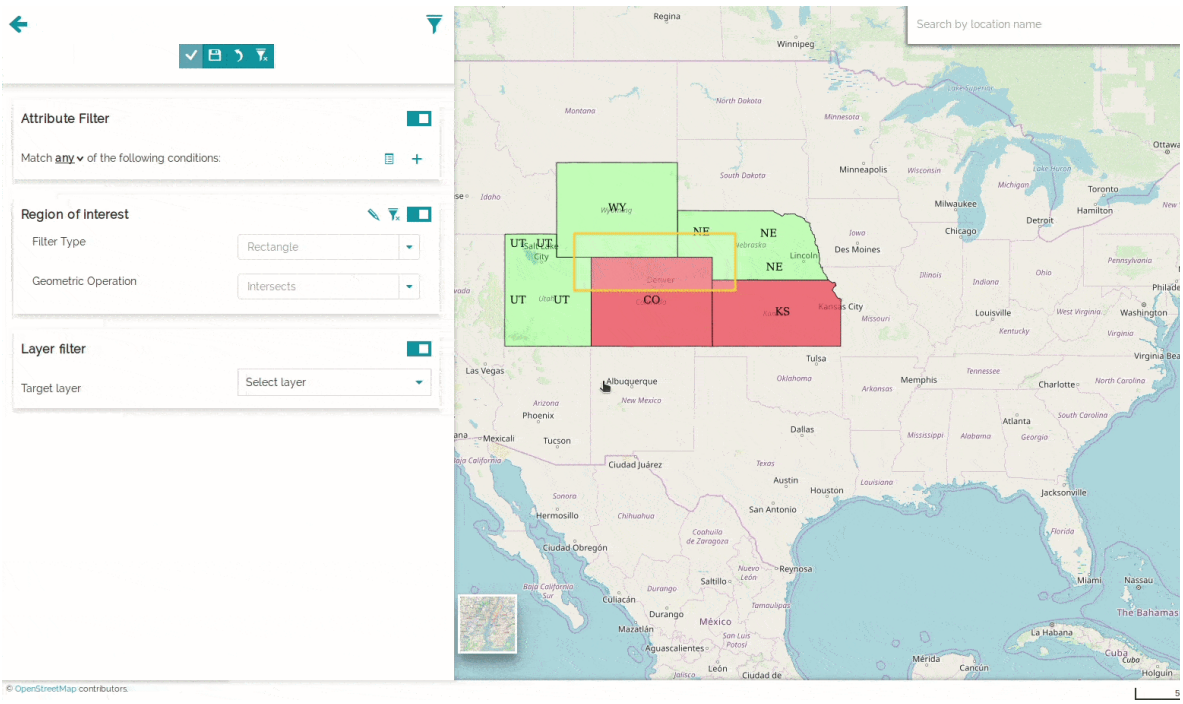
In MapStore it is possible to apply filters on layers in three different ways:

- With the *Layer Filter* tool available in TOC
- With the *Advanced Search* tool available from the Attribute Table
- With the *Quick Filter* available in the Attribute Table

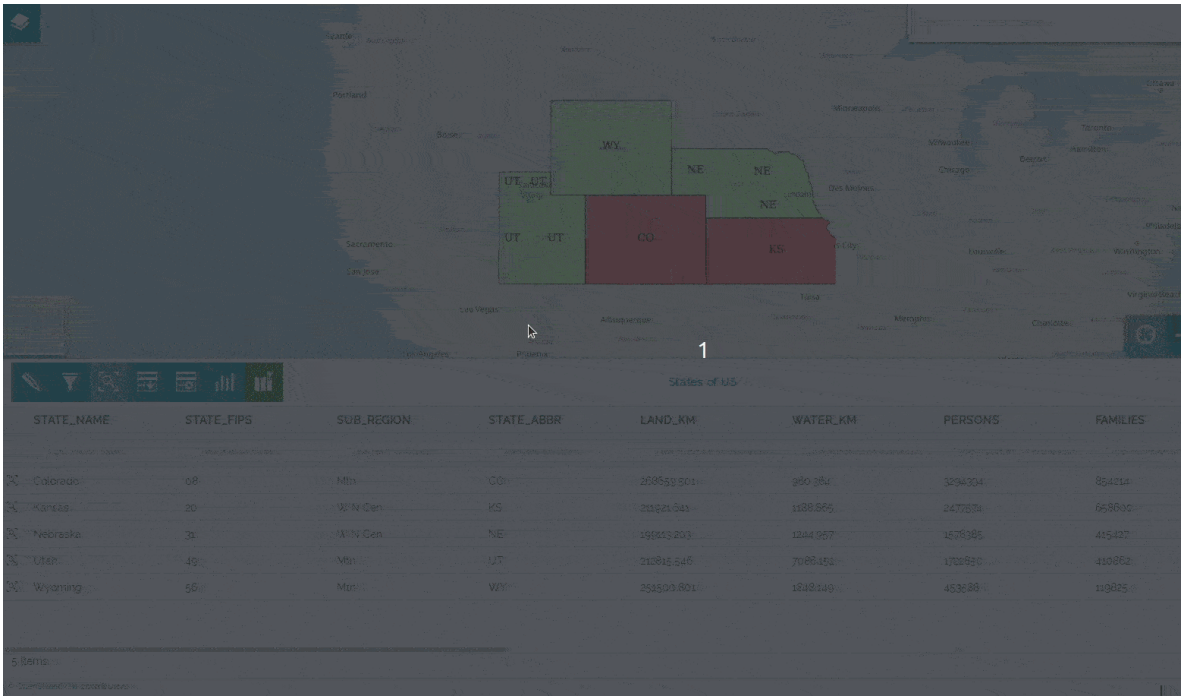
Layer Filter

This filter is applicable from the **Filter layer** button  in TOC's Layers Toolbar and it will persist in the following situations:

- Using other tools like the Identify tool:

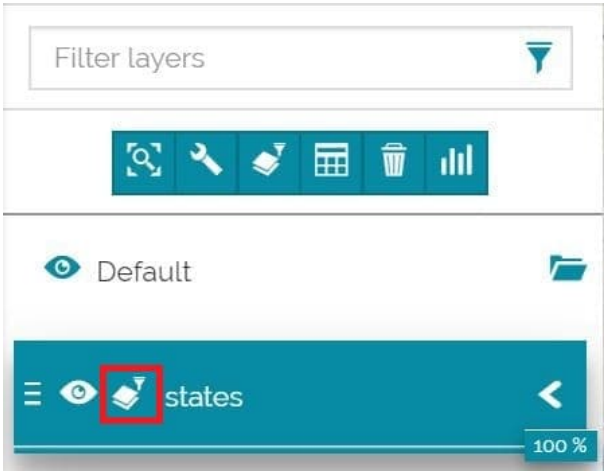


- Applying another type of filter



- Opening the map next time (you need to Save the map from the Side Toolbar after applying a filter)

Once a *Layer filter* is set, it is possible to enable/disable it simply by clicking on the button that will appear near the layer name in TOC:



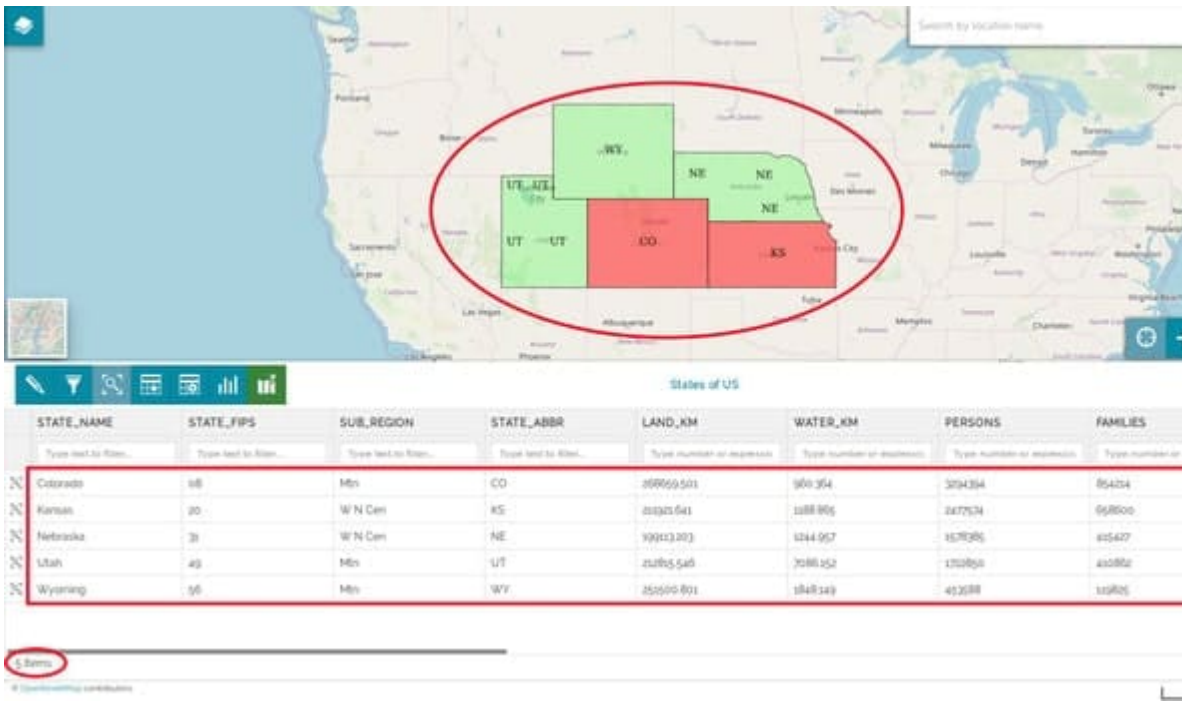
This filter is applied through the Query Panel. Once the settings are chosen, it is possible to **Apply** them. After that the user can:

- **Undo** the last changes
- **Reset** the filter to the initial situation
- **Save** the filter in order to make it persistent

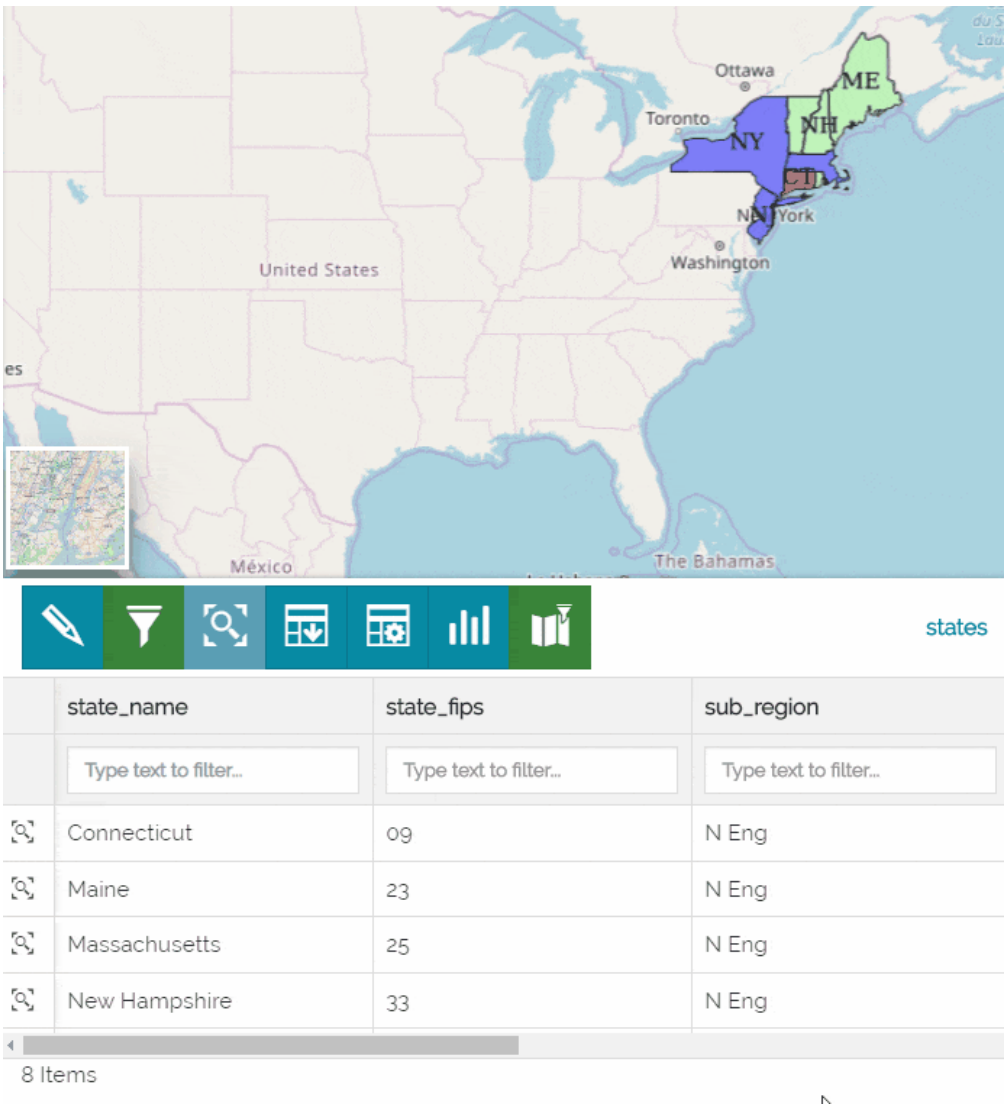
Advanced Search

This filter, applicable from **Advanced Search** button in Attribute Table, behaves as follows:

- It can be used to apply a filter to a layer for search purposes in Attribute Table: this filter is applied in `AND` to the *Layer Filter* if it is already been set.



- It is possible to sync this filter with the map through the  icon:



- It will be automatically removed/reapplied by closing/opening the Attribute Table

Also this filter is applied through the Query Panel but in this case it is not possible to Save it and make it persistent reopening the map the next time. The user is only allowed to apply it by clicking on **Search** or eventually **Reset** it.

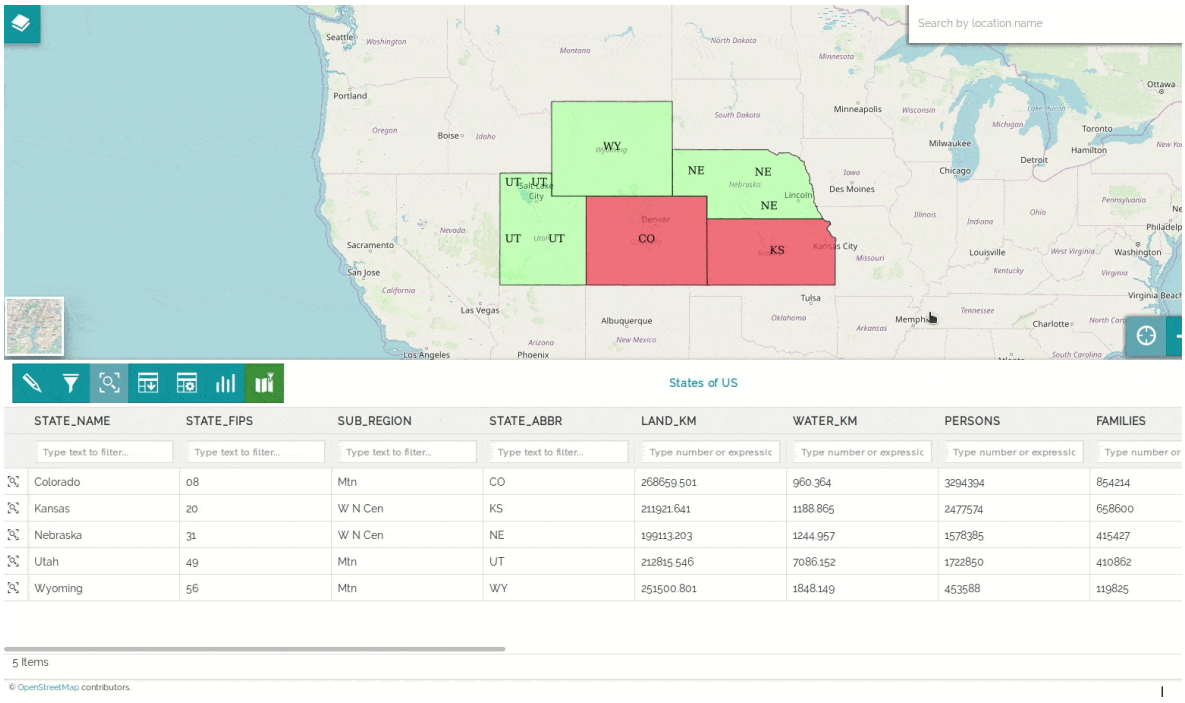
Quick Filter

The user can perform two type of quick filters:

- Filter by **attributes**
- Filter by **clicked point in the map**

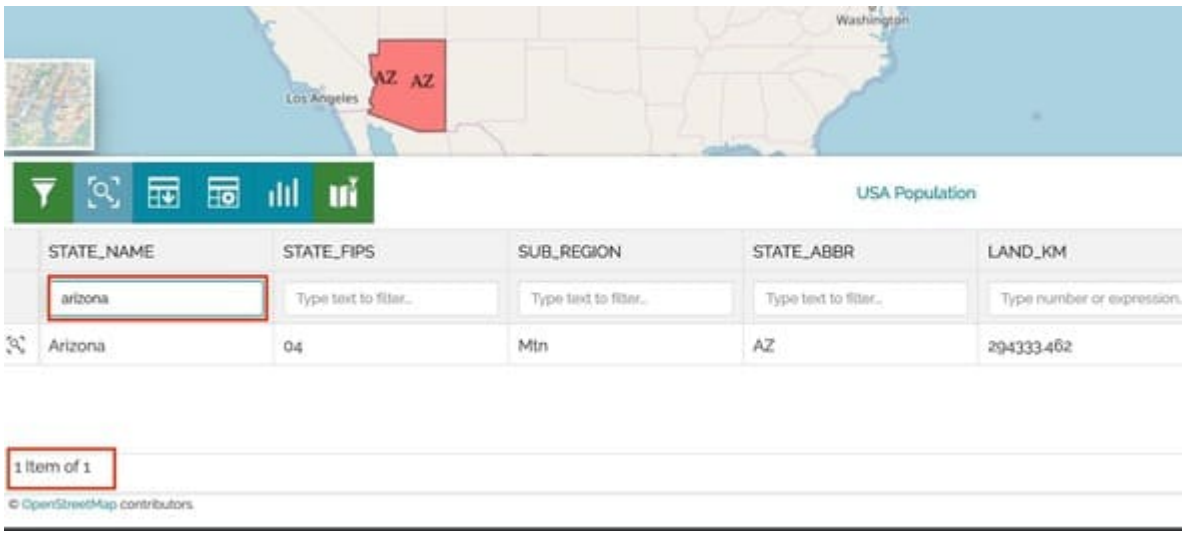
Quick Filter by attributes

This filter is available for each colum in the Attribute Table just below the field names and it can be also used in combination with other filter applied:



The user has the possibility to apply simple filters by attributes simply typing the filter's value in the available input fields (Date or Time pickers are available according to real attributes data types and a tooltip usually gives an information on how to fill the filter's input field). Filtering by one or more attributes, layer records in Attribute Table are automatically filtered accordingly.

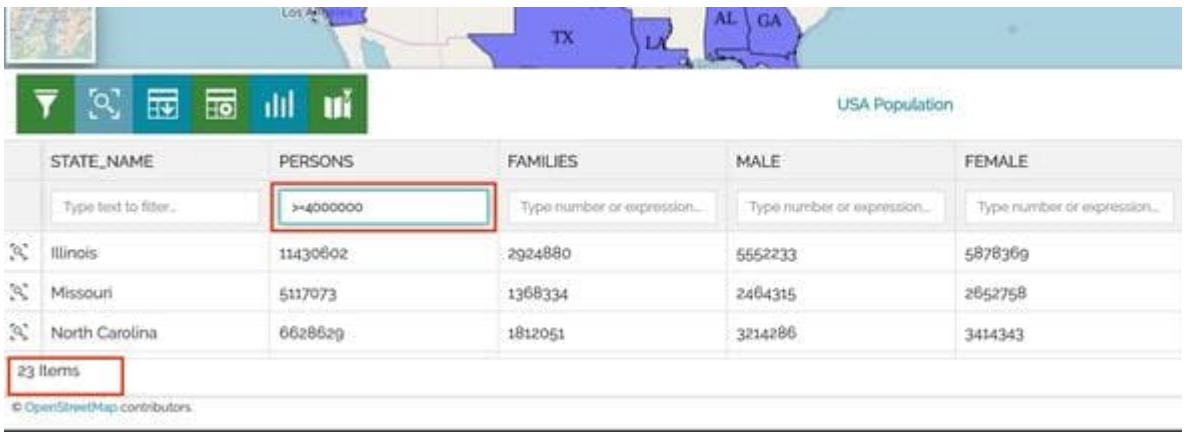
If the user wants to filter by an attribute of type String, he can simply write something inside the input box and the list of records in table will be automatically filtered by matching with the input text.



If the User wants to filter by a numeric attribute, he can type directly a number or an expression using the following operators:

- Not equal (**!=** or **!==** or **<>**)
- Equal or less than (**<=**)
- Equal or greater than (**>=**)
- Less than (**<**)
- Greater than (**>**)
- Equal (**===** or **==** or =)

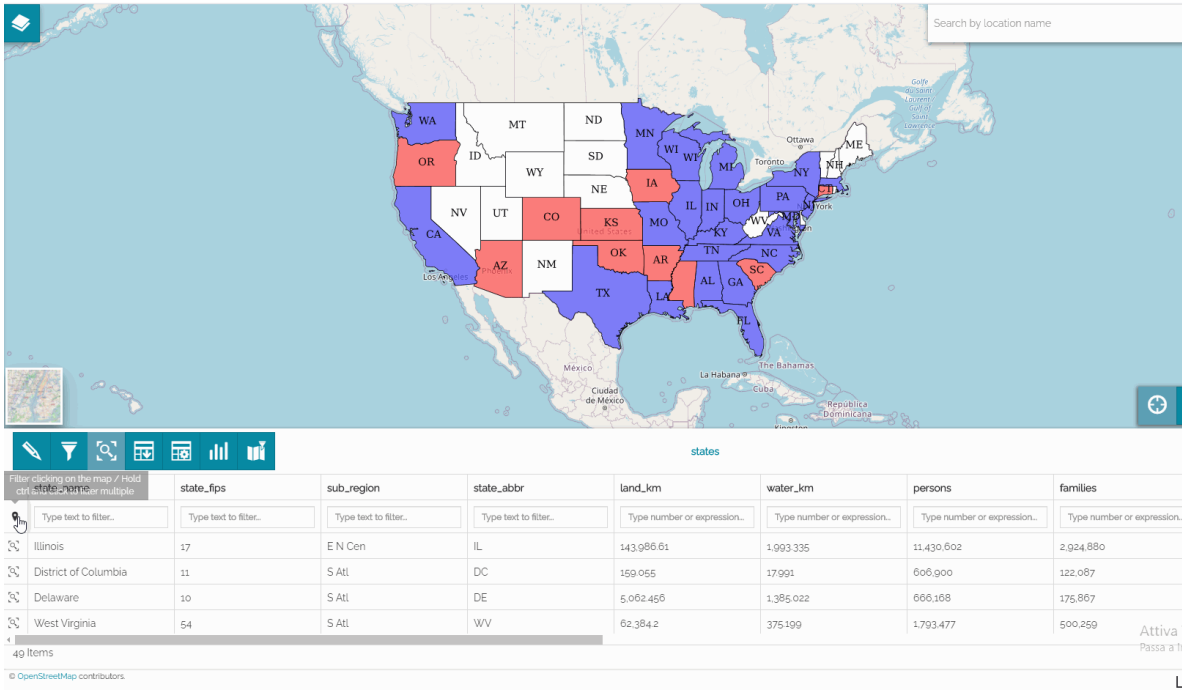
In order to filter a numerical filed matching the records *greater than* or *equal* to a certain threshold value, an example can be:



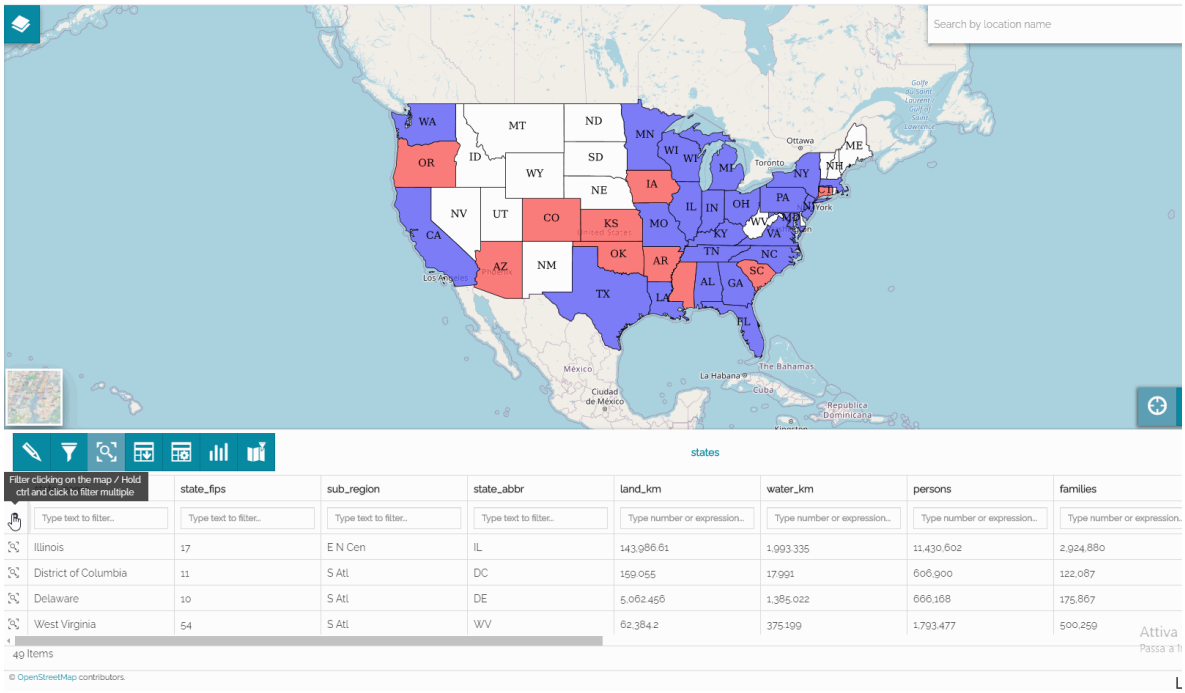
Quick Filter by map interaction

It is possible to filter records in the Attribute Table by clicking on the map or doing a selection directly in a map of multiple features. The user can activate the **Filter on the map** button (once clicked the button turns blue) and then:

- Click on the map over the features he wants to select
- Add multiple features to the selection by pressing Ctrl and clicking again over other features in map



- Add multiple features to the selection by pressing Ctrl + Alt and drawing a selection box in map



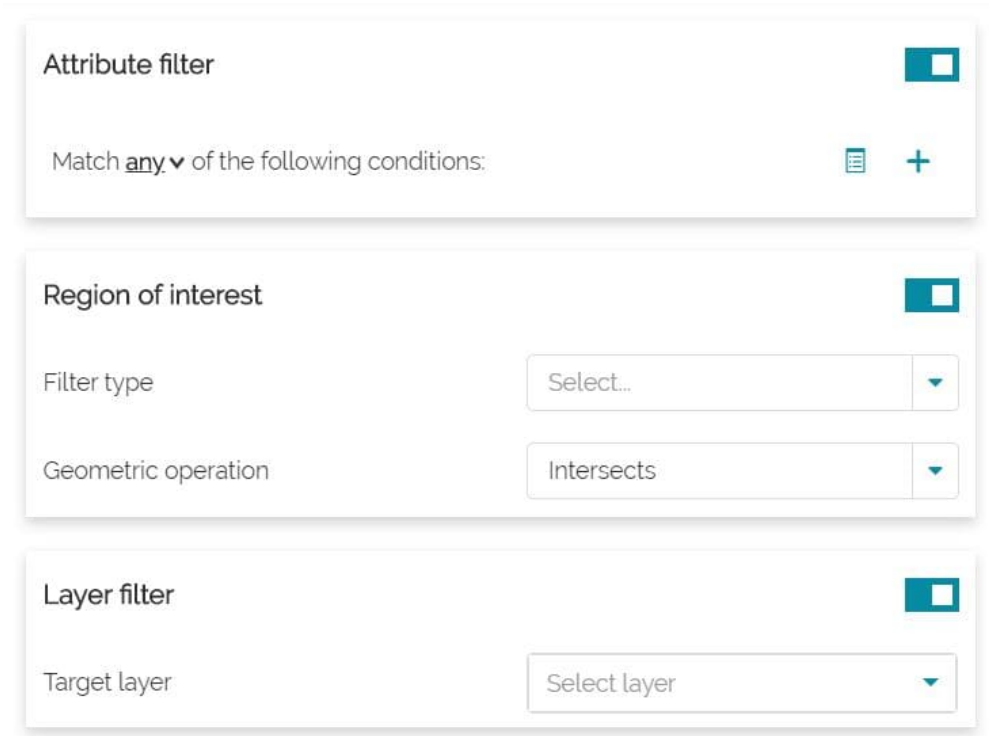
The list of records in the *Attribute Table* will be automatically filtered according to such user selection and then the user can disable the geometry filter through the **Remove filter** button.

> **Note**
>
> The *Quick Filter* remains active as long as the Attribute Table is open but, unlike the *Advanced Search*, closing the Attribute Table it will not reappear anymore if the Attribute Table is re-opened in a second time.

Query Panel

This tool is used to define advanced filters in MapStore. It includes three main sections:

- **Attribute Filter**
- **Region of Interest**
- **Layer Filter**



Attribute filter

This filter allows to set one or more conditions referred to the Attribute Table fields.
First of all it is possible to choose if the filter will match:

- **Any** conditions
- **All** conditions
- **None** conditions

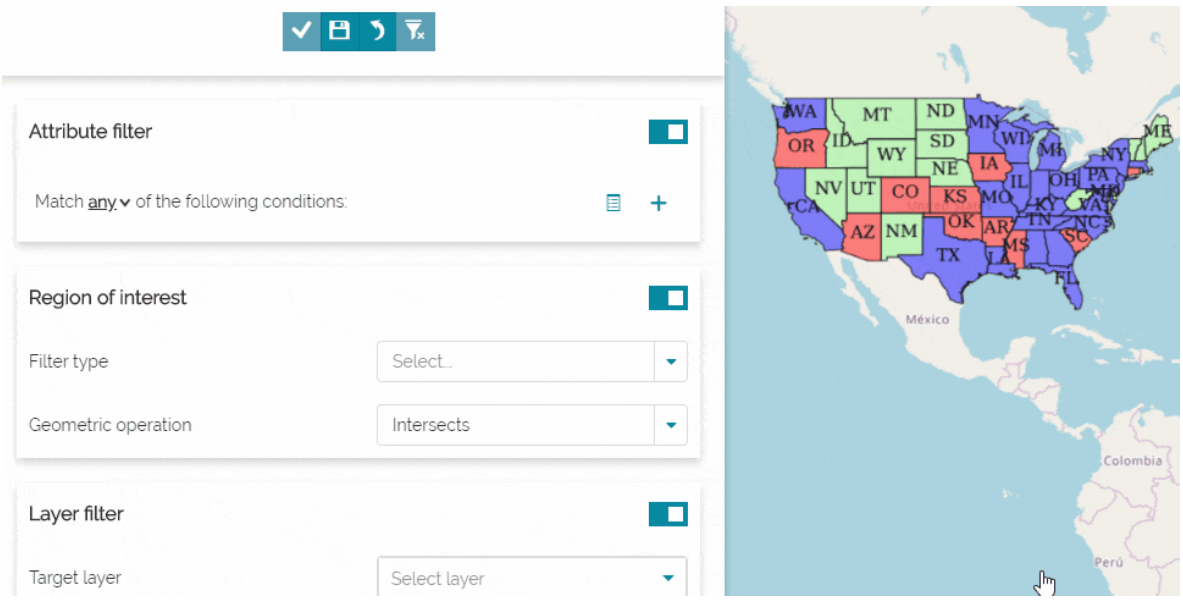
After that, the user can insert one or more conditions, that can also be grouped in one or more condition groups (use the button in order to create a group).
A condition can be set by selecting a value for each of the three input boxes:

- The first input box allows to choose a layer field
- In the second input box it is possible to choose the operation to perform (selecting a text field can be **=**, **like**, **ilike** or **isNull**, selecting a numerical field, can be **=**, **>**, **<**, **>=**, **<=**, **<>** or **><**)
- The third input box (in case of fields of type String) provides a paginated list of available field values already present in the layer's dataset (a GeoServer

WPS process is used for this). In case of numeric fields the user can simply type a value to use for the filter.

> **Note**
>
> the "paginated list of available field values" above is available only if the server provides the WPS process `gs:PagedUnique`

A simple *Attribute Filter* applied for a numerical field can be, for example:

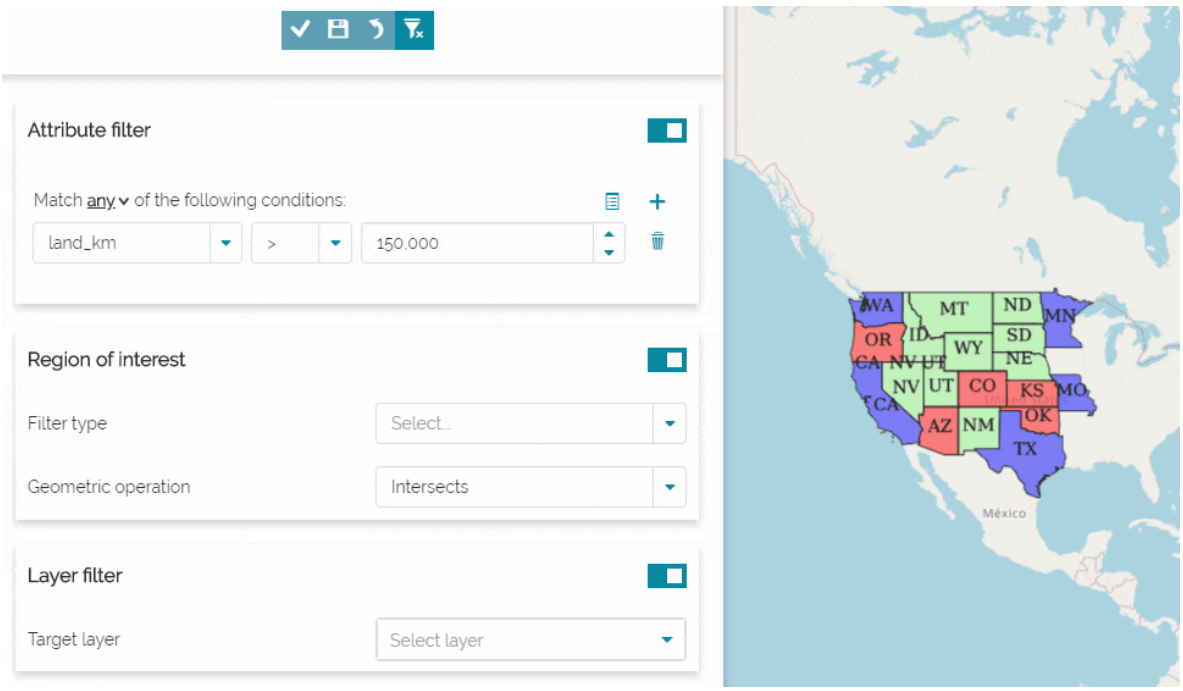


Region of interest

In order to set this filter the user can:

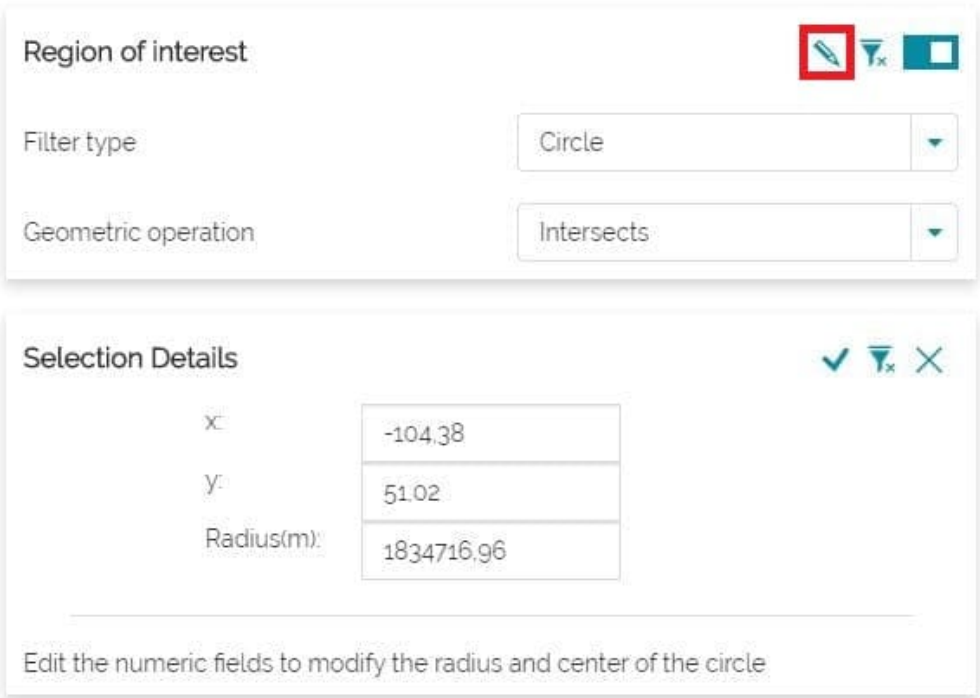
- Select the *Filter type* by choosing between **Viewport**, **Rectangle**, **Circle**, **Polygon** (selecting Rectangle, Circle or Polygon it is necessary to draw the filter's geometry on the map)
- Select the *Geometric operation* by choosing between **Intersects**, **Is contained**, **Contains**

Applying a *Circle* filter with *Intersect* operation, for example, the process could be similar to the following:



Once this filter is set, it is always possible to edit the coordinates and the dimensions of the drawn filter's geometry by clicking on the **Details** button  .

Editing a circle, for example, it is possible to change the center coordinates (*x*, *y*) and the radius dimension (*m*):



Layer filter

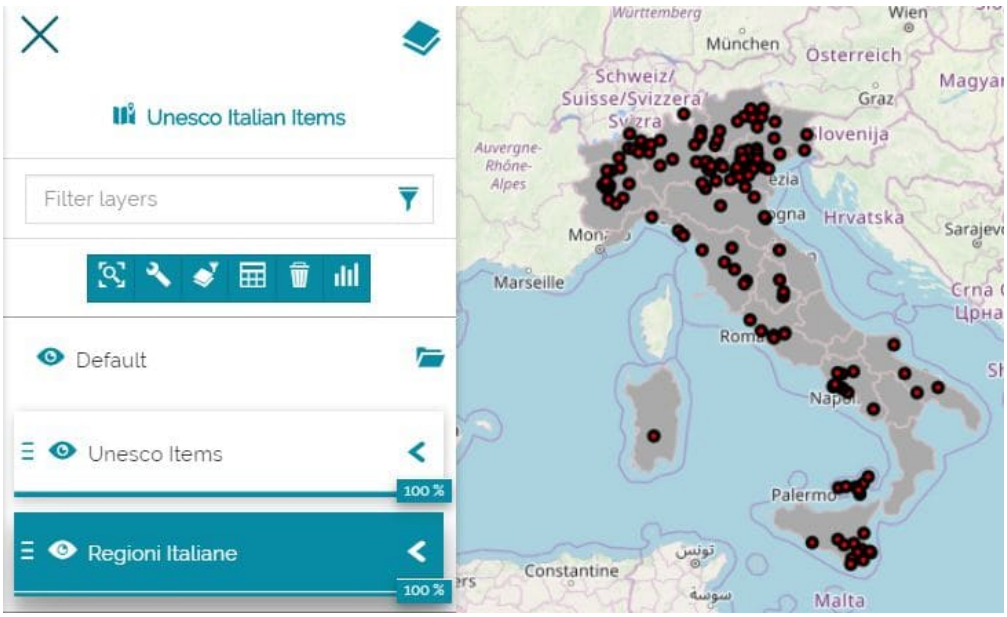
This tool allows to set cross-layer filters for a layer by using another layer or even the same one.

> **Warning**
>
> This filter tool requires the Query Layer plugin installed in GeoServer

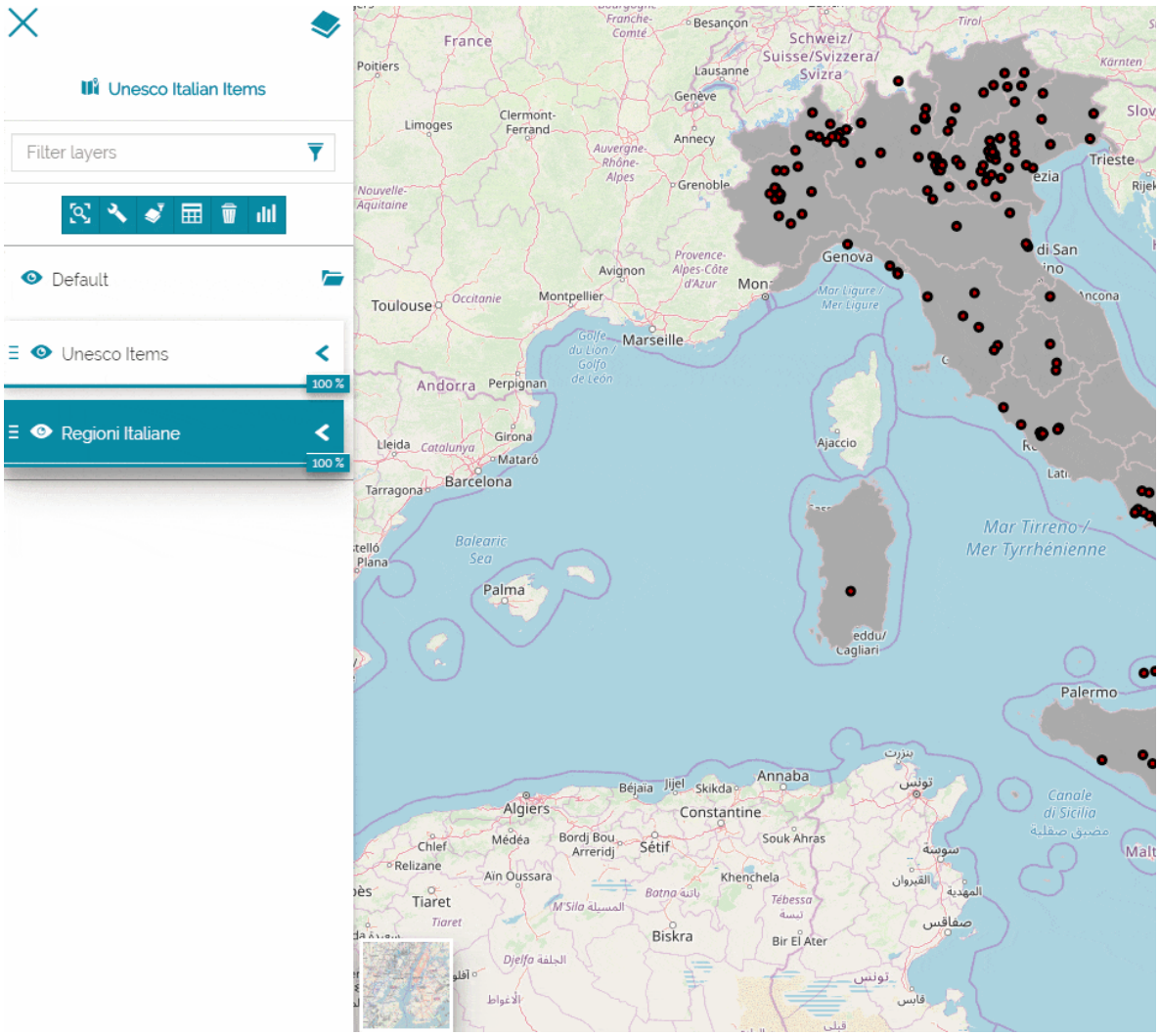
In order to set up a cross-layer filter the options below are required:

- *Target layer* (between those present in the TOC)
- *Operation* to be chosen between **Intersects**, **Is contained** or **Contains**
- Optionally some *Conditions* (see Attribute filter)

In order to better understand this type of filter, let's make an example. We suppose that the user want to filter the Italian Regions with the Unesco Item's one:

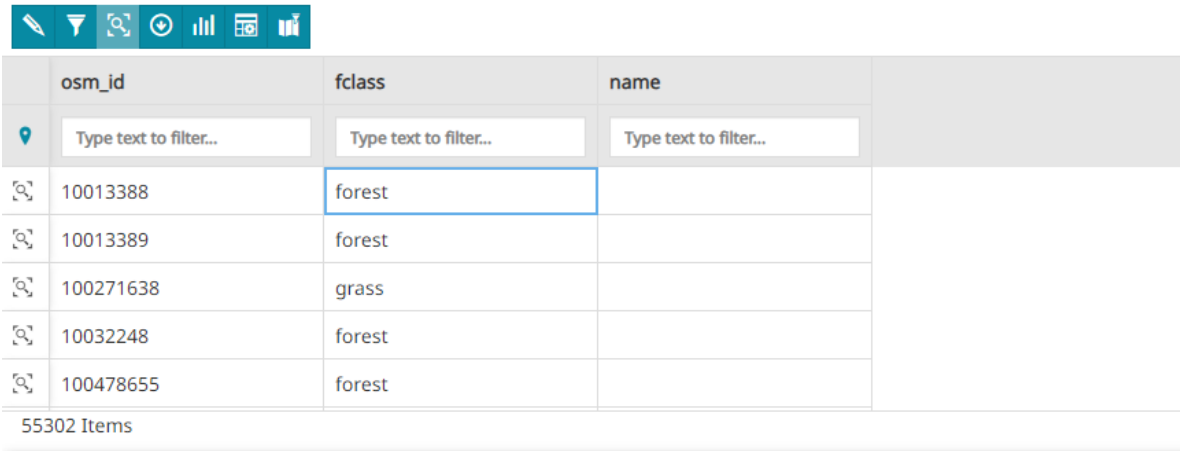


In particular, if our goal is to take a look at the Italian Regions that contain the Unesco sites with *serial code=1*, the operations to perform can be the following:


Unesco Italian Items
Filter layers
Default
Unesco Items
100 %
Regioni Italiane
100 %
France
Besançon
Schweiz/
Suisse/Svizzera/
Svizra
Lausanne
Genève
Poitiers
Limoges
Clermont-
Ferrand
Annecy
Grenoble
Trieste
Genova
Avignon
Montpellier
Toulouse
Occitanie
Marseille
Andorra
Perpignan
Girona
Lleida
Catalunya
Mataró
Barcelona
Tarragona
Ajaccio
Balearic
Sea
Palma
Mar Tirreno /
Mer Tyrrhénienne
Palermo
Annaba
Béjaia
Jijel
Skikda
Algiers
Constantine
Médéa
Bordj Bou
Arreridj
Sétif
Souk Ahras
Chlef
Relizane
Ain Oussara
Khenchela
Tiaret
Biskra
Bir El Ater
Canale
di Sicilia

Attribute Table

In GIS, the Attribute Table associated to a vector layer is a table that stores tabular information related to the layer. The columns of the table are called fields and the rows are called records. Each record of the attribute table corresponds to a feature geometry of the layer. This relation allows to find records in the table (information) by selecting features on the map and viceversa.
In MapStore, through the button in Layers Toolbar it is possible to access the Attribute table:



Accessing this panel the user can perform the following main operations:

- Edit records through the button
- Filter records in Attribute Table in different ways as described in the Set filter section below
- Opening the Advanced Search tool through the button
- Activating the filtering capabilities by clicking on map, through button
- Using the quick filter by attribute
- Download the grid data through the button
- Customize the attribute table display through the button
- Create Widgets through the button

- Zoom to features through the button available on each record or zoom to the page max extent through the button (available only if the virtual scrolling is disabled, it is enabled by default in MapStore).

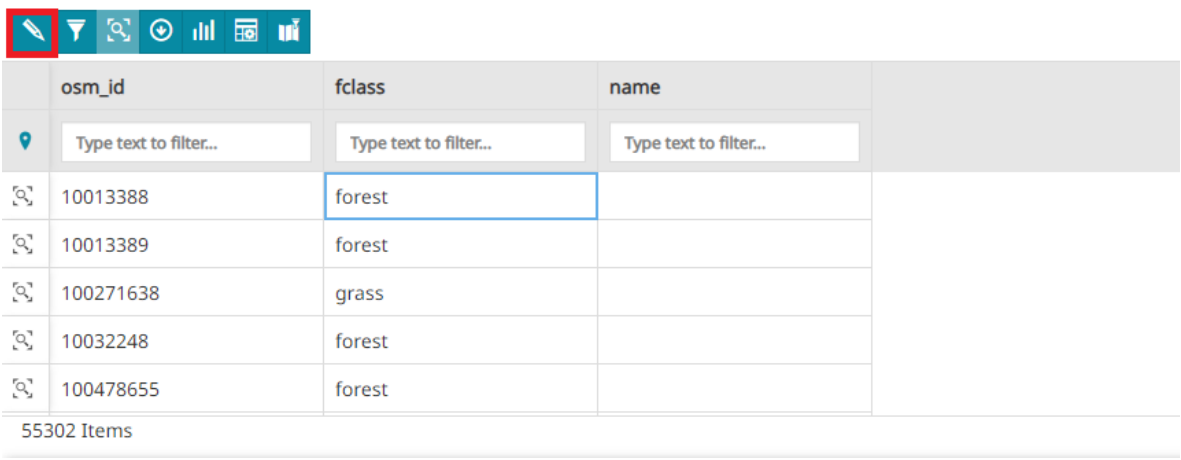
> **Warning**
>
> When GeoServer is set to strict CITE compliance for WFS (by default), the feature grid do not work correctly. This is because MapStore uses by default WFS 1.1.0 with startIndex/ maxFeatures. This is not strict compliant with WFS 1.1.0 (GeoServer supports it but the request in strict mode is invalid). To solve it un-check the CITE compliance checkbox in the "WFS" page of GeoServer "Services" configurations using the GeoServer web interface

Manage records

The basic Web Feature Service allows querying and retrieval of features. Through Transactional Web Feature Services (WFS-T) MapStore allows creation, deletion, and updating of features.

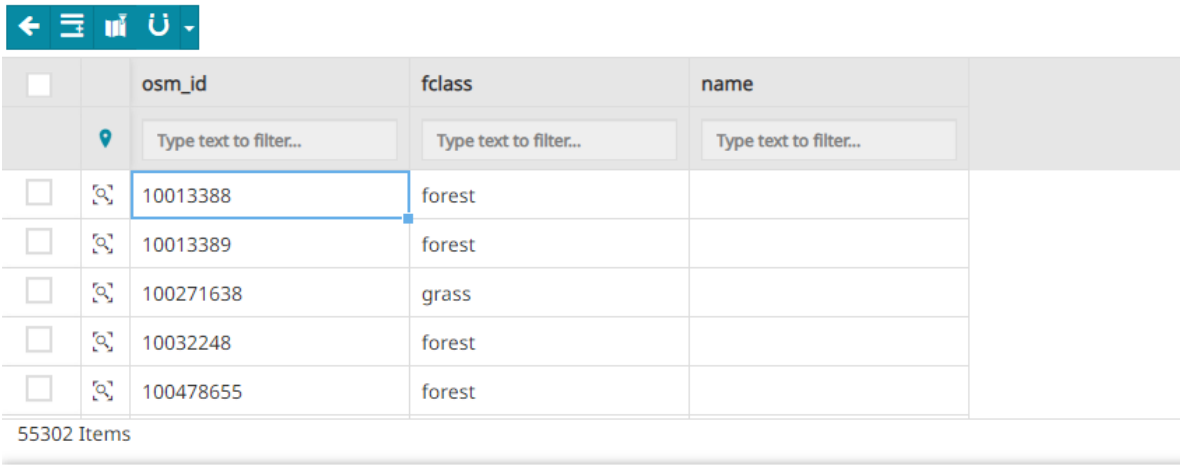
> **Warning**
>
> By default editing functionalities are available only for MapStore *Admin* users. Other users can use these tools only if explicitly configured in the plugin configuration (see the APIs documentation for more details). In any case, the user must have editing rights on the layer to edit it (see for example the GeoServer Security Settings).

The *Edit mode* can be reached from the button in *Attribute Table* panel, allowing to menage only the layer which the table refers to:



> **Note**
>
> When the *Edit mode* is enabled only the editing functionalities are available to the user, all other tools are deactivated.

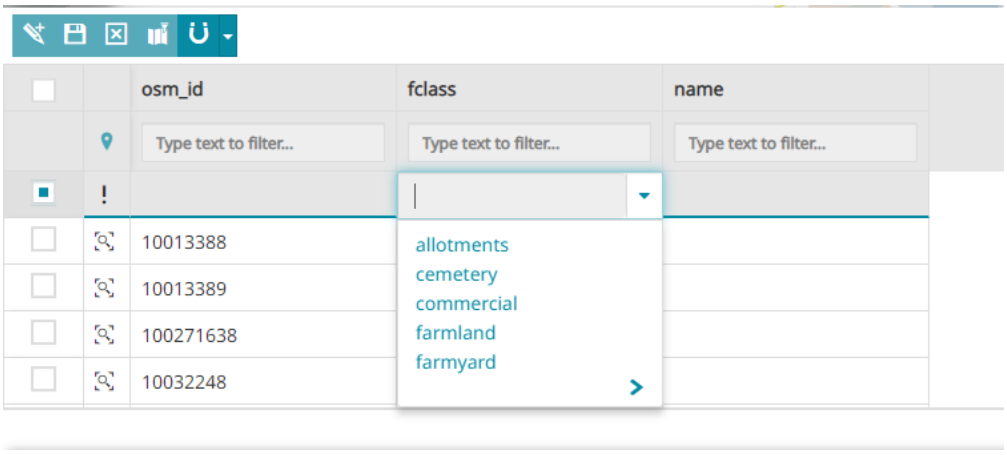
By default, in *Edit mode*, you can see a panel like that following:



Through the **Quit edit mode** button  you can stop the editing session to make the other functionalities available again.

Create new features

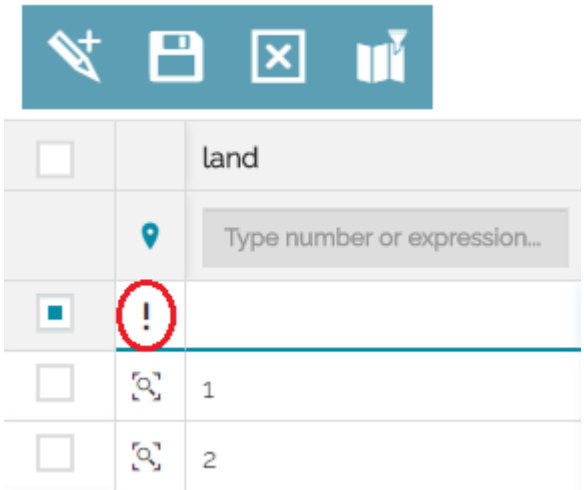
Once the *Edit mode* is enabled it is possible to create a new feature by clicking on the **Add New Feature** button  . After clicking on it the user can fill out the fields and edit the geometry of the new feature:



To edit attributes MapStore provides some input fields based on the attribute type, that forces the user to insert a valid value. If the attribute is of type `text`, MapStore will also show a dropdown menu with the list of the existing values for that attribute to allow a quick selection.

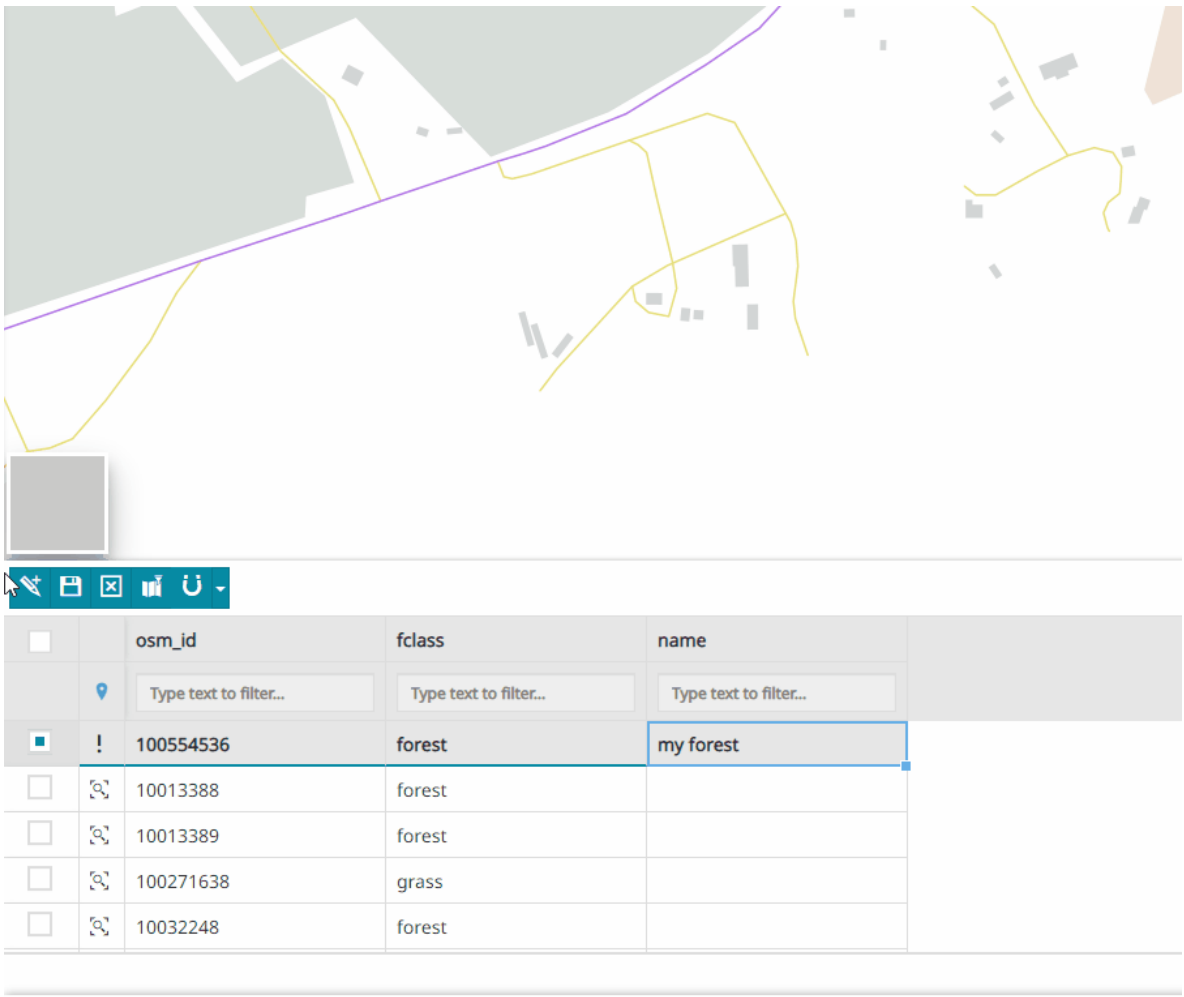
> **Note**
>
> the dropdown menu is available only if the server provides the WPS process `gs:PagedUnique`

The *Missing geometry* exclamation point ❗ in the second column of the *Attribute Table* means that the feature doesn't have a geometry yet. It's possible to add it later or draw it on the map before saving:

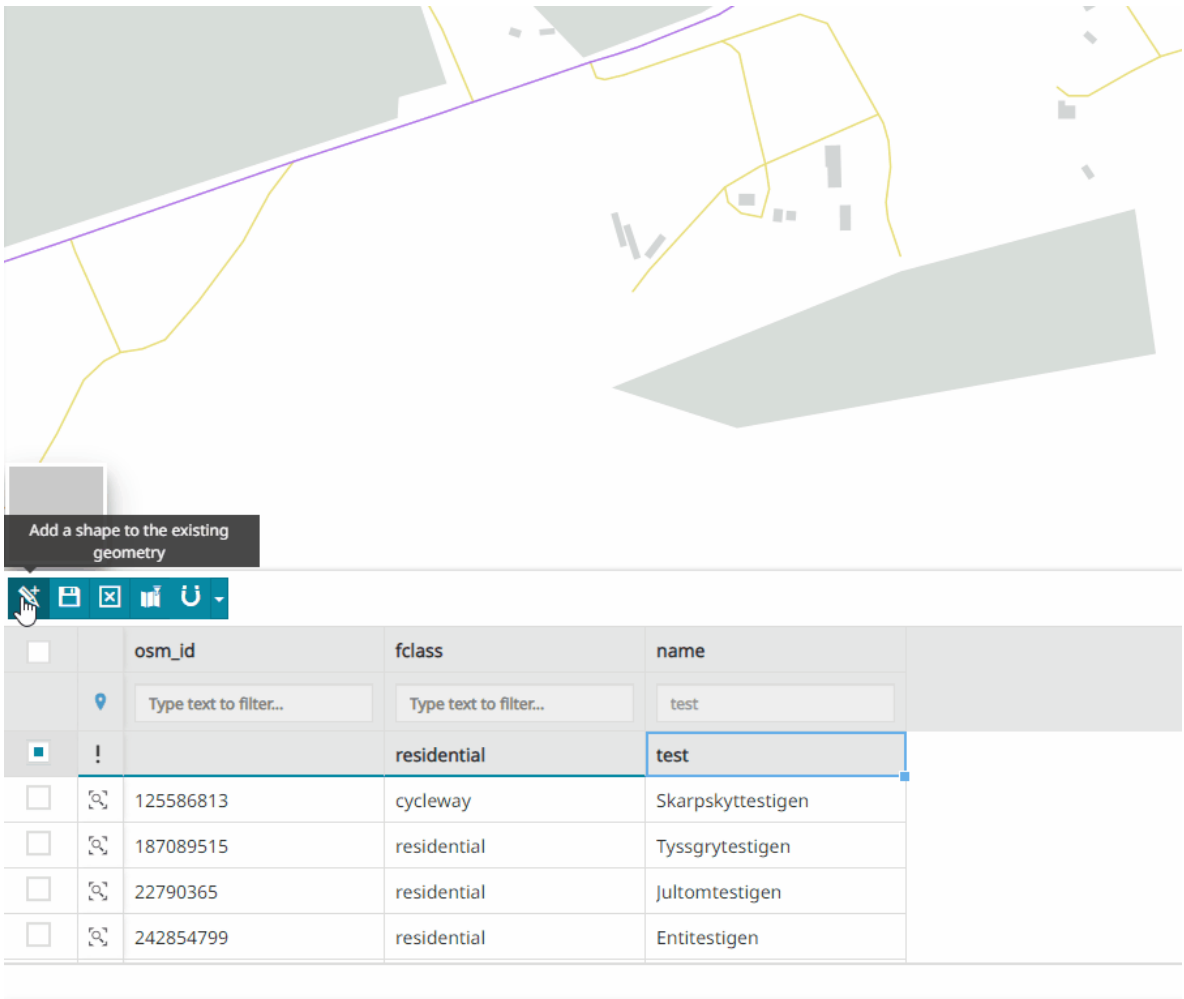


In order to save the changes made until now, there's the button, whereas to undo the changes there's the button.
Once a new record is created, it's possible to draw a geometry for it, by clicking on the button that appears once that feature is selected. The process of drawing a new geometry is a little different depending on the layer type:

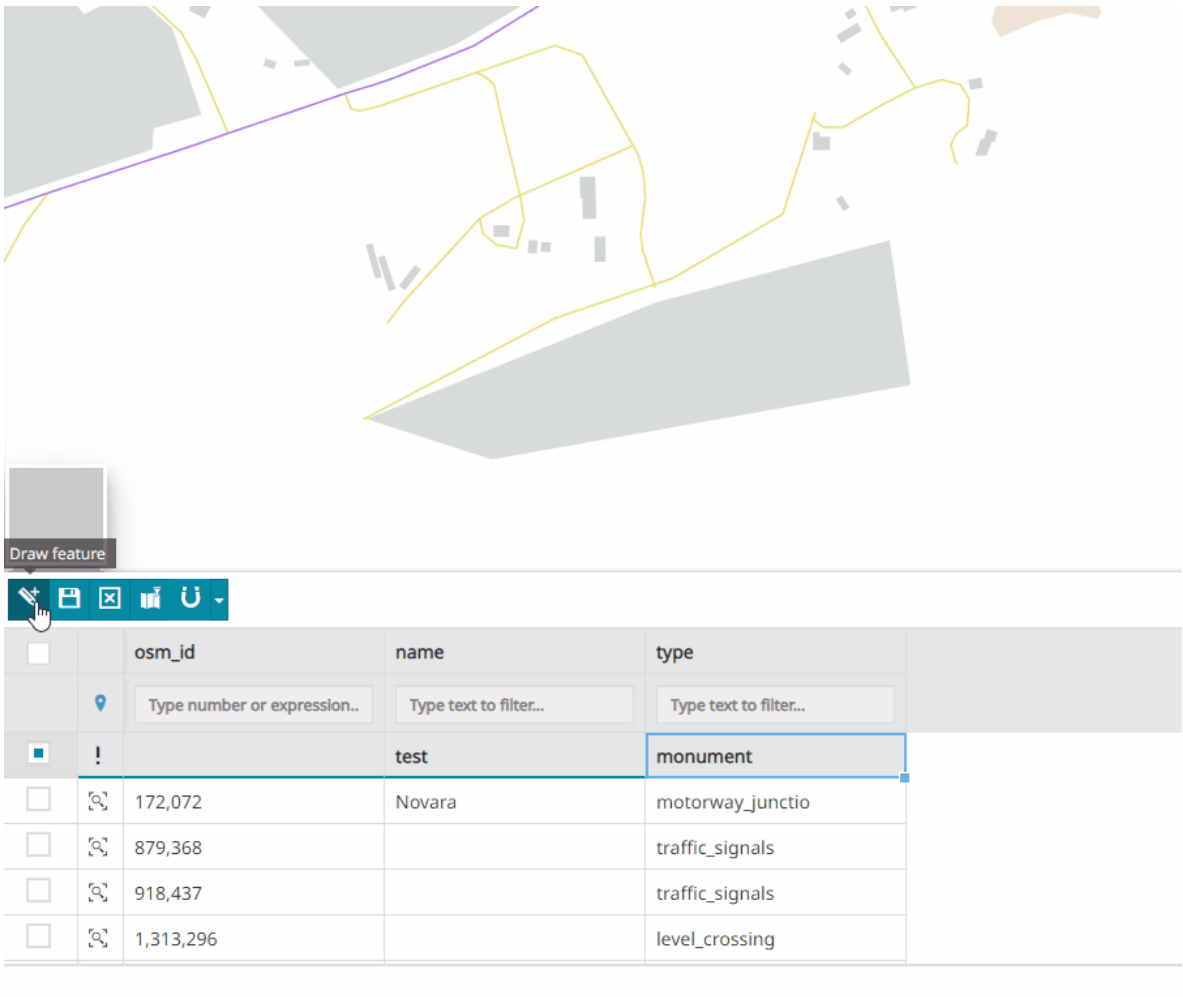
- For *Polygons* and *Multipolygons* layers, each click on the map add a new vertex (the minimum is 3). Once the vertex are set, it is possible to change the shape by creating new vertices or dragging the existing ones:



- For *Lines* and *Multilines* layers the shape drawing function works more or less in the same way. The only difference is that you need at least two vertices to draw a line and not three like for polygons:



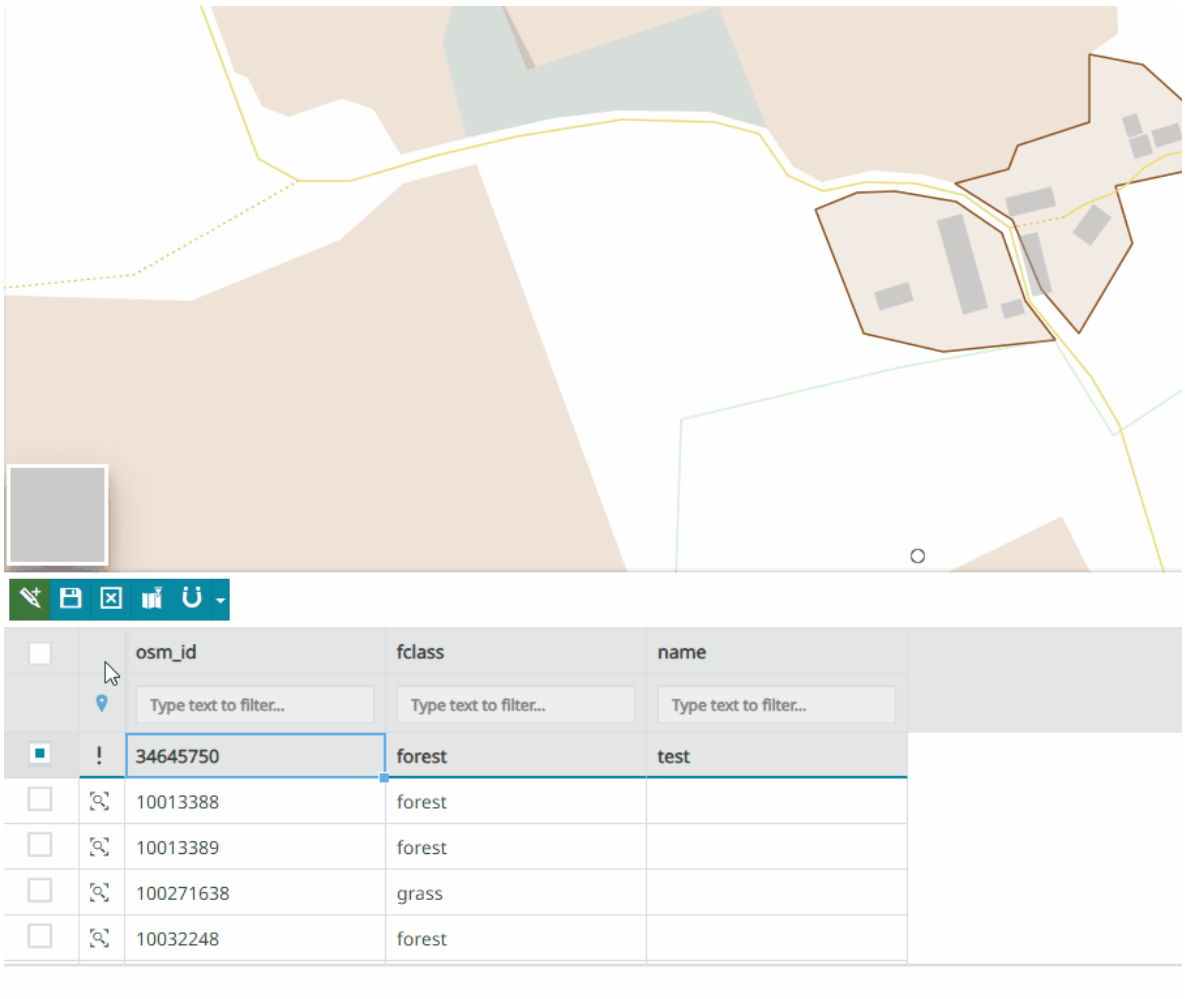
- For *Points* layers a point is drawn for each click on the map



The user is always allowed to delete the drawn shape through the button.

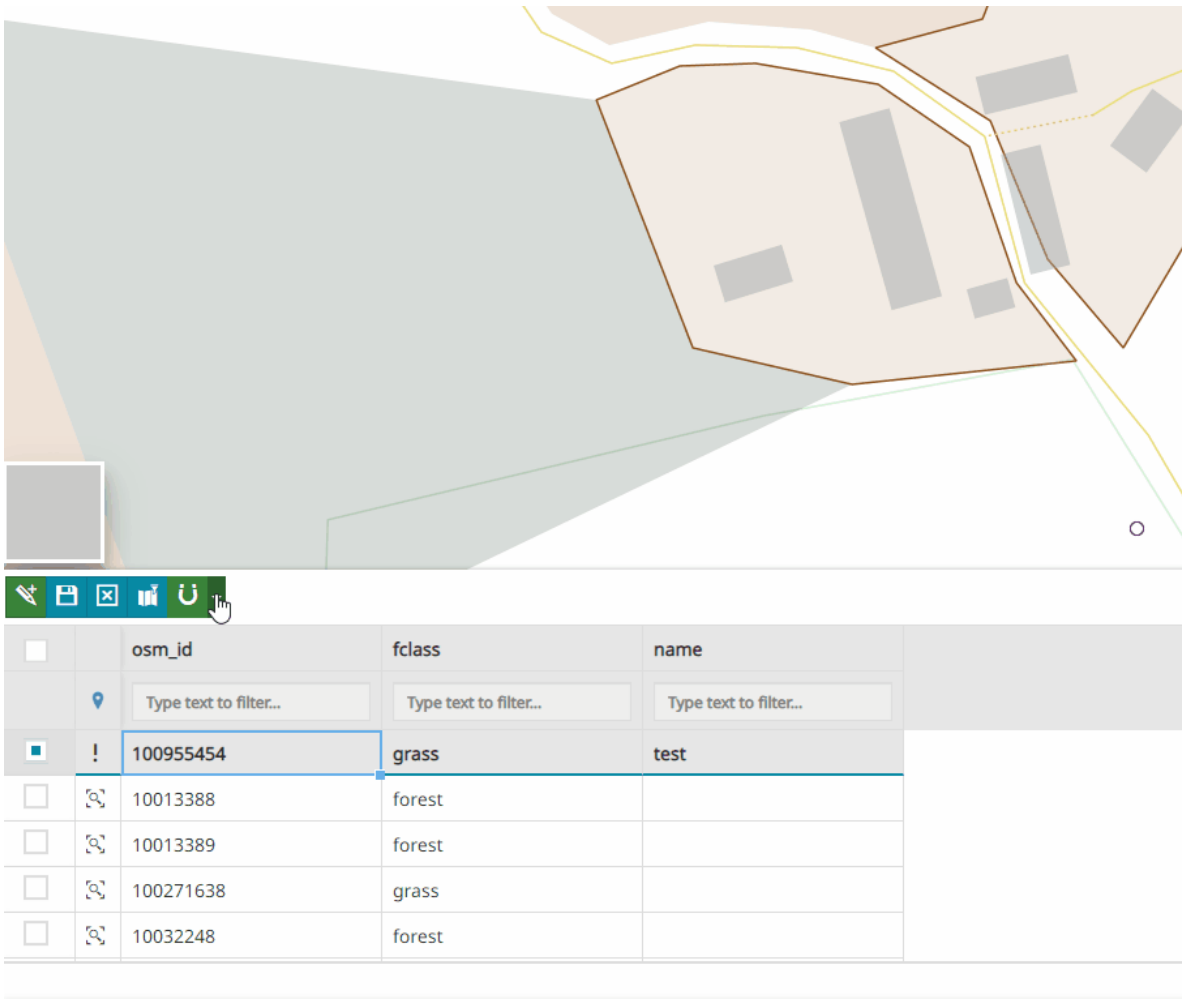
Create new geometry with Snapping

To fine tune the vertex position while editing or creating a new feature geometry, it is possible to leverage on the Snapping functionality. Through this function it is possible to snap to other vertices of features belonging to the same layer or to others while editing a feature.



The tool provides the ability to tune the snapping function so that the user can:

- Choose one of the visible map layers in TOC to be used for the snapping

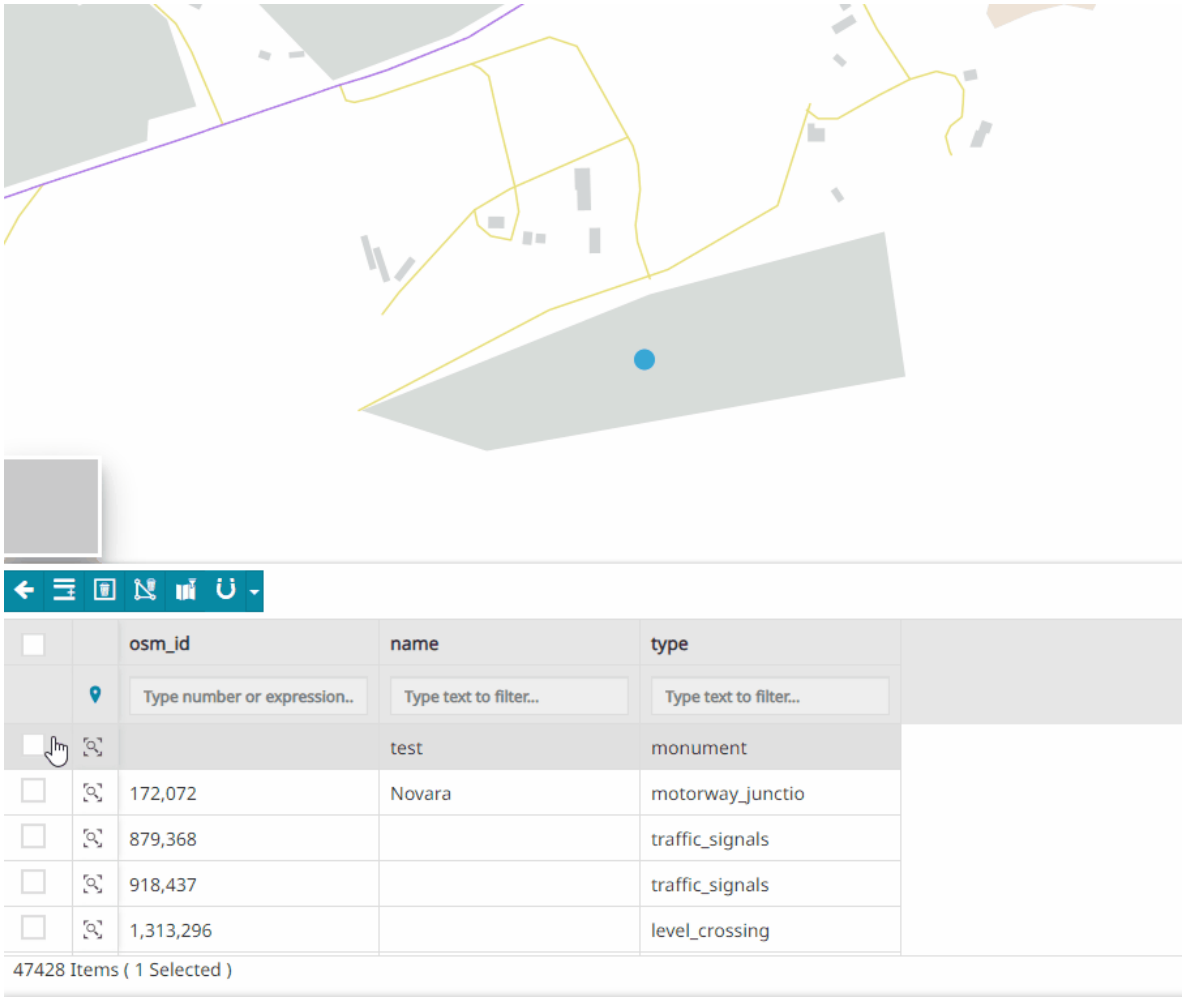


- Choose where to snap the layer, enabling/disabling the **Edge** or/and the **Vertex**
- Set **Tolerance** for considering the pointer close enough to a segment or vertex for snapping
- Choose the **Loading strategy** of features to snap with by choosing one of the available options from the dropdown menu. Available options are:
  - *bbox*: only features in the current viewport are loaded
  - *all*: all layer features are loaded

> **Note**
>
> The snapping functionality is by default set to work with the same layer in editing mode. By default, the **Edge** and the **Vertex** are enabled, the **Tolerance** is set to 10 `pixel` and the **Loading strategy** is set to *bbox*.

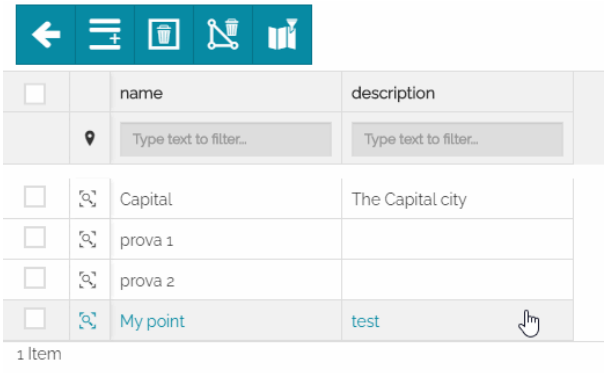
Editing and removing existing features

In order to edit an existing feature, it is necessary to switch the Attribute Table in editing mode by clicking the *Edit mode*  button. If the goal is to edit the Attribute Table records, the user can simply select them and type the desired value into the input field. However, it is also possible to modify the geometry associated with a record by editing it on the map (adding or changing its vertices).



Note

It is possible to edit the value of an attribute for multiple records at once by selecting the corresponding cell in the table and dragging the content onto the multiple cells, as follows:



With a click on **Save changes** these changes will be persistent.
In *Edit mode*, the user can also delete some features by selecting them in the table and clicking on the button.

Set filters

In the Attribute table it is possible to apply filters in three ways (as explained in the Filtering layers section):

- Advanced search
- Click on map
- Quick filter

Those filters, once applied, can be visible on the map by enabling the button.

Download the grid data

Form the Attribute table it is also possible to download the grid data through the button. The following window opens:

From this window it is possible to set:

- The **File Format** (`GML2` , `Shapefile` , `GeoJSON` , `KML` , `CSW` , `GML3.1` or `GML3.2`)
- The **Spatial Reference System** (by default `Native` or `WGS84`)

With a click on the Export button and the browser will download the file.

Customize Attribute table display

MapStore allows the user to customize the fields displayed in Attribute table mainly in two way:

- Ordering the records in alphabetic order (if it's a text field) or from the minimum to the maximum value and viceversa (if it's a numerical field):

states

state_name	state_fips	sub_region	state_abbr	land_km	w
Type text to filter...	Type text to filter...	Type text to filter...	Type text to filter...	Type number or expression...	
Illinois	17	E N Cen	IL	143986.61	1
District of Columbia	11	S Atl	DC	159.055	1
Delaware	10	S Atl	DE	5062.456	1
West Virginia	54	S Atl	WV	62384.2	3
Maryland	24	S Atl	MD	25316.345	6
Colorado	08	Mtn	CO	268659.501	9
Kentucky	21	E S Cen	KY	103961.904	1
Kansas	20	W N Cen	KS	211921.641	1
Virginia	51	S Atl	VA	102537.328	4
Missouri	29	W N Cen	MO	178445.951	2
Arizona	04	Mtn	AZ	294333.462	9
Oklahoma	40	W S Cen	OK	177877.536	3

49 Items

• Deciding which columns to show and which to hide through the  button:

States of US

STATE_NAME	STATE_FIPS	SUB_REGION	STATE_ABBR	LAND_KM
Digita testo da cercare...	Digita testo da cercare...	Digita testo da cercare...	Digita testo da cercare...	Digita numero o espressione.
Alabama	01	E S Cen	AL	131.443,119
Arizona	04	Mtn	AZ	294.333,462
Arkansas	05	W S Cen	AR	134.875,075
California	06	Pacific	CA	403.970,143
Colorado	08	Mtn	CO	268.659,501
Connecticut	09	N Eng	CT	12.549,571
Delaware	10	S Atl	DE	5.062,456
District of Columbia	11	S Atl	DC	159,055
Florida	12	S Atl	FL	139.852,123
Georgia	13	S Atl	GA	148.574,888

49 Elementi

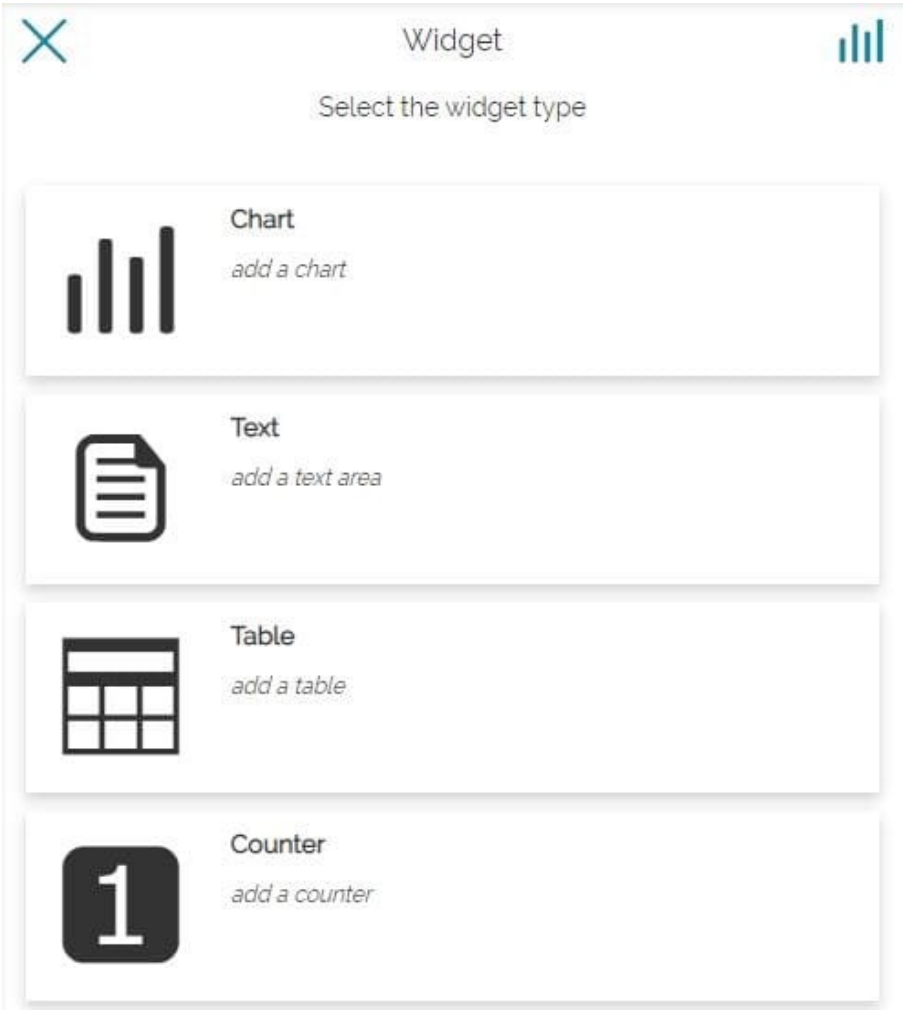
Widgets

In MapStore it is possible to create widgets from the layers added to the map. Widgets are components such as charts, texts, tables and counters, useful to describe and visualize qualitatively and quantitatively layers data and provide the user the opportunity to analyze information more effectively.

> **Note**
>
> Some widgets (in maps or in dashboards) need some WPS back-end support to work:
>
> - The `map widgets` (dashboards) needs the WPS process `gs:Bounds` to zoom to filtered data, if connected to a table.
> - For aggregate operations, `chart` and `counter` widgets need the WPS process `gs:Aggregate` available in GeoServer to work.

Add a Widget

Once at least one layer is present in the map (see Catalog section for more information about adding layers), it is possible to create a widget by selecting that layer in the TOC and by clicking on the  button from the Layer Toolbar or from the Attribute Table. Performing these operations the *Widget* panel appears:

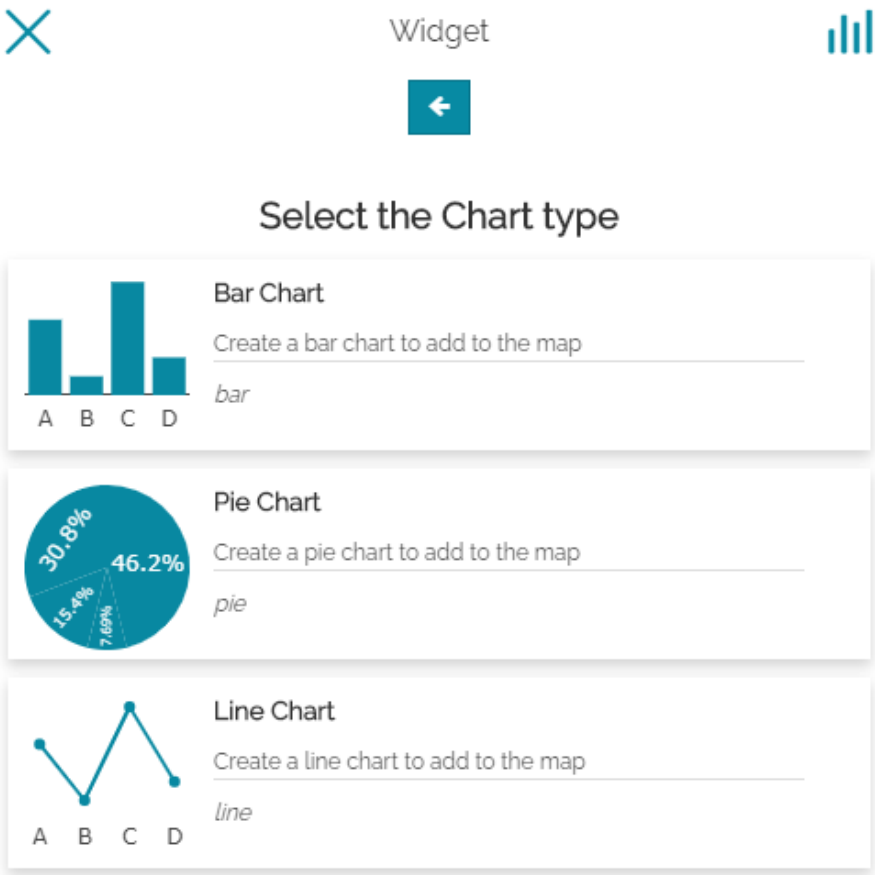


From here the user can choose between four different types of widget:

- *Chart*
- *Text*
- *Table*
- *Counter*

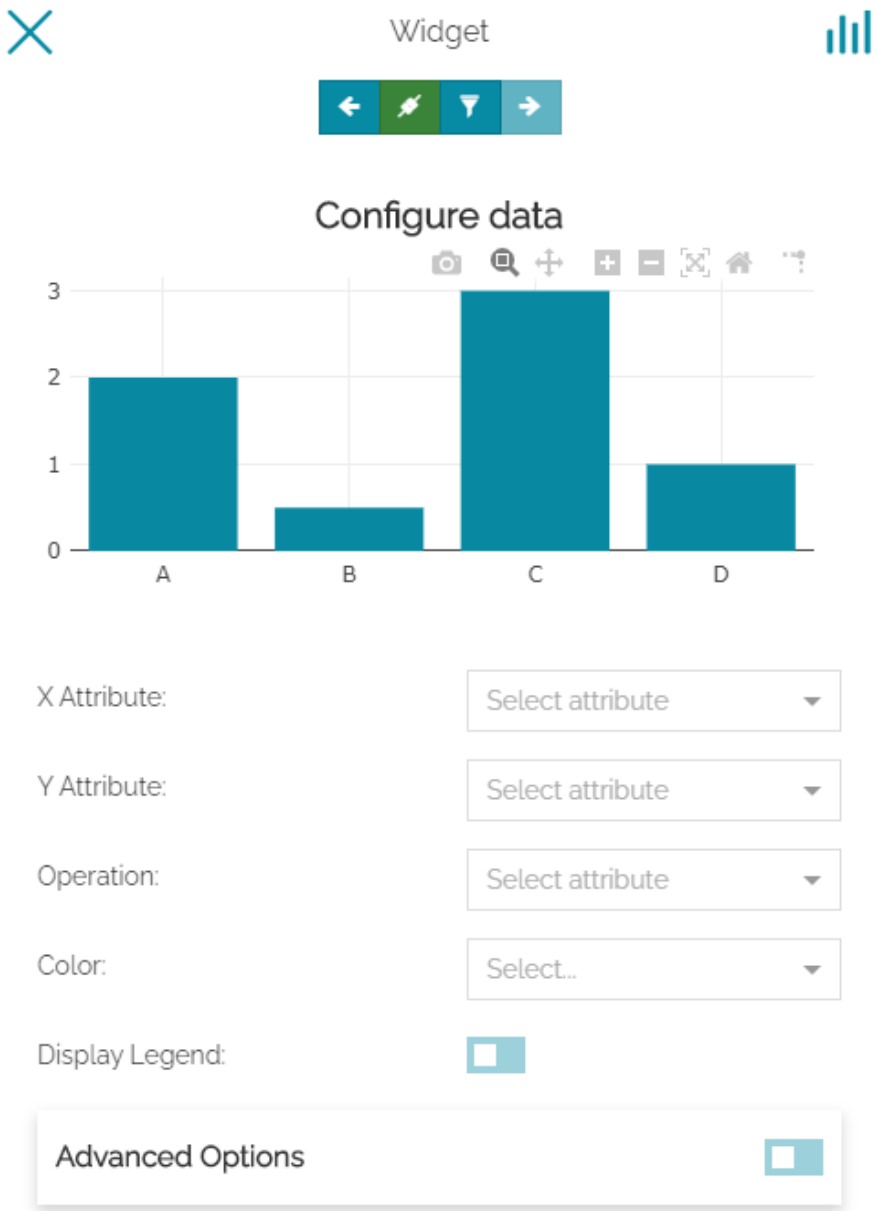
Chart

Selecting *Chart* option the following window opens:



From here it is possible to choose between *Bar Chart*, *Pie Chart* or *Line Chart*, or simply go back to widget type selection through the button.

If a chart type is selected, it can display similar the following (in this case a *Bar Chart*):



From the toolbar of this panel the user is allowed to:

- Go back to the chart type selection with the button
- Connect or disconnect the widget to the map. When a widget is connected to the map, the information displayed in the widget are automatically filtered with the map viewport. When a widget is not linked, it otherwise shows all the elements of that level regardless of the map viewport
- Configure a filter for the widget data (more information on how to configure a filter can be found in Filtering Layers section)

- Move forward  to the next step when the settings are completed

Just below the chart's preview, the following configurations are available:

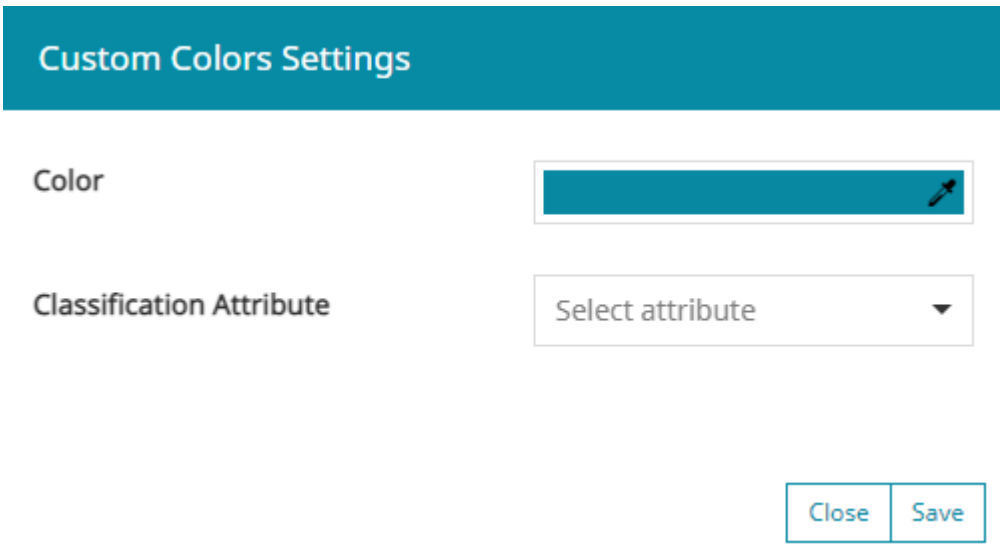
- Define the **X Attribute** of the chart (or **Group by** for *Pie Charts*) choosing between layer fields
- Define the **Y Attribute** of the chart (or **Use** for *Pie Charts*) choosing between layer fields
- Define the aggregate **Operation** to perform for the selected attribute choosing between `No Operation`, `COUNT`, `SUM`, `AVG`, `STDDEV`, `MIN` and `MAX`

> **Note**
>
> The *No operation* option is used when the aggregation method is not needed for the chart. If *No Operation* is selected, no aggregation will be carried out for the chart and the WFS service will be used to generate the chart without using the WPS process `gs:Aggregate` in GeoServer.

- Enable the chart's legend by activating **Display Legend**
- Choose the **Color** (`Blue`, `Red`, `Green`, `Brown` or `Purple`) of the chart (or the **Color Ramp** for *Pie Charts*) or choose to **Customize the color**.

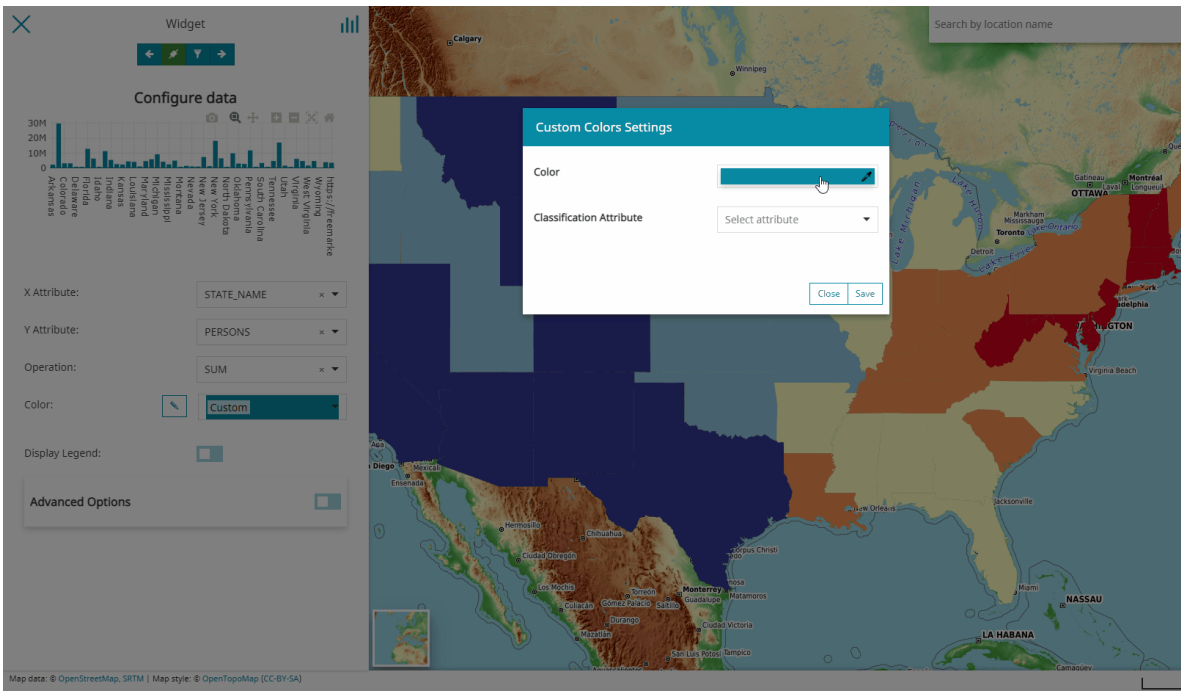
Color customization

For *Bar Charts* and *Pie Charts*, MapStore provides the possibility to customize the colors of the charts bars and slices. From the **Color** option dropdown menu, the user can select the *Custom* option and open the **Custom Colors Settings** modal through the  button.

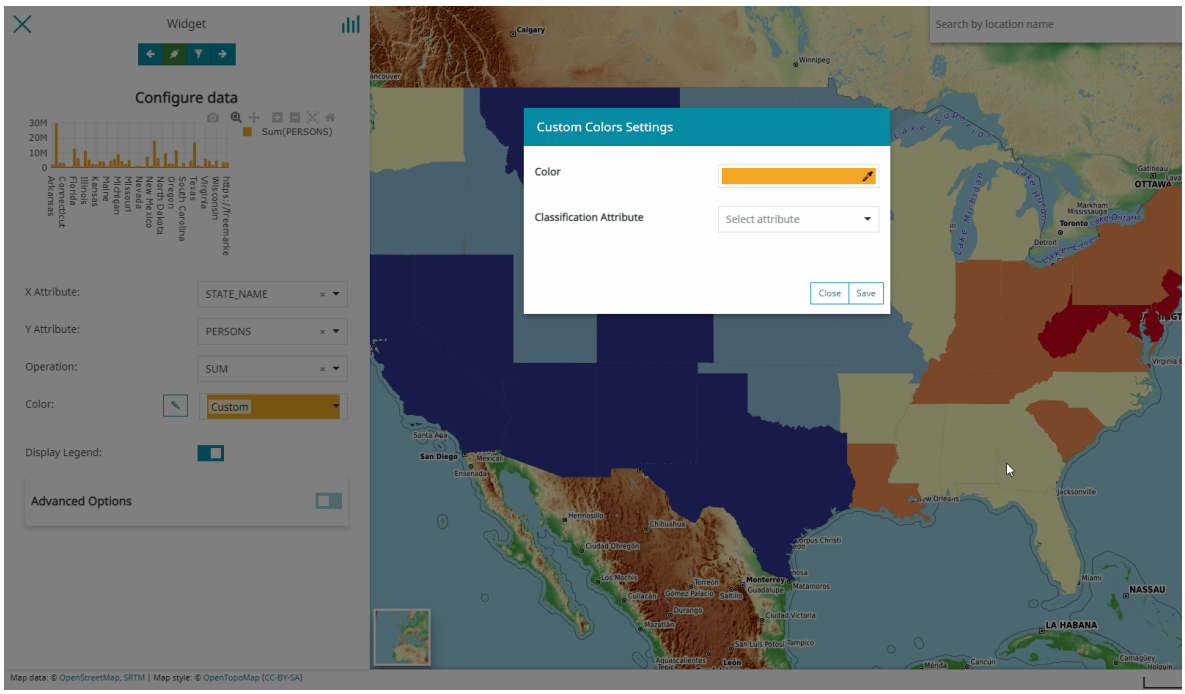


Inside this modal, the user is allowed to:

- Change the default **Color** of bars or slices (depending on the chart type) through the *Color Picker*. This color will be applied for all values for which a *Class Color* has not been configured.

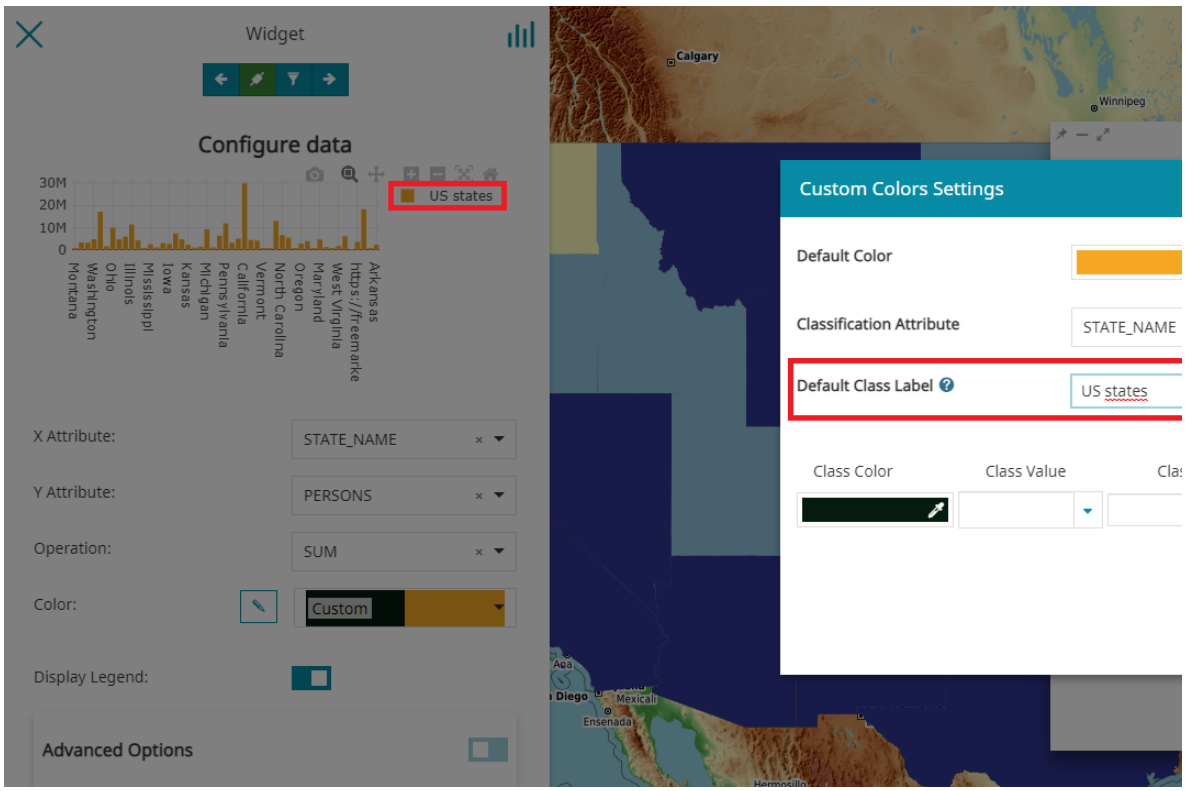


- Select an *Attribute* in the dropdown list as a **Classification attribute**.



Once the attribute is chosen, new options appear in the *Custom Color Settings* panel that allow the user to:

- Enter a **Default Class Label** to be used in the legend for all values that will not be specifically classified in the following list.

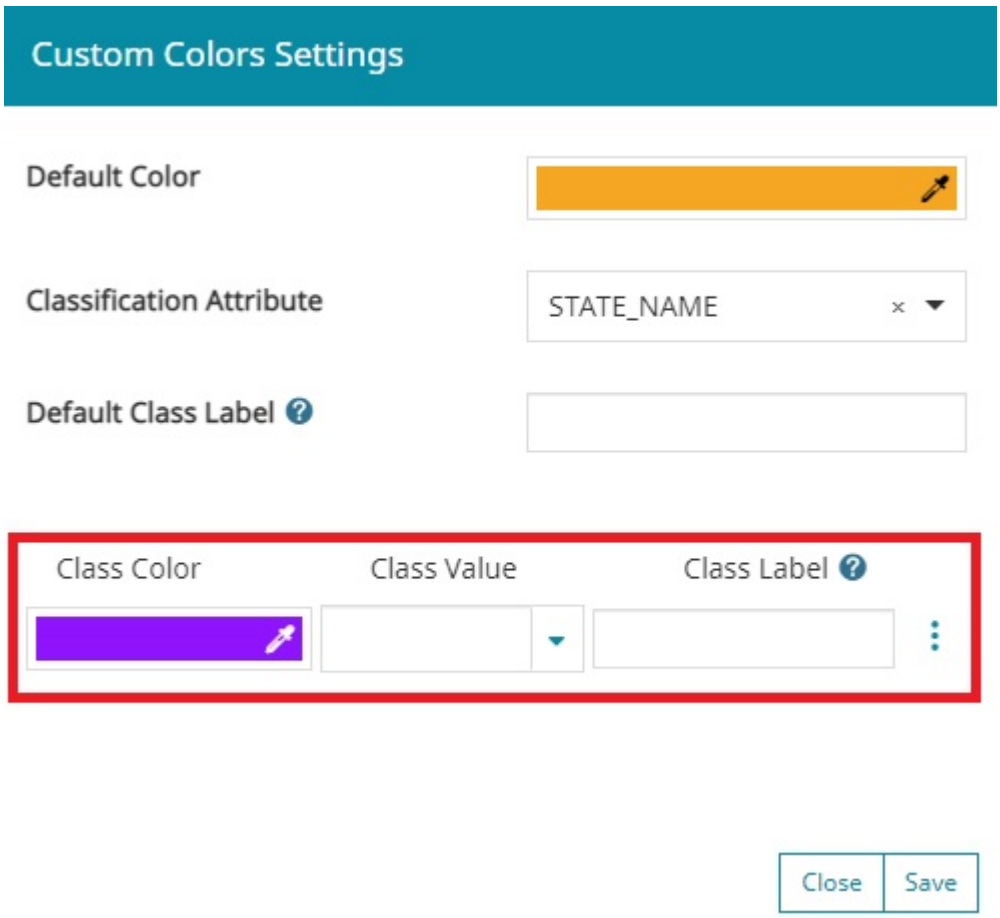


> **Note**
>
> For both *Default Class Label* and *Class Label* '${legendValue}' can be used as a placeholder for the Y Attribute (that can be further customized through the usual *Advanced Option*).

- Classify *Classification Attribute* values to assign a specific color in the chart along with its *Class Label* to use for the chart legend. Only values of type **String** or **Number** are currently supported.

Classification Attribute of type String

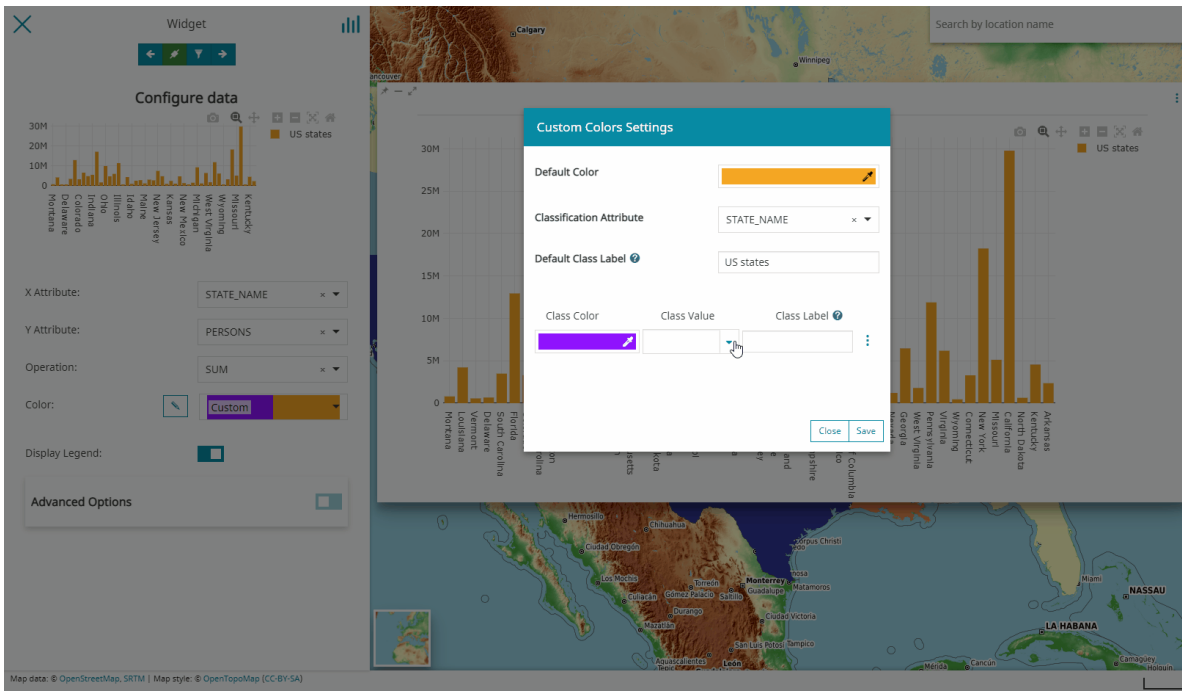
When the values of a classification attribute are of type String, the user can:



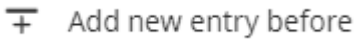
- Choose the **Class Color** through the *Color Picker*.
- Choose the value of the *Classification attribute* through the dropdown menu **Class Value**
- Enter a **Class Label** to be used in the legend for the value entered in the *Class Value*

> **Note**
>
> For *Class Label*, '${legendValue}' can be used as a placeholder for the Y Attribute (that can be further customized through the usual *Advanced Option*).

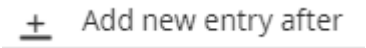
An example of *Bar charts* corresponding to this type of classification can be the following:



Through the ⋮ button the user can add new values before through the



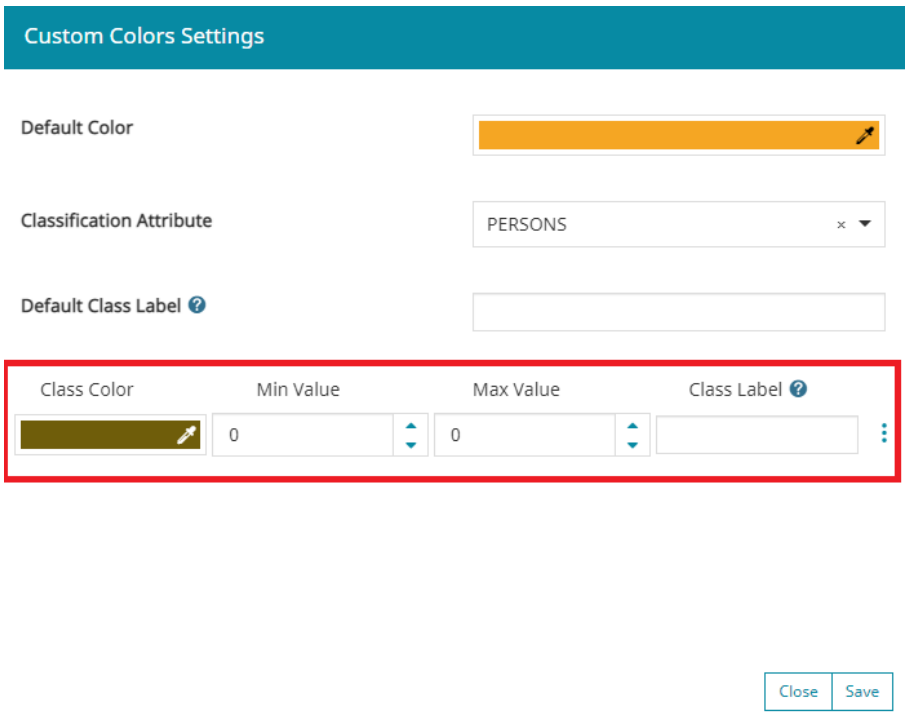
button or after through



button.

Classification Attribute of type Number

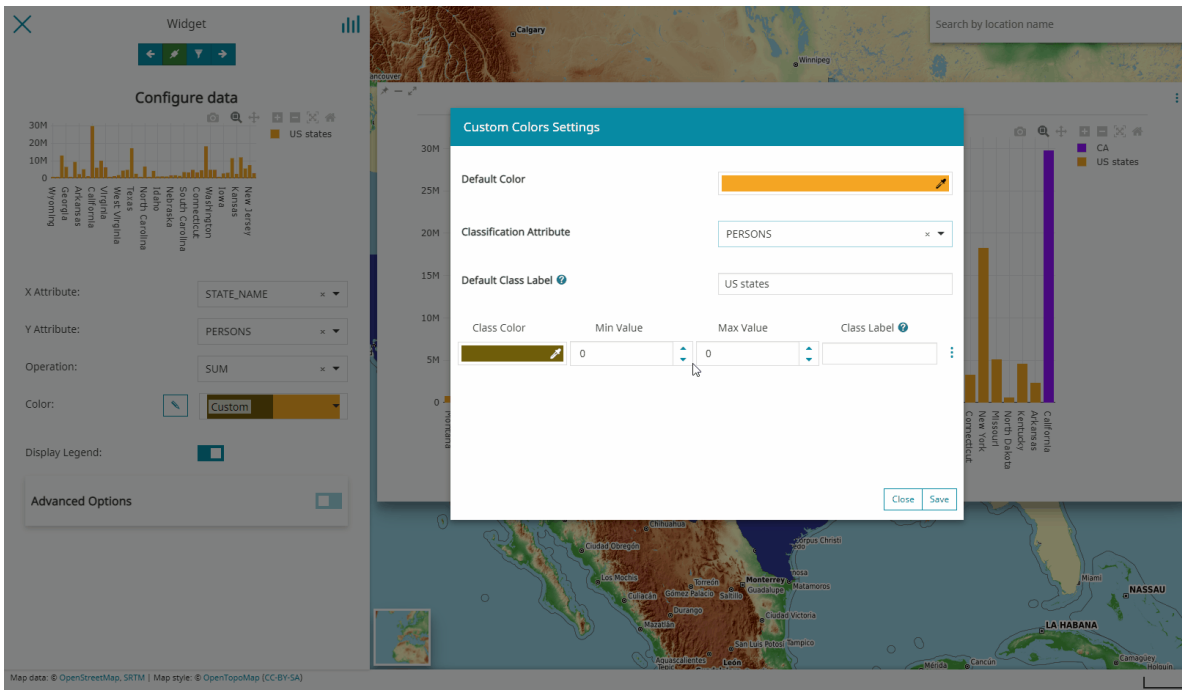
When the values of a classification attribute are numbers, the user can configure a color ramp and so:



- Choose the **Class Color** through the *Color Picker*
- Choose the **Min value** of the *Classification attribute*
- Choose the **Max value** of the *Classification attribute*
- Enter a **Class Label** to be used in the legend for the value entered in the *Class Value*

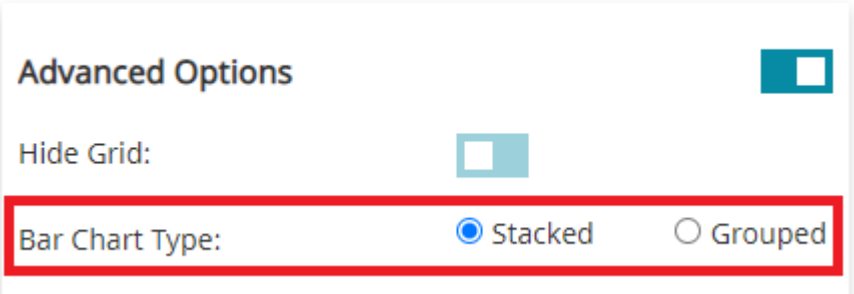
> **Note**
>
> For *Class Label* two placeholders can be used in this case: ${minValue}, can be used as a placeholder for **Min Value** and ${maxValue}, can be used as a placeholder for Max Value; the ${legendValue} can be used in the same way as specified above.

An example of Bar chart corresponding to this type of classification can be the following:



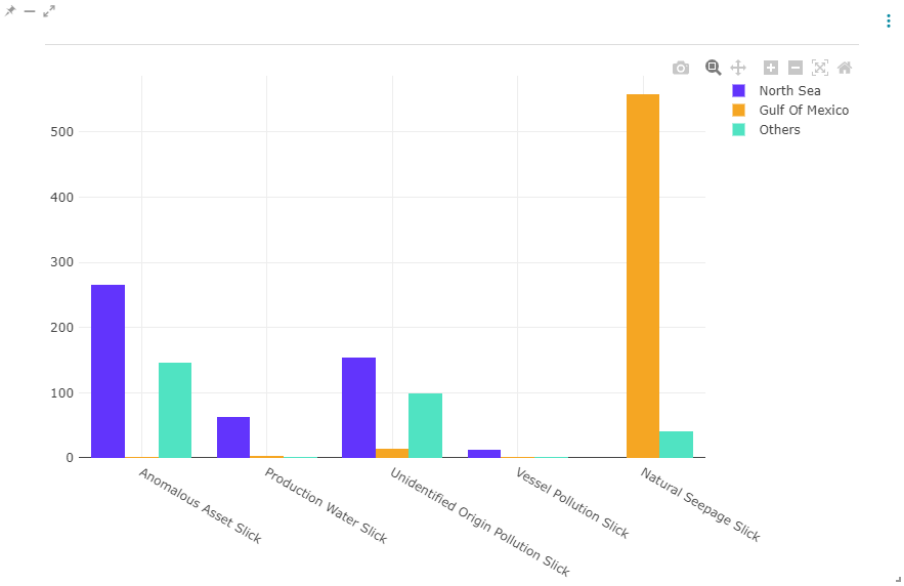
Bar Chart Type

If the *Classification attribute* is added to the *Bar Chart*, in the Advanced Options, the **Bar Chart Type** option is displayed.

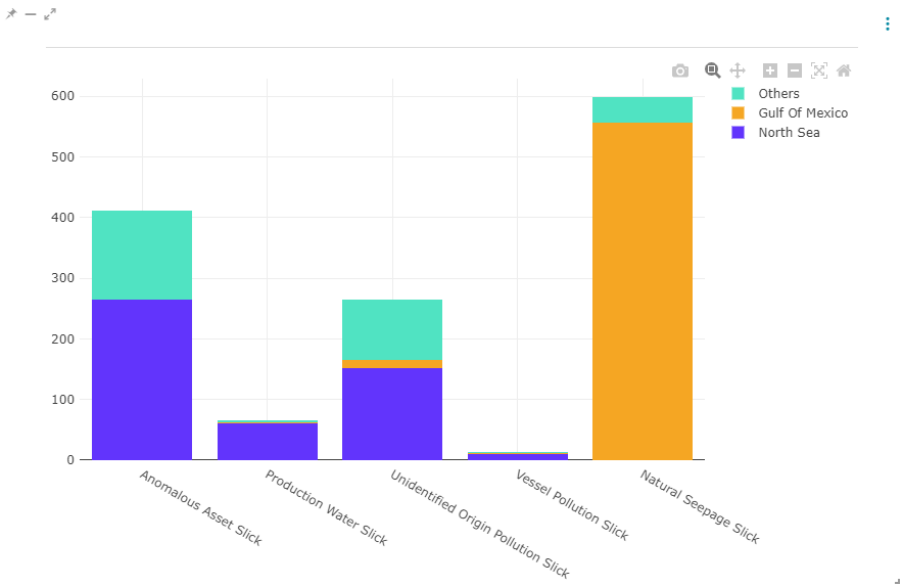


The user can customize the bars by choosing between:

- **Grouped**. An example can be the following:



- **Stacked**. An example can be the following:

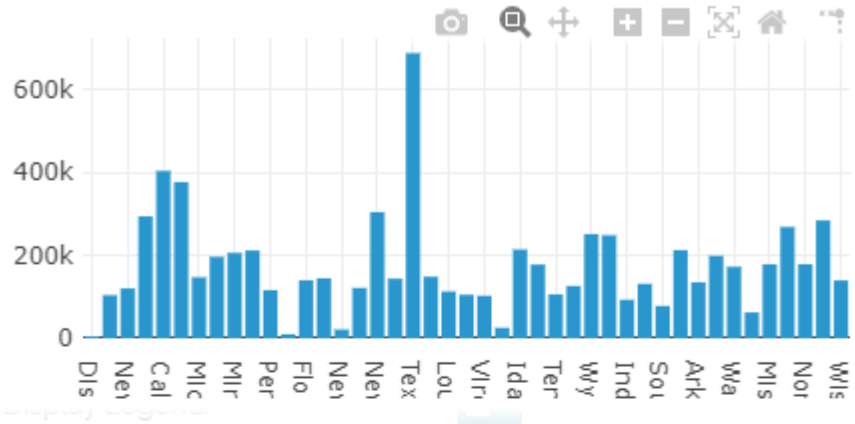


> **Note**
>
> By default, the bar chart type is **Stacked**

Advanced Options

In addition, only for *Bar Charts* and *Line Charts*, MapStore provides advanced setting capabilities through the *Advanced Options* section.

Widget



Advanced Options

Hide Grid:

Y axis

Type: Auto

Hide labels:

Format:

Prefix: e.g.: ~

Format: e.g.: .2s

Suffix: e.g.: W

Formula:

e.g. value / 100

X axis

Type: Auto

Hide labels:

Never skip labels:

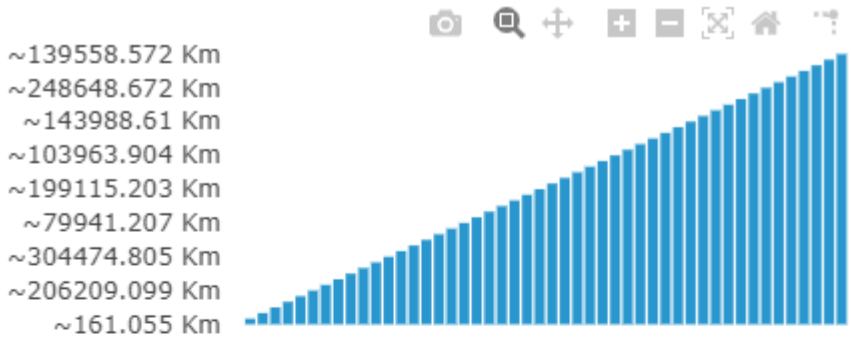
Label rotation: Auto

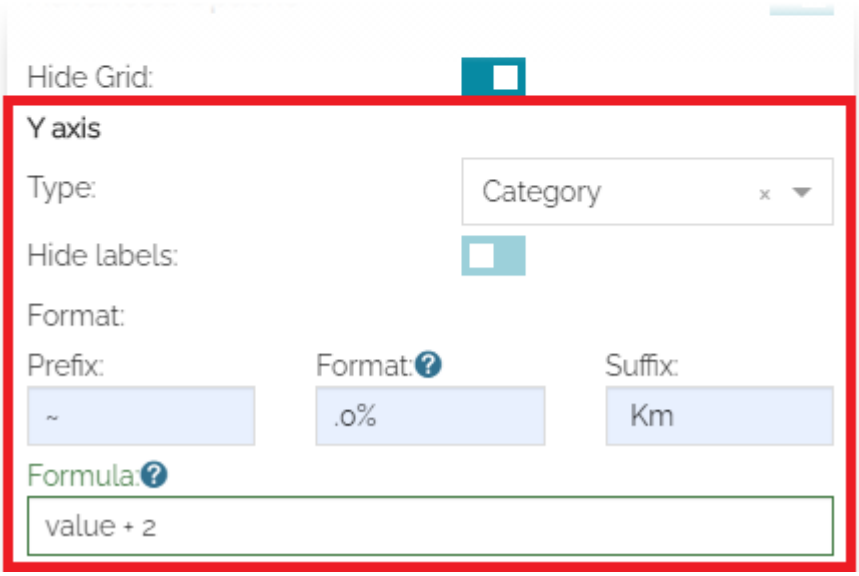
Legend

Legend Label:

Through this section, the user is allowed to:

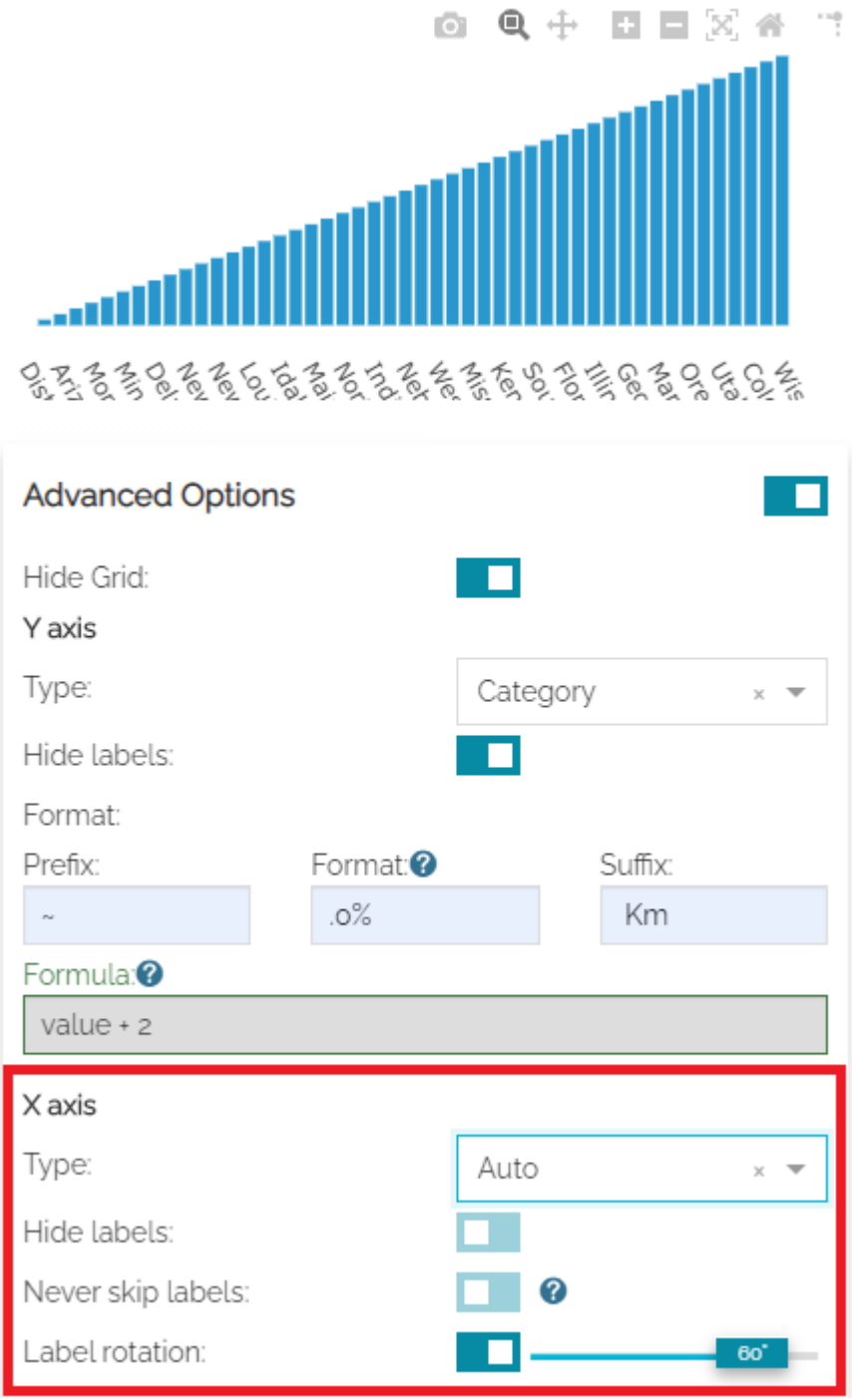
- Show/Hide the chart's grid in backgroung with the **Hide Grid** control
- Customize **Y axis** tick values by choosing the *Type* (between `Auto`, `Linear`, `Category`, `Log` or `Date`): the axis type is auto-detected by looking at data (*Auto* option is automatically managed and selected by the tool and it is usually good as default setting). The user can also choose to completely hide labels through the *Hide labels* control or customize them by adding a *Prefix* (e.g. `~`), a custom *Format* (e.g. `0%: rounded percentage, '12%'` or more) or a *Suffix* (e.g. `Km`). It is also possible to configure a *Formula* to transform tick values as needed (e.g. `value + 2` or `value / 100` or more)





> **Note**
>
> More information about the syntax options allowed for **Format** are available here and the allowed expression to be used as **Formula** are available here in the online documentation.

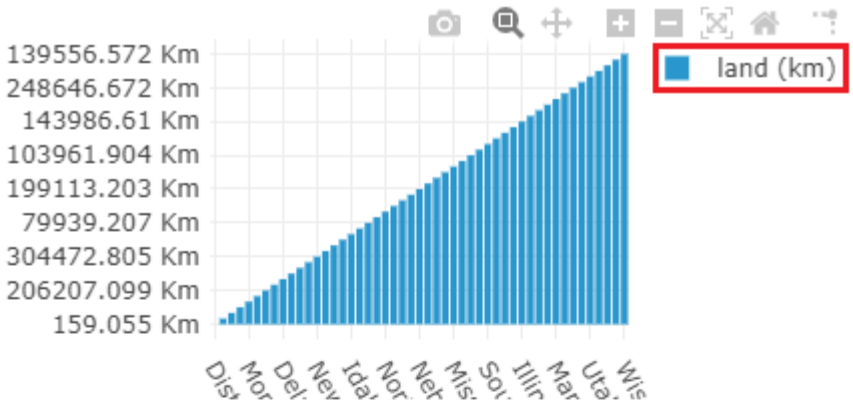
- Customize **X axis** tick values by choosing the *Type* (between `Auto`, `Linear`, `Category`, `Log` or `Date`): the axis type is auto-detected by looking at data (*Auto* option is automatically managed and selected by the tool and it is usually good as default setting). As per **Y axis**, the user can completely hide labels through the *Hide labels* control or tune the rendering of tick labels with options like *Never skip labels* (it forces all ticks available in the chart to be rendered instead of simplifying the provided set based on chart size) and *Label rotation* to better adapt X axis tick labels on the charts depending on the needs.



Warning

The tick labels available for the X axis by enabling the option **Never skip label** cannot be more than 200 in order to provide a clear chart and for performance reasons.

- Set the **Legend Label** name



Hide Grid:

Y axis

Type: Category

Hide labels:

Format:

Prefix: e.g.: ~

Format: e.g.: .2s

Suffix: Km

Formula:

e.g. value / 100

X axis

Type: Auto

Hide labels:

Never skip labels:

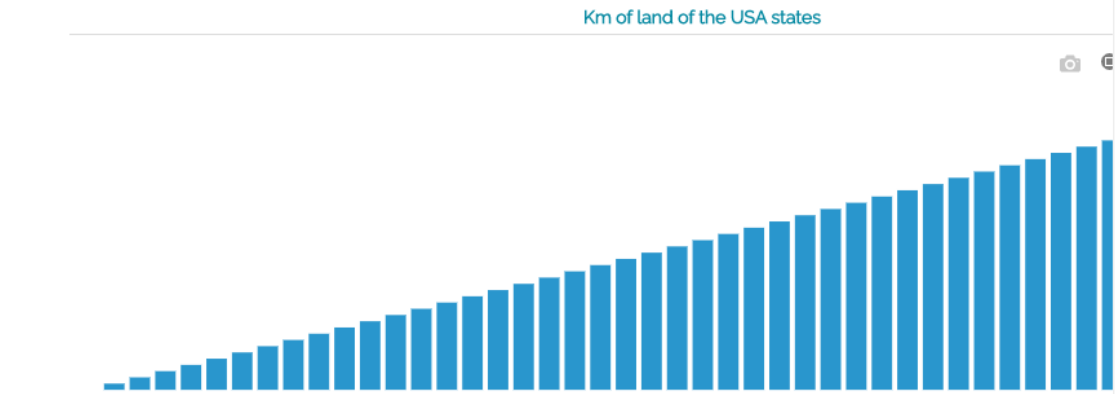
Label rotation: 60°

Legend

Legend Label: land (km)

Note

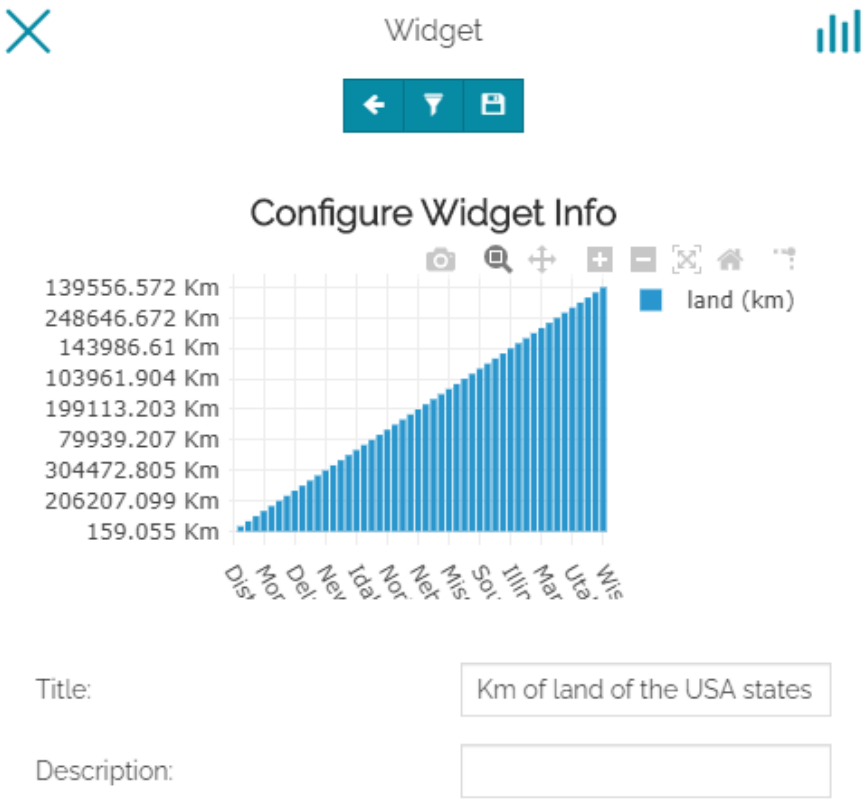
The tooltips of the X and Y axis labels are available by hovering the mouse over the charts. This way the labels are available even if the **Hide labels** option for the X and Y axis is enabled.



Warning

In order to move forward to the next step, only **X Attribute**, **Y Attribute** and **Operation** are considered as mandatory fields.

Once the settings are done, the next step of the chart widget creation/ configuration is displayed as follows:



The user can:

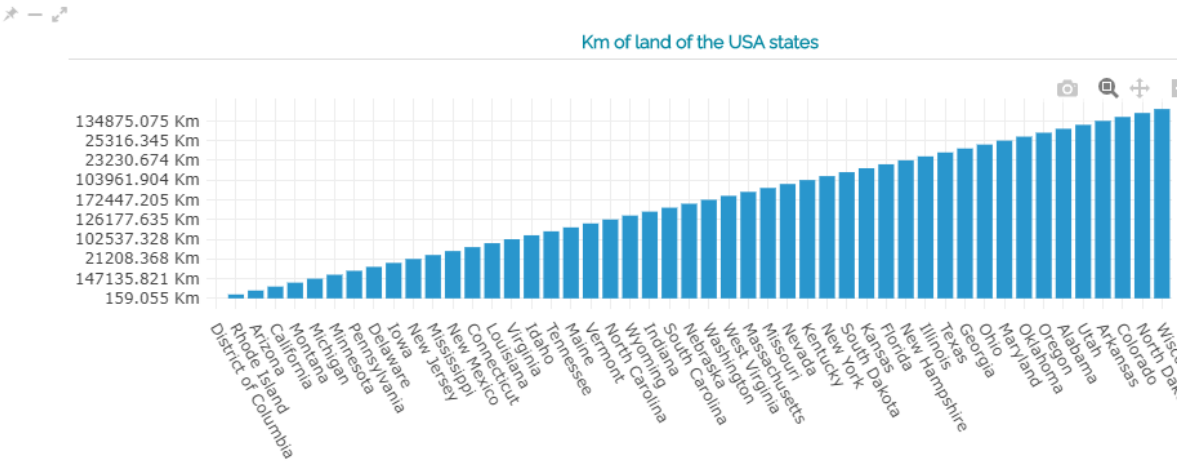
- Go back to the chart option with the  button
- Configure a filter  for the widget data (more information on how to configure a filter can be found in Filtering Layers section)
- Add the widget to the map with the  button

Just below the chart's preview, the user is allowed to set:

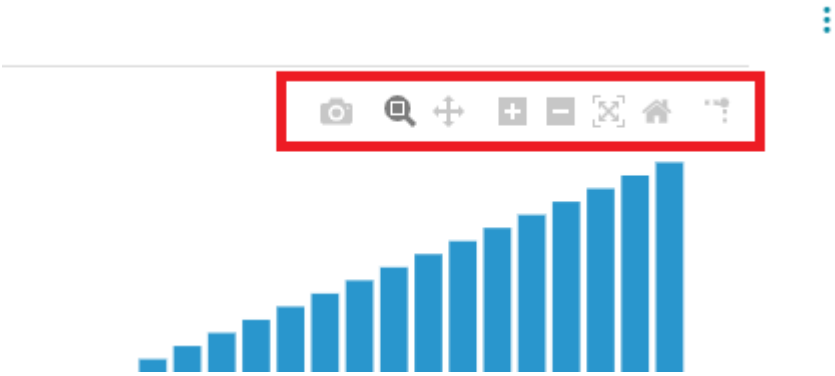
- The widget **Title**
- The widget **Description**

> **Note**
>
> None of these fields are mandatory, it is possible to save/add the widget to the map without filling them.

An example of chart widget could be:



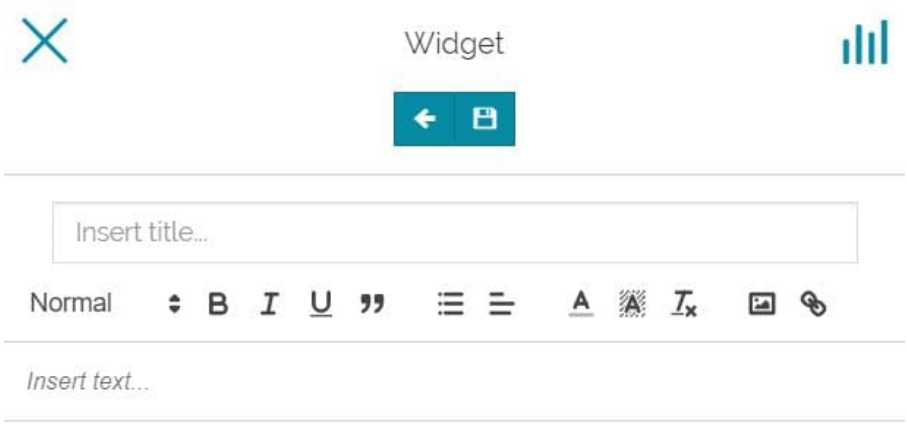
The **Chart toolbar**, displayed in the right corner of the chart allows the user to:



- **Download** the chart as a `png` through the button.
- **Zoom** the chart through the button.
- **Pan** the chart through the button.
- **Zoom in** the chart through the button.
- **Zoom out** the chart through the button.
- **Autoscale** to autoscale the axes to fit the plotted data automatically through the button.
- **Reset axes** to return the chart to its initial state through the button.
- **Toggle Spike Lines** to show dashed lines for X and Y values by hovering the mouse over the chart. This is useful to better see domain values on both axis in case of complex charts. It is possible to activate that option through the button.

Text

Creating a new text widget the following window opens:



Through the toolbar it is possible to:

- Go back to the widget type selection with the  button
- Add the widget to the map with the  button

Here the user can:

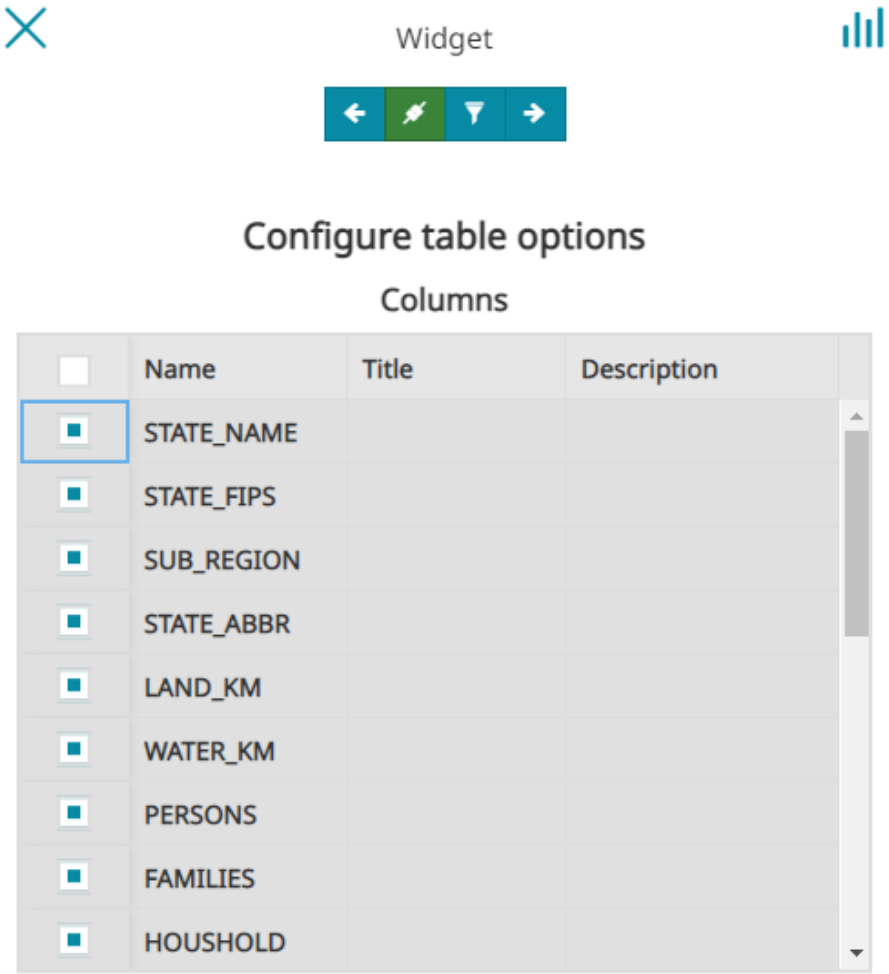
- Write the title of the widget
- Write the text of the widget
- Format the text through the Text Editor Toolbar

> **Note**
>
> None of these options are mandatory, you can add the widget to the map without filling in these fields.

An example of text widget could be:

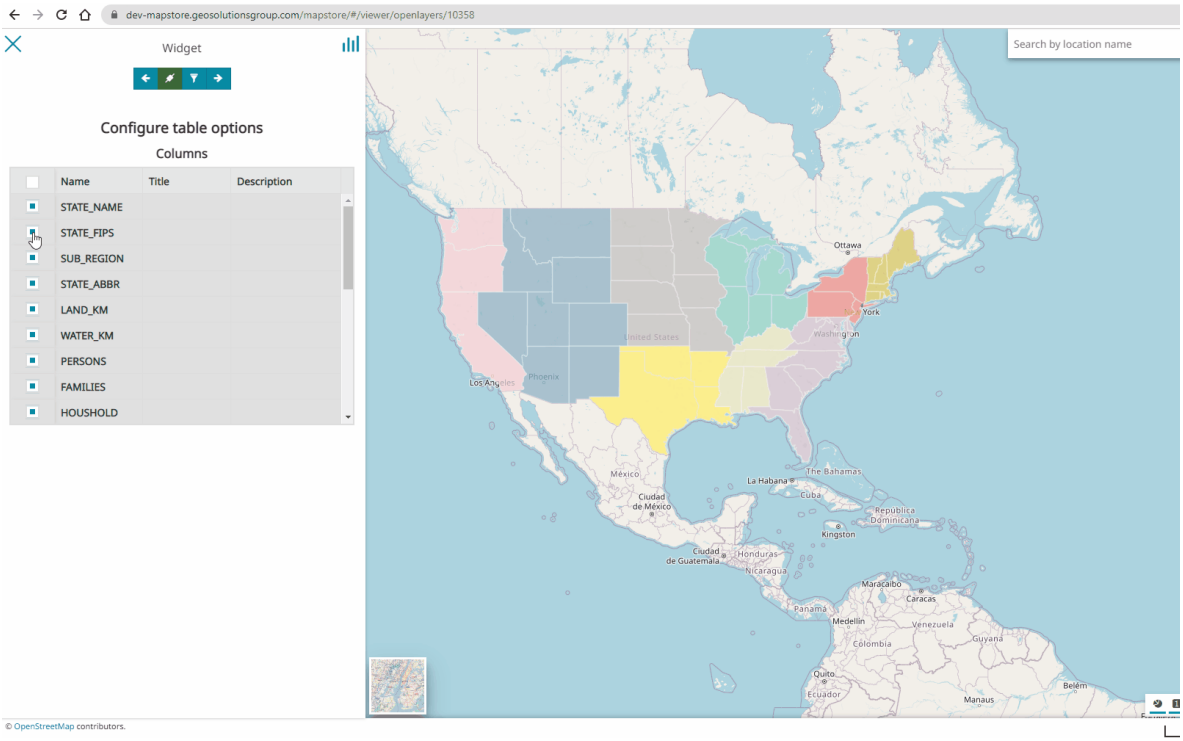
Table

Adding a table widget to the map, a panel like the following opens:



The toolbar on the top of this panel is similar to the one present in Chart section. Here the user is allowed to:

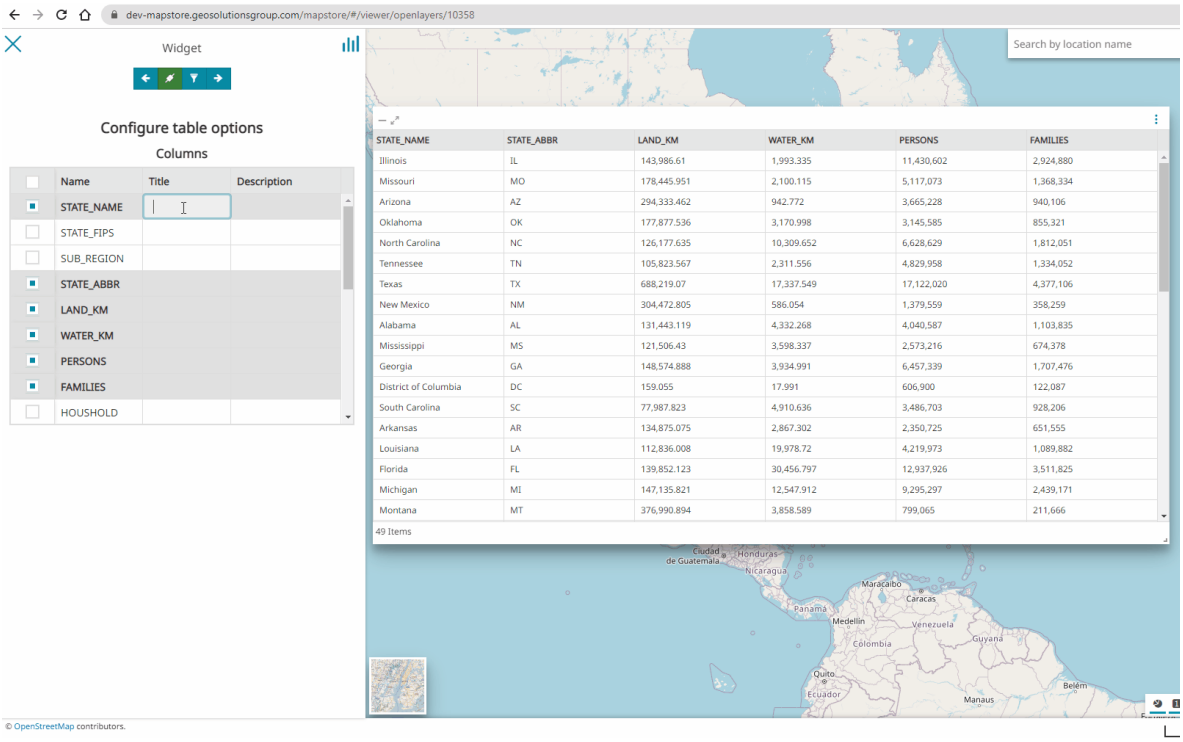
- **Enable/Disable** the layer fields that will be displayed in the widget as columns.



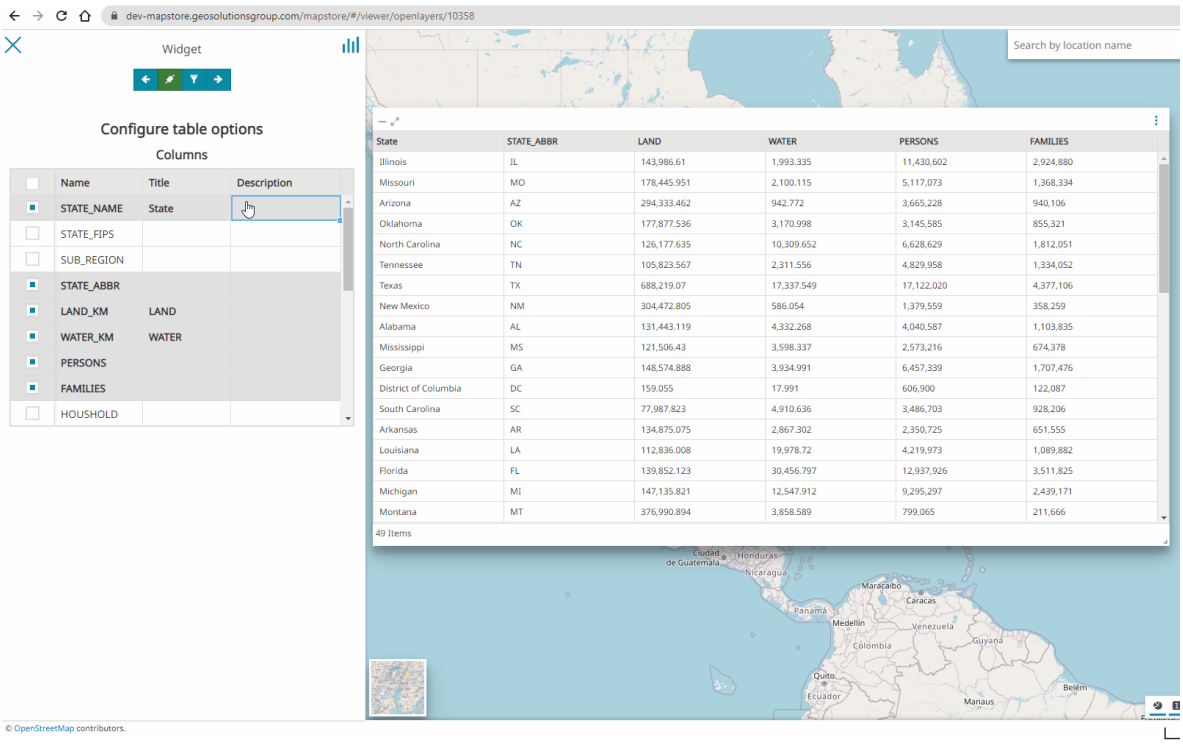
Warning

At least one field must be selected in order to move to the next configuration step.

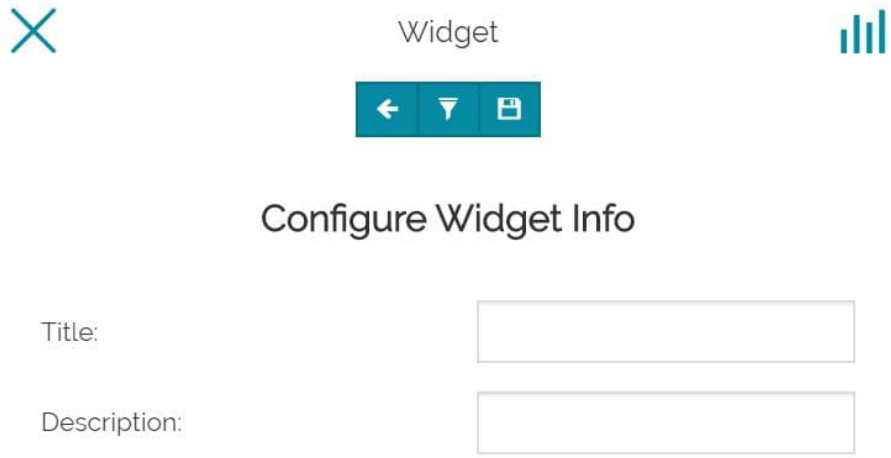
- Enter a **Title** for each column to be displayed as the table header in place of the *Name* of the layer field



- Enter a **Description** for each field to be displayed as a tooltip, visible moving the mouse on the column header.



Once the desired fields are selected, a click on the  button opens the following panel:



In this last step of the widget creation, the toolbar and the information to be inserted are similar to the ones in Chart section.
An example of table widget could be:

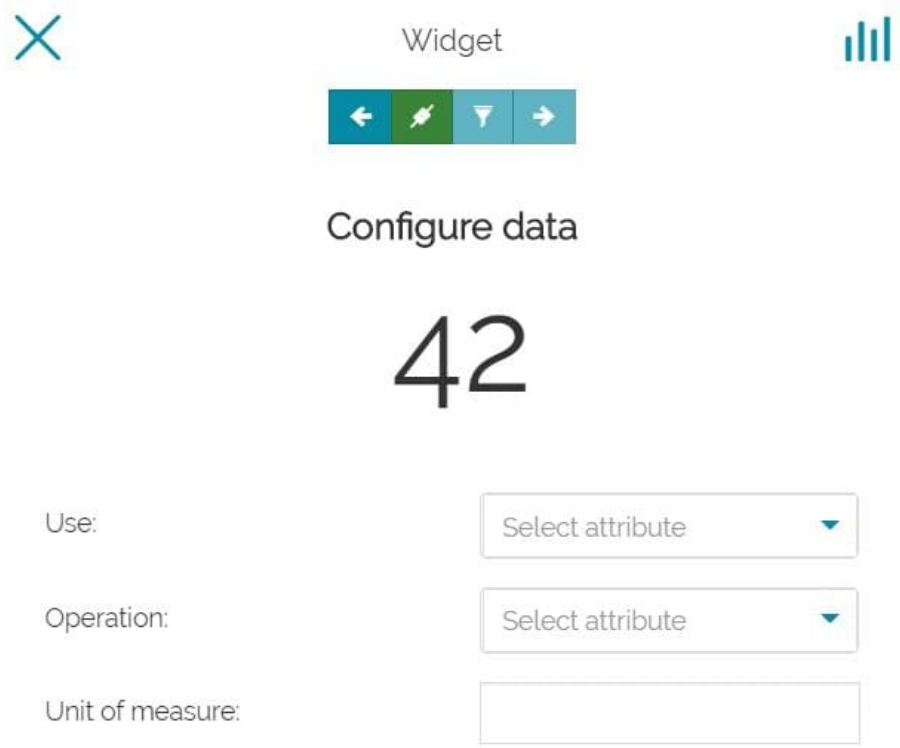
US state

State	LAND	WATER	PERSONS	FAMILIE
Illinois	143,986.61	1,993.335	11,430,602	2,924,8
Missouri	178,445.951	2,100.115	5,117,073	1,368,3
Arizona	294,333.462	942.772	3,665,228	940,106
Oklahoma	177,877.536	3,170.998	3,145,585	855,321
North Carolina	126,177.635	10,309.652	6,628,629	1,812,0
Tennessee	105,823.567	2,311.556	4,829,958	1,334,0
Texas	688,219.07	17,337.549	17,122,020	4,377,1
New Mexico	304,472.805	586.054	1,379,559	358,259
Alabama	131,443.119	4,332.268	4,040,587	1,103,8
Mississippi	121,506.43	3,598.337	2,573,216	674,378
Georgia	148,574.888	3,934.991	6,457,339	1,707,4
District of Columbia	159.055	17.991	606,900	122,087
South Carolina	77,987.823	4,910.636	3,486,703	928,206
Arkansas	134,875.075	2,867.302	2,350,725	651,555
Louisiana	112,836.008	19,978.72	4,219,973	1,089,8
Florida	139,852.123	30,456.797	12,937,926	3,511,8
Michigan	147,135.821	12,547.912	9,295,297	2,439,1
Montana	376,990.894	3,858.589	799,065	211,666

49 Items

Counter

Selecting the counter option, the following window opens:

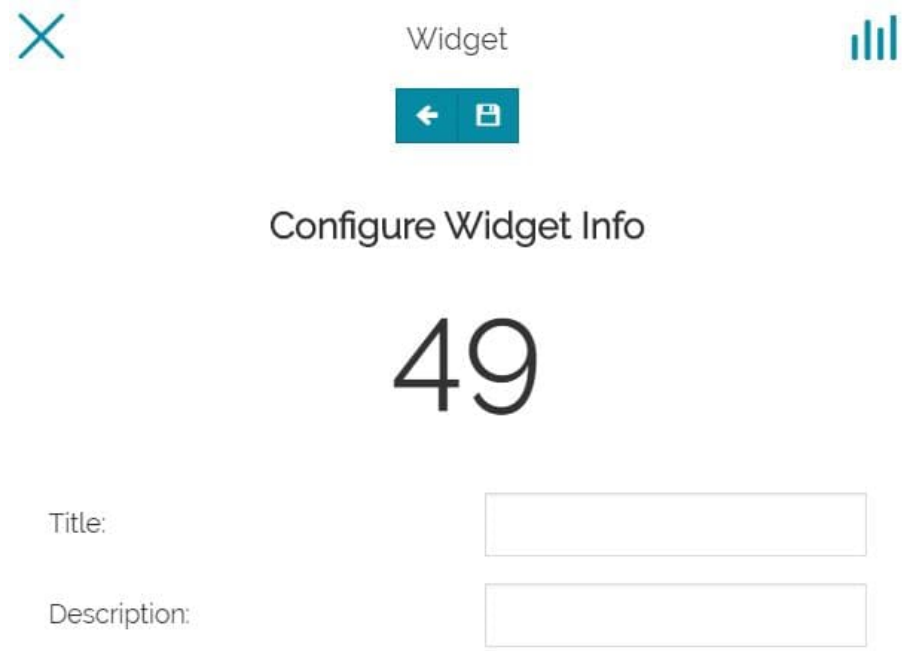


Also in this case the toolbar is similar to the one present in Chart section. The user is allowed to:

- Select the attribute to **Use**
- Select the **Operation** to perform
- Set the **Unit of measure** that will be displayed

> **Warning**
>
> In order to move forward to the next step, only the **Use** and the **Count** are considered as mandatory fields.

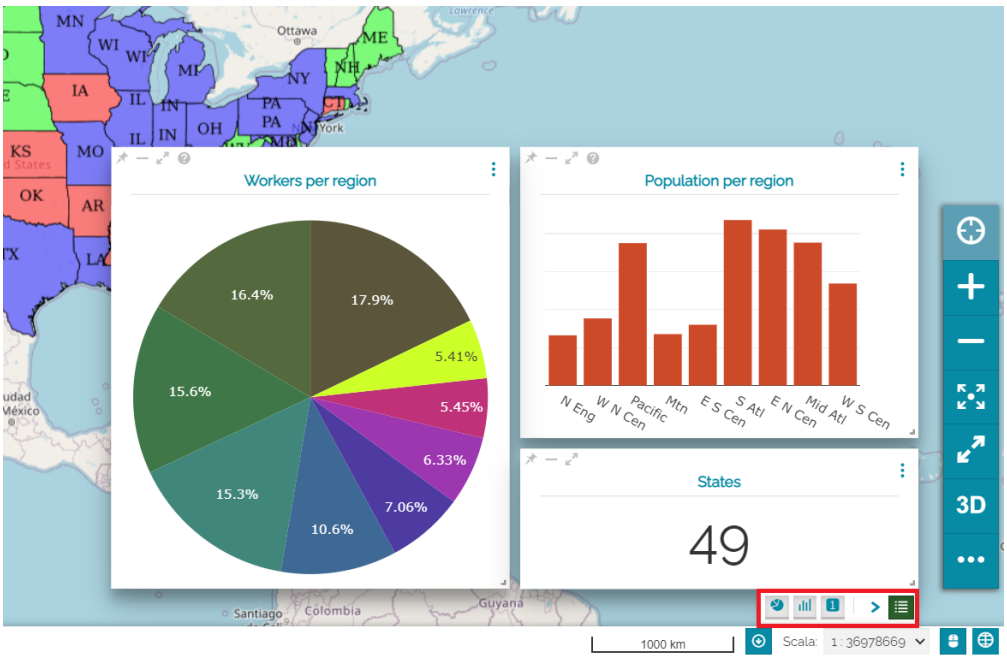
Once the  button is clicked, the panel of the last step appears:



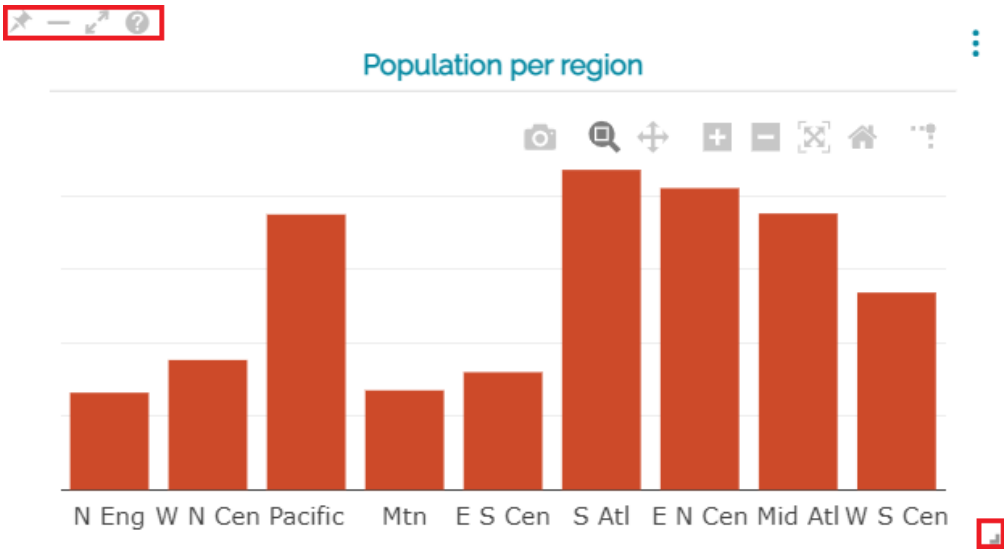
Also in this case the toolbar and the information to be inserted are similar to the ones in Chart section, with the only exception that the **Filtering** button is missing.
An example of counter widget could be:

Manage existing widgets

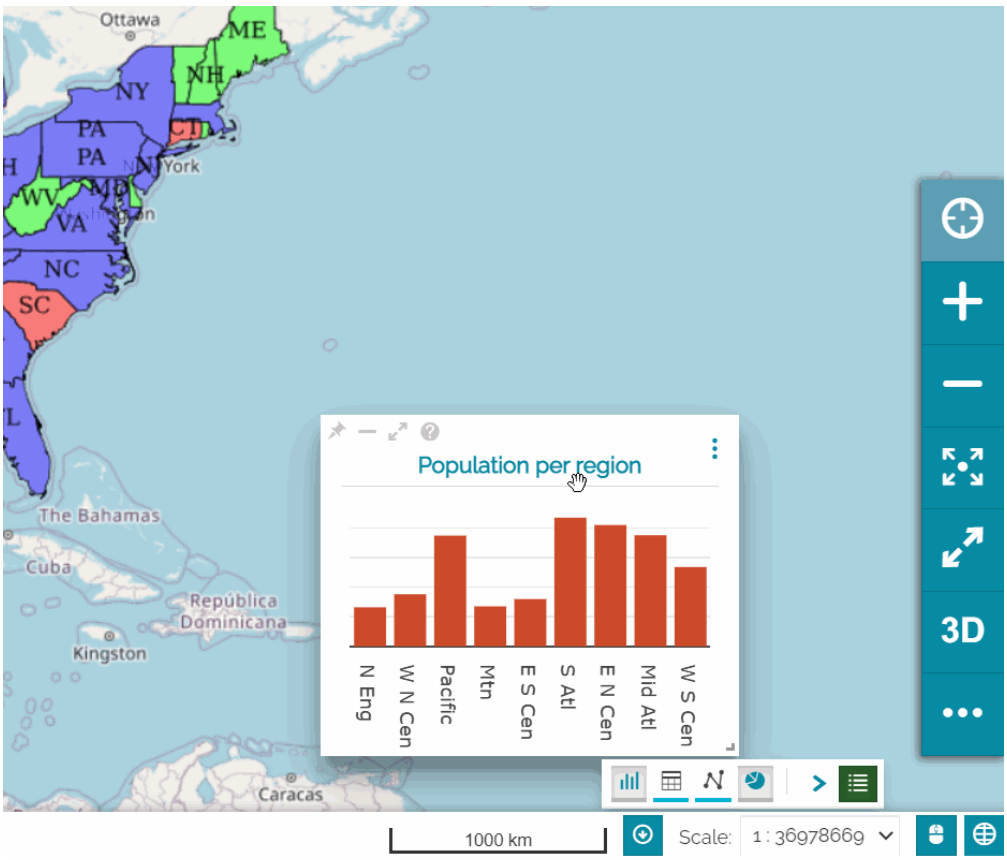
Once widgets have been created, they will be placed on the bottom right of the map viewer and the *Widgets Tray* appears:



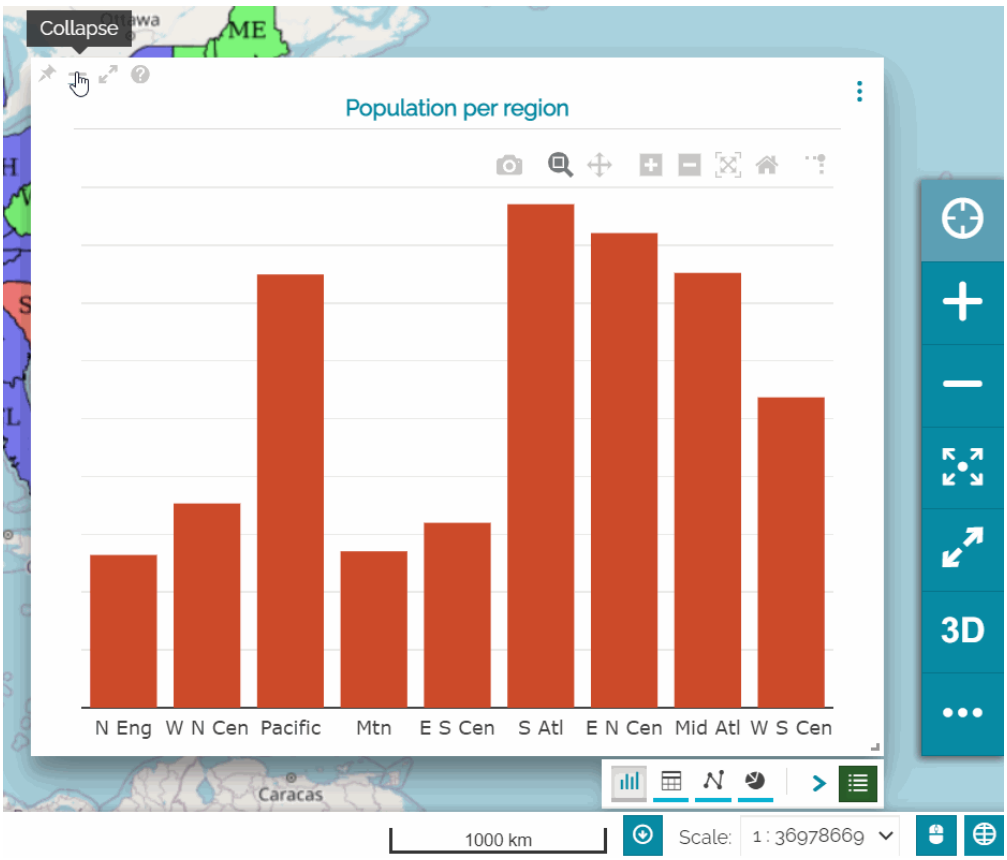
Through the buttons available on each widget the user can perform the following operations:



- Drag and drop the widget to move it within the map area of the viewer and **resize** it through the button (also available for widgets present in a dashboard)



- **Pin** the position and the dimension of the widget through the button
- **Collapse** the widget through the button and expand it again by clicking the related button in the *Widgets Tray*



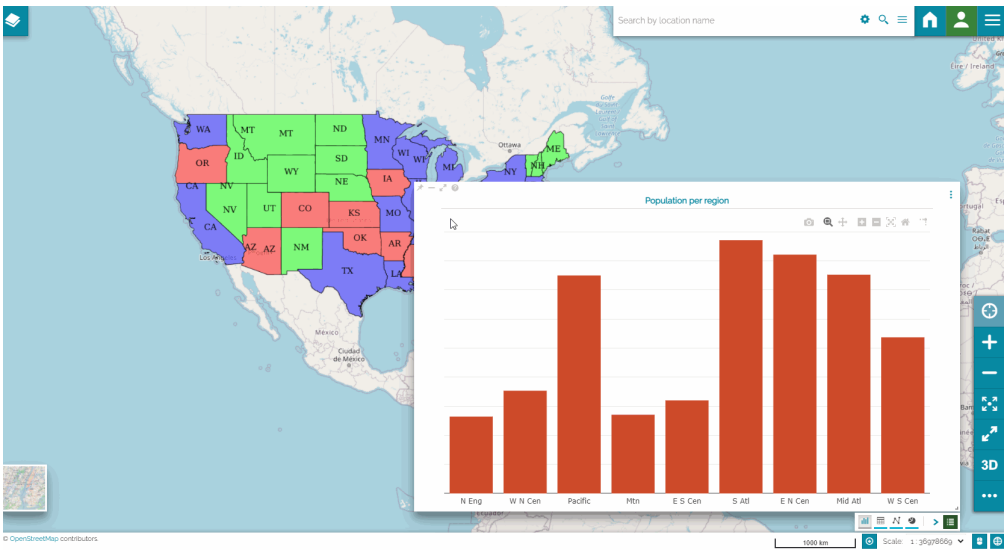
Note

The *Widgets Tray* allows the user to expand/collapse each single widget individually or all of them at once by using the button.

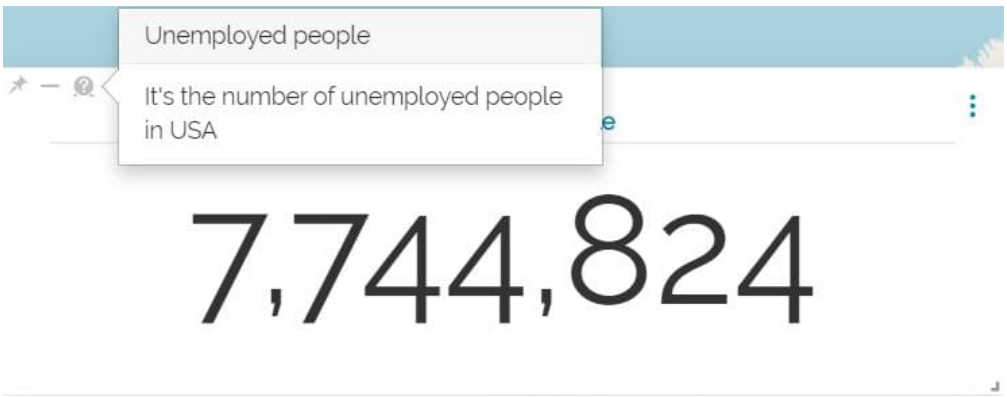
Warning

When both **Timeline** and widgets are present in a map, the *Timeline* button appears in the *Widgets Tray* allowing the user to expand and collapse it (widgets and *Timeline* can't anyhow be expanded at the same time).

- Make the widget **Full screen** through the button (also available for widgets present in a dashboard)

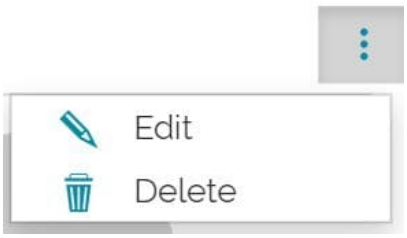


- Access to the *Title* and *Description* info through the button, if this information has been provided during the widget configuration/creation



Access widgets menu

Once a widget is added to the map, it is possible to access its **Menu** through the button. For *Text*, *Table* and *Counter* widgets, the following menu appears:



From here the user can:

- **Edit** the widget

- **Delete** the widget

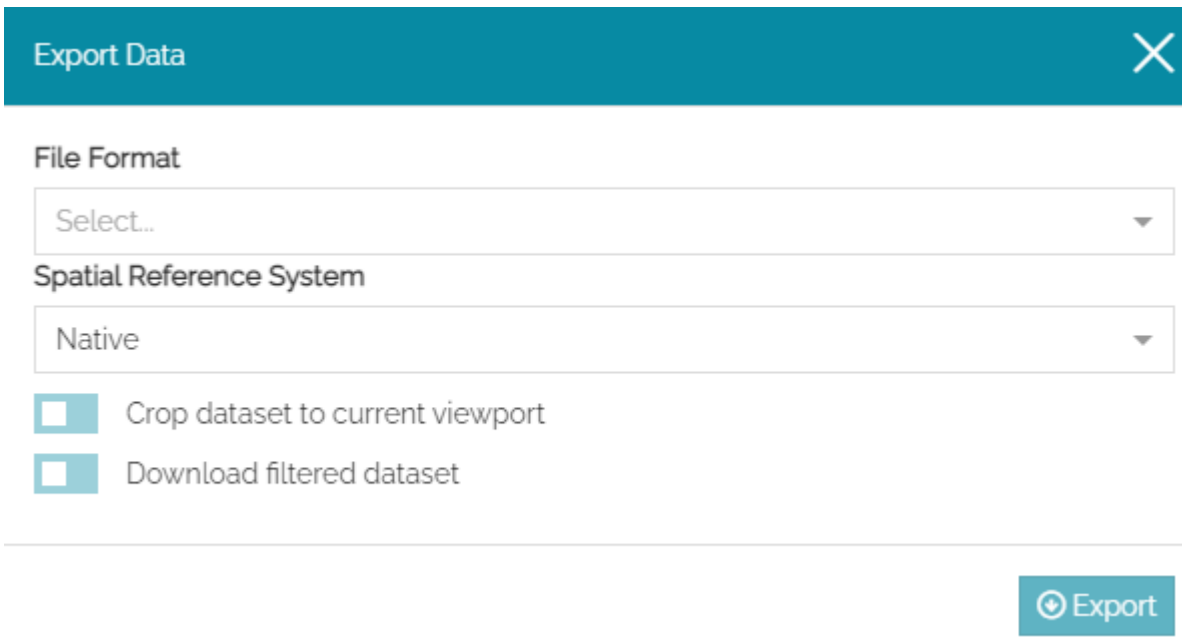
Only for *Charts*, the menu is like the following:

In particular, the user can also:

- **Show chart data** in tabular representation
- **Download data** in .csv format
- **Export Image** in .jpg format

Export Layer Data

MapStore allows to export both vector and raster layers present in TOC. In order to provide advanced export capabilities the WPS Download process must be installed and available in GeoServer. MapStore performs a preventive check for this as soon as the user opens the tool: if the WPS Download process is not available, MapStore uses the WFS service as fallback and the export options are limited (eg. only vector data can be exported). Once a layer is selected in the TOC, the user can open the **Export Data** tool by clicking the button available in the layer toolbar.

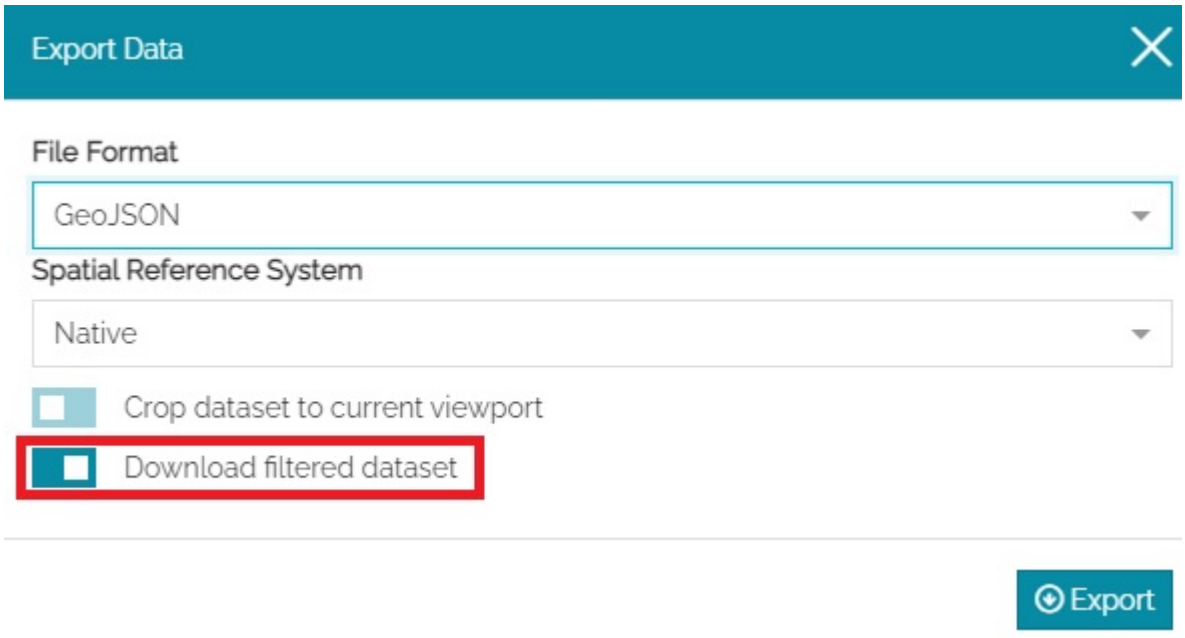


Note

Only for the **Vector layer**, the user can also download data by opening the Attribute Table from the TOC and clicking the button.

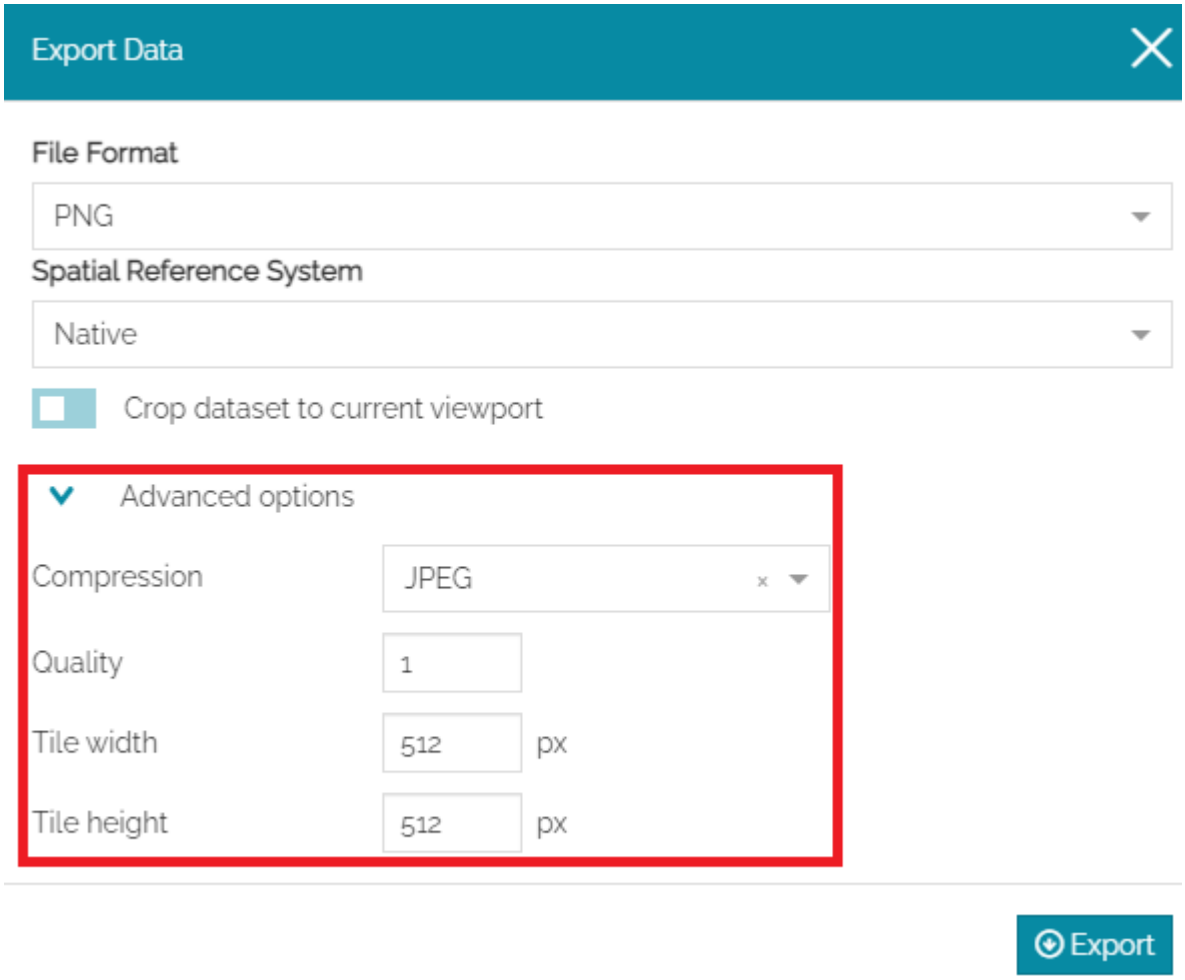
From the **Export Data** panel the user can:

- Select the **File Format**. The list of formats depends on the availability of the WPS Download process in GeoServer. If the WPS process is available, the user can choose between `GeoJSON`, `wfs-collection-1.0`, `wfs-collection-1.1`, `Shapefile` and `CSV` for *vector layers*, and between `ArcGrid`, `TIFF`, `PNG`, `JPEG`, in case of *raster layers*. If the WPS Download process is not available for some reasons, MapStore provides the list of formats valid for the WFS service by looking at the ones offered by the services capabilities (WFS Capabilities).
- Select the **Spatial Reference System** (By default `Native` or `WGS84`)
- Enable the **Crop dataset to current viewport** for downloading only the part of the layer visible on the map at that moment (this option is present in the form only if the WPS Download process is available)
- Only for *Vector layer*, allows to consider for the download also an eventual filter applied to the layer using the Filter layer tool (this option is present in the form only if the WPS Download process is available)

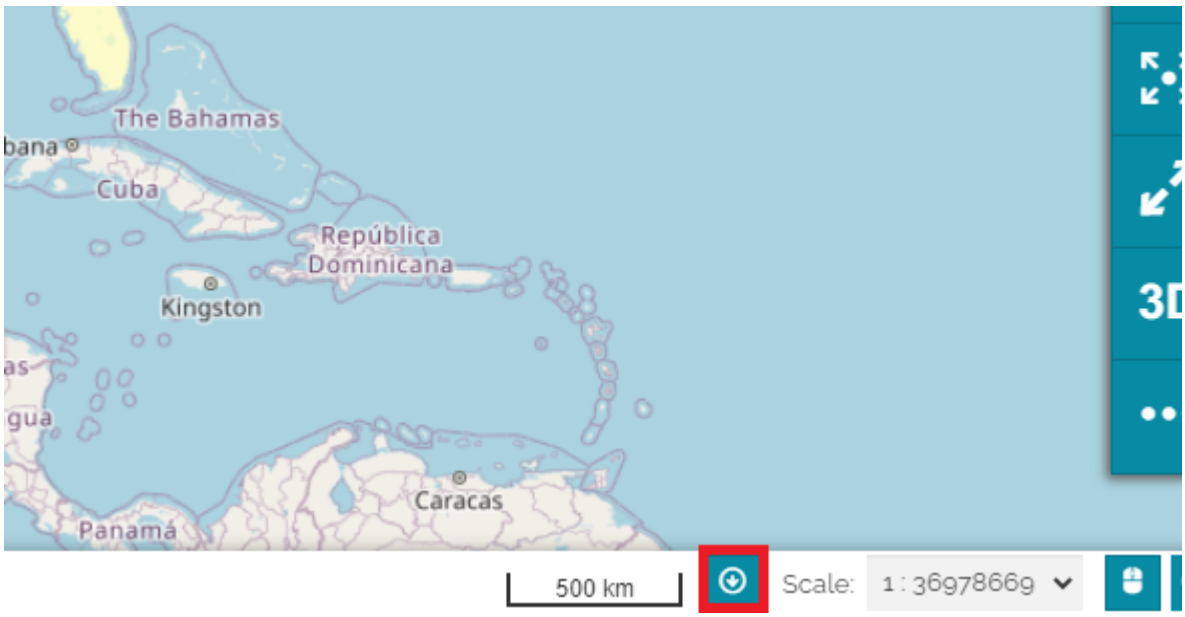


- Only for *Raster layer* (and if the WPS Download process is available) the user can open the **Advanced options** to choose:
- The **Compression type** used to store internal tiles (`CCITT RLE`, `LZW`, `JPEG`, `ZLip`, `PackBits` or `Deflate`)

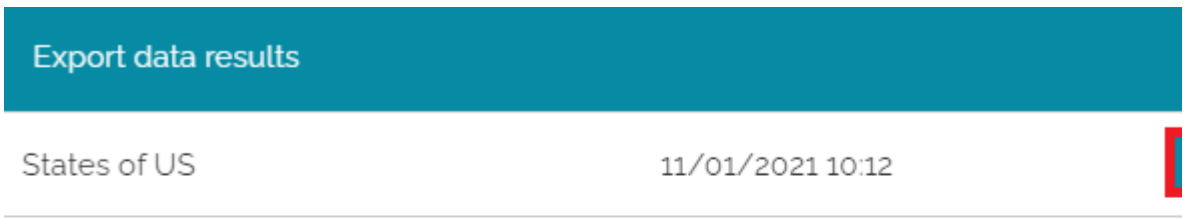
- The **Compression quality** for lossy compression (JPEG). Value is in the range [0 : 1] where 0 is for worst quality/higher compression and 1 is for best quality/lower compression
- Tile **Width** of internal tiles, in pixels
- Tile **Height** of internal tiles, in pixels



With a click on the Export button MapStore performs the export request. In case of WPS Download process available, multiple export requests can be performed from MapStore asynchronously. An information popup informs the user when an export process starts and the user can check the status of the process itself by opening the **Export Data Result** panel with a clicking on the button available on the right side of the footer.



The **Export Data Result** provides the list of exports processes started by the user and their status: as soon as the WPS completes the export operation, its status is reported by MapStore to the user (in progress, completed, and so ready for download, or failed). Therefore, the user can:



- Check for eventual reported errors: a specific icon informs the user that the process failed with a popup message.
- **Download** the final `zip` file: clicking the button
- **Remove** the final `zip` file: clicking the button

MapStore Toolbars

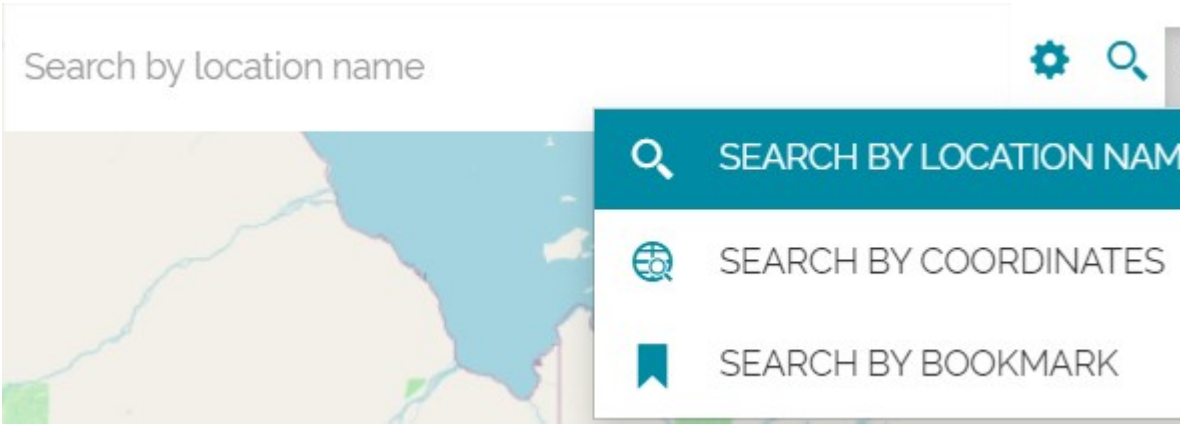
The main toolbar of MapStore, used by the user to interact on the map viewer, are:

- The **Search bar**
- The **Side toolbar**

Search Bar

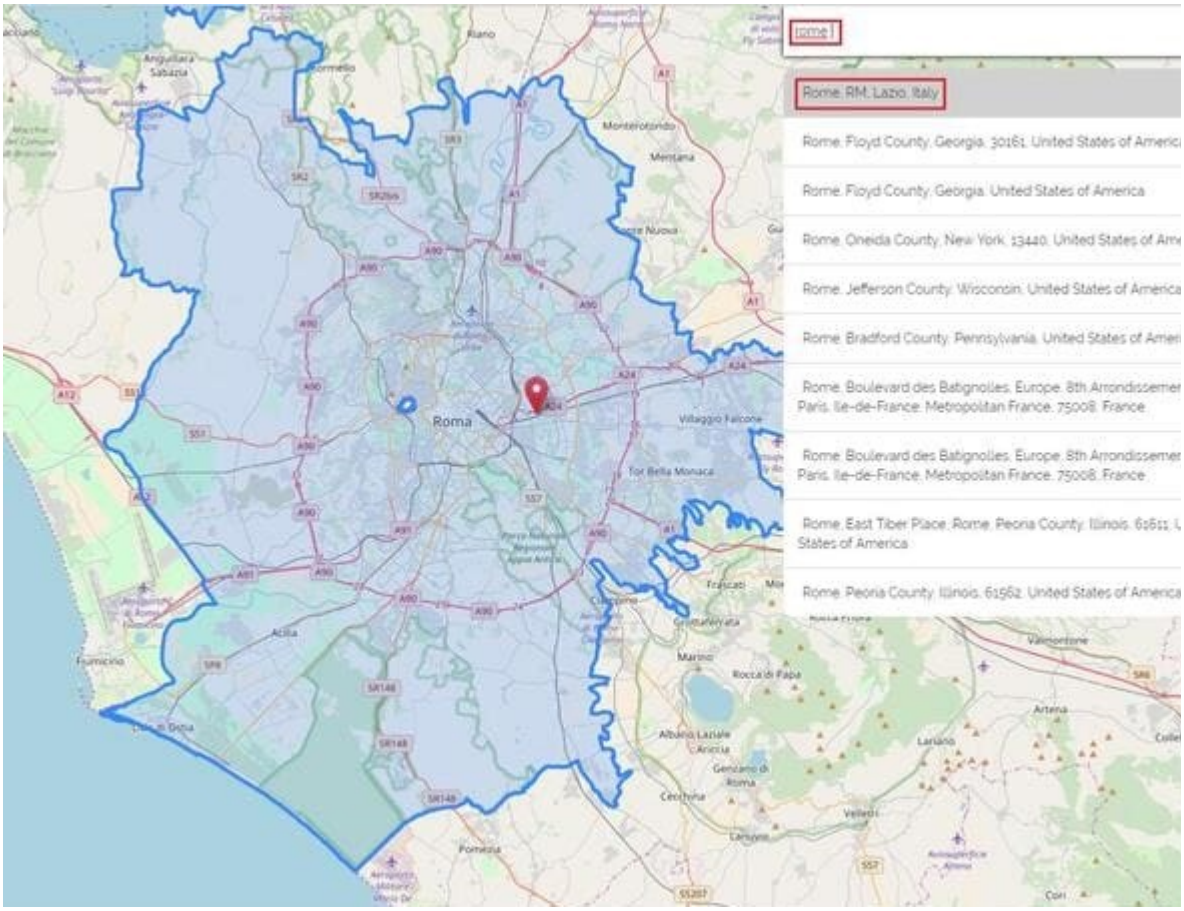
The search bar is a tool that allows the user to query the layers in order to find a specific information. In MapStore it is possible to perform the search in four different ways:

- By **Location name**
- By **Coordinates**
- By **Configuring a search service**
- By **Bookmarks**



Search by location name

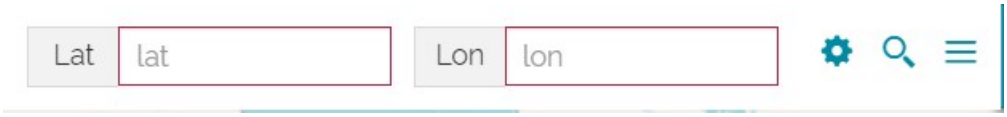
The *Search by location name*, set by default when a new map is created, allows the user to search places asking the OpenStreetMap Nominatim search engine. Typing the desired place, the Nominatim seach engine is queried; selecting then the desired record in the list of results, the map is automatically re-center/ zoomed to the chosen area that is also highlighted:



Search by coordinates

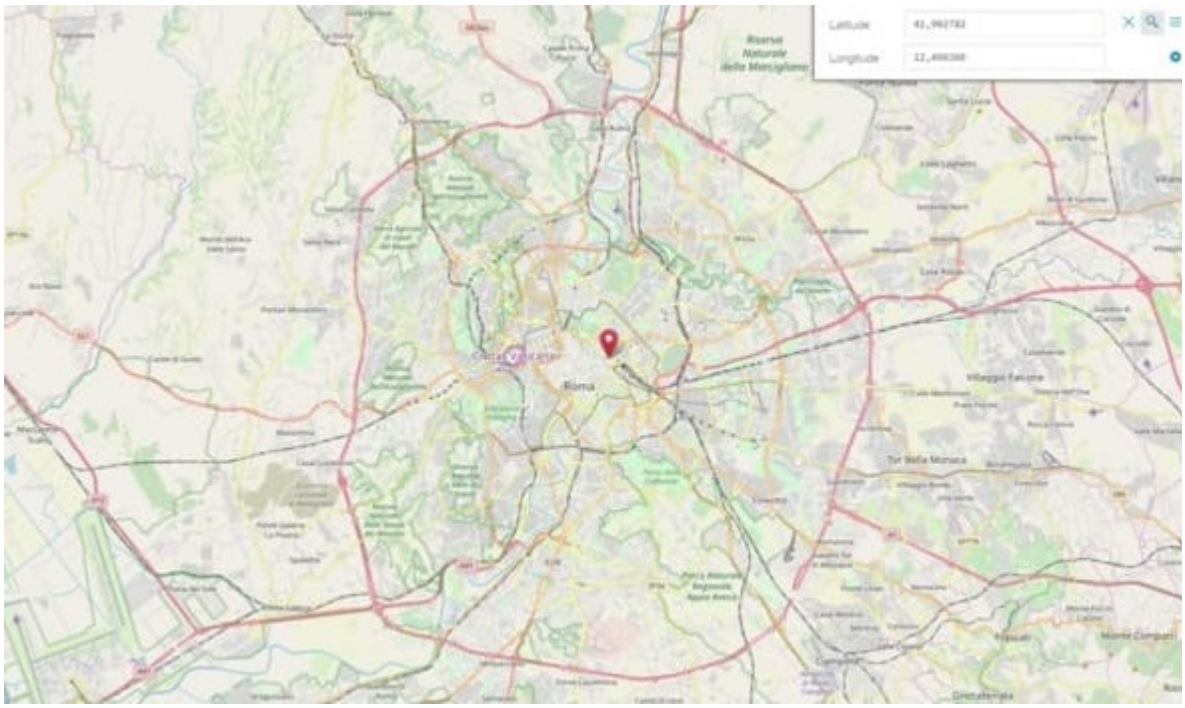
Performing a *Search by coordinates* the user can zoom to a specific point and place a marker in its position. That point can be specified typing the coordinates in two different formats:

- **Decimal** (the default format)



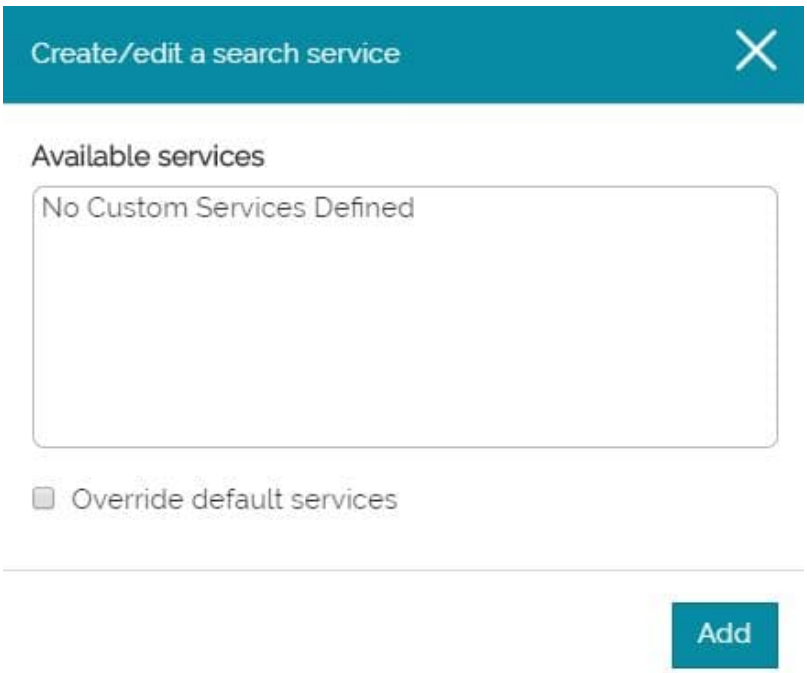
- **Aeronautical** (that can be chosen through the ⚙ button)

Once the coordinates are set, it is possible to perform the search with the 🔍 button. The displayed result is similar to the following:

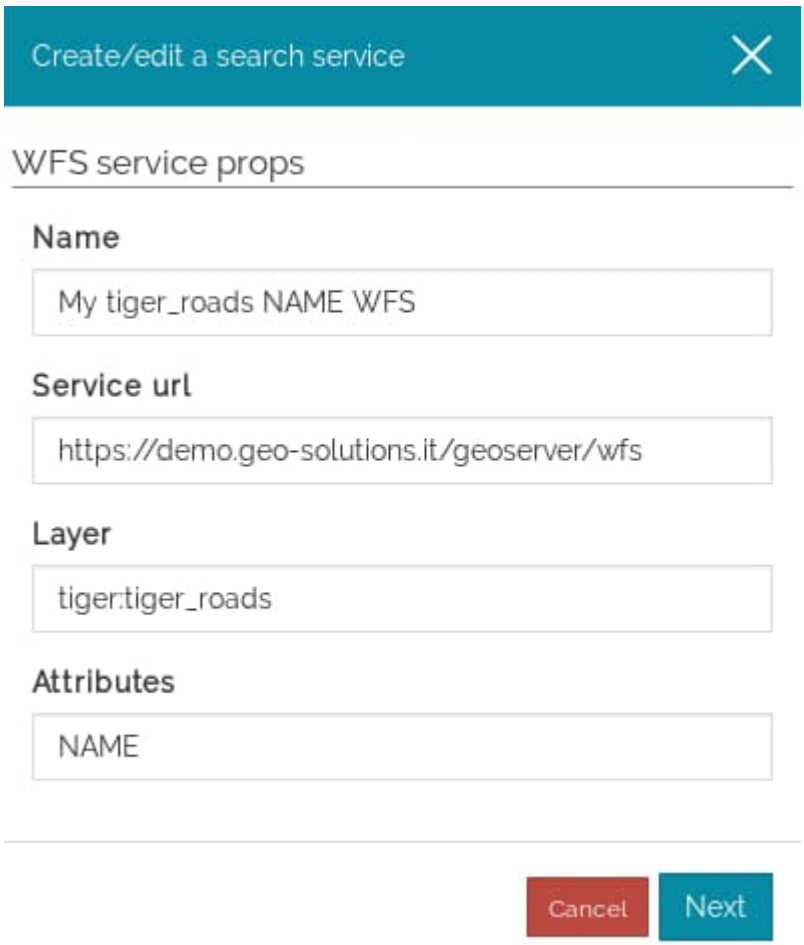


Configuring a search service

MapStore allows the user also to extend or replace the default OSM results with additional *WFS Search Services*. Selecting the **Configure Search Services** option ⚙ , the following window opens:



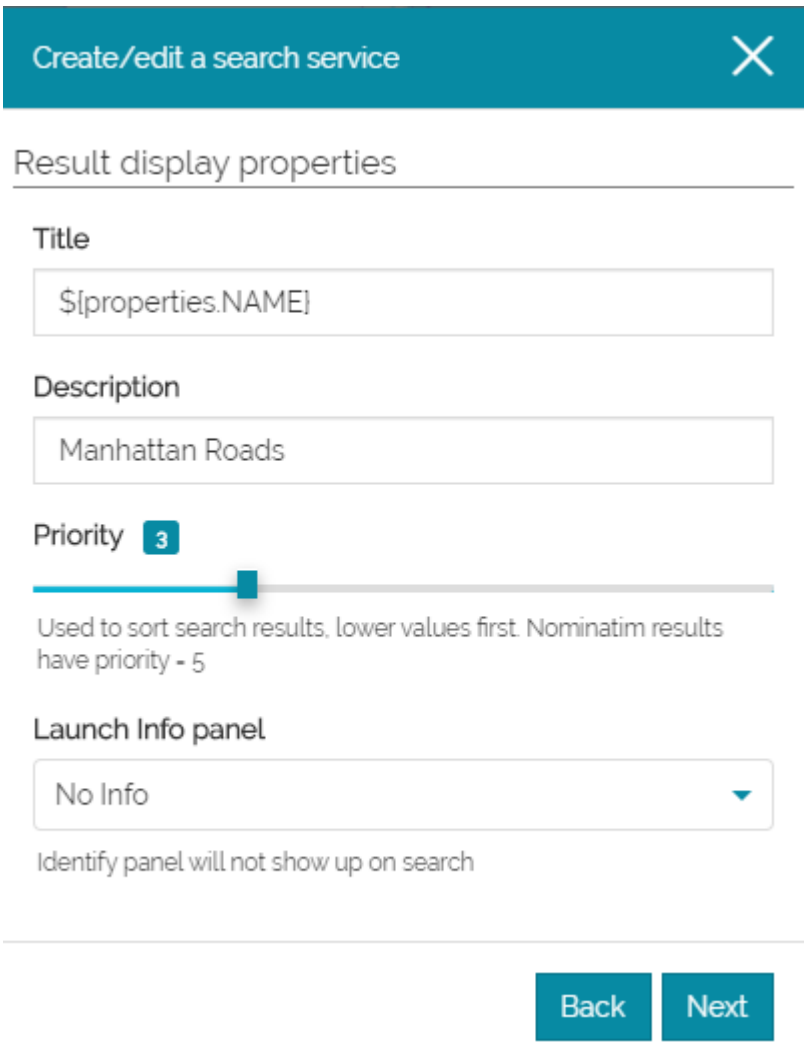
In order to create a new custom service, the Add button brings the user to a page where he can set the *WFS service properties*, for example:



In particular, the information to be entered are:

- **Name** of the service
- WFS **Service URL** the user want to call
- **Layer** to be queried
- Specific **Attributes** (comma separeted fields) the user wants to query

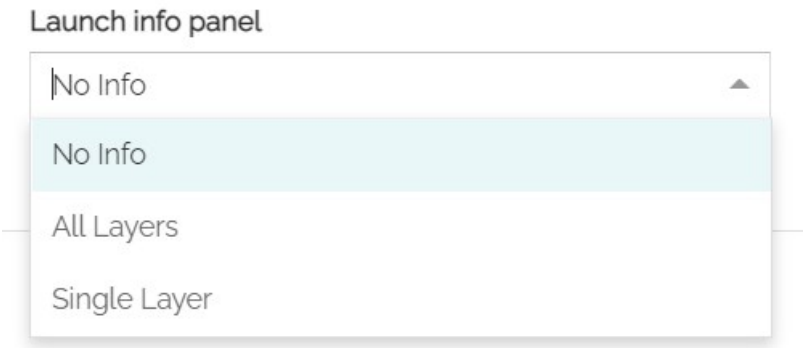
When all the options are set, by clicking on the Next button a new panel opens, where it is possible to choose the properties for the displayed results:



In this case, the user can define the following settings:

- The **Title** displayed on the top of each results row (in the previous image, for example, the chosen title for the results is the one corresponding to the attribute NAME of the feature)
- The **Description** to report in the results just below the title
- The **Priority**, a parameter which determines the position of the records in the results list. Lower values imply a higher positions in the results list and vice versa. By default the OpenStreetMap Nominatim search engine result has priority equals to 5, therefore in order to see the custom results in a higher position a lower priority value is needed
- The **Launch Info panel** allows the user to chose if and how the custom search interact with the Identify tool. In particular, with the *No Info* option, the Info panel doesn't show up once a record from the search results is

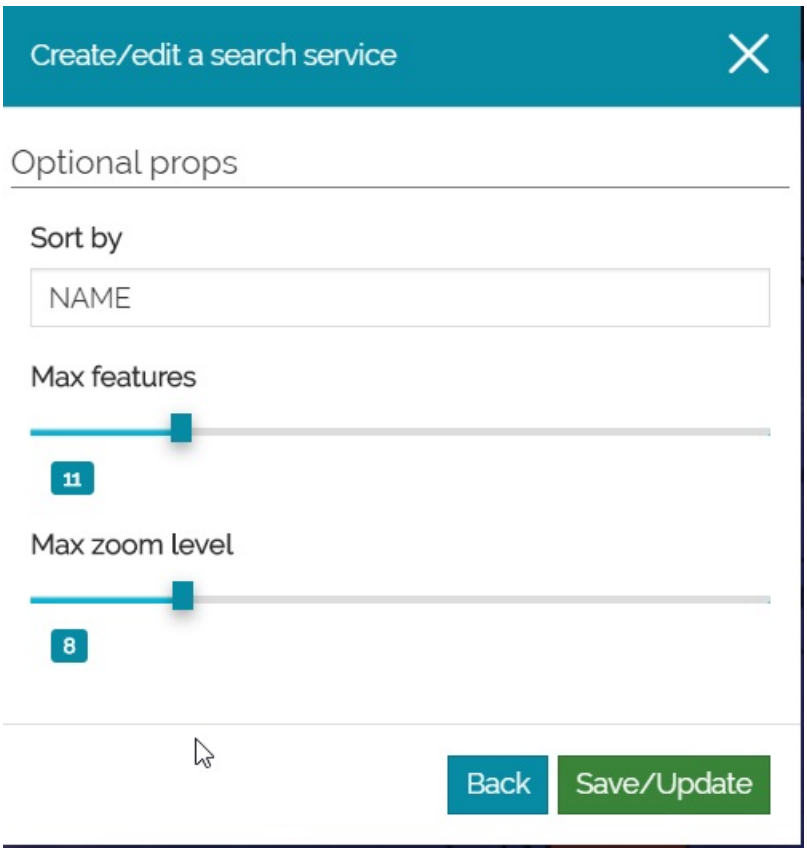
selected. Selecting *All Layers* or *Single Layer* the Identify tool is triggered, and the related panel opens displaying the information of all/single layer(s) visible in the map. With *Single Layer* instead, the Identify tool is triggered only for the layer (if it is present and visible in the map) related to the selected record in the search result list.



Note

Note that, selecting *All Layers* or *Single Layer* options, the point used for Identify request is a point belonging to the surface of the geometry of the selected record. Moreover, using *Single Layer*, the Identify request will filter results to the selected record and to its layer, using `featureid` which might be ignored by other servers, but can be used by GeoServer to select the specific feature of the results, when info_format is other than *application/json*. In order to achieve filtering of feature on servers other than GeoServer, one can select the format (*info_format*) as *application/json* for the layer to **GetFeatureInfo** from the layer settings in TOC to allow filtering features by using the ID of the selected record.

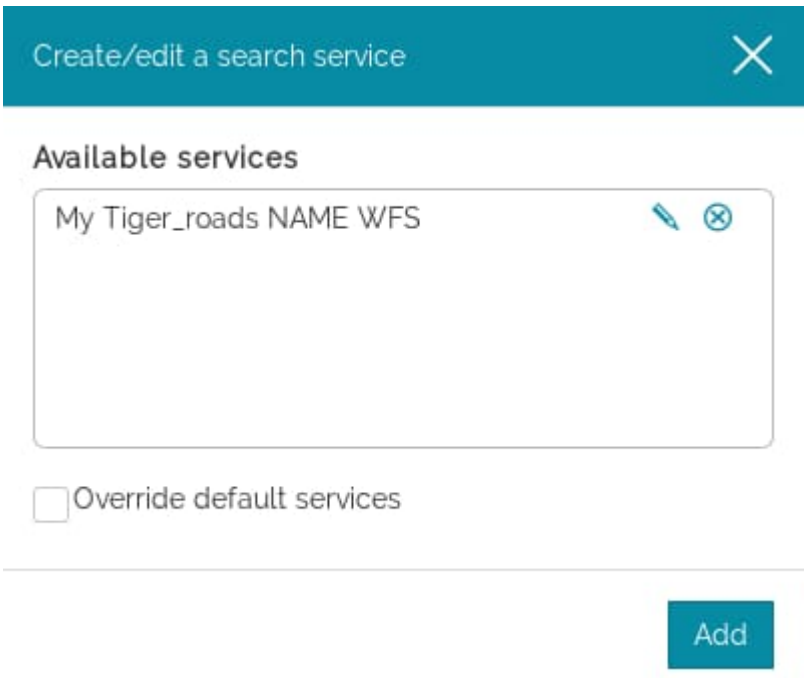
Once all the option are set, it is possible to move forward with the Next button Next that opens the *Optional properties* panel:



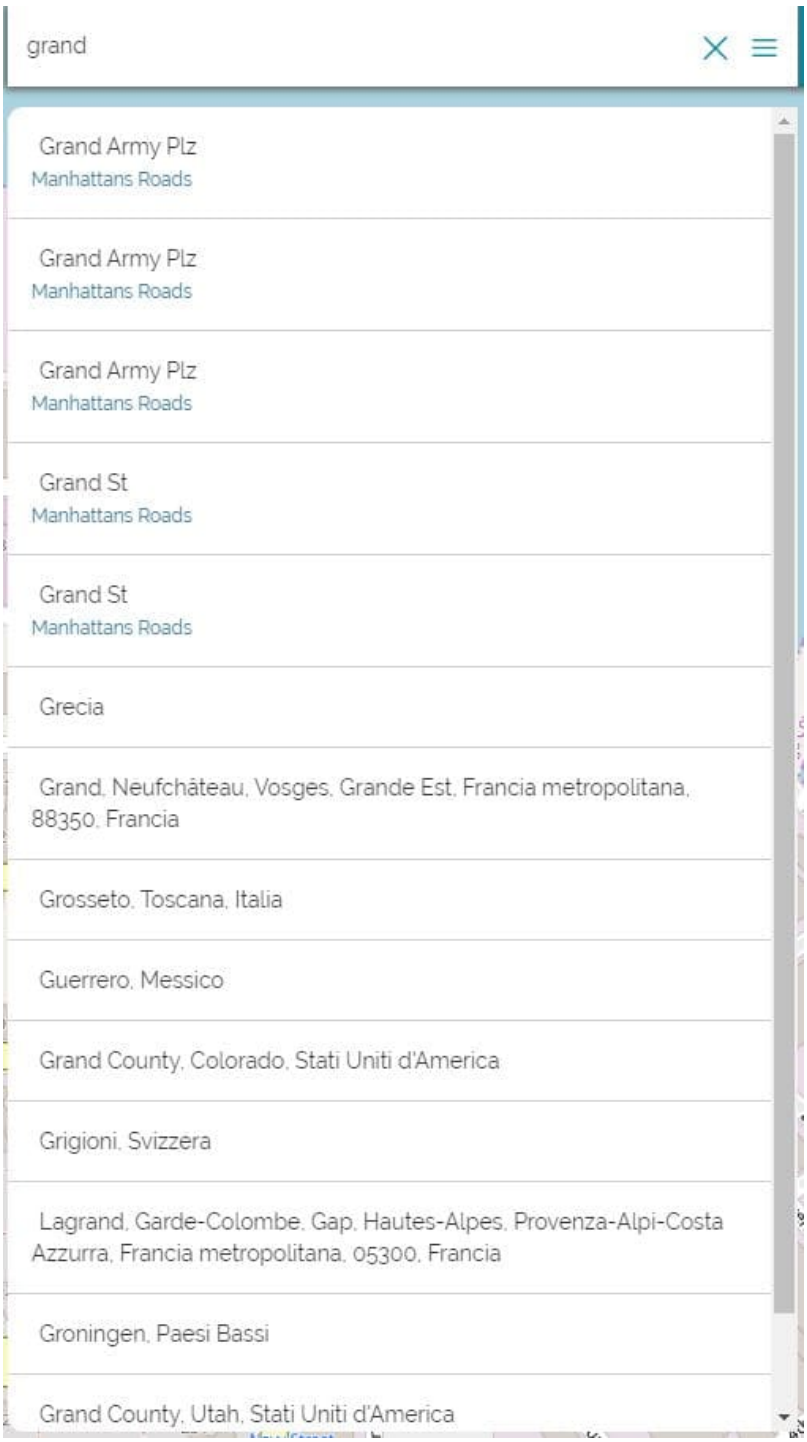
Here the user can choose:

- To **Sort** the results **by** the specified attribute
- The **Max** number of **features** (items) displayed in the custom search results
- The **Max** level of **zoom** to be set for the map when opening from the custom search result

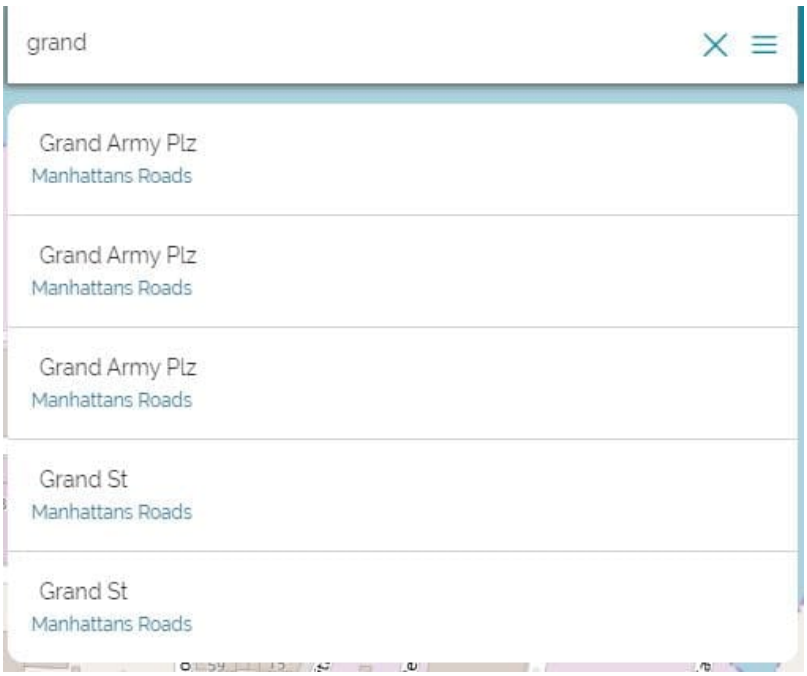
After the Save/Update it is possible to see the custom WFS search service inside the *Available services* list:



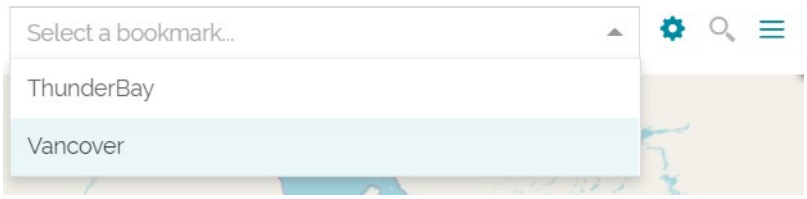
Once a search service is created, it is always possible to Edit it or Remove it from the list. By default the **Override default services** option is disabled, in that case performing a search not only the custom search service results are shown, but also the Nominatim ones:



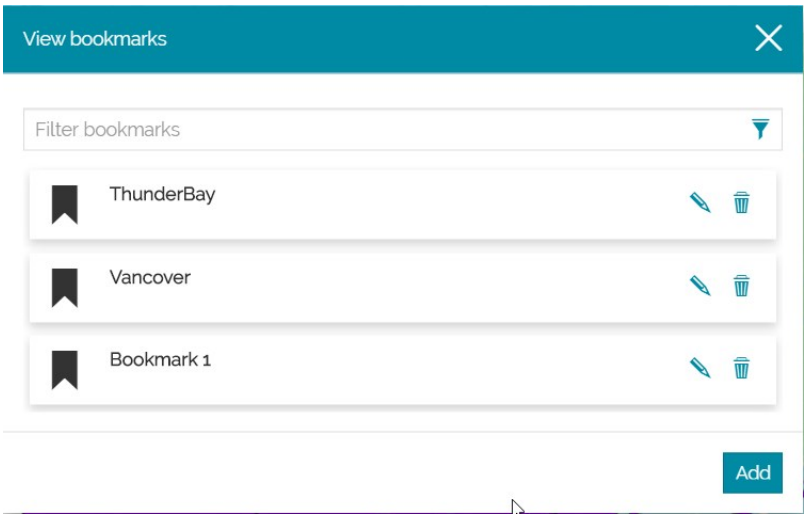
Once the **Override default services** option is enabled, only the custom search service results are shown:



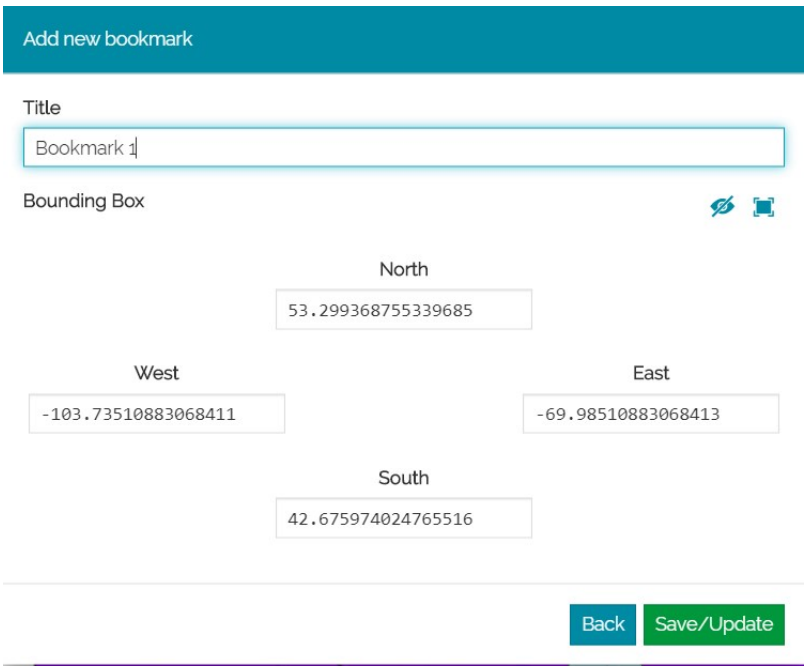
Search by bookmark



MapStore allows the user to search by the preconfigured bookmarks, which can zoom to a specific bounding box area or zoom along with reloading the visibility of the layers. Selecting the **Bookmark settings** icon, the following window opens:



In order to create a new bookmark, the Add button brings up **Add new bookmark** page where the user set the *Bookmark properties*, for example:

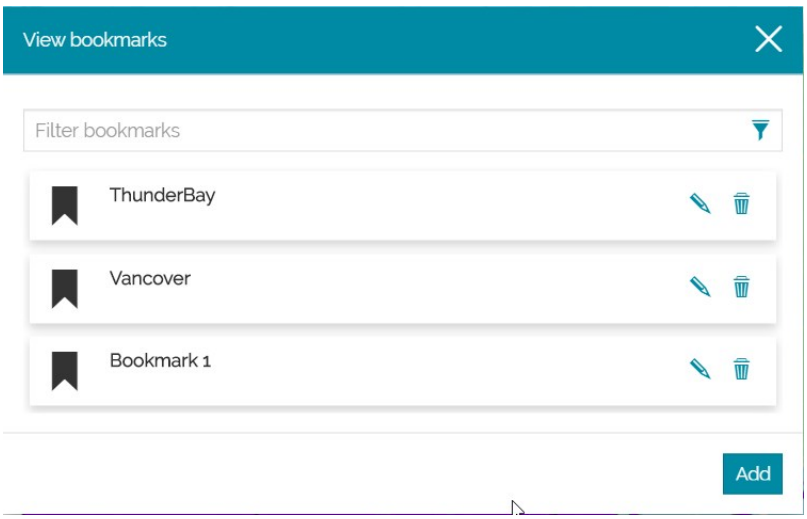


In particular, the information to be entered are:

- **Title** of the bookmark
- **Bounding Box** property the user wish to zoom to
- **West**, **South**, **East** and **North**
- **Toggle layer visibility reload**, to enable/disable the layer visibility reload when searched by bookmark

Note: The user can define bounding box value either manually or by selecting **Use current view as bounding box** to fetch the current bounding box values from the map view to populate the fields

When all the properties have been set, selecting the Save/Update it is possible to see the newly added bookmark in the **View bookmarks** list:



Once a bookmark has been created, it is always possible to *Edit* it or *Remove* it from the list.

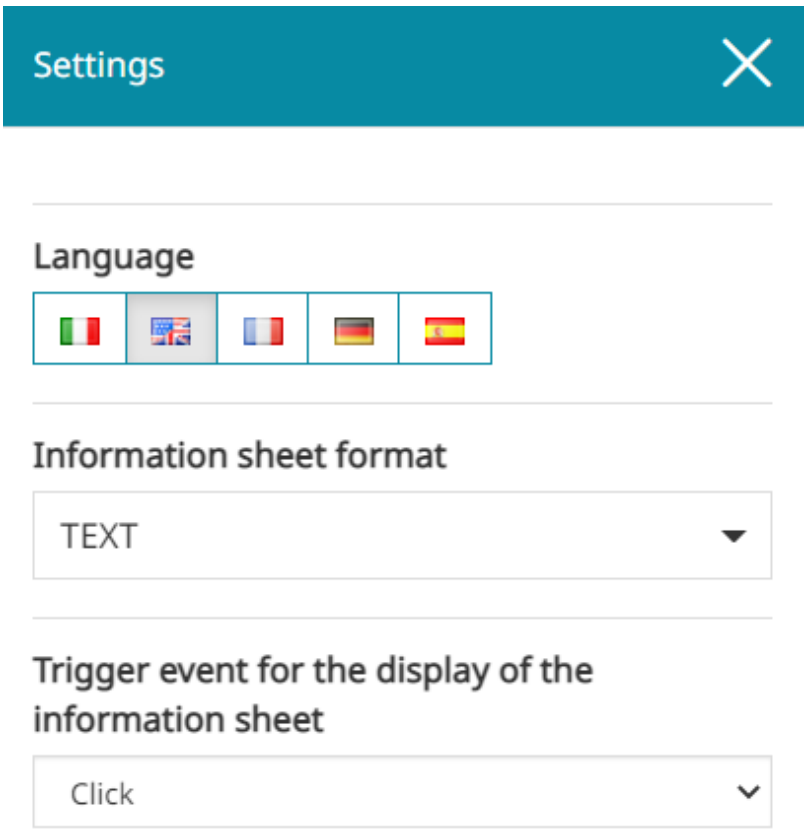
Side toolbar

The *Side Toolbar* is an important component, positioned on the right side of the map viewer, that provides to the user the access to different tools of MapStore. The following tools are the ones available by default:

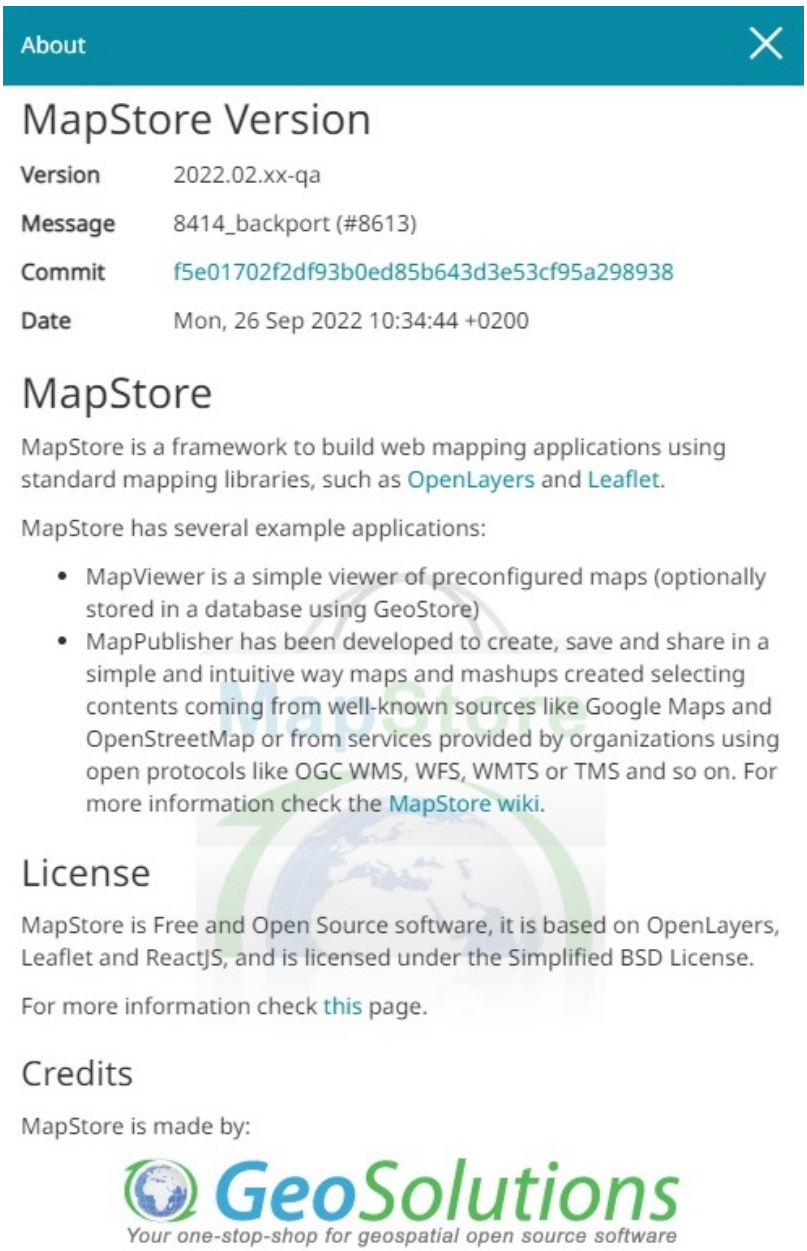
In particular, with these options it is possible to:

- Go back to the Homepage by clicking the button

- **Login/Logout** by clicking the  button (for more information see the Managing Users and Groups section)
- Print the map by clicking the button
- **Export map** in `json` format by clicking the button
- Import files from your computer by clicking the button
- Open the Catalog in order to connect to a remote service and add layers to the map by clicking the button
- Perform a Measure on the map by clicking the button
- **Save** the map by clicking the button, in order to apply the changes made in an existing map. Selecting this option, the Resources Properties window opens, already filled with the current map properties
- **Save as** when the user needs to save a copy of a map or save one for the first time by clicking the button. Selecting this option an empty Resources Properties window opens.
- **Delete Map** in order to delete the current map by clicking the button
- Access the map **Settings** by clicking the button, where the user can change the current **Language** and select the Identify options



- See the **About this map** panel by clicking the button, when Details are present
- Share the map by clicking the button
- Open the MapStore Documentation by clicking the button
- Start the **Tutorial** by clicking the button
- Know more information **About** MapStore and the deployed **Version** of *MapStore* by clicking the button



Warning

The **Save**, the **Delete Map** and the **Share** buttons are present in the *Options Menu* only when the map has already been saved once.

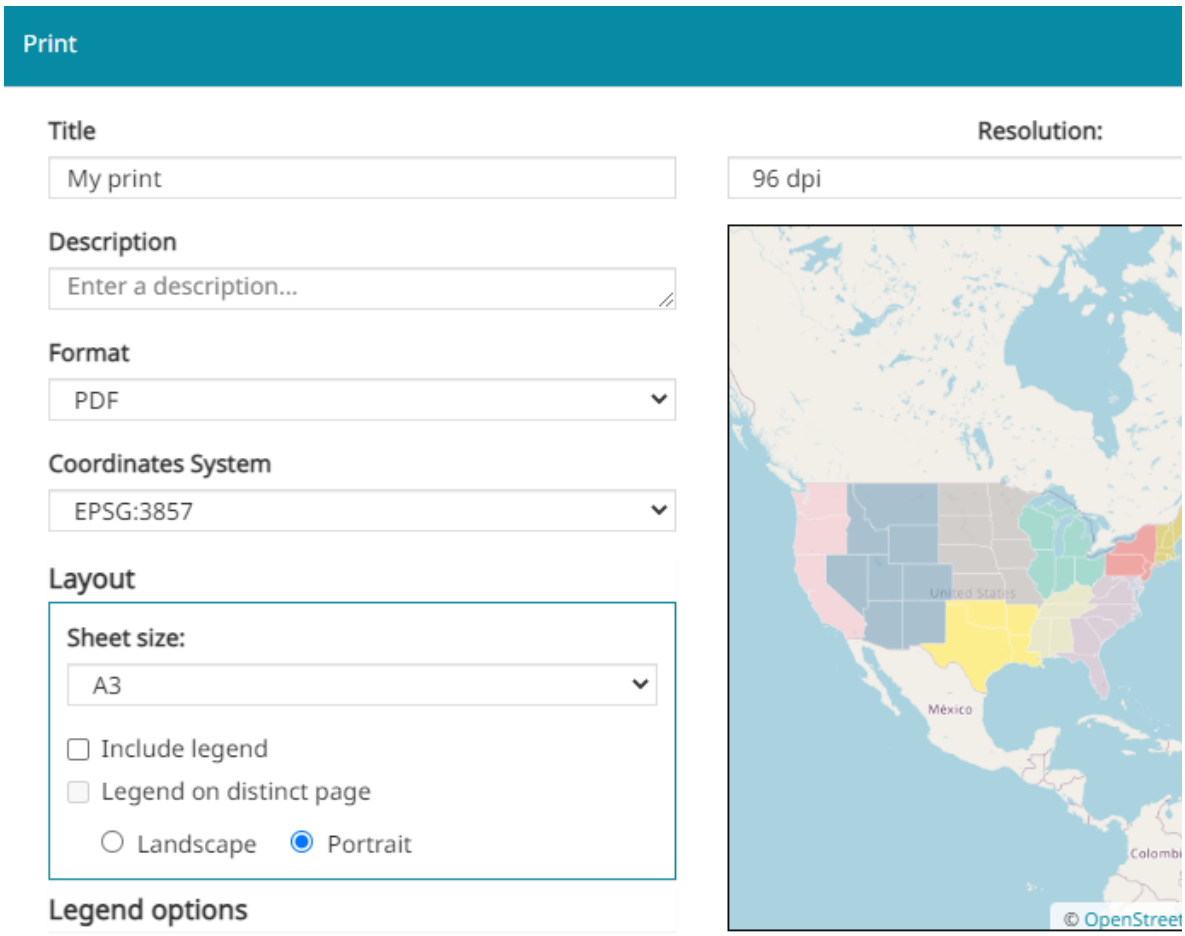
Printing a Map

In MapStore it is possible to print a map by selecting the **Print** button from Side Toolbar. The print process is composed by two main steps:

- *Print Settings* definition
- Result checking in *Preview* before download the printed file

Print settings

As soon as the *Print* button is chosen, the following window opens:

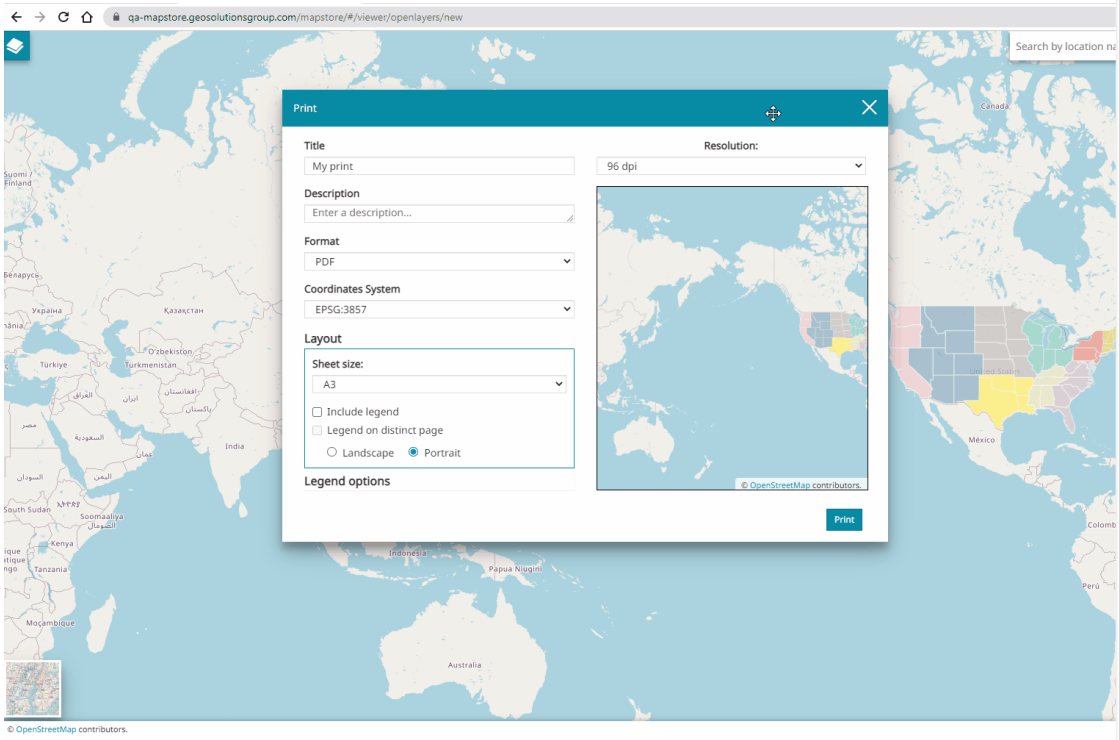


Through this window it is possible to:

- Enter a **Title** and a **Description**, that will be shown on the print page
- Change the **Format** (`PDF` , `PNG` , `JPEG`)
- Change the **Coordinates System** (`EPSG:3857` , `EPSG:4326`)
- Change the **Resolution** of the print (`96 dpi` , `150 dpi` , `300 dpi`)
- Accessing **Layout** settings
- Accessing **Legend options**

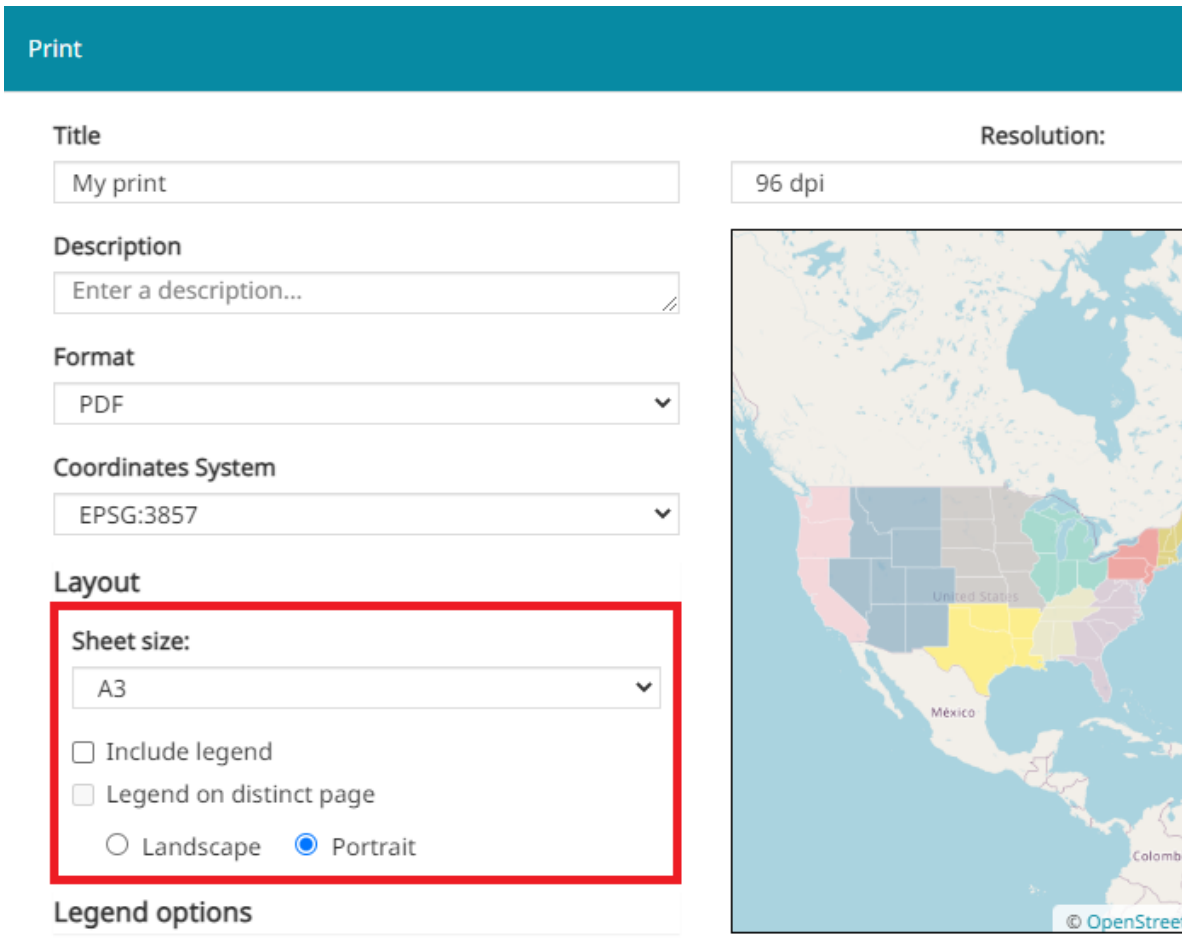
Note

In *Print settings* preview there's the map portion that will be displayed on the print sheet. In order to center the map the user can pan it until the preview displays the desired extension



Layout

Opening the **Layout** settings menu, the following menu appears:

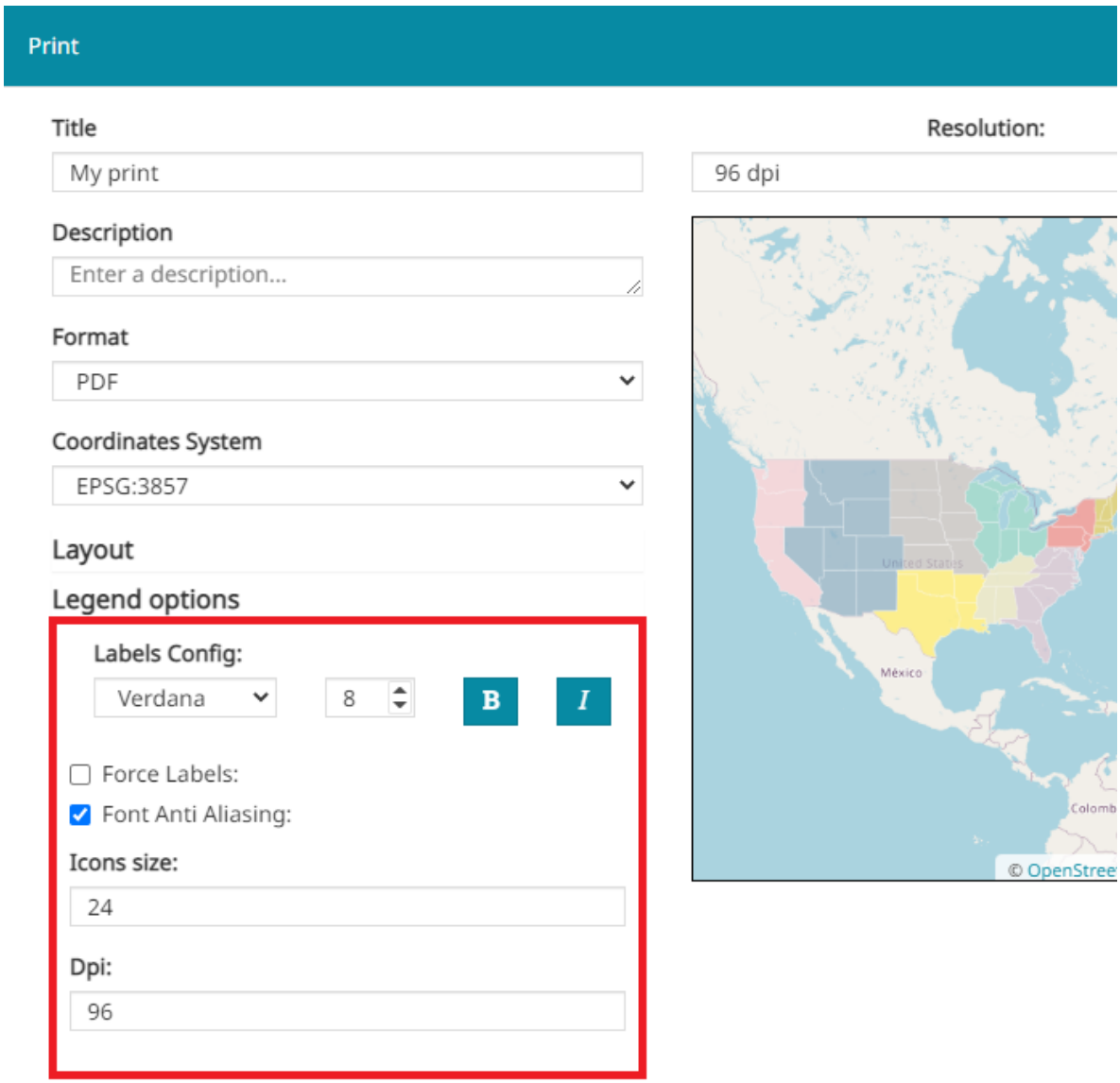


From here, in particular, it is possible to:

- Select the *Sheet size* (choosing between A3 and A4 format)
- Choose to *Include legend*
- Choose to place the *Legend on distinct page* from the map
- Select the page orientation between *Landscape* and *Portrait*

Legend options

The Legend can be customized through the **Legend options** menu:

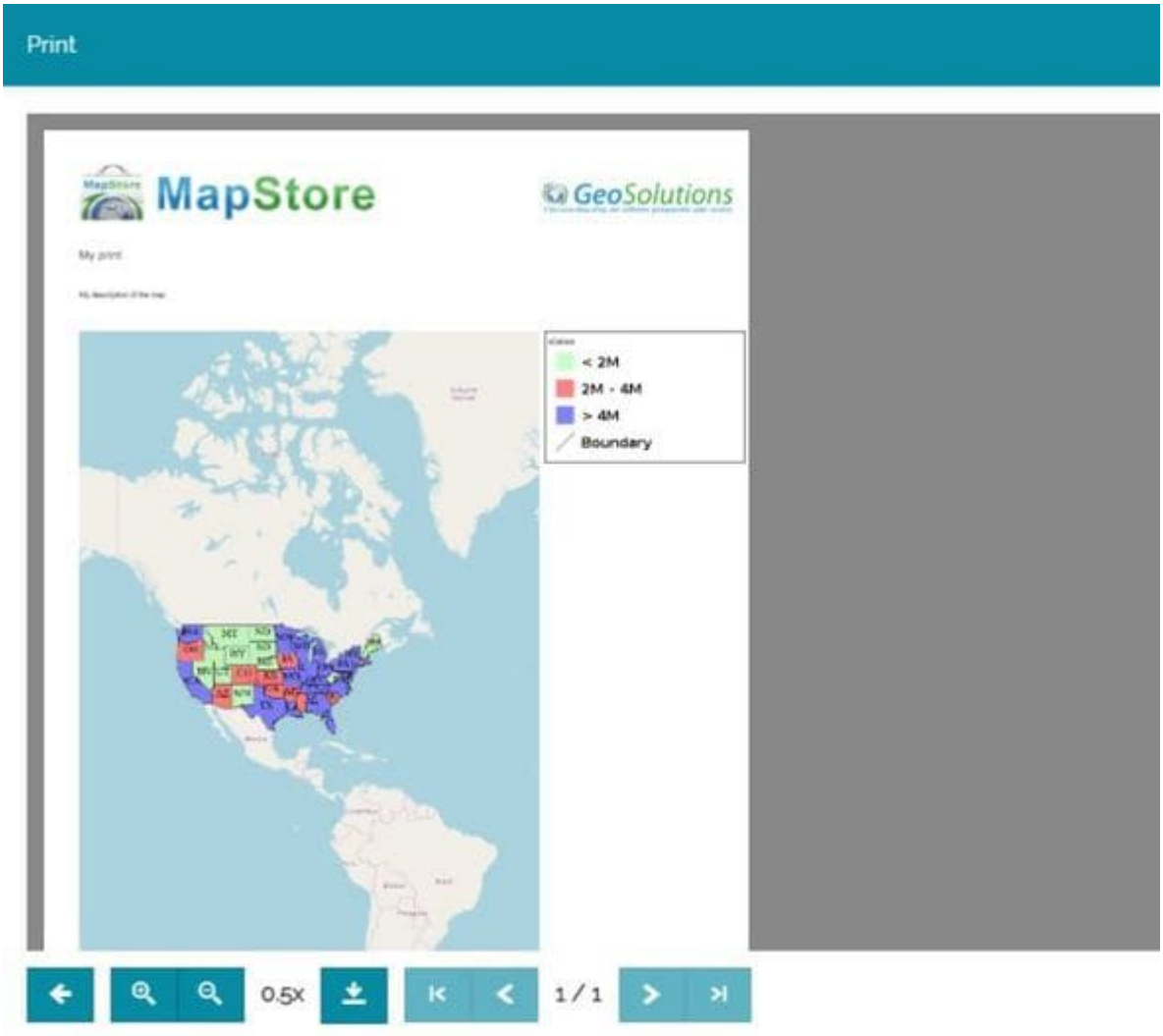


Through this menu the user is allowed to:

- Configure labels by choosing font type and size, and by adding *Bold* and *Italic* style
- Enable the *Force Labels* option, that force the display of labels even if only one rule is present (by default, if only one rule is present, the label is not displayed)
- Enable the *Font Anti Aliasing* (when Anti Aliasing is on, the borders of the labels font are smoothed improving the image quality)
- Set the *Icons size*
- Set the *Dpi* resolution of the legend

Preview

When the print settings are chosen, it is possible to access the preview by clicking on the Print button. A window similar to the following appears:



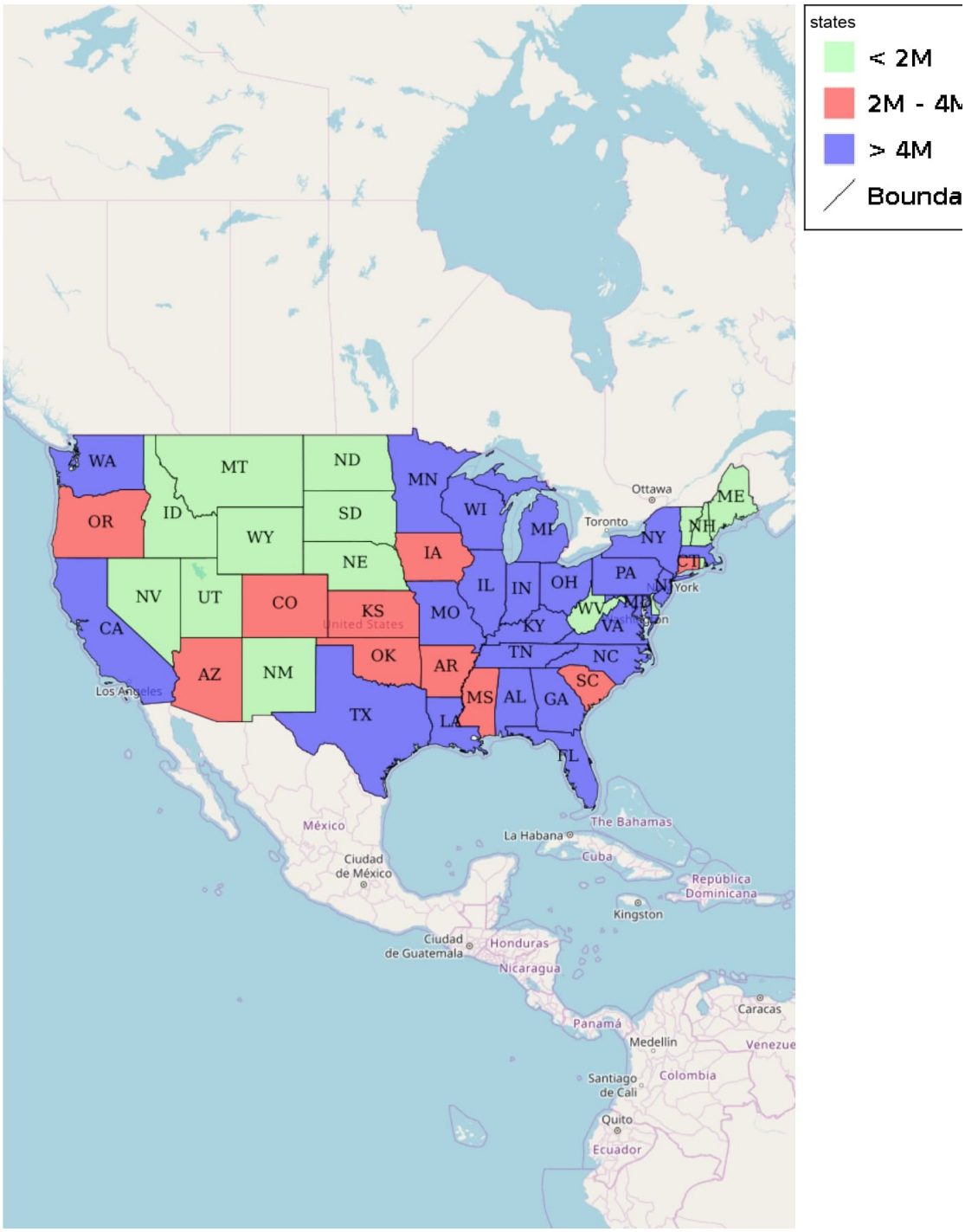
Here it is possible to:

- Zoom in/out int the preview
- Navigate between pages (when more than one page is present)
- Download the file in .pdf format

A simple printed map could be, for example, like the following:

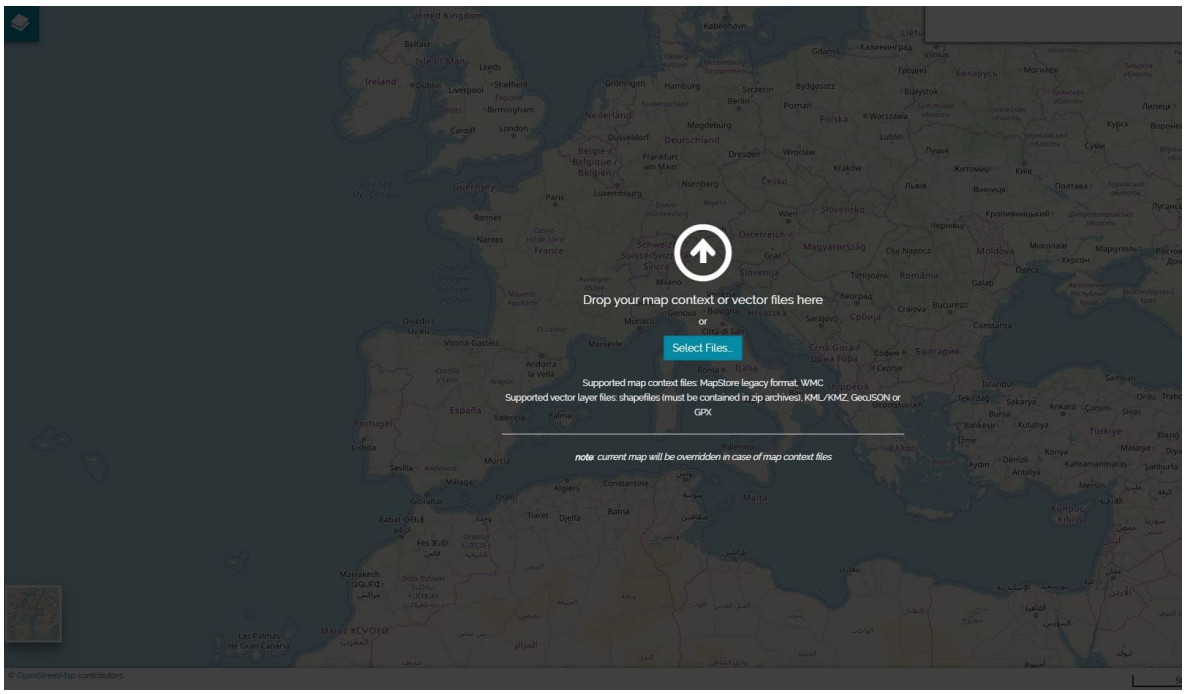
My print

My description of the map



Import Files

In MapStore it is possible to add map context files or vector files to a map. This operation can be performed by clicking from the Side Toolbar. Following these steps the import screen appears:



Here the user, in order to import a file, can drag and drop it inside the import screen or select it from the folders of the local machine through the Select Files... button. Actually there's the possibility to import two different types of files:

- *Map context files* (supported formats: MapStore legacy format, WMC)
- *Vector layer files* (supported formats: shapefiles, KML/KMZ, GeoJSON and GPX)

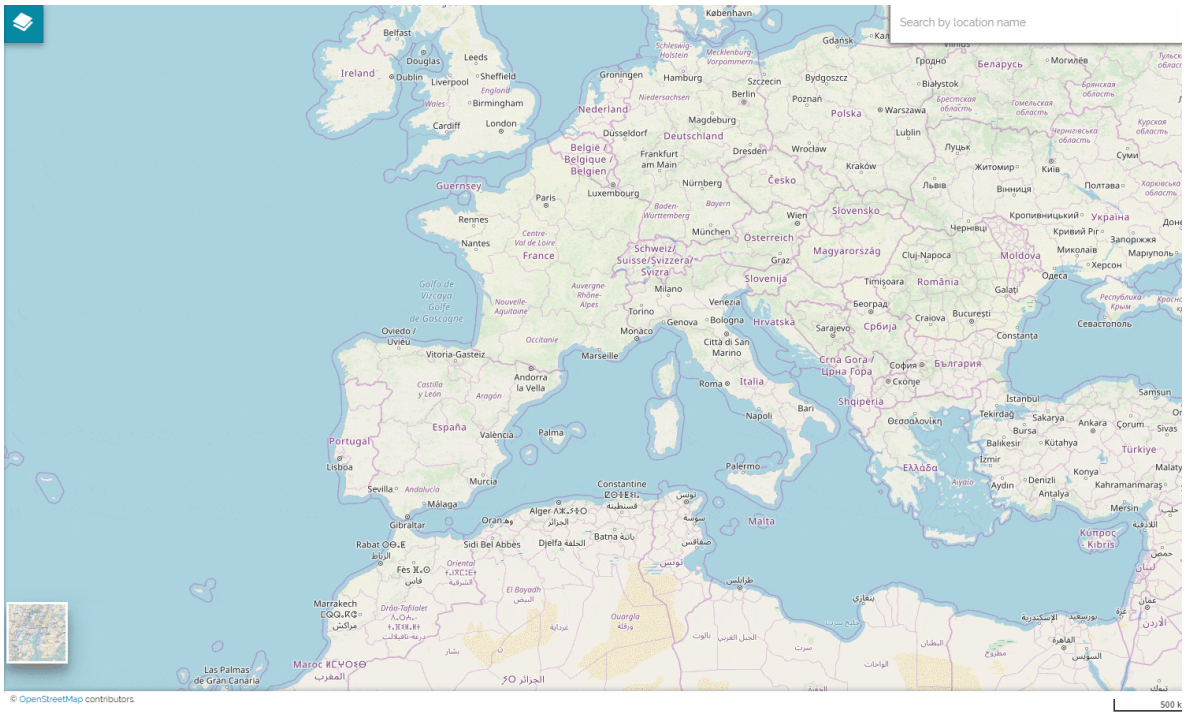
> **Warning**
>
> Shapefiles must be contained in .zip archives.

Export and Import map context files

A map context is, for example, the file that an user download selecting the button from the Side Toolbar. Map contexts can be exported in two different format:

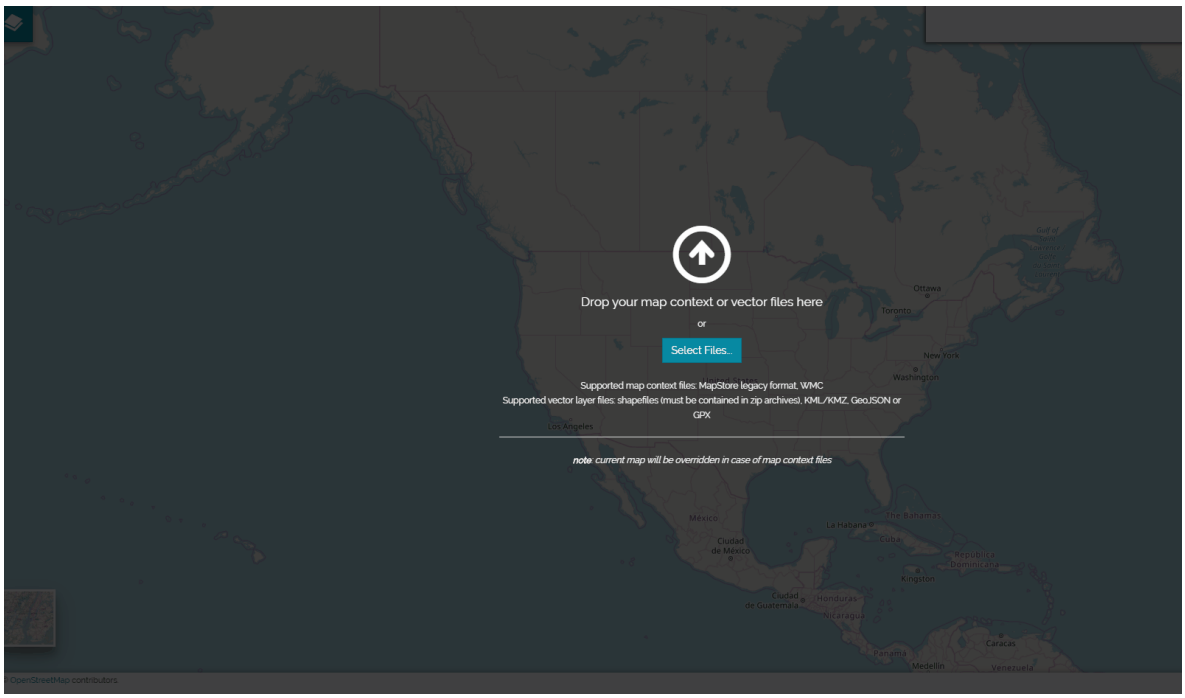
- The MapStore file, is an export in `json` format of the current map context state: current projections, coordinates, zoom, extent, layers present in the map, widgets and more (additional information can be found in the Maps Configuration section of the Developer Guide).

Adding a MapStore configuration file the behavior is similar to the following:



- The WMC (Web Map Context) file, is a `xml` format where only WMS layers present in the map are exported including their settings related to projections, coordinates, zoom and extension (additional information can be found in the Maps Configuration section of the Developer Guide).

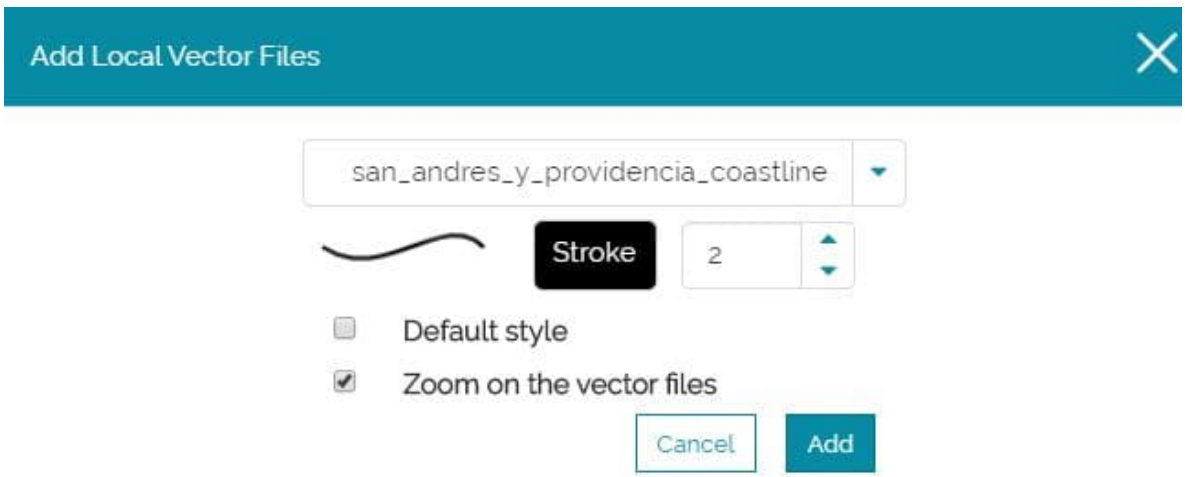
Adding a WMC configuration file the behavior is similar to the following:



> **Warning**
>
> Adding a map context file the current map context will be overridden.

Import vector files

Importing vector files, the **Add Local Vector Files** window opens:

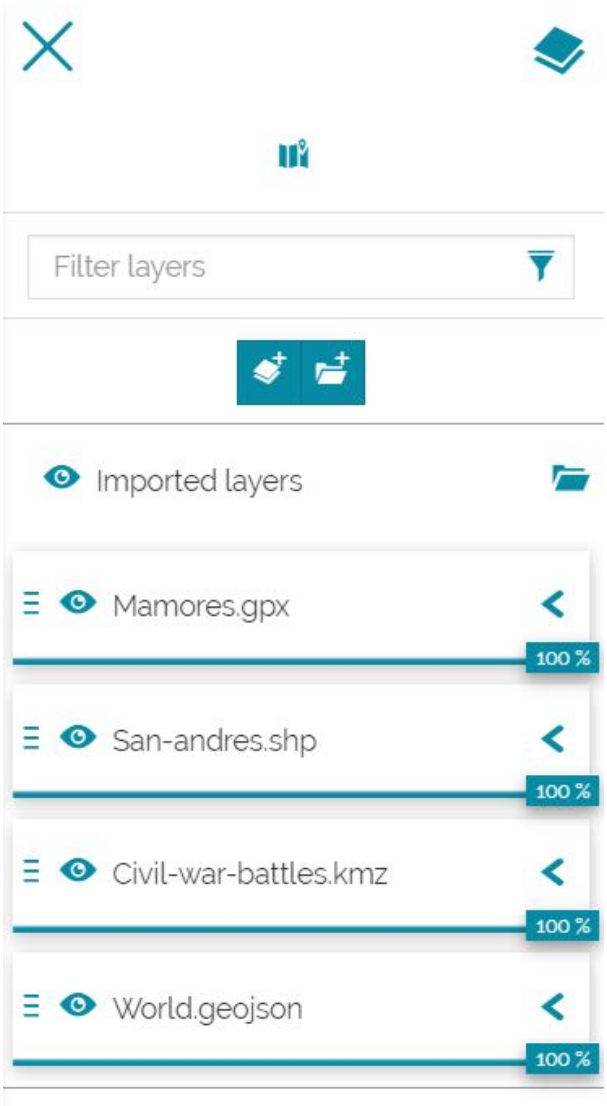


In particular, from this window, it is possible to:

- Choose the layer (when more than one layer is import at the same time)

- Set the layer style or keep the default one
- Toggle the **Zoom on the vector files**

Once the settings are done, the files can be added with the Add button and they will be immediately available in the TOC nested inside the *Imported layers* group. For example:



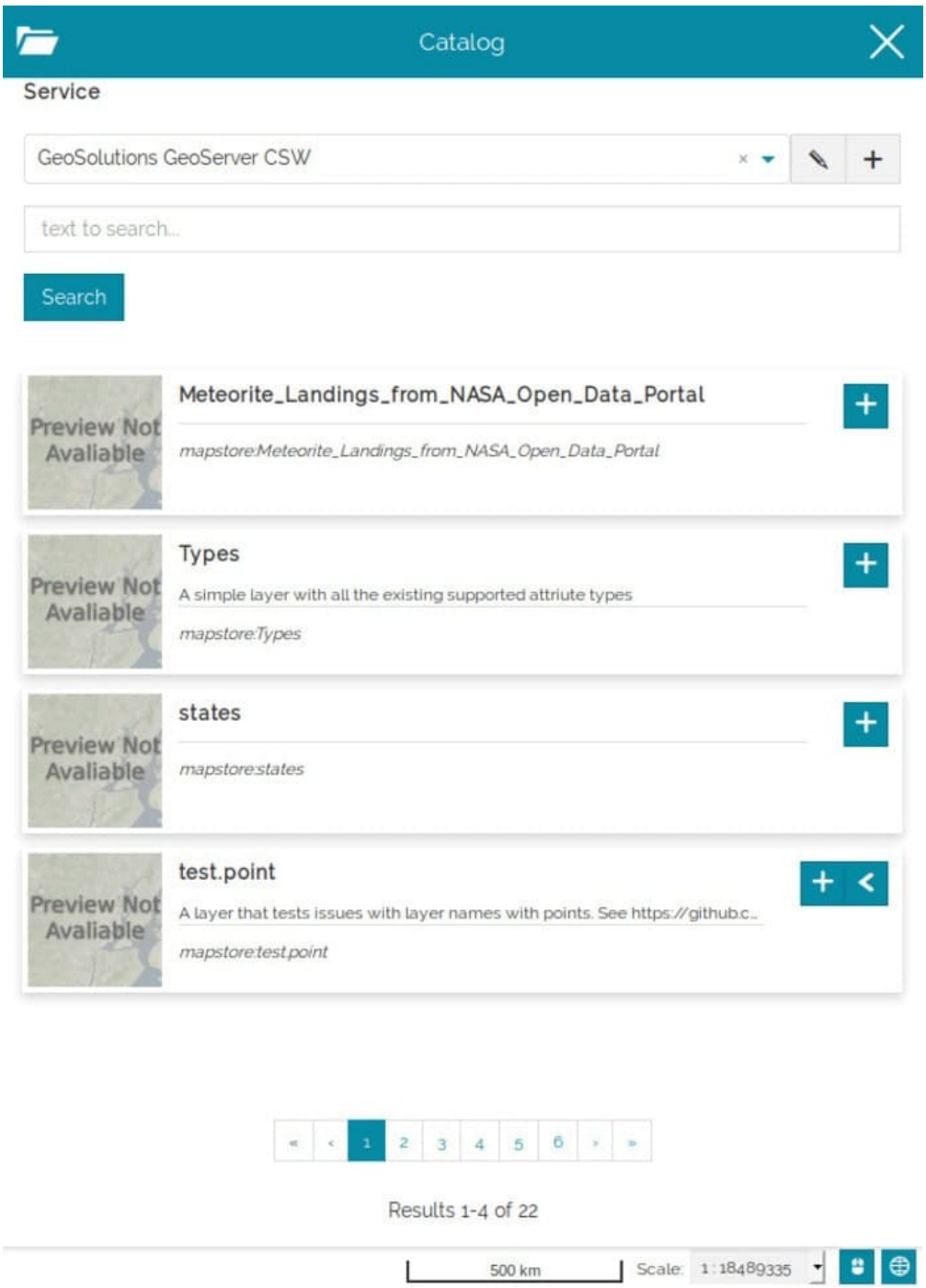
Warning

Currently is not possible to read the Attribute Table of the imported vector files and for this reason also the Layer Filter and the creation of Widgets are not allowed for those layers.

Catalog Services

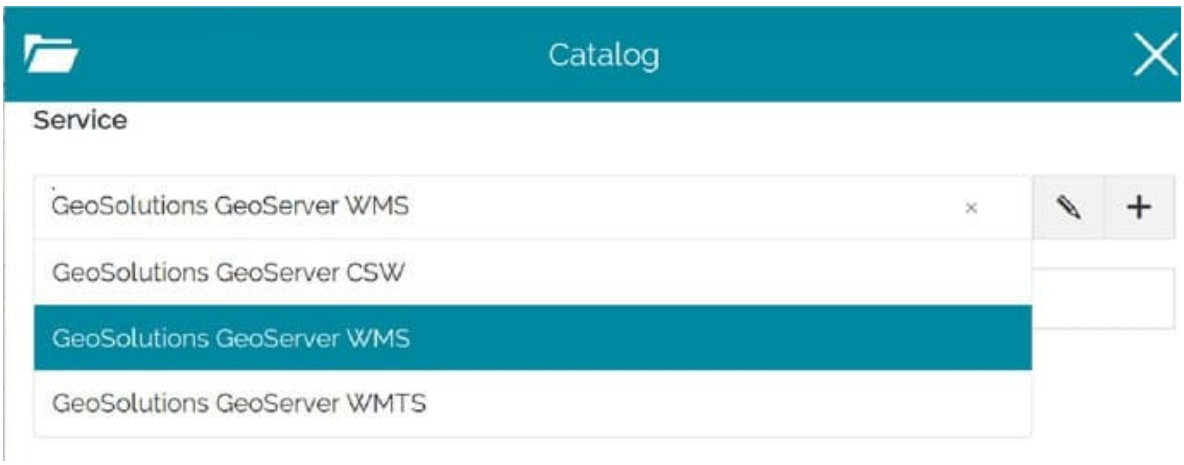
The Catalog Service for the Web (CSW) is an OGC Standard used to publish and search geospatial data and related metadata on the internet. It describes geospatial services such as Web Map Service (WMS) and Web Map Tile Service (WMTS).

In MapStore the Catalog offers the possibility to access WMS, WFS, CSW, WMTS and TMS Remote Services and to add the related layers to the map. By default, as soon as a user opens the Catalog, a CSW a WMS and a WMTS Demo Services are available, allowing to import layers from the GeoSolutions GeoServer. The user can access the Catalog with a click on the button from the Side Toolbar. As soon as you open it, the first display is like the following:

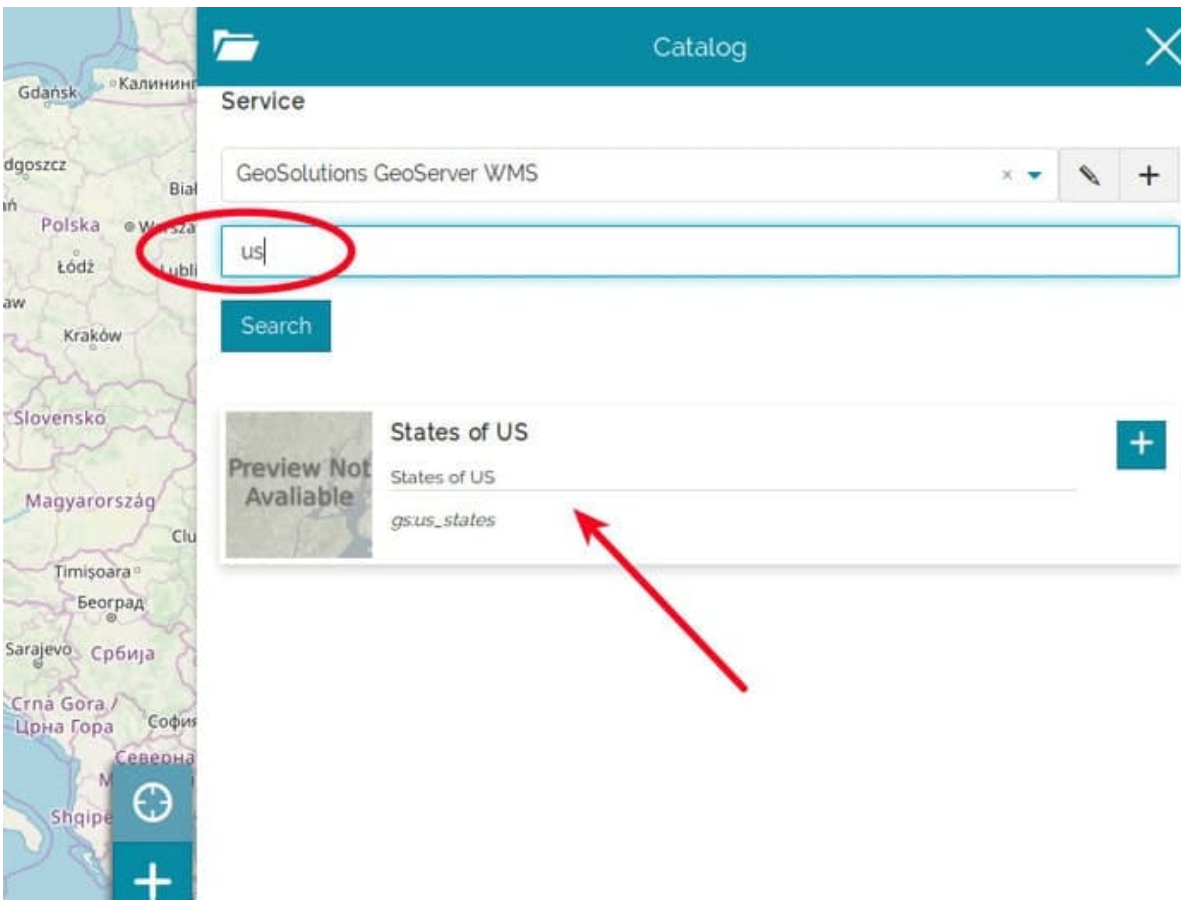


Adding Layers from Remote Services

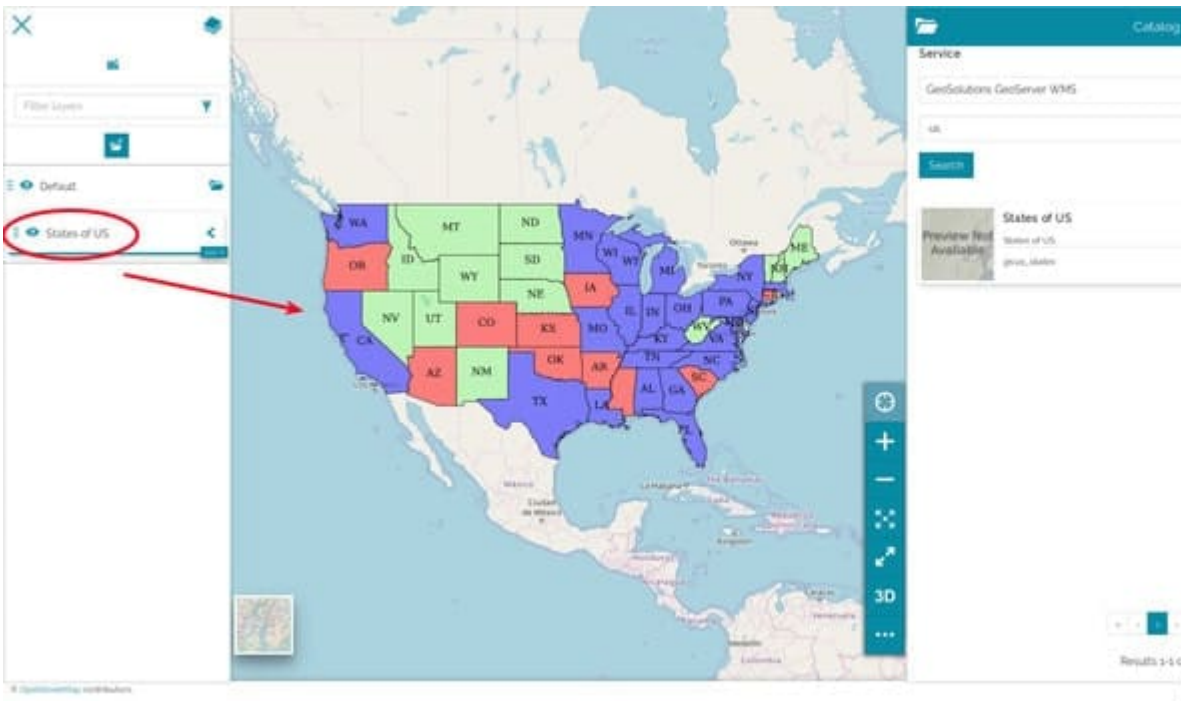
In order to add a layer, the user can first of all open the catalog and choose from the following dropdown menu the Remote Service from where the layer is going to be added:



Once the Remote Service is set, it is possible to search the desired layer by typing a text on the search bar:

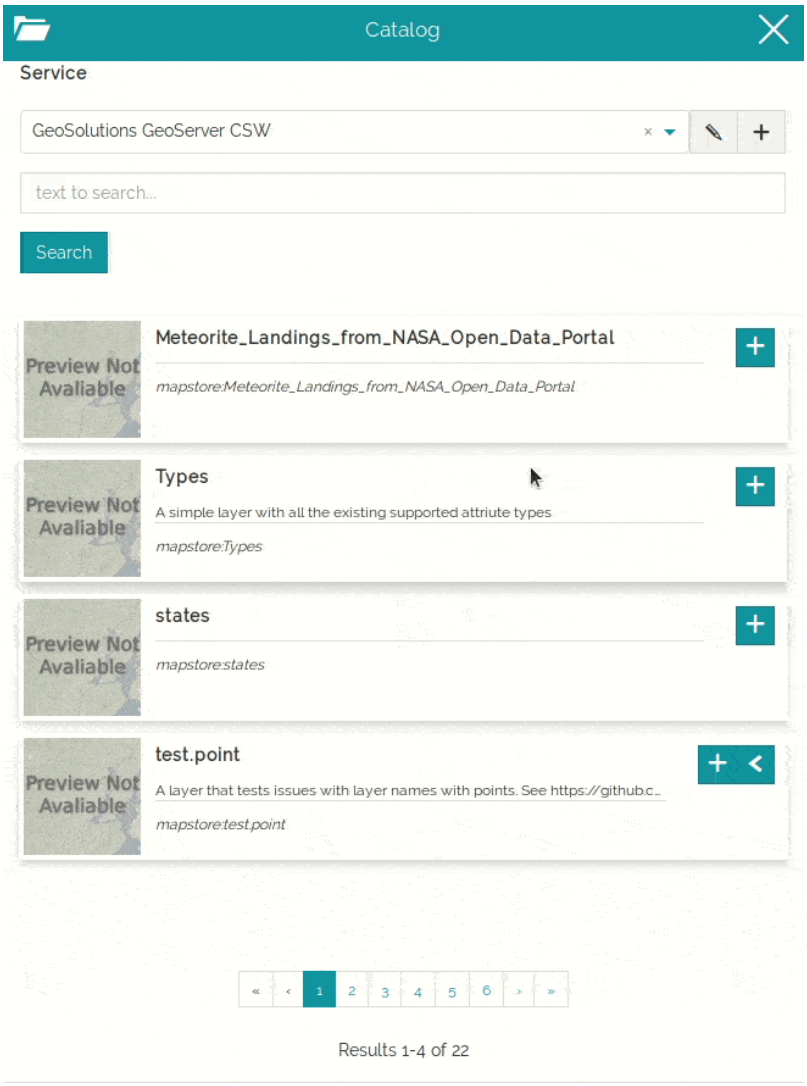


By clicking on the + button, the layer is finally added to the TOC and rendered to the map viewer:


Default
States of US
Service
GeoSolutions GeoServer WMS
Search
States of US
Preview Not Available
3D
WA
MT
ND
MN
OR
ID
SD
WI
MI
WY
NE
IA
NY
NV
UT
CO
KS
MO
IL
IN
OH
PA
CA
AZ
NM
OK
AR
TN
NC
SC
AL
GA
TX
LA
FL
ME

Note

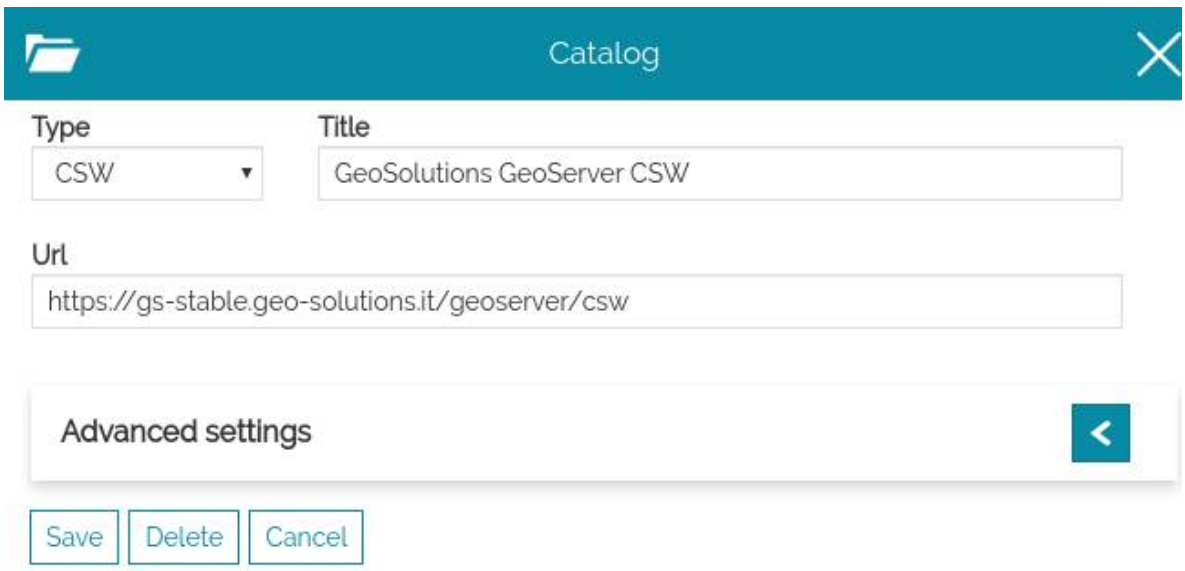
For those layers which have long descriptions or long metadata information, the content is truncated in order to fit the *Layer Card* size. In order to access the complete information, the user can expand the card using the < button:



Managing Remote Services

MapStore allows also to add new Remote Services to the map project (+) or Edit/Remove the existing ones (✎).

The adding/editing process is very similar and the only difference is that editing an existing Service the input fields will be already filled with its settings, while adding a new one all the fields will be empty. Moreover only editing an existing Service, it will be possible to remove it from the Services list.
Editing an existing Service, for example, the first display is the following:



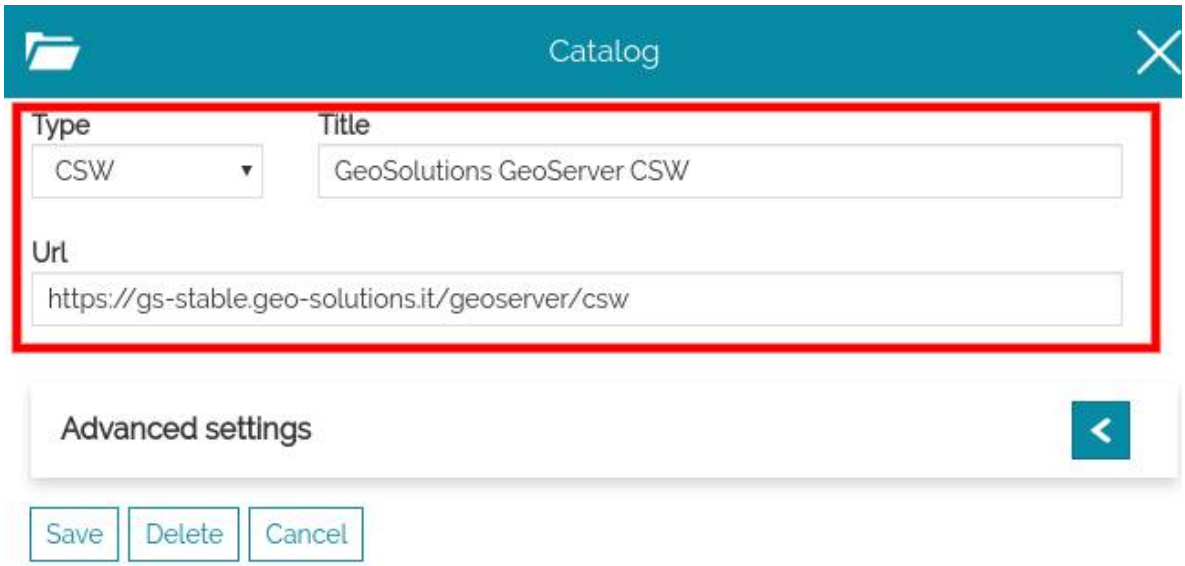
From here the user is allowed to set the Service options, that can be divided into:

- **General settings**
- **Advanced Settings**

Once the options are properly set, it is possible to Save the Service. If the user wants to discard the edits, instead, there's the Cancel button. An existing Service can finally be removed from the Services list through the Delete button (this option is not available creating a new Remote Service).

General settings

The general settings are three mandatory fields that each Remote Service needs to have:

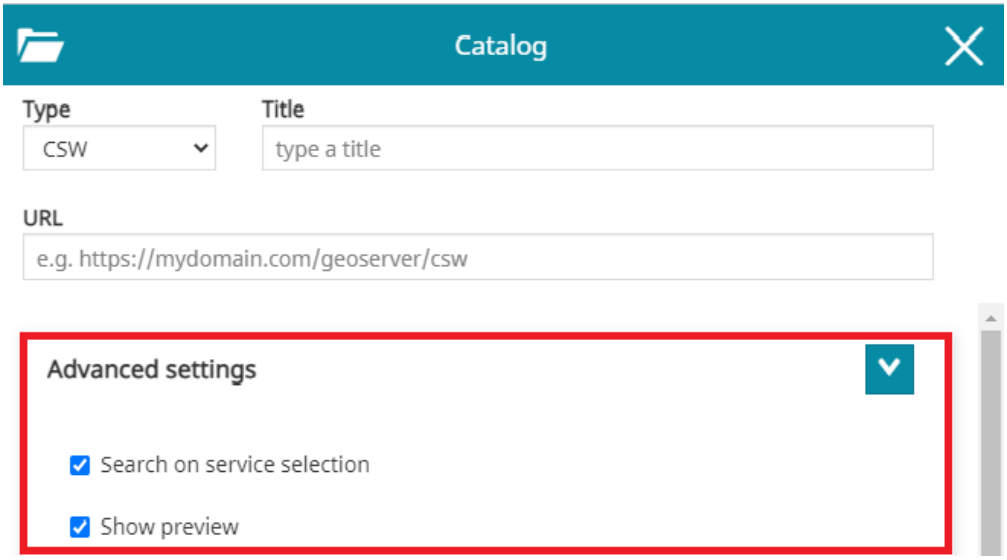


In particular:

- **Url**: the URL of the remote source service
- **Type**: the type of the remote source service (between *WMS*, *WFS*, *CSW*, *TMS*, *WMTS* and *3D Tiles*)
- **Title**: the title to assign to the catalog. This text will be used in the service selection dropdown menu for this service.

Advanced settings

The Advances settings section opens by clicking on the < icon:



The content of Advanced settings depends on the catalog type, but some options are common to all the services types:

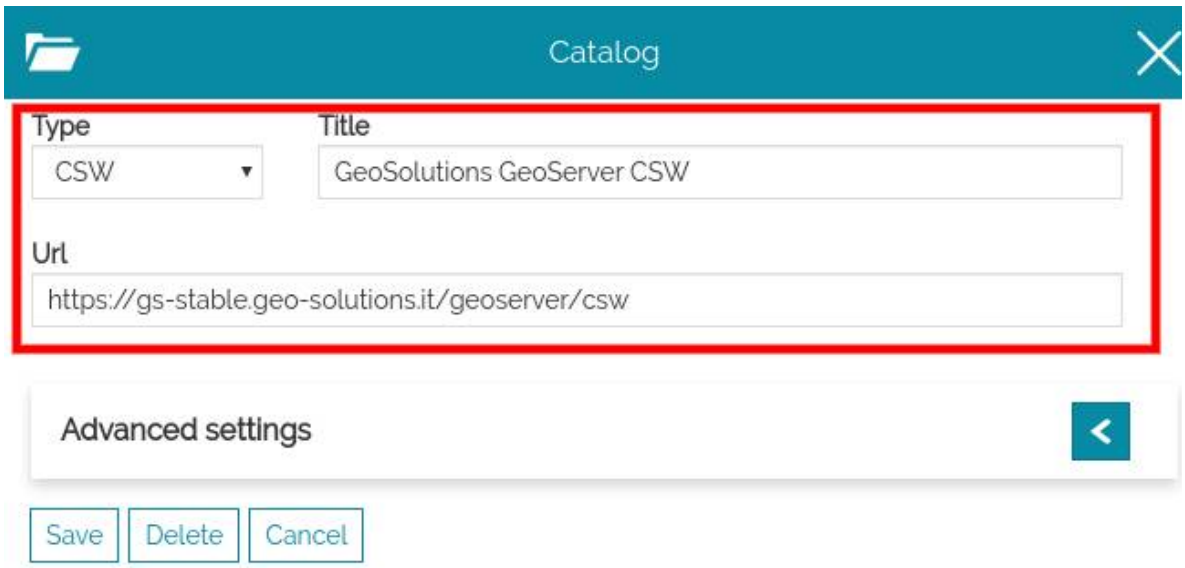
- *Search on service selection* that allow to enable/disable the automatic loading of the catalog records when the user opens that Service
- *Show preview* that can show/hide layers thumbnails in Catalog

Catalog Types

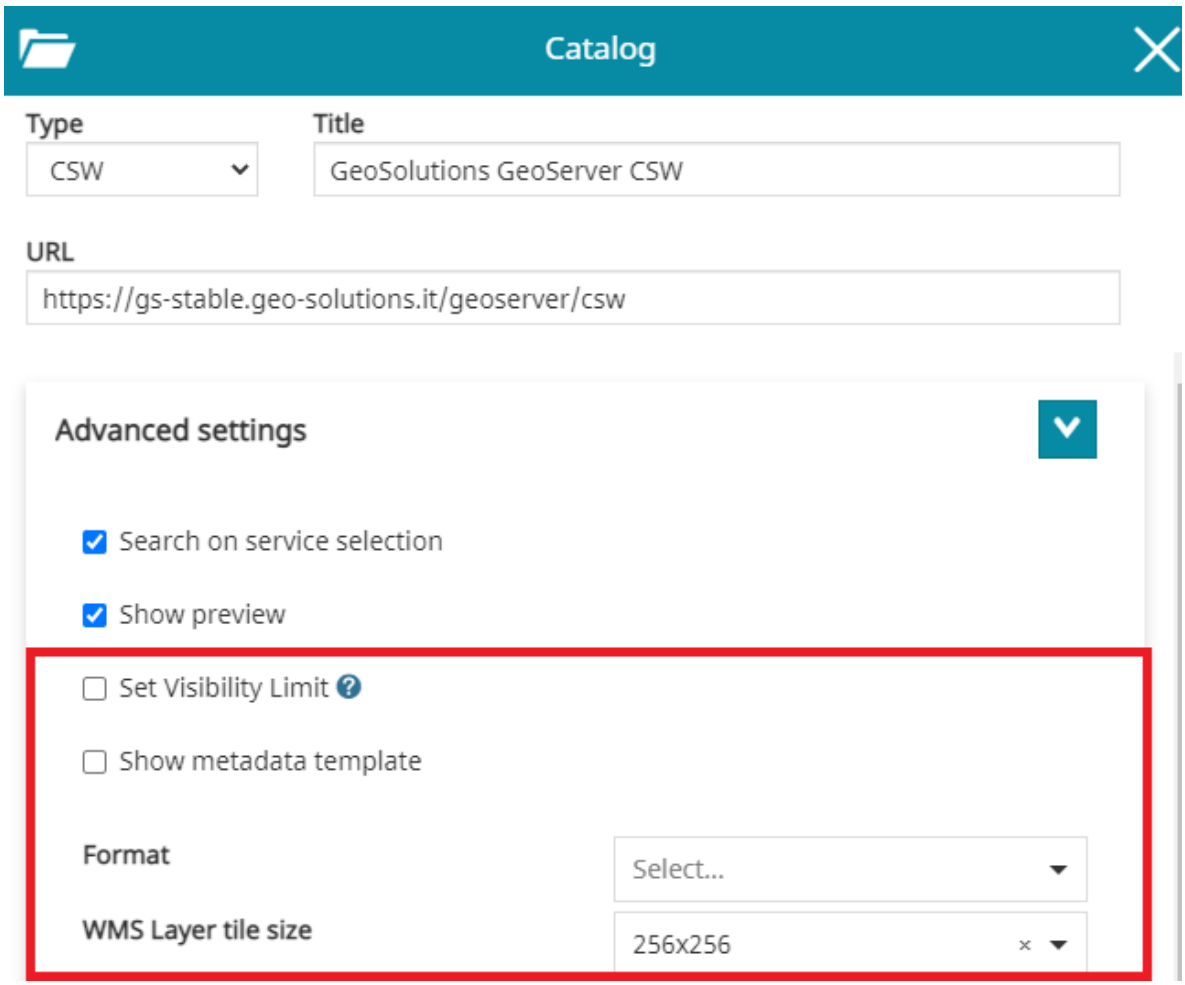
CSW Catalog

The Catalog Service for the Web (CSW) is an OGC Standard used to publish and search geospatial data and related metadata on the internet. It describes geospatial services such as Web Map Service (WMS), Web Map Tile Service (WMTS) and so on... MapStore actually supports only the **Dublin Core** metadata schemas. *ISO Metadata Profile is not supported yet*.

In **general settings of** CSW service the user can specify the title to assign to this service and the URL of the service.



Advanced Settings



- *Format*: the default image format for the layers added to the map (`png` , `png8` , `jpeg` , `vnd.jpeg-png` , `vnd.jpeg-png8` or `gif`). The format configured through this option will be automatically used for all layers

loaded from the involved catalog source (if not configured a default `image/png` is used). For layers already loaded on the map, it is possible to change the format through the Layer Settings tool as usual.

- *Layer tile size*: it represents tile size (width and height) to be used for tiles of all layers added to the map from the catalog source (`256x256` or `512x512`). For layers already loaded on the map, it is possible to change the tile size through the Layer Settings tool as usual.
- *Set Visibility Limit*: if checked and scale limits present in the WMS Capabilities (eg. MinScaleDenominator and/or MaxScaleDenominator), these will be automatically applied to the layer settings when a layer is added to the map from this source.
- *Show metadata template*: this can be enabled when the user wants to insert in the layer description a text with metadata information

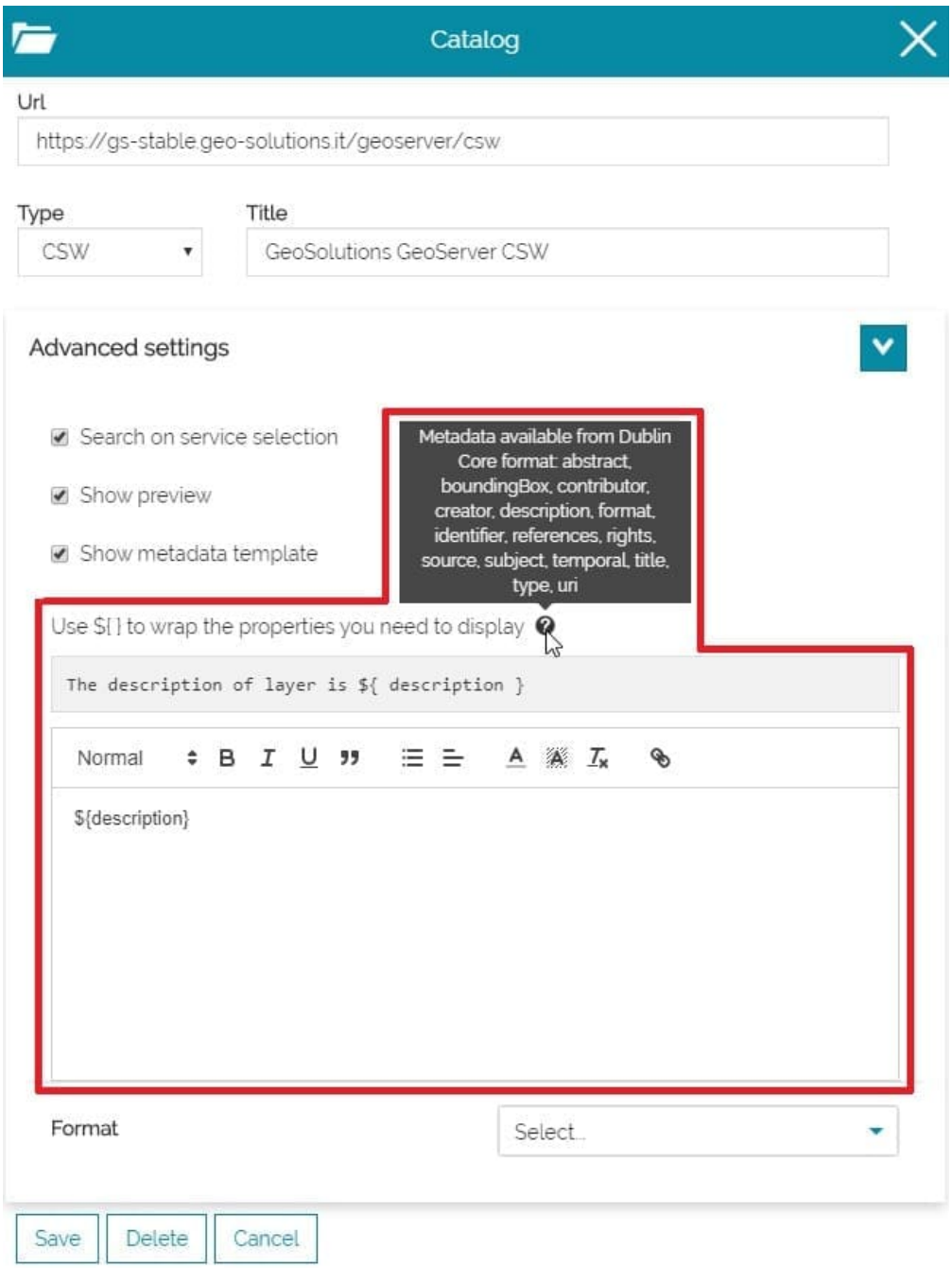
> **Warning**
>
> The *Metadata Template* function is available for **CSW Services** only.

Metadata templates

In order to better understand this function, let's make an example supposing to edit the `GeoSolutions GeoServer CSW` service:

- Change the *Format* of the image that will be rendered on the map (`png` , `png8` , `jpeg` , `vnd.jpeg-png` , `vnd.jpeg-png8` or `gif`) for layers belonging to the selected source
- *Show metadata template* can be enabled when the user wants to insert in the layer description a text with metadata information

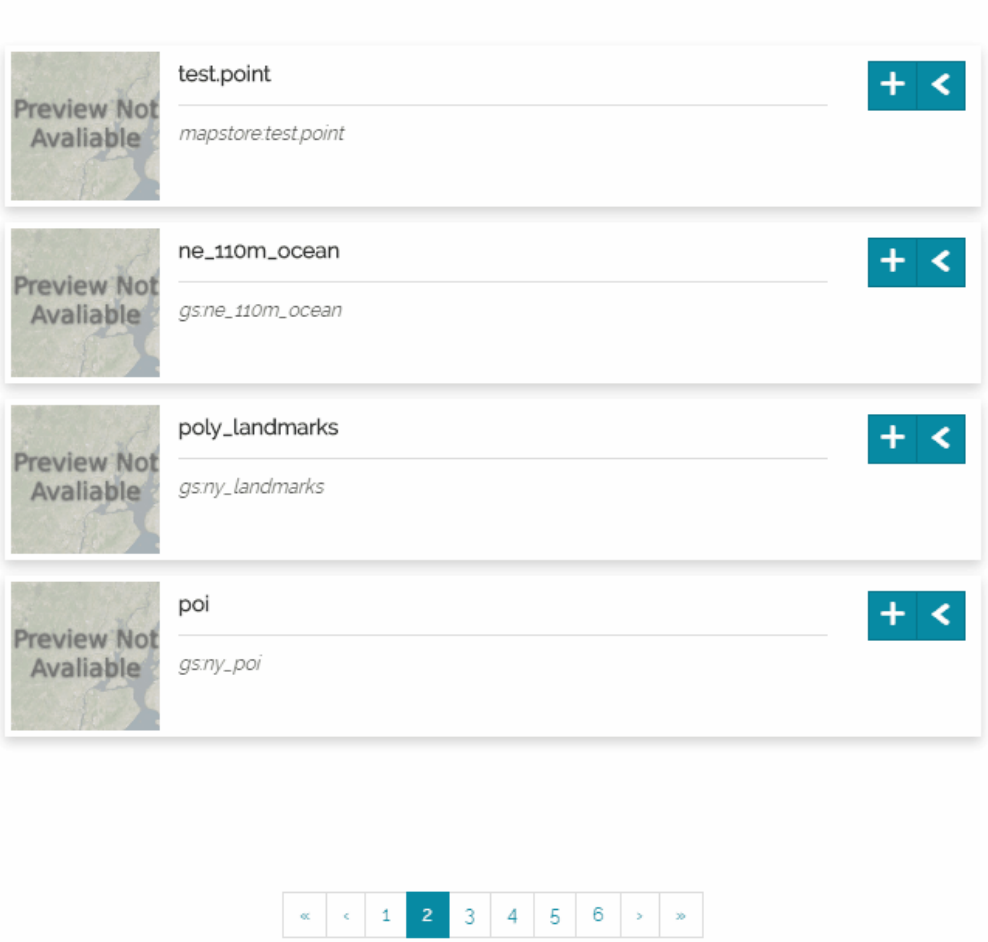
Enabling the *Show metadata template* option appears a text editor through witch it is possible to insert the custom metadata information for that service. In order to dynamically parse each layer's metadata value the user can insert the desired properties name with the format `${property_name}` :



In this case it is possible to add a text like the following, in order to present desired metadata properties:

```
title: ${title}
-------------------
description: ${description}
-------------------
abstract: ${abstract}
-------------------
boundingBox: ${boundingBox}
-------------------
contributor: ${contributor}
-------------------
creator: ${creator}
-------------------
format: ${format}
-------------------
identifier: ${identifier}
-------------------
references: ${references}
-------------------
rights: ${rights}
-------------------
source: ${source}
-------------------
subject: ${subject}
-------------------
temporal: ${temporal}
-------------------
type: ${type}
-------------------
uri: ${uri}
```

Inserting this text and saving, the result should be that each layer will show its properties in catalog with the format we set:



Note

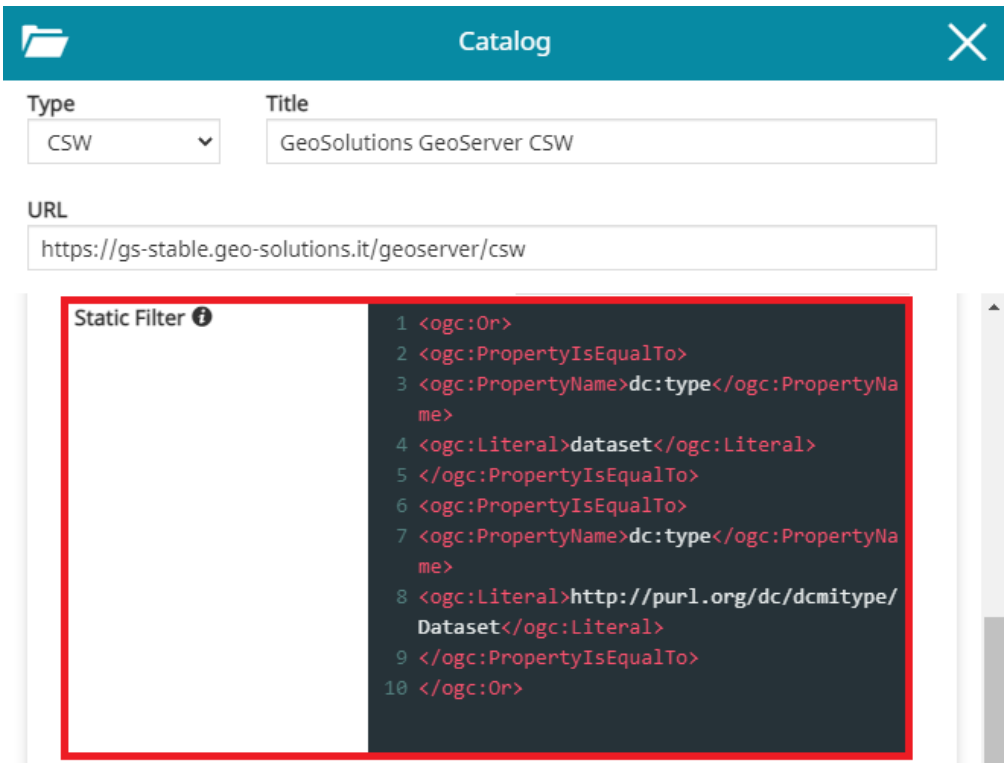
If some metadata are missing, the server response will be `source Not Available`

Static Filter and Dynamic Filter

From the *Advanced Settings* of the *CSW catalog* the user has the possibility to configure a *Static Filter* and a *Dynamic Filter* to customize the search request.

In order to better understand this function, let's make an example supposing to edit the `GeoSolutions GeoServer CSW` service:

- From the *Static Filter* text area it is possible to insert the custom filter for that service.

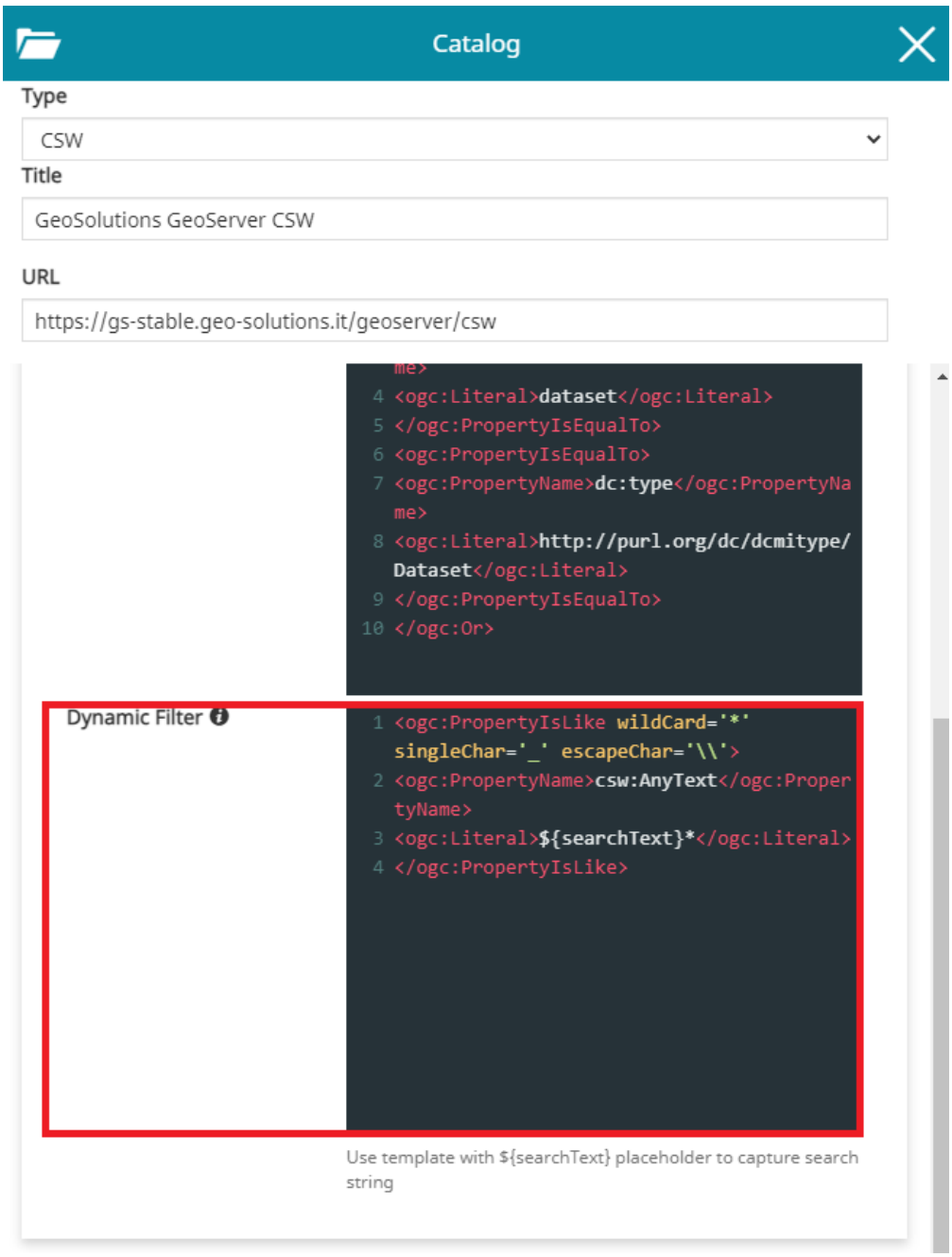


In order to present desired *Static Filter* configuration, it is possible to add a text like the following:

```
<ogc:Or>
    <ogc:PropertyIsEqualTo>
        <ogc:PropertyName>dc:type</ogc:PropertyName>
        <ogc:Literal>dataset</ogc:Literal>
    </ogc:PropertyIsEqualTo>
    <ogc:PropertyIsEqualTo>
        <ogc:PropertyName>dc:type</ogc:PropertyName>
        <ogc:Literal>http://purl.org/dc/dcmitype/Dataset</
ogc:Literal>
    </ogc:PropertyIsEqualTo>
</ogc:Or>
```

Inserting this text and saving. The filter is applied, even in empty search.

- From the *Dynamic Filter* text area it is possible to insert the custom filter to applied in AND with *Static Filter*. The template is used with ${searchText} placeholder to append search string



In this case it is possible to add a text like the following:

```
<ogc:PropertyIsLike wildCard='*' singleChar='_' escapeChar='\\'>
  <ogc:PropertyName>csw:AnyText</ogc:PropertyName>
  <ogc:Literal>${searchText}*</ogc:Literal>
</ogc:PropertyIsLike>
```

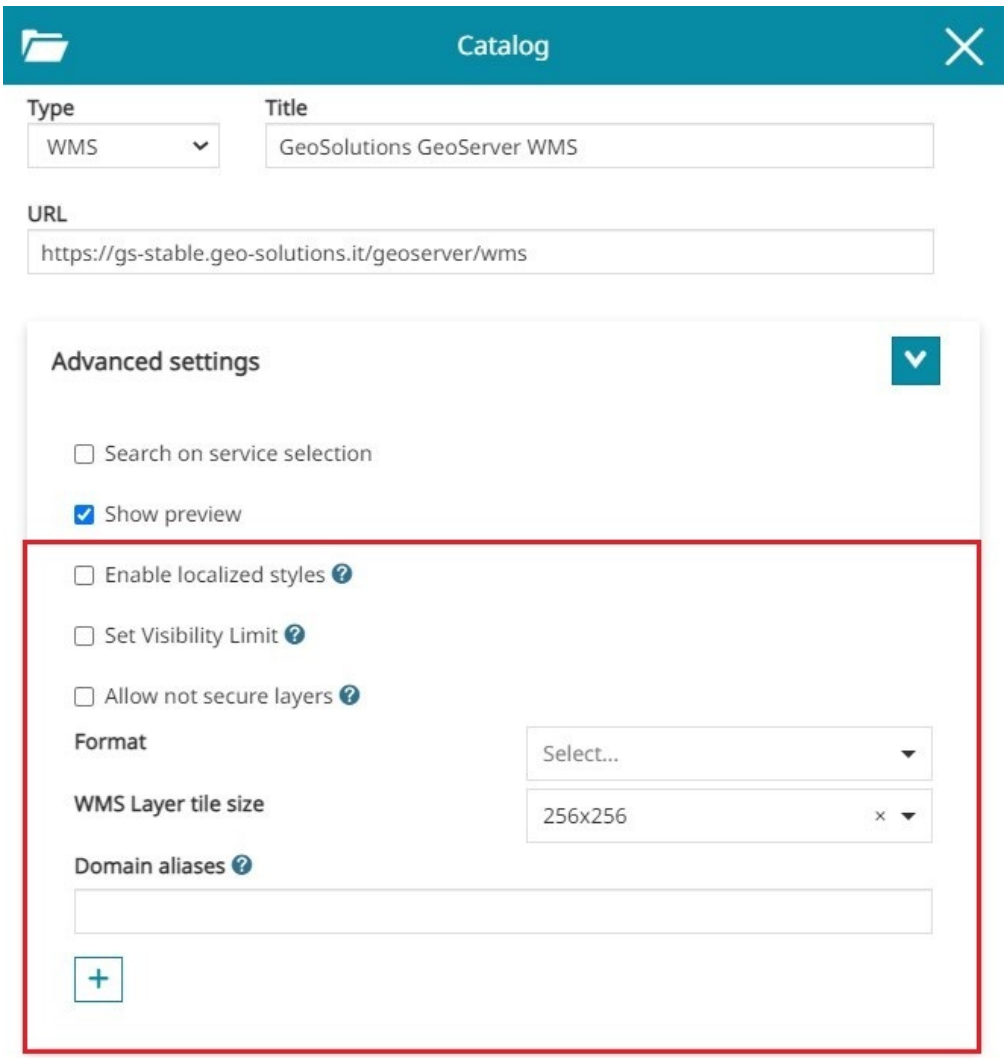
Inserting this text and saving, the filter is applied when text is typed into the service search tool.

WMS/WMTS Catalog

WMS and WMTS Services are OGC Standards protocol for publishing maps (and tile maps) on the Internet. The user can add these kind of services as catalogs to browse and add to the map the layers published using these protocols.

In **General Settings** the user can set the title he wants to assign to this service and the URL of the service to configure the service and its URL.

In addition to the standard options, only for WMS catalog sources, through the **Advanced Settings** the user can configure also the following options:



- *Localized styles* (only for the WMS service) if enabled allows to include the MapStore's locale in each **GetMap**, **GetLegendGraphic** and **GetFeatureInfo** requests to the server so that the WMS server, if properly configured, can use that locale to:
- Use localized lables for Tiles in case of vector layers (the layer's style must be properly configured for this using the ENV variable support)
- Produce a localized layer legend in case of vector layers (the layer's style must be properly configured to use the Localized tag for rule titles)
- Produce a localized output for GetFeatureInfo requests (the freemarker template need to be properly configured to retrieve the locale from the request)
- *Set Visibility Limit*: available only for WMS layers coming from CSW or WMS catalog sources type. If checked and scale limits present in the WMS

Capabilities (eg. MinScaleDenominator and/or MaxScaleDenominator), these will be automatically applied to the layer settings when a layer is added to the map from this source

- *Allow not secure layers*: if enabled allows the unsecure catalog URLs to be used (http only). Adding layers from WMS sources with this option active will also force the layer to use the proxy for all the requests, skipping the mixed content limitation of the browser.

Enabling that option, all layers added to the map from this catalog source will be localized as described above (it is possible to tune again that setting for each single layer by opening the Layer Settings in TOC).

- *Format*: the default image format for layers added to the map (`png` , `png8` , `jpeg` , `vnd.jpeg-png` , `vnd.jpeg-png8` or `gif`). The format configured through this option will be automatically used for all layers loaded from the involved catalog source (if not configured a default `image/png` is used). For layers already loaded on the map, it is possible to change the format through the Layer Settings tool as usual.

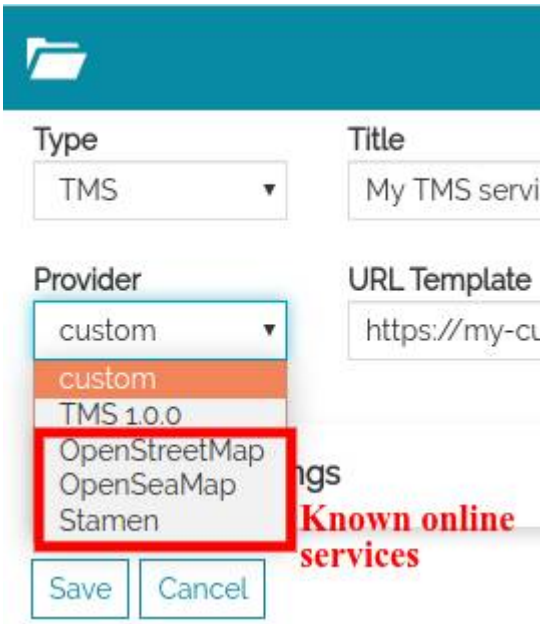
> **Note**
>
> In case of WMS services, the list of available formats is retrieved from the WMS server

- *Layer tile size*: it represents tile size (width and height) to be used for tiles of all layers added to the map from the catalog source (`256x256` or `512x512`). For layers already loaded on the map, it is possible to change the tile size through the Layer Settings tool as usual.
- *Domain aliases*: available only for WMS catalogs type. This option is used to improve the performances of the application for tiled layer requests when multiple domains can be defined server side for the configured catalog source in MapStore (domain sharding). The user can configure multiple URLs referring to the same WMS service through the **Add alias** + button. Useful information about other kind of performance improvements can be found in the MapStore online training documentation.

TMS Catalog

The Tile Map Service (TMS) specifications include some not official/not standard protocol for serving maps as tiles (i.e. splitting map up into a pyramid of images at multiple zoom levels). MapStore allows to add to the map the following services providers:

- Custom TMS service, specifying the URL template for the tiles.
- TMS 1.0.0 , setting the URL
- Select from a list of known TMS services, with all the variants.



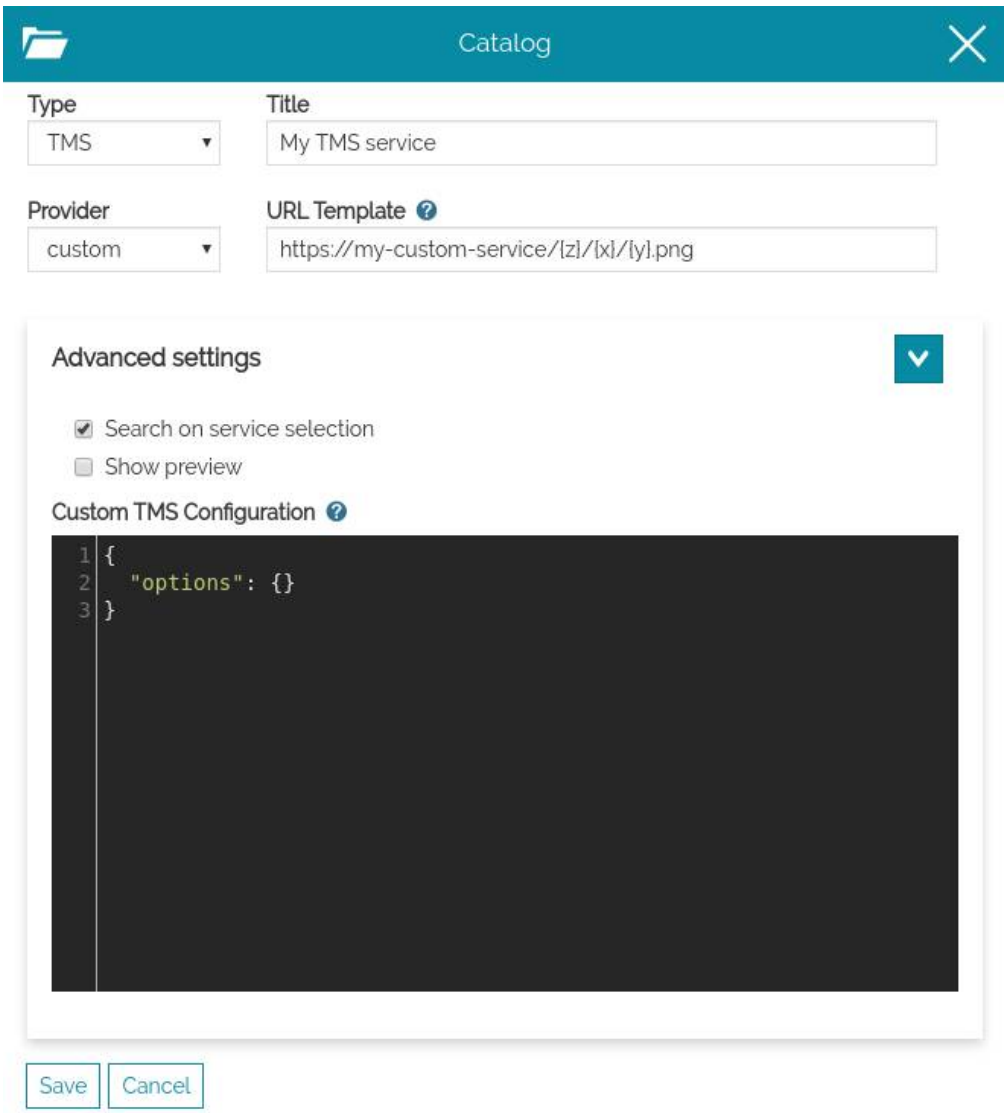
Select provider for TMS. The list of providers contains "custom", "TMS 1.0.0" and other resources

Note

Since some of these services are not standard, using them in different CRSs may cause problems. Therefore, keep in mind that changing CRS can cause problems when these levels are on the map.

Custom TMS

Selecting the **custom** provider the user can insert the tile URL template manually. The URL template is an URL with some placeholder that will be replaced with variables. The placeholder are identified by strings between brackets. e.g.: `{variable_name}`.



Edit a custom TMS

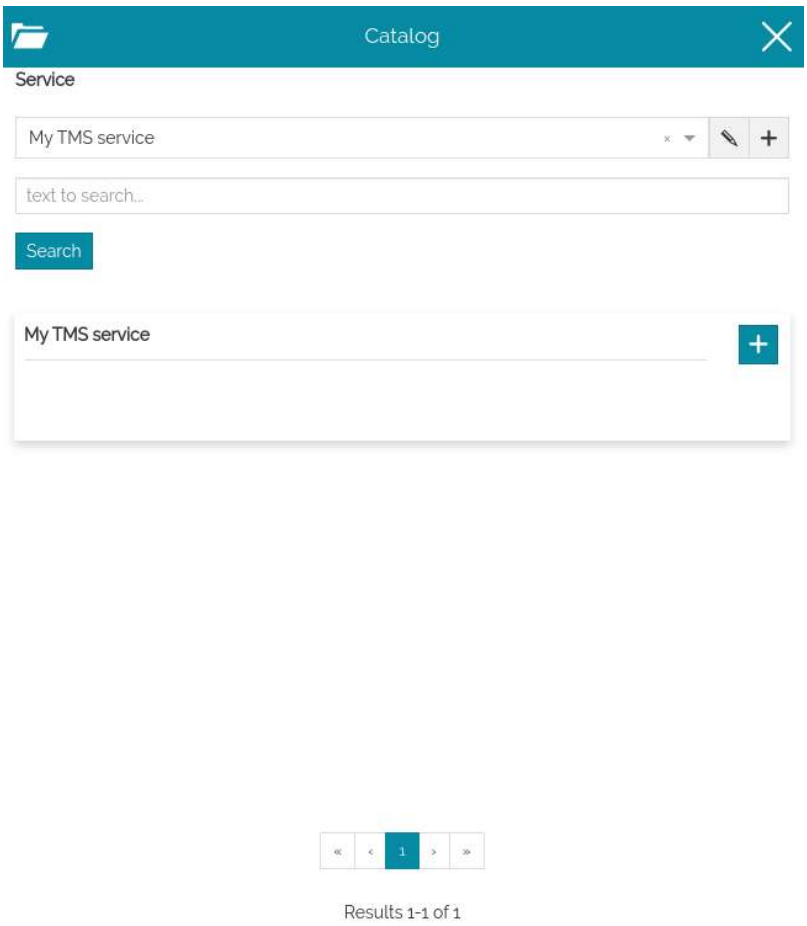
Allowed placeholder are:

- `{x}`, `{y}`, `{z}`: coordinates of the tiles
- `{s}`: subdomains, this provides support for *domain sharding*. By default this is `["a", "b", "c"]`. User can customize the default by adding options.subdomains.

example:

```
{
  "options": {
    "subdomains": ["a", "b", "c", "d", "e"]
  }
}
```

When the user saves this custom catalog service and clicks on search, he will see only one result, that can be added on the map: variants are not currently sopported in MapStore for this provider type.



Browse custom TMS service. It contains only one result

Sample custom

```
url: https://{s}.tile.opentopomap.org/{z}/{x}/{y}.png
```

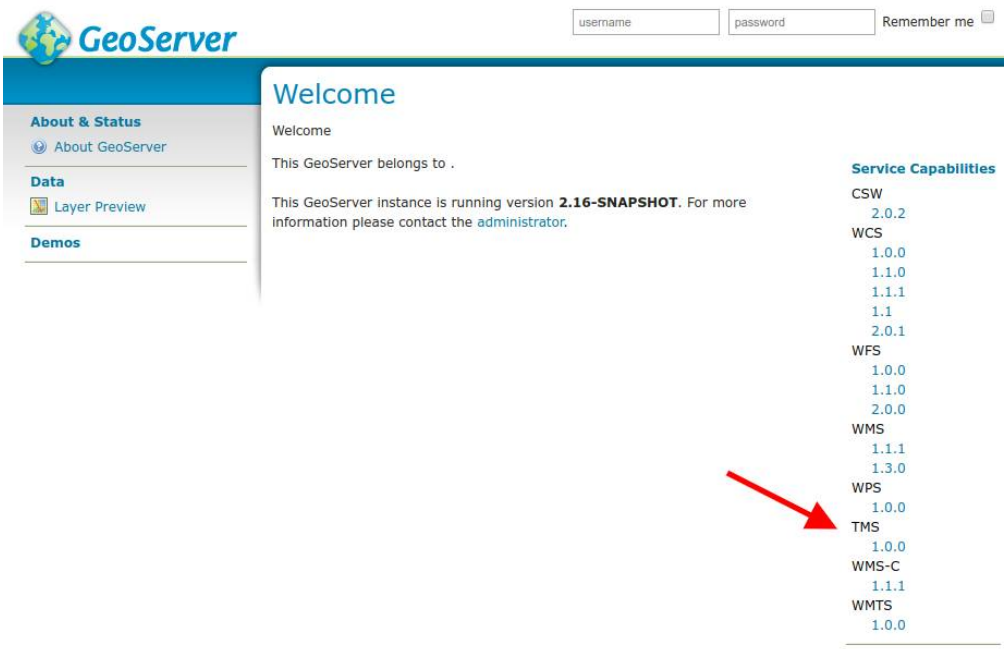
Sample custom with advanced options

```
url: https://nls-{s}.tileserver.com/nls/{z}/{x}/{y}.jpg
```

```
{
  "options": {
    "subdomains": [
      "0",
      "1",
      "2",
      "3"
    ]
  }
}
```

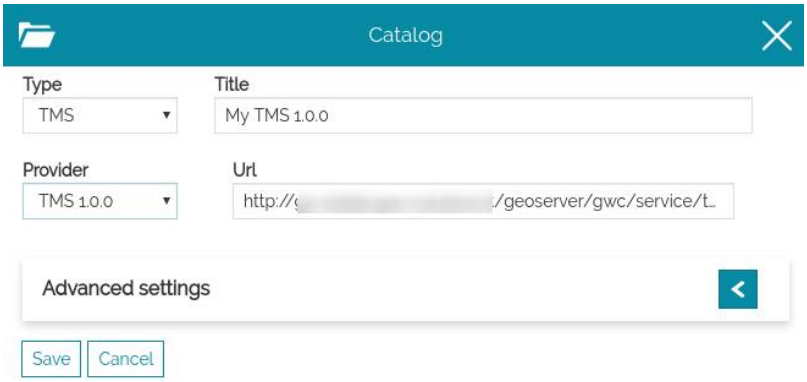
TMS 1.0.0

Selecting the "TMS 1.0.0" provider the user can insert the URL of the Tile Map Service (see TMS Specification). For instance, in GeoServer, it is the URL of the "TMS" link in the home page.

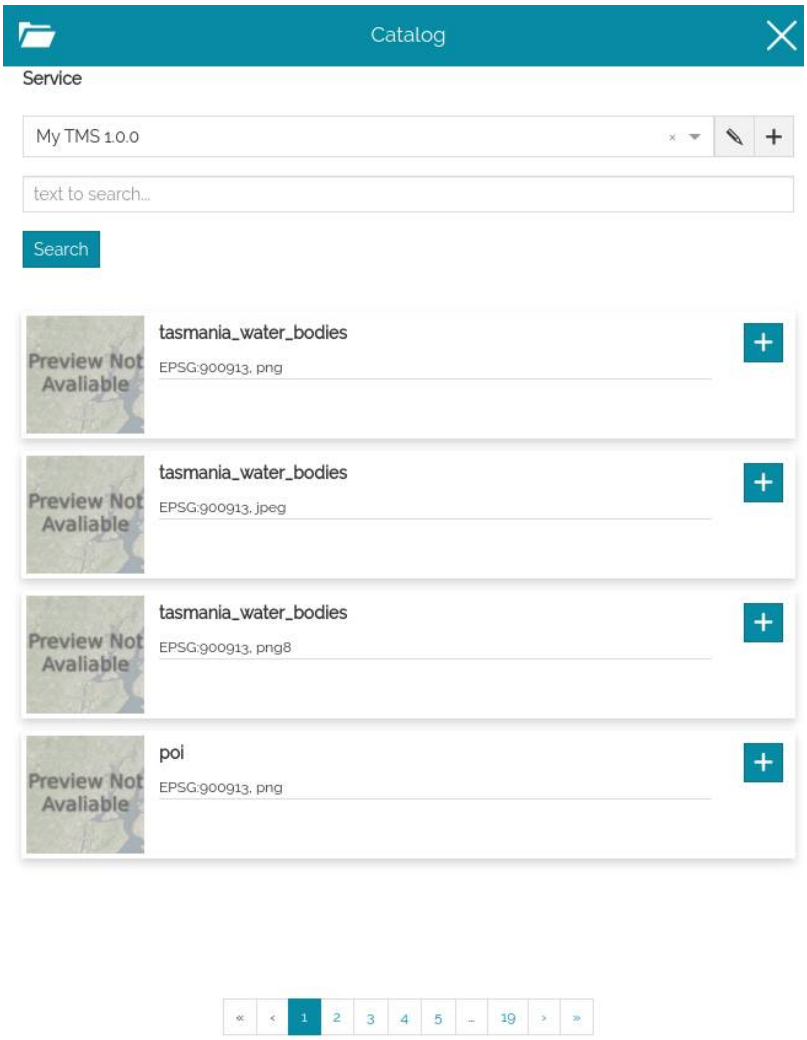


TMS 1.1.0 URL from GeoServer

When saved this, the user will be allowed to browse and add to the map the TMS layers provided by the service. MapStore will filter the layers published showing only the tile maps in the current EPSG.



Edit a TMS 1.0.0 provider



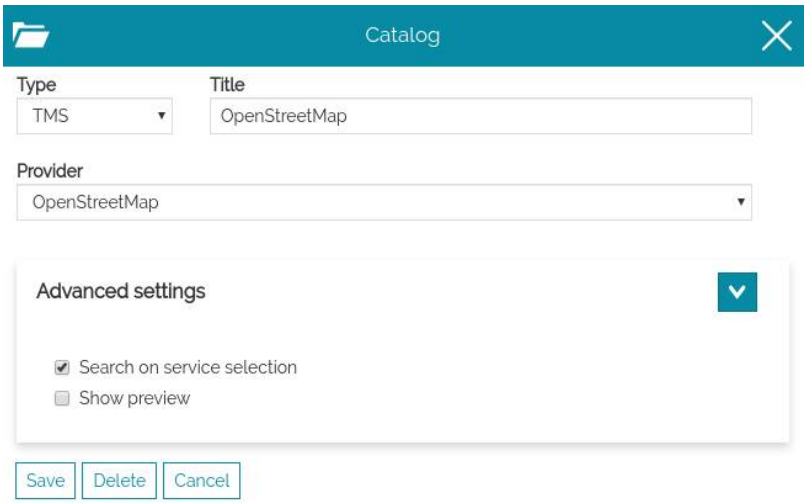
Browse TMS 1.0.0 layers

sample TMS 1.0.0 services

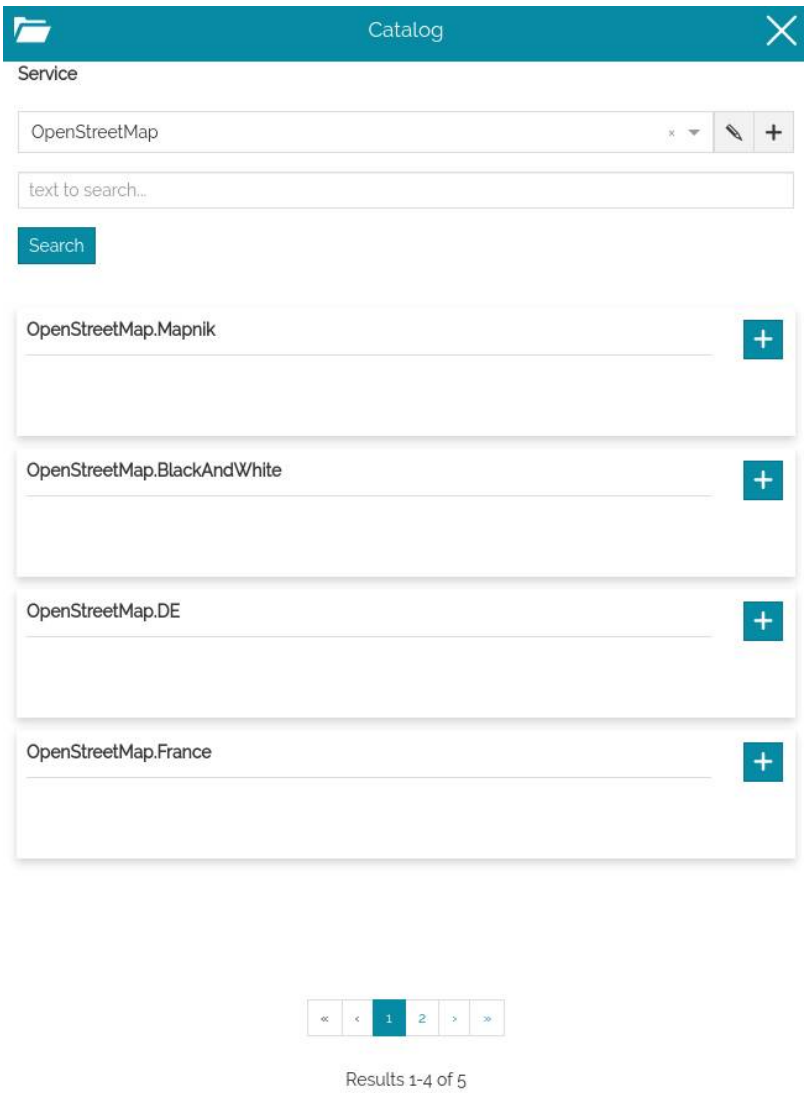
```
https://public.sig.rennesmetropole.fr/geowebcache/service/tms/
1.0.0
https://osm.geobretagne.fr/gwc01/service/tms/1.0.0
http://gs-stable.geo-solutions.it/geoserver/gwc/service/tms/
1.0.0
```

TMS Known Services

The other known services are listed as providers below "custom" and "TMS 1.0.0". They are a static list configured inside the application. Selecting one of the provider listed and saving the new catalog service allows to browse al the variants known for that service. For more information about the list of available providers, see the developer documentation about Tile Providers



Select a known TMS provider

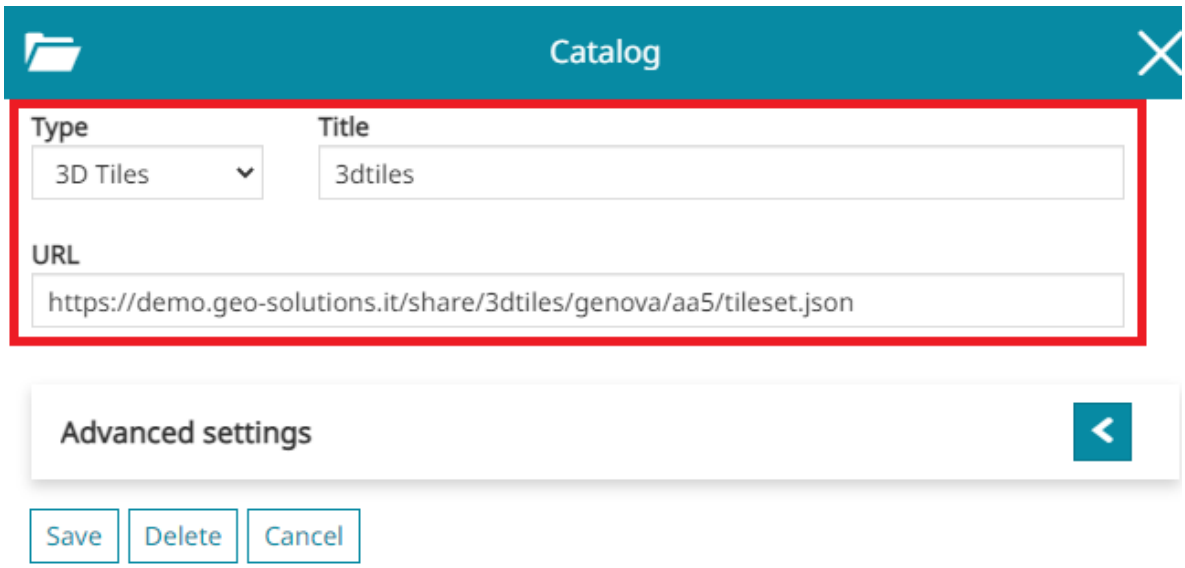


Browse the TMS variants

3D Tiles Catalog

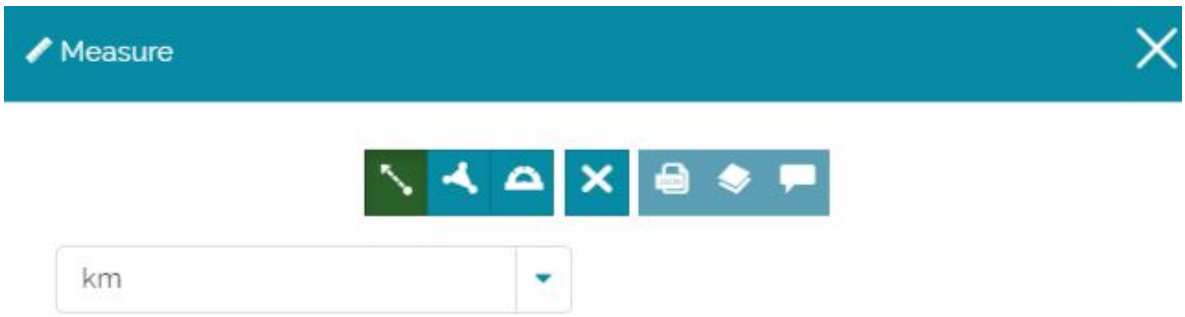
3D Tiles is an OGC specification designed for streaming and rendering massive 3D geospatial content such as Photogrammetry, 3D Buildings, BIM/CAD, and Point Clouds across desktop, web and mobile applications. MapStore allows to publish 3D Tiles contents in its 3D mode on top of the CesiumJS capabilities. Through the Catalog tool, a specific source type to load 3D Tiles in the Cesium Map can be configured as follows by specifying the URL of a reachable *tileset.json*.

In **general settings of** 3D Tiles service, the user can specify the title to assign to this service and the URL of the service.


Catalog
Type
3D Tiles
Title
3dtiles
URL
https://demo.geo-solutions.it/share/3dtiles/genova/aa5/tileset.json
Advanced settings
Save
Delete
Cancel

Performing Measurements

The Measure tools in MapStore allows the user to perform distance, area and bearing measurements on the map. It also provides some additional functionalities that are described in this section of the documentation. The tool is accessible from Side Toolbar by selecting the button that opens the following window:



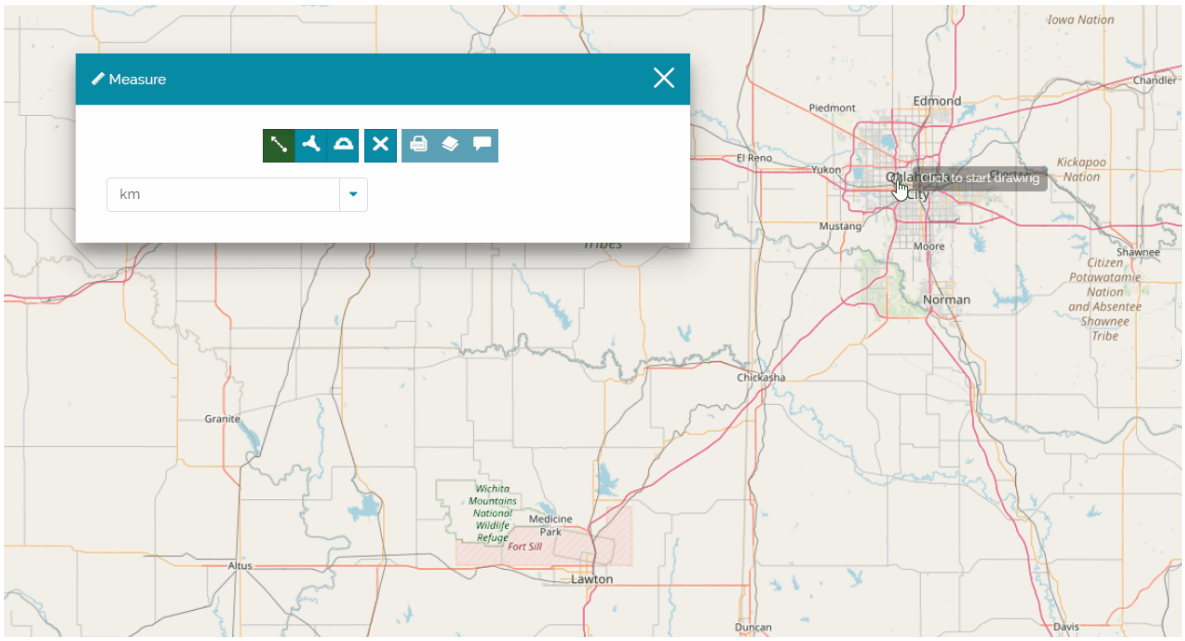
Through this window it is possible to:

- Measure distance
- Measure Area
- Measure Bearing
- Clear measures
- Export the measures to *GeoJSON*
- Add the measure as a layer in TOC
- Add the measure as an Annotation

> **Note**
>
> The user can perform more than one measurement simultaneously on the map and then cancel it with the **Clear Measures** button

Measure distance

As soon as the measure window opens, by default the measure distance option is selected  . In order to perform a distance measure, each click on the map correspond to a segment of the line (at least one segment is needed) while the double click inserts the last line segment and ends the drawing session.



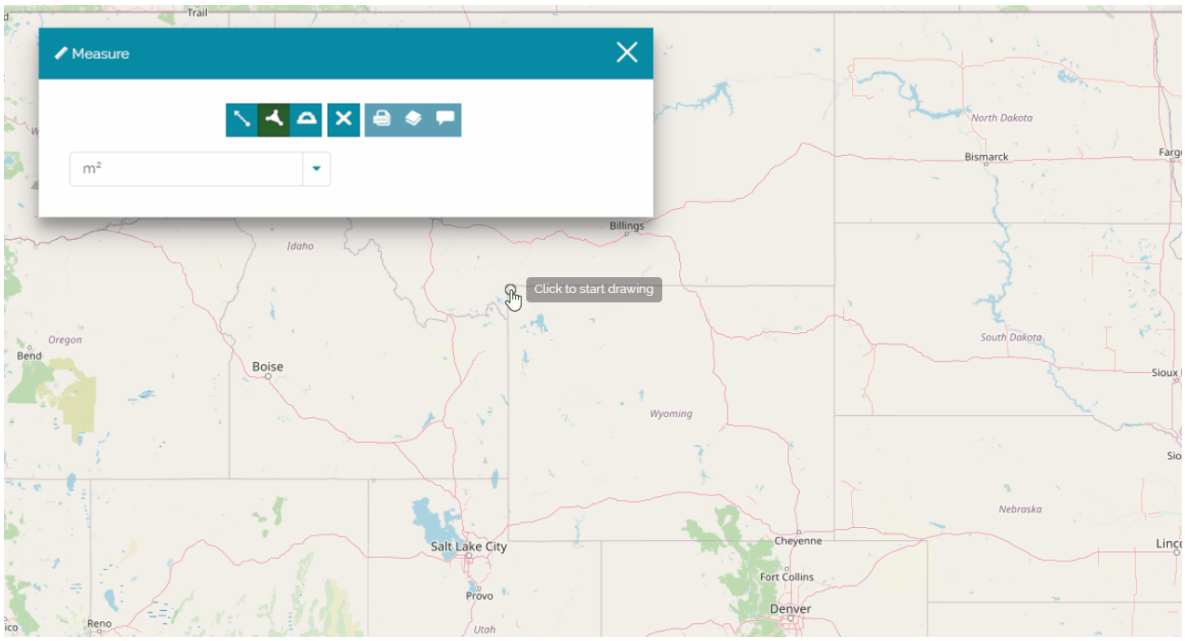
The available units of measure are:

Note

The length of a line segment is shown on the map together with the measurement of the overall distance of all segments.

Measure area

Once the **Measure Area** button is selected  , it is possible to start the drawing session (in this case at least 3 vertices need to be indicated). Same as measuring the distance, each click correspond to a vertex and the double click will indicate the last one.



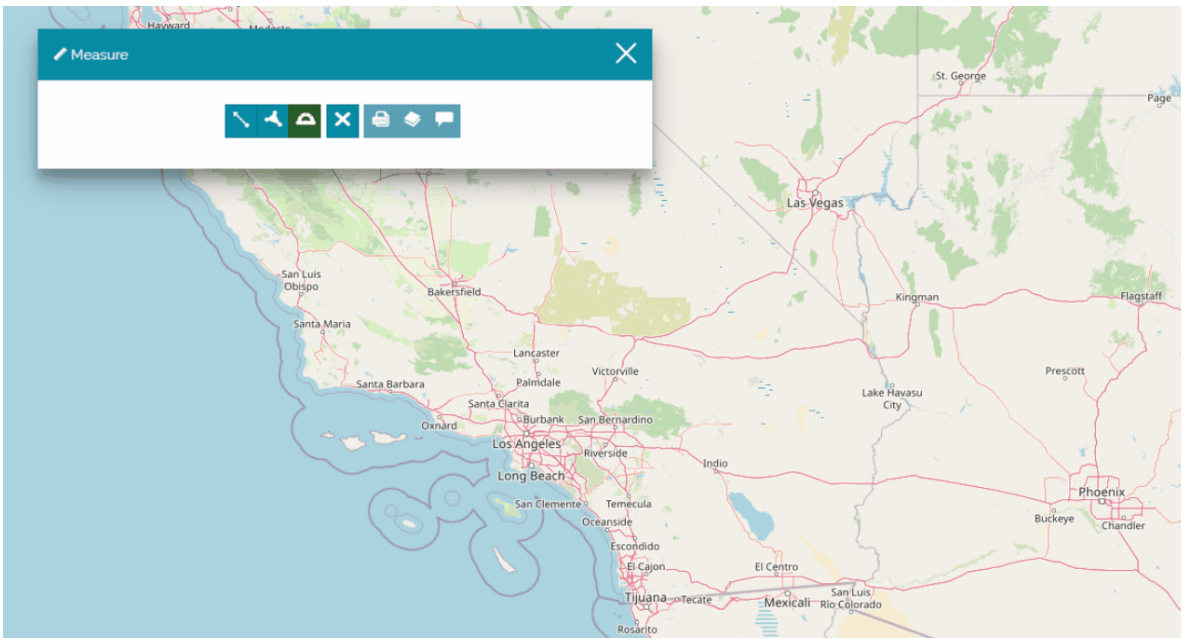
In this case the available units of measure are:

Note

Each side's length of the polygon is reported in map along with its perimeter and the area.

Measure bearing

The Bearing measurements allows you to measure directions and angles. In the quadrant bearing system, the bearing of a line is measured as an angle from the reference meridian, either the north or the south, toward the east or the west. Bearings in the quadrant bearing system are written as a meridian, an angle and a direction. For example, a bearing of N 30 W defines an angle of 30 degrees west measured from north. A bearing of S 15 E defines an angle of 15 degrees east measured from the south.
After selecting the **Measure Bearing** button  the user can draw a line with only two vertices that indicates respectively the starting and the ending point.

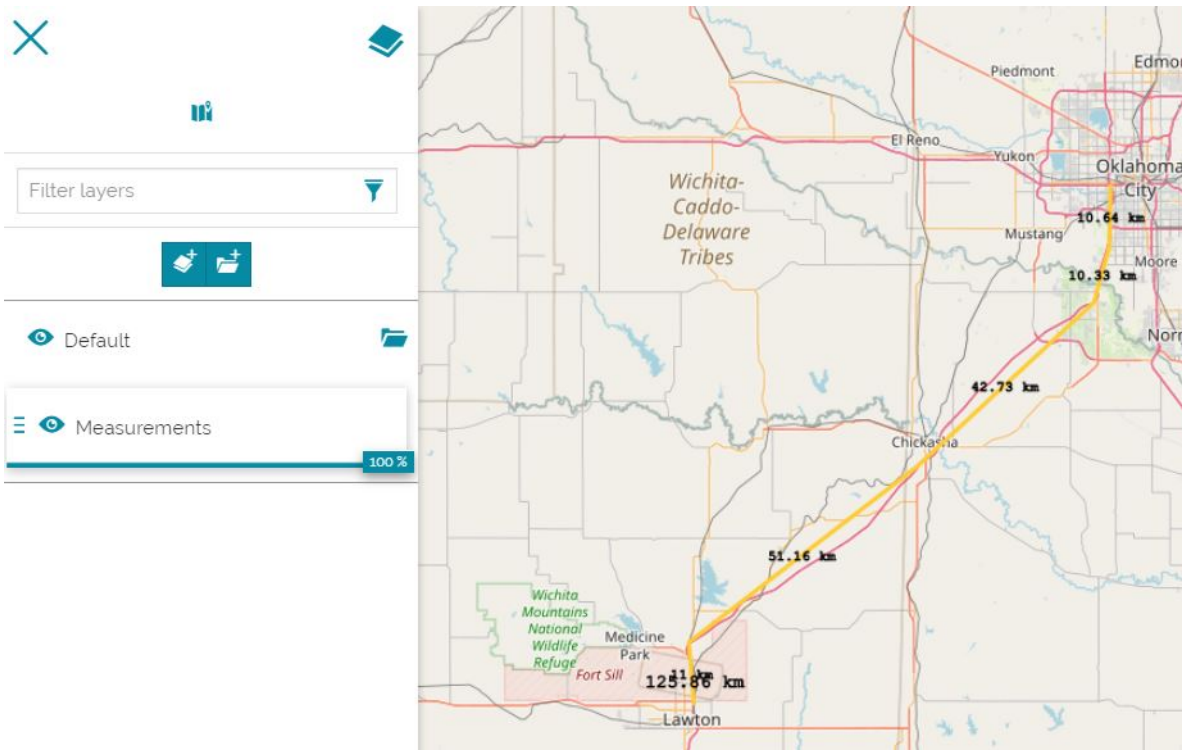


Export the measure

Measurements drawn on the map can be exported in `GeoJson` format through the  button.

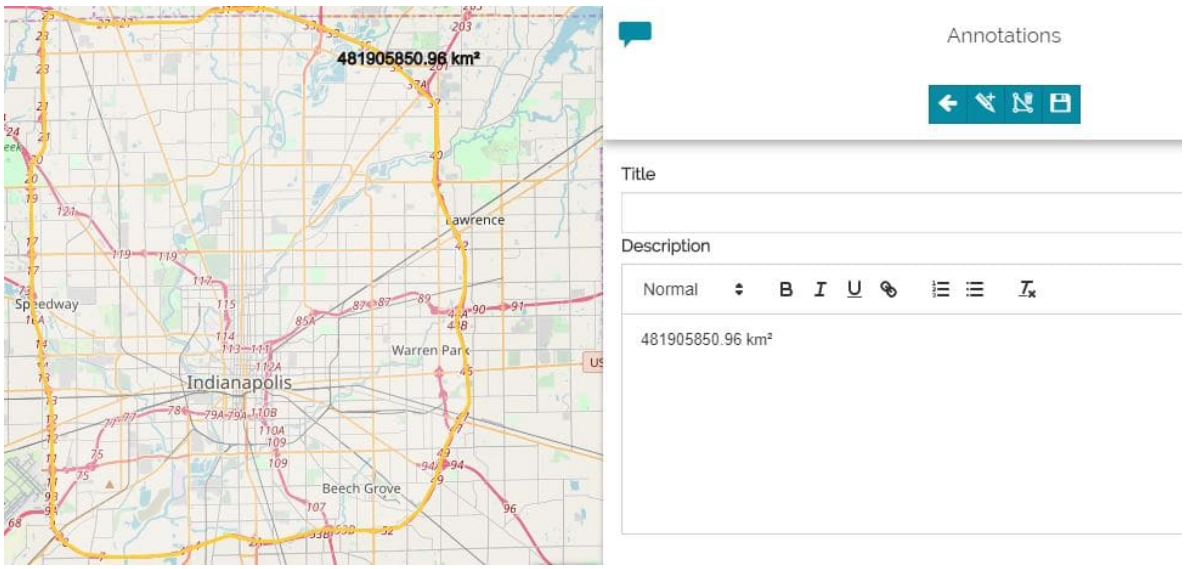
Add the measure as layer

Once a measure is drawn, it is possible to add it as a layer through the  button. The created layer is added to the Table of Contents as follows:



Add measure as annotation

Once a measure is drawn, it is possible to add it as an Annotation through the button. The following panel opens:



From this step the creation process is the same described in the Annotations section.

Annotations

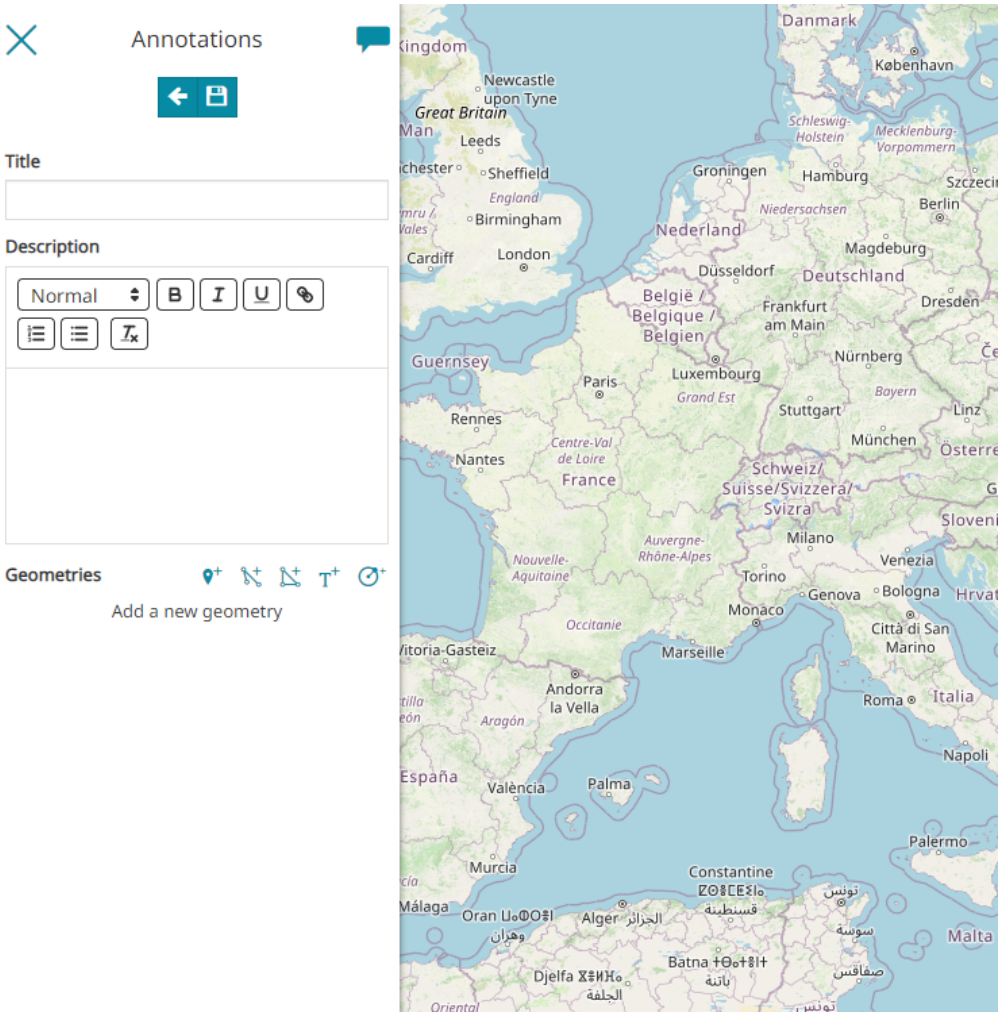
Mapstore lets you enrich the map with special features which expose additional information, mark particular position on the map and so on. Those features make up the so called **Annotations** layers.

Starting from a new map or an already existing one, the editor can access the **Annotations** button from the TOC panel on the top-left corner of the map viewer.

The annotation panel will open:

Add new Annotation

To begin, from the annotation panel, the editor can open the new annotation panel by selecting the  button.

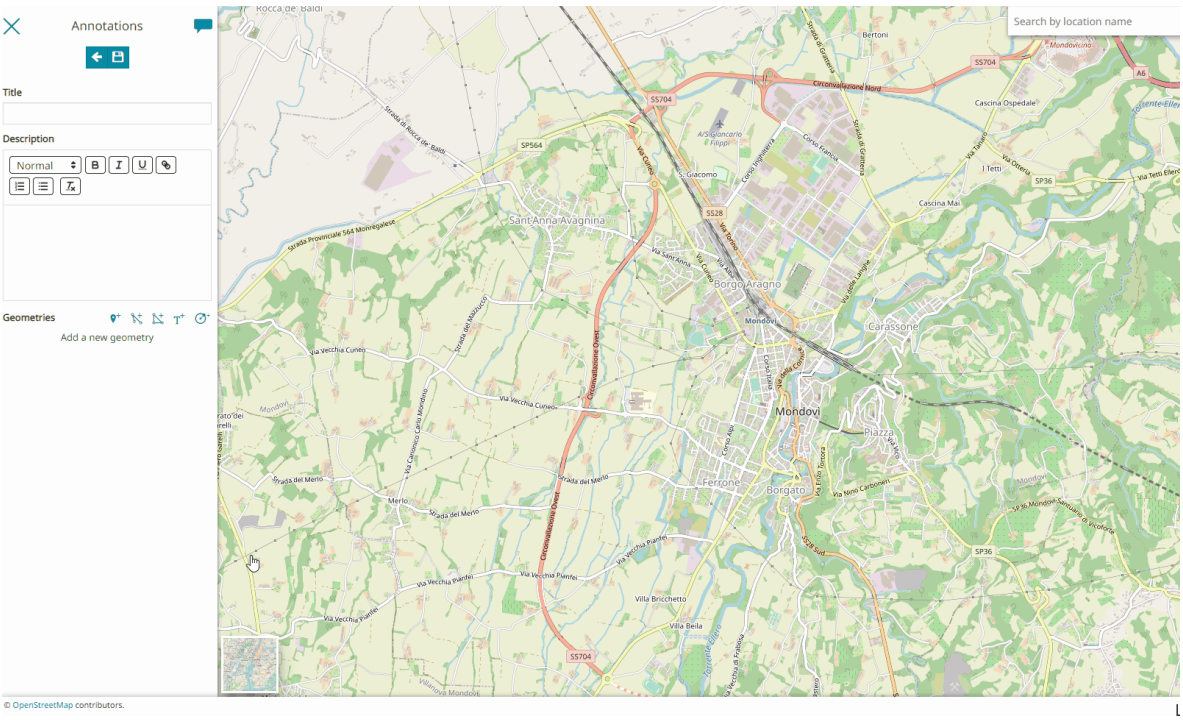


From here the editor can insert a **Title** (required), a **Description** (optional) and choose between five different types of **Geometries**:

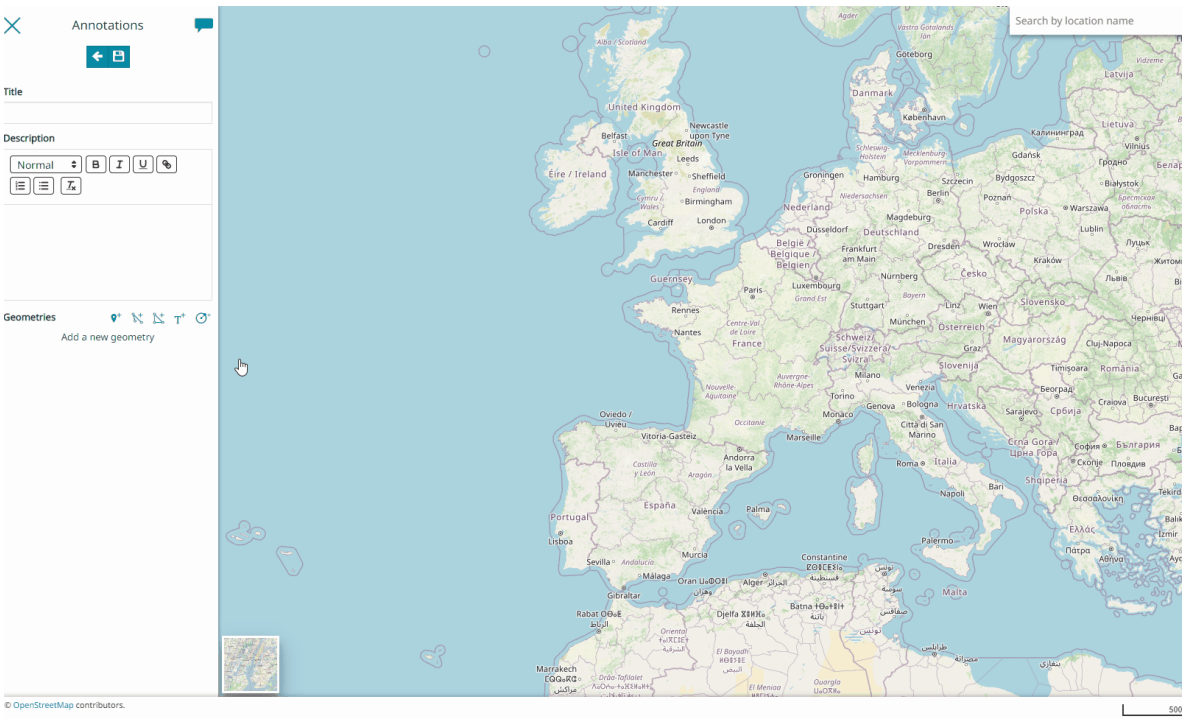
- **Marker**
- **Line**
- **Polygon**
- **Text**
- **Circle**

After selecting a geometry type, the editor can:

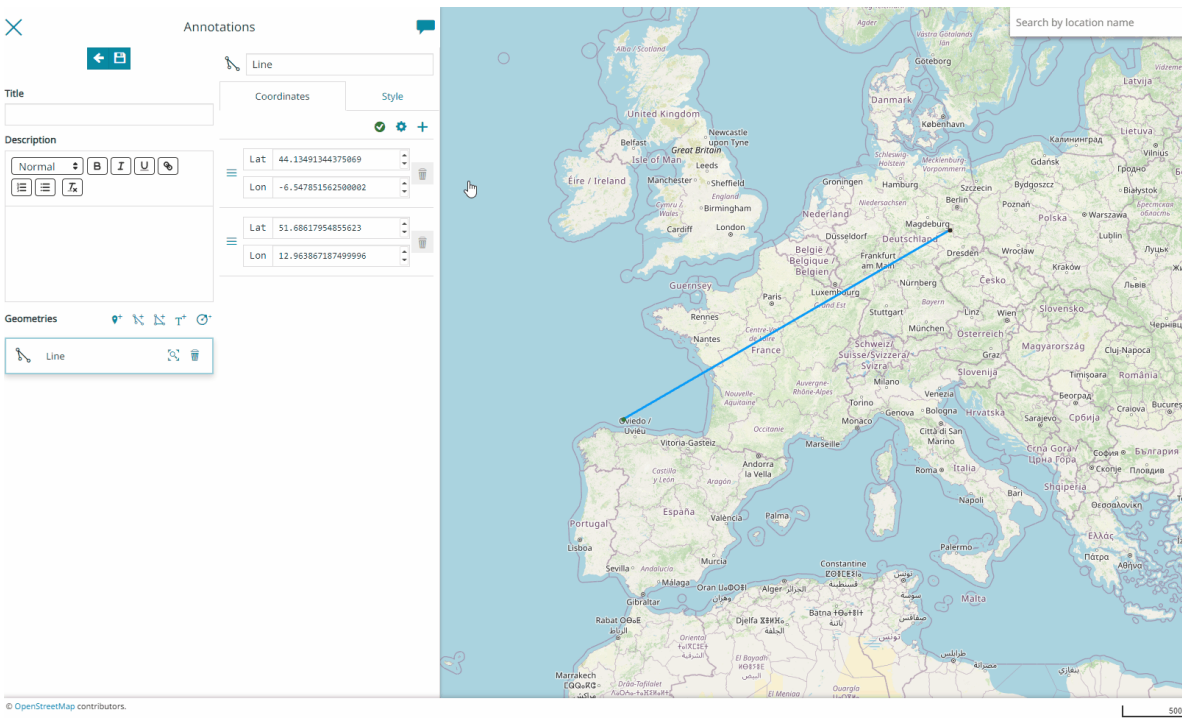
- Draw a *Geometry* on the map.



- Enter the vertices of the geometry or modify the existing ones through the **Coordinates editor** using `Decimal` or `Aeronautical` formats.

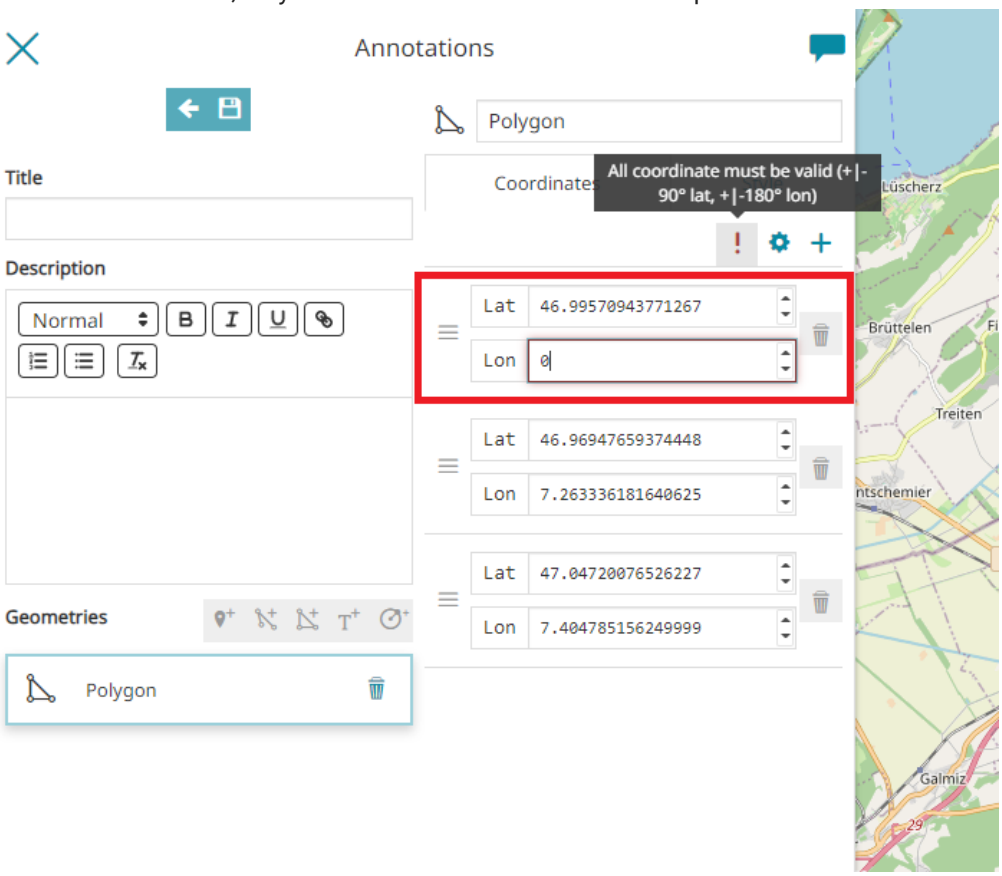


- For *Line* and *Polygon*, add new vertices using the + button and typing the `latitude` and `longitude` values.

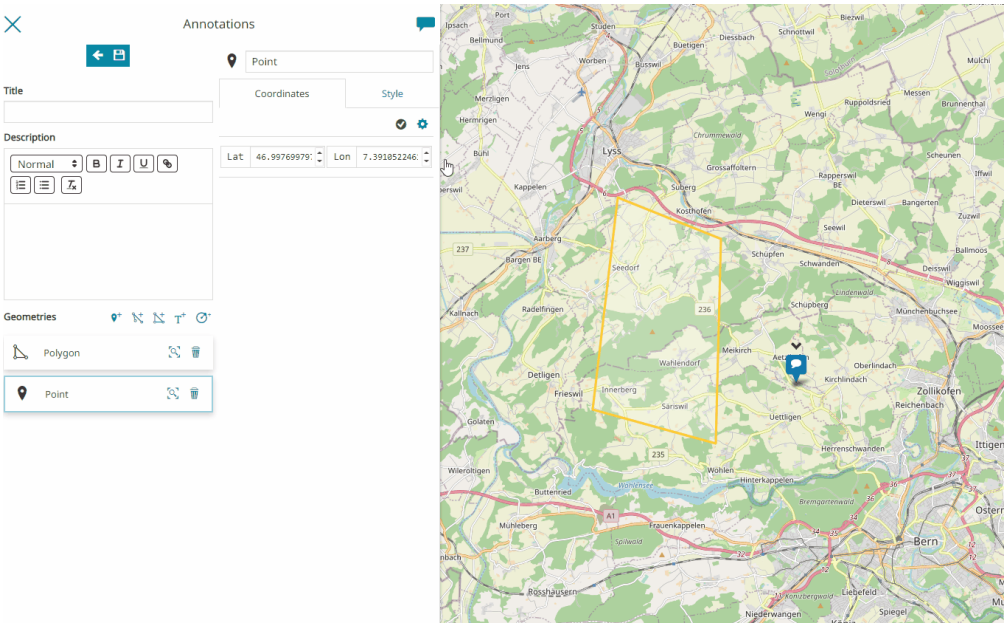


Note

If the vertices are invalid, they are notified with a red exclamation point.



In this case, it is not possible to add new geometry or save the annotation until a valid value is entered. It is still possible to interact with the geometries already present in the annotation, by zooming in on it or deleting it, as follows:

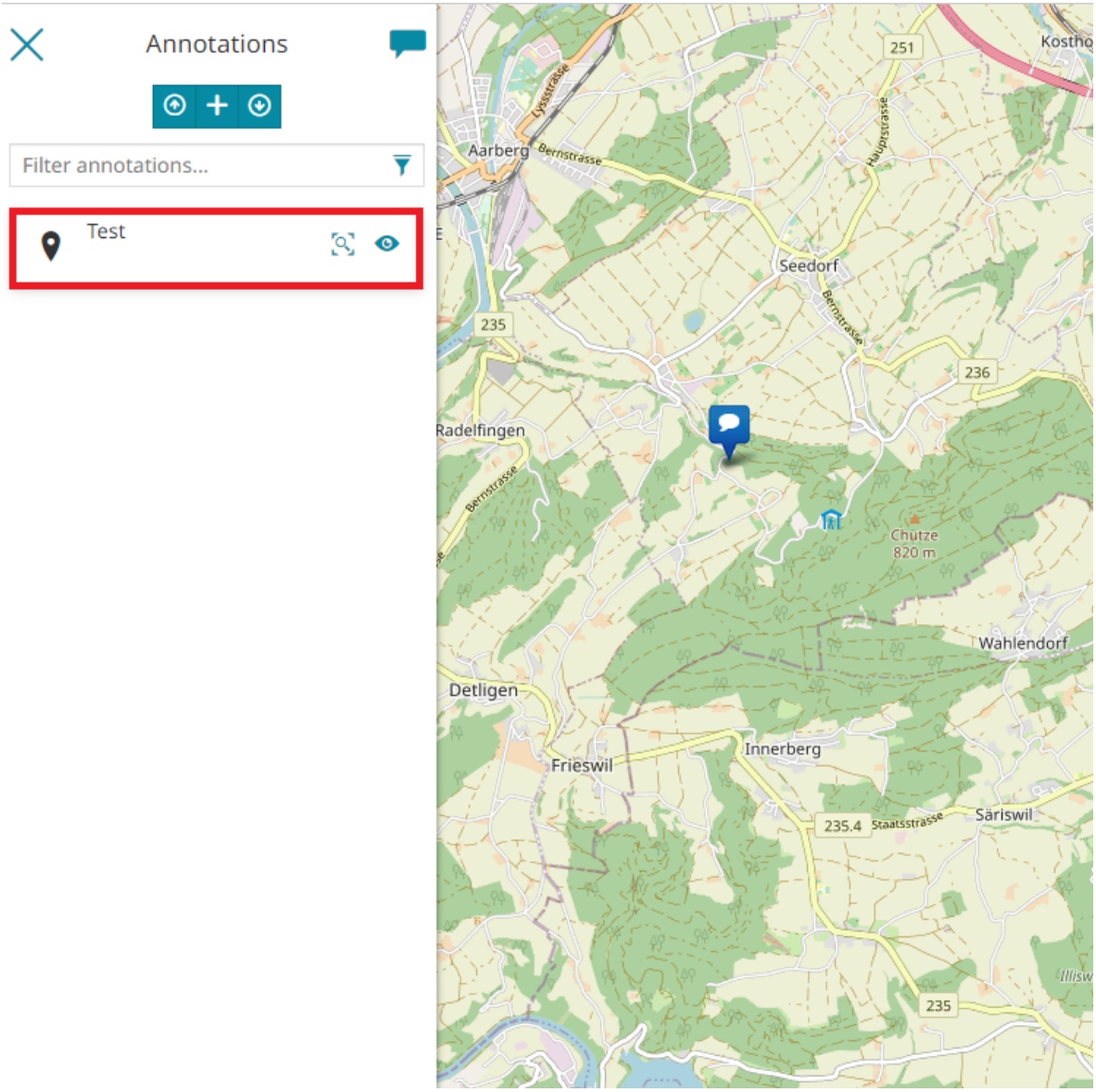


- Customize the **Style** of the annotation, as explained in the following paragraph.

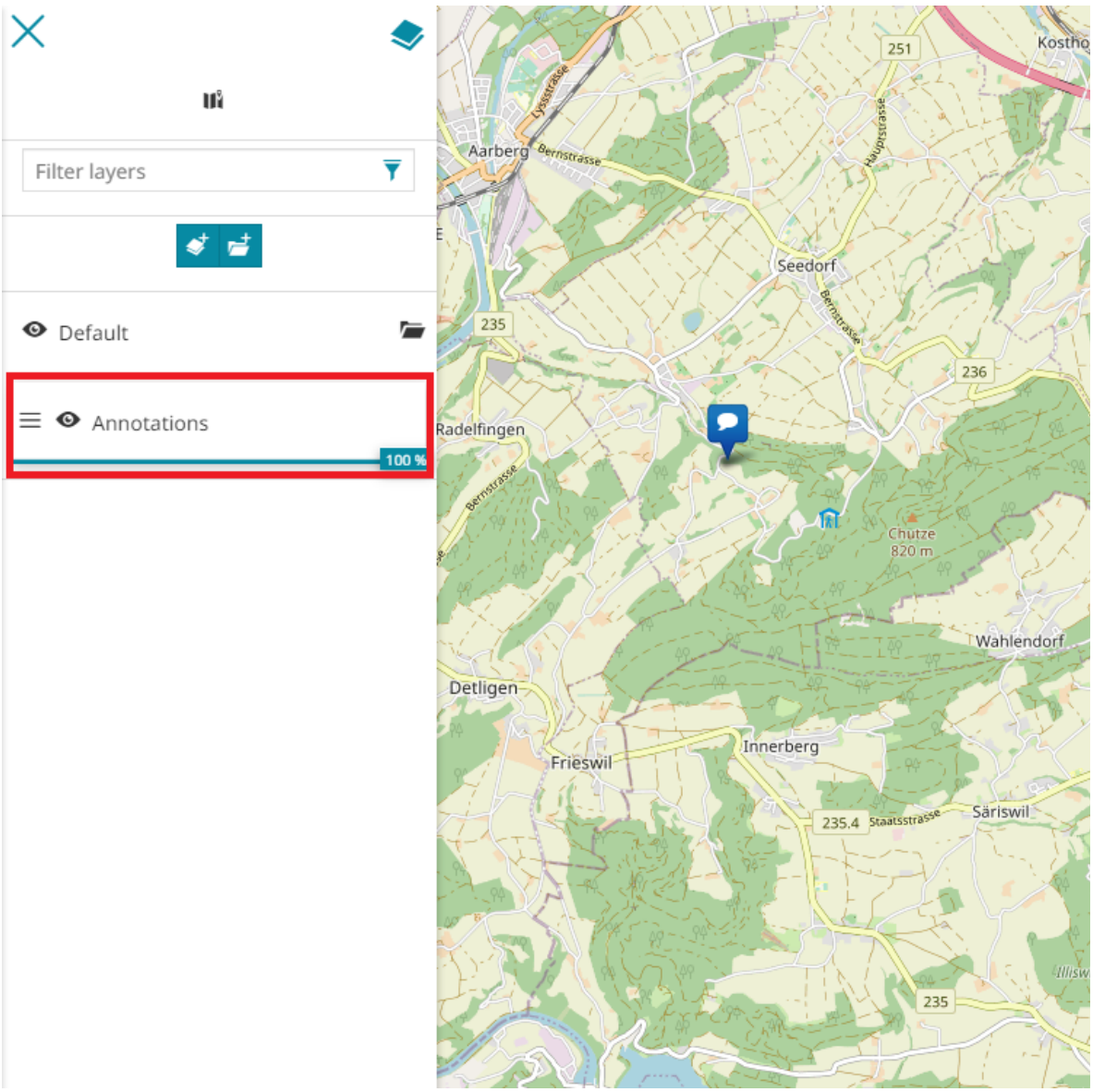
Once the geometry has been saved through the **Save** button, for each geometry created, the editor can perform the following operations:

- **Zoom** to the geometry annotation on map through the button
- **Delete** the geometry annotation through the button

Once all the *Geometries* have been created, the editor can save the annotation through the **Save** button that will be visible in the annotation list:

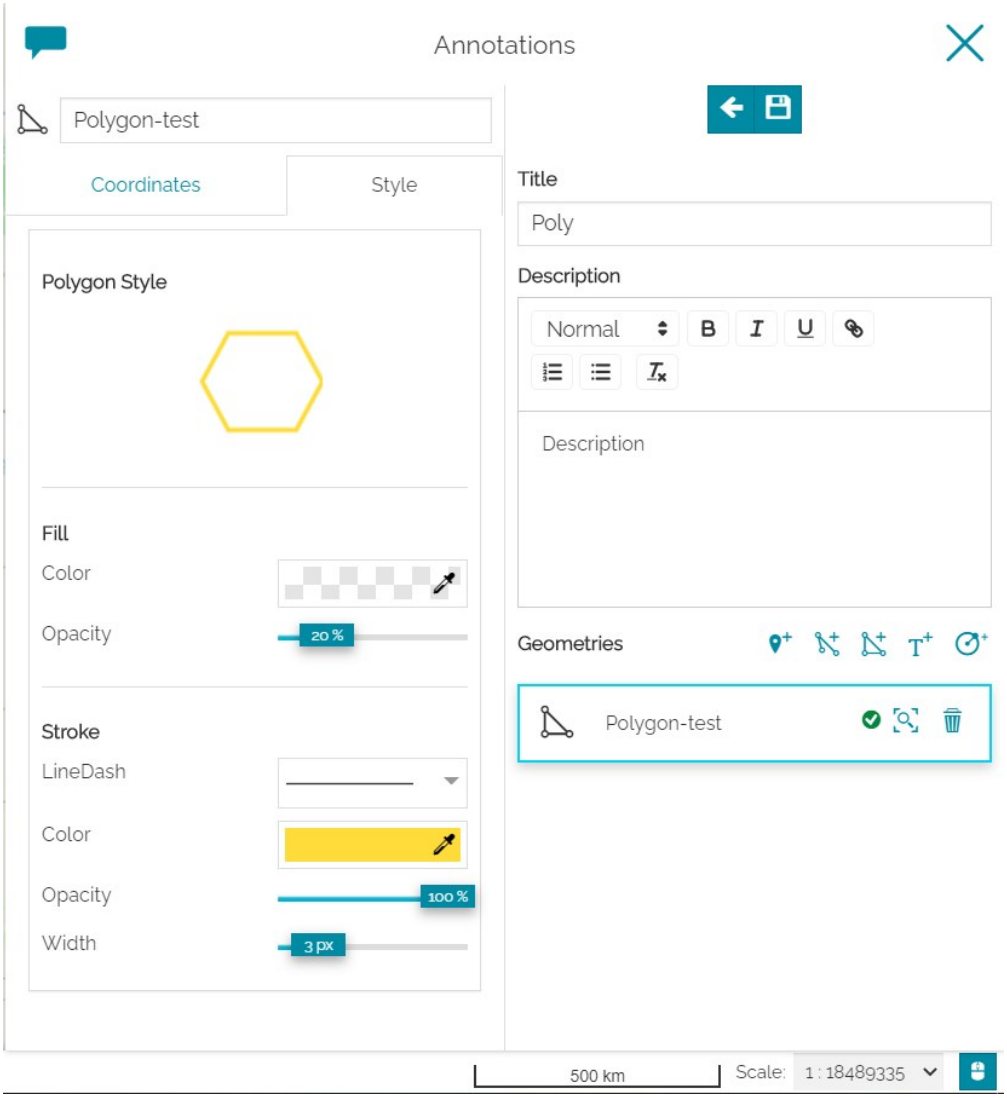


Then, if not present, a new **Annotations** layer will be created and added to the TOC



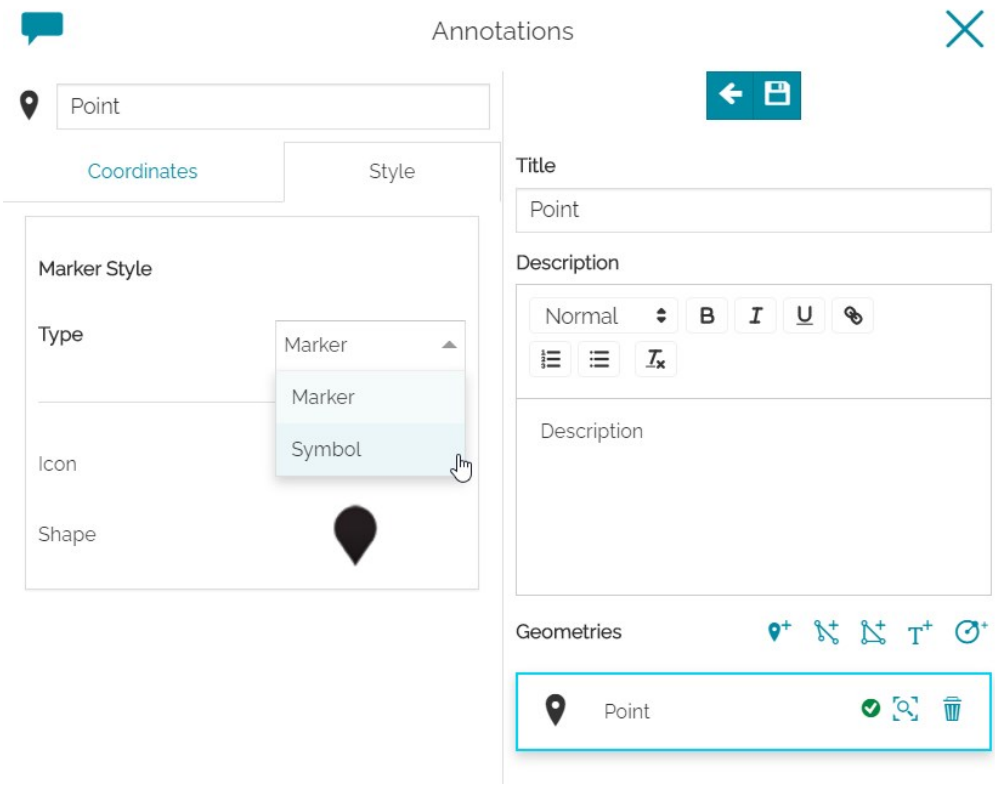
Styling Annotations

Based on which type of annotation was chosen, MapStore allows you to customize the annotation style through a powerful editor. It is accessible from the *Style* tab of the annotation viewer. During the style editing a preview placed on top of the styler form shows a preview of the edited style.

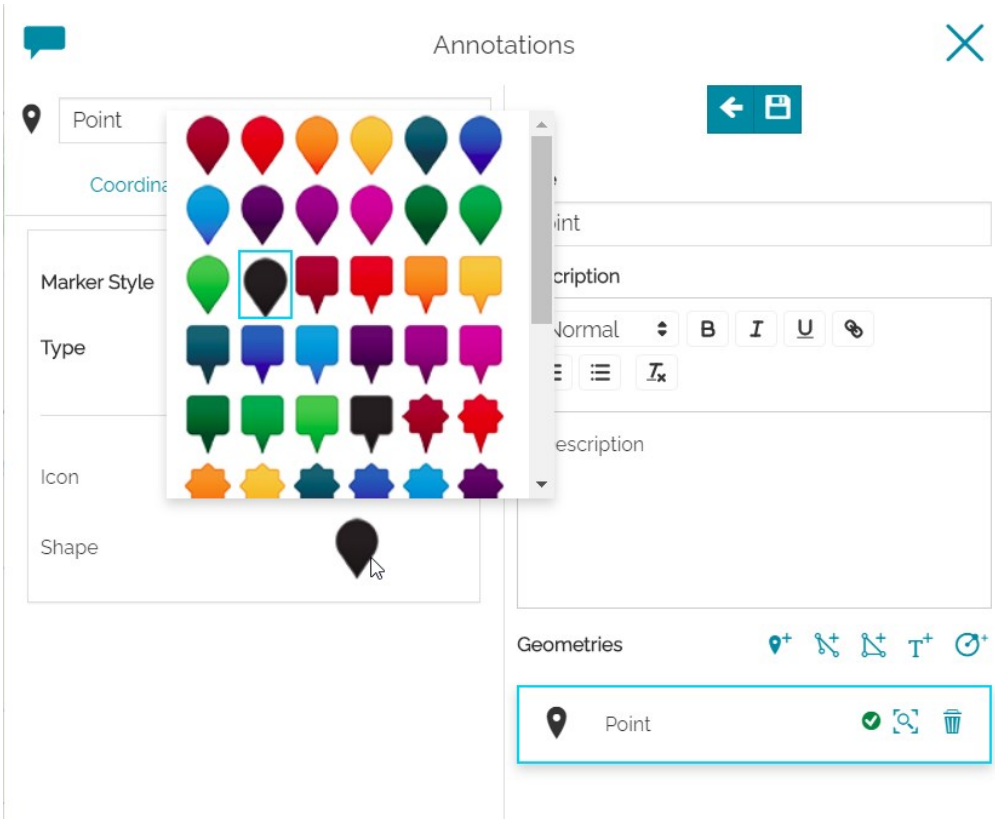


Marker

MapStore provides two types of *Marker* annotations, so you have to choose what type do you prefer using the *Type* combo box (*Marker* is the default):

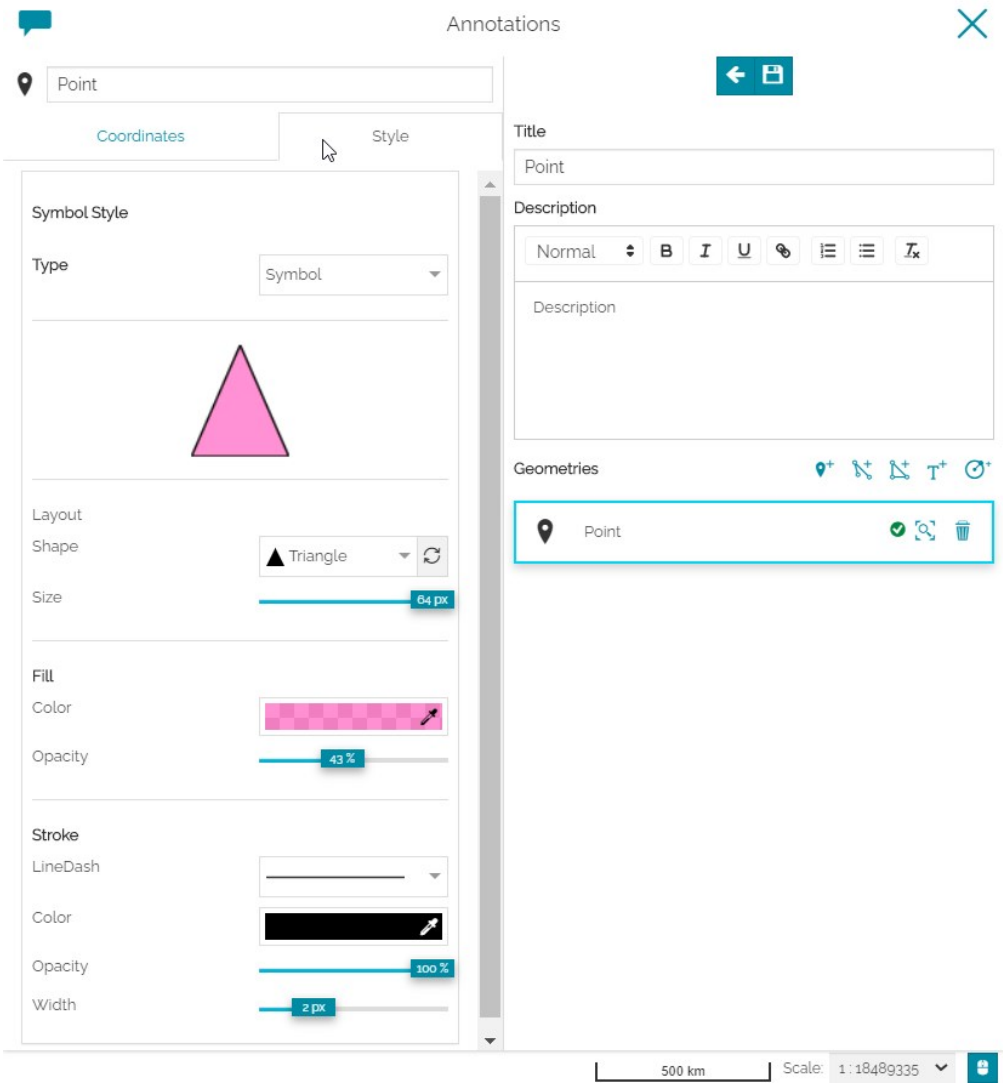


• *Marker* types can be customized through the following editor:



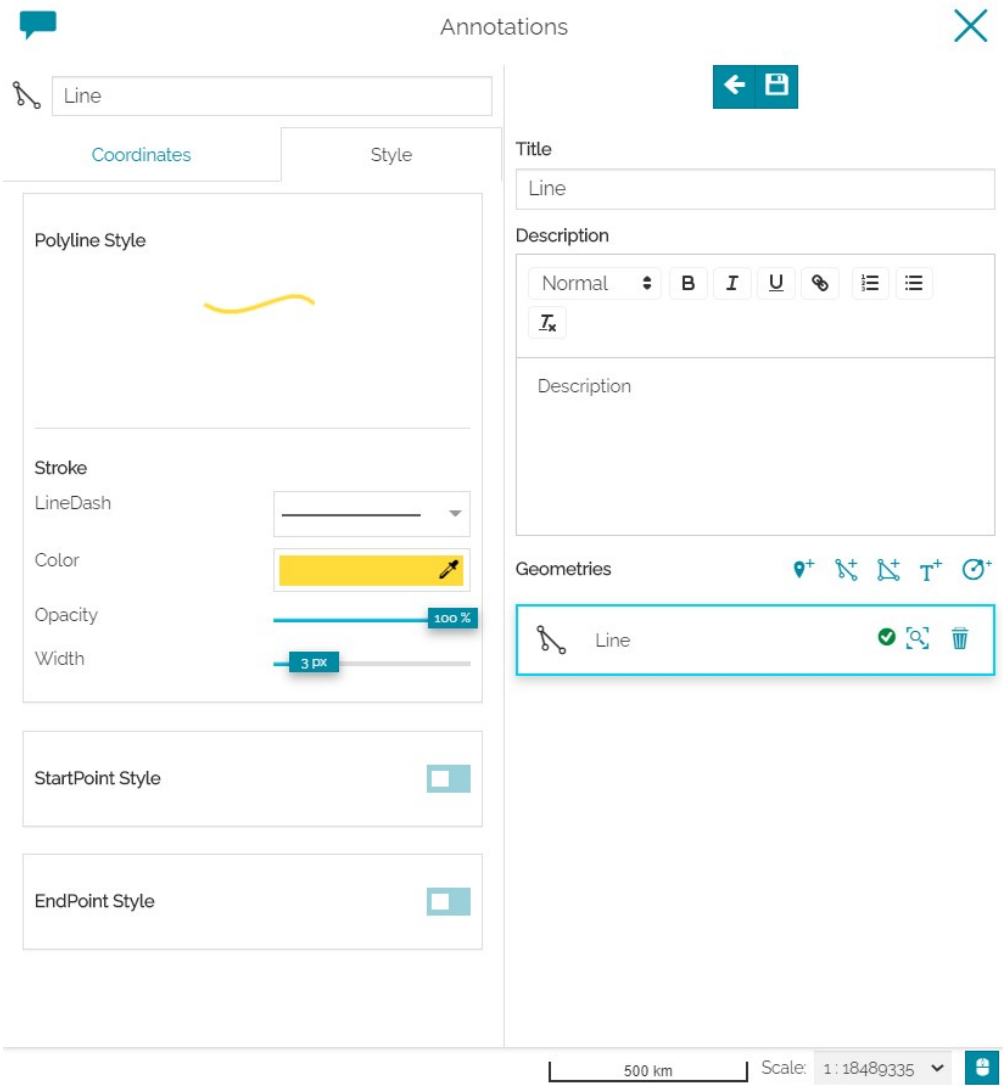
Choose the `Shape`, `Color` and `Icon` that best fit your needs.

- *Symbol* types can have different `Shape` and `Size`, a `Fill color` with a customizable `Opacity` level (%), a `Stroke` of different types (continuous, dashed, etc) and customizable `Color`, `Opacity` and `Width`. Only few symbols are provided by default in MapStore but a custom list of symbols can be configured.



Polyline

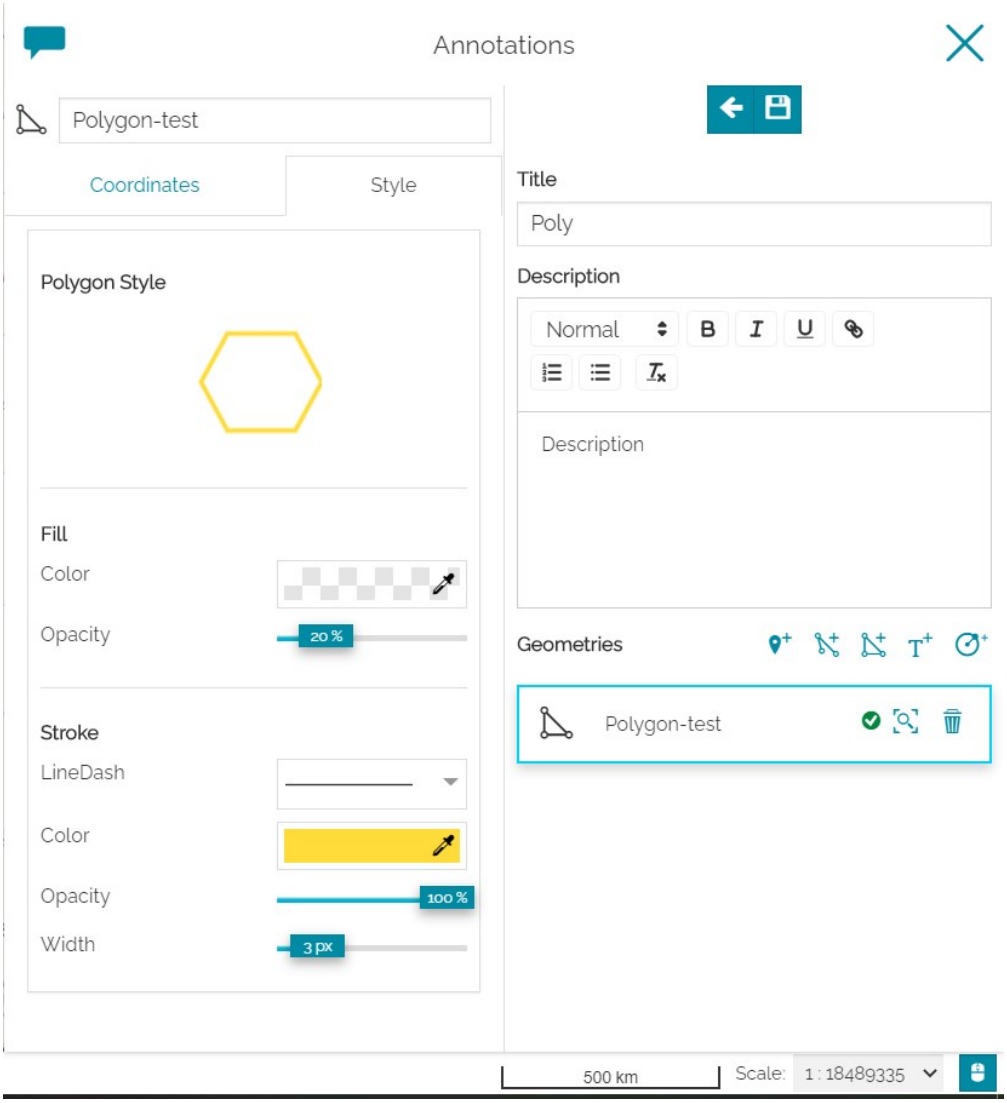
Polyline annotations can be styled using the following editor:



You can customize the `Stroke` in order to consider the `Line/Dash` type (continuous, dashed, dotted, etc), `Color`, `Opacity` and `Width`. You can also have styled *Start/End Points*: enable the StartPoint *Style/EndPoint Style* panel using the corresponding check box, the editor will be the same used for *Marker/Symbol annotations*.

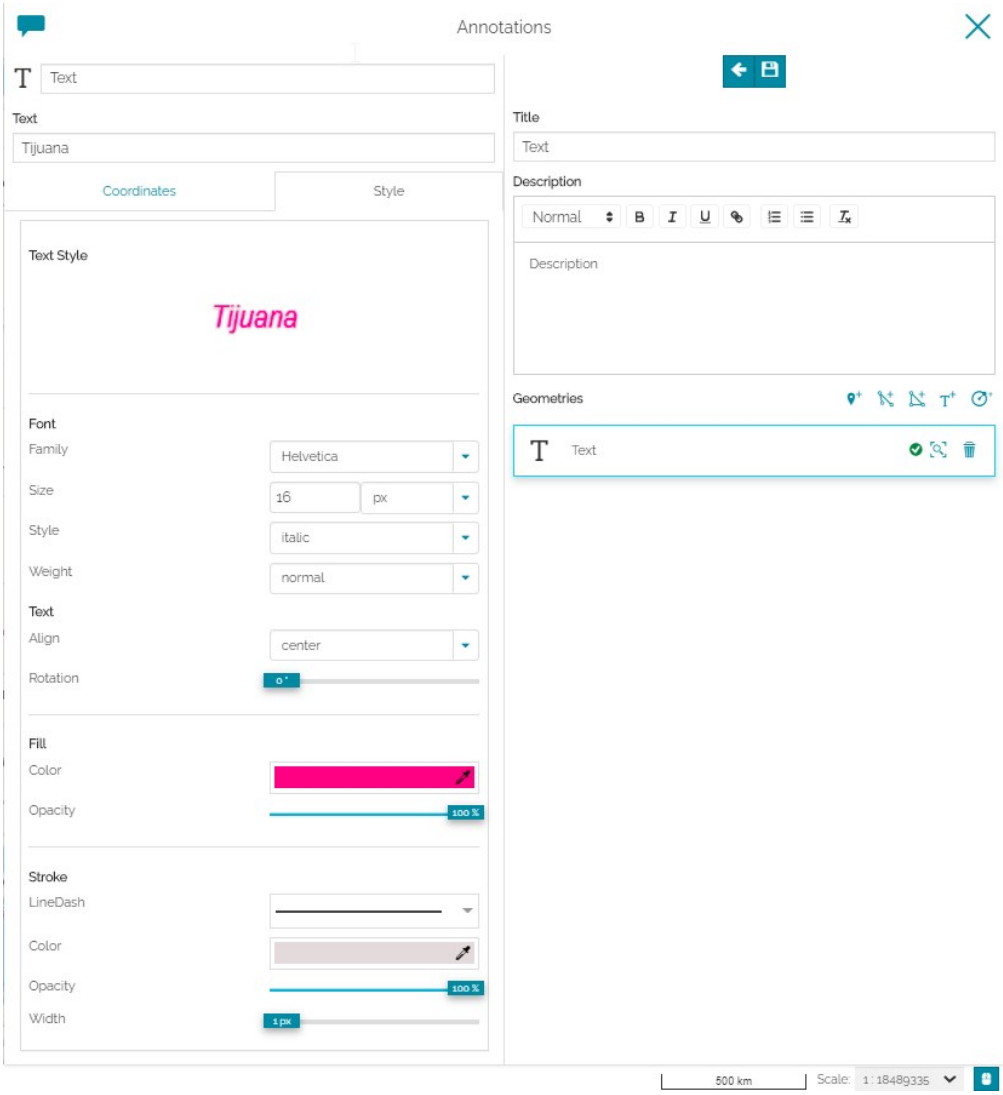
Polygon

With polygonal annotations changing the style means choose the `Shape` and the size the `Size` of the polygon, its `Fill color` (with custom `Opacity`), the type of the `Stroke` (continuous, dashed, dotted, etc), its `Color`, `Opacity` and `Width`. See the example below to better understand these options.



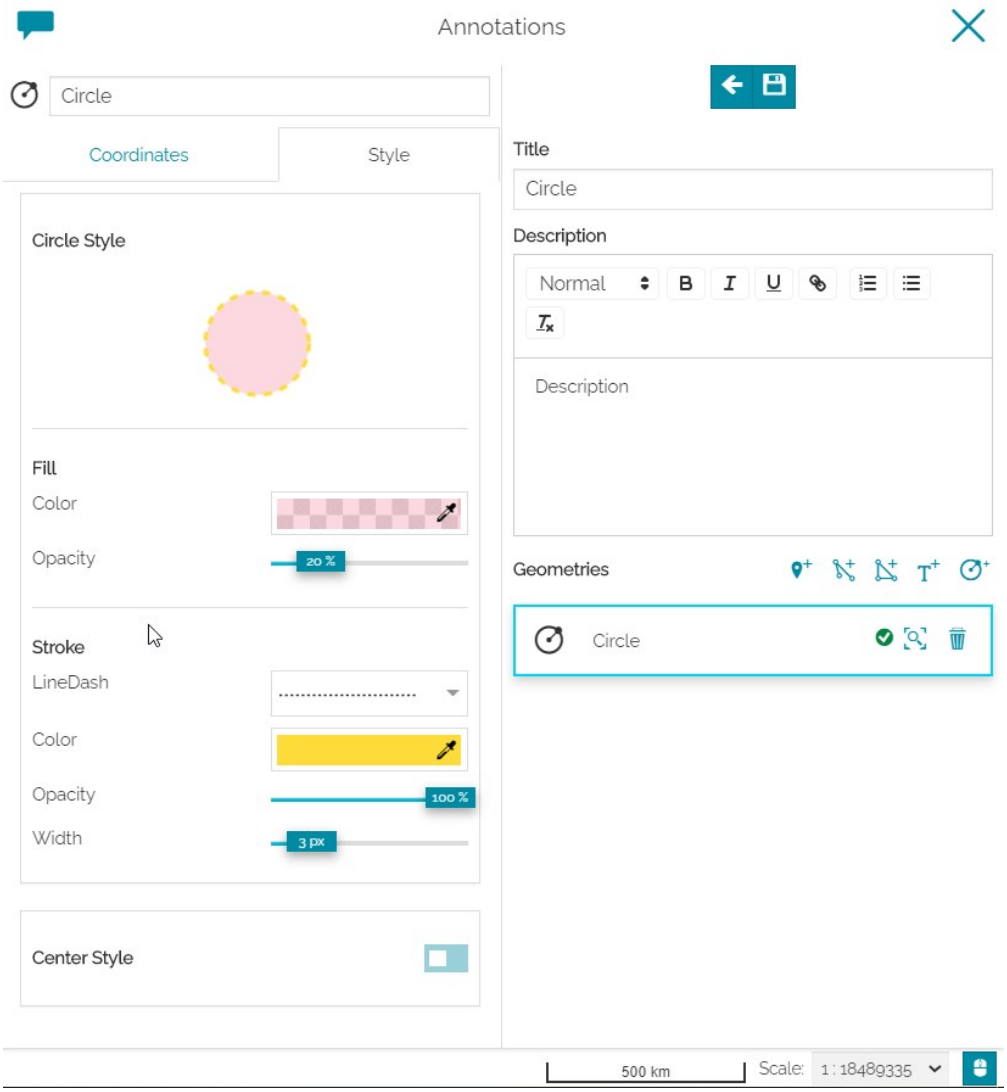
Text

Text annotations are a bit different from the geometric ones. They display a formatted text on a given point of the map. The style editor allows you to customize the text `Font` (`Family`, `Size`, `Style`, `Weight`), the `Alignment` (`left`, `center` or `right`) and `Rotation`. You can also choose the text `Fill color` and its `Opacity`, the `Stroke` type, its `Color`, `Opacity` and `Width`. Take a look at the following example.



Circle

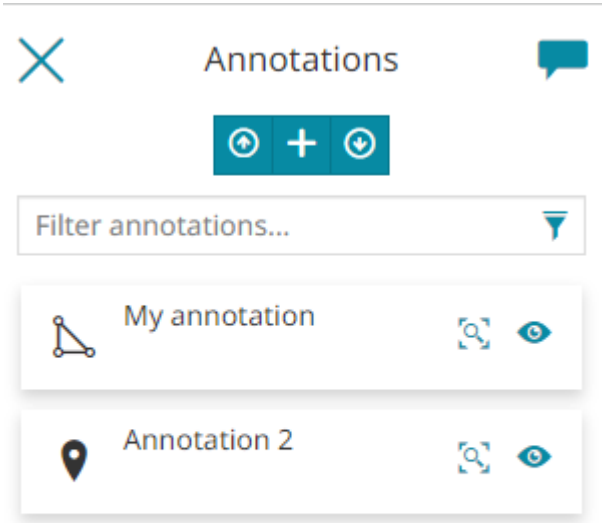
Circle annotations can have custom `Fill color` (with custom `Opacity`), `Stroke` type (continuous, dashed, dotted, etc) with custom `Color`, `Opacity` and `Width`. The *Center* can be also customized through the same editor described for *Marker* annotations. See the example below.



Click on  to apply the style.

Managing Annotations

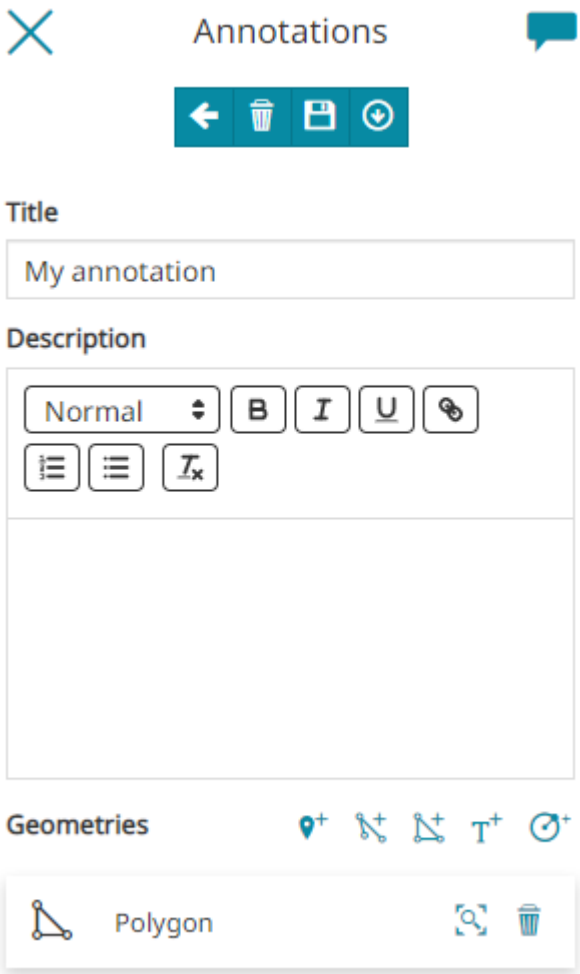
Once annotations are added to the TOC, the editor can **Manage** them by clicking to  button from the TOC toolbar and the *Main Annotations panel* will be open.



From it, the editor is allowed to:

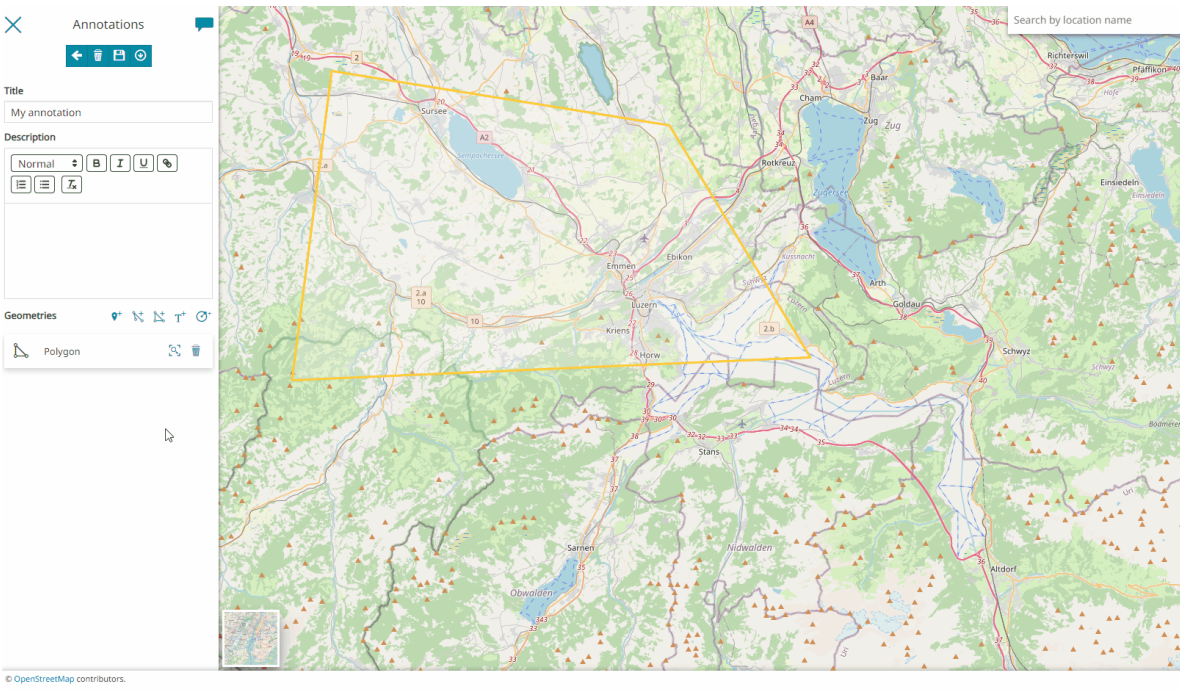
- **Download** a file with all the existing annotations by clicking on button
- **Upload** annotations from a valid `json` file by clicking on button
- **Zoom** an annotation on map by clicking on button
- **Show/Hide** an annotation on the map by clicking on button

From the *Main Annotations Panel*, by selecting an annotation from the list, the editor is returned to the *Annotation Viewer* where the annotation can be edited.



In particular, the editor can:

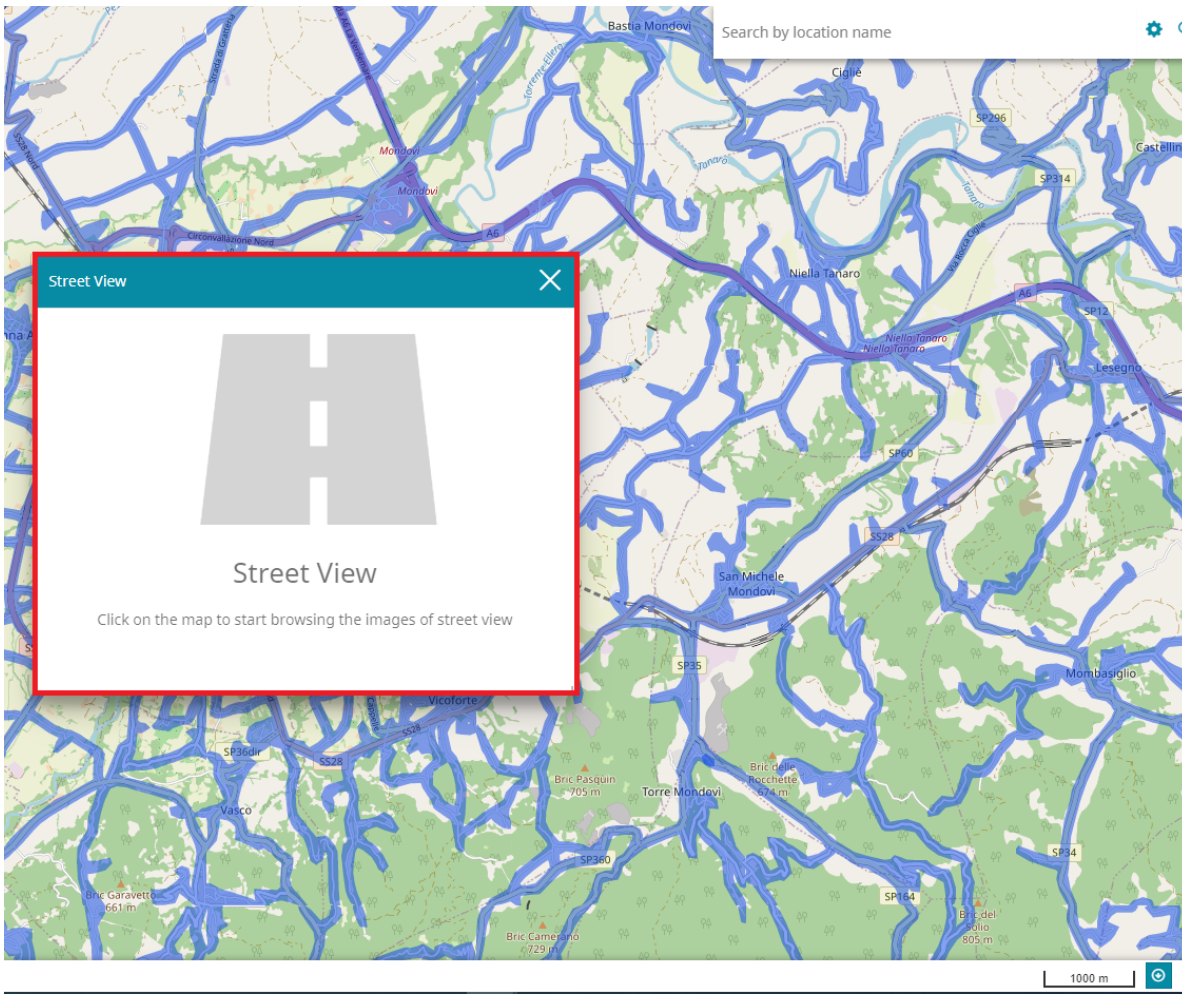
- Change the *Coordinates* and the *Style* by clicking a geometry from list of geometries.



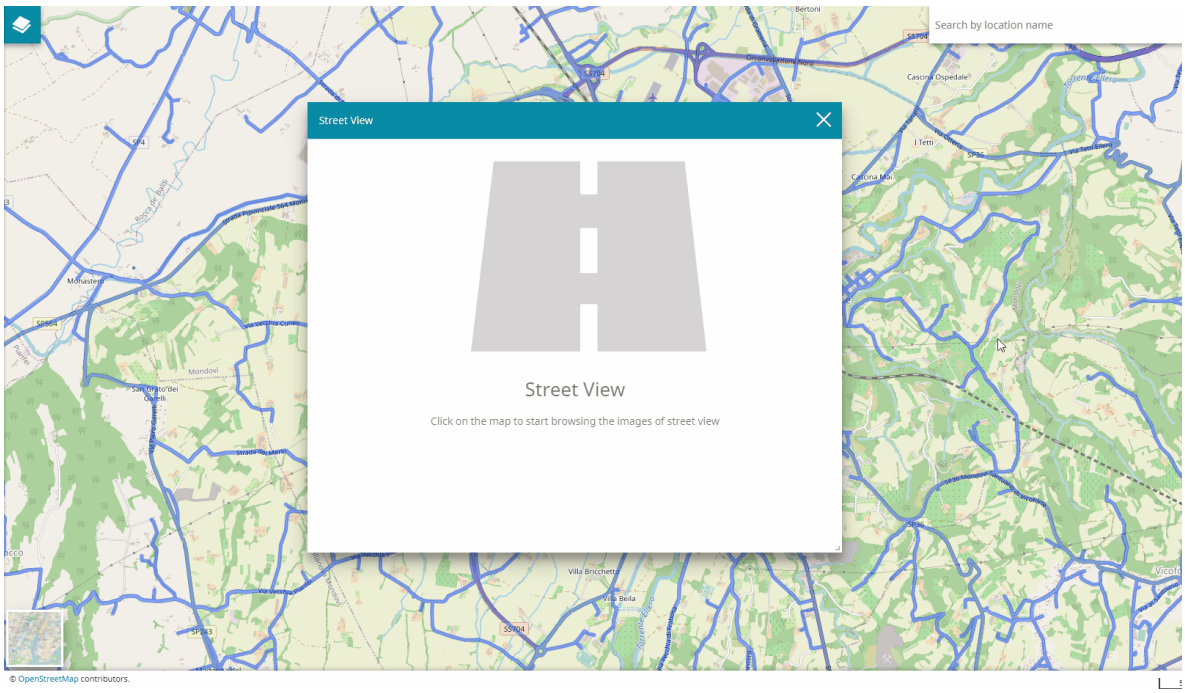
- **Download** the annotation in `json` format and reused in other maps by clicking on  button

Street View

The **Street View** tool allows the user to browse Google Street View contents in MapStore. Through the  button available in the Side Toolbar, the tool can be activated so that it is possible to navigate the map with Google Street View.

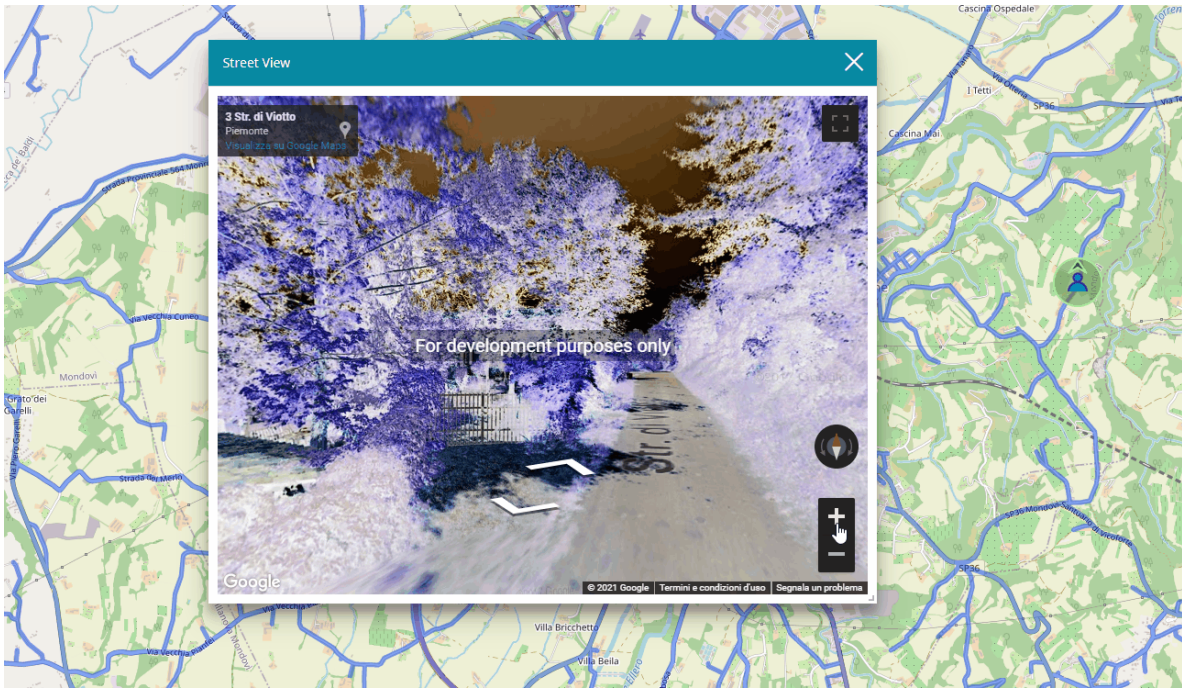


When the tool is activated, a window opens and the streets highlighted on the map so that the user can select one of them with a simple click of the mouse.

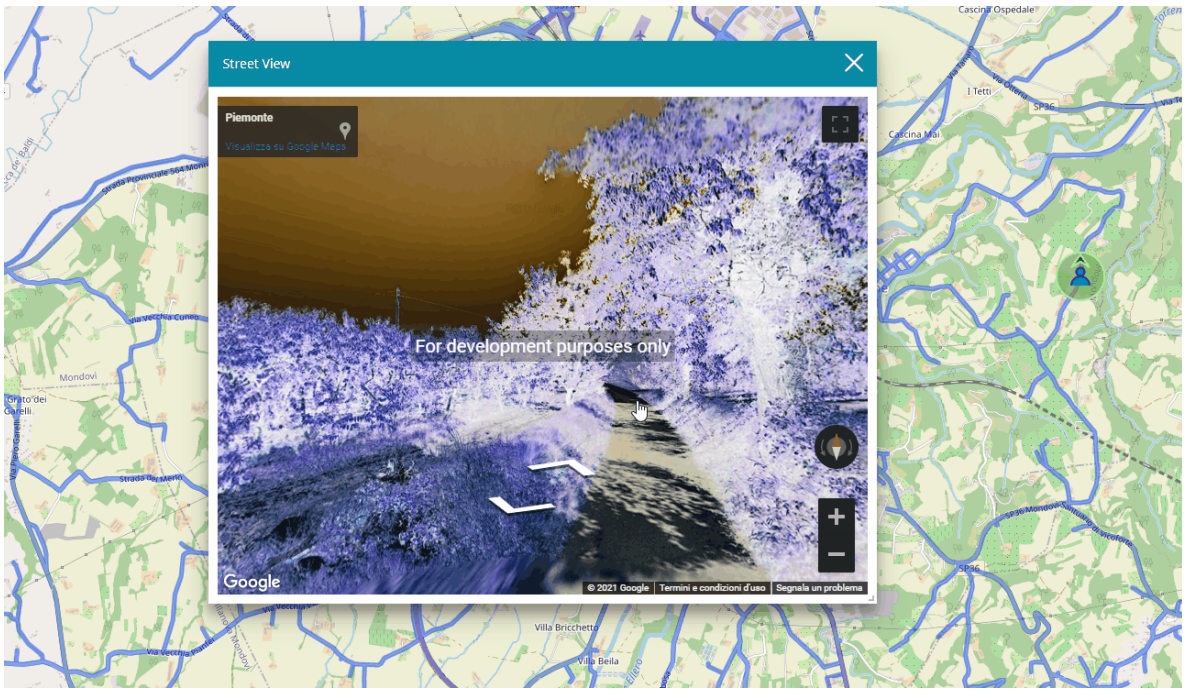


By clicking on a street in the map, the tool window displays the Street View and the user can navigate it as usual.

- **Zoom in/out**  on the street



- Use the **Pan Interaction** to navigate all-around the street

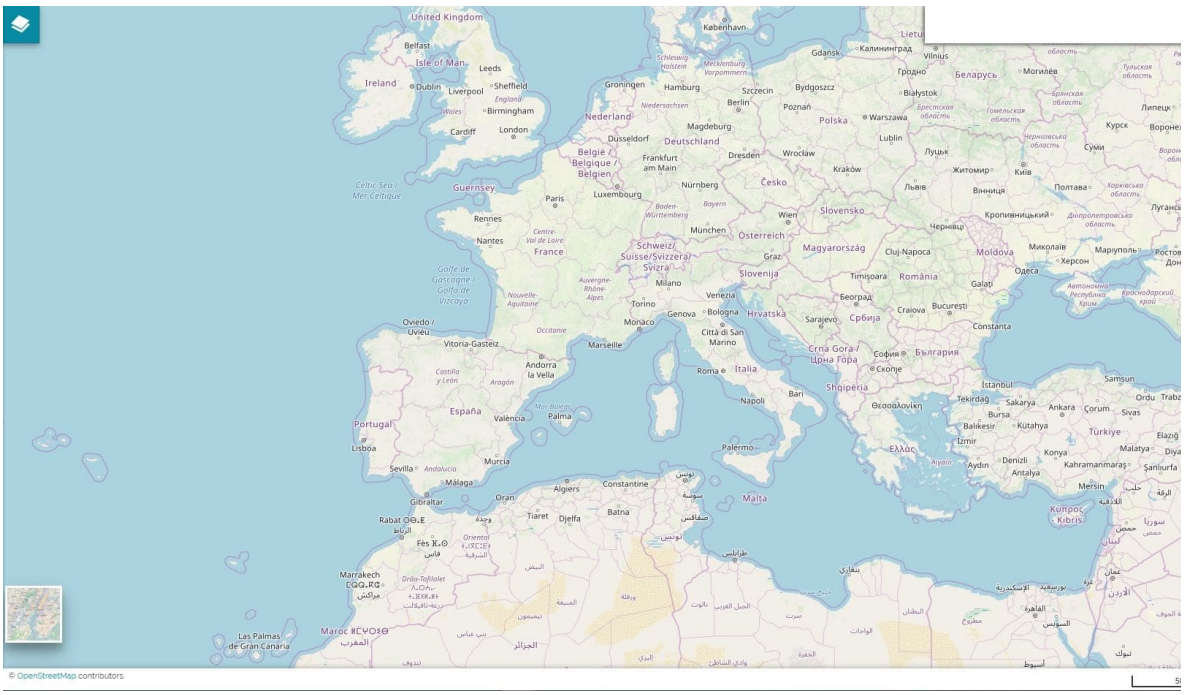


- Enable/disable the **Full Screen** 

> **Note**
>
> By default, the **Street View** plugin is ready to be configured for application contexts, and is not available in the default plugin configuration due to licensing reasons.

Sidebar

The *Sidebar* is a navigation panel containing various elements that help the user to explore the map. In particular, it is possible zooming, changing the extent, navigating in 3D mode and querying objects on the map. Moreover, the following icon  is used to expand/collapse the sidebar.



Geolocation tool

Through the *Show my position*  the user can center the map on his position. Therefore the button turns green.

The position is still active even when the user interacts with the map; with a single click on the button it is possible re-center the map on his position. To disable the position the button needs to be duble clicked.

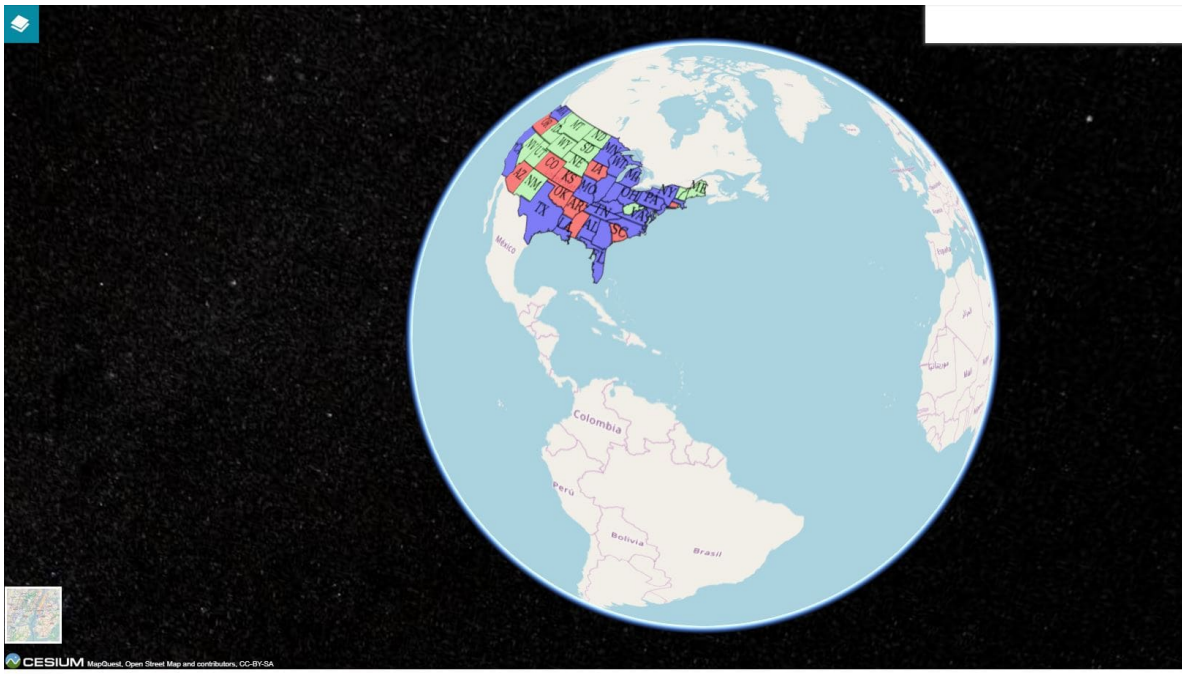
Zooming tools

MapStore provides several tools allows the user to:

- **Increase** the map zoom by using the zoom in icon
- **Decrease** the map zoom by using the zoom out icon
- **Switch to full screen** view
- **Go back** to the previous map extent in the map navigation history
- **Go forward** to the next map extent in the map navigation history
- **Zoom to the maximum extent** the map

3D Navigation

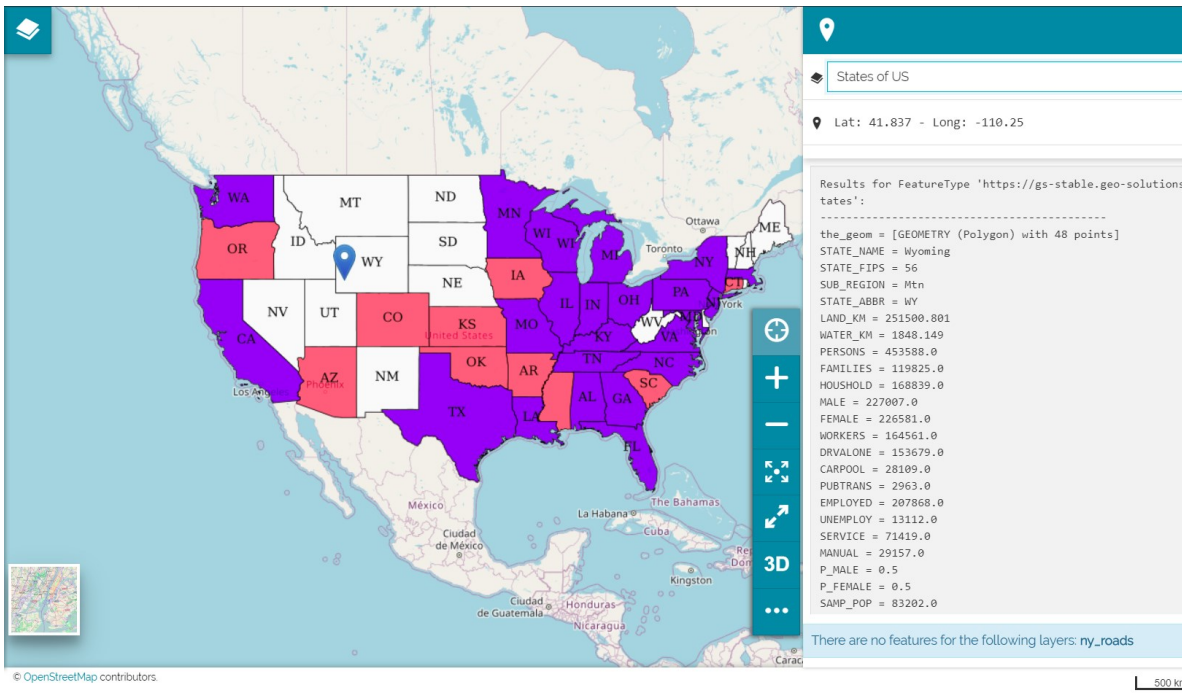
The *3D navigation* in MapStore is based on CesiumJS. If the 3D button 3D in the sidebar is clicked, the map switch in 3D mode so map contents are displayed on a 3D globe and it is possible to orbit around it through the compass place in the upper right corner of the map.



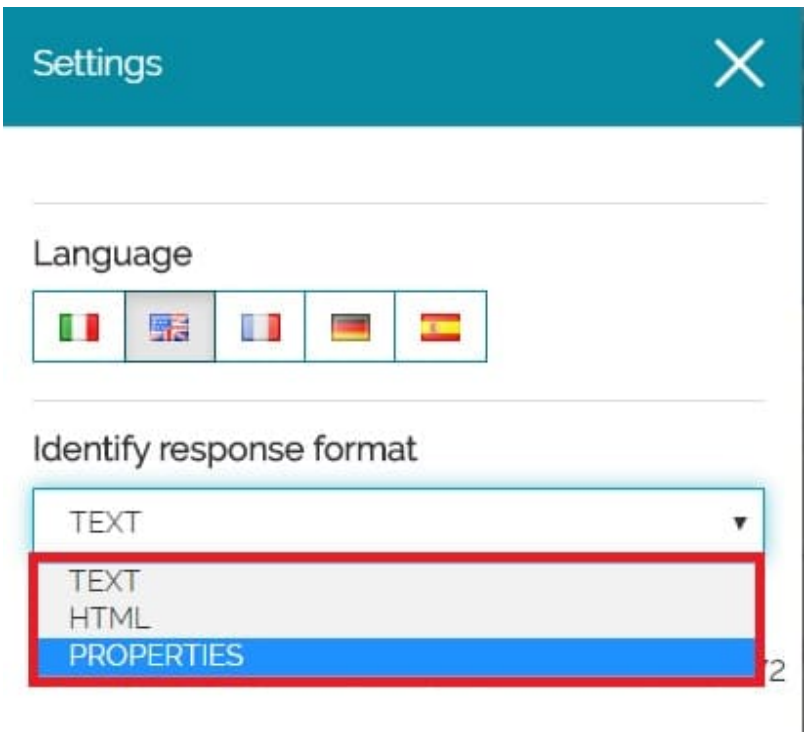
> **Note**
>
> The *3D mode* in MapStore support also the rendering **3D Tiles** layers once they are added through the *Catalog tool* as explained here.

Identify Tool

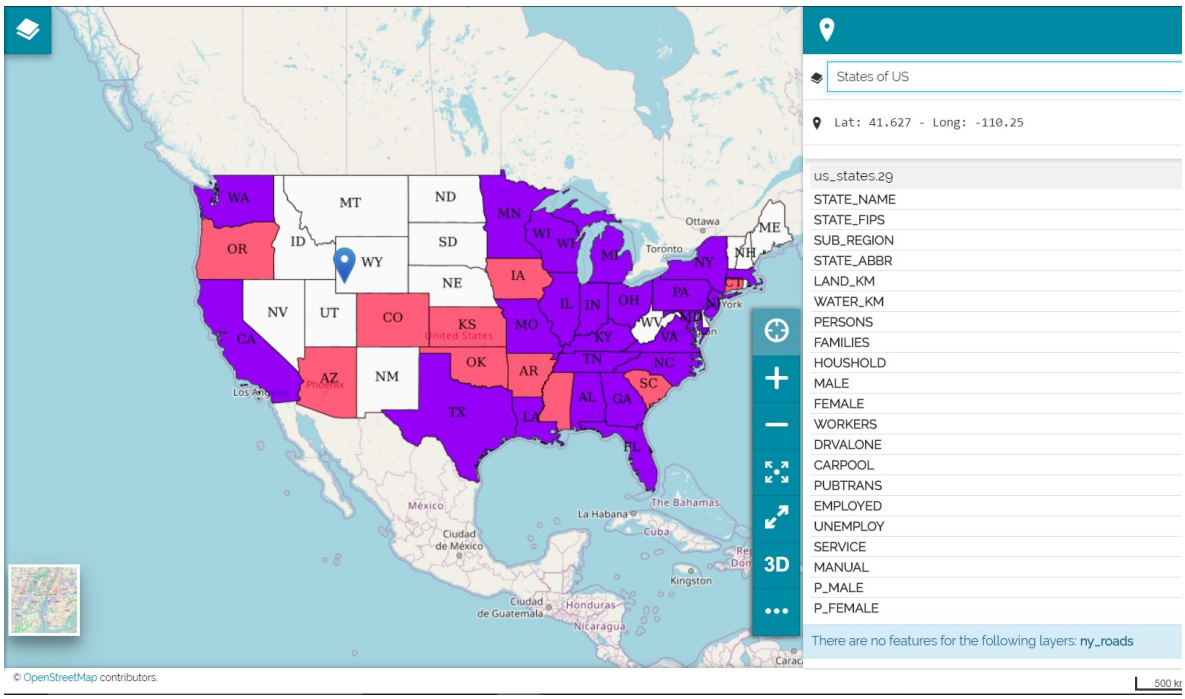
The *Identify* tool allows to retrieve information about layers on the map. The tool is active by default (the button is green). Therefore if the user click on a layer in the map, the identify panel opens containing the layers information corresponding to the clicked point in the map (also the coordinates of the clicked point are reported in the identify panel).



The layers information are reported in plain text by default. It is possible to change the format by selecting the ⚙ button in Side Toolbar where the user can select, through the *Identify response format* menu, three different formats like: **TEXT**, **HTML** and **PROPERTIES**.



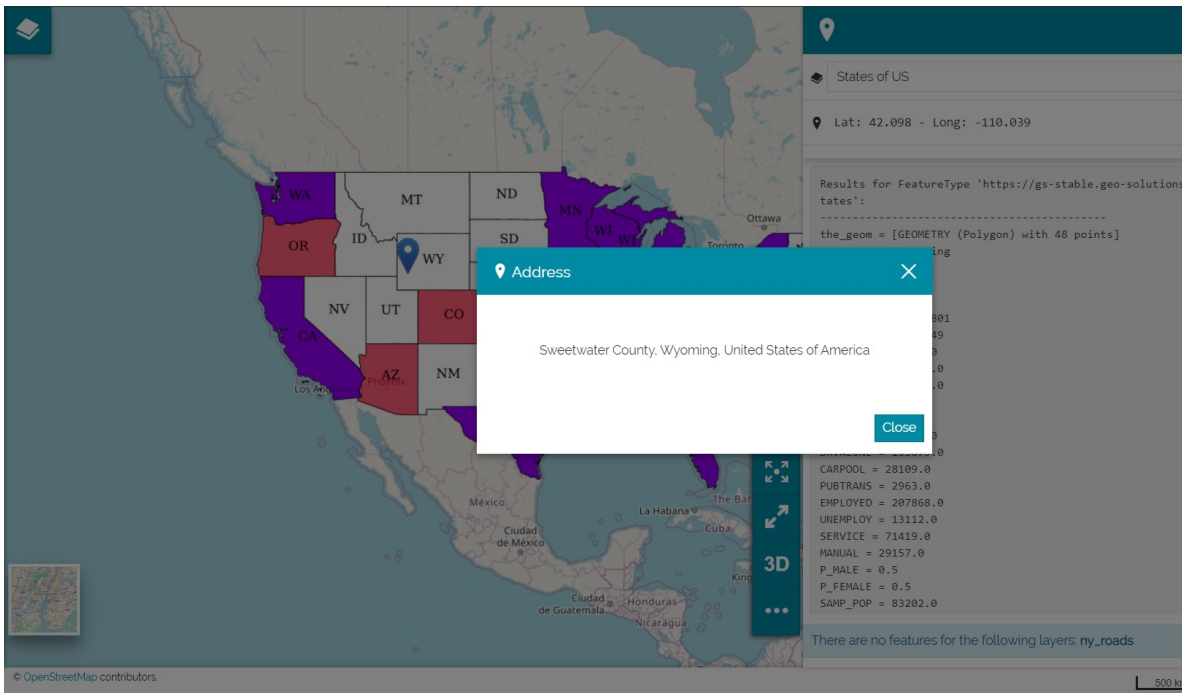
The information will be returned in the format chosen by the user. For exaple with *PROPERTIES* format as follows:



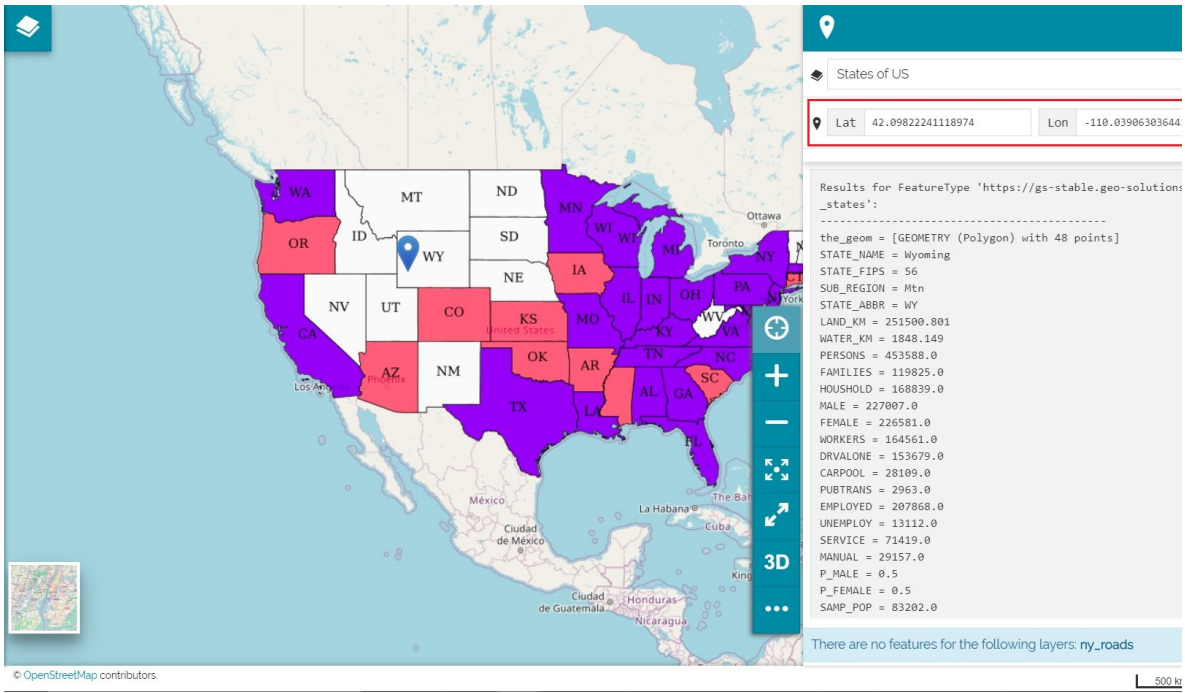
> **Warning**
>
> This global settings could be overwritten by a layer-specific configuration (see Feature Info Form).

In addition to the layers information, the following are provided by the *Identify Tool*:

- The **point address** through the *More Info* button

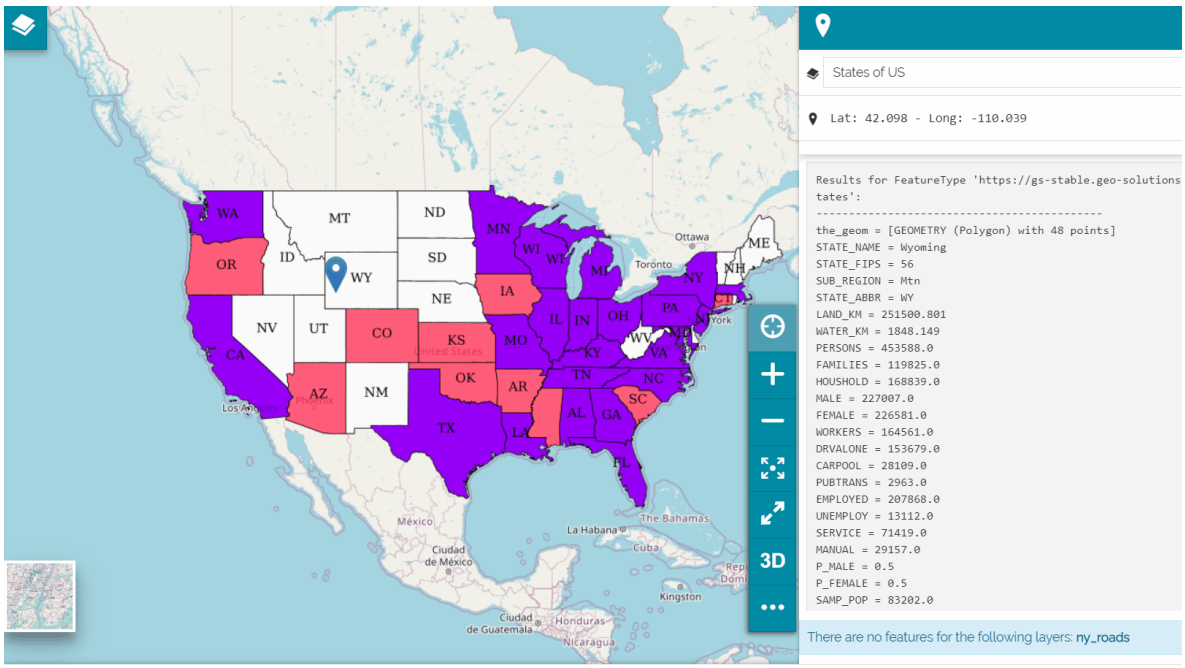


- The **coordinates** of the point

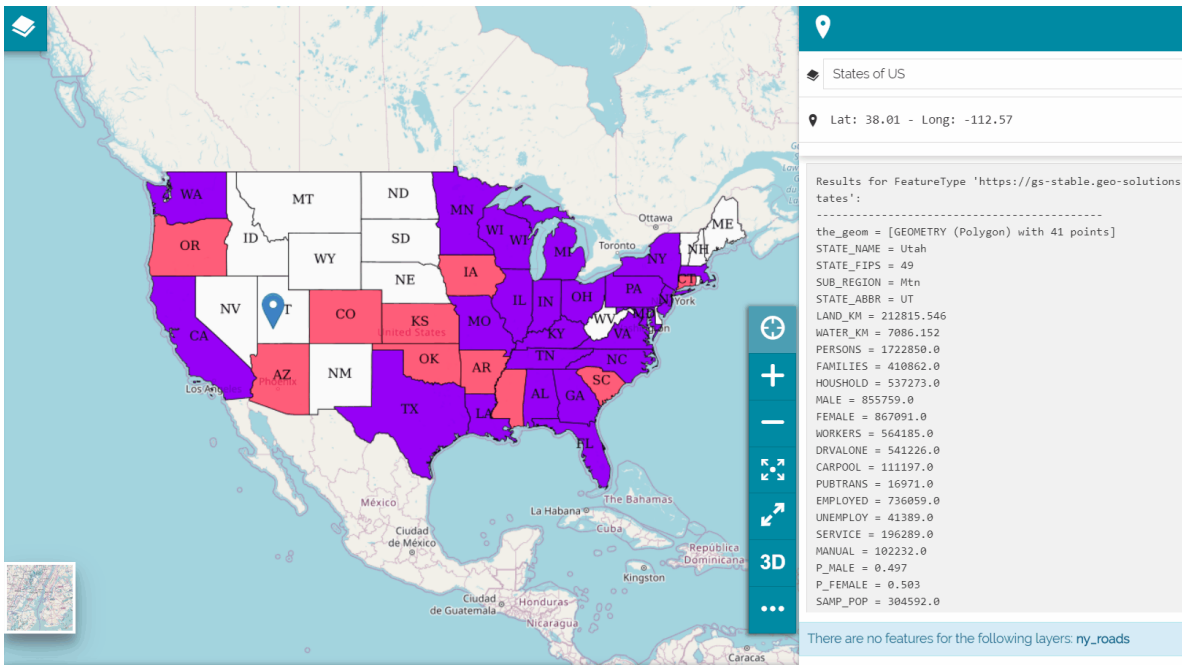


> **Note**
>
> The point coordinates are visualized in **decimal** or **areonautical** format. It is possible to change the format by the *setting* button 

- The **Highlight Features** button  allows to highlights on the map the layers features corresponding to the retrieved information in the clicked point.



- The **Edit** button  allows the user to open the Attribute Table in edit mode showing only layers records corresponding to the clicked point on the map.

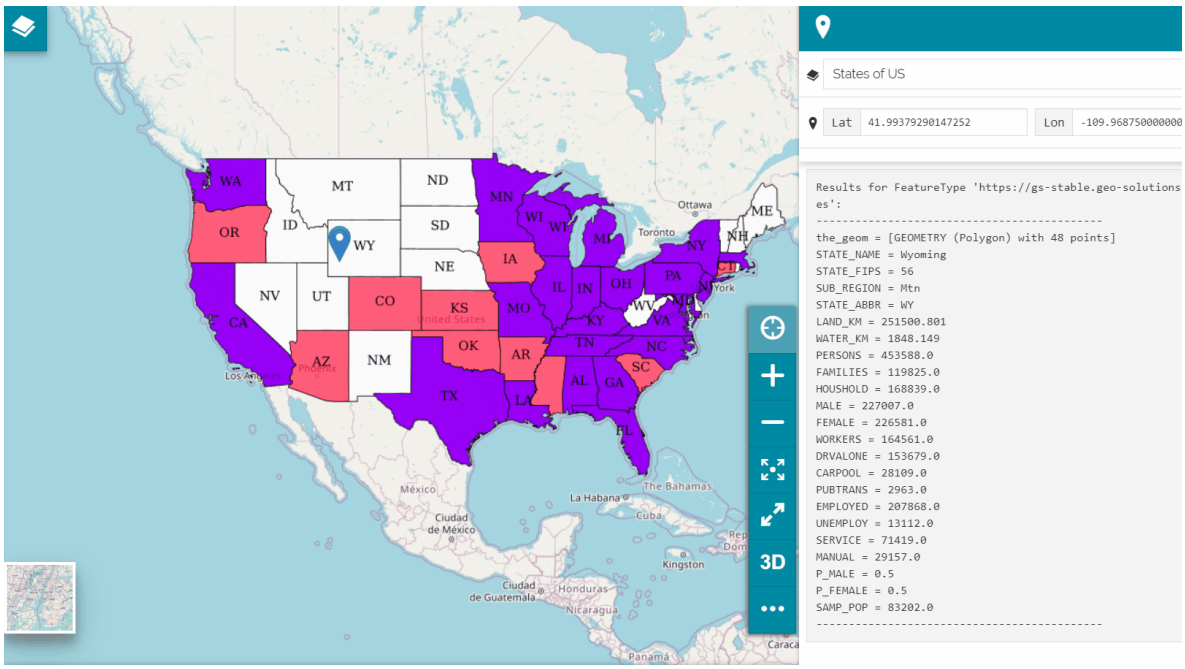


Using the Coordinates Editor

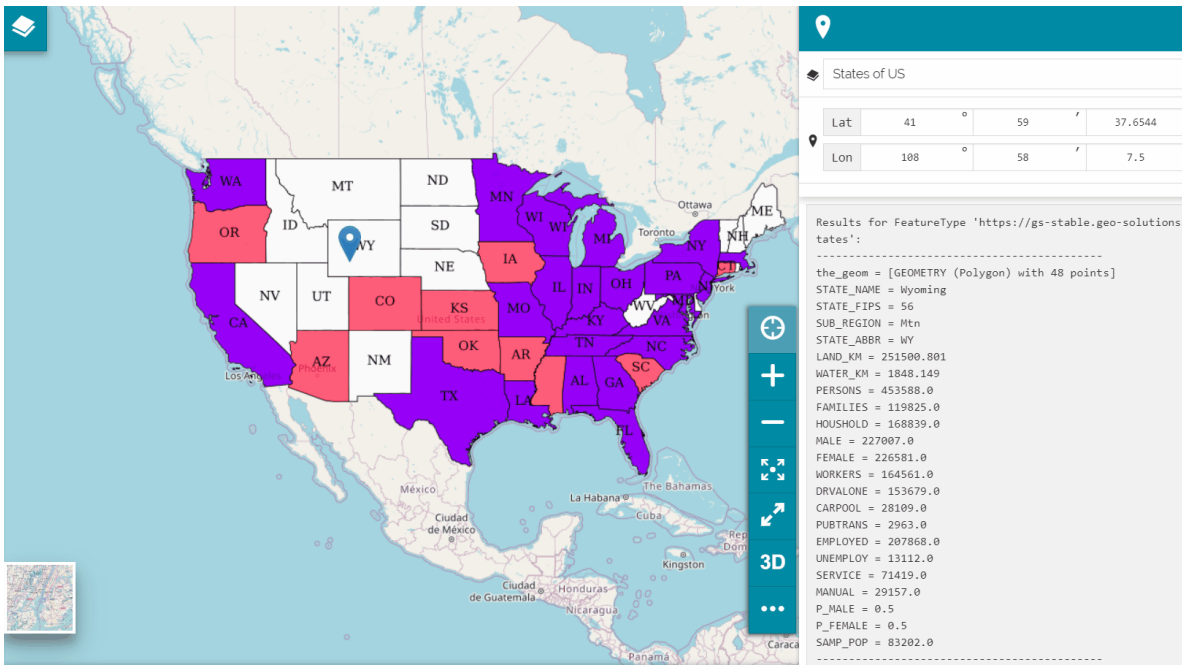
In order to Identify layers features by typing coordinates instead of clicking on the map, you can use the **Coordinate Editor**.

The coordinates can be in **decimal** or **areonautical** format depending on the user needs. It is possible to change the format by the *setting* button 

An example of search with `Decimal` coordinates as follows:



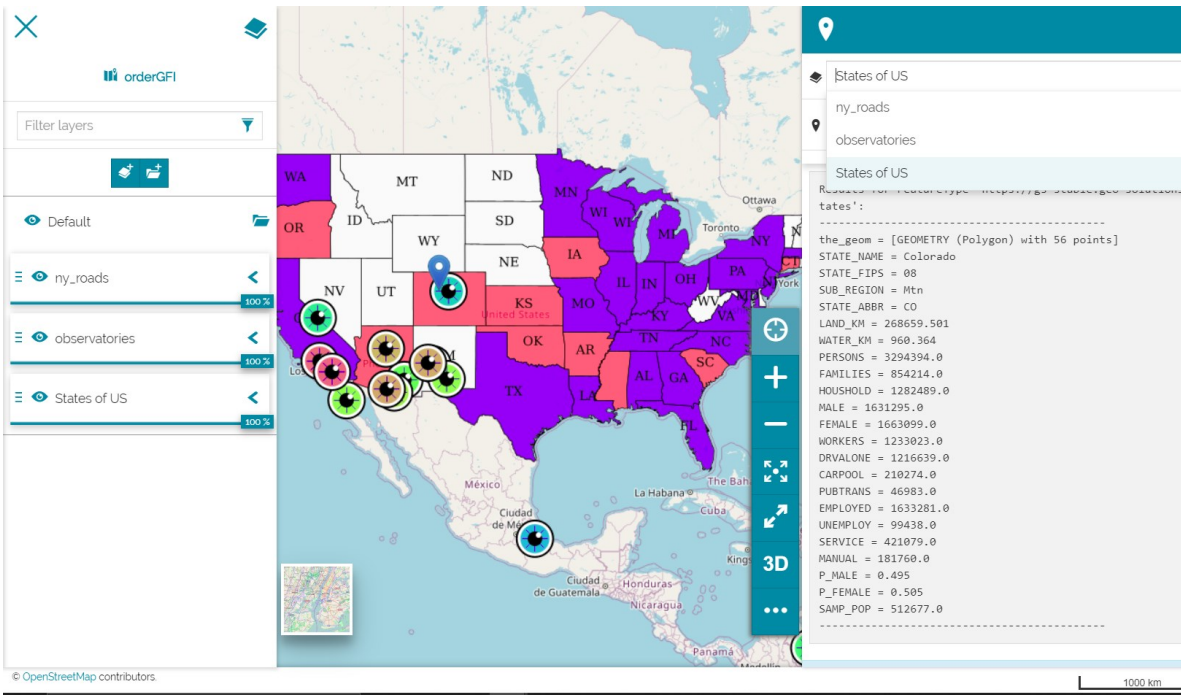
An example of search with `Aeronautical` coordinates as follows:



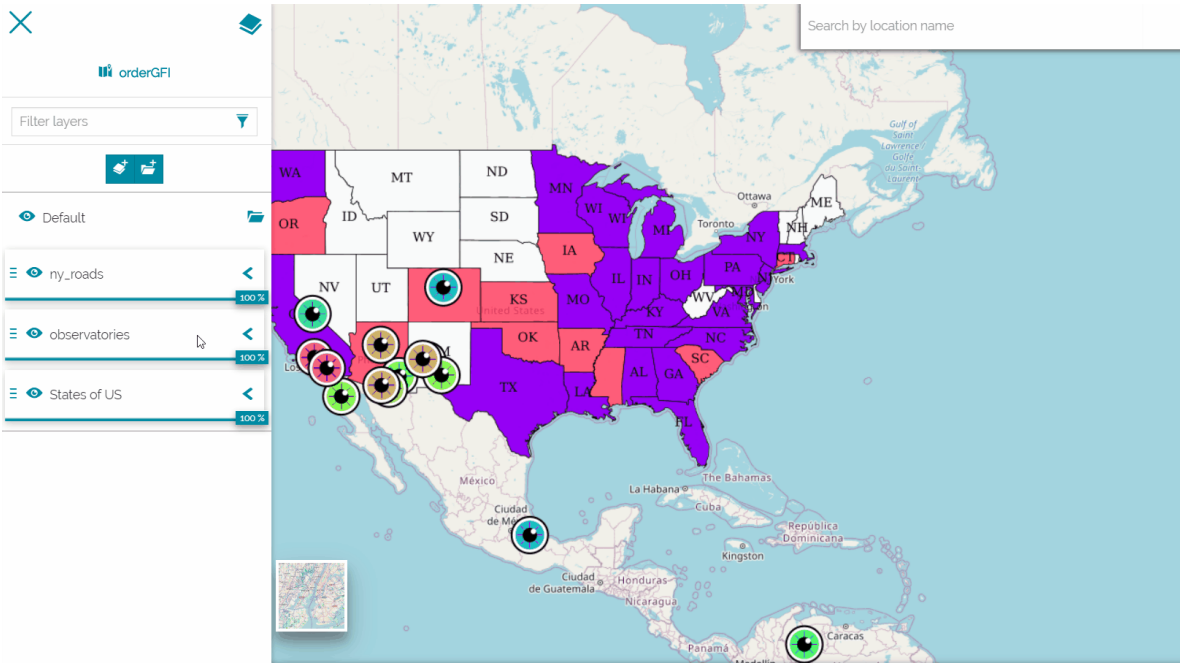
Identify Tool with more than one layer

In a map it is possible to have several overlapping layers. With the *Identify* tool the user can retrieve information on one or more overlapping layers at the same time in a certain point.

If the user clicks on the map where one or more overlapping layers are present, the identify panel opens. The panel provides the layers information, therefore the user can navigate different layers information from the **layer select** drop-down menu where the layer options have been sorted as in *TOC*.



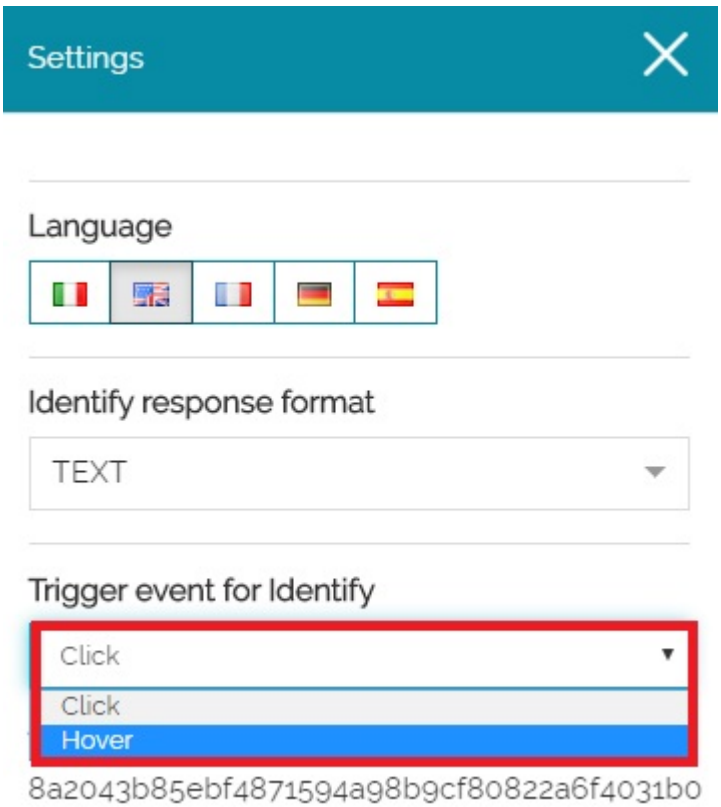
In order to have information about one layer only the user can select the layer on the Table of Contents, through the *TOC* button  , and then click on the layer in the map to perform the identify operation only for that selected layer in TOC. The identify panel opens containing the layer information corresponding to the clicked point in the map, as follows:



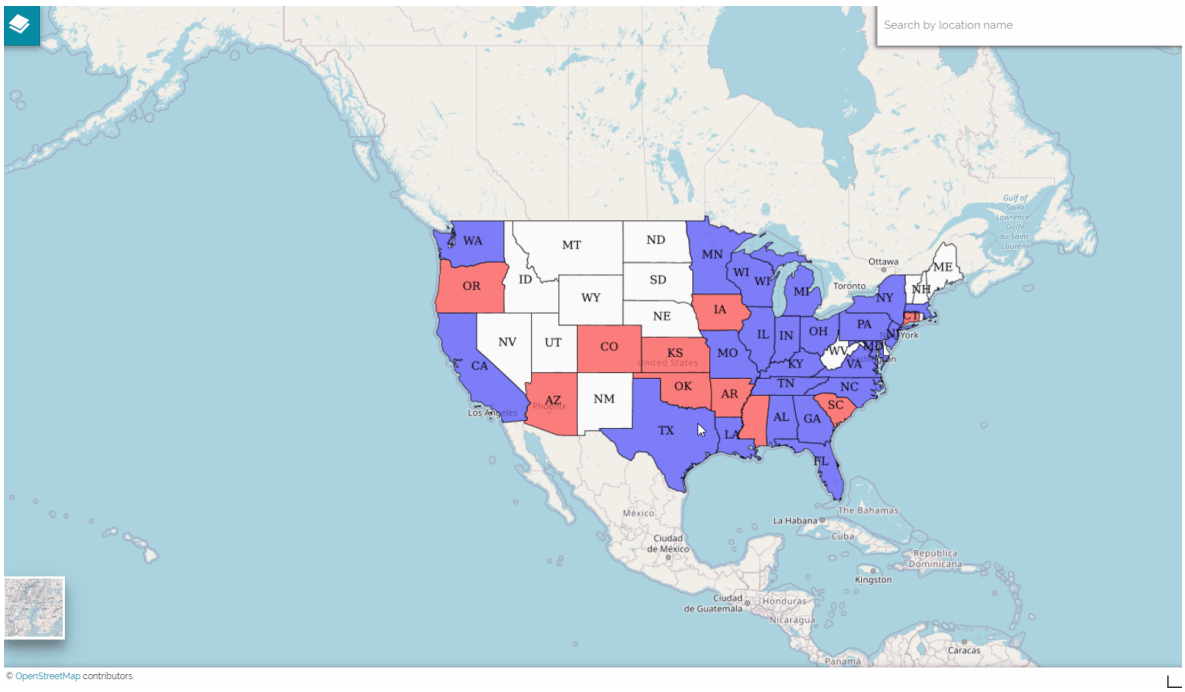
Floating Identify Tool

In MapStore the user can set the Identify tool in floating mode (**Floating Identify tool**) instead of having the default one available through a click on the map. In that case an identify popup will appears on the map as soon as the user hover over a layer in the map.

In order to activate the *Floating Identify Tool* the user can select the ⚙ button in Side Toolbar. Here he can select the **Hover** option through the *Trigger event for Identify* dropdown menu.



As soon as the option *Hover* is selected, the user can hover the mouse over a layer in the map in order to show the popup containing the identify information.


Search by location name
WA
OR
CA
ID
NV
MT
WY
UT
AZ
CO
NM
ND
SD
NE
KS
OK
TX
MN
IA
MO
AR
LA
WI
IL
MS
MI
IN
KY
TN
AL
OH
WV
GA
FL
SC
NC
VA
PA
NY
ME
NH
Ottawa
Toronto
Los Angeles
Phoenix
México
Ciudad de México
Ciudad de Guatemala
Honduras
Nicaragua
Panamá
Caracas
La Habana
Cuba
The Bahamas
República Dominicana
Kingston
© OpenStreetMap contributors.

Background Selector

The background selector, located in the bottom left corner of the *Viewer*, allows the user to add, manage and remove map backgrounds.

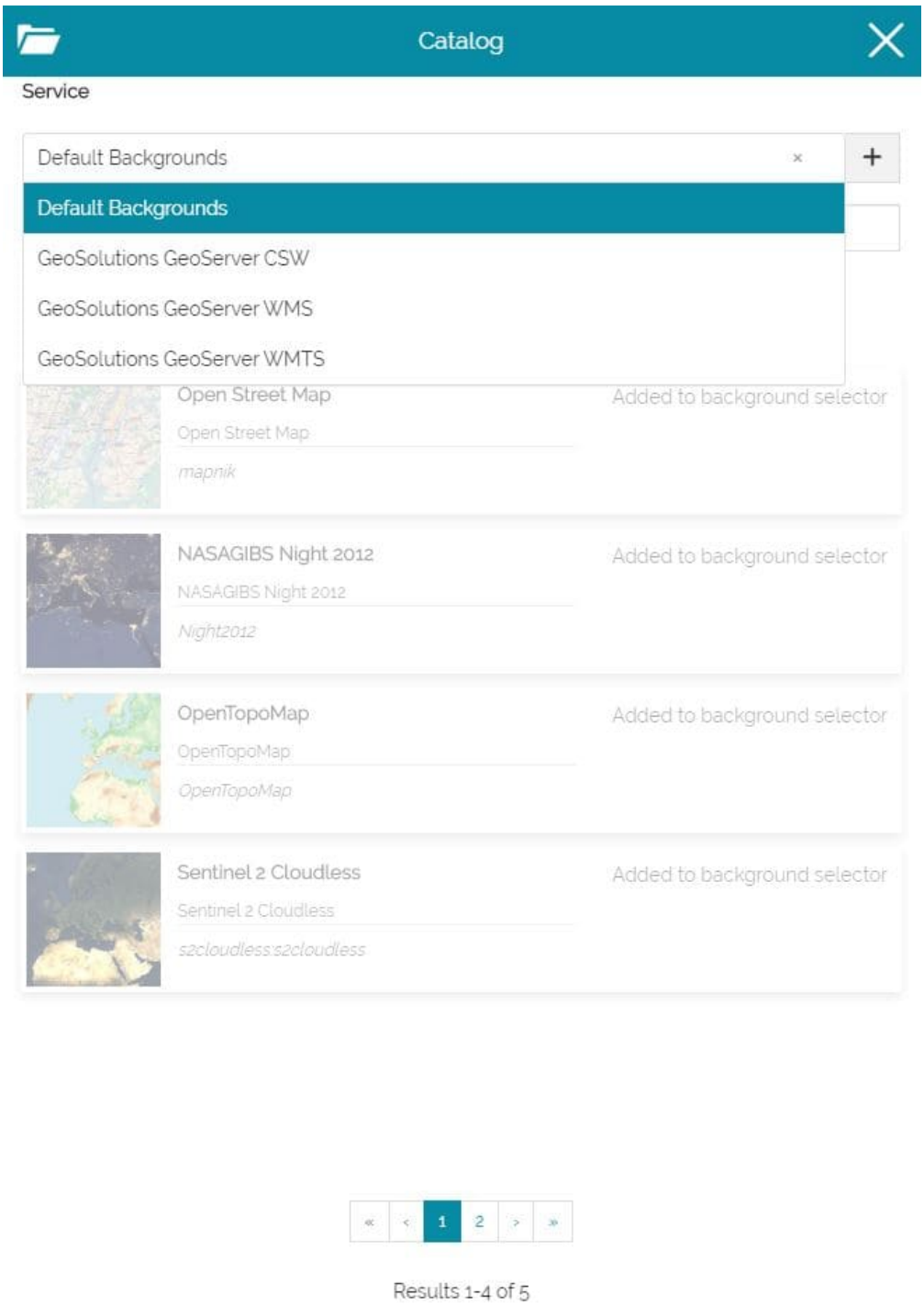
By clicking on the background selector several miniatures will be displayed. Those miniatures can be selected in order to switch from a background to another (the map backgrounds set by default in MapStore are *Open Street Map*, *NASAGIBS*, *OpenTopoMap*, *Sentinel 2* and the *Empty Background*).

For example choosing *OpenTopoMap*, the map background will change like in the following image:

If the user has editing permissions on the map (independently on the role, see Resource Properties section for more information about permissions), it is also possible to add, edit or remove backgrounds.

Add background

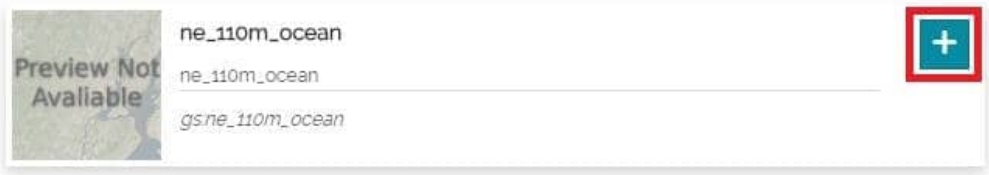
A new background can be added through the  button on the top of the background selector main card. Performing this operation the Catalog panel opens with the possibility to access the *Remote Services*:



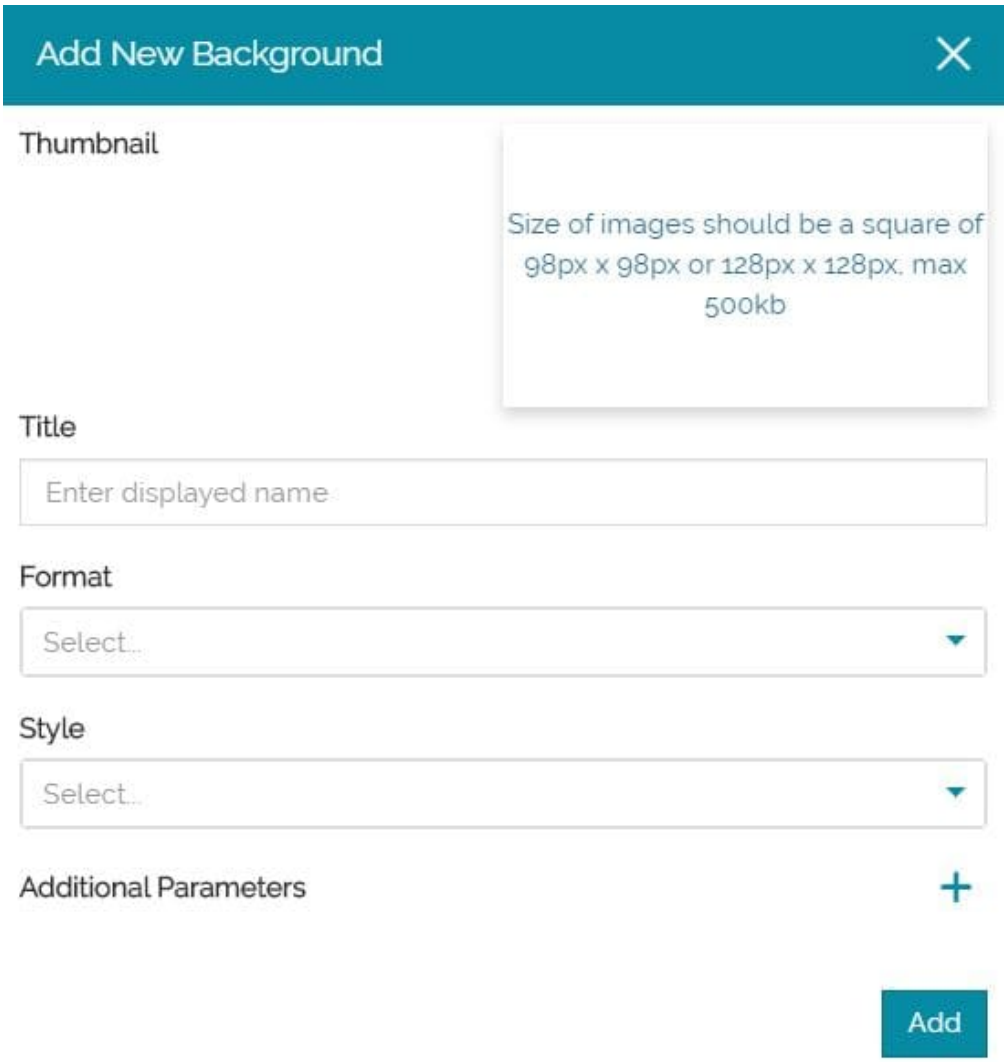
Warning

Default Backgrounds service is available only accessing the Catalog from the background selector, but if you add a new Remote Service from there, it will be available also accessing Catalog from the Side Toolbar or from TOC. *Default Backgrounds* represent a list of backgrounds that can be configured from MapStore's configuration files (more information about that can be found in Developer Guide's Map Configuration section).

From the Catalog the user can choose the layers to add to the list of backgrounds:



As soon as a WMS layer is selected, the **Add New Background** window opens:



In particular, from this window, the user can perform the following operations:

- Add a **Thumbnail** choosing the desired local file by clicking on image preview area, or simply with the drag and drop function
- Set the **Title**
- Set the **Format** (between `png`, `png8`, `jpeg`, `vnd.jpeg-png` or `gif`)

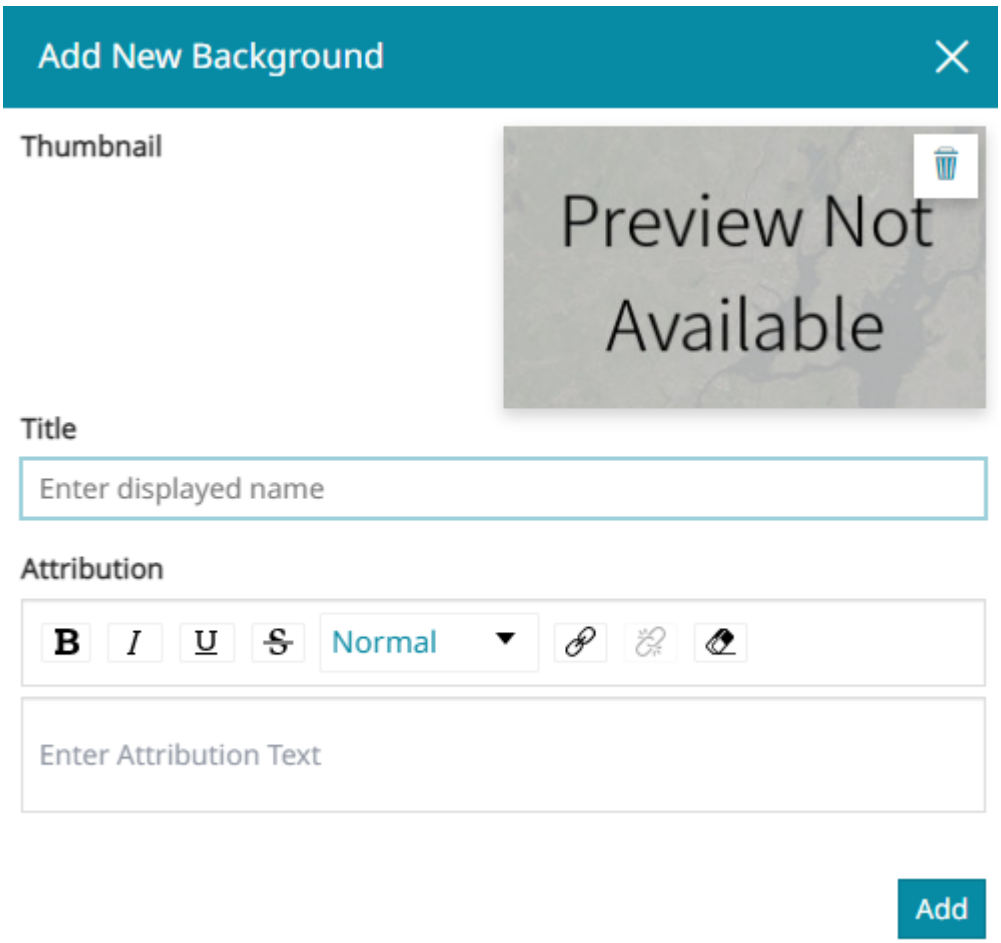
- Choose the **Style**, between the ones available for that layer
- Add **Additional Parameters** of three different types: *String*, *Number* or *Boolean* (these parameters will be added to the WMS request).

> **Warning**
>
> The thumbnail image size should be a square of 98x98px or 128x128px, max 500kb and the supported format are `jpg` (or `jpeg`) and `png`

Once the options are chosen, with the Add button the new background layer is definitively added to the background selector as a card and automatically set as the current one.

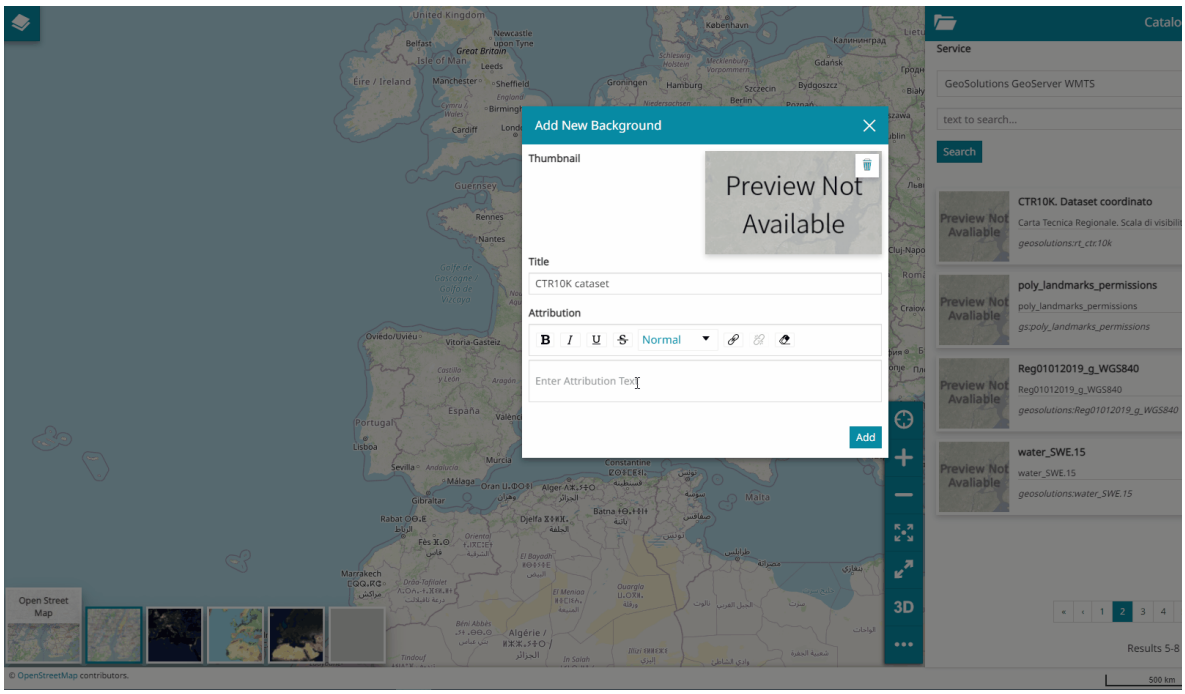
Add WMTS background

In case of a WMTS layer added as a background layer, the **Add New Background window** is a bit different:



The user can perform the following operations:

- Add a **Thumbnail** choosing the desired local file by clicking on image preview area, or simply with the drag and drop function
- Set the **Title**
- Set the **Attribution** visible at the bottom left of the footer in the map viewer.



Edit background

It is possible to edit backgrounds by clicking on settings icon on top of each background card:

> **Warning**
>
> *Default Backgrounds* layers can't be edited, with an exception for *Sentinel 2: only WMS Layers can be edited&/configured through the Background Selector*.

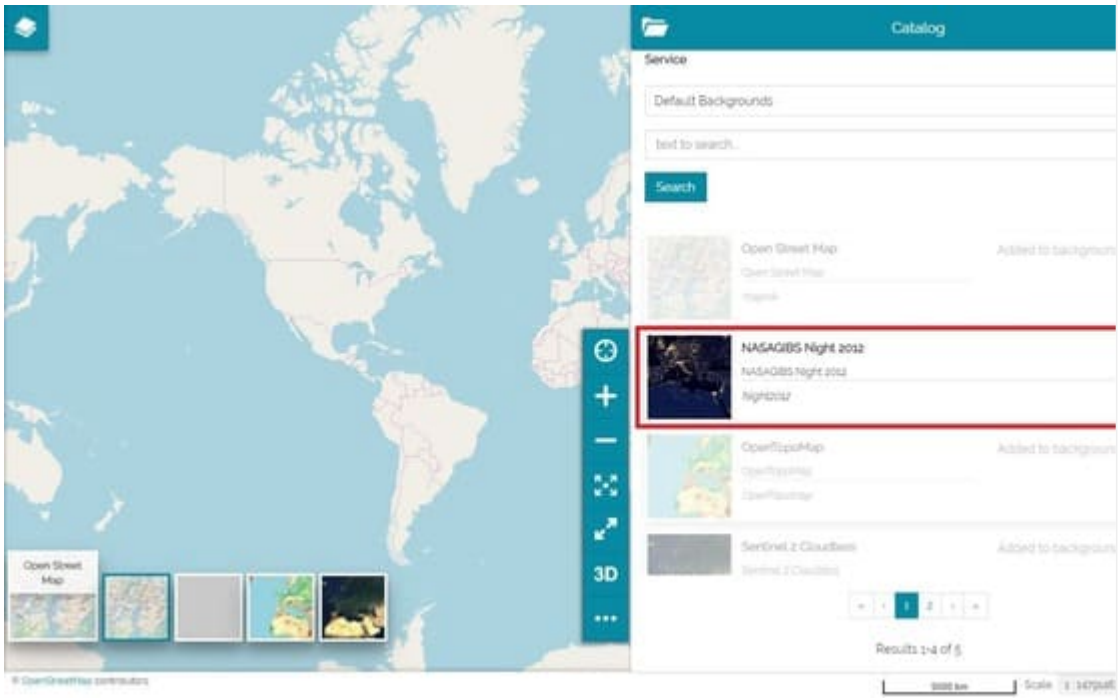
The **Edit Current Background** window opens, allowing the user to customize the same set of information when adding a new background (see previous section).

Remove background

It is possible to remove a background from the background selector by clicking on remove icon on top-right of each card

Note

By default, for new maps, all backgrounds from *Default Backgrounds* Service are added to the background selector, and in Catalog they appear grayed (it's not allowed to add the same default background twice): as soon as you remove one from the background selector, it becomes selectable from the Catalog.

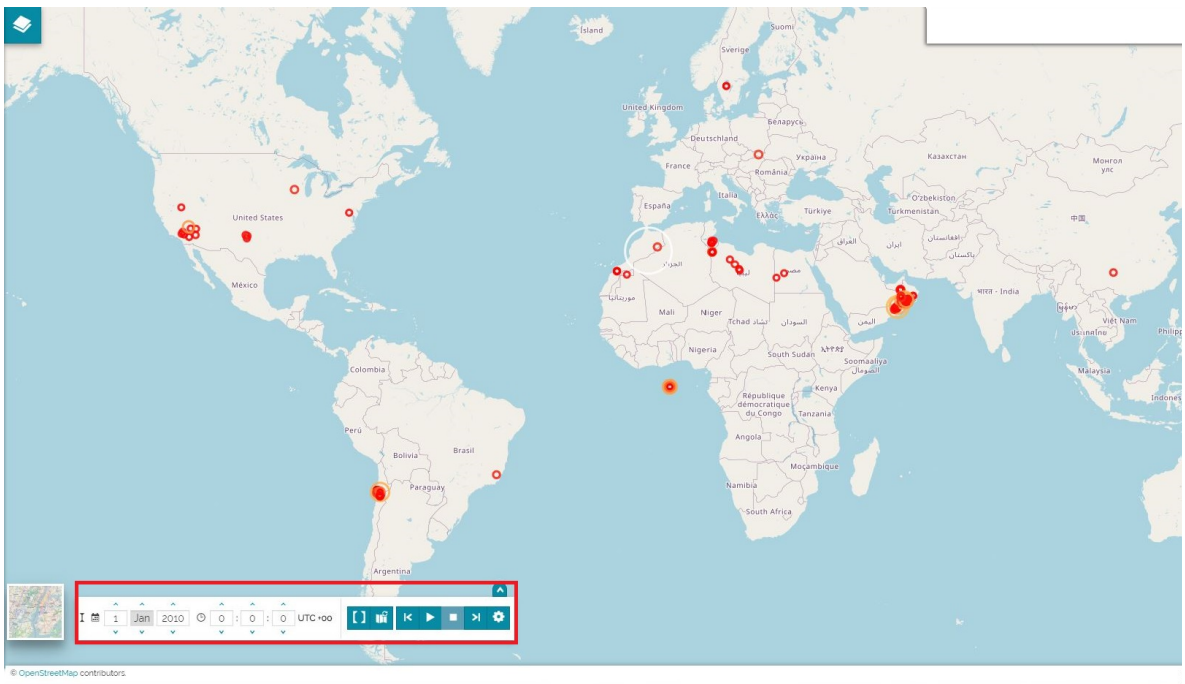


Timeline

The Timeline is a MapStore tool for managing layers with a time dimension. It makes possible to observe the layers' evolution over time, to inspect the layer configuration at a specific time instant (or in a time range) and to view different layer configurations time by time dynamically through animations.

> **Warning**
>
> The Timeline tool currently works only with WMS layers from GeoServer where the WMTS-Multidim extension is installed (WMS time values in WMS Capabilities is not supported yet). To use the MapStore Timeline at least **GeoServer 2.14.5** is required, but the recommended version is **GeoServer 2.15.2** to have a complete support for all of the features the Timeline tool can provide (e.g. the filter by viewport). From now on, the layers that the Timeline can manage will be addressed as *time layers*. From now on, the layers that the Timeline can manage will be addressed as *time layers*.

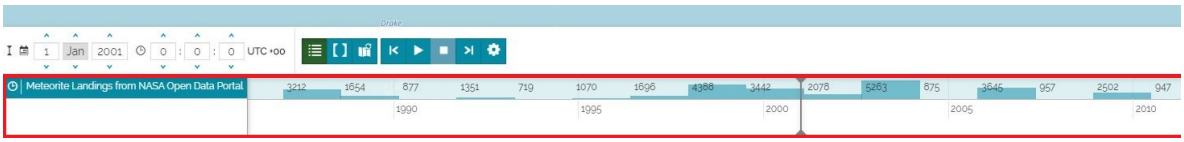
When a layer with a time dimension is added to the map, the Timeline panel becomes automatically visible and it allows the user to browse the layer over time.



> **Note**
>
> Widgets and Timeline cannot be expanded on the same map at the same time. See this section to learn more about this.

Timeline histogram

The Histogram panel opens through the **Expand time slider** button .



In the Histogram panel some of the most relevant elements are the following ones:

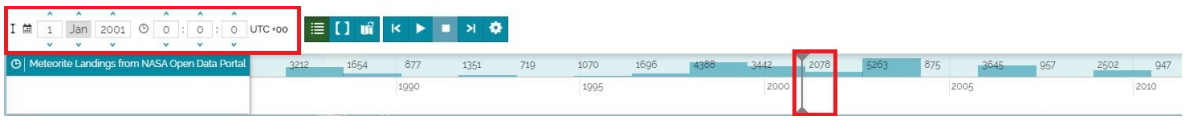
- A list of layers present in map with the time dimension available. It is possible to hide this list with the **Hide layers names** button
- The relative histogram that shows the layer' data for each time in which it is defined. In order to manage the panel the user can zoom in/out on the histogram, scroll the time axis and drag the current time cursor along it

> **Note**
>
> The highlighted layer is the one whose histogram is displayed on the panel.

> **Note**
>
> The highlighted layer in the *time layers* ' list drives the time management, from now on it will be addressed as *guide layer* (See in the Animation Settings > **Timeline Settings** > **Snap to guide layer** option).

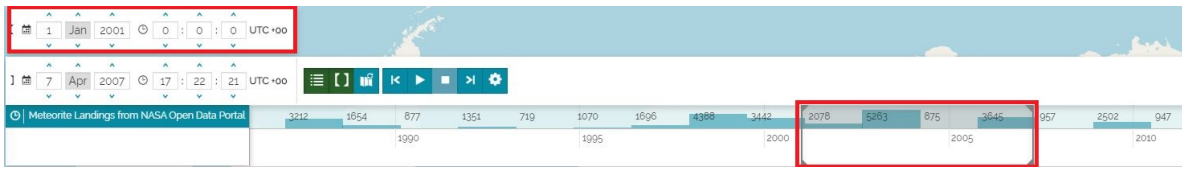
Set a Time Range

In order to see a layer in a specific time instant the user can insert a data and a time in the panel, as follows:



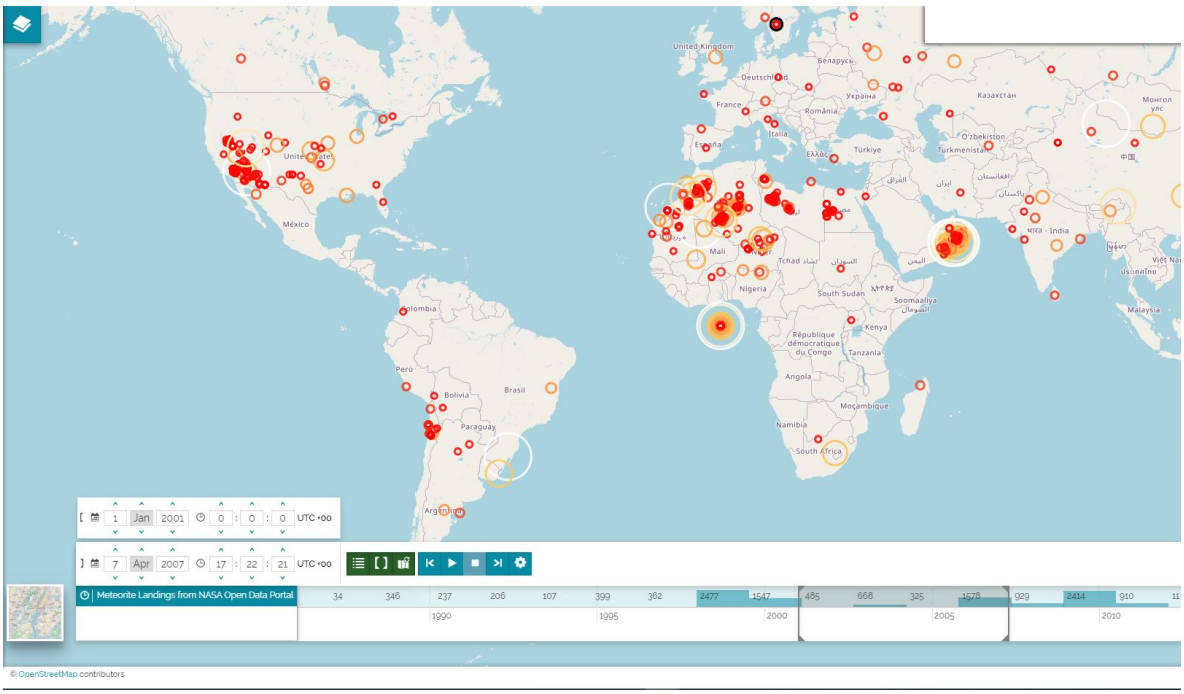
> **Note**
>
> The current time cursor changes its position according to the selected date and viceversa: when the user drag that cursor along the time axis the date/time cells will update their values.

In order to observe the layers in a finite fixed time interval the user can set a time range through the **Time range** button [ ]. A date/time control panel opens to set the range limits either by directly entering values in those cells or by dragging the limits cursors along the histogram time axis, as follows:



Show times available on map

Sometimes you might be interested to show in the timeline histogram only the times instants currently visible on the map, especially when you are exploring a big data set. This feature can be enabled by clicking the **Map Sync** button . When this tool is active the timeline will show only the times of the features available in the current map viewport.



> **Note**
>
> Map Sync feature need at least GeoServer 2.15.2

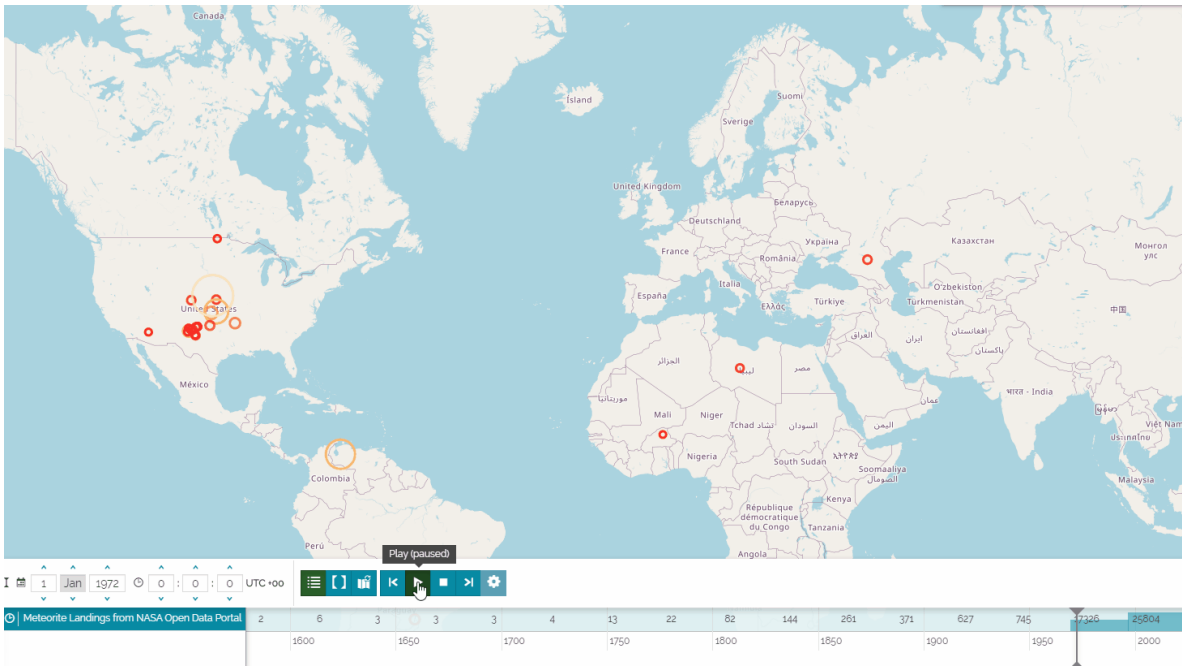
Animations

The user can start a time animation by using the timeline tool through the following buttons (by default the animation of layers in map is based on time values related the **guide layer**, see the Animation Settings section > **Timeline Settings** > **Snap to guide layer** option):

In order to start the animation the user can click on **Play** button ▶. Once the animation is started, the temporal layers in map are updated accordingly and the user can see the animation progress also in the timeline histogram. Following the sequence of steps, the cursor will shift each time to the next step in a certain time interval, the *frame duration*.
Through the **Stop** button ■ the user can stop the animation and the current time cursor remains in the last position reached.

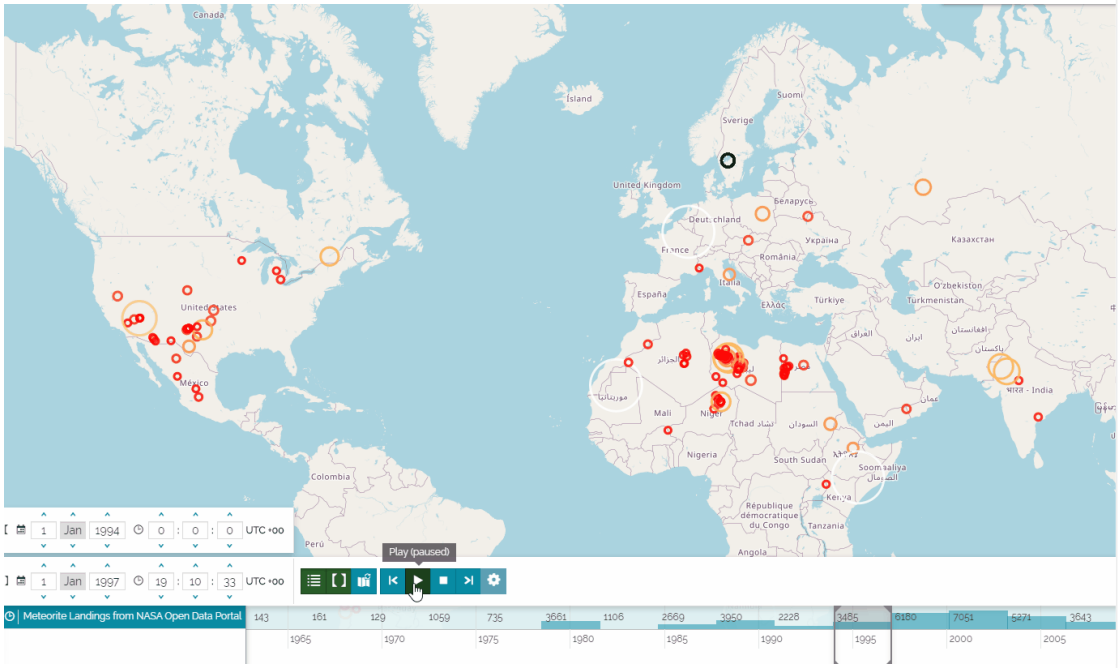
The **Step backward** button  and the **Step forward** button  allow the user to change the current time. Therefore, by clicking on one of them, the cursor changes its position (to the previous or the next step) on the histogram, the date/time values of the control cells will be updated accordingly and the layers in map are updated too.

The user can pause the animation through the **Pause** button  , as follows:



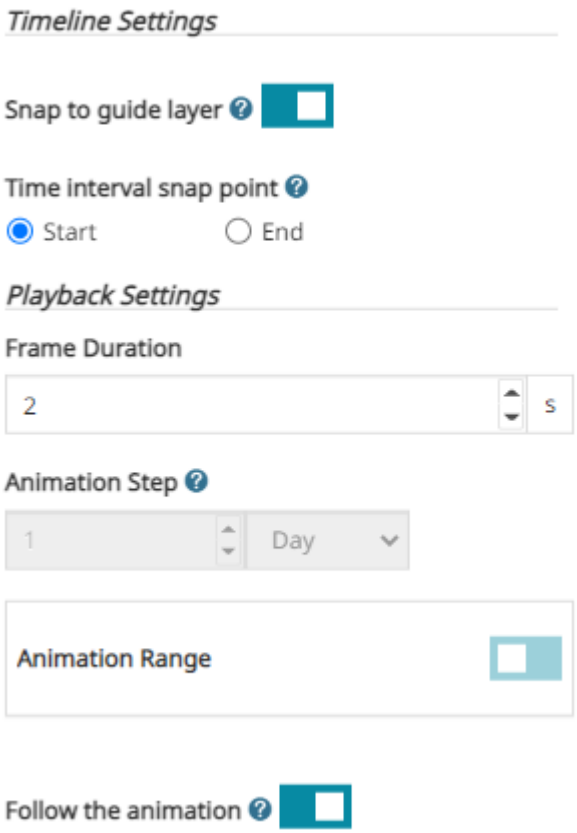
Note

The user can also specify a *time range*. During the animation, the whole range will be shifted step by step along the time axis and, in each step, the layers in map will show data corresponding to that range of time.

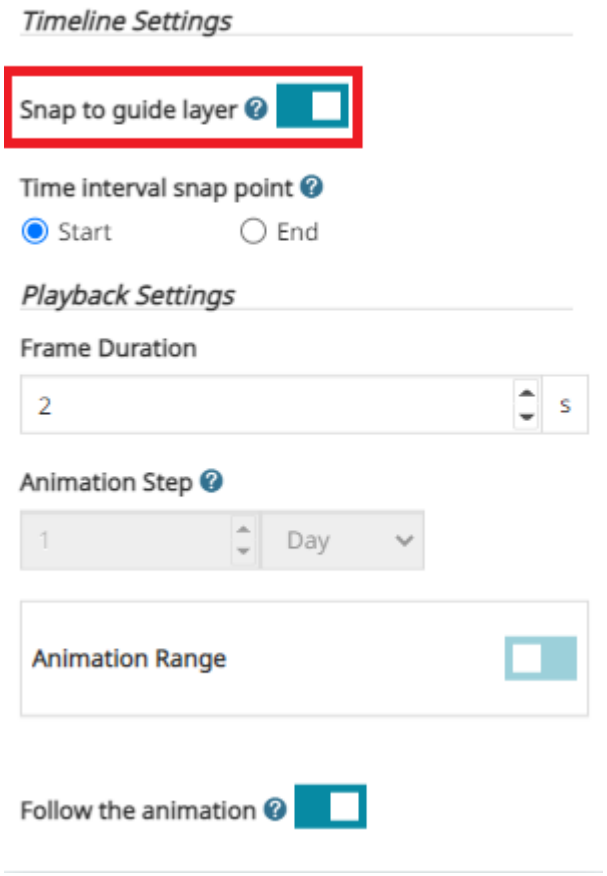


Animation Settings

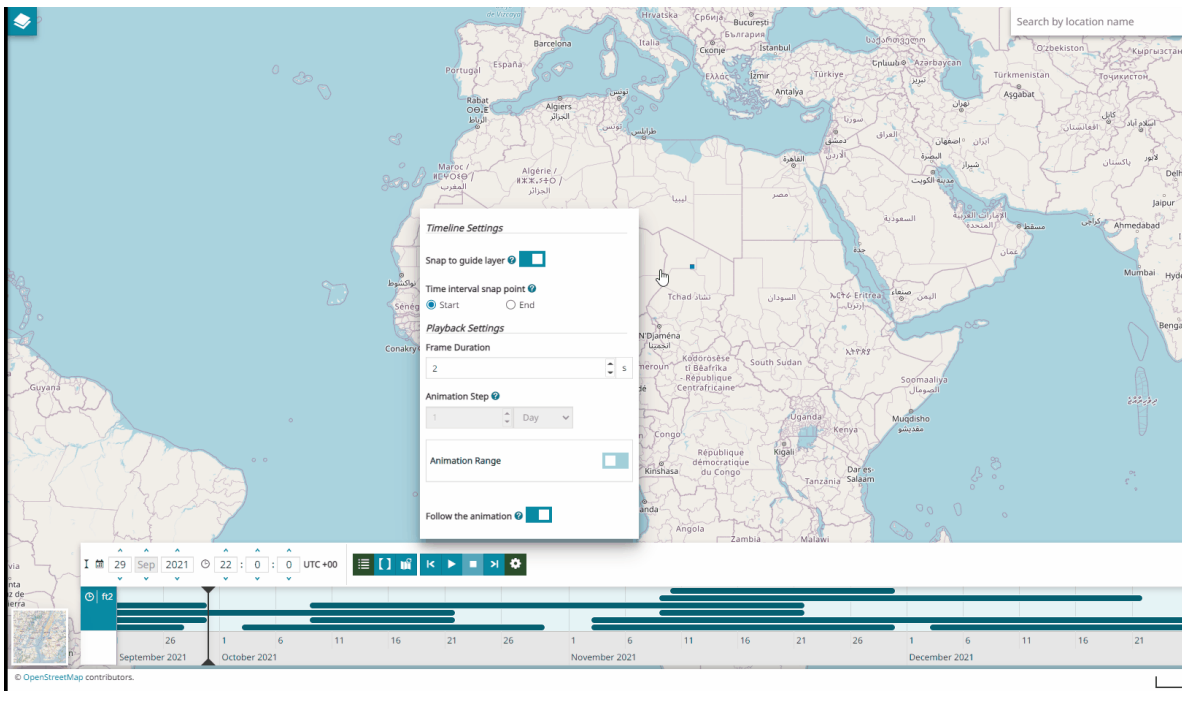
The animation behavior can be customized through the **Settings** button  . It allows the user to tune the *Timeline* and the *Playback* options.



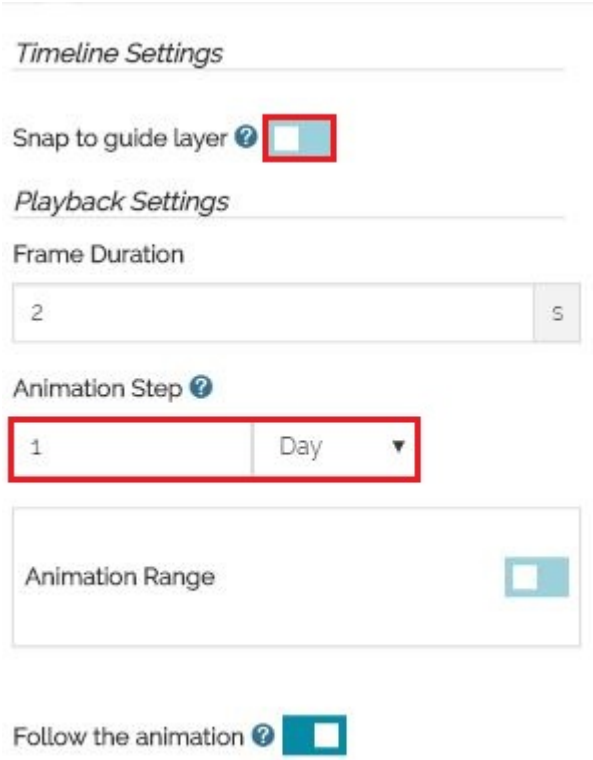
By default, the **Snap to guide layer** is enabled. It allows to force the time cursor to snap to the selected layer's data.



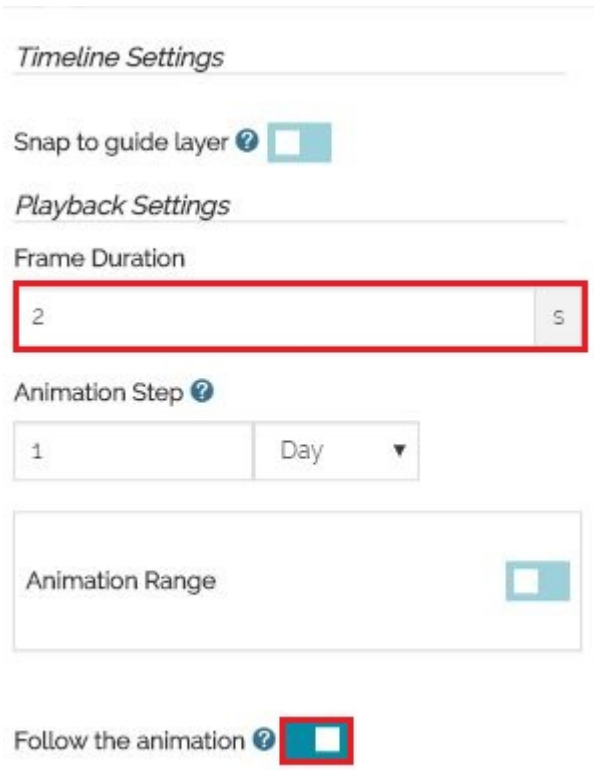
If the time dimension of the layer has time ranges defined (start/end time) instead of time instants, the user can choose the **Time interval snap point** by selecting the option `Start` or `End`. An example of snapping to the `End` point could be the following:



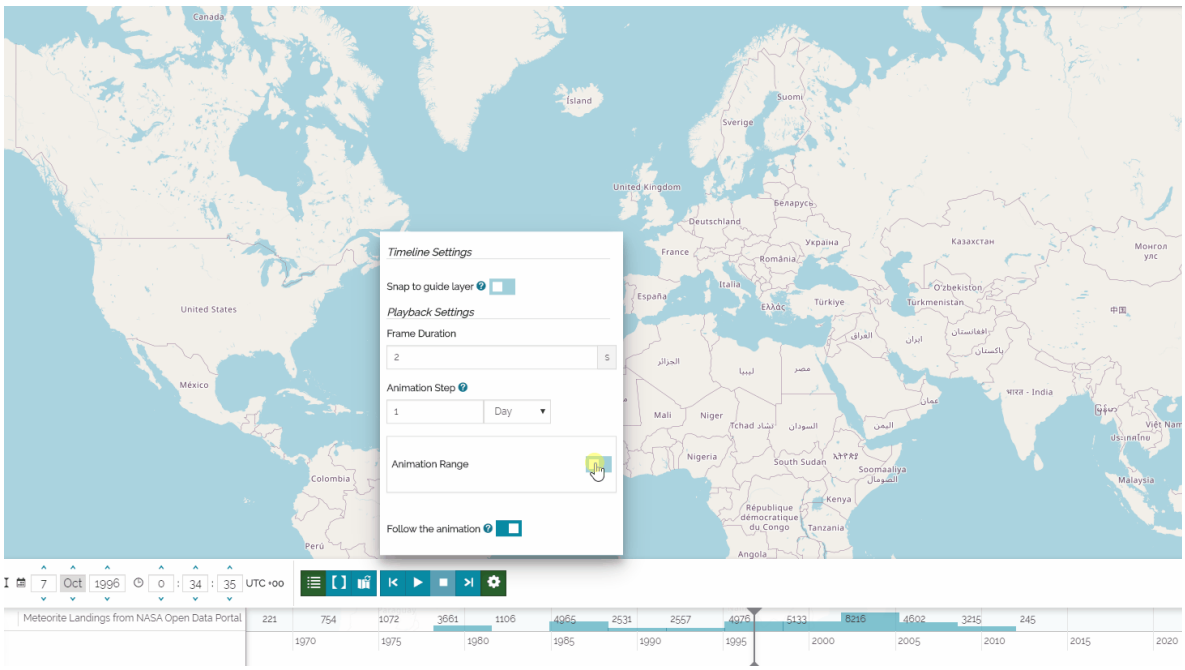
The user can disable *Snap to guide layer* to select the preferred time step through the **Animation Step** option. For example, the process could be similar to the following one:



The user can set the number of second between one animation frame and another through the **Frame Duration** and enable the **Follow the animation** to visualize the animation process also inside the histogram: the histogram will automatically move to follow the animation.



Enabling the **Animation Range** the user can bound the animation execution to a fixed time interval, the *green range*. The *green range* can be defined both dragging the *play/stop cursors* directly on the histogram or filling the *date/time control cells* of the extra panel displayed, as follows:



In order to properly set the Animation Ranger, some controls are available to help the user:

- Zoom the histogram until it fits the animation's *green range* time extension through the **Zoom to the current playback range** button 
- Extend the animation's *green range* until it fits the current view range of the histogram through the **Set to current view range** button 
- Extend the animation's green range until it fits the guide layer time extension through the **Fit to selected layer's range** button 

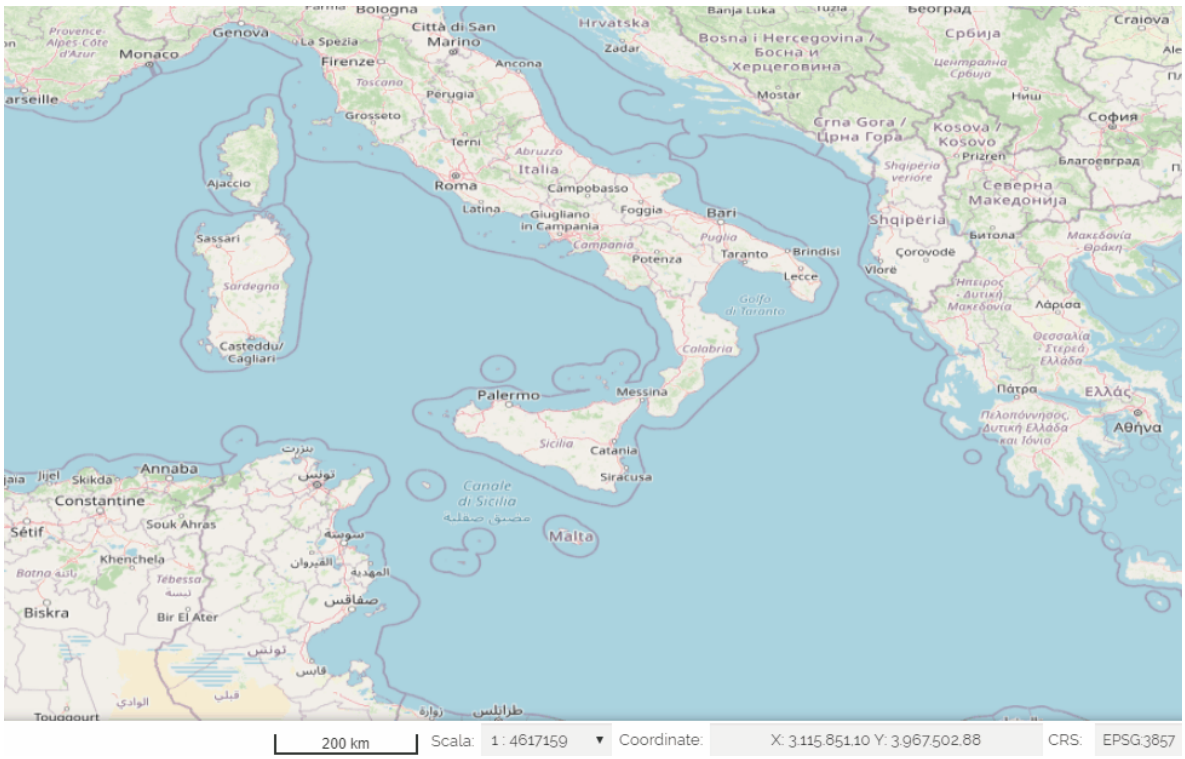
Footer

In MapStore some of the map information are reported in the *Footer*. By default, as soon as the user opens the map, the scale bar and the scale switcher are showed so that the user can change the scale bar by zooming in/out the map or by selecting a map scale through the scale switcher.

In order to visualize the map coordinates corresponding to the mouse pointer in the selected *Coordinate Reference System* of the map, the user can click on the button 

CRS Selector

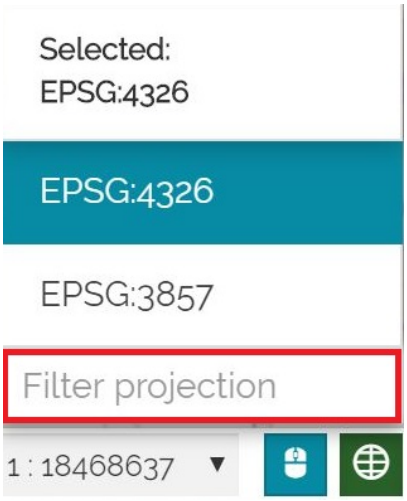
MapStore allows also to change the *Coordinate Reference System* of the map by clicking on the **Select Projection** button  . A CRS selector opens to select one of the available CRSs, as follows:



Note

The list of available CRSs depends on the CRS Selector configuration.

In order to search a desired CRS, the user can also filter the CRS list by typing in a search input field.

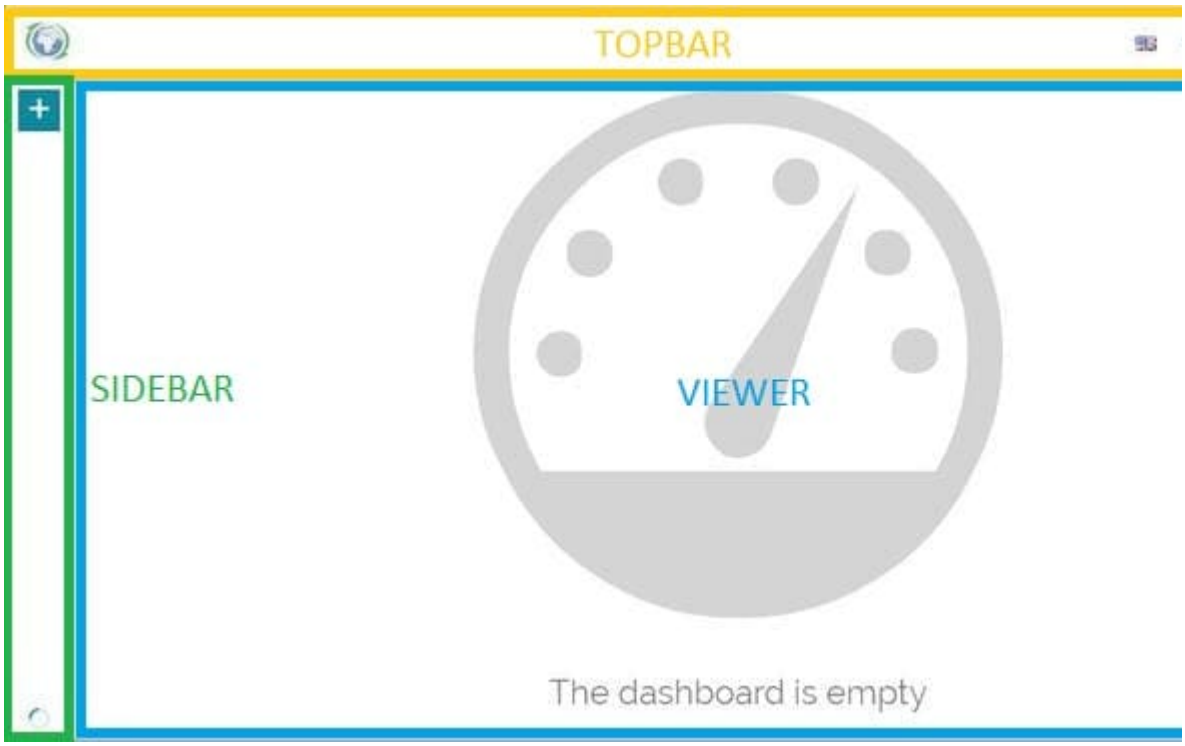


Exploring Dashboards

In MapStore, a *Dashboard* is a space where the user can add many Widgets, such as charts, maps, tables, texts and counters, and can create connections between them in order to:

1. Provide an overview to better visualize a specific data context
2. Interact spatially and analytically with the data by creating connections between widgets
3. Perform analysis on involved data/layers

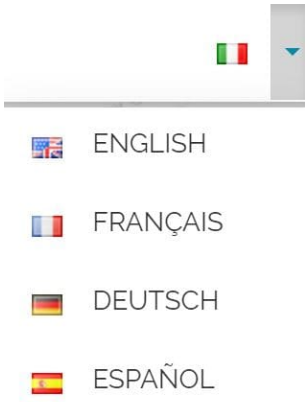
In order to create a new dashboard, the **New Dashboard** button  appears in MapStore Homepage once logged as Administrator or Normal user. With a click on it, an empty dashboard workspace appears. This page is composed of a *Topbar*, a *Sidebar* and a *Viewer*:



Topbar

Through the *Topbar* it is possible to:

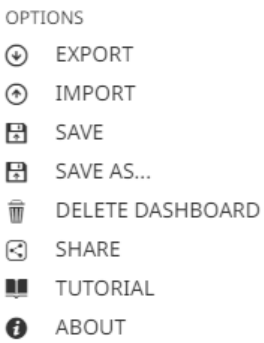
- Access GeoSolutions website with a click on the  icon
- Set the language, with the *Language switcher*:



- Go back to the Homepage with the  button
- Take a look at the account info, change password and logout, with the  button (more info about these options are available in Managing Users and Groups section)

Options Menu

In the **Options**  drop-down menu you can:



- **Export** dashboard in `json` format
- **Import** dashboard in `json` format (it will replace without asking the current dashboard)

- Save/Save as the dashboard
- **Delete** the dashboard
- Open the Share panel
- Start the **Tutorial**
- See the information about the deployed **Version** of MapStore in the **About** panel.

About

MapStore Version

Version	2022.02.xx-qa
Message	#8580 Unmute epics on plugin registration (#8583) (#8585)
Commit	6191402349d6d286fff27e6e3c8df0e5825927a8
Date	Tue, 13 Sep 2022 17:52:20 +0300

MapStore

MapStore is a framework to build web mapping applications using standard mapping libraries, such as OpenLayers and Leaflet.

MapStore has several example applications:

- MapViewer is a simple viewer of preconfigured maps (optionally stored in a database using GeoStore)
- MapPublisher has been developed to create, save and share in a simple and intuitive way maps and mashups created selecting contents coming from well-known sources like Google Maps and OpenStreetMap or from services provided by organizations using open protocols like OGC WMS, WFS, WMTS or TMS and so on. For more information check the MapStore wiki.

License

MapStore is Free and Open Source software, it is based on OpenLayers, Leaflet and ReactJS, and is licensed under the Simplified BSD License.

For more information check this page.

Credits

MapStore is made by:

GeoSolutions

Your one-stop-shop for geospatial open source software

Sidebar

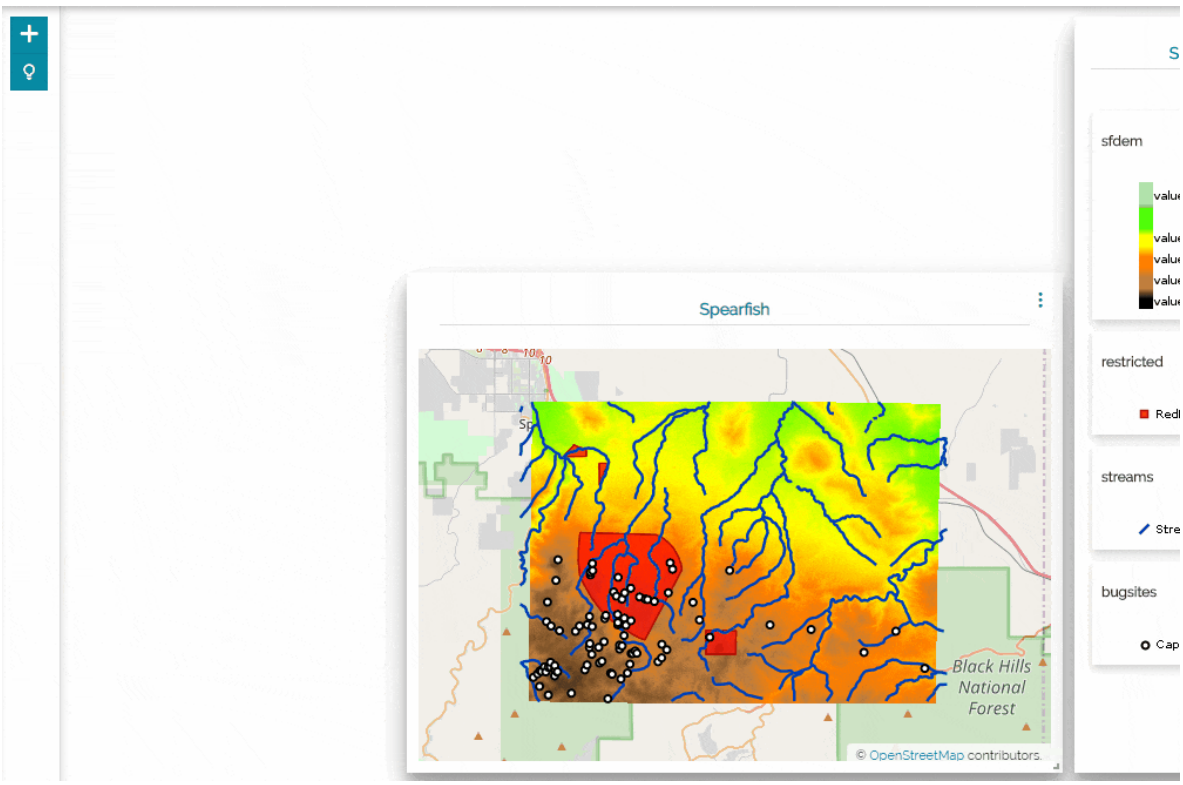
The *Sidebar* allows the user to:

- Add new widgets with the  button
- See the connections between widgets with the  button, available when connections are present (more information about this option are available in Connecting Widgets section)

Viewer

Once the widgets are added in the viewer it is possible to:

- Change widgets position by moving them with a simple *Drag and Drop* and resize them:

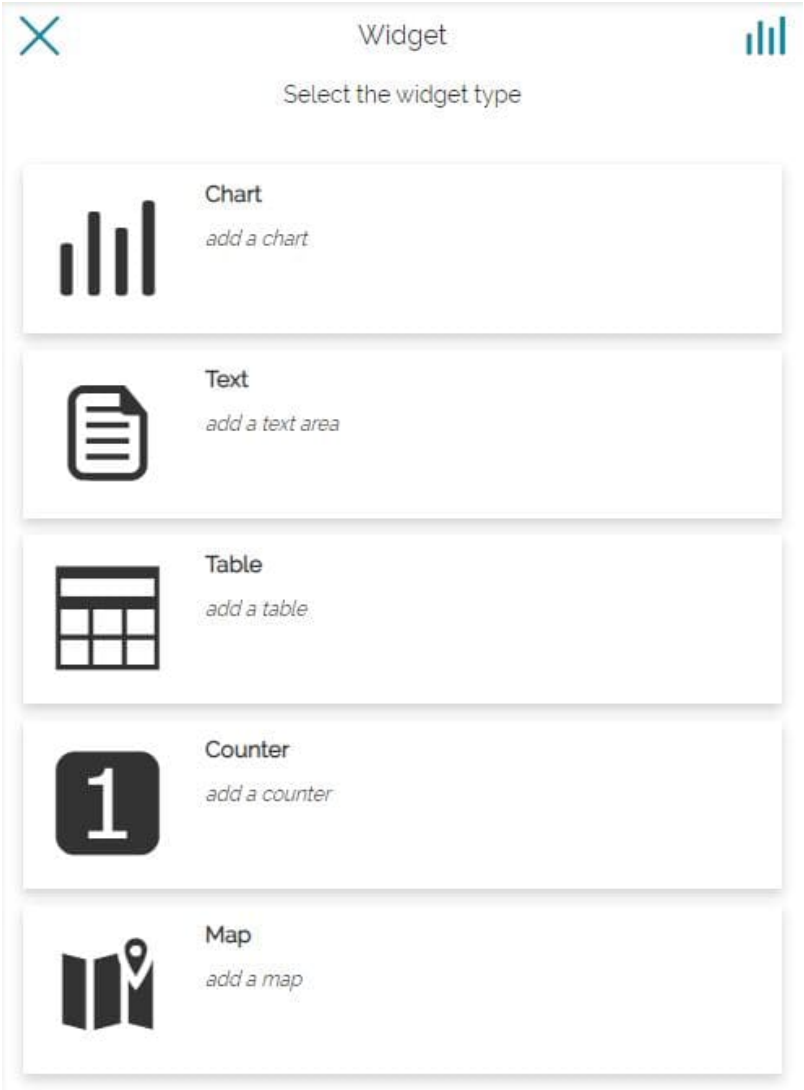


- Access widgets menu from which the user can choose between several options (more information about this menu's options can be found in Map's Access Widget Menu section)

- Take a look at the widget *Description* (more information about widget *Description* can be found in Map's Access Widgets Info section)

Adding Widgets

With a click on the + button in Sidebar the *Widget* panel opens, showing the list of the available widget types that can be added to the dashboard:



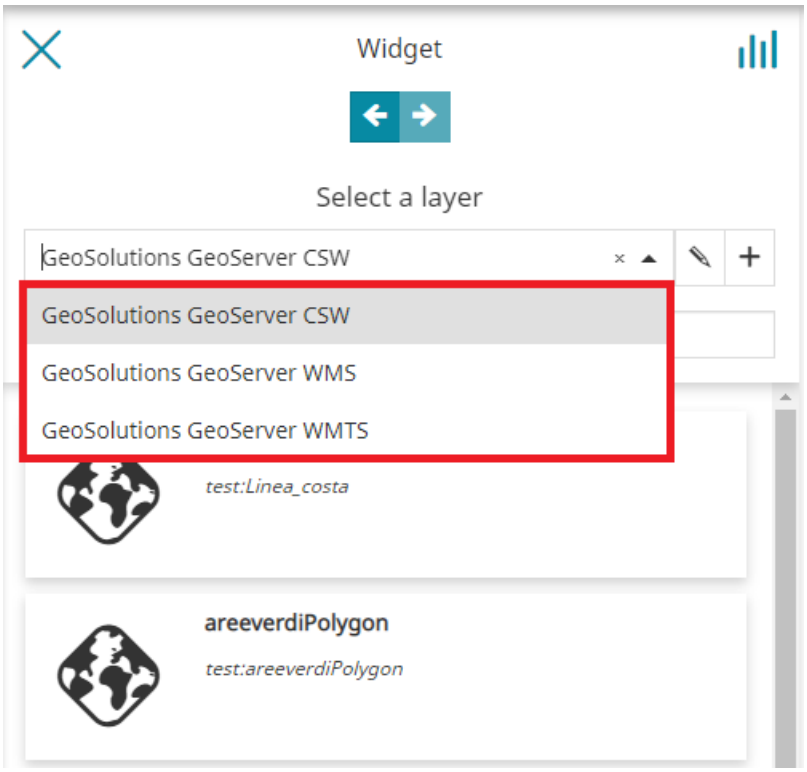
In particular, it is possible to choose between:

- **Chart**
- **Text**
- **Table**
- **Counter**

- **Map**

Creating *Chart*, *Text*, *Table* and *Counter* widgets the procedure is almost the same as that described for create widgets in maps. The only minor differences are the following:

- In dashboards as soon as the user selects the widget type, a panel appears to select the layer from which the widget will be created. MapStore allows you to choose between CSW, WMS and WMTS GeoSolutions Services, present by default, or by accessing WMS, WFS, CSW, WMTS and TMS Remote Services as explained in the Managing Remote Services section

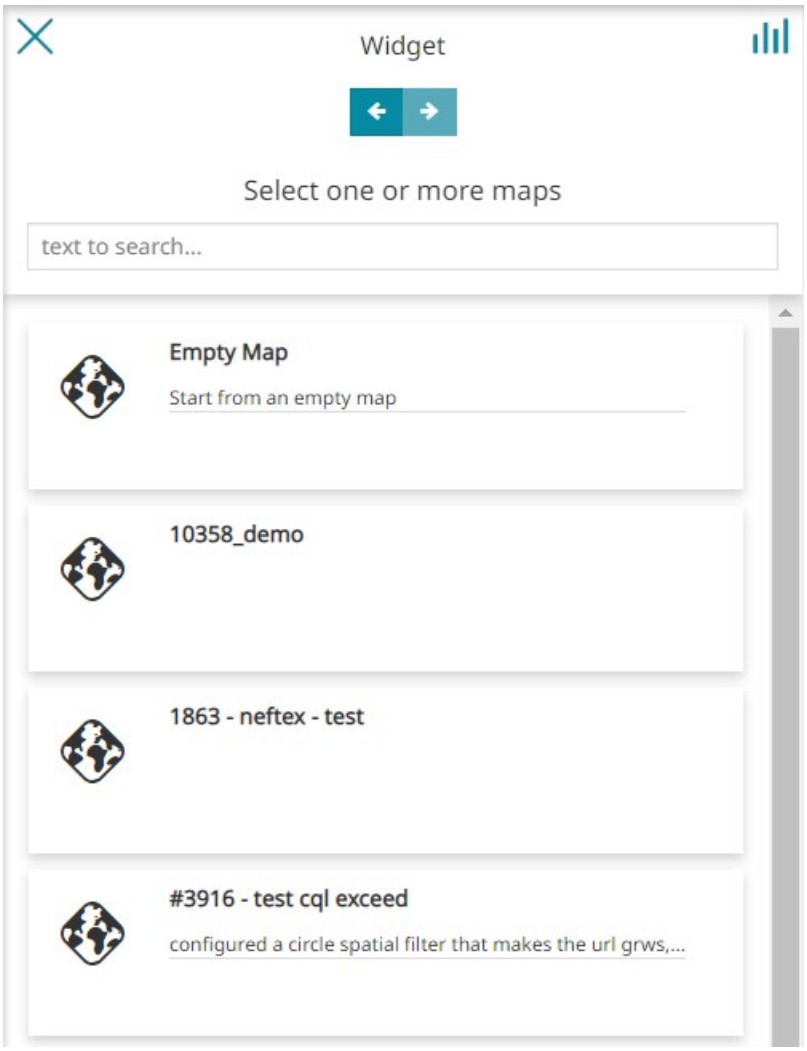


- In dashboards the possibility to connect/disconnect widgets to the map is replaced with the possibility to connect/disconnect the Map widgets together or with other widget types (this point will be better explained in Connecting Widgets section)

Creating Map type widgets, otherwise, is a functionality present only in dashboards.

Map Widget

In dashboards, selecting the Map type widget, the following panel appears:



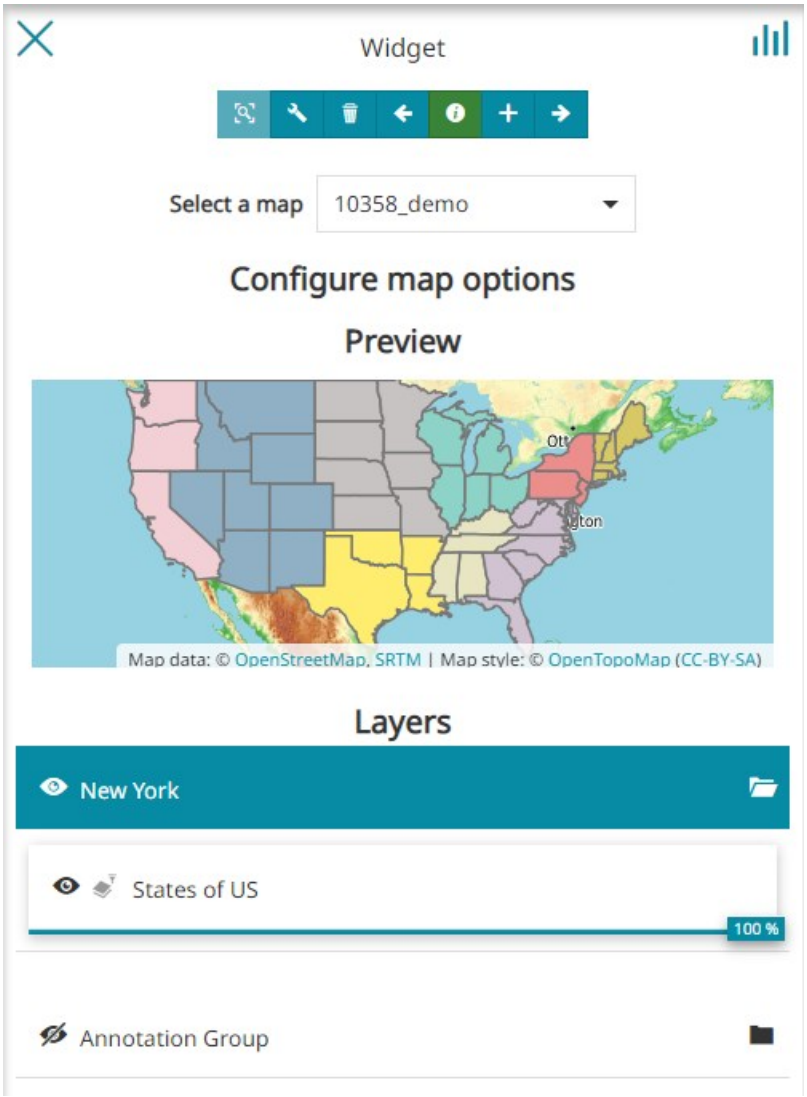
Here the user can:

- Go back to widget type selection through the button
- Search for a map by writing its title
- Select one or more maps from the list of maps (mandatory in order to move forward)
- Move forward to the next step through the button

Once a map has been selected, the panel display the layers present in the map in the preview and lists the layer associated with the map.

Note

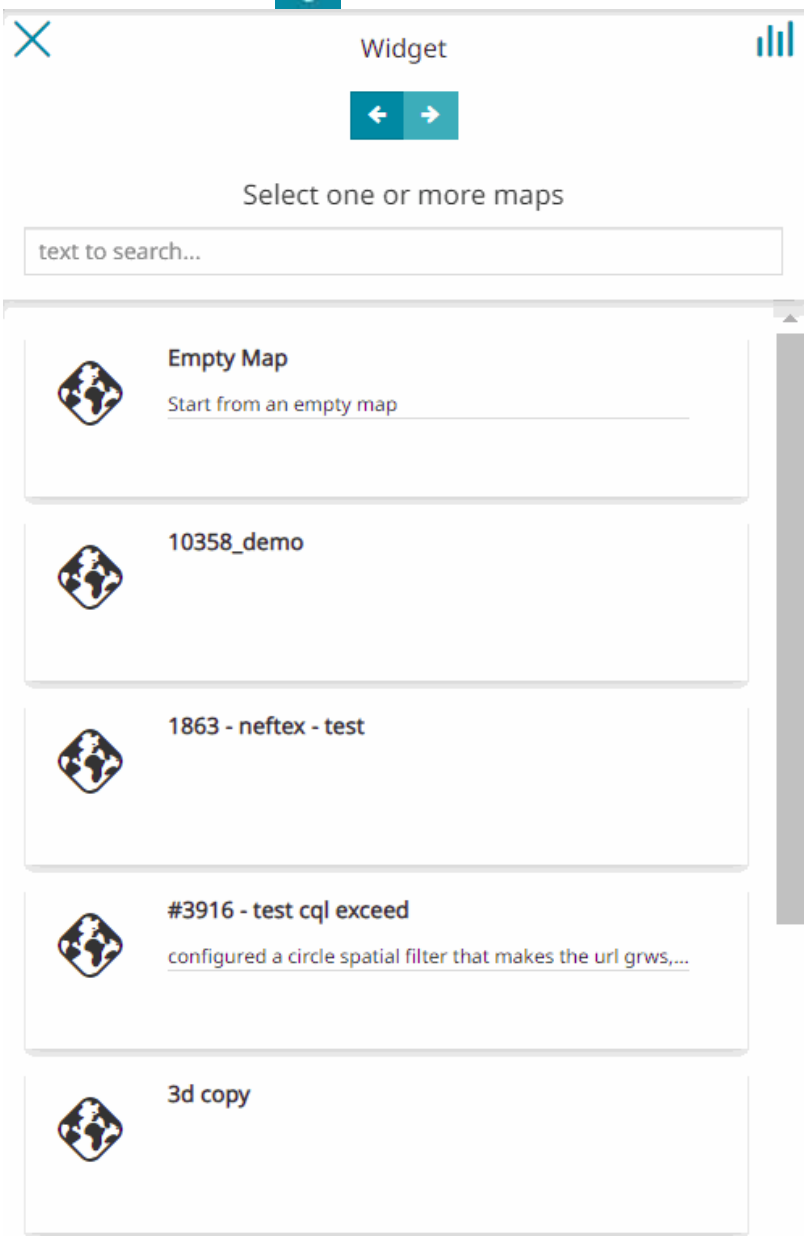
If user has selected more than one map, the map wizard displays the *map switcher* dropdown allowing user to select and configure the map.



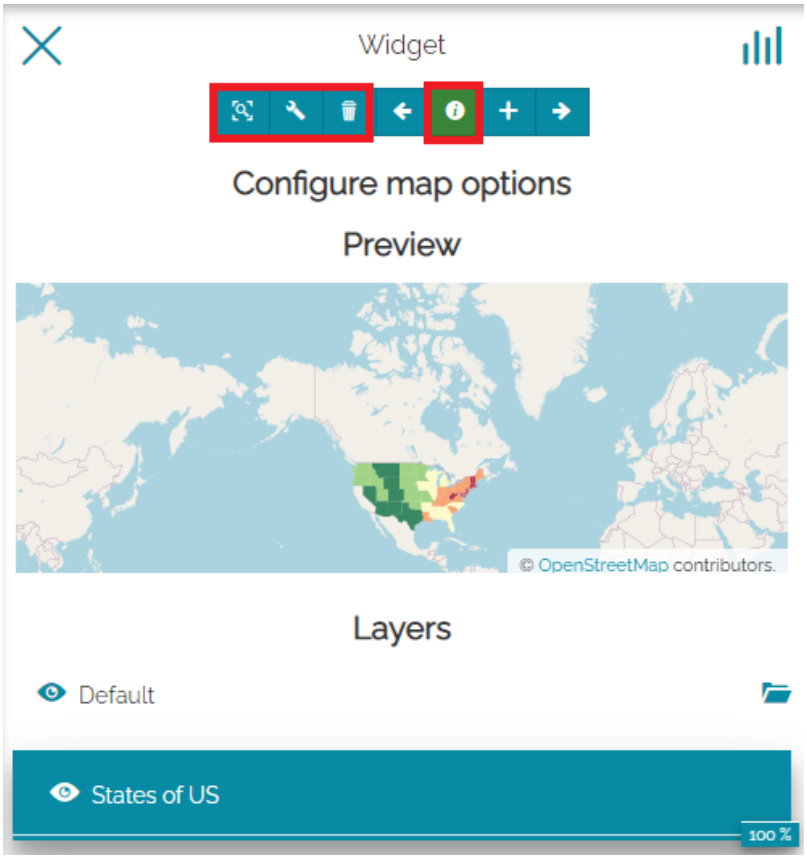
Note

If the **Empty Map** has been selected the user can:

- Create a map widget using an empty map
- If the map selection has an empty map, then the user is prompted with an option to enter map name
- Upon adding the name, the map wizard displays the map switcher allowing user to select and configure the map
- Add layers to the map through the + button, as follows:



On the **Configure map options** panel the user can toggle the layer visibility and set layers transparency, as explained in Display options section. Furthermore, the user can manage the layer with the new buttons present on the layer toolbar by selecting the layer on the layers list.



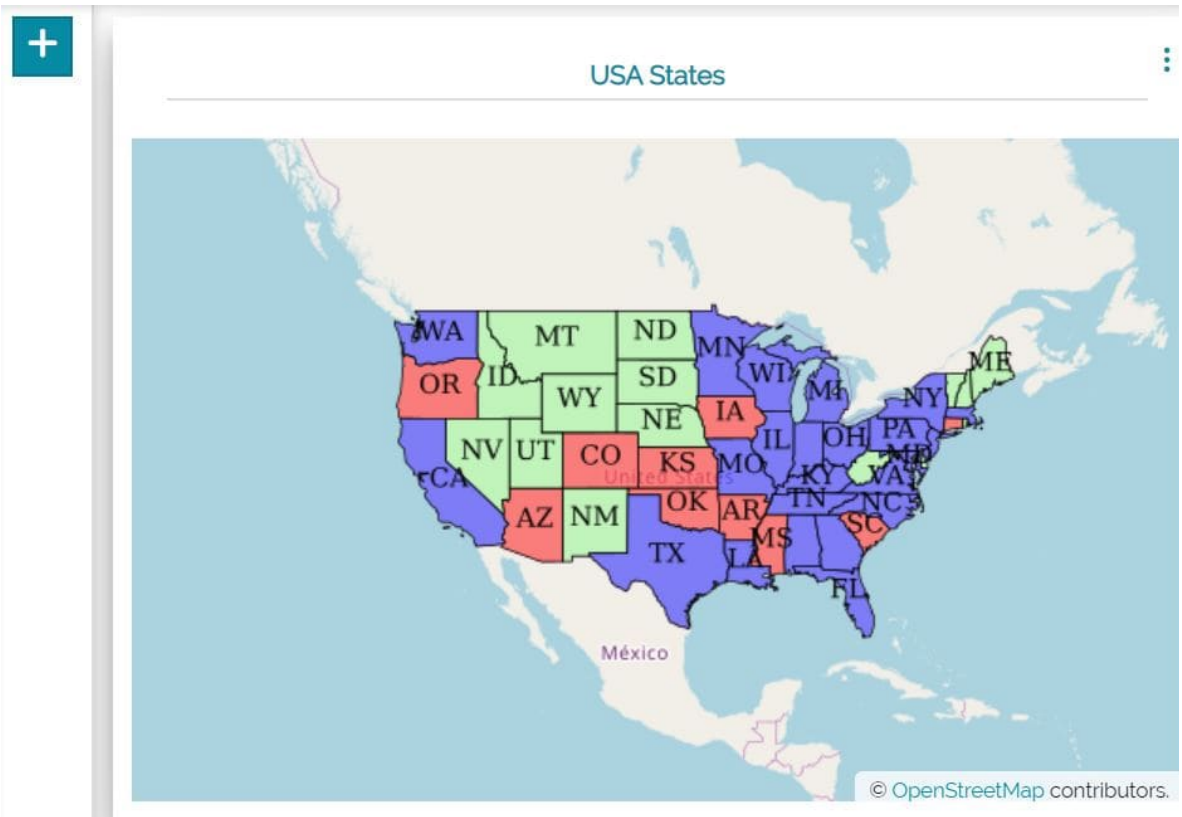
Here, the user is allowed to:

- **Zoom** to layers though the button
- Access Layer Settings through the button
- **Remove** layers through the button
- Disable/Enable the Floating Identify Tool to retrieve Identify information about layers available on the map through the button

> **Warning**
>
> The *Floating Identify* tool is active by default (the button is green)

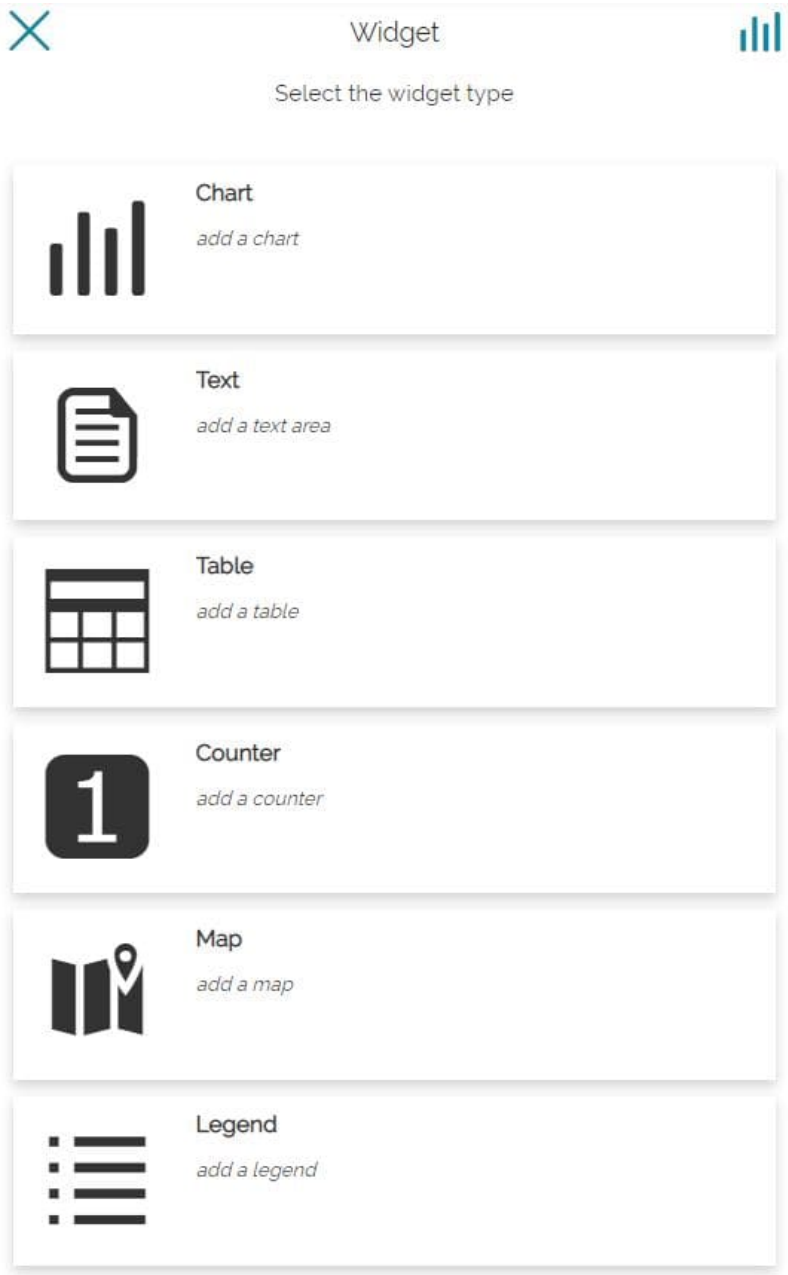
Once the → button is clicked, the last step of the process is displayed like the following:

Here the user has the possibility to insert a **Title** and a **Description** for the widget (optional fields) and to complete its creation by clicking on the 💾 button. After that, the widget is added to the viewer space:

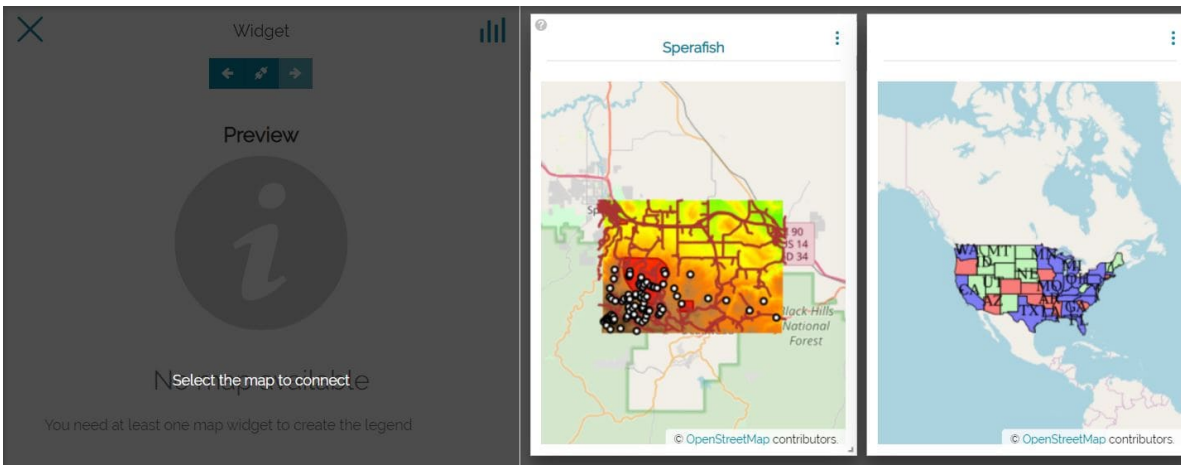


Legend widget

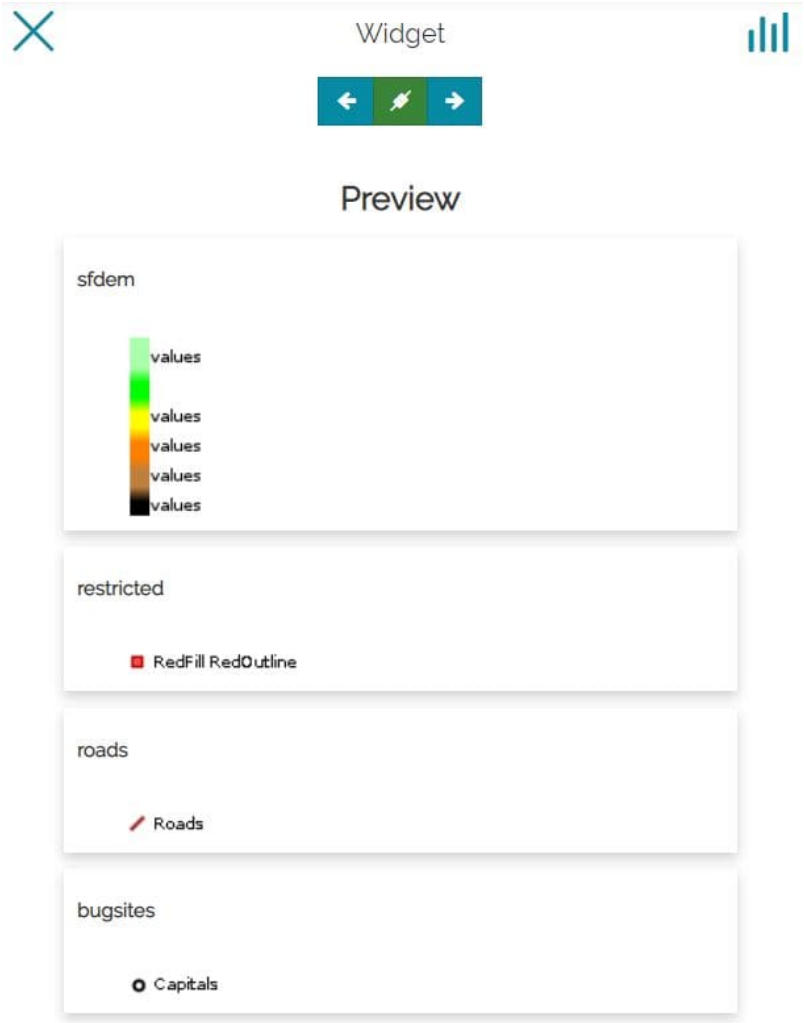
When at least one Map widget is created and added to the dashboard, there's the possibility to add also the **Legend** widget, available in the widget types list:



Selecting the Legend widget, the user can choose the Map widget to which the legend will be connected (when only a Map widget is present in the dashboard this step is skipped):



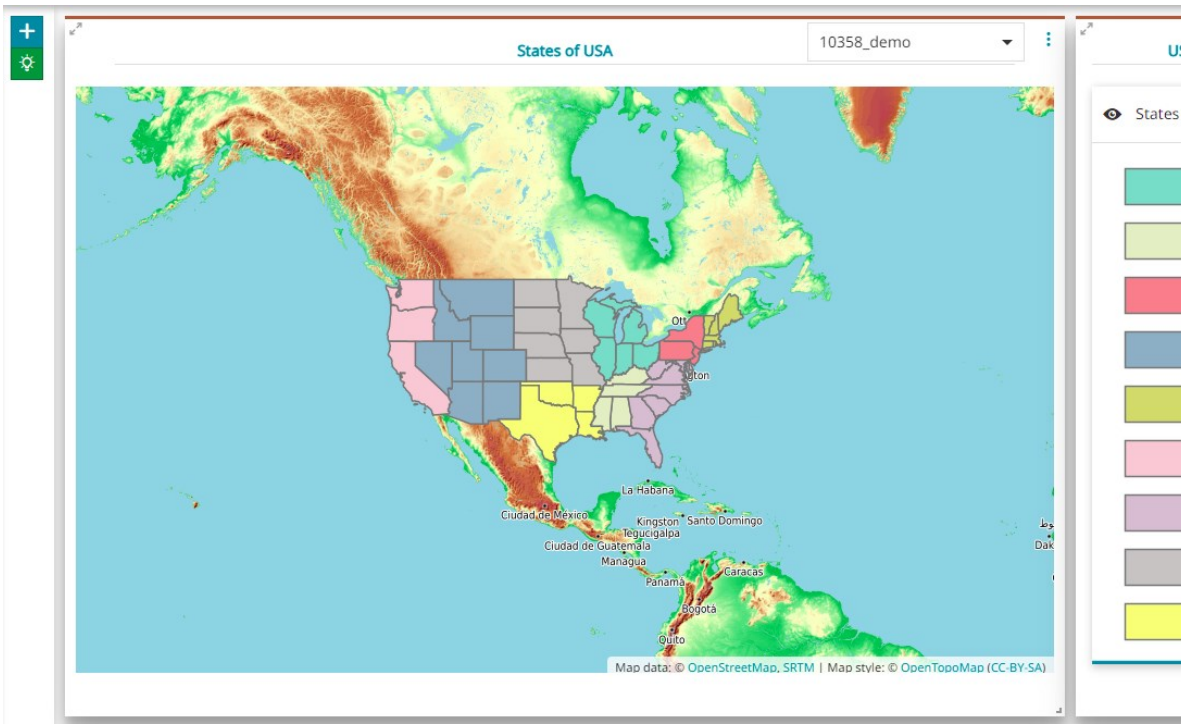
Once a Map widget is connected, the preview panel is similar to the following:



Here the user can go back  to the widget types section, connect  or disconnect  the legend to a map and move forward  to widget options. If the last option is selected, a configuration panel similar to the Map widgets one gives the possibility, before save, to set the *Title* and the *Description* for the

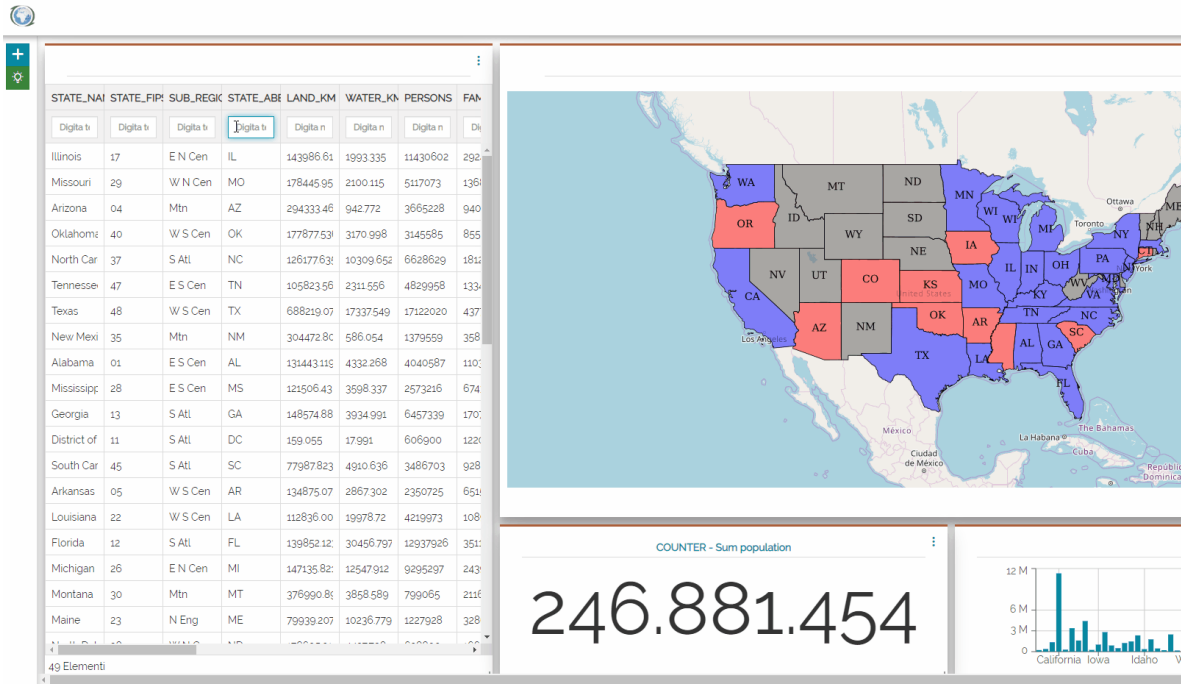
Legend widget.

An example of a Map widgets and a Legend widget is the following:

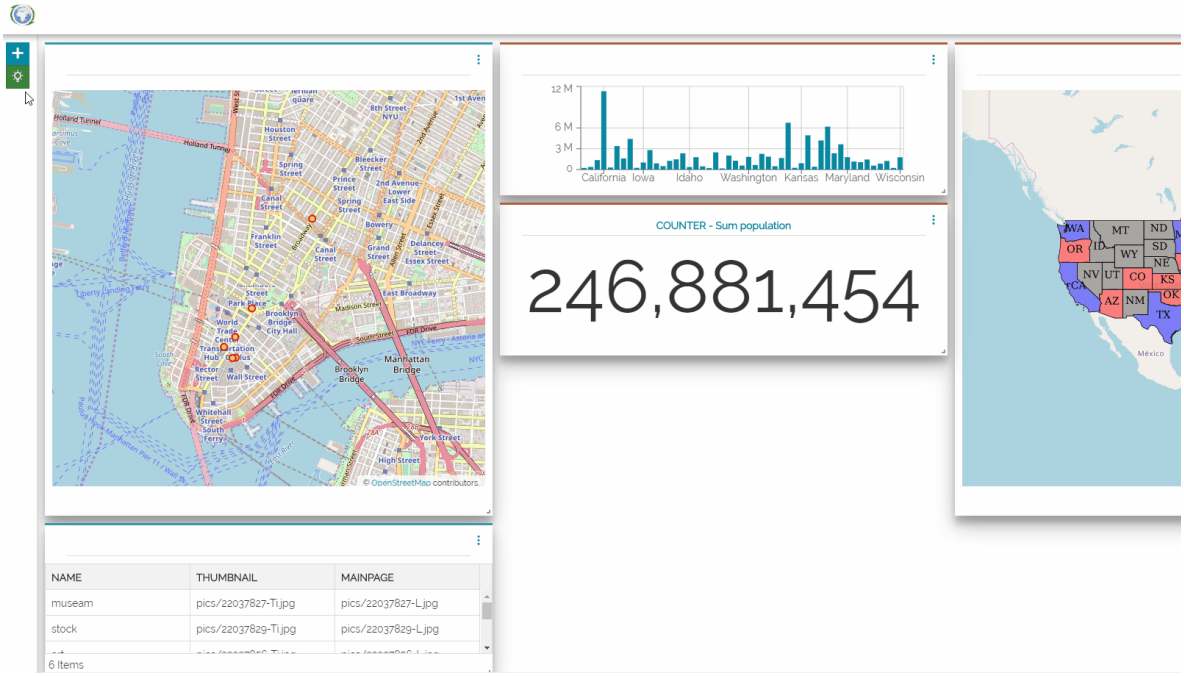


Connecting Widgets

In dashboards it is possible to connect the added widgets allowing the user to inspect and interact with more than one of them at the same time.



Once at least one connection between widgets is set, it is possible to identify the connected widgets turning on the connections button in the dashboard Sidebar making it green  . This will highlight the connected elements with a colored bar on their upper side.



In general, you can connect:

- Map widgets with other widgets
- Table widgets with other widgets

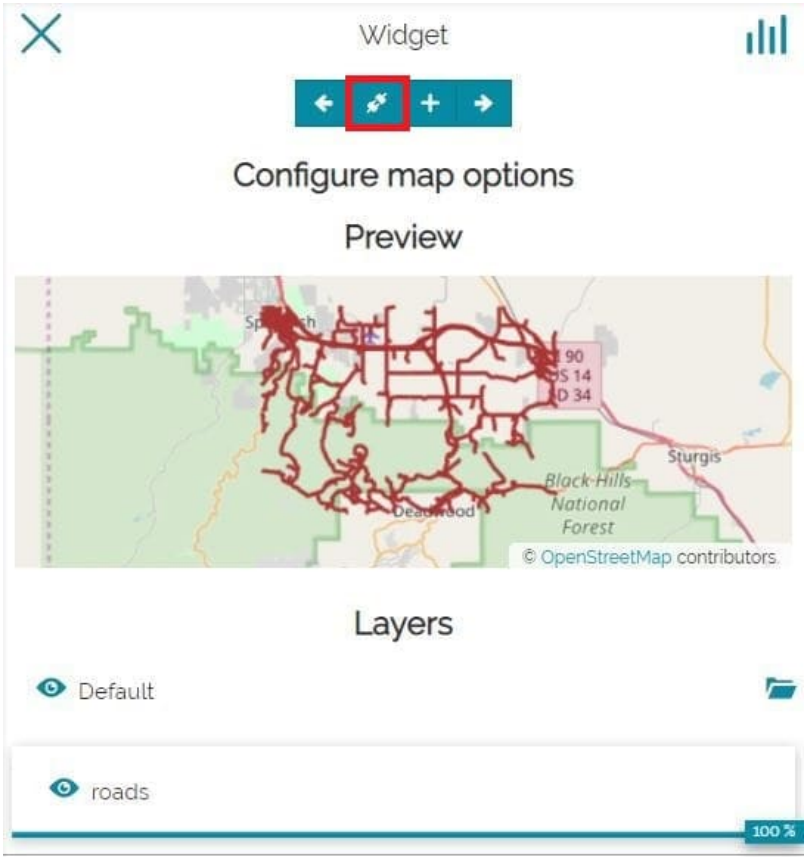
Connecting Map widgets with other widgets

In dashboards it is possible to connect Map widgets with:

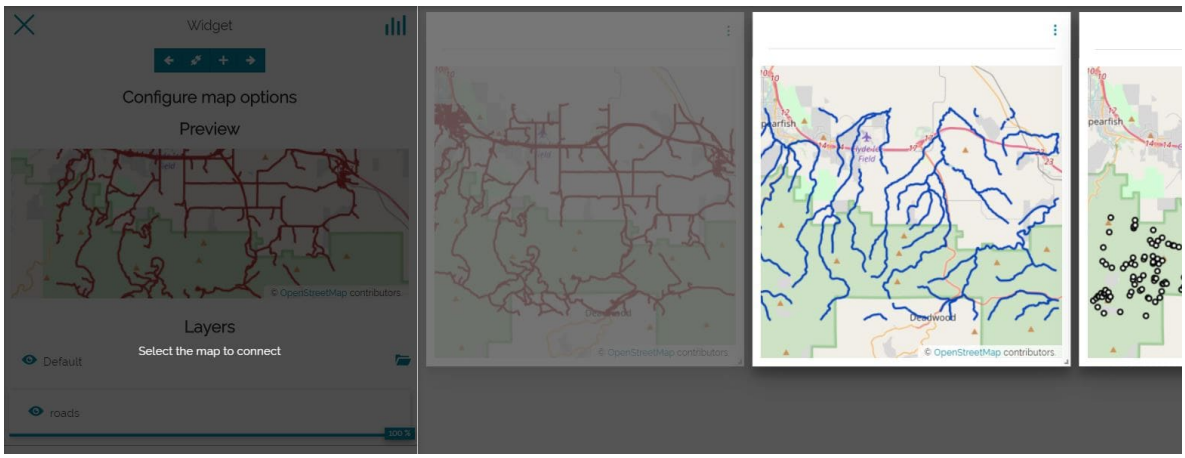
- Other Map widgets
- Chart widgets
- Table widgets
- Counter widgets
- Legend widgets

Maps with other Maps

As soon as more than one Map widget is added to the dashboard, the connect/disconnect button appears inside the *Configure map options* panel (accessible by adding a new Map widget or editing an existing one).



With a click on it, if only another Map widget is present, by default the connection will be made towards that Map widgets. When more than one Map widget is present in the dashboard, instead, it is possible choose one through a page like the following:

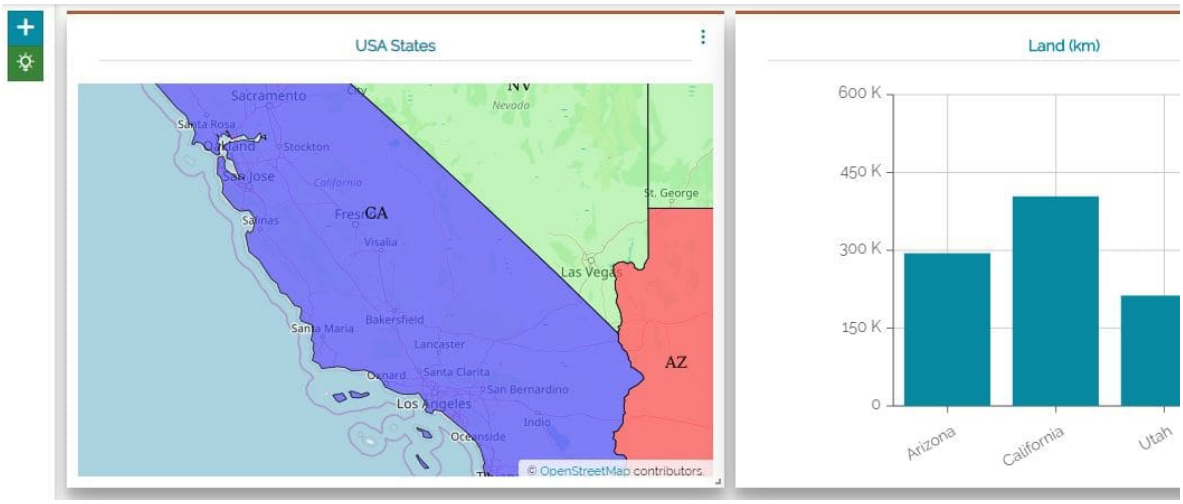


Maps with Charts, Tables and Counters

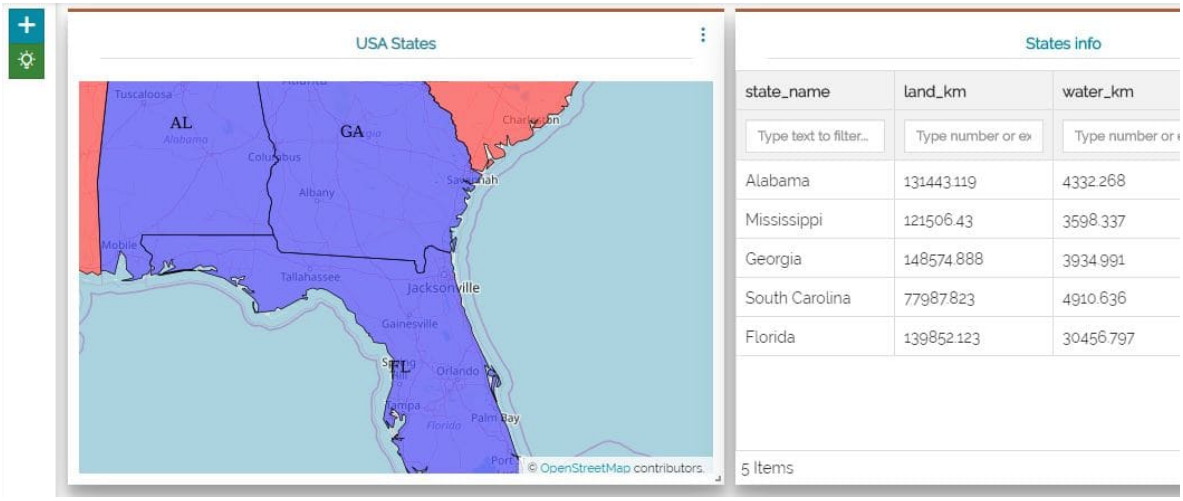
In order to connect Charts, Tables or Counters widget with Maps widget, the procedure is similar to that seen in the previous section. The result is that the

information displayed in the Chart, Table or Counter changes accordingly with the map portion displayed in the connected Map widget. For example the result could be:

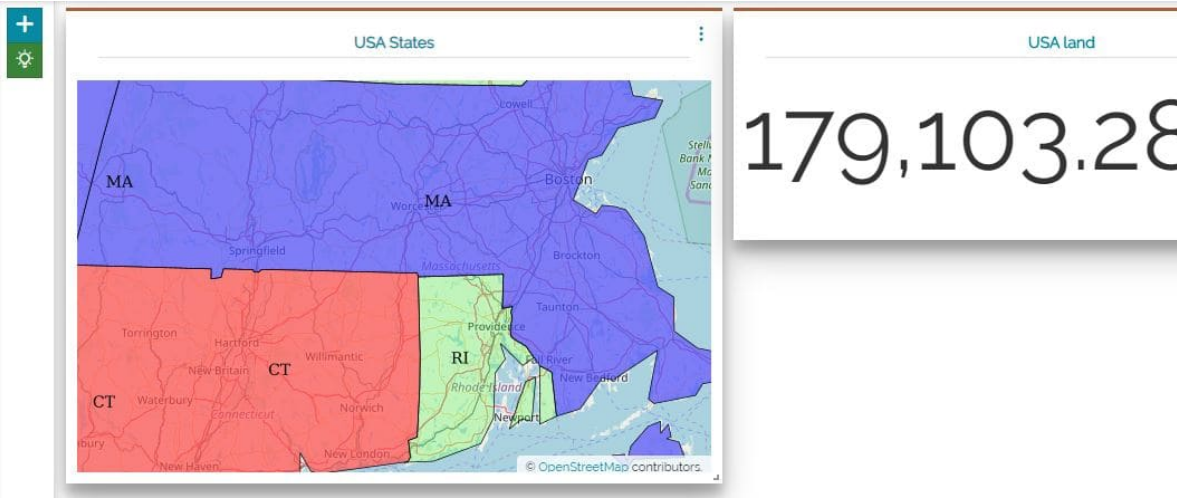
- Connecting Charts with Maps:



- Connecting Tables with Maps:



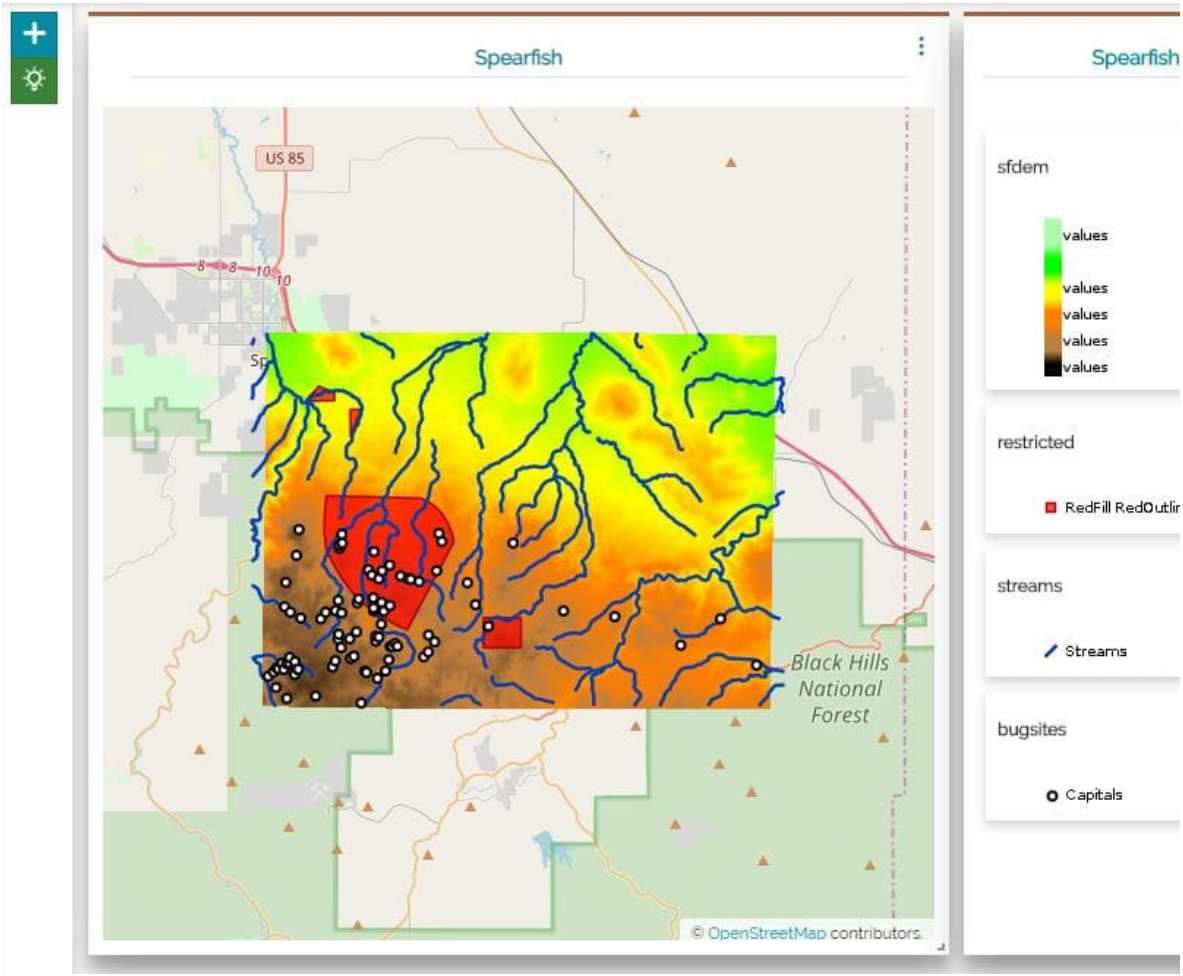
- Connecting Counters with Maps:



When a pan or zoom operation is performed in the Map widget, the other connected widgets are spatially filtered according to the Map viewport.

Maps with Legends

Also in this case the connecting procedure is similar to those seen previously, but now the information contained in the Legend widget doesn't change according with the map extension. An example can be the following:



Connecting Table widgets with other widgets

With the same procedure used for maps (see previous section) the user can connect Table widgets with:

- Map widgets
- Other table widgets, only if it refers to the same layer
- Chart widgets, only if it refers to the same layer
- Counter widgets, only if it refers to the same layer

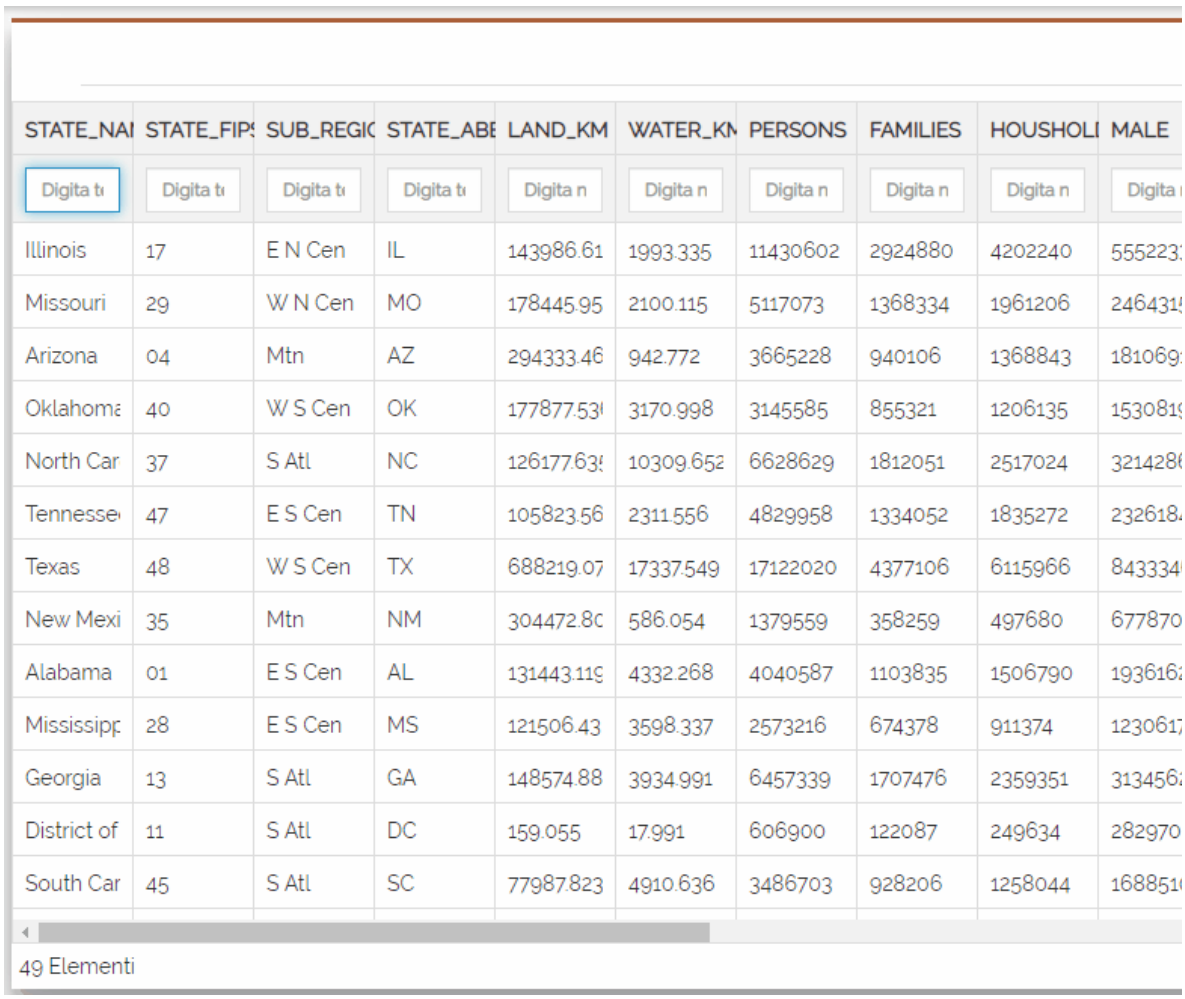
When a table is connected with other widgets, it became a *Parent Table* and a filter appears on the top.

TABLE - USA states

STATE_NAME	STATE_ABBR	LAND_KM	PERSONS	FAMILIES	MALE	FEMALE	WORKERS	CARPOOL	EMPL
Type text to f	Type text to f	Type numbe	Type numbe	Type numbe	Type numbe	Type numbe	Type numbe	Type numbe	Type
Illinois	IL	143986.61	11430602	2924880	5552233	5878369	4199206	652603	54179
Missouri	MO	178445.951	5117073	1368334	2464315	2652758	1861192	312042	23673
Arizona	AZ	294333.462	3665228	940106	1810691	1854537	1358263	239083	16038
Oklahoma	OK	177877.536	3145585	855321	1530819	1614766	1146497	190902	13691
North Carolina	NC	126177.635	6628629	1812051	3214286	3414343	2484061	529828	32384
Tennessee	TN	105823.567	4829958	1334052	2326184	2503774	1777371	321170	22306
Texas	TX	688219.07	17122020	4377106	8433346.0000	8688674	6192585	1142908	76873
New Mexico	NM	304472.805	1379559	358259	677870	701689	491847	87085	57621
Alabama	AL	131443.119	4040587	1103835	1936162	2104425	1455669	267191	17417
Mississippi	MS	121506.43	2573216	674378	1230617	1342599	890741	184019	10287
Georgia	GA	148574.888	6457339	1707476	3134562	3322777	2346478	465734	30815
District of Colu	DC	159.055	606900	122087	282970	323930	229975	36621	30399
South Carolina	SC	77987.823	3486703	928206	1688510	1798193	1268959	277209	16034
Arkansas	AR	134875.075	2350725	661555	1133076	1217649	849722	163025	99478

49 items

It is possible to apply a filter in the *Parent Table* simply by typing a text in the input field present at the top of each column:



STATE_NAI	STATE_FIPS	SUB_REGIC	STATE_ABE	LAND_KM	WATER_KN	PERSONS	FAMILIES	HOUSHOLI	MALE
Digita t	Digita t	Digita t	Digita t	Digita n	Digita n	Digita n	Digita n	Digita n	Digita
Illinois	17	E N Cen	IL	143986.61	1993.335	11430602	2924880	4202240	555223
Missouri	29	W N Cen	MO	178445.95	2100.115	5117073	1368334	1961206	246431
Arizona	04	Mtn	AZ	294333.46	942.772	3665228	940106	1368843	181069
Oklahoma	40	W S Cen	OK	177877.53	3170.998	3145585	855321	1206135	153081
North Car	37	S Atl	NC	126177.63	10309.652	6628629	1812051	2517024	321428
Tennesse	47	E S Cen	TN	105823.56	2311.556	4829958	1334052	1835272	232618
Texas	48	W S Cen	TX	688219.07	17337.549	17122020	4377106	6115966	843334
New Mexi	35	Mtn	NM	304472.80	586.054	1379559	358259	497680	677870
Alabama	01	E S Cen	AL	131443.119	4332.268	4040587	1103835	1506790	193616
Mississipp	28	E S Cen	MS	121506.43	3598.337	2573216	674378	911374	123061
Georgia	13	S Atl	GA	148574.88	3934.991	6457339	1707476	2359351	313456
District of	11	S Atl	DC	159.055	17.991	606900	122087	249634	282970
South Car	45	S Atl	SC	77987.823	4910.636	3486703	928206	1258044	168851

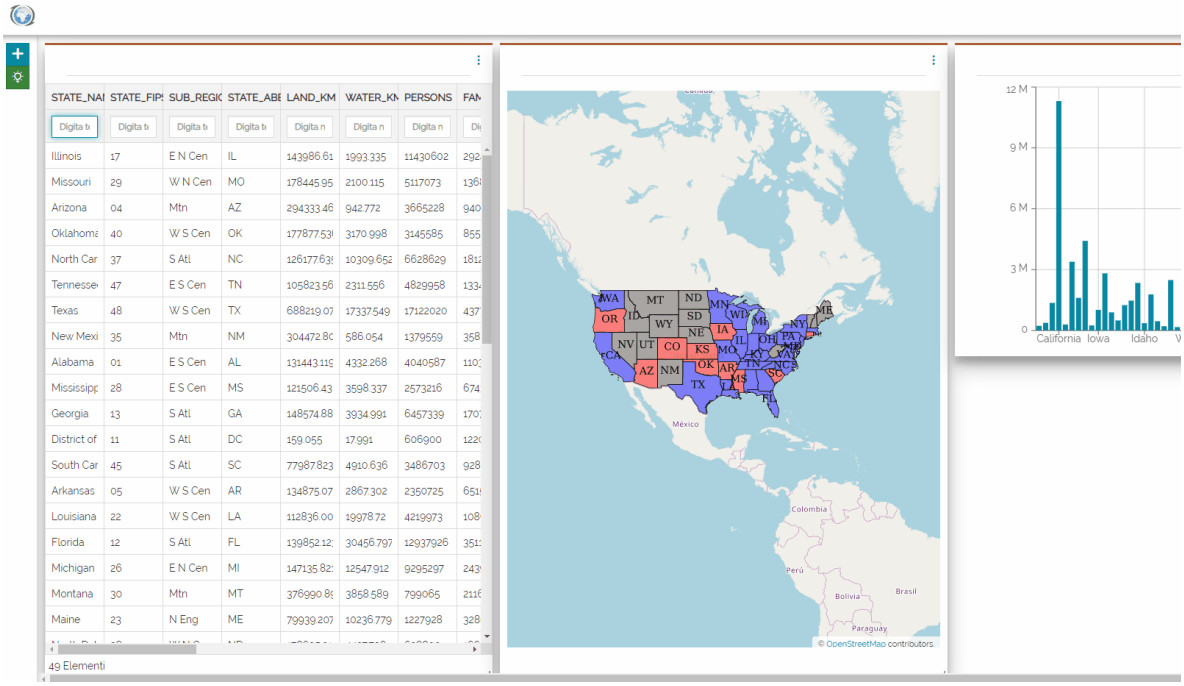
49 Elementi

A Map widget that is connected to a Parent Table receives the alphanumeric filter of the Table and:

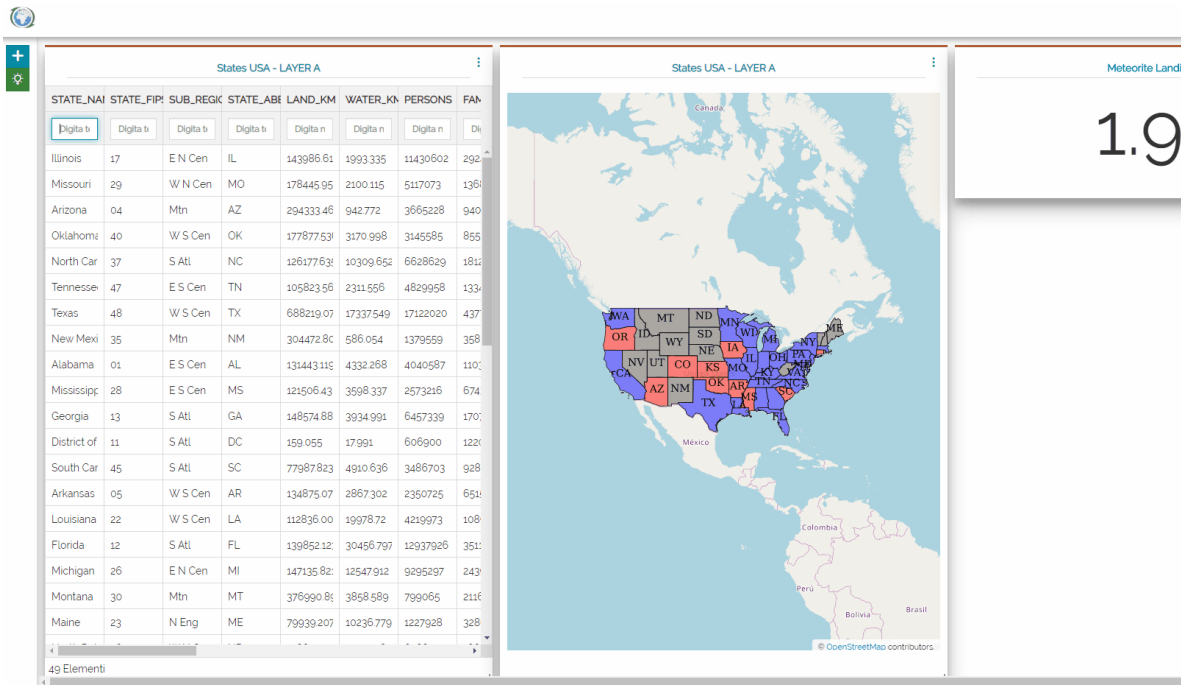
- Performs a zoom to the extent that contains all the Table widget records (the result of the filter in the Table)
- If the Map widget contains the same dataset (layer) of the Parent Table, also the layer on map is filtered accordingly

Once a widget is connected to a map widget that is connected to a Parent Table at the same time:

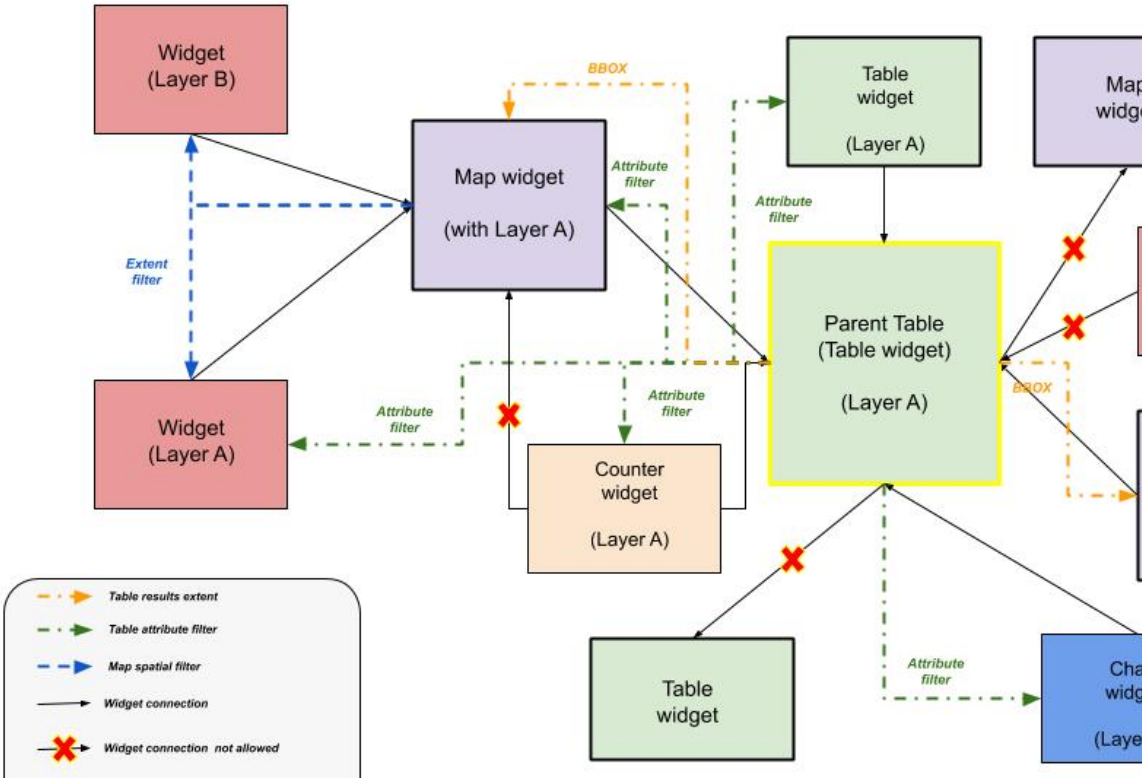
- If the widget has been created on the same dataset (layer) of the Parent Table then two filters will be applied in AND to the widget itself: the spatial filter of the Map widget and the attribute filter defined in the Parent Table



- If the dataset isn't the same, only the spatial filter of the Map widget will be applied as usual: in the following example, the Counter refers to a level other than that configured for the Parent Table

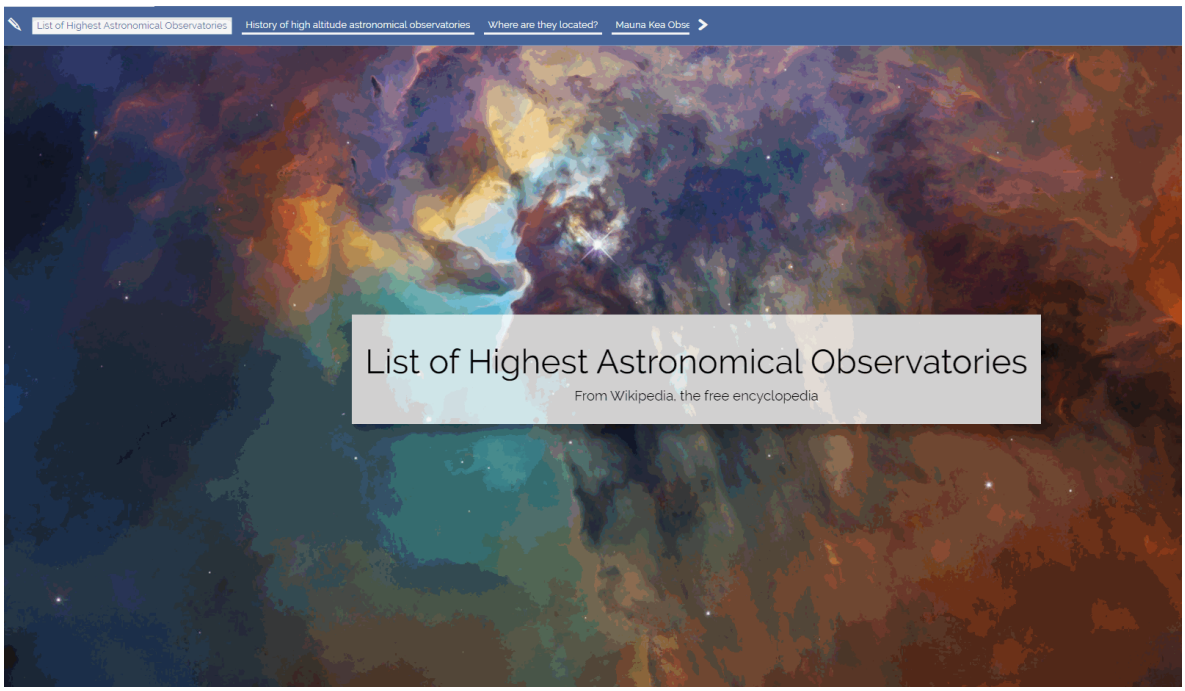


There are different combinations of connections, the image below illustrates the allowed ones by reporting also the kind of filters applied for each case


Widget
(Layer B)
BBOX
Table
widget
(Layer A)
Map
widg
Map widget
(with Layer A)
Attribute
filter
Attribute
filter
Extent
filter
Parent Table
(Table widget)
(Layer A)
Widget
(Layer A)
Attribute
filter
Attribute
filter
BBOX
Counter
widget
(Layer A)
Table results extent
Table attribute filter
Map spatial filter
Widget connection
Widget connection not allowed
Table
widget
Attribute
filter
Cha
widg
(Laye

Exploring Story

In MapStore, GeoStory is a tool that allows to create inspiring and immersive stories by combining text, interactive maps, and other multimedia content like images and video or other third party contents. Through this tool you can simply tell your stories on the web and then publish and share them with different groups of MapStore user or make them public to everyone around the world.



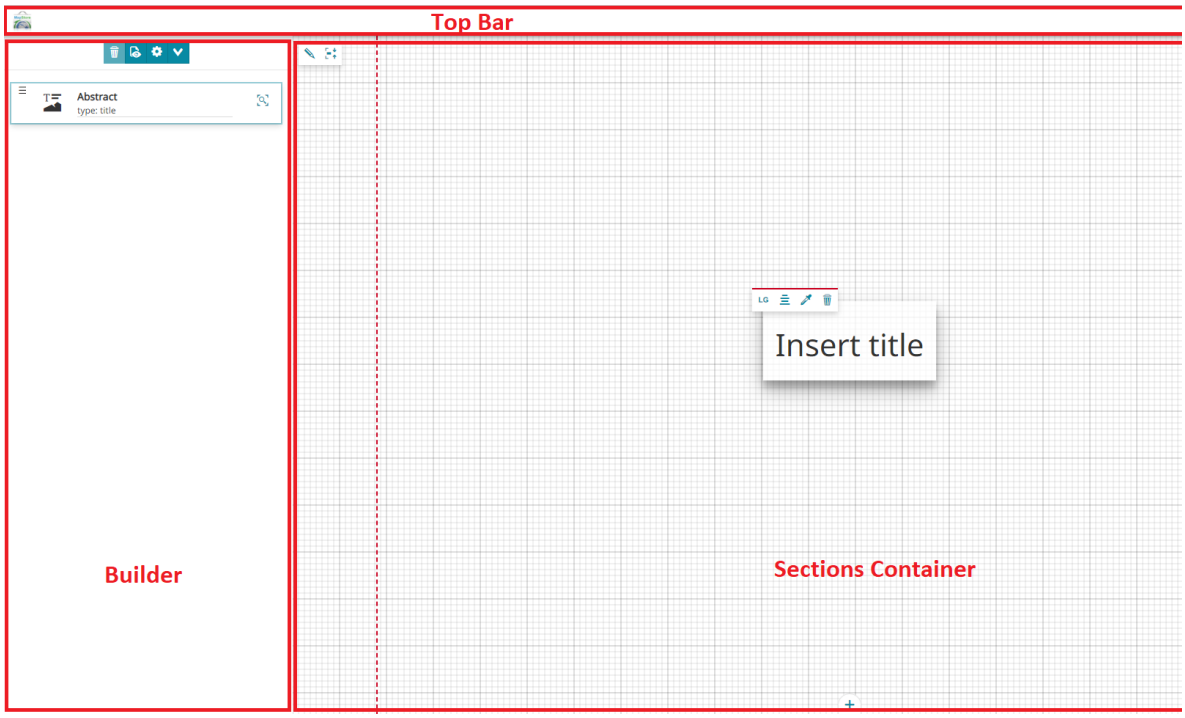
The user can approach a story in two different ways:

- Creating a new story or editing an existing one through the *Edit Mode*
- Enjoy the story and interact with it, through the *View Mode*

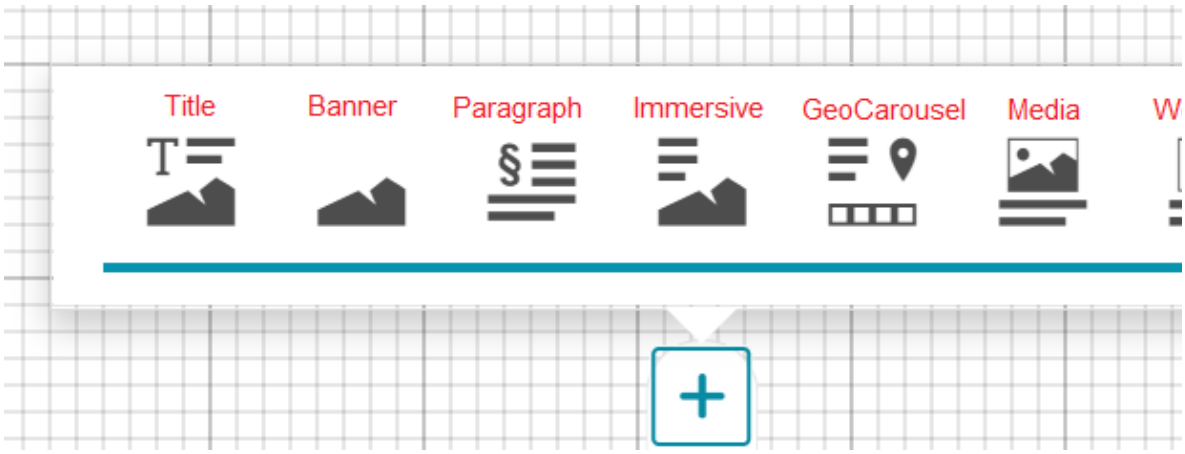
Edit Mode

The *Edit Mode* allows the user to edit a story by adding, removing or modifying the elements inside it. This mode and its tools are used both to edit an existing story and to create a new one.
In order to create a new story, the user can click on the **New GeoStory** button  on MapStore home page. As soon as the user clicks on that button in home

page the story editor opens, it is composed of three main elements: the Topbar, the *Builder* and the *Sections Container* (later simply called *Container*).



The Story content is organized in Sections, that can be added with the + button in the *Container* area. In particular, the user can add to the story the following kind of sections:

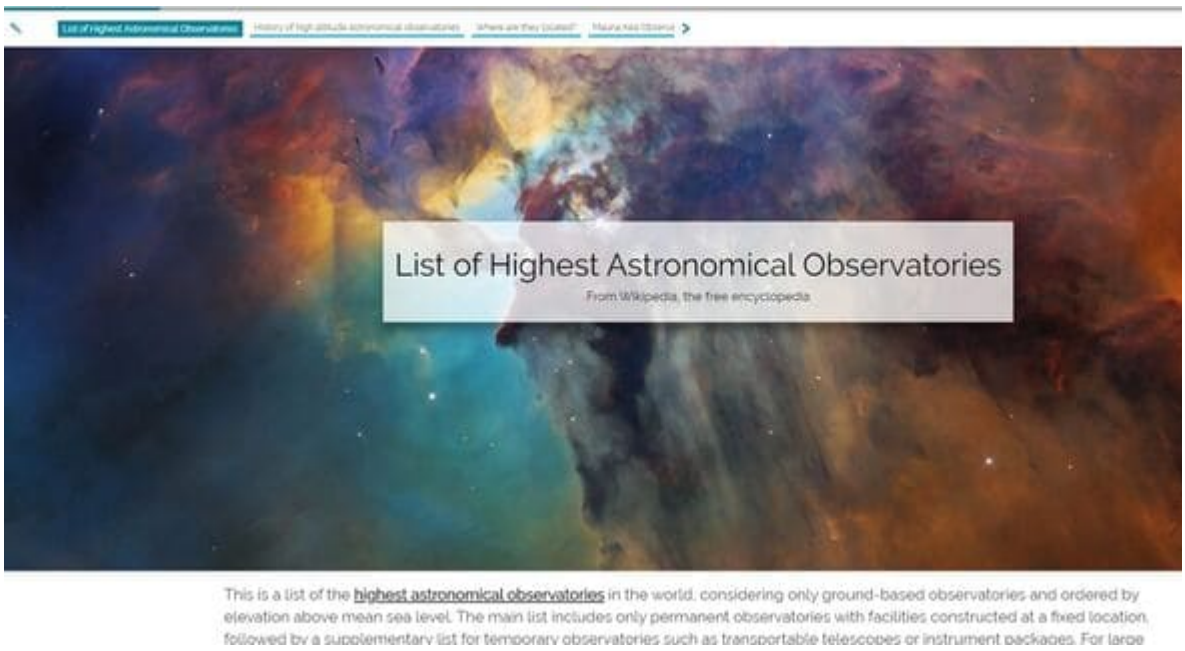


- Title Section
- Banner Section
- Paragraph Section
- Immersive Section
- GeoCarousel Section

- Media Section
- Web Page Section

View Mode

The *View Mode* corresponds to the final result of your story composition that will be visible to end users on the web.
The user can access the *View Mode* also during the Story editing in order to have a preview of its work on the Story itself. The **Show preview** button  in Builder's toolbar allows to do that and the first display looks like the following:



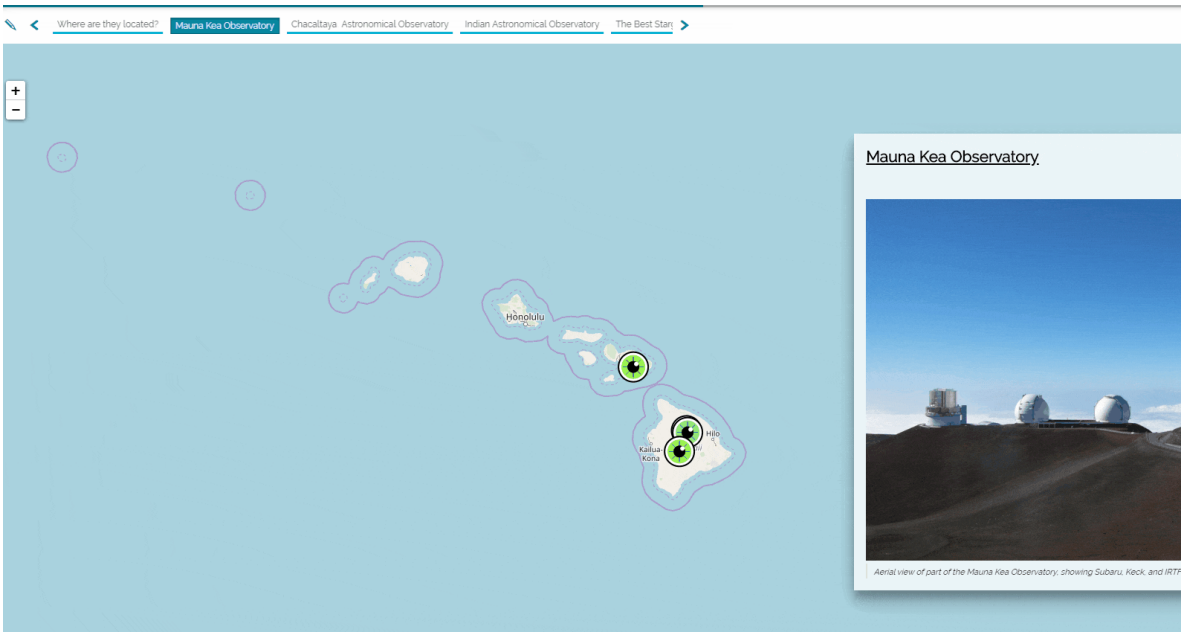
On top of the page there is a **Top bar** in which the story's informations (if properly configured in edit mode) are displayed.

The elements available in the top bar can be the following:

- **Edit Story** button  that allows to switching back to the *Edit Mode*
- **Navigation bar** allows to navigate between different sections of the story
- The **Title** of the story configured in the story settings (edit mode)
- The **Logo** of the story chosen by the story editor in the story settings

> **Warning**
>
> In order to set up this information the story editor needeeds to go back in Edit Mode and open the **Setting** button  as explained in Story Setting.

The Story layout allows to navigate contents by scrolling up and down the Story page by using the mouse or the **Navigation bar**.



Story Settings

The Story Settings panel allows the editor to customize the theme of the story and configure which additional components should appear to end users in the story view.

The Story settings panel is available in Edit Mode and it can be opened by clicking on the **Settings** button  .

Story theme

Default Theme

Background

Text

Font Family — Select...

Overlay

Background

Text

Shadow

Links

Text

Story Header

Title

Enter Title

Font size — Select...

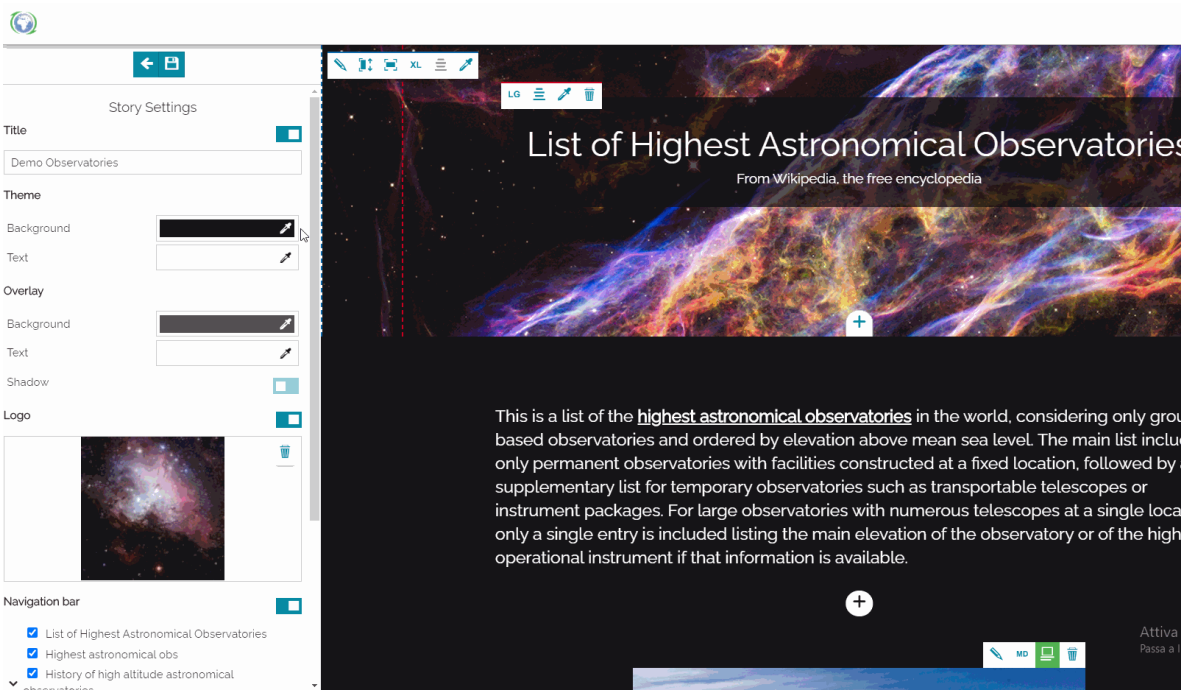
Logo

Navigation bar

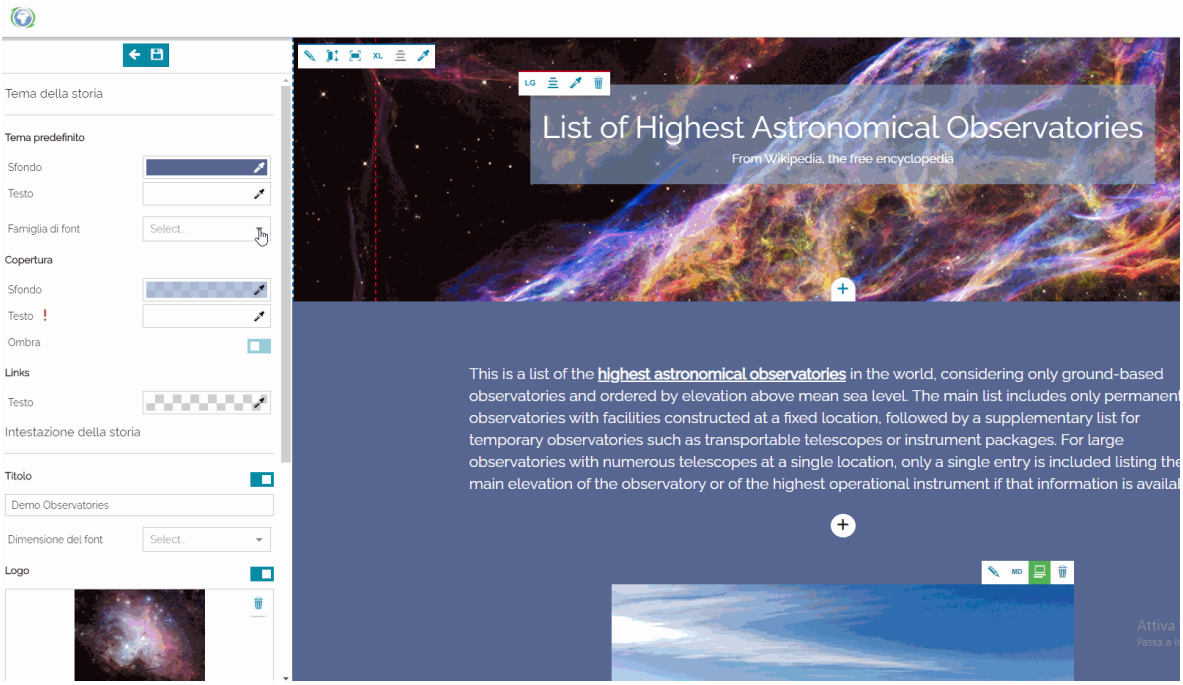
Story Theme

The editor can customize the different components of the story through the following sections:

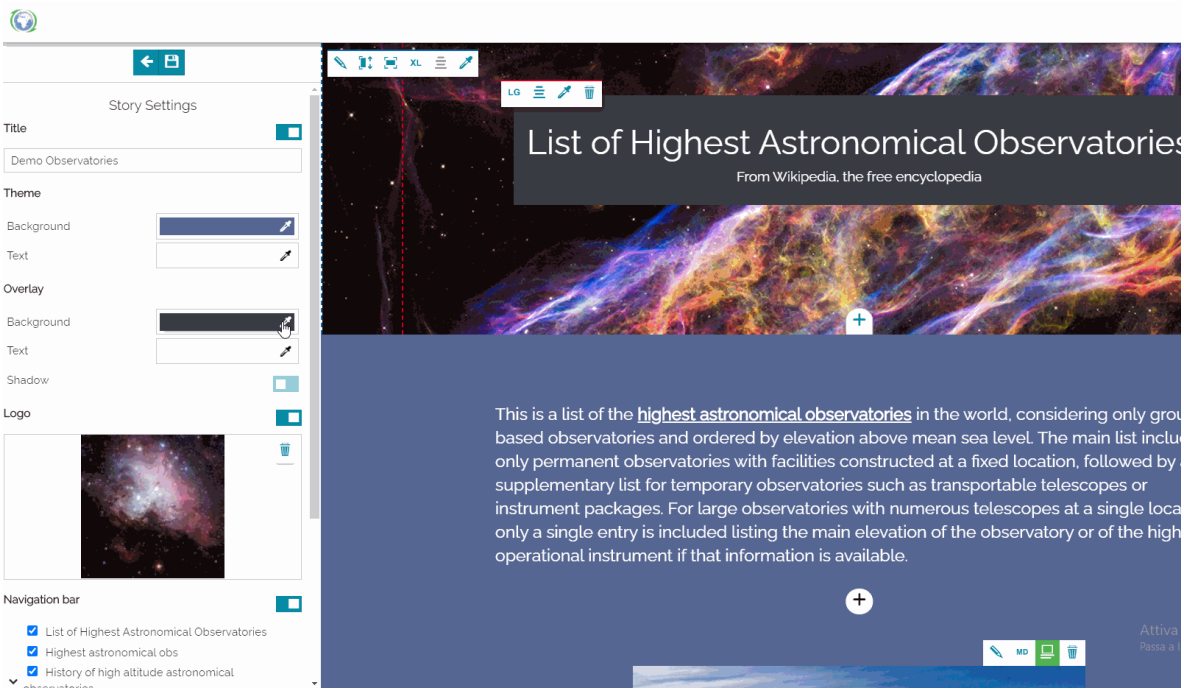
- The **Theme** to choose the default background and text color and the default font of the whole story: clicking on the *Change Color* button a color picker appears to allow selecting the desired color:



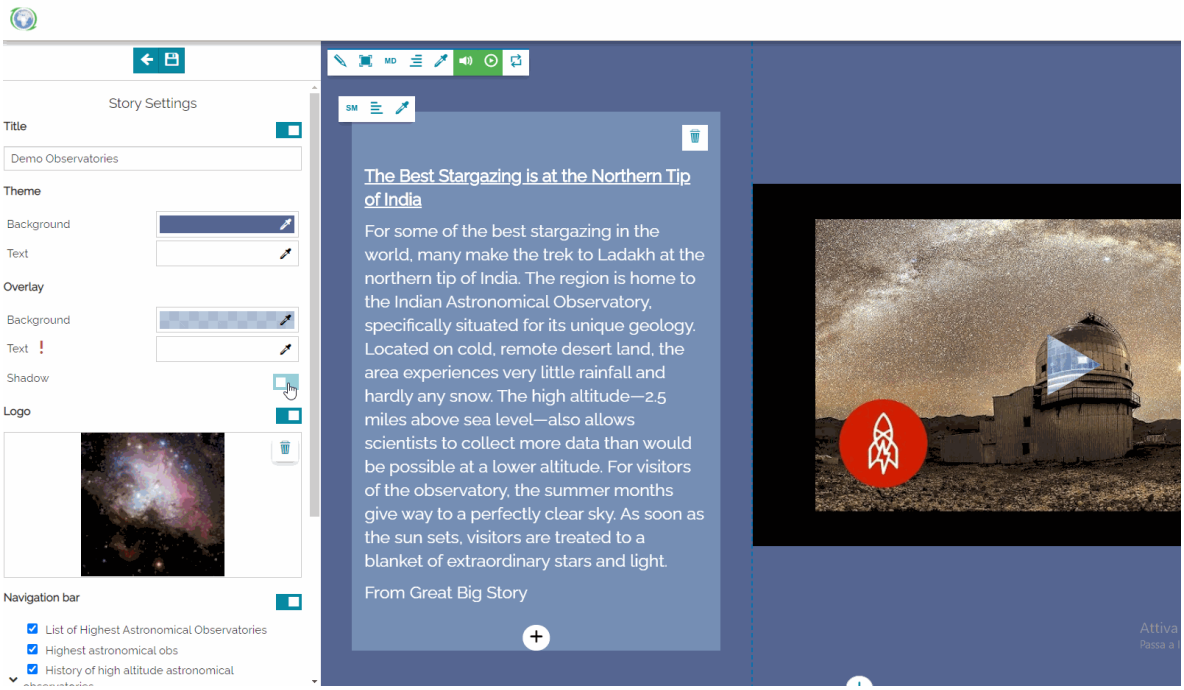
- The *Font Family* to choose the default text font present in the whole story: clicking on the search bar a dropdown menu opens to allow selecting the desired font (*Inherit*, *Arial*, *Georgia*, *Impact*, *Tahoma*, *Time New Roman*, *Verdana*). The default list of Font Families can be customized in the MapStore configuration file.



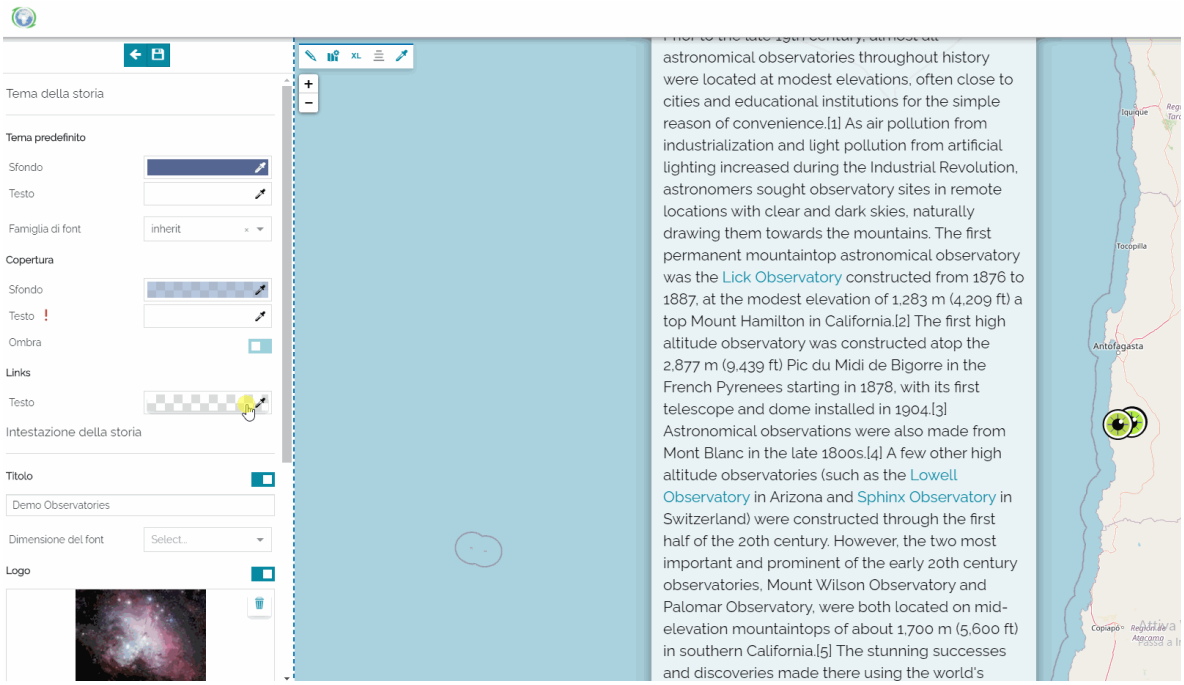
- The **Overlay** to choose the default background and text color of overlay *contents* present in the Title Section and in the Immersive Section as well: clicking on the *Change Color* button a color picker appears to allow selecting the desired color:



- The *Shadow* of overlay *contents* present in the Title Section and in the Immersive Section: to enable or disable the shadow:

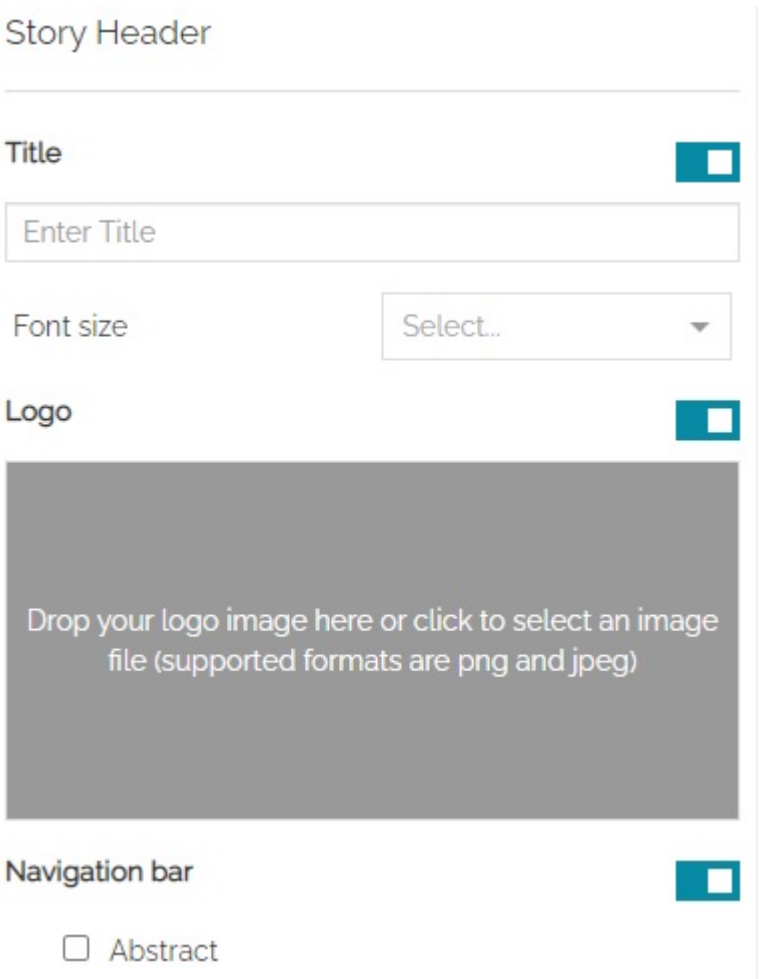


- The color of **Links**. The story editor can choose the default color of the hyperlink that may be present in the *Text Contents*.

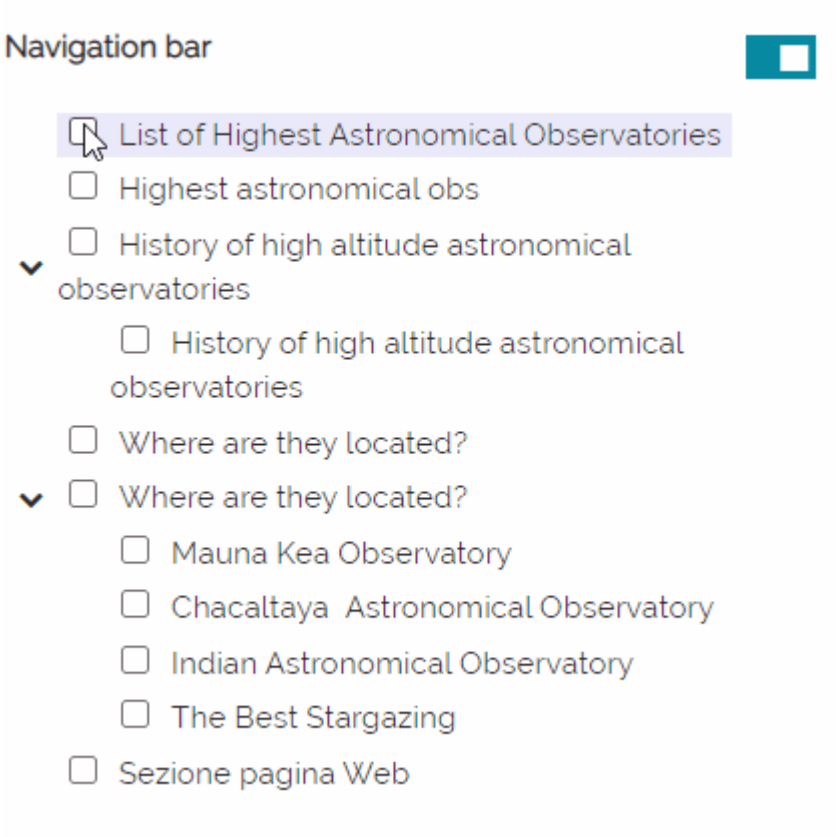


Story Header

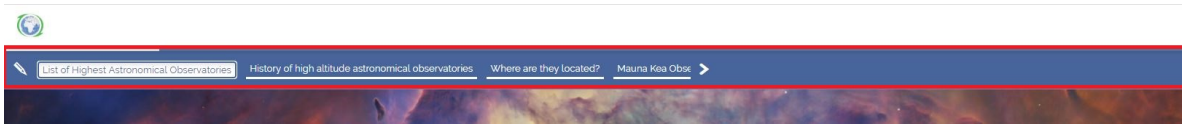
In the top bar of the story, if the editor enables them, the following components are added:



- The **Title** of the story, the default value is the title given to the story's resource in MapStore.
- The **Font Size** of the title: clicking on the search bar a dropdown menu opens to allow selecting the desired size (*14 px*, *16 px*, *18 px*, *24 px*, *28 px*).
- The story **Logo**, that can be for example an image that represents your organization or something connected to the story itself.
- The **Navigation Toolbar** to improve the story navigation for end users. Each section of the story is reported in a tree and the editor can establish which section should appear in the toolbar to allow end users to quickly navigate the story.



Saving the story settings and going back in View Mode, the top bar looks like this:



Title Section

As soon as you create a new Story, by default, only a Title Section (the cover) is present in the workspace. In this section the story editor can customize two different elements: the title text and the cover's background.

Content

By default, the title section has the following placeholder with an empty background behind it:

With a click inside the text area, it expands and the Text Editor Toolbar appears allowing the story editor to type and/or edit the title text:

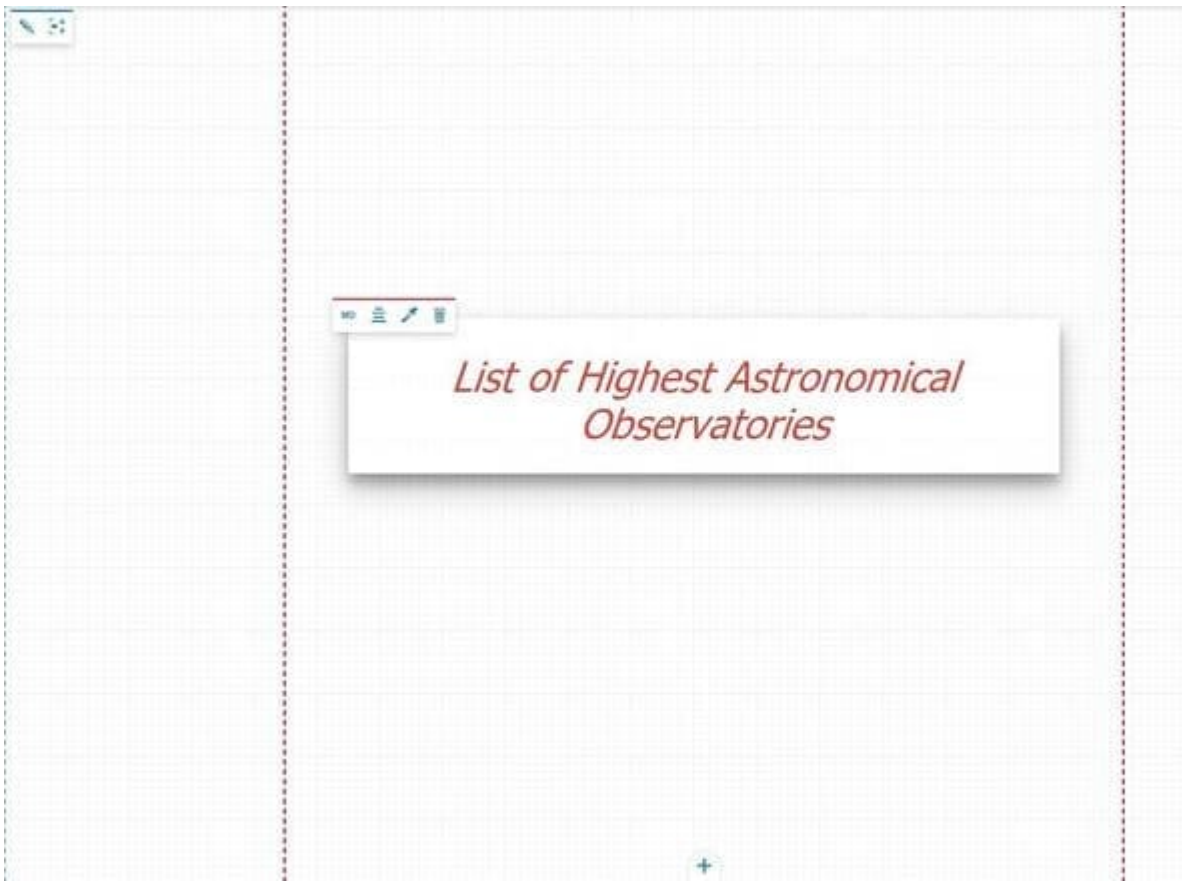
Once the text has been written, it is possible to configure the text area position and its style from the component's toolbar:

The text window toolbar allows the user to change the following settings:

- The **Change size** button MD allows to change the size of the text window in *Small*, *Medium*, *Large* or *Full*.
- The **Align content** button allows to align the text window, inside the Container, on the *Left*, *Center* or *Right*.
- The **Change field theme** button allows to change the text window theme in *Default* (same default theme settings of the story, see Story Settings), *Bright*, *Dark* or *Custom* (allows to customize background and text colors and enable or disable the shadow)
- The **Remove** button allows delete the title section.

> **Note**
>
> When a section has only one content, and the story editor remove that content, the entire section will be also automatically deleted.

Setting a title with *Large* size, aligned on the *Center* of the Container and with a *Bright* theme, the result is something like:



Background

For a Title sections it is possible to customize the background through the background editing toolbar:

In case of an empty background, the background editing toolbar allows to:

- Add a media content as a background, with the **Change media source** button that opens the Media Editor
- Change the height of the section through the **Fit/adapt content** button

It is possible to add three types of media contents as a background: images, videos or maps.

Images

Once an image is added for the background, the result is something like this:

In this case the background editing toolbar allows to customize the image background through the following settings:

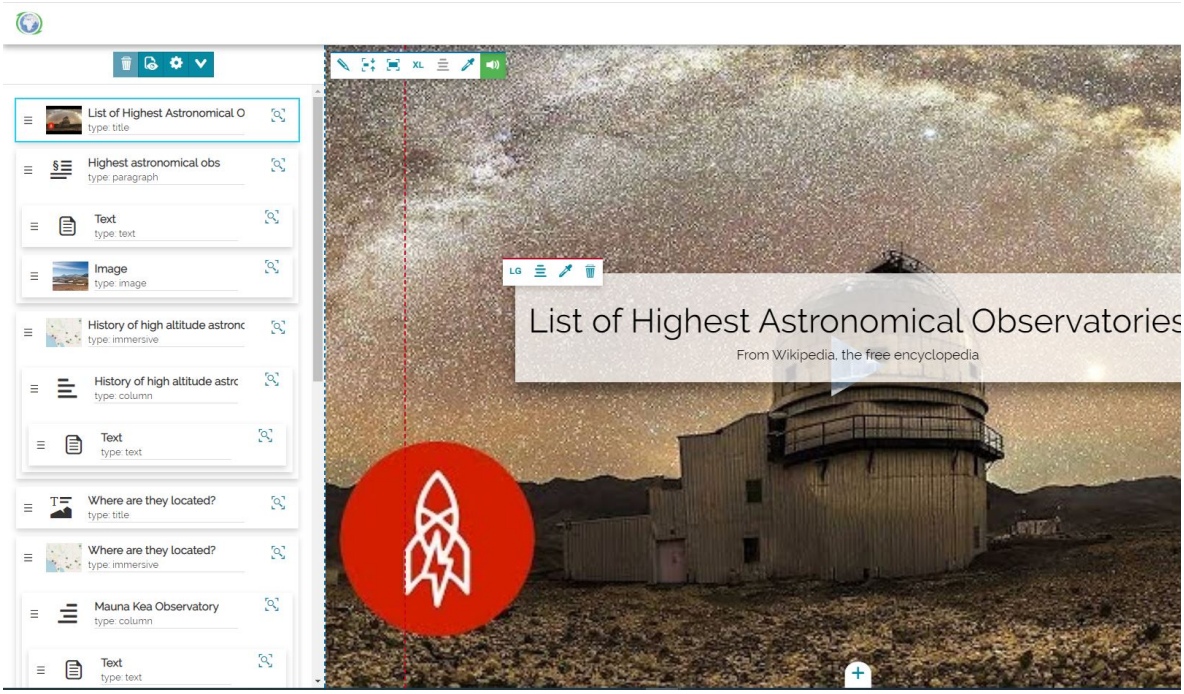
- **Change media source** allows to select the media content to use for the section, clicking on this button the Media Editor opens
- Change the content height through the **Fit/adapt content** button

- Change the relation image/container, choosing between making the background cover the whole container or making the whole background visible inside the container
- **Change size** XL between *Small*, *Medium*, *Large* or *Full*
- **Align content** on the *Left*, *Center* or *Right*
- **Change the background theme** to set the colour of the empty background between *Default* (same default theme settings of the story, see Story Settings), *Bright*, *Dark* or *Custom* (allows to customize the color of the background).

> **Warning**
>
> The *Align content* and the *Change field theme* buttons are disabled if the image size is full screen

Videos

Once a video is added for the background, the result is something like this:



The background toolbar, in this case, changes a little bit by including an additional button:

- The **Audio**, enabled by default, through which you can enable or mute the video.

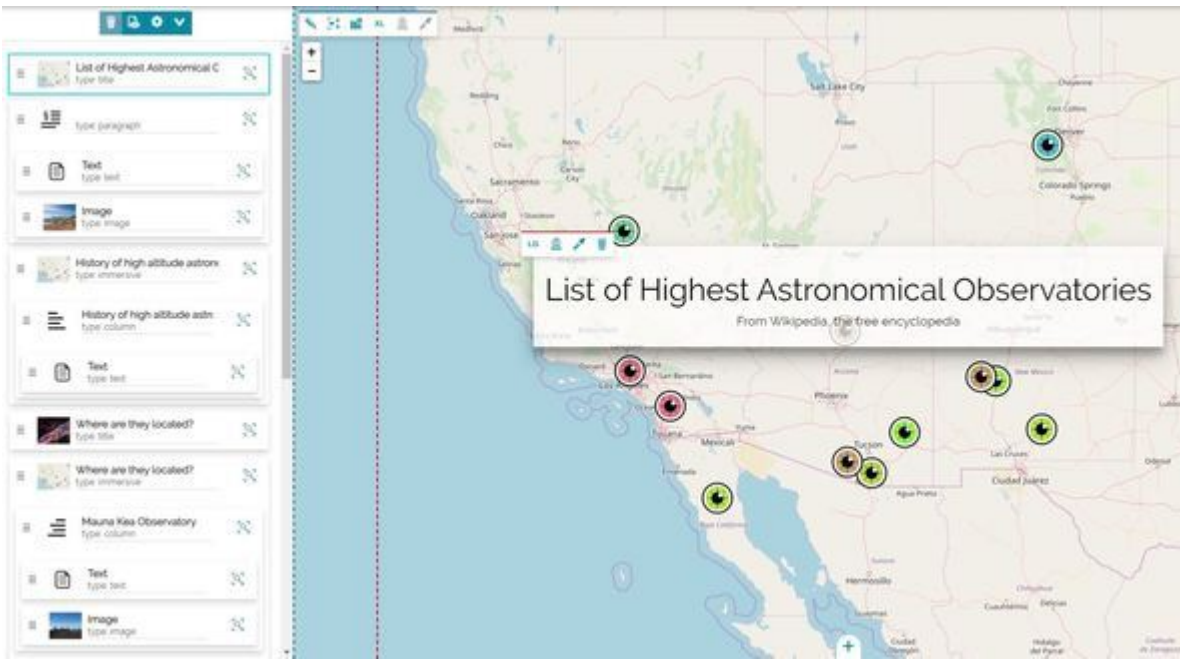
By clicking on the *Make the whole background visible inside the container* button other two buttons appear to perform the following operations:

- **Enable Autoplay** to play the video automatically once the user is on it
- **Enable Loop** to continuously repeat the video

> **Note**
>
> The video play will be available only in *View Mode* of the story: it is not available in edit mode except in the media editor as a preview.

Maps

In this case, adding a map as background, the result will be like this:

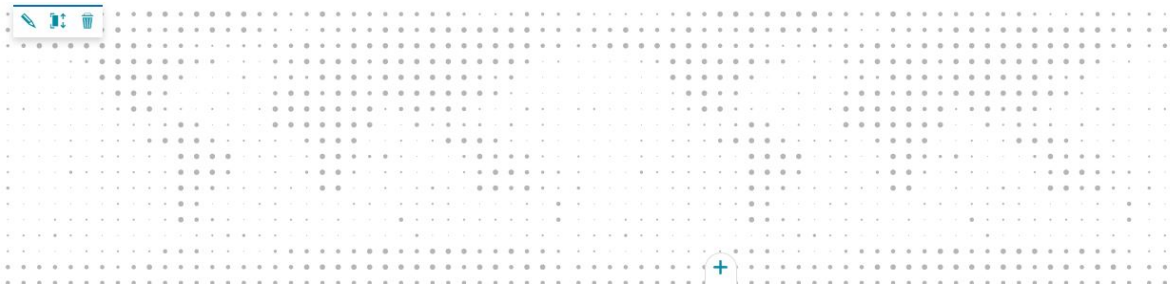


The background toolbar, in this case, changes a little bit by including an additional button:

- The **Edit map configuration** through which it is possible to Configure the Map

Banner Section

The Banner Section is similar to the Title Section and it is useful to easily create a story banner without the title text content.

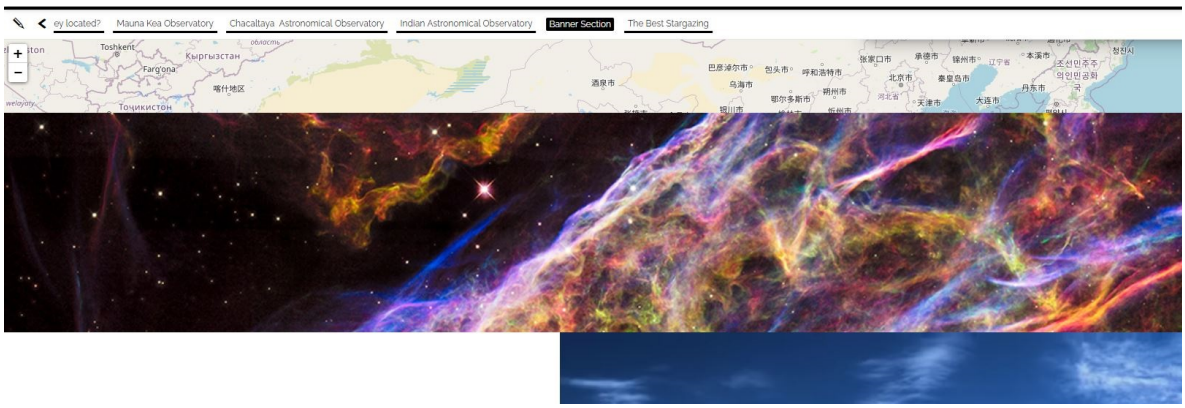


From the background editing toolbar the user can do the following actions:

- Add a media content as a background opening the Media Editor through the **Change media source** button and choose between images, videos or maps.
- Change the height of the section through the **Fit/adapt content** button
- Remove the banner section through the **Remove** button

Once the media content has been added as a background of the section, the editing toolbar changes to enable different functionality depending on the content inserted: as explained here.

Below is an example of an image added as background in the Banner Section:



Paragraph Section

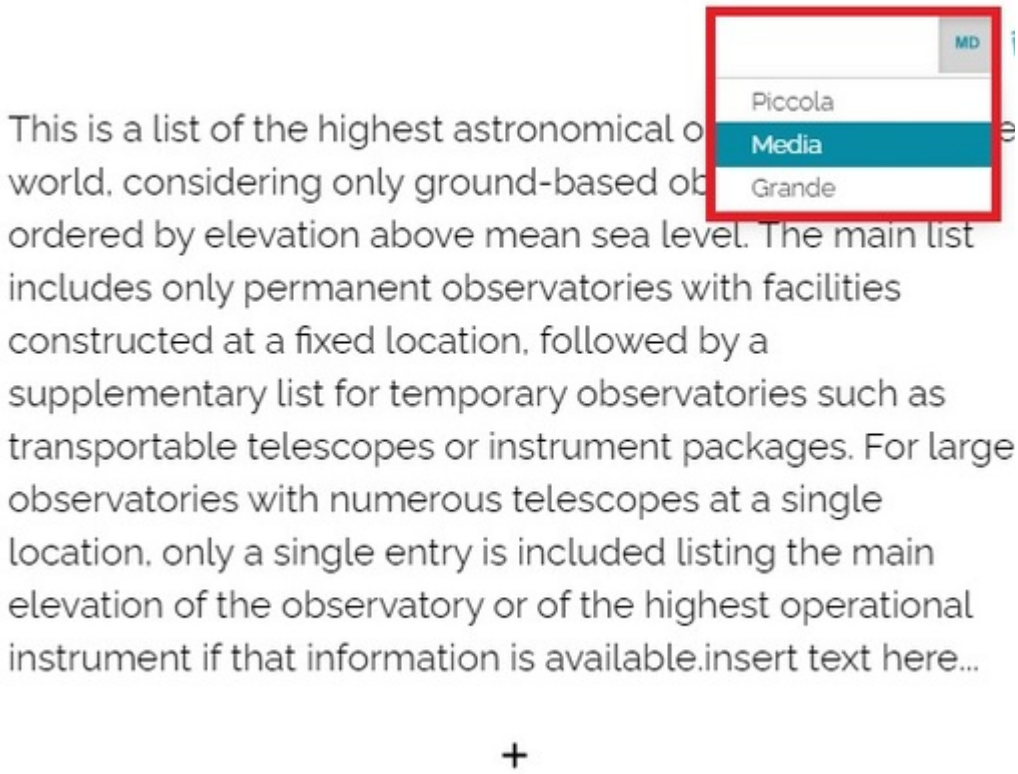
The Paragraph Section allows to insert a textual content to the story. The story editor can also click on the + button to add additional contents to this section (like media, other paragraphs or embed third party contens). It is possible to choose between:

- **Text Content** to add another text content just below the current one
- **Media Content** to open the Media Editor to add an image, a map or a video.
- **Web Page Content** </> to add an external web page

Text Content

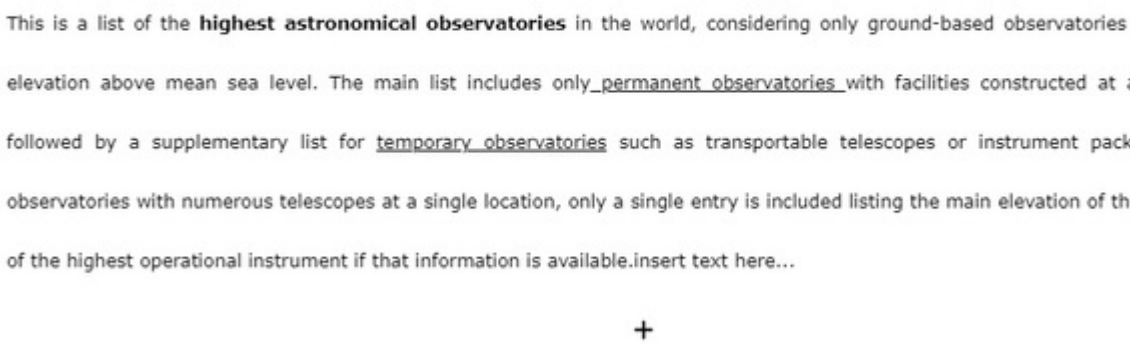
By default, as soon as a Paragraph is added, an empty text content is already present as a placeholder and the content toolbar allows to:

- Change the size of the text content: clicking on the **Change Size** button ↔ a dropdown menu appears to allow selecting between *Small*, *Medium* or *Full* size:



- Delete the *Paragraph Section* through the **Remove** button

The editor can write a text by clicking on the text content and customize it through the Text Editor Toolbar. A possible result of adding and formatting the text can be the following:

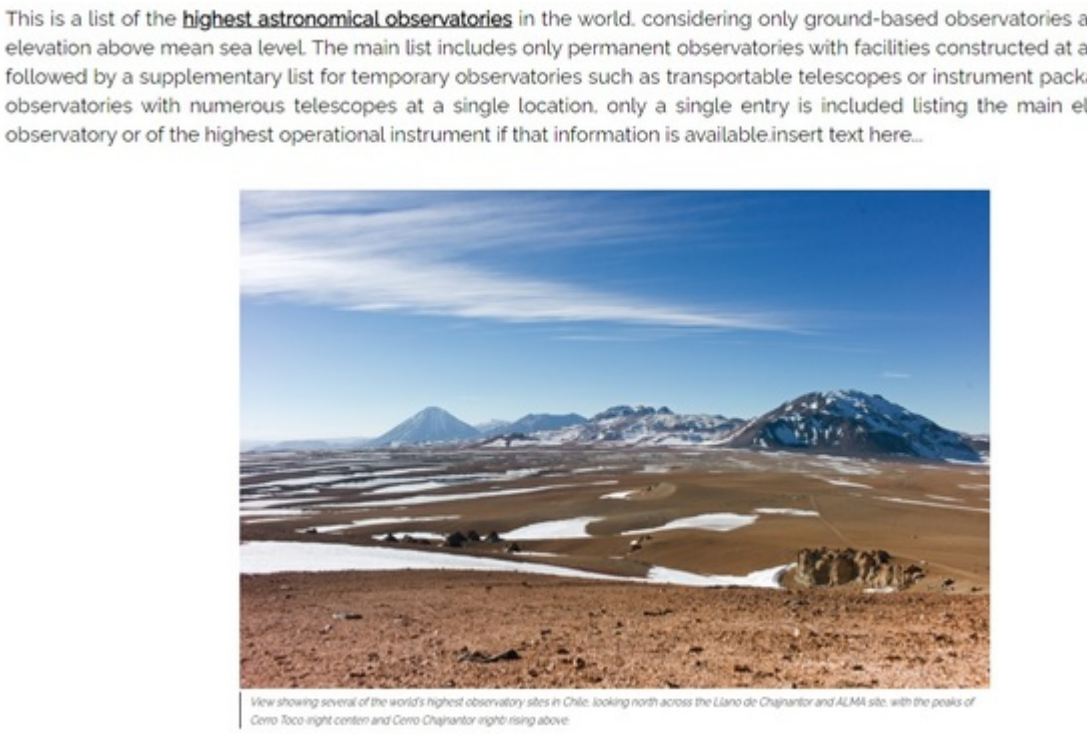


Media Content

Adding a media content, the Media Editor opens to allow adding the supported media (like Image, Map or Video).

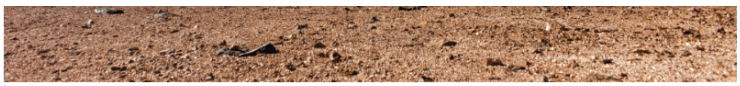
Images

An image added inside the paragraph section can be customized through the Image Content Toolbar. Below is an example of a small, center-aligned image, just below a text content:



Videos

A video added inside the paragraph section can be customized through the Video Content Toolbar. Below is an example of a video, just below an image content:



View showing several of the world's highest observatory sites in Chile, looking north across the Llano de Chajnantor and ALMA site, with the peaks of Cerro Toco (right center) and Cerro Chajnantor (right) rising above.

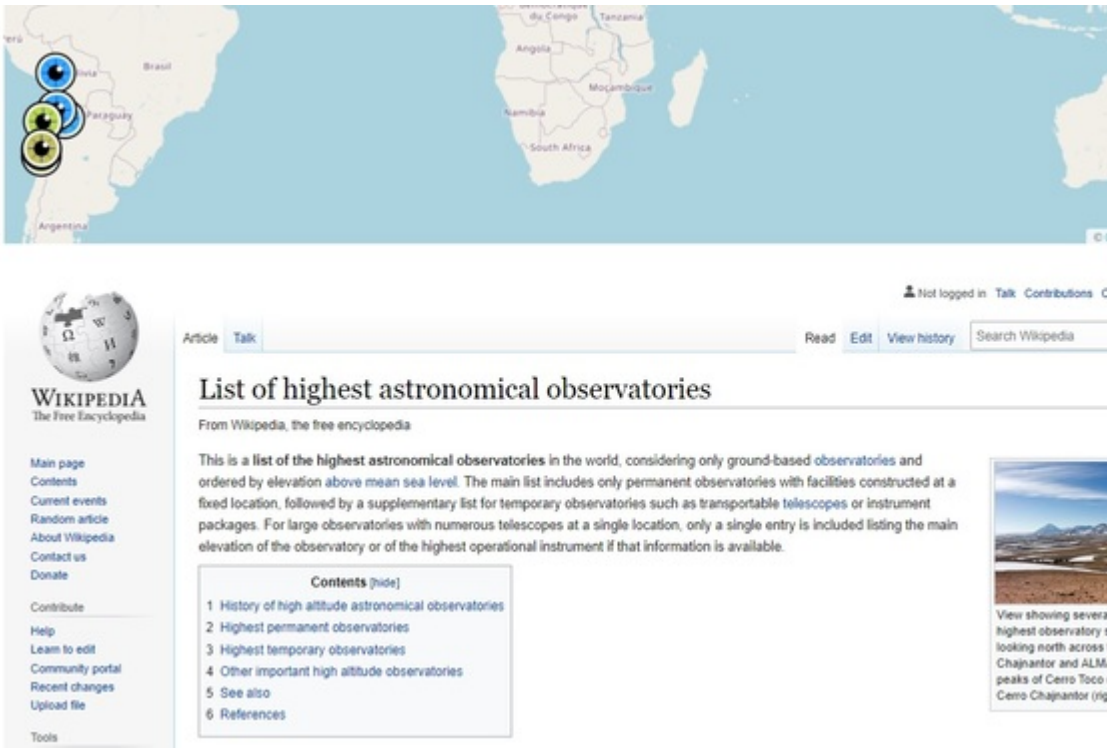
Maps

A map added inside the paragraph section can be customized through the Map Content Toolbar. Below is an example of a large, center-aligned map, just below an Image content:

View showing several of the world's highest observatory sites in Chile, looking north across the Llano de Chajnantor and ALMA site, with the peaks of Cerro Toco (right center) and Cerro Chajnantor (right) rising above.

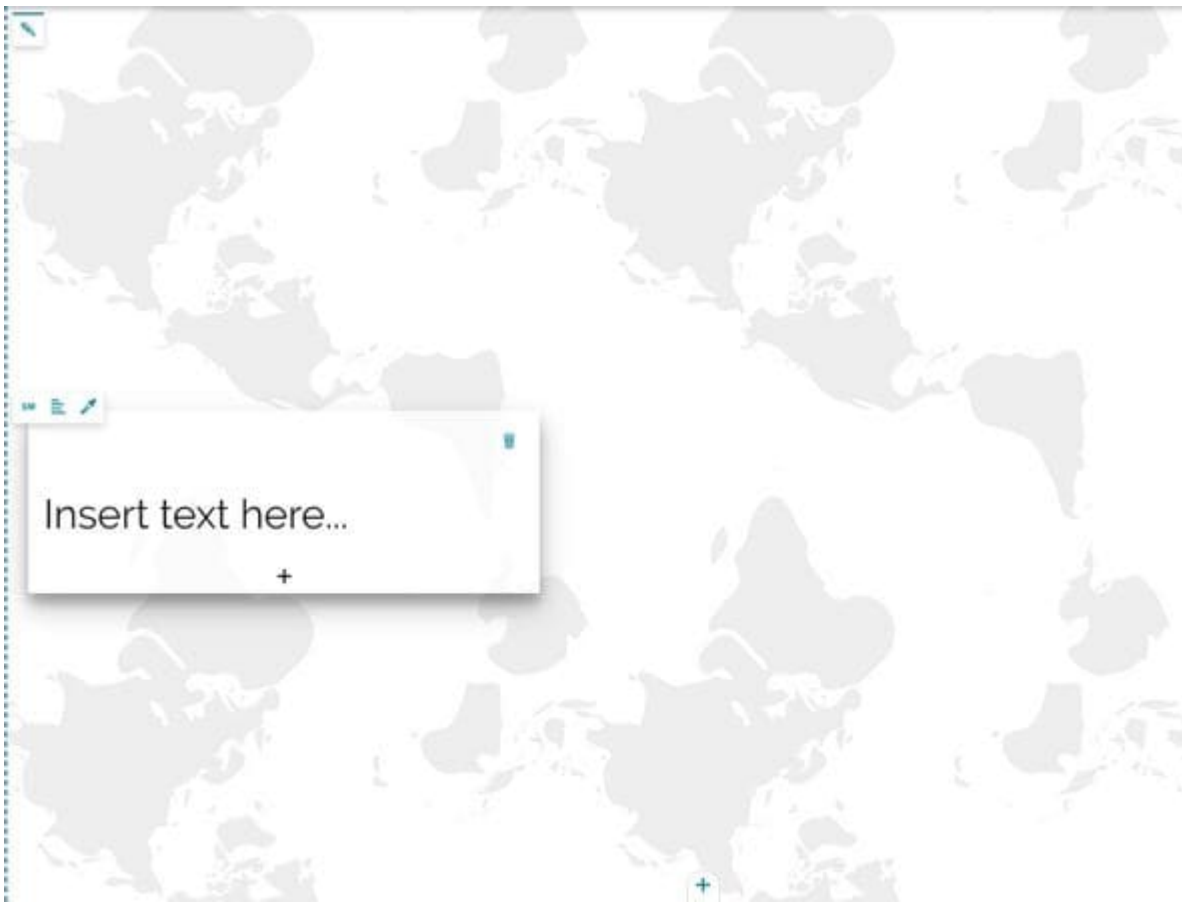
Web Page Content

Adding a web page content, the Web Page Windows opens allowing the user to add the URL of an extenal web page. A web page added inside the paragraph section can be customized through the Web Page Content Toolbar. Below is an example of a medium, center-aligned web content, just below a map one:



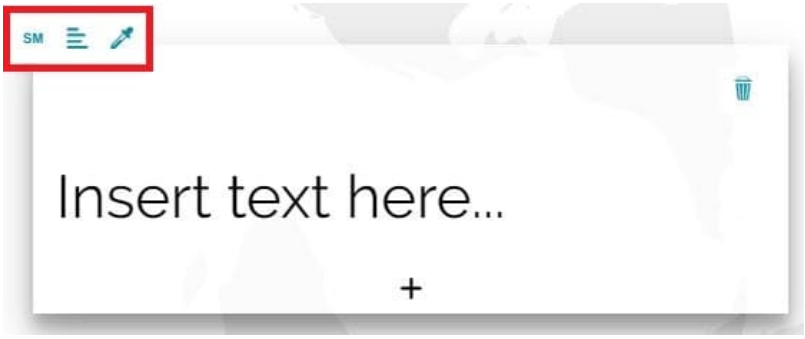
Immersive Section

The immersive section is composed of two elements: the background and the immersive content. As soon as you add an immersive section to your story, an empty background with an empty text content will be displayed.



Content

Inside an *Immersive Section* the story editor can customize the content area through the **Immersive Content Toolbar**:

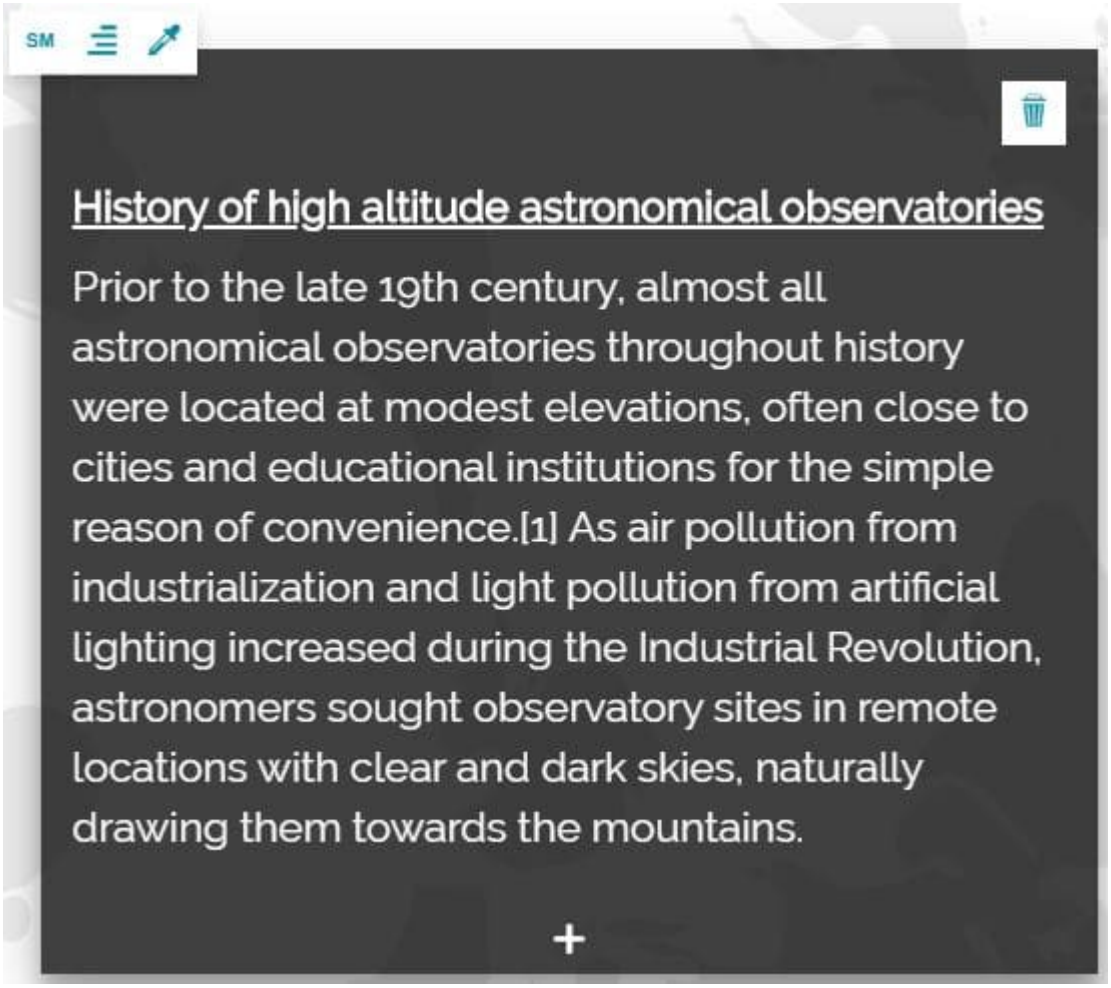


In particular, it possible to:

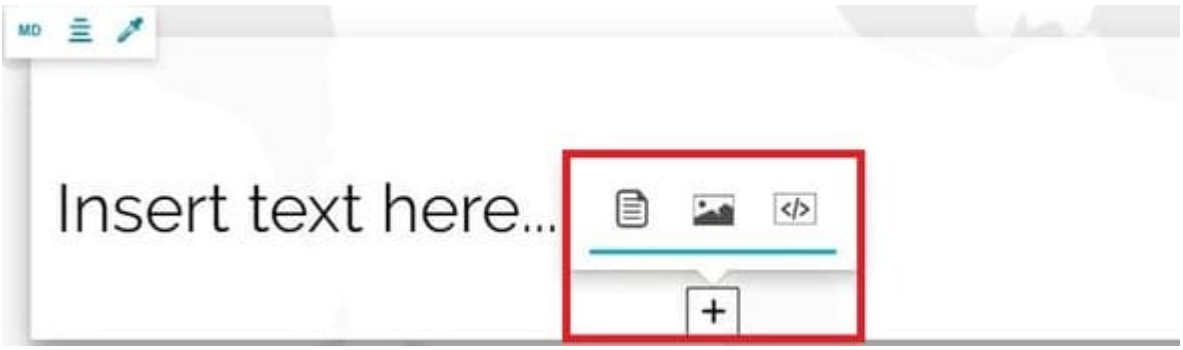
- The **Change size** button MD allows to change the size of the text window in *Small*, *Medium*, *Large* or *Full*.
- The **Align content** button allows to align the text window, inside the Container, on the *Left*, *Center* or *Right*.
- The **Change field theme** button allows to change the text window theme in *Default* (same default theme settings of the story, see Story Settings), *Bright*, *Dark* or *Custom* (allows to customize background and text colors and enable or disable the shadow)

Below is an example of a small Immersive Content, aligned to the *Right* and with a *Dark* field theme:

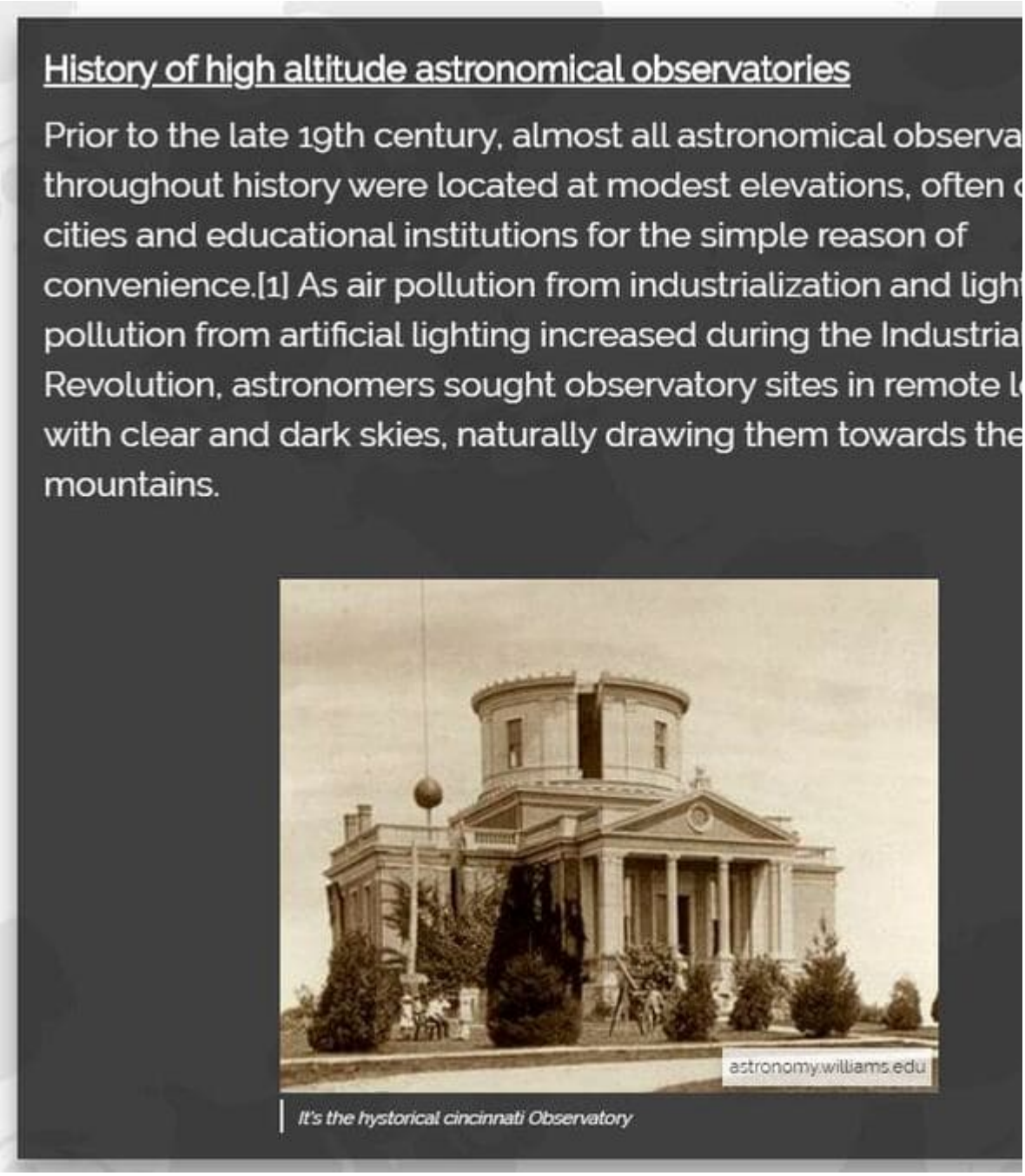
As soon as you add a text content, it appears available just below the current one. With a simple click inside it, the user can write the text and customize the text formatting through the Text Editor Toolbar. An example of a text content can be the following:



The immersive content can include text, media contents or web pages. A new content can be added inside the immersive content column through the + button, or it can be removed through the 🗑 button.

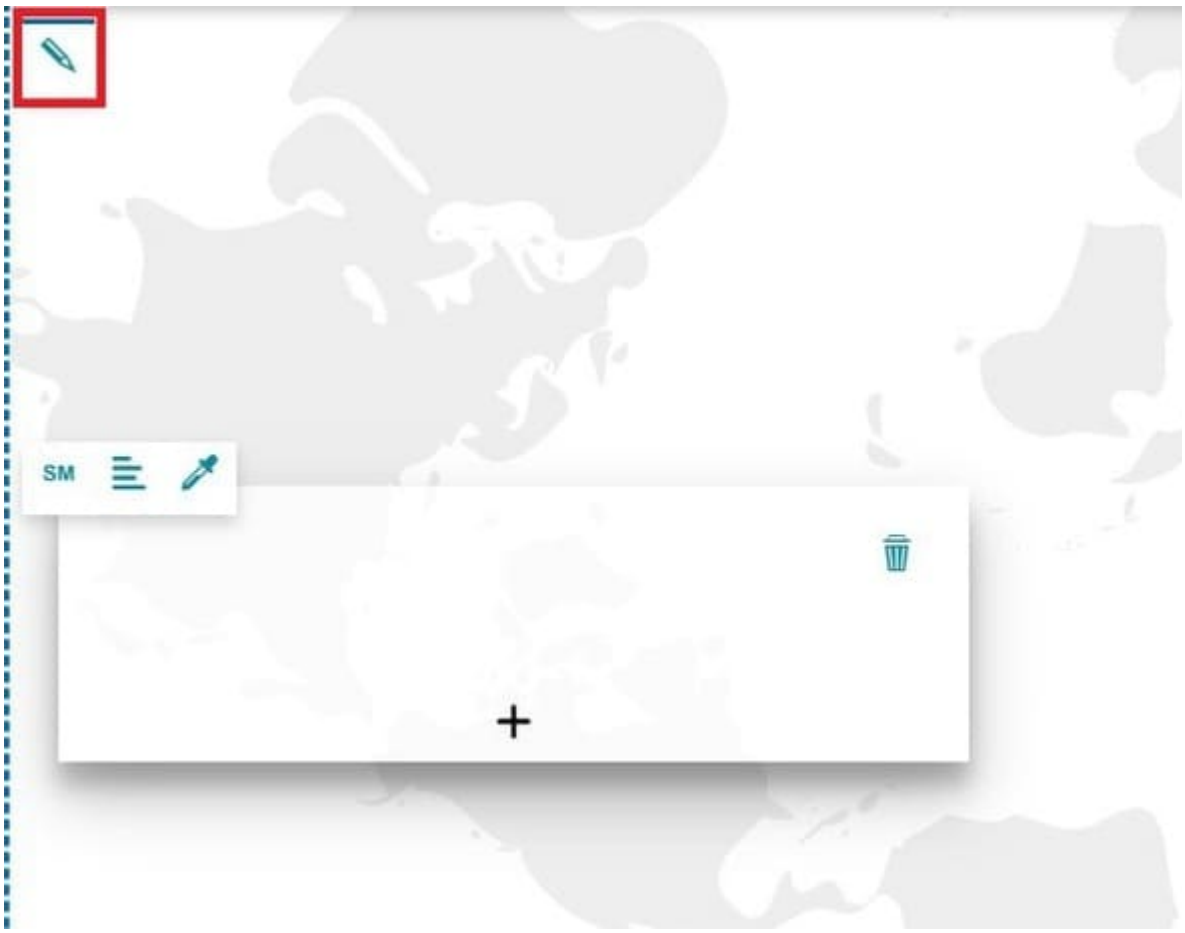


Adding a media content, the Media Editor appears to allow the story editor to add an Image, a Map or a Video. It is also possible to add a Web Page content as it is explained in the Web Page Section. An example of immersive content with a text and an image can be the following:



Background

For Immersive sections, it is possible to customize the background through the background editing toolbar:



The background editing toolbar, when no media are applied, allows to:

- Add a media as a background of the section, with the **Change media source** button that opens the Media Editor

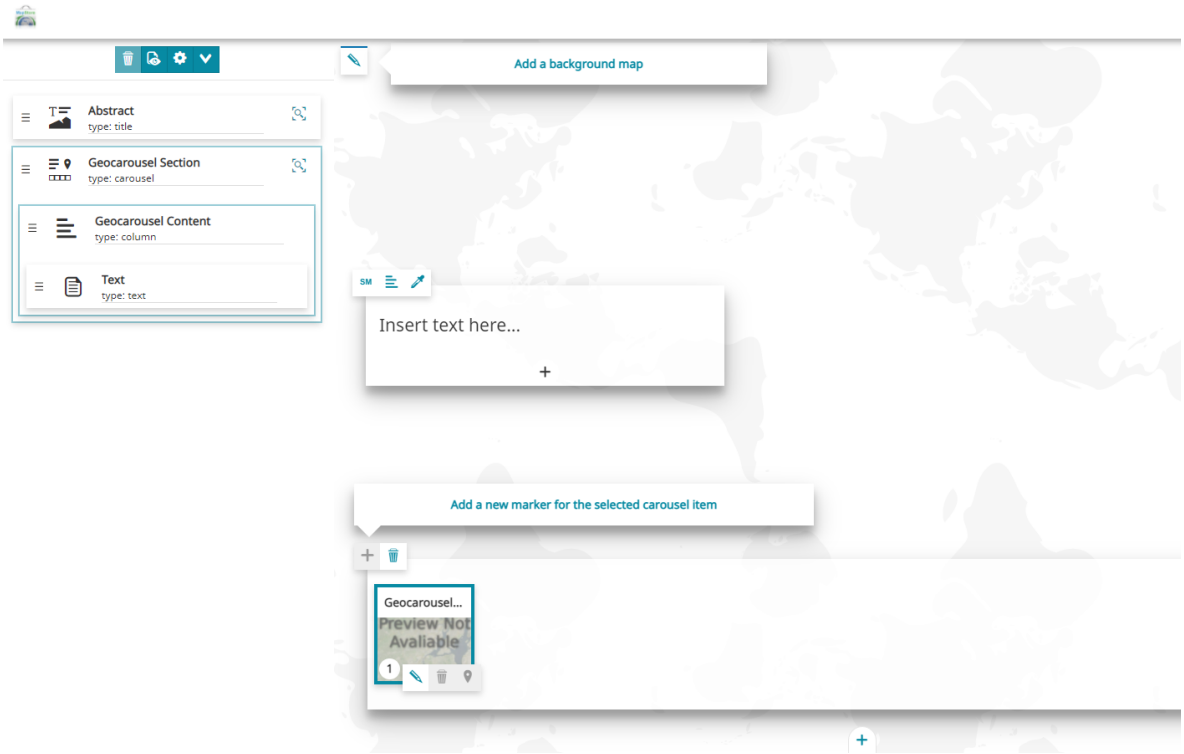
Once a media (*image*, *video* or *map*) is added to the background, an editing toolbar appears in the upper left corner of the section allowing the user to manage the background content.

The **Background editing toolbar** changes depending on the type of media added to the background, as it is explained in the Background section.

> **Note**
>
> Only for Immersive Section, when the user try to add another section of the same type just below the current one, the added section is actually another immersive content, that fits inside the same immersive section.

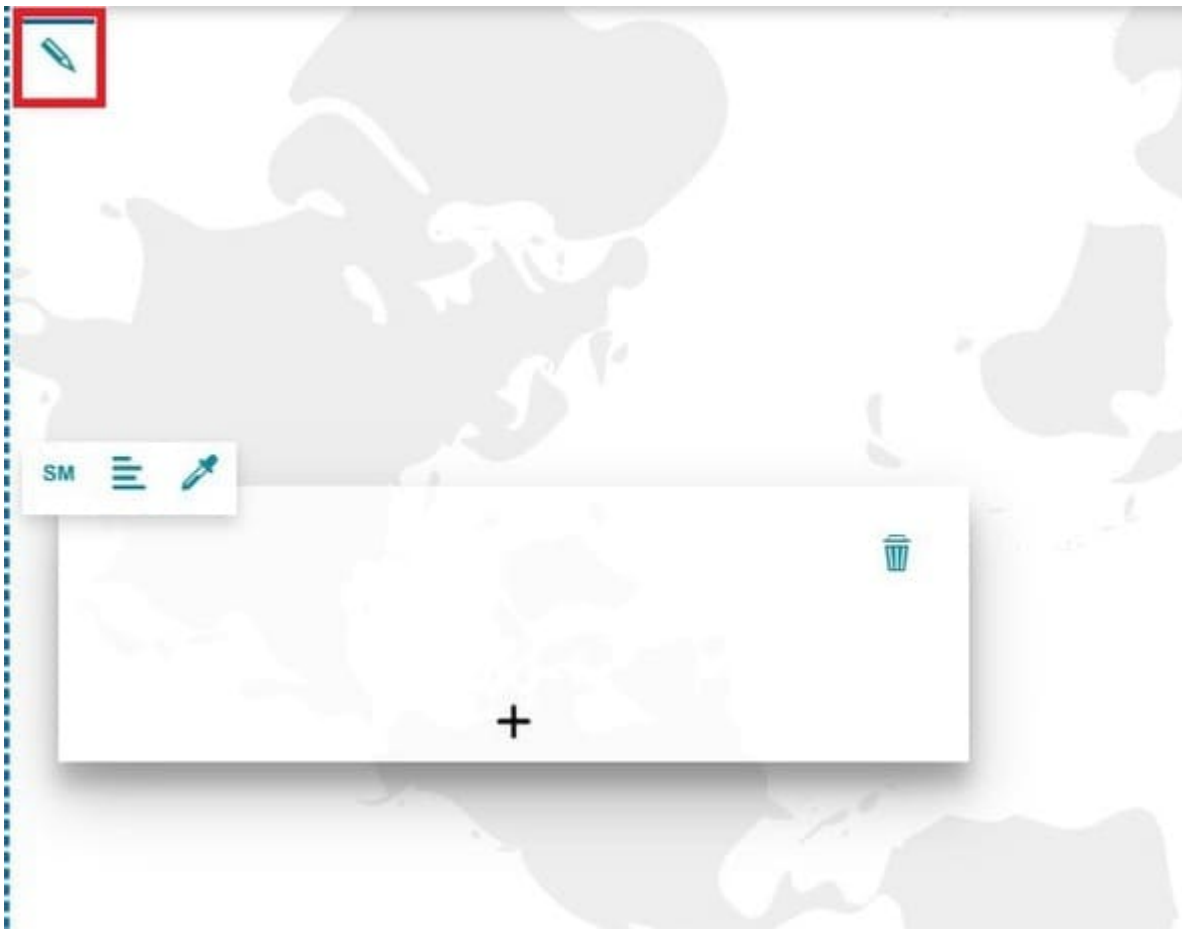
GeoCarousel Section

The GeoCarousel section allows another kind of immersive experience than the Immersive Section. The story editor can define a list of carousel cards to be presented with an accompanying descriptive content and a geographic location. In edit mode it is composed of three elements: the background map, the descriptive panel and the carousel panel where the editor can manage carousel items.



Background

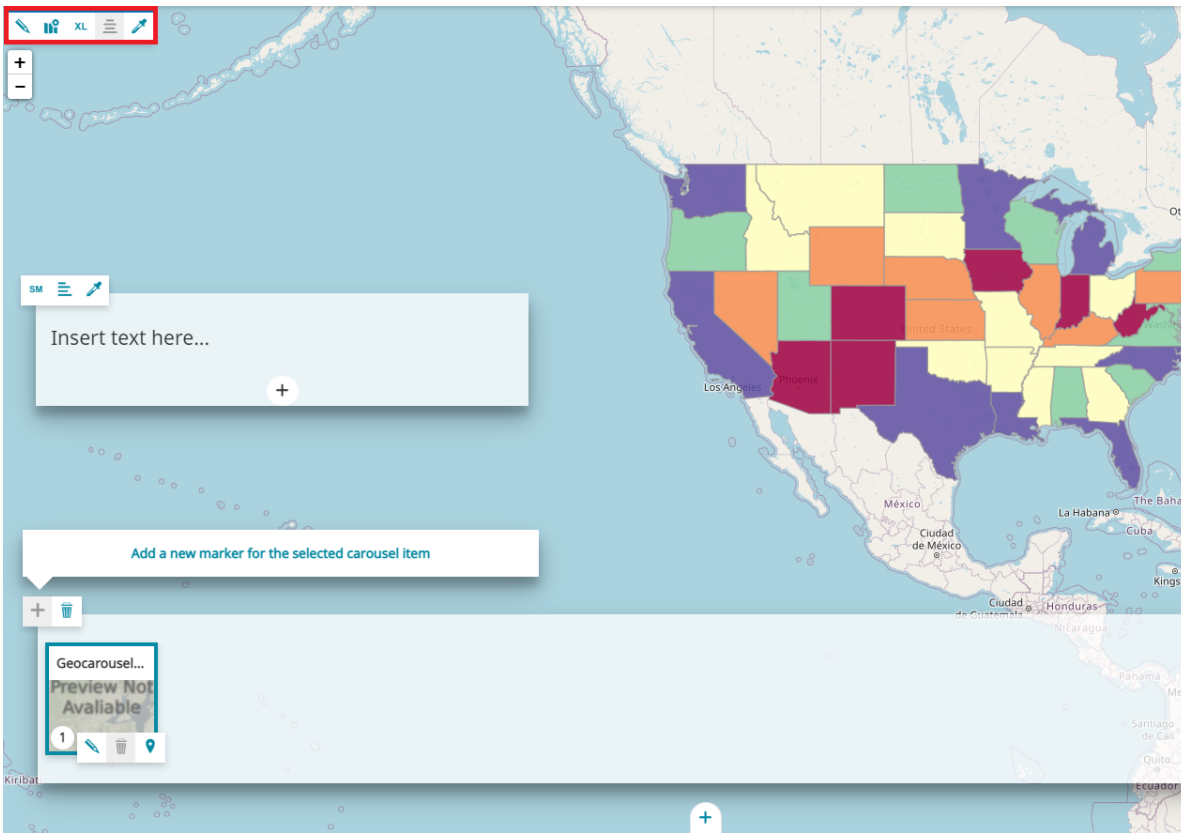
The background editing toolbar allows to add a map as a background of the section, with the **Change media source** button  that opens the Media Editor as usual.



Note

In the *GeoCarousel Section* the story editor, unlike the Immersive Section and the Title Section, can only add a **map** as a background.

Once a *map* is selected for the background, the editing toolbar appears in the upper left corner of the section allowing the story editor to manage the background content.



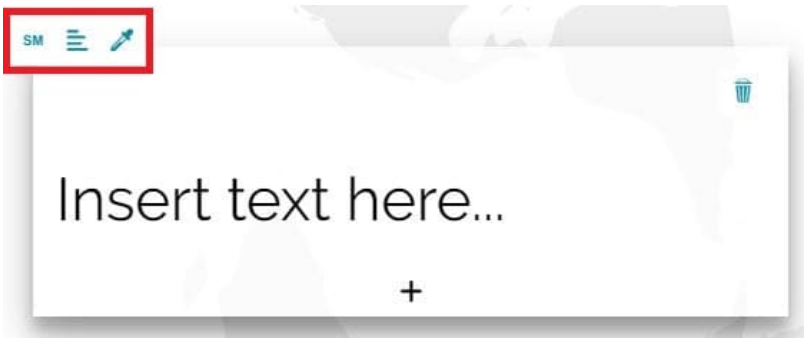
The **Background editing toolbar** allows the following actions:

- **Change media source** allows to select the media content to use for the section, clicking on this button the Media Editor opens.
- The **Edit map configuration** allows to Configure the Map
- **Change size** of the section between *Small*, *Medium*, *Large* or *Full*
- **Align content** on the *Left*, *Center* or *Right*
- **Change the background theme** to set the colour of the empty background between *Default* (same default theme settings of the story, see Story Settings), *Bright*, *Dark* or *Custom* (allows to customize the color of the background).

> **Warning**
>
> The *Align content* and the *Change field theme* buttons are disabled if the map size is full screen.

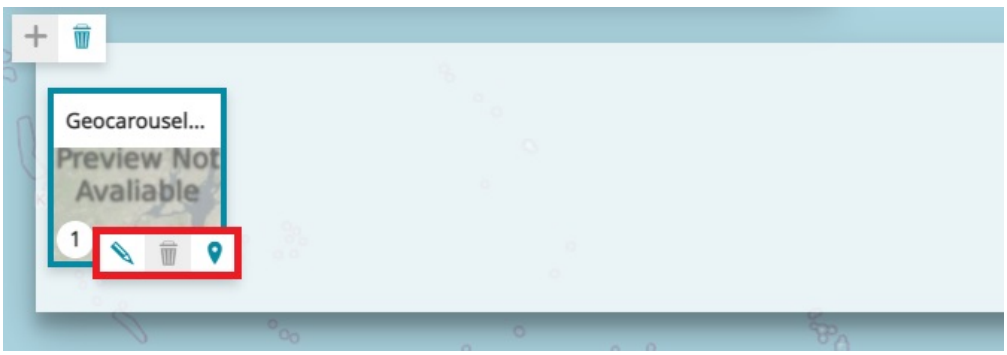
Descriptive panel

The Descriptive panel allows to put descriptive content such as *text*, *image*, *video* or *map* for the different cards composing the GeoCarousel section. The story editor can customize it through the **Content Toolbar**, as it is explained in the Content section.



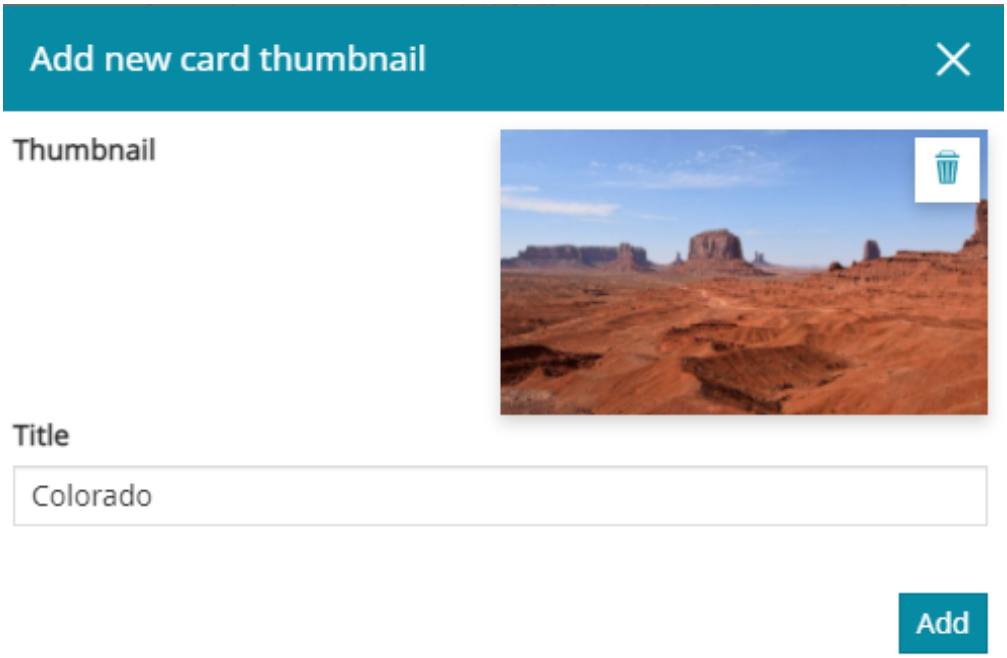
Carousel

The carousel is composed of a list of cards to be associated with a geographic location. It is located at the bottom of the GeoCarousel section and as soon as the section is added to the story, it has by default the following empty card ready to be configured:

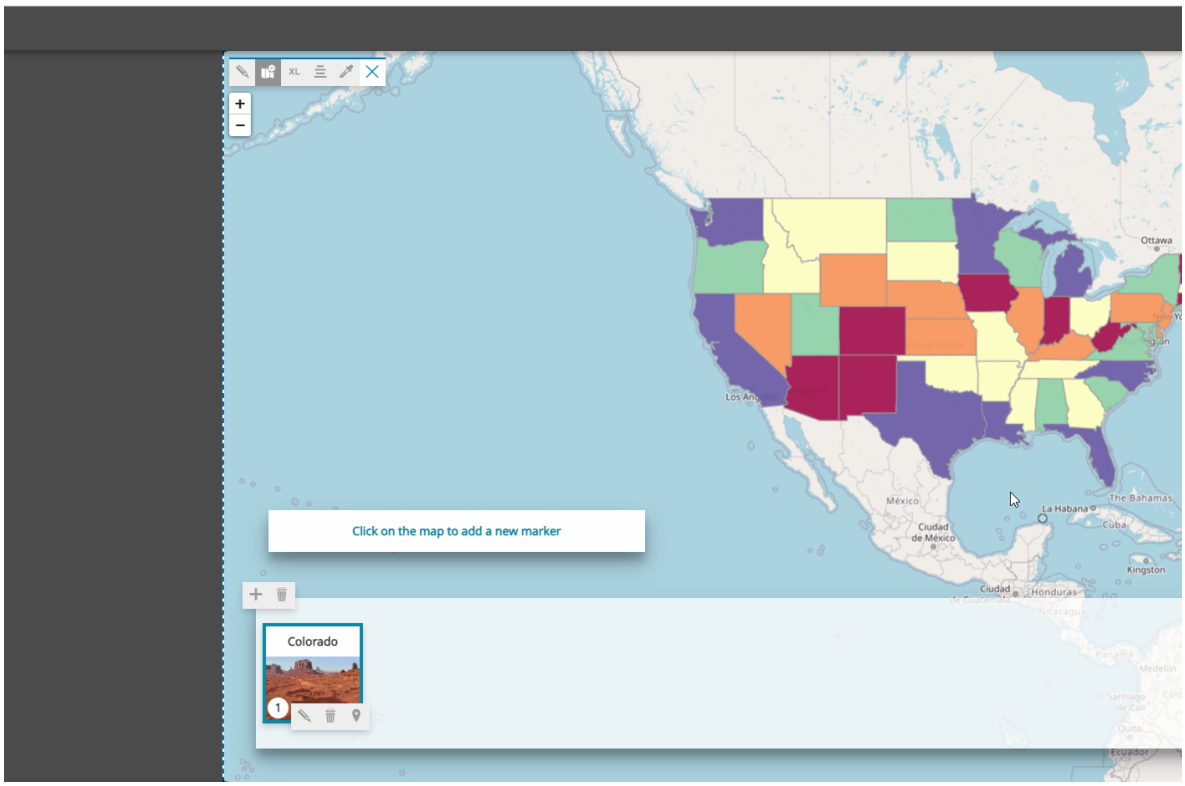


Once a card is selected in edit mode, the story editor can perform the following operations through the **Cards editing toolbar**:

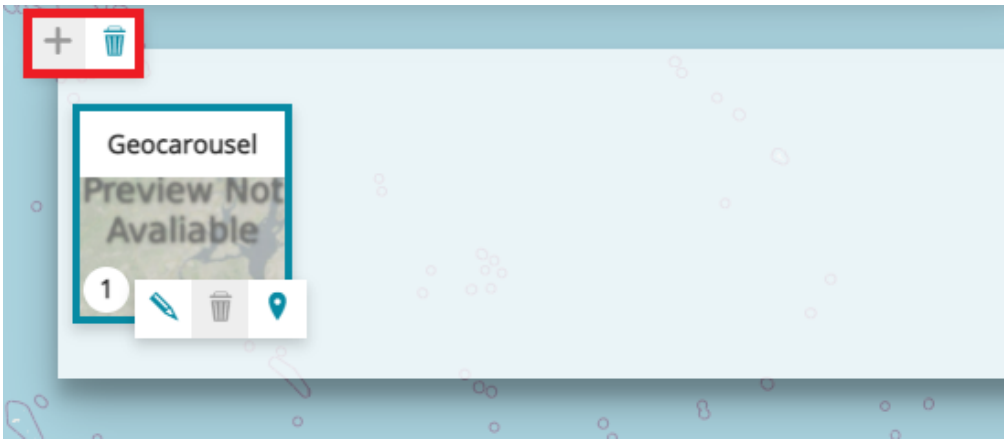
- **Edit** the card: clicking on this button the **Edit Card** panel opens to allow adding *Thumbnail* and *Title*. An example can be the following:



- **Delete** the card
- **Add marker** on map or modify the current marker position: clicking on this button the **Map Inline Editor** opens, and the story editor can click a point on map to add a new marker or change its position as follows:



In the upper left corner of the Carousel panel, a **Carousel toolbar** allows to:



- **Add card** + to the carousel
- **Remove** the *GeoCarousel Section*

> **Note**
>
> Each carousel item as well as its marker on the map is numbered to be better identified.

GeoCarousel section in View Mode

In a GeoCarousel section, in view Mode, the user can perform the following operations:

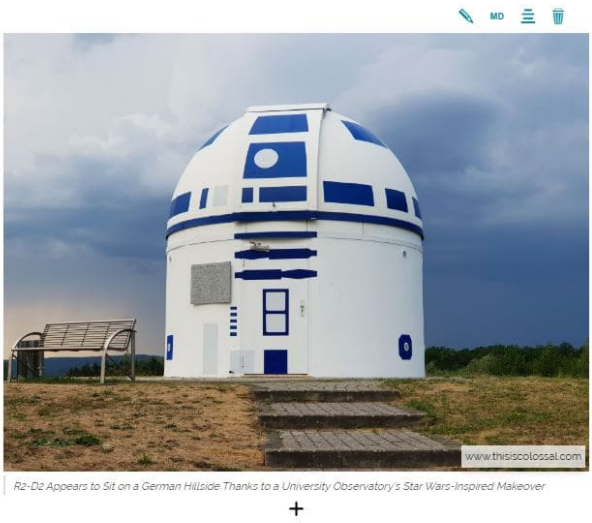
- Select a *carousel card* to view related descriptive content

- Select a *marker* on the map to display its carousel card name popup and view its descriptive content

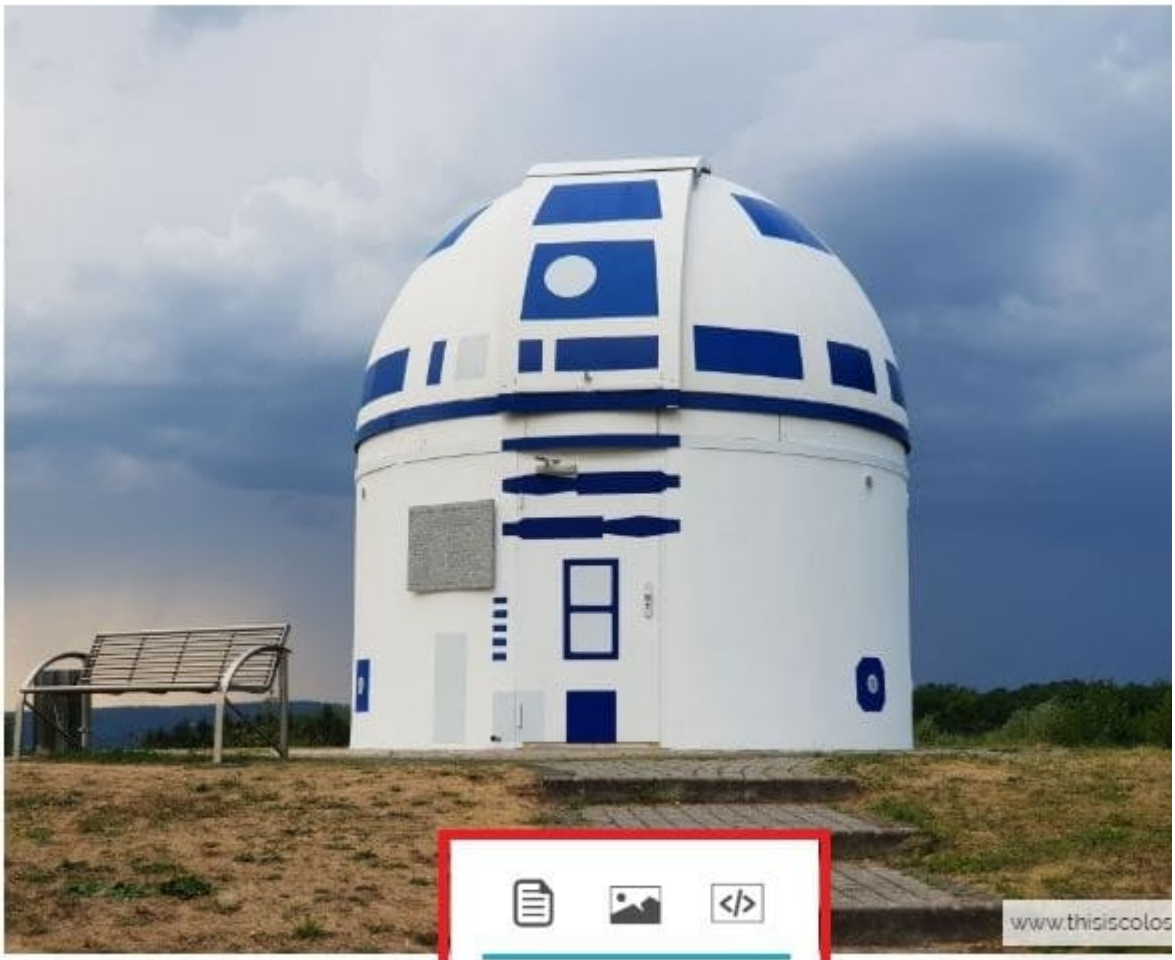
- Use the left and right arrows < to browse the different geocarousel content

Media Section

Media Sections are similar to Paragraph Sections but the main difference is that as soon as the story editor try to add a new Media Section, the Media Editor appears, asking to define the media that is going to be added. An example of a new Media Section with an image added can be like the following:



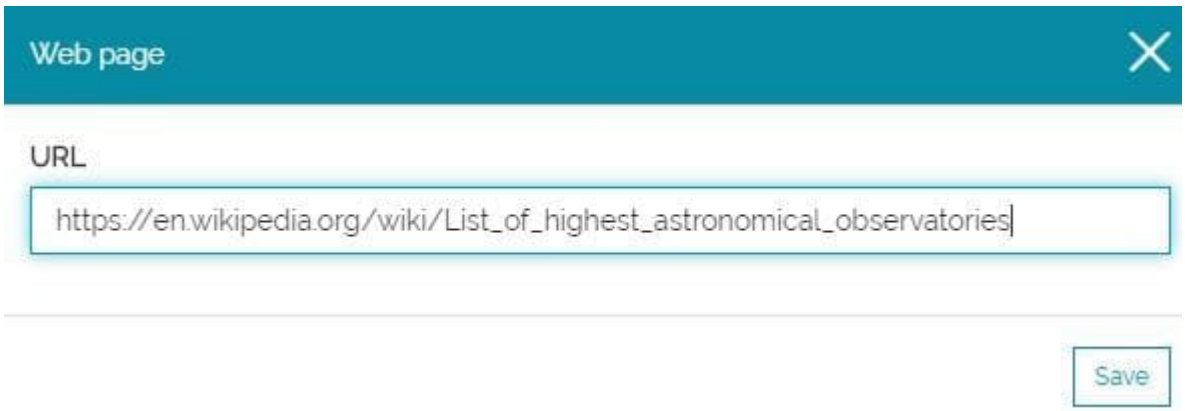
Once the first media is added to the Media Section it is possible to add new media, text contents or web page contents or remove the existing ones as decribed also in the Paragraph Sections.



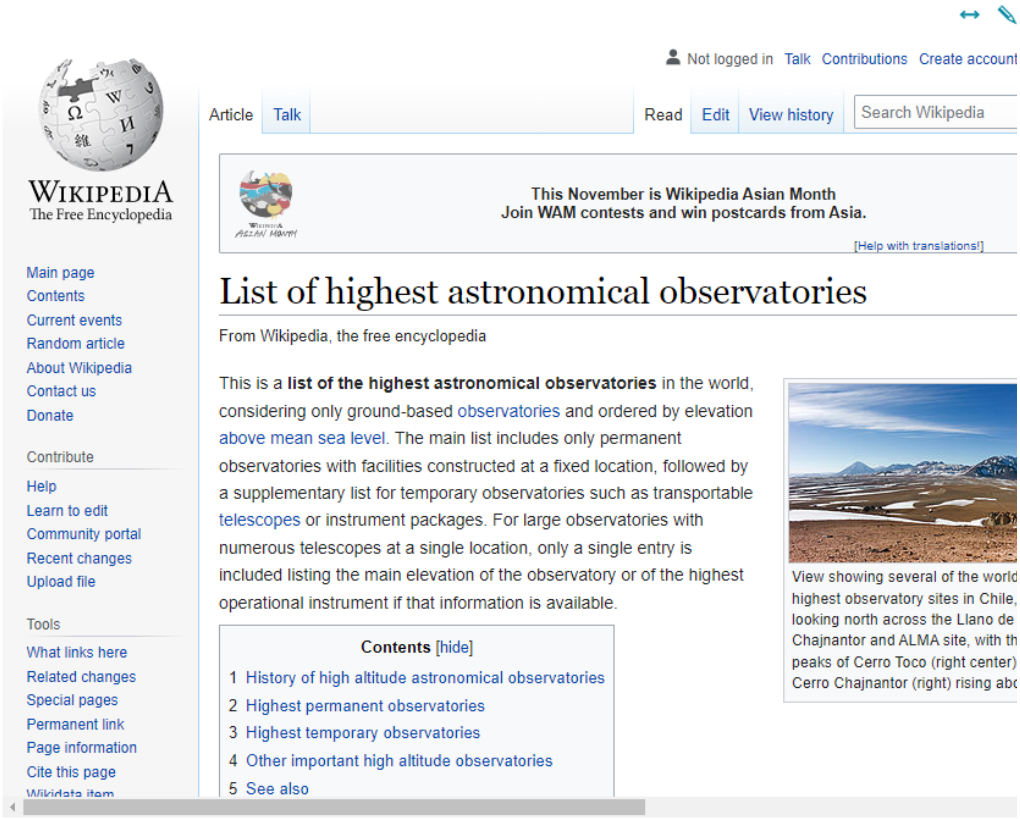
R2-D2 Appears to Sit on a German Hillsid[illegible] Star Wars-Inspired Makeover

Web Page Section

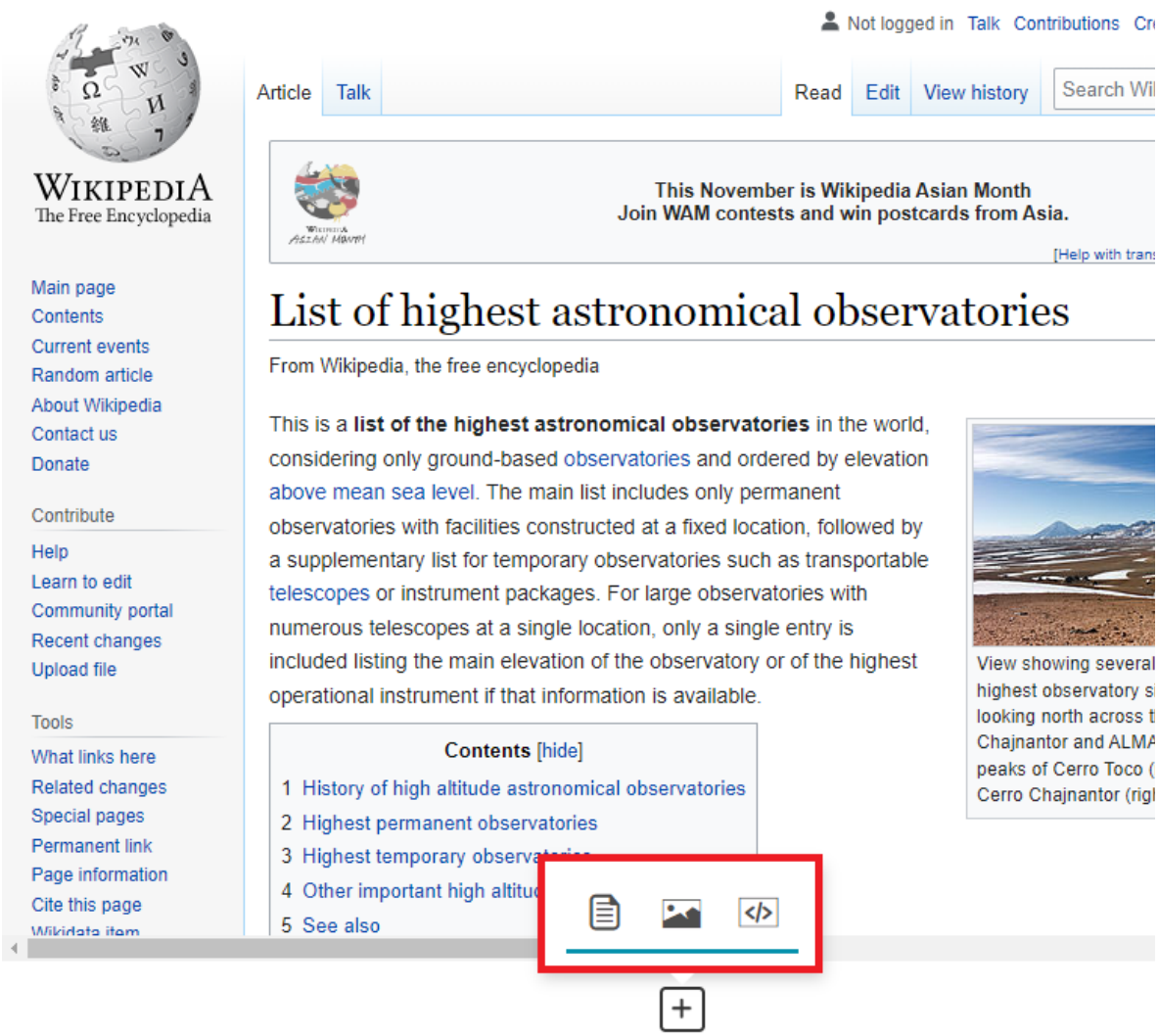
Through this kind of sections the story editor can embed third party contents in the story (like other web pages available on the web). The Web Page Section is similar to the Paragraph Section and the Media Section: adding this section a modal window opens to specify the URL of the web page that is going to be added.



Below an example of a Web Page Section that embed a Wikipedia site page:

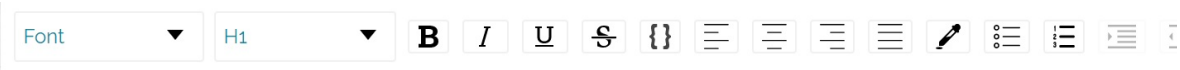


It is possible to add or remove multiple Web Page contents in the same way of text and media contents as it is explained in the Paragraph Sections.

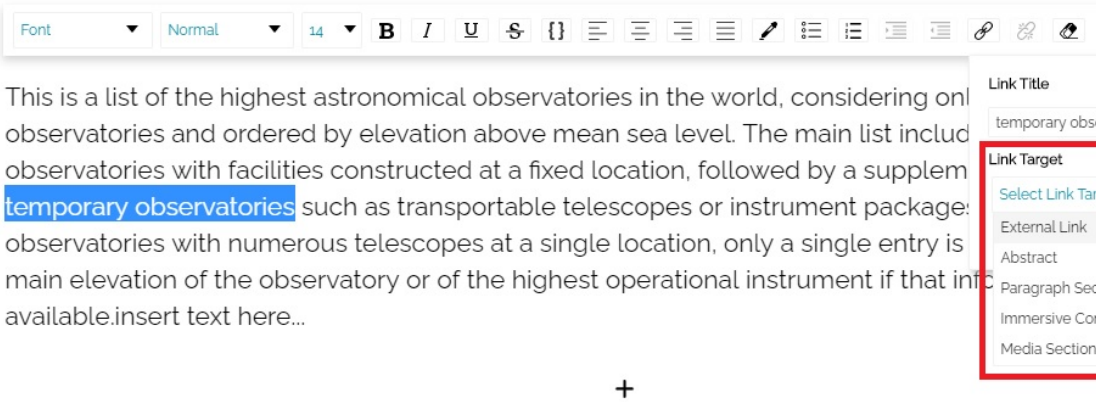

Not logged in Talk Contributions Cr
WIKIPEDIA
The Free Encyclopedia
Article Talk
Read Edit View history
Search Wi
This November is Wikipedia Asian Month
Join WAM contests and win postcards from Asia.
[Help with trans
Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate
Contribute
Help
Learn to edit
Community portal
Recent changes
Upload file
Tools
What links here
Related changes
Special pages
Permanent link
Page information
Cite this page
List of highest astronomical observatories
From Wikipedia, the free encyclopedia
This is a list of the highest astronomical observatories in the world, considering only ground-based observatories and ordered by elevation above mean sea level. The main list includes only permanent observatories with facilities constructed at a fixed location, followed by a supplementary list for temporary observatories such as transportable telescopes or instrument packages. For large observatories with numerous telescopes at a single location, only a single entry is included listing the main elevation of the observatory or of the highest operational instrument if that information is available.
View showing several highest observatory s looking north across t Chajnantor and ALMA peaks of Cerro Toco (Cerro Chajnantor (rig
Contents [hide]
1 History of high altitude astronomical observatories
2 Highest permanent observatories
3 Highest temporary observa
4 Other important high altitu
5 See also

Text Editor Toolbar

With the *Text Editor Toolbar* it is possible to customize the text by modifying the following aspects:



- **Font** to choose the text font (*Inherit*, *Arial*, *Georgia*, *Impact*, *Tahoma*, *Time New Roman*, *Verdana*)
- **Block Type** by choosing between the available ones (*Normal*, *H1*, *H2*, *H3*, *H4*, *H5*, *H6*, *Blockquote*, *Code*)
- **Text style** to insert text in *Bold* , *Italic* , *Underline* or *Strikethrough*
- **Monospace** to insert the same space between words
- **Alignment** inside the text window (*Left*, *Center*, *Right* or *Justify*)
- **Color Picker** to change the text color
- **Bullet list** to create a *Unordered* list or *Ordered* list
- **Indent/Outdent** to indent the text in relation to the left margin or to the right margin
- **Link** to configure a hyperlink for the selected portion of text. The GeoStory editor can define hyperlinks to external web pages by choosing the *External link* option in the **Link target** dropdown menu and entering the related URL. As an alternative, it is also possible to define a hyperlink to other sections of the same GeoStory by choosing one of the sections available in the **Link target** dropdown menu.

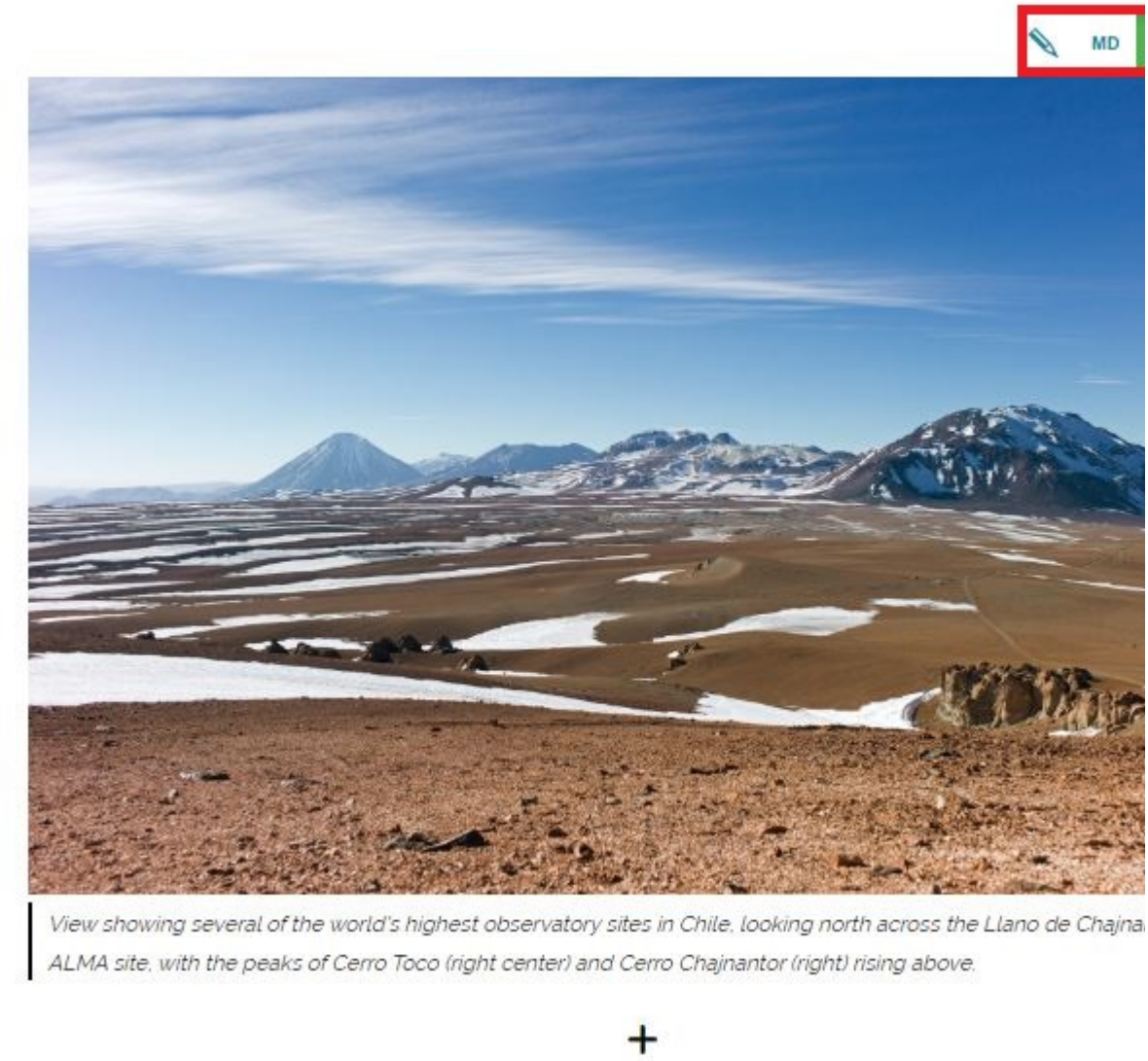


> **Note**
>
> In order to setup an hyperlink to an external website, the protocol must be specified (e.g., `http://` or `https://`).

- **Remove** to remove the formatting

Image Content Toolbar

As soon as an image content is added, the Image Content Toolbar appears on top of the image:



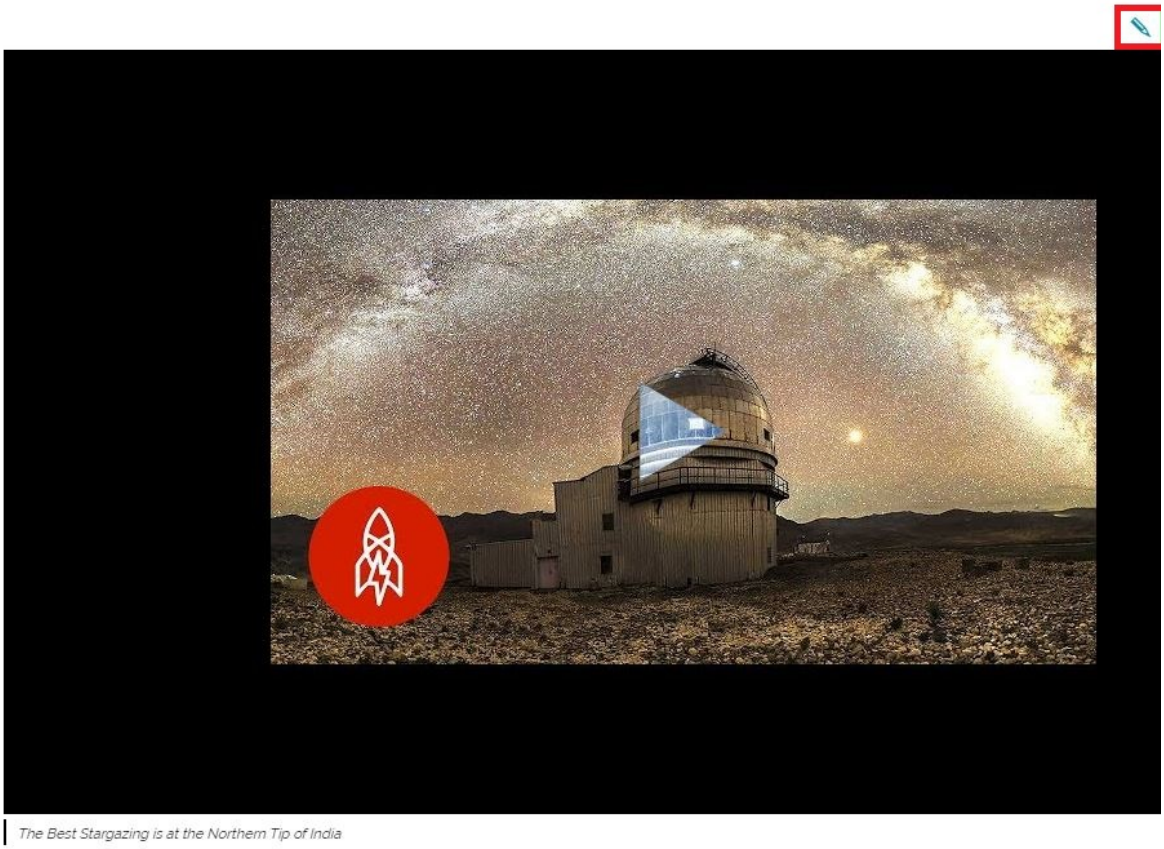
Through this toolbar, the story editor is able to perform the following operation:

- **Change media source** accessing the Media Editor
- **Change size** LG choosing between *Small*, *Medium*, *Large*

- **Hide caption** button  to show/hide the description under the image: this button is present only if a description has been provided for the image resource (see the Media Editor tool for example)
- **Remove**  the image content

Video Content Toolbar

As soon as a video content is added, the Video Content Toolbar appears on top of the video:



The Best Stargazing is at the Northern Tip of India

Through this toolbar, the story editor is able to perform the following operation:

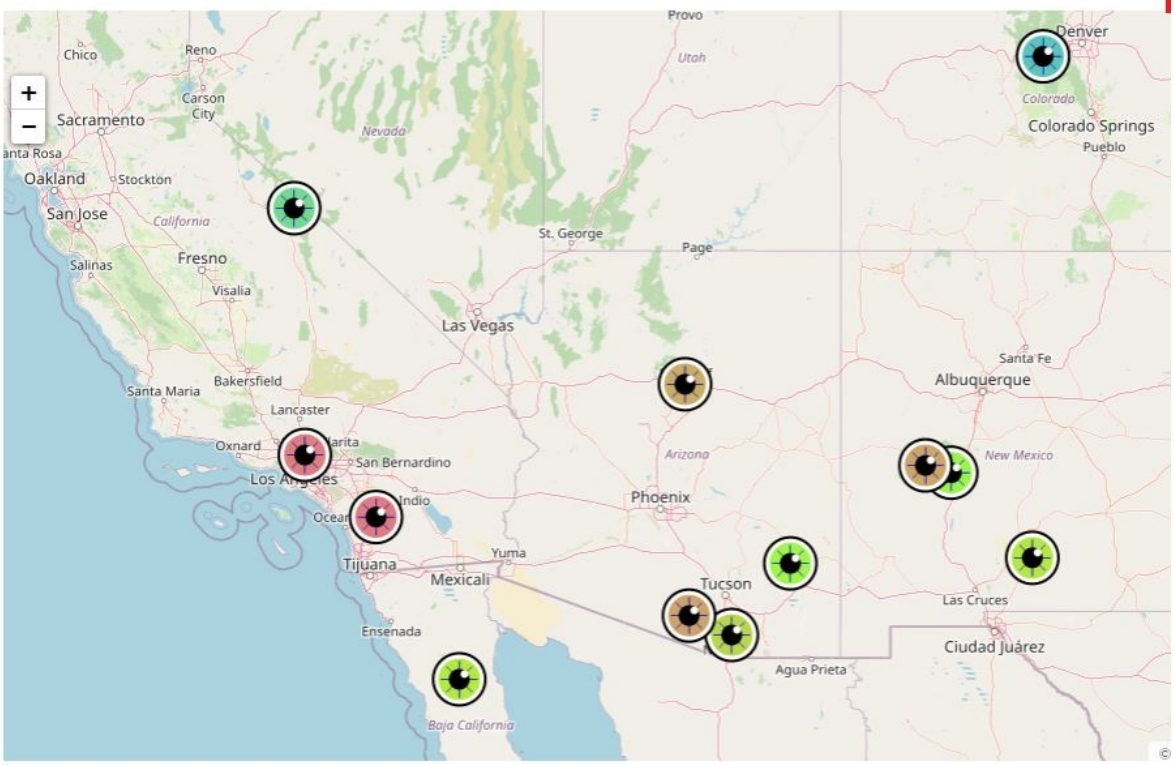
- **Change media source** to open the Media Editor and change (or configure) the media content
- **Mute video** to disable the video audio
- **Enable Autoplay** to play the video automatically once the user is on it
- **Enable Loop** to continuously repeat the video

- **Hide caption** button  to show/hide the description under the video: this button is present only if a description has been provided for the video resource (see the Media Editor tool for example)
- **Remove** the video content

> **Note**
>
> Inside the Media Editor you can watch a preview of the video before adding it to the story. The video play will be available only in *View Mode* of the story: it is not available in edit mode except in the media editor as a preview.

Map Content Toolbar

As soon as a map content is added, the Map Content Toolbar appears on top of the map component:



In particular, through this toolbar, the story editor is able to perform the following operation:

- **Change media source** accessing the Media Editor
- **Edit map configuration** through which it is possible Configure the map
- **Change size** LG choosing between *Small*, *Medium*, *Large*

- **Hide caption** button  to show/hide the description under the map: this button is present only if a description has been provided for the map resource (see the Media Editor tool for example)
- **Remove**  the map content

Web Page Content Toolbar

As soon as a Web Page Content is added, the toolbar appears on top of the content itself:

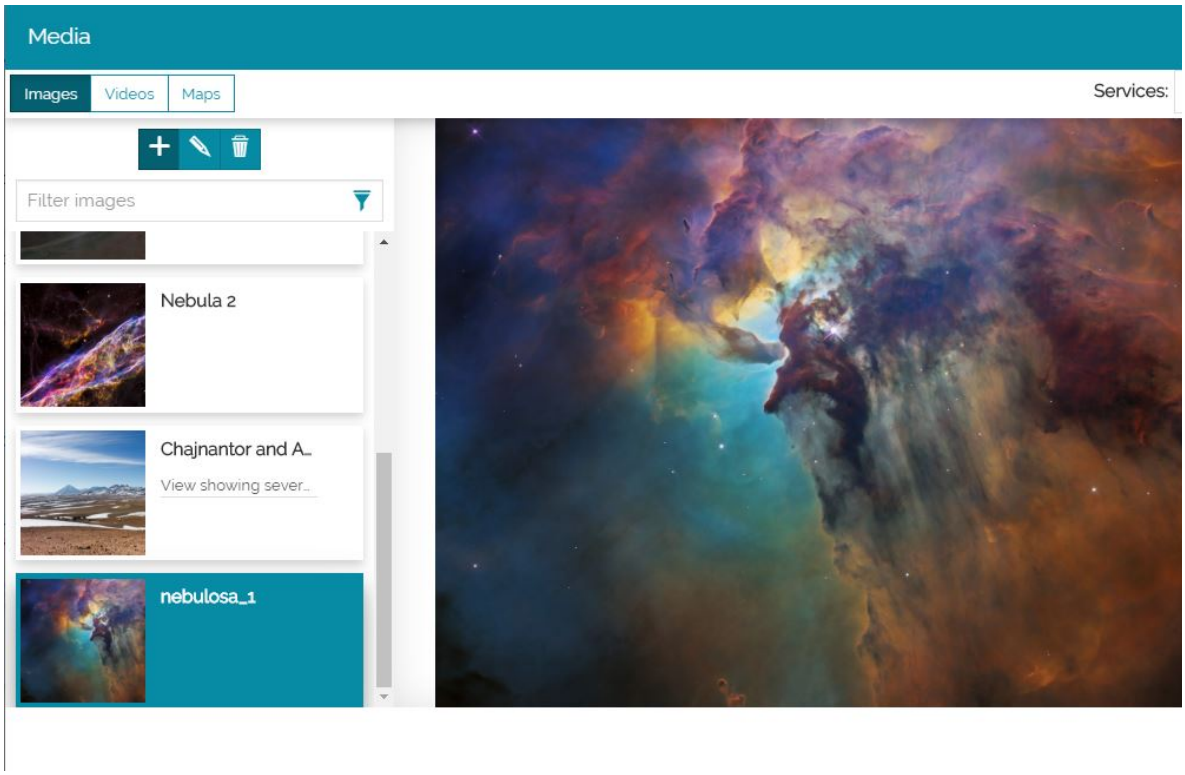
In particular, through this toolbar, the editor is able to perform the following operation:

- **Change horizontal size** ↔ choosing between *Small*, *Medium*, *Large* and *Full*
- **Edit web page URL** ✎ accessing the web page windows to change the web URL

- **Change vertical size** choosing between *Small*, *Medium*, *Large*
- **Remove** the Web Page content

Media Editor Window

The **Change media source** button allows to access the Media Editor:

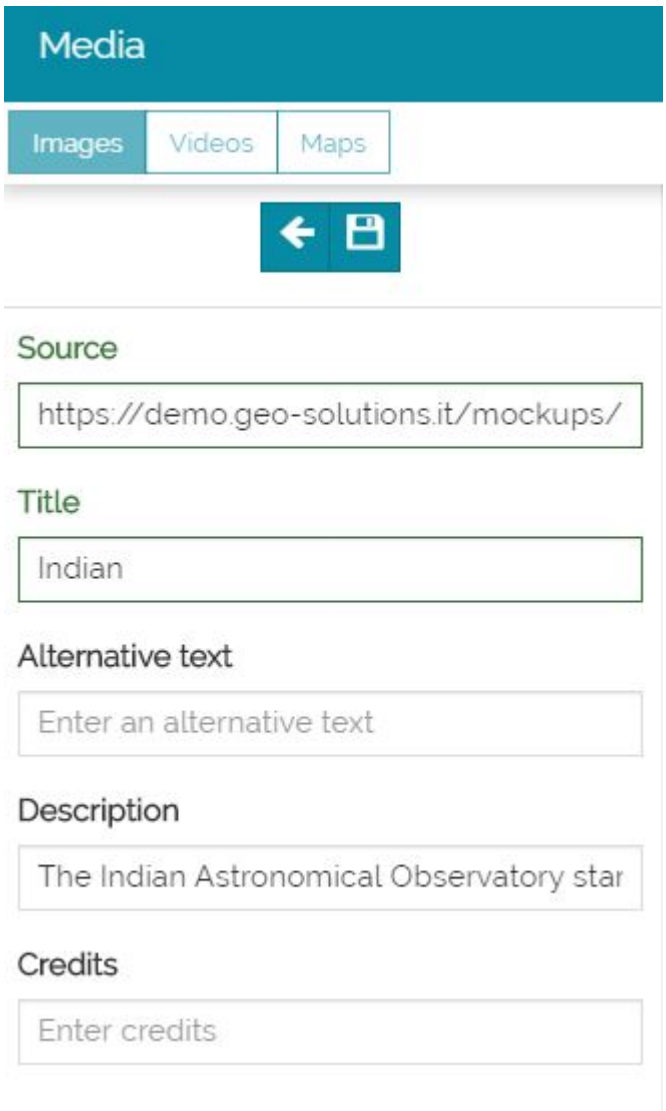


Through this window, you can add or edit three different types of media:

- Images
- Videos
- Maps

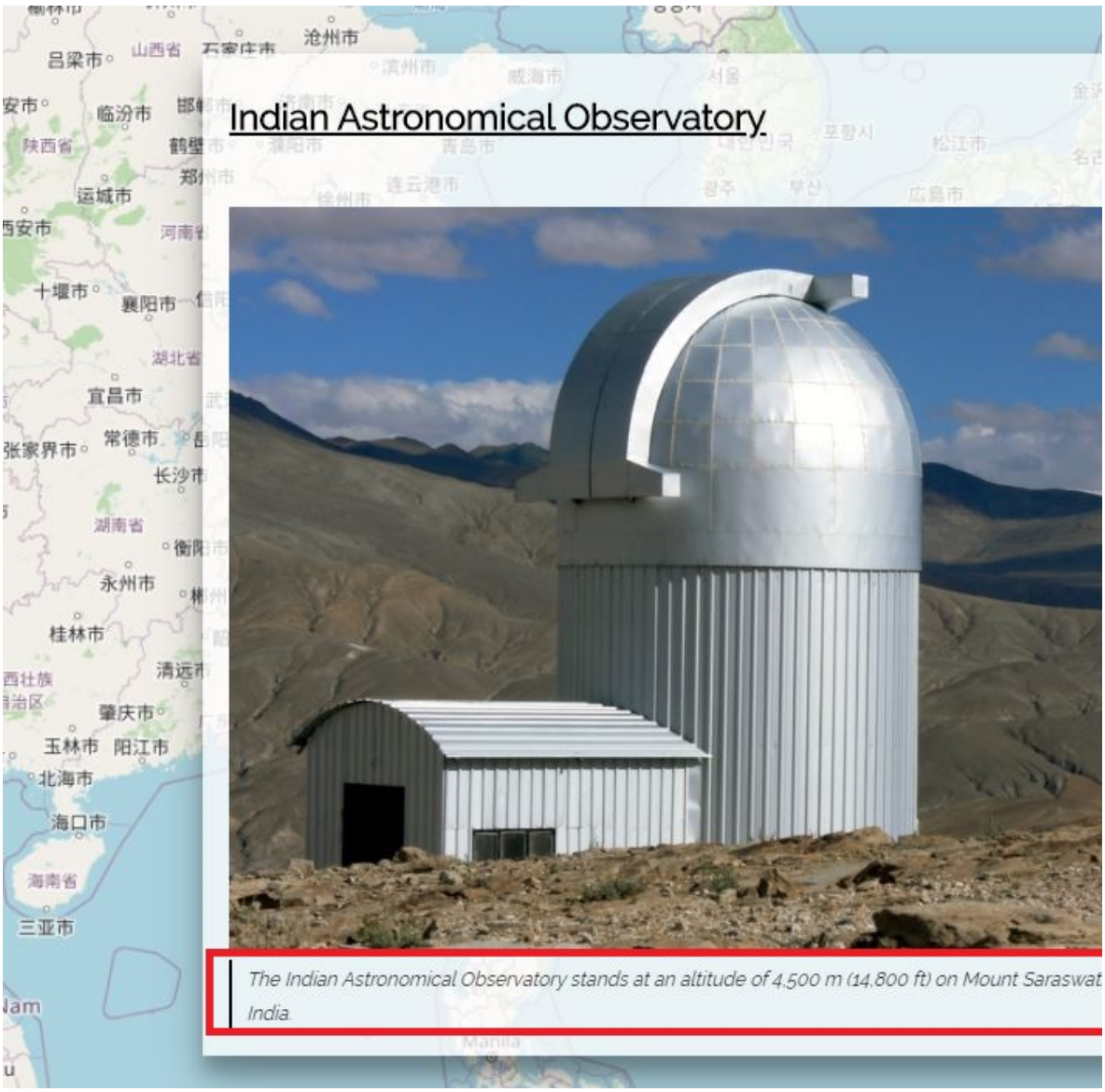
Images

In order to add an image, the stroy editor can click on the **Images** tab Images in order to switch to the images section. In this section of the media editor window, with a click on the **Add** button , it is possible to define the image settings.



In particular, it is possible to insert the following parameters:

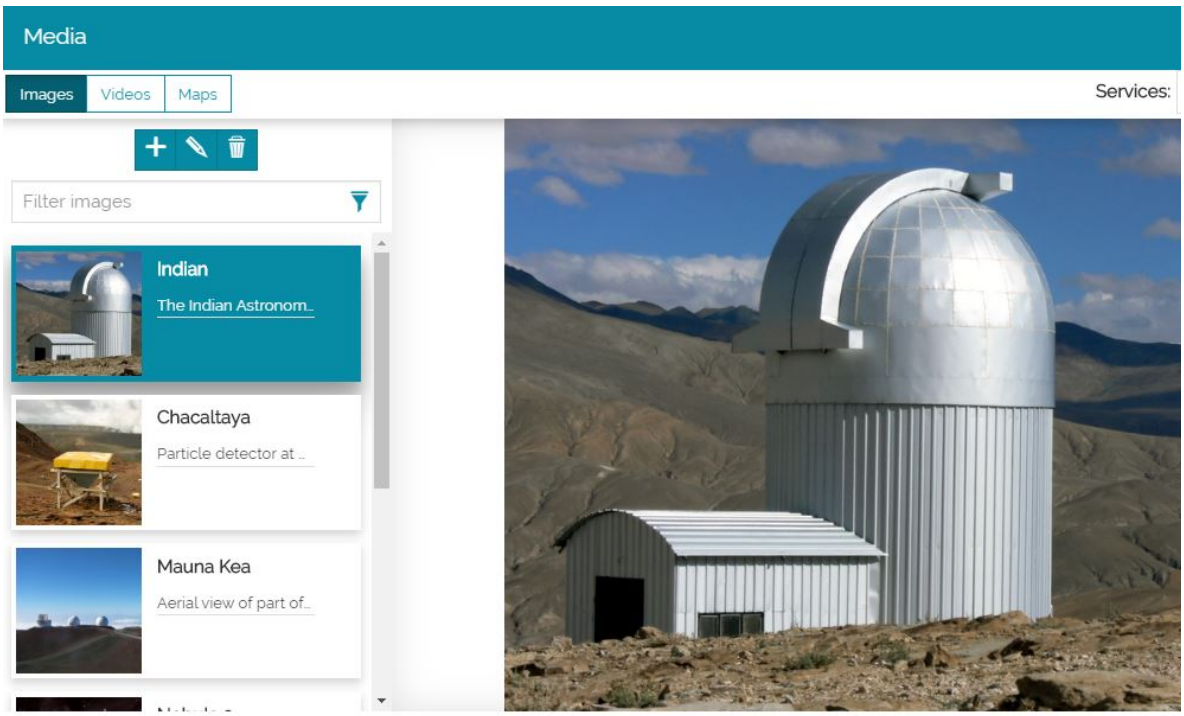
- The **Source** of the image (its URL)
- The **Title** of the image
- The **Alternative text**, that appears if the link is broken
- A **Description**, used to explain the contents of the image. The description is available in the images preview list and, if the image is added as content in the *Pharagraph section*, in the *Immersive section* or in the *Media section*, under the image, as follows:



- The **Credits**, displayed on the bottom-right of the image

> **Warning**
>
> **Source** and **Title** are mandatory fields.

With a click on **Save**  , the image is included in the list of the available images ready to be selected for the current story. The image also becomes immediately available on preview.



Note

It is always possible to change the image settings through the **Edit** button  also in a second time.

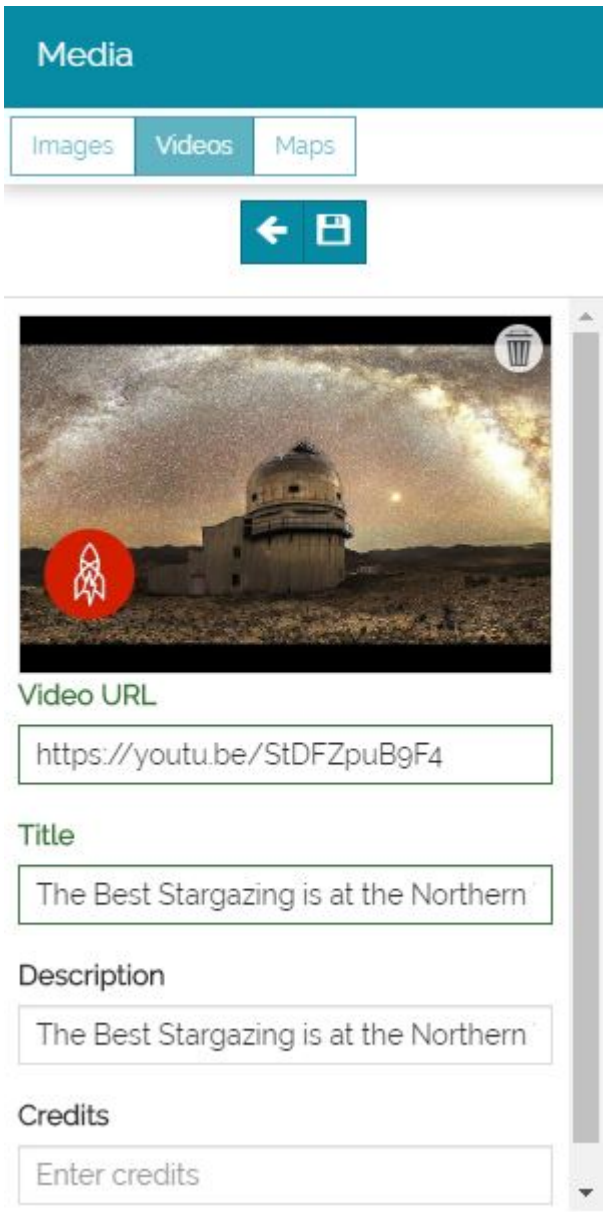
Once selectd in the list, the image can be included in the story by clicking on the **Apply** button 

.

Videos

In order to add a video, the stroy editor can click on the **Video** tab 

in order to switch to the videos section. In this section of the media editor window, with a click on the **Add** button  , it is possible to define the video settings.

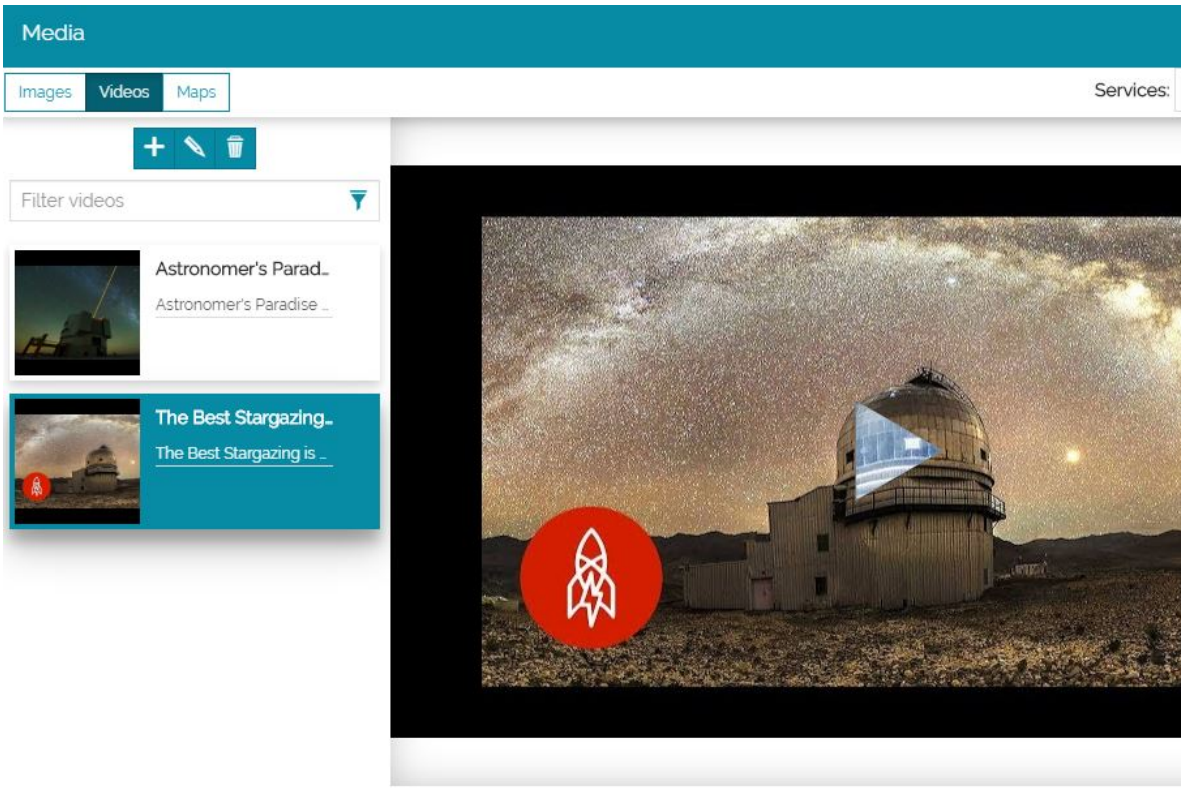


In particular, it is possible to insert the following parameters:

- The **URL** of the video
- The **Title** of the video
- A **Description**, used to explain the contents of the video. The description is available in the viedos preview list and it can appear under the video itself if the video is added as a content of a section (*Pharagraph section*, *Immersive section* or *Media section*)
- The **Credits**, displayed on the bottom-right of the video

> **Warning**
>
> **Video URL** and **Title** are mandatory fields.

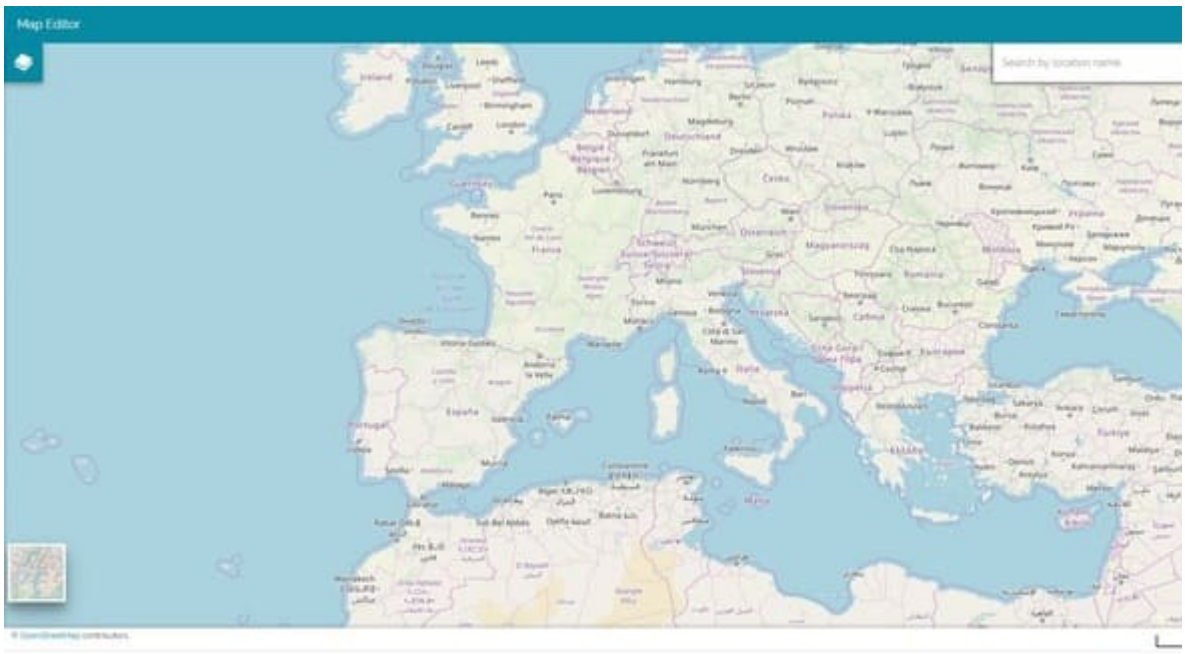
With a click on **Save**  , the video is included in the list of the available videos ready to be selected for the current story. The video also becomes immediately available on preview.



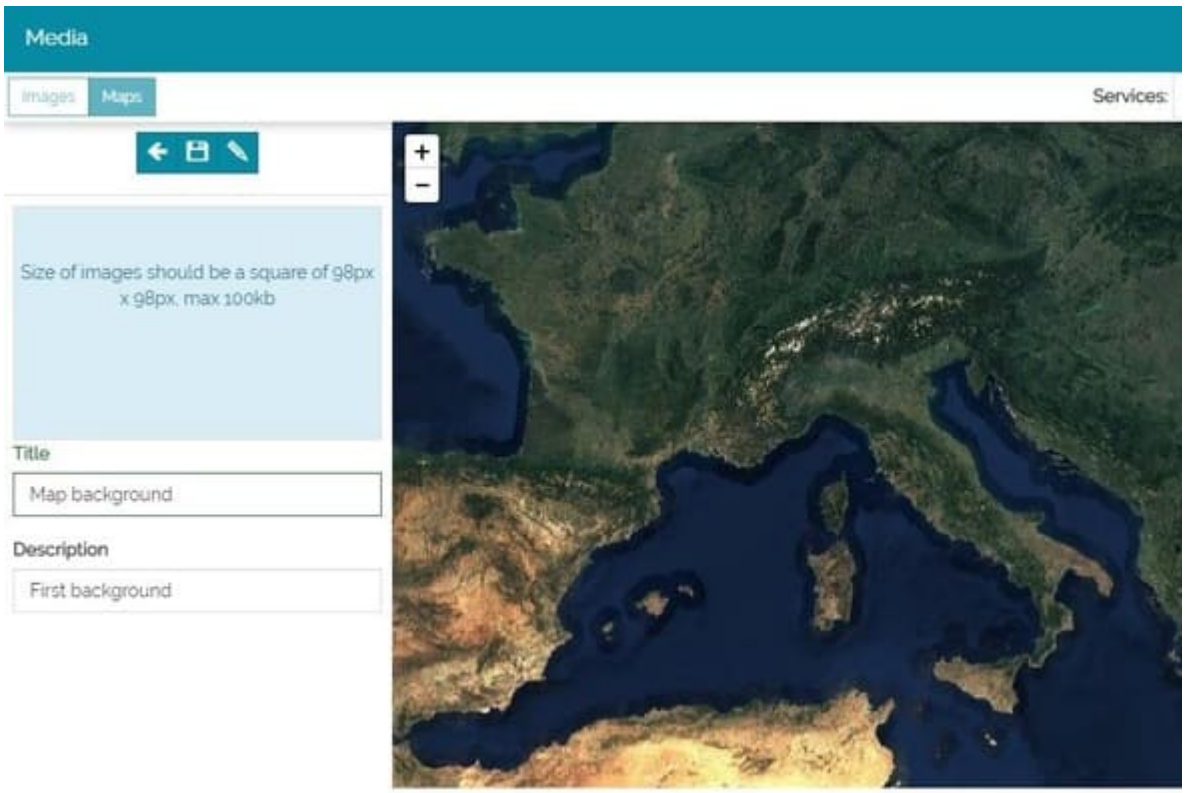
Once selectd in the list, the video can be included in the story by clicking on the **Apply** button  .

Maps

To add a map to the story, the story editor can click on the **Maps** tab  and so swich to the map section of the Media Editor. Here the list of maps ready to be used for the story is available: the editor can search and select a map in the list to apply it in the story or create a map from scratch by clicking on the **Add** button  to open the Map Editor.

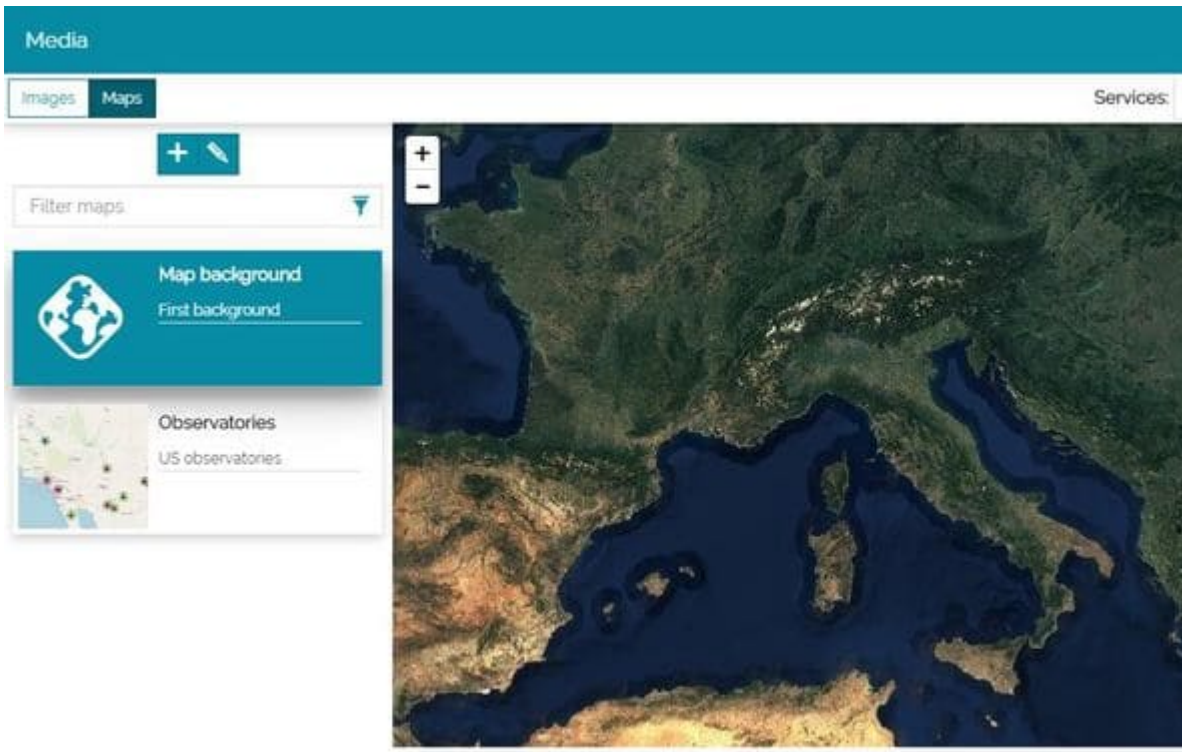


Once the map is ready, the story editor can click on **Ok** button ok to proceed with the next step and therefore insert the related map metadata like **Thumbnail**, **Title** and **Description**.



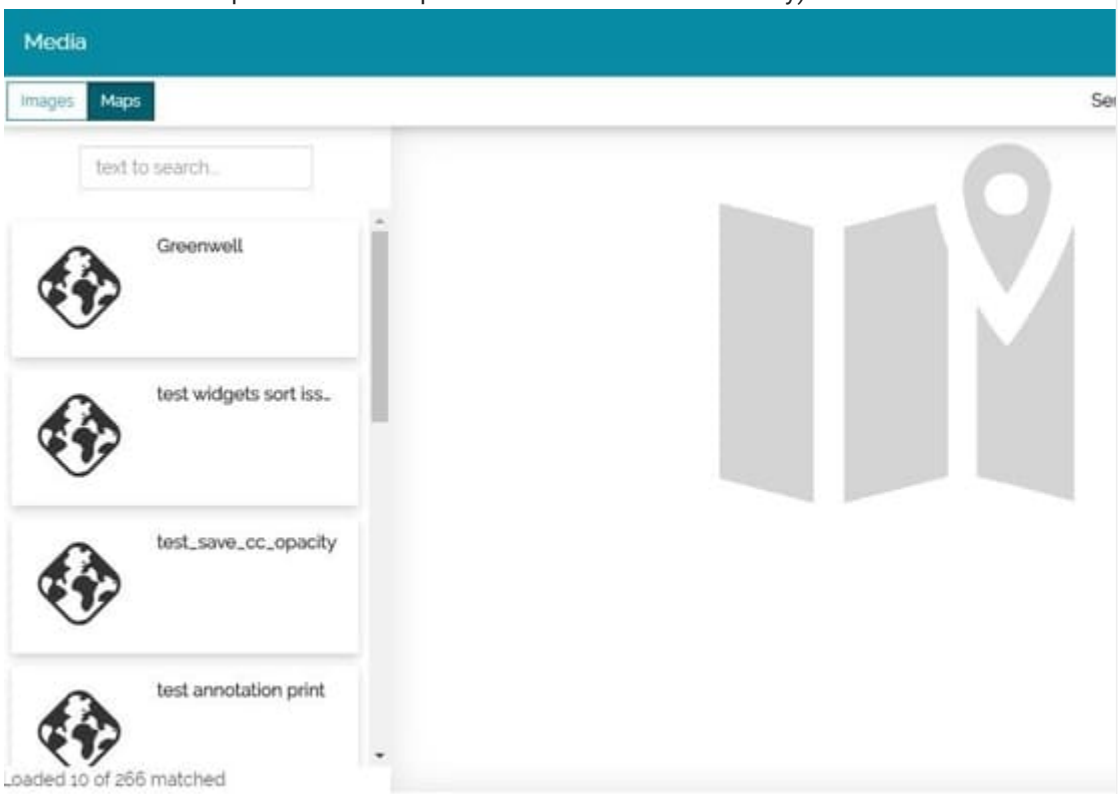
> **Note**
>
> The description is available in the maps preview list and it can appear under the map itself if the map is added as a content of a section (*Pharagraph section*, *Immersive section* or *Media section*).

With a click on **Save** button the map is saved and it will be available in the list of the *Current Story* maps. The story editor can select it to be used as a map component in the story by clicking on **Apply** button Apply .



Note

By default a drop down list on the top right corner of the Media Editor allows to switch between maps currently used in the story (so the story editor can use again a map already present in the story if needed) and existing maps already available inside the MapStore catalog (that means that also classic maps created in MapStore can be used inside a Story).

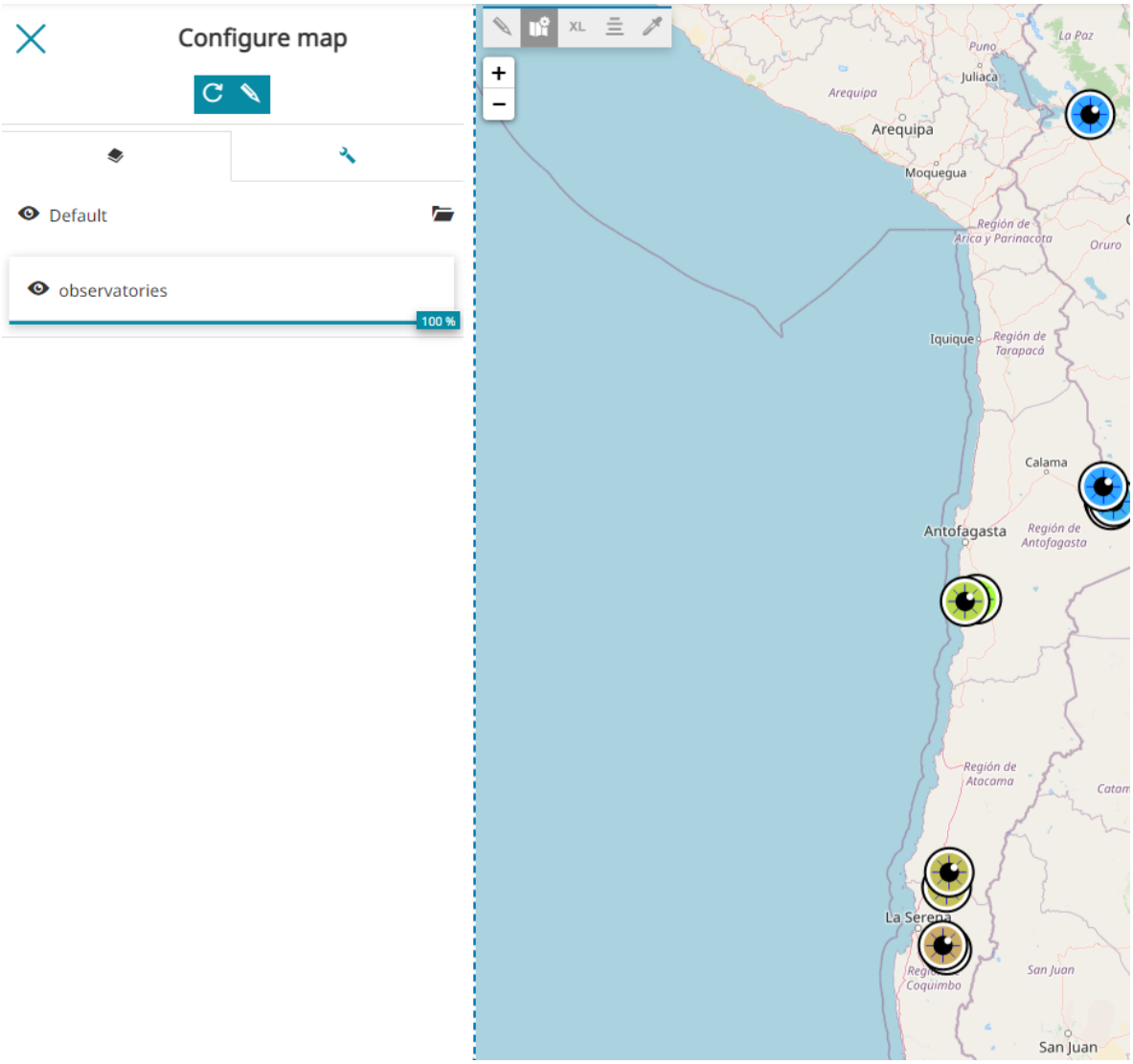


Below is an example of a *Map* used as a background of a Title Section:


List of Highest Astronomical Observatories
From Wikipedia, the free encyclopedia

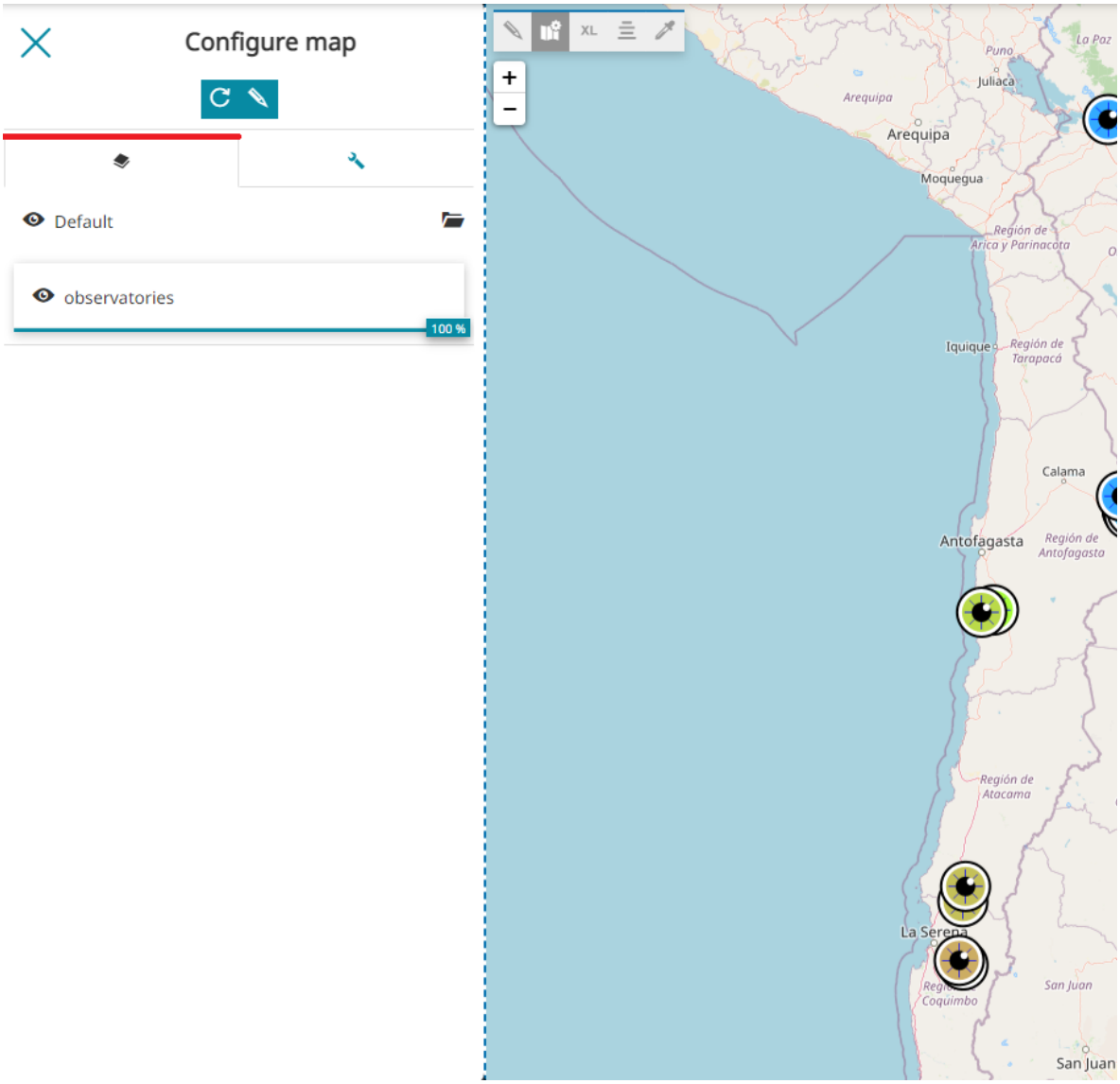
Configure the map

With the Edit map configuration button the **Map Inline Editor** opens to give the opportunity to do quick customizations (like basic map settings, layer opacity and something more) to the map (more advanced customizations then, are allowed only through the *Advanced Map Editor*).



Layers

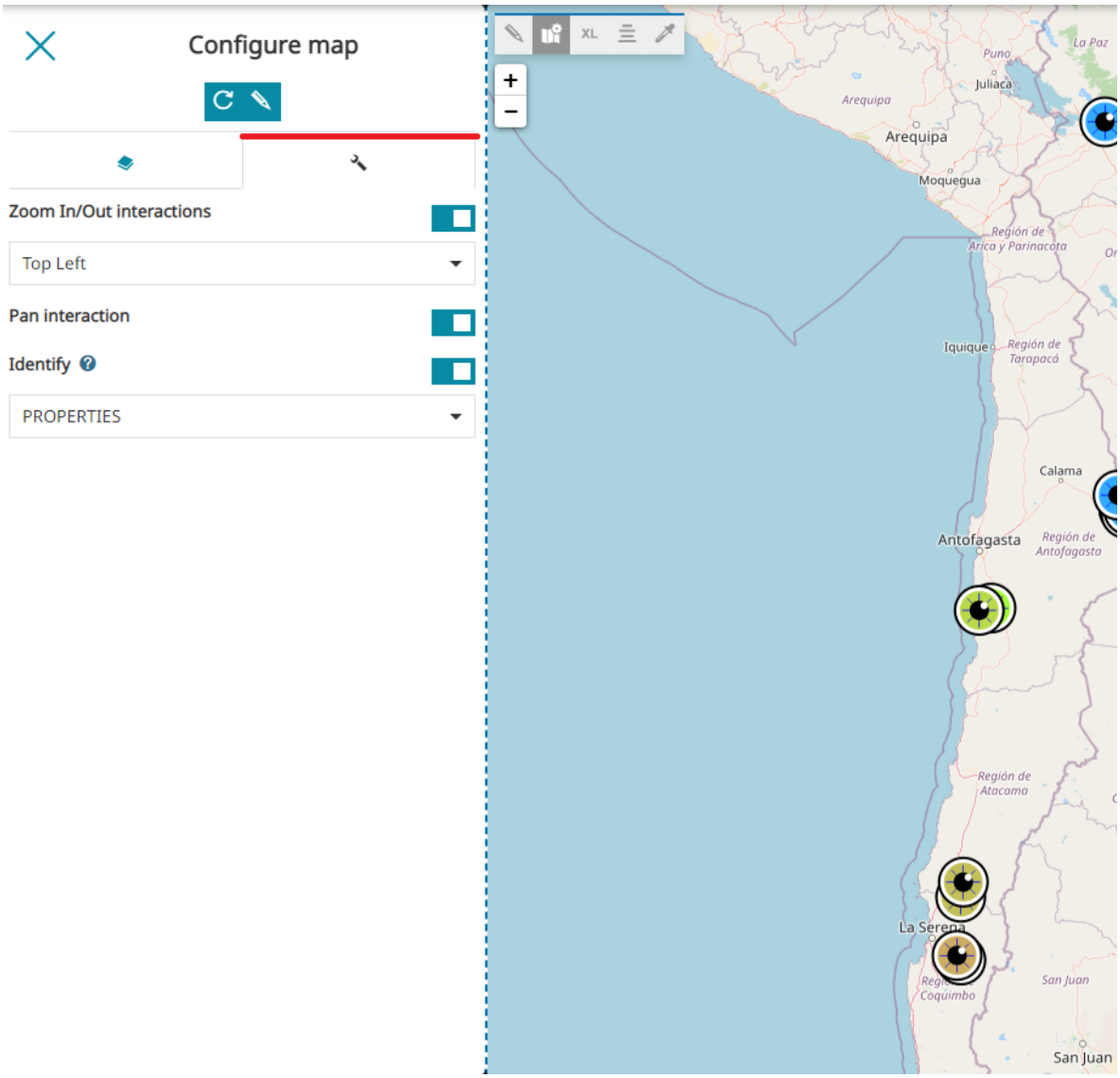
The Map Inline Editor opens with the Layers section available, where it is possible to edit the layers settings (by selecting a layer in the TOC) and the visibility of layers present in the map:



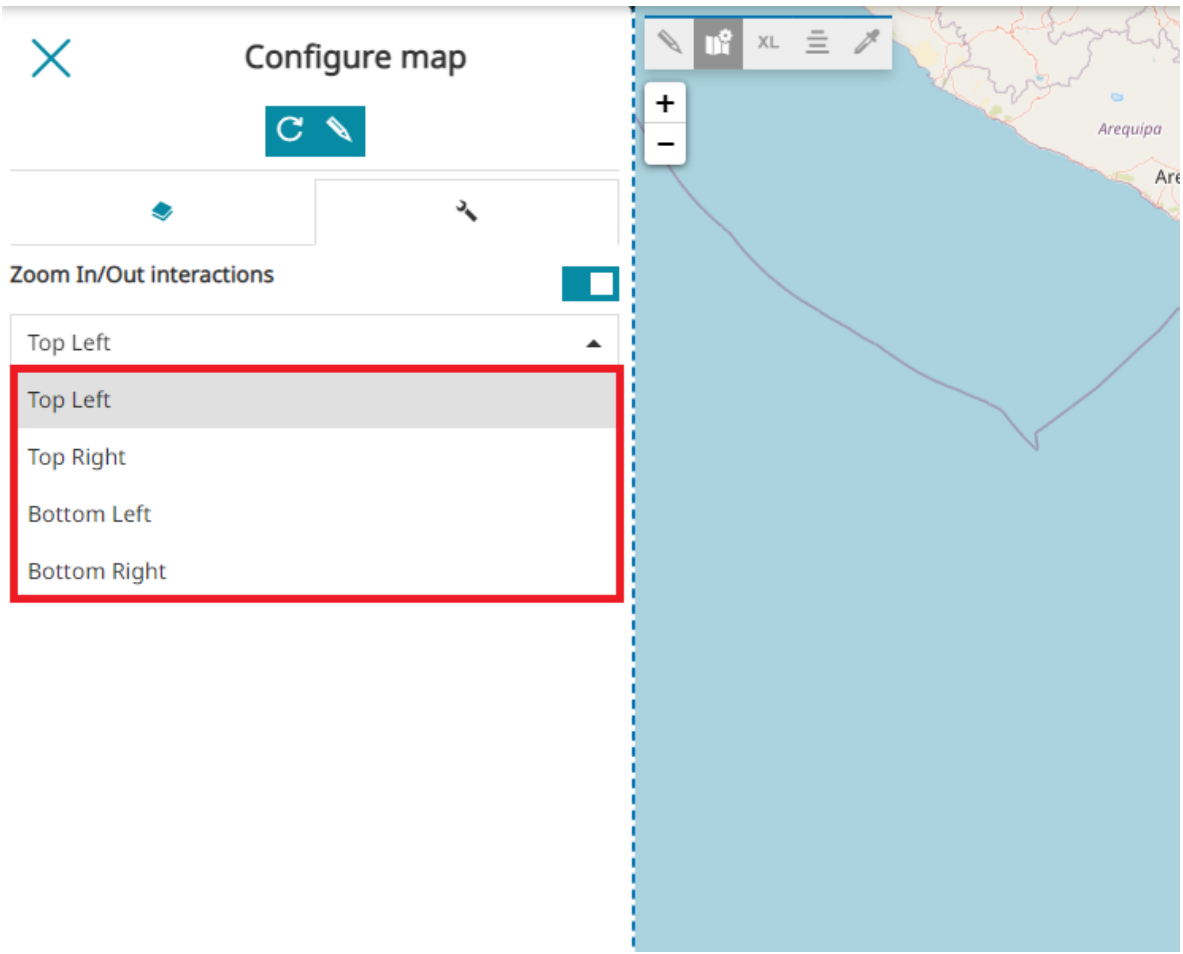
- Control the layer transparency by scrolling left and right the transparency bar  .
- Toggle the layer visibility by switching off  and on  the "eye" icon.

Setting

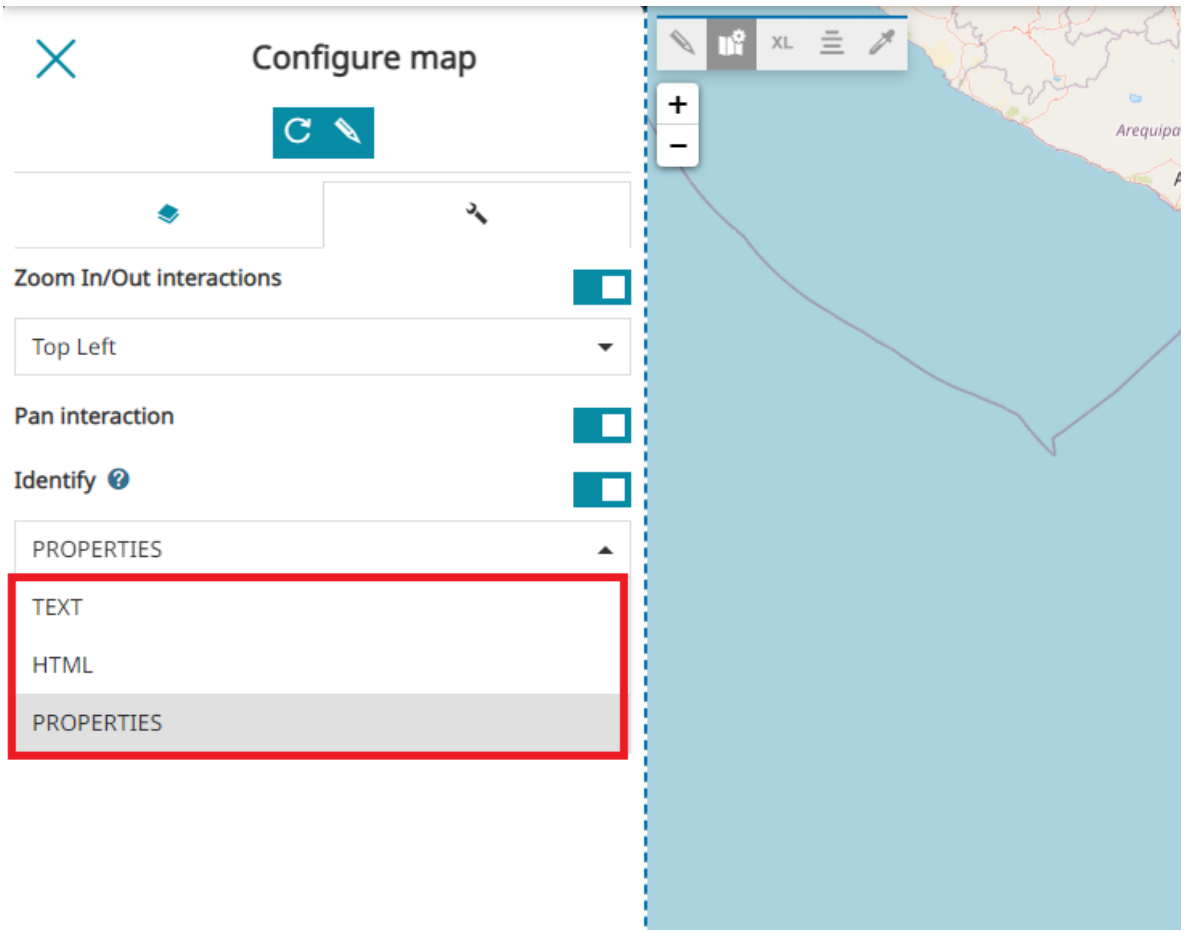
The *Setting section* allows the user to:



- Enable/disable the **Zoom in/out** on the map
- Change the position of the **Zoom in/out** by choosing one of the options available in the dropdown menu

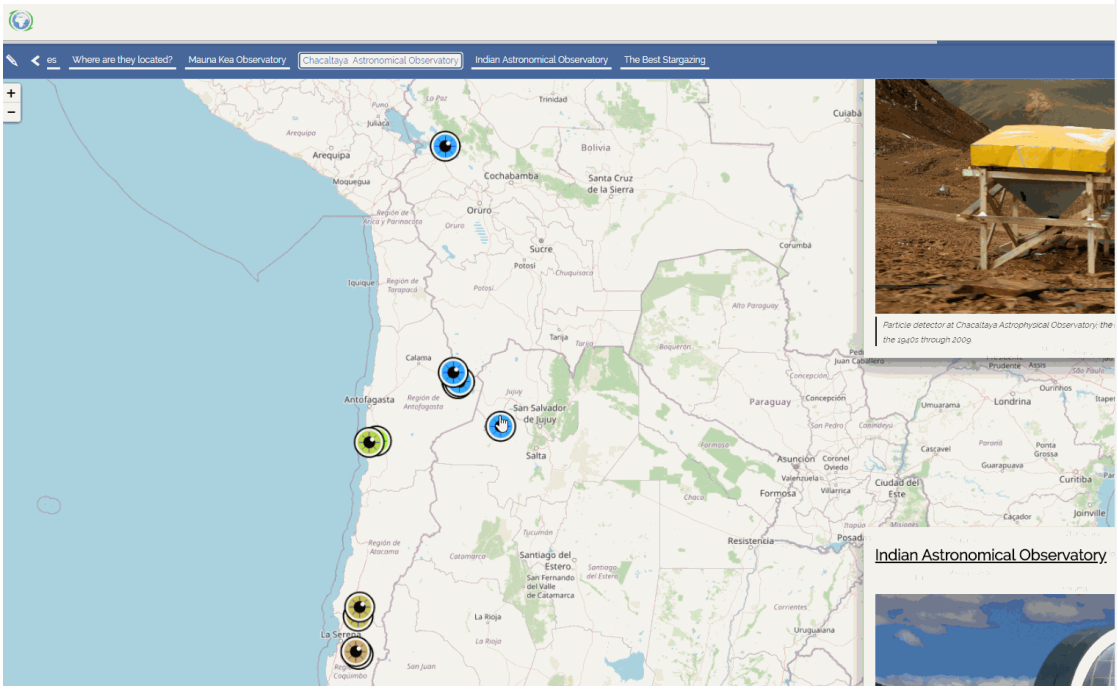


- Enable/disable the **Pan interaction** on the map
- Enable/disable the **Identify** on the map. As reported in the Identify tool section, also for map sections in a story it is possible to enable the *Identify tool* in one of the format supported by MapStore (*TEXT*, *HTML* or *PROPERTIES*)



Note

The Identify request is performed as usual when the user clicks on a layer in the map, as follows:



Advanced map editor

Inside the Map Inline Editor Toolbar the **Advanced map editor** button is also available to allow advanced customization to the map: clicking on that button, a MapStore viewer opens for this purpose.

The available tools to modify the map are the following:

- Adding the **Layers** by using the CATALOG button in the *Option menu* as it is explained in the Catalog Services.
- Adding **Annotations** by clicking on the ANNOTATIONS button in the *Option menu* as it is explained in the Adding Annotations.
- **Import** a map or a vectorial file by clicking on the IMPORT button in the *Option menu* as it is explained in the Import files.
- **Change Background** as it is explained in the Background Selector
- **Edit Layers** by clicking on the Layers button as it is explained in the Table of Contents

Once the advanced map editing is complete, it is possible click on **Apply** to see the final result in the story.


List of Highest Astronomical Observatories
From Wikipedia, the free encyclopedia

Requirements

In this section you can have a glance of the minimum and recommended versions of the tools needed to build/debug/install MapStore

War Installation

You can download a java web container like *Apache Tomcat* from and *Java JRE*

Tool	Link	Minimum	Recommended	Maximum
Java	link	8	9	11[1]
Tomcat	link	8.5	9	9[1]

Debug / Build

These tools needs to be installed (other than **Java** in versions above above):

Tool	Link	Minimum	Recommended	Maximum
npm	link	5	6	6.14.13[2]
NodeJS	link	10	12	14.17.0[2]
Java (JDK)	link	8	9	11[1]
Maven	link	3.1.0	3.6	
python[3]	link	2.7.9	3.7	

Notes

Here some notes about some requirements and reasons for max version indicated, for future improvements and maintenance :

- [1] About Java and Tomcat
  - For execution tested on Java v11.
  - Build with success with v11, only smoke tests passing on v13, errors with v16.(Details on issue #6935)
  - Running with Tomcat 10 causes this issue #7524.
- [2] About NodeJS and NPM:
  - NPM 7 not supported yet.
  - NPM 6.14.15 causes this issue on MapStore project system. No other know issues.
  - *If you are using Node >= 12 you can remove the -max_old_space_size=2048 config for the `compile` script*
- [3] Python is only needed for building documentation.

Running in Production

System requirements

Resource	Minimum	Recommended
Processor	2 Core	2 Core
Memory	2 GB	4 GB

Database

In production a PostgreSQL database is recommended:

Tool	Link	Minimum	Recommended	Maximum
Postgres	link	9.6	13	13

Quick Setup and Run

Clone the repository:

```
git clone https://github.com/geosolutions-it/MapStore2.git
```

If needed, install NodeJS version >= 8 from here, then update npm to version >= 5, using:

```
npm install -g npm
```

Start the demo locally:

```
npm cache clean  # this is useful to prevent errors on Windows
during install

npm install

npm start
```

Then point your preferred browser to http://localhost:8081.

note: This running demo uses https://dev-mapstore.geosolutionsgroup.com/mapstore/ as back-end.

Other useful commands

```
# Run tests
npm test

# run test with hot reload
npm run continuoustest
```

```
#generate test documentation
npm run doctest
```

Quick Build and Deploy

Install latest Maven, if needed, from here (version 3.1.0 is required).

Build the deployable war:

```
./build.sh [version_identifier]
```

Where version_identifier is an optional identifier of the generated war that will be shown in the settings panel of the application.

Deploy the generated `mapstore.war` file (in product/target) to your favourite J2EE container (e.g. Tomcat).

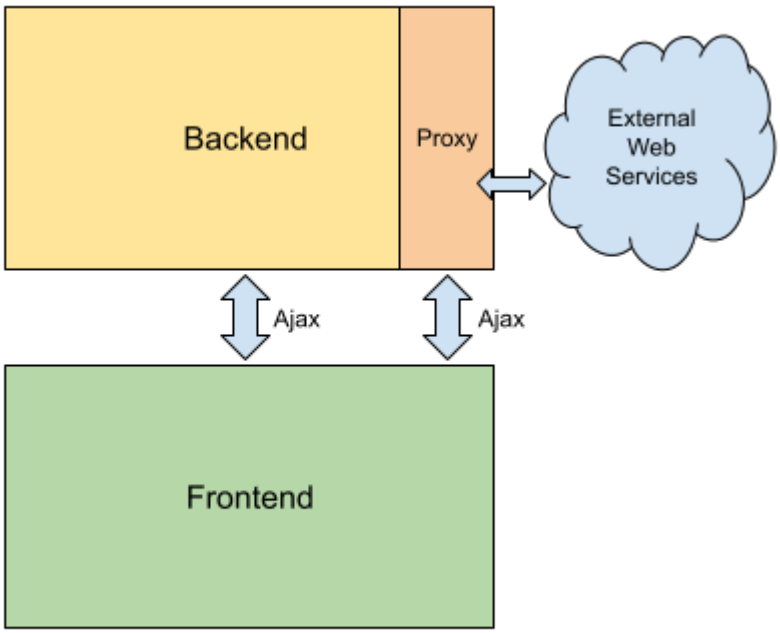
Here you can find how to setup the database.

Infrastructure

MapStore leverages a full separation of concerns between the **backend** and the **frontend**.

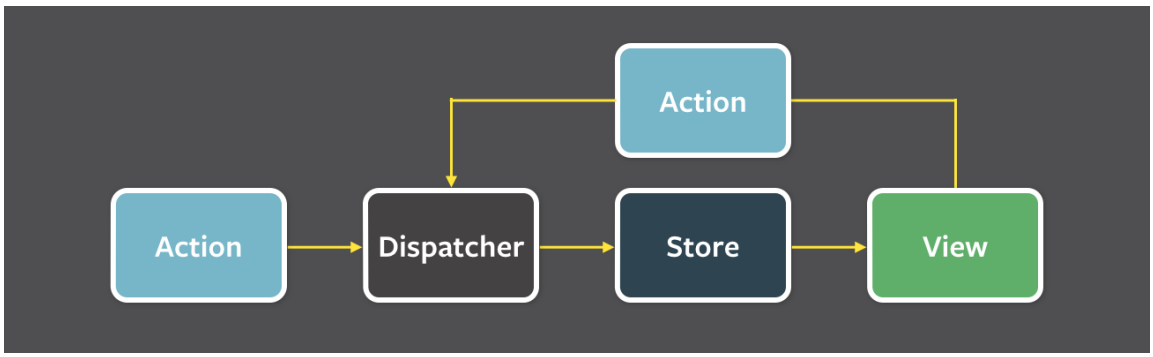
The frontend is a Javascript web application communicating with MapStore own web services using AJAX and external ones through an internal, configurable, *proxy*.

The backend is a suite of web services, developed in Java and deployed into a J2EE container (e.g. Apache Tomcat).

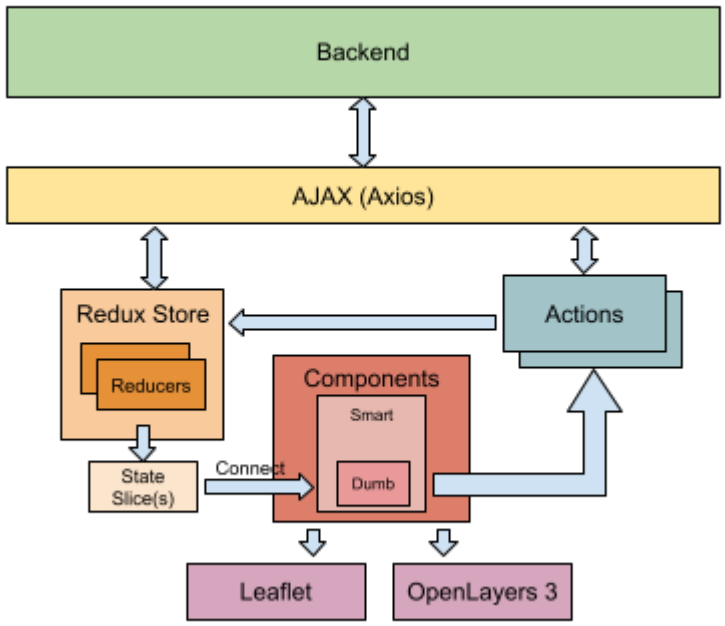


Frontend

The frontend is based on the ReactJS library and the Redux architecture, which is a specific implementation of the Flux architecture.



It allows plugging different mapping libraries (with **Leaflet** and **OpenLayers** as our first implementation targets) abstracting libraries implementation details using ReactJS *web components* and *actions based communication*.

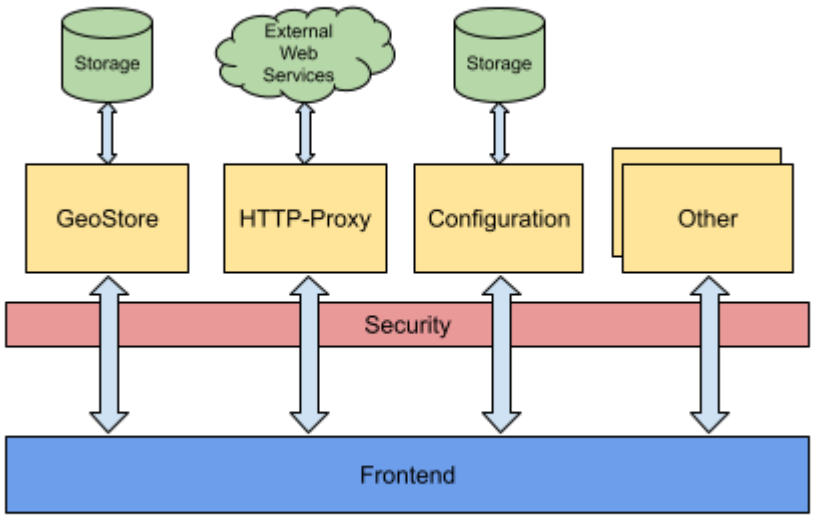


Backend

Backend services include at least (but not only) these ones:

- Generic, configurable, **HTTP-Proxy** to avoid CORS issues when the frontend tries to communicate with external services, based on the GeoSolutions http-proxy project.
- Internal **storage** for non structured resources (json, XML, etc.) based on the GeoSolutions GeoStore project.

- **Configuration** services, to allow full application(s) and services configurability
- **Security** with the ability to configure authentication using an internal or external service, and a flexible authorization policy for services and resources access.



Developing with MapStore

MapStore is both an application and a framework. This guide is both for developers who want to extend MapStore and for those who want to create their custom application using MapStore as a framework.

MapStore as an application

MapStore is 99% client side, and uses some Java back-end services

Back-end mainly consists in services included from external projects (GeoStore, MapFish Print, HTTP-Proxy...) plus some small service owned by MapStore, all written in Java.

Developing with MapStore as an application means to develop directly on the project. You can add plugins or improve the existing code base and, hopefully, send pull requests on GitHub to include your improvements in the main project.

MapStore as a Framework

The recommended way to use MapStore as a framework is to create a project that includes MapStore as a sub-folder. For this purpose we created a script that generates the main folder structure and the necessary files Project Creation Script.

This setup allows to create your application or customizations inside the `js` directory and/or add custom back-end services (the set-up allows to create a project that builds a Java WAR package). Keeping your customization separated and MapStore as a git sub-modules has the followind advantages:

- **Clear separation between the framework and your customization**
- **Easy framework update**: updating the git sub-module.
- **Easy customization of MapStore**: You can fork the project, if you need hard customization. If your customization can be included in MapStore, you can

do a pull request to the main project and work on a branch while waiting the pull request merge.

Folders structure

This is the overall framework folder structure:

```
+-- package.json
+-- pom.xml
+-- build.sh
+-- .editorconfig
+-- Dockerfile
+-- .travis.yml
+-- ...
+-- build       (build related files)
    +-- karma.conf.*.js
    +-- tests.webpack.js
    +-- webpack.config.js
    +-- prod-webpack.config.js
    +-- docma-config.json
    +-- testConfig.json
    +-- ...
+-- java        (java backend modules)
    +-- pom.xml
    +-- services
    +-- web
    +-- printing
+-- translations (i18n localization files)
    |  +-- data.en-US.json
+-- utility (general utility scripts and functions)
    |  +-- eslint
    |  +-- build
    |  +-- projects
    |  +-- translations
+-- web         (frontend module)
    +-- client
    |   +-- index.html (demo application home page)
        +-- plugins (ReactJS smart components with required
reducers)
        +-- components (ReactJS dumb components)
        |   +-- category
        |   |   +-- <component>.jsx (ReactJS component)
        |   |   +-- ...
        |   |   +-- __tests__       (unit tests folder)
        |   |       +-- <component>-test.jsx
        |   +-- ...
        +-- actions    (Redux actions)
```

```
        +-- configs    (JSON config files like
localConfig.json, pluginsConfig.json, new.json,
newgeostory.json, etc)
        +-- epics      (redux-observable epics)
        +-- reducers   (Redux reducers)
        +-- stores     (Redux stores)
           ...
           product (the MapStore main application)
            +...
```

Developing with MapStore

Due to the dual nature of the project (Java backend and JavaScript frontend) building and developing using the MapStore framework requires two distinct set of tools

- Apache Maven for Java
- NPM for JavaScript.

A basic knowledge of both tools is required.

Frontend

To start developing the MapStore framework you have to:

- download developer tools and frontend dependencies locally:

```
npm install
```

After a while (depending on the network bandwidth) the full set of dependencies and tools will be downloaded to the **node_modules** sub-folder.

- start the development instance with:

```
npm start
```

Then point your preferred browser to http://localhost:8081. By default the front-end works using the online dev server as back-end. This configuration is useful for a quick startup, but is not the suggested configuration if you want to develop. To learn how to connect the front-end dev server to a local back-end read the following instructions.

Connect Front-end to local back-end

By default `npm start` uses the online dev server as a backend. This configuration needs to be changed to develop locally in order to access all the functionalities.

To use a local back-end you have to:

- **Remove auth configuration dedicated to GeoServer** from `localConfig.json --> authenticationRules` (it provides the MapStore/ GeoServer integration for dev-server, that is not present in your local back-end)

```
        "urlPattern": ".*geostore.*",
        "method": "bearer"
      }, {
        "urlPattern": ".*rest/config.*",
        "method": "bearer"
-       },
-       {
-         "urlPattern": "http(s)?\\:\\/\\/gs-stable\\.geo-
solutions\\.it\\/geoserver/.*",
-         "authkeyParamName": "authkey",
-         "method": "authkey"
    }],
```

- **Run the back-end locally**. See the dedicated section in this page
- **Setup dev-server to use the local back-end**, applying this changes to `buildConfig.js` --> devServer configuration. (the configuration of the port and path depends on how you configured the local back-end.

```
devServer: {
        // ...
        proxy: {
            '/rest/': {
                target: "http://localhost:8080/mapstore" //
port 8080, mapstore path
            },
            '/proxy': {
```

```
                target: "http://localhost:8080/mapstore", //
port 8080, mapstore path
                secure: false
            }
        }
    },
    // ...
```

- **re-run** `npm start`

Debugging the frontend

The development instance uses file watching and live reload, so each time a MapStore file is changed, the browser will reload the updated application.

Use your favorite editor / IDE to develop and debug on the browser as needed.

We suggest to use one of the following:

- Visual Studio Code with the following plugins:
- ESLint `dbaeumer.vscode-eslint`
- EditorConfig for VSCode `editorconfig.editorconfig`
- Atom with the following plugins:
- editorconfig
- linter
- linter-eslint
- react
- lcovinfo
- minimap & minimap-highlight-selected
- highlight-line & highlight-selected
- Sublime Text Editor with the following plugins:
- Babel
- Babel snippets
- Emmet

Redux Dev Tools

When you are running the application locally using `npm start` you can debug the application with redux dev tools using the flag ?debug=true

```
http://localhost:8081/?debug=true#/
```

It also integrates with the browser's extension, if installed.

This way you can monitor the application's state evolution and the action triggered by your application.

Frontend unit tests

To run the MapStore frontend test suite you can use:

`npm test`

You can also have a continuously running watching test runner, that will execute the complete suite each time a file is changed, launching:

`npm run continuoustest`

Usually during the development you may need to execute less tests, when working on some specific files.

You can reduce the tests invoked in `npm run continuoustest` execution by editing the file `tests.webpack.js` and modifying the directory (`/web`) and/or the regular expression that intercept the files to execute.

To run ESLint checks launch:

`npm run lint`

To run the same tests Travis will check (before a pull request): `npm run travis`

More information on frontend building tools and configuration is available here

Back-end

In order to have a full running MapStore in development environment, you need to run also the back-end java part locally. In this section you will find how to start the back-end and how to develop with it.

Defaults Users and Database

Running MapStore back-end locally, on start-up you will find the following users:

- `admin` , with ADMIN role and password `admin`
- `user` with USER role with password `user`

You can login as `admin` to set-up new users and access to all the features reserved to `ADMIN` users.

The database used by default in this mode is H2 on disk. You can find the files of the database in the directory `webapps/mapstore/` starting from your execution context. Check how to set-up database in the dedicated section of the documentation.

Running Back-end

When we say "running the back-end", in fact we say that we are running some sort of a whole instance of MapStore locally, that can be used as back-end for your front-end dev server, or for debugging of the back-end itself.

Embedded tomcat

MapStore is configured to use a tomcat maven plugin-in to build and run mapstore locally. To use it you have to:

- make sure to run at least once `mvn install` in the root directory, to make `mapstore-product` artifact available.
- `cd product` directory
- run `mvn cargo:run`

Your local back-end will now start at http://localhost:8080/mapstore/. If you want to change the port you can edit the dedicated entry in `product/pom.xml`, just remember to change also the dev-server proxy configuration on the front-end in the same way.

Local tomcat instance

If you prefer, or if you have some problems with `mvn cargo:run`, you can run MapStore back-end in a tomcat instance instead of using the embedded one. To do so, you can :

- download a tomcat standalone here and extract to a folder of your choice
- To generate a war file that will be deployed on your tomcat server, go to the root of the Mapstore project that was git cloned and run `./build.sh`. This might take some time but at the end a war file named `mapstore.war` will be generated into the `product/target` folder.
- Copy the `mapstore.war` and then head back to your tomcat folder. Look for a `webapps` folder and paste the `mapstore.war` file there.
- To start tomcat server, go to the terminal, `cd` into the root of your tomcat extracted folder and run `./bin/startup.sh` (unix systems) or `./bin/startup.bat` (Windows). The server will start on port `8080` and Mapstore will be running at `http://localhost:8080/mapstore`. For development purposes we're only interested in the backend that was started on the tomcat server along with Mapstore.

Even in this case you can connect your front-end to point to this instance of MapStore.

Debug

To run or debug the server side part of MapStore we suggest to run the back-end in tomcat (embedded or installed) and connect in remote debugging to it. This guide explains how to do it with Eclipse. This procedure has been tested with Eclipse Luna.

Enable Remote Debugging

for embedded tomcat you can configure the following:

```
# Linux
export MAVEN_OPTS="-Xdebug -Xnoagent -Djava.compiler=NONE -
Xrunjdwp:transport=dt_socket,address=4000,server=y,suspend=n"
```

```
# Windows
set MAVEN_OPTS=-Xdebug -Xnoagent -Djava.compiler=NONE -
Xrunjdwp:transport=dt_socket,address=4000,server=y,suspend=n
```

then start tomcat

```
cd product
mvn cargo:run
```

For your local tomcat, you can follow the standard procedure to debug with tomcat.

Setup eclipse project

- Run eclipse plugin

```
mvn eclipse:eclipse
```

- Import the project in eclipse from **File --> Import**
- Then select Existing project into the Workspace
- Select root directory as MapStore root (to avoid eclipse to iterate over all node_modules directories looking for eclipse project)
- import all projects

Start Debugging with eclipse

- Start Eclipse and open **Run --> Debug Configurations**
- Create a new Remote Java Application selecting the project "mapstore-product" setting:
- host localhost
- port 4000
- Click on *Debug* Remote debugging is now available.

> **NOTE** With some version of eclipse you will have to set `suspend=y` in mvn options to make it work. In this case the server will wait for the debug connection at port 4000 (`address=4000`)

Building and deploying

Maven is the main tool for building and deploying a complete application. It takes care of:

- building the java libraries and webapp(s)
- calling NPM as needed to take care of the frontend builds
- launching both backend and frontend test suites
- creating the final war for deploy into a J2EE container (e.g. Tomcat)

To create the final war, you have several options:

- full build, including submodules and frontend (e.g. GeoStore)

```
./build.sh [version_identifier] [profiles]
```

Where `version_identifier` is an optional identifier of the generated war that will be shown in the settings panel of the application and profiles is an optional list of comma delimited building profiles (e.g. `printing`, `ldap`)

- fast build (will use the last compiled version of submodules and compiled frontend)

```
mvn clean install -Dmapstore2.version=[version_identifier] [profiles]
```

- binary build (produces also the binary)

```
mvn clean install -Dmapstore2.version=[version_identifier] -Pbinary
```

Building the documentation

MapStore uses JSDoc to annotate the components, so the documentation can be automatically generated using docma. Please see jsdoc for further information about code documentation.

Refer to the existing files to follow the documentation style:

- actions
- reducers
- components
- epics
- plugins

To install docma:

```
npm install -g docma
```

While developing you can generate the documentation to be accessible in the local machine by:

```
npm run doctest
```

The resulting doc will be accessible from `http://localhost:8081/mapstore/docs/`

For the production deploy a different npm task must be used:

```
npm run doc
```

The documentation will be accessible from the */mapstore/docs/* path

The generated folders can be removed with:

```
npm run cleandoc
```

Build the mkdocs and generate md files to test in local machine by:

```
npm run build-doc
```

Start the built-in dev-server of mkdocs to preview and test documentation live by:

```
npm run serve-doc
```

Understanding frontend building tools

Frontend building is delegated to NPM and so leverages the NodeJS ecosystem.

In particular:

- a **package.json** file is used to configure frontend dependencies, needed tools and building scripts
- **babel** is used for ES6/7 and JSX transpiling integrated with the other tools (e.g. webpack)
- **webpack-dev-server** is used to host the development application instance
- **mocha**/**expect** is used as a testing framework (with BDD style unit-tests)
- **webpack**: as the bundling tool, for development (see webpack.config.js), deploy (see prod-webpack.config.js) and test (see test.webpack.js)
- **karma** is used as the test suite runner, with several plugins to allow for custom reporting, browser running and so on; the test suite running is configured through different configuration files, for **single running** or **continuous testing**
- **istanbul**/**coveralls** are used for code coverage reporting

Index of main npm scripts

Command	Description
`npm install`	download dependencies and init developer environment
`npm start`	start development instance
`npm run compile`	run single build / bundling
`npm test`	run test suite once
`npm run continuoustest`	run continuous test suite running (useful during developing)
`npm run lint`	run ESLint checks
`npm run mvntest`	run tests from Maven
`npm run travis`	run the test build used for travis

Including the printing engine in your build

The printing module is not included in official builds by default.

To build your own version of MapStore with the this module, you can use the **printing** profile running the build script:

```
./build.sh [version_identifier] printing
```

For more information or troubleshooting about the printing module you can see the dedicated section

Main Frontend Technologies

The main tecnologies used on the mapstore 2 are:

- ReactJS (View)
- Redux (state management)

ReactJS

ReactJS 0.16.x is used to develop MapStore. The main purpose of ReactJS is to allow writing the **View** of the application, through the composition of *small components*, in a *declarative way*.

Components are written using a "templating" language, called **JSX**, that is a sort of composition of HTML and Javascript code. The difference between JSX and older approaches like *JSP* is that JSX templates are mixed with Javascript code inside javascript files.

ReactJS component example

Component definition:

```
class MyComponent extends React.Component {
   render() {
       return <h1>{this.props.title}</h1>;
   }
}
```

Component usage:

```
React.render(<MyComponent title="My title"/>, document.body);
```

Properties, State and Event handlers

Components can define and use **properties**, like the title one used in the example. These are immutable, and cannot be changed by component's code.

Components can also use **state** that can change. When the state changes, the component is updated (re-rendered) automatically.

```
class MyComponent extends React.Component {
   state = {
      return {
         title: 'CHANGE_ME'
      };
   };
   changeTitle = () => {
      this.setState({
         title: 'CHANGED'
      });
   };
   render() {
       return <h1 onClick={this.changeTitle}>{this.state.title}
</h1>;
   }
}
```

In this example, the initial state includes a title property whose value is `CHANGE_ME`.

When the h1 element is clicked, the state is changed so that title becomes `CHANGED`.

The HTML page is automatically updated by ReactJS, each time the state changes (each time this.setState is called). For this reason we say that JSX allows to declaratively describe the View for each possible application state.

Lifecycle hooks

Components can re-define some lifecycle methods, to execute actions in certain moments of the component life. Lifecycle API is changed in react 16 so please refer to the official documentation.

Redux

Redux, and its companion react-redux are used to handle the application state and bind it to ReactJS components.

Redux promotes a unidirectional dataflow (inspired by the Flux architecture) and immutable state transformation using reducers, to achieve predictable and reproducable application behaviour.

A single, global, **Store** is delegated to contain all the application state.

The state can be changed dispatching **Actions** to the store.

Each action produces a new state (the state is never changed, a new state is produced and that is the new application state), through the usage of one or more **reducers**.

(Smart) Components can be connected to the store and be notified when the state changes, so that views are automatically updated.

Actions

In Redux, actions are actions descriptors, generated by an action creator. Actions descriptors are usually defined by an **action type** and a set of parameters that specify the action payload.

```
const CHANGE_TITLE= 'CHANGE_TITLE';

// action creator
function changeTitle(newTitle) {
    return {
        type: CHANGE_TITLE,
        title: newTitle
    };
}
```

Reducers

Reducers are functions that receive an action and the current state and:

- produce a new state, for each recognized action
- produce the current state for unrecognized actions
- produce initial state, if the current state is undefined

```
function reducer(state = {title: "CHANGE_ME"}, action) {
    switch (action.type) {
        case CHANGE_TITLE:
            return {title: action.title};
        default:
            return state;
    }
}
```

Store

The redux store combines different reducers to produce a global state, with a slice for each used reducer.

```
var rootReducer = combineReducers({
   slice1: reducer1,
   slice2: reducer2
});
var initialState = {slice1: {}, slice2: {}};

var store = createStore(rootReducer, initialState);
```

The Redux store receives actions, through a dispatch method, and creates a new application state, using the configured reducers.

```
store.dispatch(changeTitle('New title'));
```

You can subscribe to the store, to be notified whenever the state changes.

```
store.subscribe(function handleChange() {});
```

Redux Middlewares

Redux data flow is synchronous. To provide asynchronous functionalities (e.g. Ajax) redux needs a middleware. Actually MapStore uses 2 middlewares for this purpose:

- Redux thunk (going to be fully replaced by redux-observable)
- Redux Observable

Redux thunk

This middleware allows to perform simple asynchronous flows by returning a function from the action creator (instead of a action object).

```
// action creator
function changeTitleAsync() {
    return (dispatch, getState) => {
        myAsyncPromise.then( (newTitle) => {
            dispatch({
                type: CHANGE_TITLE,
                title: newTitle
            };)
        });
    }
}
```

This middleware is there from the beginning of the MapStore history. During the years, some better middlewares have been developed for this purpose. We want to replace it in the future with redux-observable.

Redux Observable and epics

This middleware provides support for side-effects in MapStore using rxjs. The core object of this middleware is the `epic`

```
```

```
function (action$: Observable<Action>, store: Store):
Observable<Action>;
```

The epic is a function that simply gets as first parameter an `Observable` (stream) emitting the actions emitted by redux. It returns another `Observable` (stream) that emits actions that will be forwarded to redux too.

So there are 2 streams:

- Actions in
- Actions out

A simple epic example can be the following:

```
const pingEpic = action$ =>
  action$.filter(action => action.type === 'PING')
    .mapTo({ type: 'PONG' });
```

Every time a 'PING' action is emitted, the epic will emit also the 'PONG' action.

See :

- Introduction to RxJS for MapStore Developers
- redux-observable site
- rxjs Observable as a reference for methods
- setting up the middleware to integrate epics with your store

Redux and ReactJS integration

The **react-redux** library can be used to connect the Redux application state to ReactJS components.

This can be done in the following way:

- wrap the ReactJS root component with the react-redux **Provider** component, to bind the Redux store to the ReactJS view

```
React.render(
    <Provider store={store}>
        {() => <App />}
    </Provider>,
    document.getElementById('container')
);
```

- explicitly connect one or more (smart) components to a all or part of the state (you can also transform the state and have computed properties)

```
connect(function(state) {
    return {
        title: state.title,
        name: state.namespace + '.' + state.name,
    };
})(App);
```

The connected component will get automatic access to the configured slice through properties:

```
function render() {
   return <div><h1>{this.props.title}</h1><p>{this.props.name}</
p></div);
}
```

Plugins Architecture

MapStore fully embraces both ReactJS and Redux concepts, enhancing them with the **plugin** concept.

A plugin in MapStore is a smart ReactJS component that is:

- **connected** to a Redux store, so that some properties are automatically wired to the standard MapStore state
- **wired** to standard actions for common events

In addition a plugin:

- declares some **reducers** that need to be added to the Redux store, if needed
- declares some **epics** that need to be added to the redux-observable middleare, if needed
- is fully **configurable** to be easily customized to a certain level

Building an application using plugins

To use plugins you need to:

- declare available (required) plugins, properly requiring them from the root application component
- load / declare plugins configuration
- create a store that dynamically includes plugins required reducers
- use a PluginsContainer component as the container for your plugins enabled application slice (can be the whole application or just a part of it)

Declare available plugins

Create a plugins.js file where you declare all the needed plugins:

plugins.js:

```
module.exports = {
    plugins: {
        MyPlugin: require('../plugins/My')
    },
    requires: {}
};
```

Load / Create plugins configuration object

Use pluginsConfig.json to configure your plugins.

pluginsConfig.json:

```
{
    ...
    "standard": [
        {
            "name": "Map",
            "cfg": {
                "zoomControl": false,
                "tools": ["locate"]
            }
        },
        ...
    ],
    ...
}
```

Declare a plugins compatible Redux Store

Create a store that properly initializes plugins reducers and epics (see standardStore.js) :

store.js:

```
const {combineReducers} = require('../utils/PluginsUtils');
const {createDebugStore} = require('../utils/DebugUtils');

module.exports = (plugins) => {
  const allReducers = combineReducers(plugins, {
```

```
    ...
  });
  return createDebugStore(allReducers, {});
};
```

Use a PluginContainer to render all your plugins

In the root application component require plugins declaration and configuration and use them to initialize both the store and a PluginsContainer (see our PluginContainer.jsx):

App.jsx:

```
const {pluginsDef} = require('./plugins.js');
const pluginsCfg = require('./pluginsConfig.json');

const store = require('./store')(pluginsDef);

const plugins = PluginsUtils.getPlugins(pluginsDef);

ReactDOM.render(<PluginsContainer plugins={plugins}
mode="standard" pluginsConfig={pluginsCfg}/>, ...container...);
```

Developing a plugin

An example is better than a thousand words:

My.jsx:

```
// this is a dumb component
const MyComponent = require('../components/MyComponent');
const {connect} = require('../utils/PluginsUtils');

// let's wire it to state and actions
const MyPlugin = connect((state) => ({
   myproperty: state.myreducer.property
}), {
   myaction
})(MyComponent);
```

```
// let's export the plugin and a set of required reducers
const myreducer = require('../reducers/myreducer');
module.exports = {
   MyPlugin,
   reducers: {myreducer}
};
```

Internationalization

MapStore offers the support for internationalization (I18N). To provide this functionality MapStore uses react-intl. In this section you can find which configuration and JS files are involved in the I18N system.

How MapStore chooses the current language

MapStore first checks the browser's language. If it is not supported, MapStore will be visible in english, if present, or the first language available. Anyway the locale can be forced using a flag locale=codeLang where codeLang can be one en,it,de... e.g.

```
localhost:8081/?locale=en#/
```

A user can change the selected language from UI. MapStore will load the proper files to update the page localized in the selected language.

Configuration files

To provide support to a specific language MapStore need to have the necessary setup in the LocaleUtils.js file (see below [section for details about to configure this file]). In addition you need the proper translations files.

Let's imagine that the variable code is 'en', CODE is 'EN' standing for english. For each language you need to have **messages file** containing the localized strings, a **flag image** to identify the language and some **html fragments** (optional) for some specific plugins.

- Messages: located in `web\client\translations` folder. For each language there is a json file named data.code-CODE.json. e.g. `data.en-EN.json` .
- Flags: located in `web\client\components\I18N\images\flags` folder. For each language flag image named code-CODE.png of 16px x 11px is required.

- Fragments: actually only for cookies policy .... (required only if the Cookie plugin is present) located in `web\client\translations\fragments\cookie` folder and named cookieDetails-code-CODE.html. We recommend to add it for any language you want to support at least by copying the english version.

How to configure supported languages in MapStore

You can configure MapStore to provide to the user only a restricted list of selectable languages by setting "initialState.defaultState.locales" variable in `localConfig.json` . e.g :

```
"defaultState":
{
    "locales": {
        "supportedLocales": {
            "en": {
                "code": "en-US",
                "description": "English"
            },
            "it": {
                "code": "it-IT",
                "description": "Italiano"
            }
        }
    }
}
```

Setting locales in localConfig.json file is doable only for supported locales present in LocaleUtils.js. The default behavior is to use those already configured in "supportedLocales" object. You can customize the messages by editing the data.code-CODE.json files.

The `locale` property determines the language to use for the application. If not specified, the language will be selected checking the browser's locale first. If the browser locale is not supported, MapStore will select the first language available in `supportedLocales` .

Example of localConfig.json with the optional locale property.

```
{
    "locale": "it-IT", // locale code

    "defaultState": {
        "locales": {
            "supportedLocales": {
                "en": {
                    "code": "en-US",
                    "description": "English"
                },
                "it": {
                    "code": "it-IT",
                    "description": "Italiano"
                }
            }
        }
    }
}
```

The property `locale` could be useful inside custom application where the locale is stored in other sources rather than using the browser language:

```
// example
import { getConfigProp } from '@mapstore/framework/utils/
ConfigUtils';
import cookies from 'js-cookie';

// ...
const locale = cookies.get('app_locale'); // locale code it-IT
for example
if (locale) {
  setConfigProp('locale', locale);
}
// ...
```

How to add a new language

Let's say we want to add the russian language. In order to add a new language to MapStore you need to follow these steps:

- *Update the* `localConfig.json` *file in* `web\client` *folder adding the new language entry*: Add the following in the `initialState.defaultState.locales` object

```
"ru": {
    code: "ru-RU",
    description: "Российский"
}
```

- *Update the LocaleUtils.js file in* `web\client\utils` : add a param in the ensureIntl() function like and the relative require i.e: `'intl/locale-data/jsonp/ru.js'`

```
require('intl/locale-data/jsonp/ru.js');
```

- *update the addLocaleData() call with the new locale obj* i.e.:

```
const ru = require('react-intl/locale-data/ru');
addLocaleData([...en, ...it, ...fr, ...de, ...es, ...ru]);
```

- *add the flag image for the selected language* inside `web\client\components\I18N\images\flags` naming it `ru-RU.png`
- *add the new translations file* inside `web\client\translations` naming it `data.ru-RU.json` (remember to change the locale property of this file into `ru-RU`)
- *create a fragment related to the cookie module* inside `web\client\translations\fragments\cookie` naming it `cookieDetails-ru-RU.html`

Custom Dependencies

Mapstore has some custom dependencies in order to fix bugs not integrated in the official libraries yet. All these customized libraries are available on npm registry.

Here is a list of customizations:

library	version	issue	reason	github
wkt-parser	1.2.1	#2175	Fixes axis order recognition. For this reason we customized it with "@geosolutions@wkt-parser 1.2.2"	https://github.com/geosolutions-it/wkt-parser/tree/release
proj4	2.4.5-alpha	#2175	Fixes axis order recognition. For this reason we customized it with "@geosolutions@proj4 2.4.6" and its wkt-parser dependency with "@geosolutions@wkt-parser 1.2.2". Note that shpjs will use this customized version of proj4 and wkt-parser	https://github.com/geosolutions-it/proj4js/tree/release_2.4.6
react-joyride	1.10.1		is a re-publish on npm of a fix made here , we therefore are using "@geosolutions@react-joyride 1.10.2"	https://github.com/geosolutions-it/react-joyride/tree/release
mocha	6.2.0-uncaught	#3693	Customized in order to make some test run. More in detail, we removed uncaught exceptions handler because it was making some test failing (waiting for a better solution). Published "@geosolutions/mocha 6.2.1-3". mocha is being moved from node_modules/@geosolutions/mocha to node_modules/mocha in order to make the test be runnable	https://github.com/geosolutions-it/mocha/tree/release_v6.2.1
jsdoc	3.4.3	#1978	ES6 syntax not parsed by Docma, so we published "@geosolutions/jsdoc 3.4.4" with other related dependencies also on our npm, like acorn-jsx, espree and tv4	https://github.com/geosolutions-it/jsdoc/tree/release
acorn-	4.0.1	#1978	Added support for instance	https://

Aliases

Only proj4 and react-joyride are using aliases in order to maintain original webpack requires like:

- const proj4 = require("proj4");
- const joyride = require('react-joyride').default;

see this for current status of aliases

More info

Here you can find more information about customization

Styling and Theming

The look and feel is completely customizable either using one of the included themes, or building your own. Themes are built using less.
You can find the default theme here: https://github.com/geosolutions-it/MapStore2/tree/master/web/client/themes/default

Theme Structure

```
.
+-- themes/
|   +-- theme-name/
|       +-- icons/
|           +-- icons.eot
|           +-- icons.svg
|           +-- icons.ttf
|           +-- icons.woff
|           +-- icons.woff2
|       +-- img/
|       +-- less/
|           +-- mixins/
|               +-- bootstrap.less
|               +-- css-properties.less
|               +-- theme.less
|           +-- mapstore.less
|           +-- common.less
|           +-- style-module.less
|           +-- .less files for all the other modules
|       +-- base.less
|       +-- bootstrap-theme.less
|       +-- bootstrap-variables.less
|       +-- icons.less
|       +-- ms2-theme.less
|       +-- ms-variables.less
|       +-- theme.less
|       +-- variables.less
```

`theme.less` is the entry point for all the main imports and it needs to be properly required in `buildConfig.js` or in your `webpack.config.js` in the themeEntries.

`theme.less` imports

file	description
base.less	contains a declaration of font colors and background defined for data-ms2-container attribute which is usually the body tag
icons.less	contains font-face declaration for glyphs, it extends the bootstrap glyphicons to use custom MapStore icons
bootstrap-theme.less	contains all the less style for bootstrap components
ms2-theme.less	contains all the less style for MapStore components
variable.less	contains the import of mapstore variables and the override of bootstrap variables

below an example of entry configuration:

```
entry: {
    ...other entries,
    'themes/theme-name': path.join(__dirname, 'path-to', 'theme-name', 'theme.less')
},
```

MapStore uses a `themeEntries` function to automatically create the entries for default themes that can be found under the `web/client/themes` directory

```
const themeEntries = require('./themes.js').themeEntries;

entry: {
    ...other entries,
    ...themeEntries
},
```

Default themes in `web/client/themes` directory are useful to have an overview of the structure described above.

Note: we suggest to place the theme folder inside a `themes` directory for MapStore project

Structure of .less files

Each less file that represent a MapStore plugin or component is composed by two sections:

- **Theme section** includes all the styles and classes that should change based on css variables. All the new declared selector must be included in a special function called `#ms-components-theme` . The `#ms-components-theme` function provide access to all the available variables of the theme via the `@theme-vars` argument.
- **Layout section** includes all the styles and classes that should not change in a simple customization.

Example:

```
// **************
// Theme
// **************
#ms-components-theme(@theme-vars) {
    // here all the selectors related to the theme
    // use the mixins declared in the web/client/theme/default/
less/mixins/css-properties.less
    // to use one of the @theme-vars

    // eg: use the main background and text colors to paint my
plugin
    .my-plugin-class {
```

```
            .color-var(@theme-vars[main-color]);
            .background-color-var(@theme-vars[main-bg]);
        }
    }

    // **************
    // Layout
    // **************

    // eg: fill all the available space in the parent container
    with my plugin
    .my-plugin-class {
        position: absolute;
        height: 100%;
        width: 100%;
    }

    // here
```

ms-variables.less

MapStore uses basic less variables to change theme colors, buttons sizes and fonts. It possible also to override bootstrap less variable for advanced customization. Basic variables can be found in the ms-variable.less file

New declarations in MapStore should have the following structure:

global: `@ms-rule-value`

local: `@ms-name-of-plugin--rule-value`

- `@ms` suffix for MapStore variable
- `name-of-plugin` for local variable it's important to write the name of plugin in kebab-case
- `rule-value` value to use in compiled CSS, some examples:
    - `color` generic color variable
    - `text-color` color for text
    - `background-color` color for background
    - `border-color` color for border

less/ directory

The `less/` directory contains all the modules needed to create the final CSS of MapStore.

Each file in this directory is related to a specific plugin or component and the files are named based on the plugin's name are referring to.

`common.less` file can be used for generic styles.

inline styles

Inline styles should be applied only for values that change dynamically during the lifecycle of the application, all others style should be moved to the related .less file.

The main reason of this choice is to allow easier overrides of styles in custom projects.

Add New Theme

To support a new theme for mapstore product:

1. create a new folder in the themes folder with the name of your theme
2. create less files in the folder (at least `theme.less`, as the main file and `variables.less`, to customize standard variables)
3. add the new theme to the index file, with the id corresponding to the theme folder name

If you are not using themeEntries a new entry needs to be added in the `buildConfig.js`

You can then switch your application to use the theme adding a new section in the `appConfig.js` file:

```
  initialState: {
      defaultState: {
```

```
        ...
        theme: {
            selectedTheme: {
                id: <your theme id>
            }
        },
        ...
    }
}
```

Custom Theme for project

In a mapstore project normally the theme configuration is placed in the `themes/` directory

Styles can be overridden declaring the same rules in a less module placed in a new project.

Below steps to configure a custom theme and override styles:

- add the following files to the themes folder of the project:

```
.
+-- themes/
|   +-- default/
|       +-- less/
|           +-- my-custom-module.less
|       +-- theme.less
|       +-- variables.less
```

- import in `theme.less` all the needed less module

```
@import "../../MapStore2/web/client/themes/default/theme.less";
@import "./variables.less";
@import "./less/my-custom-module.less";
```

- update webpack configuration to use the custom style (webpack.config.js, prod-webpack.config.js)

```
    {
        '__PROJECTNAME__': path.join(__dirname, "js", "app"),
        '__PROJECTNAME__-embedded': path.join(__dirname,
"MapStore2", "web", "client", "product", "embedded"),
        '__PROJECTNAME__-api': path.join(__dirname,
"MapStore2", "web", "client", "product", "api")
    },
-   themeEntries,
+   {
+       "themes/default": path.join(__dirname, "themes",
"default", "theme.less")
+   },
    ...
```

- update `variables.less` to override existing variables

```
/* change primary color to blue */
@ms-primary: #0000ff;
```

- update `my-custom-module.less` to override existing rules or add new rules

```
/* change the background color of the page*/
.page {
    background-color: #d9e6ff;
}
```

Custom Theme for contexts

You can configure a list of themes to be used inside a context.

In order to do that you have to:

- create the themes in the `themes/` folder as described below
- edit **ContextCreator** plugin in the `localConfig.json`

example

```
```

```
{
    "name": "ContextCreator",
    "cfg": {
        "documentationBaseURL": "https://
mapstore.geosolutionsgroup.com/mapstore/docs/api/plugins",
        "backToPageDestRoute": "/context-manager",
        "backToPageConfirmationMessage": "contextCreator.undo",
        "themes": [{
                "id": "complete-theme-override",
                "type": "link",
                "href": "dist/themes/complete-theme-
override.css",
                "defaultVariables": {
                    "ms-main-color": "#000000",
                    "ms-main-bg": "#FFFFFF",
                    "ms-primary-contrast": "#FFFFFF",
                    "ms-primary": "#078aa3",
                    "ms-success-contrast": "#FFFFFF",
                    "ms-success": "#398439"
                }
            },
            {
                "id": "partial-theme-override",
                "type": "link",
                "href": "dist/themes/partial-theme-override.css"
            },
            {
                "id": "only-css-variables",
                "type": "link",
                "href": "dist/themes/only-css-variables.css"
            }
        ],
        "basicVariables": {
            "ms-main-color": "#000000",
            "ms-main-bg": "#FFFFFF",
            "ms-primary-contrast": "#FFFFFF",
            "ms-primary": "#078aa3",
            "ms-success-contrast": "#FFFFFF",
            "ms-success": "#398439"
        }
    }
}
```

for each theme you can define: - **id** id of the theme equal to its name - **type** values can be - **link** will require a href property - **href** path to find the css once built - **defaultVariables** variables of the theme used to initialize the pickers (optional)

basicVariables these are the variables used as default values if a theme is not selected (optional)

Suggested ways to create a custom theme for a context

Complete theme override

This example will create a complete css file and is not recommended if you want a light version and you just need to customize the variables (for this check next paragraph)

Add the following files to the themes folder of the project

```
+-- themes/
|   +-- theme-name/
|       +-- theme.less
|       +-- variables.less
```

in `theme.less` put

```
/*
 * This example will contain a complete mapstore theme with
some customization
 * it will be selectable inside context theme step selector
*/

/*
 * it includes the main theme and this will recompile the whole
theme
*/
@import "../../MapStore2/web/client/themes/default/theme.less";

/*
```

```
 * it includes some changes to css variables
*/
@import "./variables.less";

/*
* Note: You can always expand it with new less/css rules
*/
```

in `variables.less` you can put the mapstore variables customizations

```
/*
 * A variable that will override the default css one
*/
@ms-primary: #2E13FE;
```

Only css variables

This example is perfect if you just want to customize a few colors of the theme

```
+-- themes/
|   +-- theme-name/
|       +-- theme.less
|       +-- variables.less
```

in `theme.less` put

```
/*
 * This example is the lightest version of all three examples
 * it will be selectable inside context theme step selector
 * this examples is limited to changing the css variables only,
 * but you can always expand it as we did for partial-theme-
override
*/

/*
 * This will import as (reference) https://lesscss.org/features/
#import-atrules-feature-reference
 * It's used to import external files, but without adding the
imported styles
 * to the compiled output unless referenced.
 *
```

```
*/
@import (reference) "../../MapStore2/web/client/themes/default/
theme.less";

/*
 * it includes some changes to css variables
*/
@import "./variables.less";

/*
 * this will create only one class with the :root selector
inside
 * it's important to place the variable overrides before
calling the css-variable mixin generator
 * which is called .get-root-css-variables
*/
.get-root-css-variables(@ms-theme-vars);

/*
* Note: You can always expand it with new less/css rules
*/
```

In the `variables.less` you can do put your variable customizations

partial theme override

```
+-- themes/
|   +-- theme-name/
|       +-- less/
    |       +-- plugin-name.less
```

```
/*
 * We can use this method when we want to customize some part
of the theme
 * without the need to include the theme in its completeness
*/

/*
 * here you can apply some other overrides, like the size of
thumbnails for backgrounds
*/
@import "./less/drawer-menu.less";
```

```
/*
* Note: You can always expand it with new less/css rules
*/
```

Note: These three styles are an example on how is possible to approach on the mapstore customizations. You could extend/combine them together to create a more complex theme.

Tips

- When you develop locally
- and you want to reduce the building time
- and you don't need themes that are not the default theme
- then you can comment this in the `webpack-config.js`

```
    {
        ["themes/default"]: path.join(__dirname, "themes",
"default", "theme.less")
        /*,
        ["themes/complete-theme-override"]:
path.join(__dirname, "themes", "complete-theme-override",
"theme.less"),
        ["themes/partial-theme-override"]: path.join(__dirname,
"themes", "partial-theme-override", "theme.less"),
        ["themes/only-css-variables"]: path.join(__dirname,
"themes", "only-css-variables", "theme.less")
        */
    },
```

Working with Extensions

The MapStore2 plugins architecture allows building your own independent modules that will integrate seamlessly into your project.

Extensions are plugins that can be distributed as a separate package (a zip file), and be installed, activated and used at runtime. Creating an extension is similar to creating a plugin. If you are not familiar with plugins, please, read the Plugins HowTo page first.

Developing an extension

The easiest way to develop an extension is to start from the MapStoreExtension project that gives you a sandbox to create/test and build your extension.

Read the readme of the project to understand how to run, debug and build a new extension starting from the sampleExtension in the project.

Here you can find some details about the structure extension files, useful for development and debugging.

An extension example

A MapStore extension is a plugin, with some additional features.

```
import {connect} from "react-redux";

import Extension from "../components/Extension";
import Rx from "rxjs";
import { changeZoomLevel } from "../../../web/client/actions/
map";

export default {
    name: "SampleExtension",
    component: connect(state => ({
        value: state.sampleExtension &&
state.sampleExtension.value
```

```
    }), {
        onIncrease: () => {
            return {
                type: 'INCREASE_COUNTER'
            };
        }, changeZoomLevel
    })(Extension),
    reducers: {
        sampleExtension: (state = { value: 1 }, action) => {
            if (action.type === 'INCREASE_COUNTER') {
                return { value: state.value + 1 };
            }
            return state;
        }
    },
    epics: {
        logCounterValue: (action$, store) =>
action$.ofType('INCREASE_COUNTER').switchMap(() => {
            /* eslint-disable */
            console.log('CURRENT VALUE: ' +
store.getState().sampleExtension.value);
            /* eslint-enable */
            return Rx.Observable.empty();
        })
    },
    containers: {
        Toolbar: {
            name: "SampleExtension",
            position: 10,
            text: "INC",
            doNotHide: true,
            action: () => {
                return {
                    type: 'INCREASE_COUNTER'
                };
            },
            priority: 1
        }
    }
};
```

As you can see from the code, the most important difference is that you need to export the plugin descriptor **WITHOUT** invoking `createPlugin` on it (this is done in `extensions.js` in dev environment and when installed it will be done by the extensions load system). The extension definition will import or define all the needed dependencies (components, reducers, epics) as well as the plugin configuration elements (e.g. containers).

Dynamic import of extension

MapStore supports dynamic import of plugins and extensions.

Dynamically imported plugins or extensions uses lazy-loading: components, reducers and epics will be loaded once plugin or extension is in the list of plugins configured for the current page (eg. via `localConfig.json` or plugins selected to be included in a context).

> **Note**
>
> Application context could have plugins configured to be loaded optionally using the Extensions Library. Such plugins will be loaded only after being directly activated by the user in the extensions library UI.

Regardless if extension uses lazy-loading or not, its epics will be muted once extension is not rendered on the page. For more details see Epic state.

There are few changes required to make extension loaded dynamically:

1. Create `Module.jsx` file in `js/extension/plugins/` and populate it with `js/extension/plugins/Extension.jsx` content.
2. Update content of `js/extension/plugins/Extension.jsx` to be like:

```
import {toModulePlugin} from "@mapstore/utils/
ModulePluginsUtils";
import { name } from '../../../config';

export default toModulePlugin(name, () => import(/*
webpackChunkName: 'extensionName' */ './Module'));
```

3. Update `js/extensions.js` and remove `createPlugin` wrapper from `Extension` export. File content should look like:

```
import Extension from './extension/plugins/Extension';
import { name } from '../config';


export default {
    [name]: Extension
};
```

Distributing your extension as an uploadable module

The sample project allow you to create the final zip file for you.

The final zip file must have this form:

- the file named `index.js` is the main entry point, for the module.
- an `index.json` file that describes the extension, an example follows
- `assets` folder, that contains additional bundles (js, css) came out from the bundle compilation. All additional files (js chunks, css ...) must stay in this folder.
- optionally, a `translations` folder with localized message files used by the extension (in one or more languages of your choice)

```
my-extension.zip
|── index.js
├── index.json
├── assets
    ├── css
        └── 123.abcd.css
        └── ...
    └── js
        └── 456.abcd.js
        └── ...
└── translations
    └── data.en_EN.json
    └── ...
```

index.json

The `index.json file should contain all the information about the extension:

- An `id` that identifies the extension
- A `version` to show in UI. Semantic versioning is suggested.+
- `title` and `description` to display in UI, mnemonic hints for the administrator
- `plugins` the list of plugins that it adds to the application, with all the data useful for the context manager. Format of the JSON object for plugins is suggested here

```
{
    "id": "a_unique_extension_identifier",
    "version": "1.0.0",
    "title": "the title of the description",
    "description": "a description of the extension",
    "plugins": [{
         "name": "MYPlugin",
         "title":
"extensions.a_unique_extension_identifier.title",
         "description": "",
         "defaultConfig": {},
         "...": "..."
    }]
}
```

`plugins` section contains the plugins defined in the extension, and it is needed to be configured in the context-editor. See Context Editor Configuration

Installing Extensions

Extensions can be uploaded using the context creator UI of MapStore. The storage and configuration of the uploaded zip bundle is managed by a dedicated MapStore backend service, the ***Upload Service***. The Upload Service is responsible for unzipping the bundle, storing javascript and the other extension

assets in the extensions folder and updating the configuration files needed by MapStore to use the extension:

- `extensions.json` (the extensions registry)
- `pluginsConfig.json.patch` (the context creator plugins catalog patch file)

Updating Extensions

Please refer to the How to update extensions section of user guide to get more information about extensions update workflow.

Extensions and datadir

Extensions work better if you use a datadir, because when a datadir is configured, extensions are uploaded inside it, so they can ***live*** outside the application main folder (and you don't risk to overwrite them when you upgrade MapStore to a newer version).

Extensions for dependent projects

Extensions build in MapStore actually can run only in MapStore product. They can not be installed in dependent projects. If you have a custom project, and you want to add support for extensions, you will have to create your build system for extensions dedicated to your application, to build the Javascript with the correct paths. Moreover, to enable extensions to work with the datadir in a dependent project (MapStore product is already configured to use it) you need to configure (or customize) the following configuration properties in your `app.jsx`:

Externalize the extensions configuration

Change `app.jsx` to include the following statement:

```
ConfigUtils.setConfigProp("extensionsRegistry", "rest/config/
load/extensions.json");
```

Externalize the context plugins configuration

Change `app.jsx` to include the following statement:

```
ConfigUtils.setConfigProp("contextPluginsConfiguration", "rest/
config/load/pluginsConfig.json");
```

Externalize the extensions assets folder

Change `app.jsx` to include the following statement:

```
ConfigUtils.setConfigProp("extensionsFolder", "rest/config/
loadasset");
```

Assets are loaded using a different service, `/rest/config/loadasset` .

Managing drawing interactions conflict in extension

Extension could implement drawing interactions, and it's necessary to prevent a situation when multiple tools from different plugins or extensions have active drawing, otherwise it could end up in an unpredicted or buggy behavior.

There are two ways how drawing interaction can be implemented in plugin or extension:

- Using DrawSupport (e.g. Annotations plugin)
- By intercepting click on the map interactions (e.g. Measure plugin)

Making another plugins aware of your extension starts drawing

If your extension using DrawSupport - you're on the safe side. Extension will dispatch `CHANGE_DRAWING_STATUS` action. This action can be traced by another plugins or extensions, and they can control their tools accordingly.

If your extension is using `CLICK_ON_MAP` action and intercepts it perform any manipulations on click - you need to make sure that your extension also dispatch

`REGISTER_EVENT_LISTENER` action (see `Measure` plugin as an example) when your extension activates drawing.

It should also dispatch `UNREGISTER_EVENT_LISTENER` once drawing interaction stops.

Making your extension aware of another plugin drawing

There is a helper utility named `shutdownToolOnAnotherToolDrawing` . This is a wrapper for a common approach to dispatch actions that will toggle off drawing interactions of your extension whenever another plugin or extension starts drawing.

extensionEpics.js:

```
export const toggleToolOffOnDrawToolActive = (action$, store)
=> shutdownToolOnAnotherToolDrawing(action$, store,
'yourToolName');
```

with this code located in extension's epics your tool `yourToolName` will be closed whenever: - feature editor is open - another plugin or extension starts drawing.

"shutdownToolOnAnotherToolDrawing" supports passing custom callback to determine whether your tool is active (to prevent garbage action dispatching if it's already off) and custom callback to list actions to be dispatched.

Using "ResponsiveContainer" for dock panels

Starting with MapStore v2022.02.00, layout improvements have been introduced which, in addition to other changes, introduce a new sidebar menu to be used instead of the burger menu.

All extensions using `DockPanel` or `DockablePanel` components have to be updated if their dock panel is rendered on the right side of the screen, next to the new sidebar menu.

Following changes should be applied (`MapTemplates` plugin can be a reference for the changes needs to be applied): 1. Make your extension aware of the map layout changes by getting corresponding state value using following selector:

```
createSelector(
    ...
    state => mapLayoutValuesSelector(state, { height: true,
right: true }, true),
    ...
    (dockStyle) => ({
        dockStyle
    })
)
```

It will get offset from the right and the bottom that needs to be applied to the `ResponsiveContainer`

1. Replace `DockPanel`, `DockablePanel`, `ContainerDimensions` (if used) with the `ResponsiveContainer` and make sure that dock content is a child of `ResponsiveContainer`:

was:

```
return (
    <DockPanel
        open={props.active}
        position="right"
        size={props.size}
        bsStyle="primary"
        title={<Message msgId="mapTemplates.title"/>}
        style={{ height: 'calc(100% - 30px)' }}
        onClose={props.onToggleControl}>
        {!props.templatesLoaded && <div className="map-
templates-loader"><Loader size={352}/></div>}
        {props.templatesLoaded && <MapTemplatesPanel
            templates={props.templates}
            onMergeTemplate={props.onMergeTemplate}
            onReplaceTemplate={props.onReplaceTemplate}
            onToggleFavourite={props.onToggleFavourite}/>}
    </DockPanel>
)
```

become:

```
return (
    <ResponsivePanel
        containerStyle={props.dockStyle}
        style={props.dockStyle}
        containerId="map-templates-container"
        containerClassName="dock-container"
        className="map-templates-dock-panel"
        open={props.active}
        position="right"
        size={props.size}
        bsStyle="primary"
        title={<Message msgId="mapTemplates.title"/>}
        onClose={props.onToggleControl}
    >
        {!props.templatesLoaded && <div className="map-
templates-loader"><Loader size={352}/></div>}
        {props.templatesLoaded && <MapTemplatesPanel
            templates={props.templates}
            onMergeTemplate={props.onMergeTemplate}
            onReplaceTemplate={props.onReplaceTemplate}
            onToggleFavourite={props.onToggleFavourite}/>}
    </ResponsivePanel>
);
```

With the applied changes dock will be rendered properly both for layout with `BurgerMenu` and `SidebarMenu`.

Making other dock panels closed automatically when extension panel is open

All the dock panels open next to the sidebar should be mutually excluded. Active dock panel should be closed whenever another panel is open.

Array at `state.maplayout.dockPanels.right` contains list of panels that can be extended or modified by extension by dispatching `updateDockPanelsList` action.

Please note that adding dock into the list will automatically close previously active panel, so it's a good idea to dispatch the action on app initializing or when

dock panel is open. Measurement plugin can be used as a reference of implementation, see `openMeasureEpic` & `closeMeasureEpic` in `epics/measurement.js`.

Printing Module

The **printing module** of MapStore is a back-end service **not included by default in the binary package** that allows to create a printable PDF from the current map.

> **Note**
>
> The **printing module** is required by the **Print plugin** of MapStore, so if you want to have the Print plugin working in your application, you have to include also the printing module in your MapStore installation.

Including the printing module in MapStore

Because MapStore doesn't include the printing module by default, to use it you need to build from the source a `MapStore.war` that includes it or add the missing files to an existing MapStore deployed.

Building from the Source

If you want to include the printing module in your MapStore, by building the source code, you have to add the profile `printing` (profiles can be added as 2nd argument of the `build.sh` script, after the version that is the 1st. If you have more then one profile, you can add them separated by `,`):

```
./build.sh [version_identifier] printing
```

MapStore projects also allow to use the `printing` profile to include this module. So you can use the same `printing` profile to build your custom MapStore project including the printing module.

Adding to an existing MapStore

If you have an existing and deployed instance of MapStore and you want to add the printing module, you can build only the printing extension as a zip running `mvn clean install -Pprintingbundle` from the official Mapstore project. The zip bundle will created in `java/printing/target/mapstore-printing.zip` .

You can copy the content of this zip bundle into the root of mapstore application (`<app_root>` , for instance `webapps/mapstore` in Tomcat):

- files from zip directory `WEB-INF/classes` must be placed in `<app_root>/WEB-INF/classes`
- files from zip directory `WEB-INF/lib` must be placed in `<app_root>/WEB-INF/lib`

for the printing configuration files (if they are missing)

- these files must be placed in `<app_root>/printing`

Then restart your java container.

Configuring the print

This printing module includes the MapStore printing engine, that is a fork of MapPhish print (version 2), with some additional functionalities you can find in the Wiki page.

!!! note: The module was originally written for GeoServer, so on the Github wiki you can find information about downloading and installing it in GeoServer, but if you include the engine directly in MapStore you don't need any other installation.

MapStore

The MapStore print module on the front-end is implemented by the `Print` plugin inside `localConfig.json`. Make you sure to have this plugin in `plugins/(mode)` section. If so, this will will automatically check the presence of the back-end module and show the entry in the Burger Menu, if the back-end service is present.

In `localConfig.json` you will find also a `printUrl` configuration that refers to the (relative) URL where the main entry point of the application is available. (the default should enough)

Print Settings

This fork uses the same configuration files of the original library to define the various print layouts and options. This files is in the directory `resources/geoserver/print`, and they will be copied in `WEB-INF/classes` in the final `war` file.

- `config.yml` : The main file that configures the layout. More information about this configuration file in the original documentaiton
- `print_header.png` : The header, referred in `config.yml`
- `Arrow_north_CFCF.svg` the north indicator, referred in `config.yml`

Troubleshooting

I can not see the "Print" entry in the menu

Please check if:

- "Print" plugin is present in `localConfig.json --> plugins --> desktop`.
- You are using a desktop browser. The plugin is not designed for a mobile devices (tablet, smartphone...), for this reason it is not included in `plugins --> mobile`

- The service at url `http(s)://<your-domain>/<applciation-base-paath>/pdf/info.json` is responding. Example: `https://example.com/mapstore/pdf/info.json`. The URL of this `info.json` is configured (by default as relative URL) in `localConfig.json` --> `printUrl` entry.
- Looking at the JSON returned by the request above, the URLs in the entries `printURL` and `createURL` are reachable, and the domain, (the port) and the schema (http/https) of these URLs are the same of MapStore.

I have an error printing (using Reverse Proxy/HTTPS)

When you open MapStore from the browser, MapStore do a request to the main entry point (`info.json`) of the printing module. This entry point provides a set of URLs where to find all the print related services. These URLs are generated starting from the current request.

A common practice is to use a reverse proxy in front of a Java Application Server, and so MapStore (this is used also to add https, if the reverse proxy is part of a web server). If this reverse proxy is not properly configured anyway, MapStore will not be able to correctly generate the URLs of the printing services, and this may cause an error when you try to print a PDF.

To avoid this problem, you can use several solution, depending on your setup and your reverse proxy.

Setting up your proxy

Some typical solutions are:

- Using AJP instead of http (this forwards all the information by default)

```
# Example for Apache HTTP server
ProxyPass        /mapstore   ajp://localhost:8010/mapstore
ProxyPassReverse /mapstore   ajp://localhost:8010/mapstore
```

- Using rewrite engine to rewrite the requests. (apache web server)

```
```

```
# example for Apache HTTP server
RewriteEngine On
RewriteCond %{HTTPS} off
RewriteRule ^ https://%{HTTP_HOST}%{REQUEST_URI}
RedirectMatch ^/$ /mapstore/
ProxyRequests Off
ProxyPreserveHost On
ProxyVia full
```

- Using non-standard headers, `X-Forwarded-Host` and `X-Forwarded-Proto`.

```
# example for nginx
proxy_set_header X-Forwarded-Proto https;
```

Forcing PRINT_BASE_URL of printing module

If, for any reason, you can not modify the proxy configuration, MapStore printing module provides a system variable `PRINT_BASE_URL` that you can set to force the URLs returned by `info.json` to be resolved from it.

A useful trick can be to set as a relative URL (relative to MapStore) to make it work in any context (only for MapStore).

```
JAVA_OPTS= "$JAVA_OPTS -DPRINT_BASE_URL=pdf"
```

or you can use the absolute URL:

```
JAVA_OPTS= "$JAVA_OPTS -DPRINT_BASE_URL=https://example.com/
mapstore/pdf"
```

How to use a CDN

The LeafletDraw plugin and the MapStore theme are linked via rawgit.com but in production it should be used a proper CDN.
Once you have a stable version: - upload the LeafletDraw plugin and the MapStore theme on your CDN - edit the index.html file to use your published resources.

FAQ

Troubleshooting

Autowatch doesn't work on Linux.

You should need to increase `max_user_watches` variable for inotify.

```
echo fs.inotify.max_user_watches=524288 | sudo tee -a /etc/
sysctl.conf && sudo sysctl -p
```

Other References

- How to use a CDN

Code conventions

In order to preserve quality, maintainability and testability when you develop in MapStore you should follow the following rules and best practices.

TL;DR

- Access to the state using state selectors
- Prefer plugins `cfg` over `initialState` for plugins configurations
- Use `web/client/libs/ajax` in your hooks or in redux-observable for async

Access to the state using state selectors

Is **strongly recommended** to not access to the state directly inside the `mapStateToProps` function of `react-redux` . Use (or define) selectors in the `selectors` directory. This provides the following advantages:

- Selectors can be reused in epics.
- Ready to use optimization with reselect
- Simplify future refactoring
- Easy unit testing and bug identification

Wrapping all the access to the state inside well-defined selector makes easier to add functionalities and will increase code maintainability. You should always reuse existing selectors (or create new ones) to access to the application state for core application functionalities. It will help also future refactoring because any change to the state structure (from the reducer point of view) or data source (from the components point of view) requires only changes to the interested selectors.

A selector should be placed into the proper `selectors/<state-slice>.js` file with the same name of the relative reducer. When a selector retrieves data from

more than one state slices, you should place it in the selector nearest by concern. For instance `isFeatureGridOpen` should be placed into `featuregrid`

If you don't work on a core functionality, where the state is shared between many components, defining the selector directly in the plug-in is not denied.

Prefer plugin configuration over `initialState`

In order to create self contained plugins that can be reused you should prefer to configure the plugins using `cfg` . Using `initialState` should be considered deprecated. When the configuration is needed at an higher level (e.g. application state, for epics or to share this information), you should properly initialize the state of the plugin on your own triggering an action on mount/unmont. (`cfg` are passed to the plugin as react props).

Use custom axios version for async requests

Using `web/client/libs/ajax` (a customized axios with some interceptors) for AJAX request contains interceptors to support `proxyUrl` and `authenticationRules` settings specified in `localConfig.json` ,so you should prefer to use this enhanced version of axios.

Using axios + RxJS means that you will have to wrap axios calls in something like:

```
Rx.Observable.defer( () => axios.post(...)).map...
```

Use defer to allow the usage of RxJS retry. We still not support real AJAX cancellation at all, but we would like to provide some utility function/operator to bind `axios` cancellation functionalities into the RxJS flow in the future.

Documentation guidelines

Each new feature/tool in MapStore should be documented in the User Guide in order to explain the involved functionalities and illustrate how it works.

All new front-end technologies, development procedures, best practices and guidelines on the involved components in MapStore should be properly documented too: the Developers Guide must be kept up-to-date for this.

The Developer and User guide documentation are built on the Read the Docs hosting platform. The MapStore's documentation files are available in the docs/ section of this repository; Mkdocs is used in MapStore as documentation generator, you can look at the available online documentation for more information on how to use it (MapStore uses his own customized MkDocs Material theme for both User and Developer documentations).

Building documentation

The documentation is built on RTD (Read the Docs) documentation hosting platform.

But in order to build it locally, there are certain steps that needs to be followed

1. **Python installation**
2. Install **Python 3**
3. **Pip** is automatically installed when python is downloaded from python.org, if not, follow this instruction to install pip
4. **Libraries installation**

   Install **all** the libraries/plugins in `docs/requirements.txt` using ***pip*** while matching the exact version present

   Example

   For `mkdocs-material==3.2.0`, the installation using pip is as follows `pip install mkdocs-material==3.2.0`

5. Build the docs using the command `mkdocs build`. This will build the documentation and puts the built files into `site` folder and the pdf generated into `site\pdf\mapstore_documentation.pdf`
6. Alternatively, the command `mkdocs serve` starts the built-in dev-server, of MkDocs, that lets us preview the documentation as we work on it
7. The documentation can be launched using `index.html` in `site` folder

> **Note**
>
> When creating a link to internal document (.md) files, make sure to use full link instead of a relative path to the file. As using relative path will not work in exported PDF document. ***Example***: Instead of creating a link `[FAQ]('../dev-faq/')`, use `[FAQ]('../dev-faq/#faq')` or `[FAQ]('../dev-faq.md#faq')` or `[FAQ]('dev-faq.md#faq')`

Migration Guidelines

General update checklist

- updating an existing installation
- updating a MapStore project created for a previous version

To update an existing installation you usually have to do nothing except eventually to execute queries on your database to update the changes to the database schema.

In case of a project it becomes a little more complicated, depending on the customization.

This is a list of things to check if you want to update from a previous version valid for every version.

- Take a list to migration notes below for your version
- Take a look to the release notes
- update your `package.json` to latest libs versions
- take a look at your custom files to see if there are some changes (e.g. `localConfig.js`, `proxy.properties`)
- Some changes that may need to be ported could be present also in `pom.xml` files and in `configs` directory.
- check for changes also in `web/src/main/webapp/WEB-INF/web.xml`.
- Optionally check also accessory files like `.eslinrc`, if you want to keep aligned with lint standards.
- Follow the instructions below, in order, from your version to the one you want to update to.

Migration from 2022.01.02 to 2022.02.00

HTML pages optimization

We removed script and css link to leaflet CDN in favor of a dynamic import of the libraries in the main bundle, now leaflet is only loaded when the library is selected as map type of the viewer. You can update the project HTML files by removing these tags:

```
- <link rel="stylesheet" href="https://cdnjs.cloudflare.com/
ajax/libs/leaflet/1.3.1/leaflet.css" />
- <link rel="stylesheet" href="https://cdnjs.cloudflare.com/
ajax/libs/leaflet.draw/1.0.2/leaflet.draw.css" />
```

```
- <script src="https://cdnjs.cloudflare.com/ajax/libs/leaflet/
1.3.1/leaflet.js"></script>
- <script src="https://cdnjs.cloudflare.com/ajax/libs/
leaflet.draw/1.0.2/leaflet.draw.js"></script>
```

We also made asynchronous the script to detect valid browser. This should slightly improve the initial requests time. You can updated the script in your project as following:

```
<script async type="text/javascript" src="https://unpkg.com/
bowser@2.7.0/es5.js" onload="checkBrowser()"></script>
<script type="text/javascript">
    function checkBrowser() {
        var browserInfo =
bowser.getParser(window.navigator.userAgent);
        var isValidBrowser = browserInfo.satisfies({
            "edge": ">1",
            "chrome": ">1",
            "safari": ">1",
            "firefox": ">1"
        });
        if (!isValidBrowser) {
            window.location.href = "unsupportedBrowser.html"
```

```
            document.querySelector("container").style.display =
"none";
        }
    }
</script>
```

Update plugins.js to make upstream plugins use dynamic import

We've updated `plugins.js` in MapStore to make most of the plugins use dynamic import. `plugins.js` of your project have to be updated separately.

Please use `web\client\product\plugins.js` file as a reference listing plugins whose definition can be changed to support dynamic import.

To use dynamic import for plugin, please update its definition to look like:

```
{
    ...
    AnnotationsPlugin: toModulePlugin('Annotations', () =>
import(/* webpackChunkName: 'plugins/annotations' */ '../
plugins/Annotations')),
    ...
}
```

See Dynamic import of extension to have more details about transforming extensions to use dynamic import.

Version plugin has been removed

We no longer maintain the Version plugin since we have moved its content inside the About plugin (see here for more details)

We suggest you to clean up your project as well:

- remove Version entry it from a local list of plugins.js
- remove Version entries it from a localConfig
- add About entry into other pages of mapstore plugins array, suggest list is:
- dashboard

- geostory
- mobile
- remove Define plugins in webpack-config.js or prod.webpack-config.js, since we have moved these definition to a more general *build/buildConfig.js* file
- check that in your package.json you have this extends rule

```
"eslintConfig": {
    "extends": [
      "@mapstore/eslint-config-mapstore"
    ],
    ...
```

- edit the version of the *@mapstore/eslint-config-mapstore* to **1.0.5** in your package.json so that the new globals config will be inherited

> **Note**
>
> this may fail on gha workflows, in that case we suggest to edit directly your package.json with globals taken from mapstore framework

Support for OpenID

MapStore introduced support for OpenID for google and keycloak. In order to have this functionalities and to be aligned with the latest version of MapStore you have to update the following files in your projects:

- `geostore-spring-security.xml` (your custom spring security context) have to be updated adding the beans and the `security:custom-filter` entry in the `<security:http>` entry, as here below:

```
        <security:csrf disabled="true"/>
        <security:custom-filter
ref="authenticationTokenProcessingFilter"
before="FORM_LOGIN_FILTER"/>
        <security:custom-filter
ref="sessionTokenProcessingFilter" after="FORM_LOGIN_FILTER"/>
+        <security:custom-filter ref="keycloakFilter"
```

```
before="BASIC_AUTH_FILTER"/>
+         <security:custom-filter ref="googleOpenIdFilter"
after="BASIC_AUTH_FILTER"/>
        <security:anonymous />
    </security:http>

    <security:authentication-manager>
        <security:authentication-provider
ref='geoStoreUserServiceAuthenticationProvider' />
    </security:authentication-manager>
+
+
+    <bean id="preauthenticatedAuthenticationProvider"
class="it.geosolutions.geostore.services.rest.security.PreAuthenti
+    </bean>
+
+    <!-- OAuth2 beans -->
+    <context:annotation-config/>
+
+    <bean id="googleSecurityConfiguration"
class="it.geosolutions.geostore.services.rest.security.oauth2.goog
>
+
+    <!-- Keycloak -->
+
+   <bean id="keycloakConfig"
class="it.geosolutions.geostore.services.rest.security.keycloak.Ke
>
+
+    <!-- END OAuth2 beans-->
+
+    <!--  security integration inclusions  -->
+    <import resource="classpath*:security-integration-$
{security.integration:default}.xml"/>
```

- `web.xml` : add the following content to the file:

```
@@ -34,6 +34,17 @@
        <listener-
class>org.springframework.web.context.ContextLoaderListener</
listener-class>
    </listener>

+    <!-- Allow to use RequestContextHolder -->
```

```
+    <filter>
+        <filter-name>springRequestContextFilter</filter-name>
+        <filter-
class>org.springframework.web.filter.RequestContextFilter</
filter-class>
+    </filter>
+    <filter-mapping>
+        <filter-name>springRequestContextFilter</filter-name>
+        <url-pattern>/*</url-pattern>
+    </filter-mapping>
+
+
    <!-- Spring Security Servlet -->
    <filter>
```

- `applicationContext.xml` for consistency, we added `mapstore-ovr.properties` files to be searched in class-path and in the data-dir, as for the other properties files:

```
@@ -49,6 +49,7 @@
         <property name="order" value="10"/>
         <property name="locations">
             <list>
+                 <value>classpath:mapstore-ovr.properties</
value>
                 <value>file:${datadir.location:}/geostore-
datasource-ovr.properties</value>
                 <value>file:${datadir.location:}/mapstore-
ovr.properties</value>
             </list>
```

Upgrading the printing engine

The mapfish-print based printing engine has been upgraded to align to the latest official 2.1.5 in term of functionalities.

An update to the MapStore printing engine context file (`applicationContext-print.xml`) is needed for all projects built with the printing profile enabled. The following sections should be added to the file:

```
```

```
+<bean id="threadResources"
class="org.mapfish.print.ThreadResources">
+    <property name="connectionTimeout" value="30000"/>
+    <property name="socketTimeout" value="30000" />
+    <property name="globalParallelFetches" value="200"/>
+    <property name="perHostParallelFetches" value="30" />
+</bean>



+
+<bean id="metricRegistry"
class="com.codahale.metrics.MetricRegistry" lazy-init="false"/>
+<bean id="healthCheckRegistry"
class="com.codahale.metrics.health.HealthCheckRegistry" lazy-
init="false"/>
+<bean id="loggingMetricsConfigurator"
class="org.mapfish.print.metrics.LoggingMetricsConfigurator"
lazy-init="false"/>
+<bean id="jvmMetricsConfigurator"
class="org.mapfish.print.metrics.JvmMetricsConfigurator" lazy-
init="false"/>
+<bean id="jmlMetricsReporter"
class="org.mapfish.print.metrics.JmxMetricsReporter" lazy-
init="false"/>
```

Also, remember to update your project pom.xml with the updated dependency:

- locate the print-lib dependency in the pom.xml file
- replace the dependency with the following snippet

```
<dependency>
    <groupId>org.mapfish.print</groupId>
    <artifactId>print-lib</artifactId>
    <version>geosolutions-2.1.0</version>
    <exclusions>
        <exclusion>
            <groupId>commons-codec</groupId>
            <artifactId>commons-codec</artifactId>
        </exclusion>
        <exclusion>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-annotations</artifactId>
```

```
        </exclusion>
        <exclusion>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-core</artifactId>
        </exclusion>
        <exclusion>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Finally, to enable printing in different formats than PDF, you should add the following to your `config.yml` file (at the top level):

```
formats:
  - '*'
```

Replacing BurgerMenu with SidebarMenu

There were several changes applied to the application layout, one of them is the Sidebar Menu that comes to replace Burger menu on map viewer and in contexts. Following actions need to be applied to make a switch:

- Update localConfig.json and add "SidebarMenu" entry to the "desktop" section:

```
{
    "desktop": [
        ...
        "SidebarMenu",
        ...
```

```
    ]
}
```

- Remove "BurgerMenu" entry from "desktop" section.

Using Sidebar Menu in new contexts

Contents of your `pluginsConfig.json` need to be reviewed to allow usage of new "SidebarMenu" in new contexts. Existing contexts need to be updated separately, please refer to the next chapter for instructions.

- Find "BurgerMenu" plugin configuration in `pluginsConfig.json` and remove `"hidden": true` line from it:

```
    {
    "name": "BurgerMenu",
    "glyph": "menu-hamburger",
    "title": "plugins.BurgerMenu.title",
    "description": "plugins.BurgerMenu.description",
    "dependencies": [
        "OmniBar"
    ]
}
```

- Add `SidebarMenu` entry to the "plugins" array:

```
{
    "plugins": [
        ...
        {
            "name": "SidebarMenu",
            "hidden": true
        }
        ...
    ]
}
```

- Go through all plugins definitions and replace `BurgerMenu` dependency with `SidebarMenu`, e.g.:

```
    {
      "name": "MapExport",
      "glyph": "download",
      "title": "plugins.MapExport.title",
      "description": "plugins.MapExport.description",
      "dependencies": [
        "SidebarMenu"
      ]
    }
```

- Also the `StreetView` plugin needs to depend from `SidebarMenu`.

```
{
      "name": "StreetView",
      "glyph": "road",
      "title": "plugins.StreetView.title",
      "description": "plugins.StreetView.description",
      "dependencies": [
        "SidebarMenu"
      ]
}
```

Updating existing contexts to use Sidebar Menu

Contexts created in previous versions of MapStore will maintain old Burger Menu. There are two options allowing to replace it with the new Sidebar Menu:

- Using manual update.
- Using SQL query to update all contexts at once.

Before going with one of the approaches, please make sure that changes to `pluginsConfig.json` from previous chapter are applied.

To update context manually:

1. Go to the context manager (#/context-manager) and edit context you want to update.
2. Move to the step 3: Configure Plugins.
3. Find "Burger Menu" on the right side (enabled plugins) and move it to the left column.

4. Save context

Note: "Burger Menu" has higher priority over the "Sidebar Menu", so it will always be used if it's added to the list of enabled plugins of the context.

To update all contexts at once:

This is a sample SQL query that can be executed against the MapStore DB to replace the Burger Menu with the new Sidebar for existing application contexts previously created:

```
UPDATE geostore.gs_stored_data SET stored_data =
regexp_replace(gs_stored_data.stored_data,'{"name":"BurgerMenu"},'
FROM geostore.gs_resource
WHERE gs_stored_data.resource_id = gs_resource.id AND
        gs_resource.category_id = (SELECT id FROM
geostore.gs_category WHERE name = 'CONTEXT') AND
        gs_stored_data.stored_data ~
'.*{"name":"BurgerMenu"},.*';
```

Note: Schema name could vary depending on your installation configuration.

Updating extensions

Please refer to the extensions documentation to know how to update your extensions.

Using `terrain` layer type to define 3D map elevation profile

A new `terrain` layer type has been created in order to provide more options and versatility when defining an elevation profile for the 3D map terrain. This `terrain` layer will substitute the former `wms` layer (with `useForElevation` attribute) used to define the elevation profile.

> **Note**
>
> The `wms` layer (with `useForElevation` attribute) configuration is still needed to show the elevation data inside the MousePosition plugin and it will display the terrain at the same time. The `terrain` layer type allows a more versatile way of handling elevation but it will work only as terrain visualization in the 3D map viewer.

The `additionalLayers` object on the `localConfig.json` file should adhere now to the terrain layer configuration. Serve the following code as an example:

```
{
    "name": "Map",
    "cfg": {
        "additionalLayers": [{
            "type": "terrain",
            "provider": "wms",
            "url": "https://host-sample/geoserver/wms",
            "name": "workspace:layername",  // name of the
geoserver resource
            "littleendian": false,
            "visibility": true
        }]
    }
}
```

> **Note**
>
> When using `terrain` layer with `wms` provider, the format option in layer configuration is not needed anymore as Mapstore supports only `image/bil` format and is used by default

Migration from 2022.01.00 to 2022.01.01

MailingLists plugin has been removed

`MailingLists` plugin has ben removed from the core of MapStore. This means you can remove it from your `localConfig.json` (if present, it will be anyway ignored by the plugin system).

Migration from 2021.02.02 to 2022.01.00

This release includes several libraries upgrade on the backend side, in particular the following have been migrated to the latest available versions:

Library	Old	New
Spring	3.0.5	5.3.9
Spring-security	3.0.5	5.3.10
CXF	2.3.2	3.4.4
Hibernate	3.3.2	5.5.0
JPA	1.0	2.1
hibernate-generic-dao	0.5.1	1.3.0-SNAPSHOT
h2	1.3.168	1.3.175
javax-servlet-api	2.5	3.1.0

This requires also the **upgrade of Tomcat to at least version 8.5.x**.

Updating projects configuration

Projects need the following to update to this MapStore release:

- update dependencies (in `web/pom.xml`) copying those in `MapStore2/java/web/pom.xml`, in particular (where present):

Dependency	Version	Notes
mapstore-services	1.3.0	Replaces mapstore-backend
geostore-webapp	1.8.0	

- update packagingExcludes in `web/pom.xml` to this list:

```
WEB-INF/lib/commons-codec-1.2.jar,
WEB-INF/lib/commons-io-1.1.jar,
WEB-INF/lib/commons-logging-1.0.4.jar,
WEB-INF/lib/commons-pool-1.3.jar,
WEB-INF/lib/slf4j-api-1.5*.jar,
WEB-INF/lib/slf4j-log4j12-1.5*.jar,
WEB-INF/lib/spring-tx-5.2.15*.jar
```

- upgrade **Tomcat to 8.5** or greater
- update your `geostore-spring-security.xml` file to add the following setting, needed to disable CSRF validation, that MapStore services do not implement yet:

```
<security:http ... >
    ...
    <security:csrf disabled="true"/>
    ...
</security:http>
```

- remove the spring log4j listener from `web.xml`

```
 <!-- spring context loader
    <listener>
        <listener-
class>org.springframework.web.util.Log4jConfigListener</
```

```
listener-class>
    </listener>-->
```

- If one of the libraries updated is used in your project, you should align the version with the newer one to avoid jar duplications
- Some old project may define versions of spring and/or jackson in maven properties. You can remove these definition and the dependency from main `pom.xml` since they should be inherited from spring. In particular you may need to remove these properties :

```
-         <jackson.version>1.9.10</jackson.version>
-         <jackson.databind-version>2.2.3</jackson.databind-
version>
-         <jackson.annotations-version>2.5.3</
jackson.annotations-version>
```

Upgrading CesiumJS

CesiumJS has been upgraded to version 1.90 (from 1.17) and included directly in the mapstore bundle as async import.

Downstream project should update following configurations:

- remove all executions related to the cesium library from the pom.xml

```
    <id>html, configuration files and images</id>
    <phase>process-classes</phase>
    <goals>
        <goal>copy-resources</goal>
    </goals>
    <configuration>
        <outputDirectory>${basedir}/target/mapstore</
outputDirectory>
        <encoding>UTF-8</encoding>
        <resources>
            <resource>
                <directory>${basedir}/../web/client</directory>
                <includes>
                    <include>**/*.html</include>
```

```
                    <include>**/*.json</include>
                    <include>**/img/*</include>
                    <include>product/assets/symbols/*</include>
                    <include>**/*.less</include>
                </includes>
                <excludes>
                    <exclude>node_modules/*</exclude>
                    <exclude>node_modules/**/*</exclude>
-                    <exclude>**/libs/Cesium/**/*</exclude>
                    <exclude>**/test-resources/*</exclude>
                </excludes>
            </resource>
        </resources>
    </configuration>

-<execution>
-    <id>CesiumJS-navigation</id>
-    <phase>process-classes</phase>
-    <goals>
-        <goal>copy-resources</goal>
-    </goals>
-    <configuration>
-        <outputDirectory>${basedir}/target/mapstore/libs/
cesium-navigation</outputDirectory>
-        <encoding>UTF-8</encoding>
-        <resources>
-            <resource>
-                <directory>${basedir}/../web/client/libs/
cesium-navigation</directory>
-            </resource>
-        </resources>
-    </configuration>
-</execution>
```

- remove all the external script and css related to cesium and cesium-navigation now included as packages

```
-<script src="https://cesium.com/downloads/cesiumjs/releases/
1.42/Build/Cesium/Cesium.js"></script>
-<link rel="stylesheet" href="https://cesium.com/downloads/
cesiumjs/releases/1.42/Build/Cesium/Widgets/widgets.css" />
-<script src="libs/cesium-navigation/cesium-navigation.js"></
script>
```

```
-<link rel="stylesheet" href="libs/cesium-navigation/cesium-
navigation.css" />
```

- This step is needed only for custom project with a specific `publicPath` different from the default one. In this case you may need to specify what folder deliver the cesium build (by default `dist/cesium`). To do that, you can add the `cesiumBaseUrl` parameter in the webpack dev and prod configs to the correct location of the cesium static assets, widgets and workers folder.

Migration from 2021.02.01 to 2021.02.02

Style parsers dynamic import

The style parser libraries introduced a dynamic import to reduce the initial bundle size. This change reflects to the `getStyleParser` function provided by the VectorStyleUtils module. If a downstream project of MapStore is using `getStyleParser` it should update it to this new version:

```
- // old use of parser
- const parser = getStyleParser('sld');

+ // new use of parser
+ getStyleParser('sld')
+     .then((parser) => {
+         // use parser
+     });
```

Migration from 2021.02.00 to 2021.02.01

This update contains a fix for a minor vulnerability found in `log4j` library. For this reason you may need to update the dependencies of your project

> **Note**
>
> This vulnerability **is not** CVE-2021-44228 but only a couple of smaller ones, that involve `Log4J` (CVE-2021-44228 is for `Log4J2`). Anyway MapStore is not prone to these vulnerabilities with the default configuration. For more information, see the dedicated blog post

here the instructions:

Align `pom.xml` files

Here the changes in `pom.xml` and `web/pom.xml` to update:

- Change `mapstore-backend` into `mapstore-services` and set the version to `1.2.2`

```
    <dependency>
    <groupId>it.geosolutions.mapstore</groupId>
-       <artifactId>mapstore-backend</artifactId>
-       <version>1.2.1</version>
+       <artifactId>mapstore-services</artifactId>
+       <version>1.2.2</version>
    </dependency>
```

- Set `geostore-webapp` version to `1.7.1`

```
    <dependency>
    <groupId>it.geosolutions.geostore</groupId>
    <artifactId>geostore-webapp</artifactId>
-       <version>1.7.0</version>
+       <version>1.7.1</version>
    <type>war</type>
    <scope>runtime</scope>
    </dependency>
```

- Set `http_proxy` version to `1.1.1` (should already be there)

```
    <dependency>
    <!-- ... -->
```

```
    <groupId>proxy</groupId>
    <artifactId>http_proxy</artifactId>
-       <version>1.1.0</version>
+       <version>1.1.1</version>
    <type>war</type>
    <scope>runtime</scope>
    </dependency>
```

- Set `print-lib` version `geosolutions-2.0` to version `geosolutions-2.0.1`

```
    <dependency>
        <groupId>org.mapfish.print</groupId>
        <artifactId>print-lib</artifactId>
-         <version>geosolutions-2.0</version>
+         <version>geosolutions-2.0.1</version>
    </dependency>
```

Migration from 2021.01.04 to 2021.02.00

Theme updates and CSS variables

The theme of MapStore has been updated to support CSS variables for some aspects of the style, in particular colors and font families. The `web/client/themes/default/variables.less` file contains all the available variables described under the `@ms-theme-vars` ruleset. It is suggested to :

- update the lesscss variables in the projects because the variables starting with `@ms2-` will be deprecated soon:

`@ms2-color-text` -> `@ms-main-color` `@ms2-color-background` -> `@ms-main-bg` `@ms2-color-shade-lighter` -> `@ms-main-border-color`

`@ms2-color-code` -> `@ms-code-color`

`@ms2-color-text-placeholder` -> `@ms-placeholder-color`

`@ms2-color-disabled` -> `@ms-disabled-bg` `@ms2-color-text-disabled` -> `@ms-disabled-color`

`@ms2-color-text-primary` -> `@ms-primary-contrast`

`@ms2-color-primary` -> `@ms-primary` `@ms2-color-info` -> `@ms-info` `@ms2-color-success` -> `@ms-success` `@ms2-color-warning` -> `@ms-warning` `@ms2-color-danger` -> `@ms-danger`

- The font family has been update to `Noto Sans` so all the html need to be updated removing the previous font link with:

```
<link rel="preconnect" href="https://fonts.gstatic.com">
<link href="https://fonts.googleapis.com/css2?
family=Noto+Sans&display=swap" rel="stylesheet">
```

- if you are importing `react-select` or `react-widgets` inline css/less in your own project, you have to remove the import. Now the stile of these libraries is managed at project level

Project system

During this release MapStore we started an rewrite of the project system, organized in different phases.

The first phase of this migration has been identified by this pull request. In this phase we are supporting the backward compatibility as much as possible, introducing the new project system in parallel with the new one (experimental). In the future the current script will be deprecated in favor of the new one.

Here below the breaking changes introduced in this release to support this new system:

This section will tell you how to migrate to support the following changes:

- Minor changes to `prod-webpack.config.js`
- Move front-end configuration files in `configs` folder
- Back-end has been reorganized

Minor changes to `prod-webpack.config.js`

Minor changes to `prod-webpack.config.js`:

```
diff --git a/project/standard/templates/prod-webpack.config.js
b/project/standard/templates/prod-webpack.config.js
index 175bf3398..6d97e2c0f 100644
--- a/project/standard/templates/prod-webpack.config.js
+++ b/project/standard/templates/prod-webpack.config.js
@@ -2,8 +2,8 @@ const path = require("path");

 const themeEntries = require('./MapStore2/build/
themes.js').themeEntries;
 const extractThemesPlugin = require('./MapStore2/build/
themes.js').extractThemesPlugin;
-const ModuleFederationPlugin = require('./MapStore2/build/
moduleFederation').plugin;
 const HtmlWebpackPlugin = require('html-webpack-plugin');
+const ModuleFederationPlugin = require('./MapStore2/build/
moduleFederation').plugin;

 const paths = {
     base: __dirname,
@@ -24,17 +24,19 @@ module.exports = require('./MapStore2/build/
buildConfig')(
     paths,
     [extractThemesPlugin, ModuleFederationPlugin],
     true,
-    "dist/",
+    undefined,
     '.__PROJECTNAME__',
     [
         new HtmlWebpackPlugin({
             template: path.join(__dirname,
'indexTemplate.html'),
+            publicPath: 'dist/',
             chunks: ['__PROJECTNAME__'],
             inject: "body",
             hash: true
         }),
         new HtmlWebpackPlugin({
             template: path.join(__dirname,
'embeddedTemplate.html'),
+            publicPath: 'dist/',
             chunks: ['__PROJECTNAME__-embedded'],
             inject: "body",
             hash: true,
```

```
@@ -42,13 +44,15 @@ module.exports = require('./MapStore2/build/
buildConfig')(
         }),
         new HtmlWebpackPlugin({
             template: path.join(__dirname, 'apiTemplate.html'),
+            publicPath: 'dist/',
             chunks: ['__PROJECTNAME__-api'],
-            inject: 'head',
+            inject: 'body',
             hash: true,
             filename: 'api.html'
         }),
         new HtmlWebpackPlugin({
             template: path.join(__dirname, 'geostory-embedded-
template.html'),
+            publicPath: 'dist/',
             chunks: ['geostory-embedded'],
             inject: "body",
             hash: true,
@@ -56,6 +60,7 @@ module.exports = require('./MapStore2/build/
buildConfig')(
         }),
         new HtmlWebpackPlugin({
             template: path.join(__dirname, 'dashboard-embedded-
template.html'),
+            publicPath: 'dist/',
             chunks: ['dashboard-embedded'],
             inject: 'body',
             hash: true,
@@ -63,6 +68,7 @@ module.exports = require('./MapStore2/build/
buildConfig')(
         })
     ],
     {
+        "@mapstore/patcher": path.resolve(__dirname,
"node_modules", "@mapstore", "patcher"),
         "@mapstore": path.resolve(__dirname, "MapStore2",
"web", "client"),
         "@js": path.resolve(__dirname, "js")
     }
```

Move front-end configuration files in `configs` folder

We suggest you to move them as well from root to configs folder, and align your `app.jsx` configuration with the new standard (if you changed the location of configs). This will allow to use the data dir in an easy way. So:

- Move the following files in `configs` directory:
- `localConfig.json`
- `new.json`
- `pluginsConfig.json`
- `config.json`
- `simple.json`
- If changed something in `app.jsx` about configuration, align to get the files moved in config.
- To allow MapStore to copy the correct file in the final war, you have to change `web/pom.xml` execution `copy-resources` for id `config files` this way (this only if you didn't customized `localConfig.json`):

```
        <goal>copy-resources</goal>
                </goals>
                    <goal>copy-resources</goal>
                </goals>
                    <configuration>
-                       <outputDirectory>${basedir}/target/
__PROJECTNAME__/MapStore2/web/client</outputDirectory>
+                       <outputDirectory>${basedir}/target/
__PROJECTNAME__/MapStore2/web/client/configs</outputDirectory>
                        <encoding>UTF-8</encoding>
                        <resources>
                            <resource>
-                               <directory>${basedir}/../
MapStore2/web/client</directory>
+                               <directory>${basedir}/../
MapStore2/web/client/configs</directory>
                                <includes>
                                    <include>localConfig.json</
include>
                                </includes>
```

Back-end has been reorganized

In particular:

- all the java code has been moved from `web/src/` to the `java/` and `product/` directories (and `release`, already existing).
- `mapstore-backend` has been renamed into `mapstore-services`.
- Some servlets have been added in order to provide native support to data dir and make it work with the new `configs` directory.

So you will have to:

- Align the `pom.xml` to the latest versions of the libs
- Edit the `web.xml` and change the `*-servlet.xml` files to expose the new services

> **Note**
>
> Future evolution of the project will avoid you to keep your own copies of the pom files as much as possible, reducing the boilerplate and making migration a lot easier. For this reasons these migration guidelines will change soon.

Here below the details of the changes.

ALIGN `POM.XML` FILES TO LATEST VERSIONS OF THE LIBS

Here the changes in `pom.xml` and `web/pom.xml to update:

- Change `mapstore-backend` into `mapstore-services` and set the version to `1.2.1`

```
    <dependency>
    <groupId>it.geosolutions.mapstore</groupId>
-       <artifactId>mapstore-backend</artifactId>
-       <version>1.1.2</version>
+       <artifactId>mapstore-services</artifactId>
```

```
+       <version>1.2.1</version>
    </dependency>
```

- Set `geostore-webapp` version to `1.7.0`

```
    <dependency>
    <groupId>it.geosolutions.geostore</groupId>
    <artifactId>geostore-webapp</artifactId>
-       <version>1.6.0</version>
+       <version>1.7.0</version>
    <type>war</type>
    <scope>runtime</scope>
    </dependency>
```

- Set `http_proxy` version to `1.1.0` (should already be there)

```
    <dependency>
    <!-- ... -->
    <groupId>proxy</groupId>
    <artifactId>http_proxy</artifactId>
-       <version>1.1.0</version>
+       <version>1.1-SNAPSHOT</version>
    <type>war</type>
    <scope>runtime</scope>
    </dependency>
```

EDIT THE `WEB.XML` AND CHANGE THE `*-SERVLET.XML` FILES TO EXPOSE THE NEW SERVICES

- Copy from mapstore to folder `web/src/main/webapp/WEB-INF/` the files:
- `configs-servlet.xml`
- `extensions-servlet.xml`
- `loadAssets-servlet.xml`
- Remove the old `dispatcher-servlet.xml` (it has been replaced by `loadAssets-servlet.xml` for backward compatibility)
- Align `web/src/main/webapp/WEB-INF/web.xml` with the new servlets as changes below (remove `dispatcher` entry in favour of the following).

```
```

```
@@ -1,6 +1,6 @@
 <?xml version="1.0" encoding="UTF-8"?>
 <web-app id="WebApp_ID" version="2.4"
-    xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
+    xmlns:javaee="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://
java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

     <!-- pick up all spring application contexts -->
@@ -19,13 +19,16 @@

     <context-param>
       <param-name>proxyPropPath</param-name>
-      <param-value>/proxy.properties</param-value>
+      <param-value>/proxy.properties,${datadir.location}/
proxy.properties</param-value>
     </context-param>

-    <!-- spring context loader -->
-    <listener>
+    <!-- <context-param> <param-name>log4jConfigLocation</
param-name> <param-value>file:${config.dir}/log4j.xml</param-
value>
+        </context-param> -->
+
+    <!-- spring context loader -->
+    <listener>
         <listener-
class>org.springframework.web.util.Log4jConfigListener</
listener-class>
-    </listener>
+    </listener>

     <!--
       - Loads the root application context of this web app at
startup.
@@ -33,8 +36,8 @@
       -
WebApplicationContextUtils.getWebApplicationContext(servletContext
     -->
     <listener>
-        <listener-
class>org.springframework.web.context.ContextLoaderListener</
listener-class>
-    </listener>
+        <listener-
```

```
class>org.springframework.web.context.ContextLoaderListener</
listener-class>
+    </listener>

     <!-- Spring Security Servlet -->
     <filter>
@@ -46,7 +49,7 @@
         <url-pattern>/rest/*</url-pattern>
     </filter-mapping>

-  <!-- GZip compression -->
+    <!-- GZip compression -->
     <filter>
         <filter-name>CompressionFilter</filter-name>
         <filter-
class>net.sf.ehcache.constructs.web.filter.GzipFilter</filter-
class>
@@ -65,17 +68,38 @@
     </filter-mapping>

     <!--  Backend Spring MVC controllers -->
+    <!--  Backward compatibility -->
     <servlet>
-        <servlet-name>dispatcher</servlet-name>
+        <servlet-name>loadAssets</servlet-name>
         <servlet-
class>org.springframework.web.servlet.DispatcherServlet</
servlet-class>
         <load-on-startup>1</load-on-startup>
     </servlet>
     <servlet-mapping>
-        <servlet-name>dispatcher</servlet-name>
+        <servlet-name>loadAssets</servlet-name>
         <url-pattern>/rest/config/*</url-pattern>
     </servlet-mapping>
+    <!--  Configs -->
+    <servlet>
+        <servlet-name>configs</servlet-name>
+        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</
servlet-class>
+        <load-on-startup>2</load-on-startup>
+    </servlet>
+    <servlet-mapping>
+        <servlet-name>configs</servlet-name>
+        <url-pattern>/configs/*</url-pattern>
+    </servlet-mapping>
+    <!-- Extensions -->
+    <servlet>
```

```
+        <servlet-name>extensions</servlet-name>
+        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</
servlet-class>
+        <load-on-startup>3</load-on-startup>
+    </servlet>
+    <servlet-mapping>
+        <servlet-name>extensions</servlet-name>
+        <url-pattern>/extensions/*</url-pattern>
+    </servlet-mapping>

-    <!-- CXF Servlet -->
+    <!-- CXF Servlet -->
     <servlet>
         <servlet-name>CXFServlet</servlet-name>
         <servlet-
class>org.apache.cxf.transport.servlet.CXFServlet</servlet-
class>
@@ -97,7 +121,7 @@
       <url-pattern>/proxy/*</url-pattern>
     </servlet-mapping>

-   <!-- Printing Servlet -->
+    <!-- Printing Servlet -->
     <servlet>
        <servlet-name>mapfish.print</servlet-name>
        <servlet-
class>org.mapfish.print.servlet.MapPrinterServlet</servlet-
class>
```

Data directory has been reorganized and is now available also for product

The new organization of the data directory is:

- `configs` will contain all json files (`localConfig.json` , `new.json` , `pluginsConfig.json` , ...) and all the `.patch` files applied to them.
- `extensions` folder contains all the data for the extensions, including `extensions.json`
- The root contains the properties files to configure database, proxy and other configs

To organize your old data directory accordingly to the new specification.

- Move all `.json` and `.json.patch` files in `configs` folder (except `extensions.json`)
- Move the directory `dist/extensions` to simply `extensions`.
- The file `extensions.json` have to be moved in `extensions/extensions.json`.
- Edit the file `extensions/extensions.json` changing the paths from `dist/extensions/<Plugin-Name>/...` to `<Plugin-Name>/...`

You can set it up by configuring `datadir.location` java system property.
Changes to paths or configuration files are not required anymore.

Configurations

- Embedded now uses popup as default. Align localConfig.json `plugins --> embedded --> Identify` with the latest one:

```
{
    "name": "Identify",
    "cfg": {
        "showInMapPopup":true,
        "viewerOptions": {
            "container": "{context.ReactSwipe}"
        }
    }
}
```

Migration from 2021.01.01 to 2021.01.03

Generally speaking this is not properly a breaking change, but more a fix to apply to your installations. Certificate for 'cesiumjs.org' has expired at 2021.05.05, so to continue using CesiumJS with MapStore you will have to replace all the URLs like `https://cesiumjs.org/releases/1.17` in `https://cesium.com/downloads/cesiumjs/releases/1.17`. This is the main fix of this minor release. See this pull request on GitHub as a sample to apply these changes to your project.

Migration from 2021.01.00 to 2021.01.01

Update embedded entry to load the correct configuration

Existing MapStore project could have an issue with the loading of map embedded page due to the impossibility to change some configuration such as localConfig.json or translations path in the javascript entry. This issue can be solved following these steps: 1 - add a custom entry named `embedded.jsx` in the `js/` directory of the project with the content:

```
import {
    setConfigProp,
    setLocalConfigurationFile
} from '@mapstore/utils/ConfigUtils';

// Add custom (overriding) translations
// example for additional translations in the project folder
// setConfigProp('translationsPath', ['./MapStore2/web/client/
translations', './translations']);
setConfigProp('translationsPath', './MapStore2/web/client/
translations');
// __PROJECTNAME__ is the name of the project used in the
creation process
setConfigProp('themePrefix', '__PROJECTNAME__');

// Use a custom plugins configuration file
// example if localConfig.json is located in the root of the
project
// setLocalConfigurationFile('localConfig.json');
setLocalConfigurationFile('MapStore2/web/client/
```

```
localConfig.json');

// async load of the standard embedded bundle
import('@mapstore/product/embedded');
```

2 - update the path of the embedded entry inside the `webpack.config.js` and `prod-webpack.config.js` files with:

```
// __PROJECTNAME__ is the name of the project used in the
creation process
'__PROJECTNAME__-embedded': path.join(__dirname, "js",
"embedded"),
```

Locate plugin configuration

Configuration for Locate plugin has changed and it is not needed anymore inside the Map plugin

- old localConfig.json configuration needed 'locate' listed as tool inside the Map plugin and as a separated Locate plugin

```
// ...
{
    "name": "Map",
    "cfg": {
        "tools": ["locate"],
        // ...
    }
},
{
    "name": "Locate",
    // ...
}
// ...
```

- new localConfig.json configuration removes 'locate' from tools array and it keeps only the plugin configuration

```
```

```
// ...
{
    "name": "Map",
    "cfg": {
        // ...
    }
},
{
    "name": "Locate",
    // ...
}
// ...
```

Update an existing project to include embedded Dashboards and GeoStories

Embedded Dashboards and GeoStories need a new set of javascript entries, html templates and configuration files to make them completely available in an existing project.

The steps described above assume this structure of the MapStore2 project for the files that need update:

```
MapStore2Project/
|-- ...
|-- js/
|    |-- ...
|    |-- dashboardEmbedded.jsx (new)
|    |-- geostoryEmbedded.jsx (new)
|-- MapStore2/
|-- web/
|    |-- ...
|    |-- pom.xml
|-- ...
|-- dashboard-embedded-template.html (new)
|-- dashboard-embedded.html (new)
|-- ...
|-- geostory-embedded-template.html (new)
|-- geostory-embedded.html (new)
|-- ...
|-- prod-webpack.config.js
|-- ...
|-- webpack.config.js
```

1) create the entries files for the embedded application named `dashboardEmbedded.jsx` and `geostoryEmbedded.jsx` in the js/ folder with the following content (see links): - dashboardEmbedded.jsx - geostoryEmbedded.jsx

2) add the html files and templates in the root directory of the project with these names and content (see links): - dashboard-embedded-template.html - dashboard-embedded.html - geostory-embedded-template.html - geostory-embedded.html

3) update webpack configuration for development and production with the new entries and the related configuration:

- webpack.config.js

```js
module.exports = require('./MapStore2/build/buildConfig')(
    {
        // other entries...,
        // add new embedded entries to entry object
        "geostory-embedded": path.join(__dirname, "js",
"geostoryEmbedded"),
        "dashboard-embedded": path.join(__dirname, "js",
"dashboardEmbedded")
    },
    // ...
);
```

- prod-webpack.config.js

```js

module.exports = require('./MapStore2/build/buildConfig')(
    {
        // other entries...,
        // add new embedded entries to entry object
        "geostory-embedded": path.join(__dirname, "js",
"geostoryEmbedded"),
        "dashboard-embedded": path.join(__dirname, "js",
"dashboardEmbedded")
    },
    // ...
    [
        // new HtmlWebpackPlugin({ ... }),
        // add plugin to copy all the embedded html and inject
the correct bundle
        new HtmlWebpackPlugin({
            template: path.join(__dirname, 'geostory-embedded-
template.html'),
            chunks: ['geostory-embedded'],
            inject: "body",
            hash: true,
            filename: 'geostory-embedded.html'
        }),
        new HtmlWebpackPlugin({
            template: path.join(__dirname, 'dashboard-embedded-
template.html'),
            chunks: ['dashboard-embedded'],
            inject: 'body',
```

```
            hash: true,
            filename: 'dashboard-embedded.html'
        })
    ],
    // ...
);
```

4) Add configuration to localConfig.json in the plugins section related to Share functionalities (Only with custom localConfig.json in the project): - Dashboard share configuration

```js
"dashboard": [
    // ...
    {
        "name": "Share",
        "cfg": {
            "showAPI": false,
            "advancedSettings": false,
            "shareUrlRegex": "(h[^#]*)#\\/dashboard\\/([A-Za-
z0-9]*)",
            "shareUrlReplaceString": "$1dashboard-
embedded.html#/$2",
            "embedOptions": {
                "showTOCToggle": false,
                "showConnectionsParamToggle": true
            }
        }
    },
    // ...
]
```

- Dashboard share configuration

```js
"geostory": [
    // ...
    {
        "name": "Share",
        "cfg": {
            "embedPanel": true,
            "showAPI": false,
            "advancedSettings": {
                "hideInTab": "embed",
                "homeButton": true,
                "sectionId": true
            },
            "shareUrlRegex": "(h[^#]*)#\\/geostory\\/([^\\/]*)\
\/([A-Za-z0-9]*)",
            "shareUrlReplaceString": "$1geostory-embedded.html#/
$3",
            "embedOptions": {
                "showTOCToggle": false
            }
        }
    },
    // ...
```

```
]
```

5) update the web/pom.xml to copy all the related resources in the final *.war file with these new executions

```
<!-- __PROJECTNAME__ should be equal to the one in use in the
project, see other executions how they define the
outputDirectory path  -->
<execution>
    <id>only dashboard-embedded.html</id>
    <phase>process-classes</phase>
    <goals>
        <goal>copy-resources</goal>
    </goals>
    <configuration>
        <outputDirectory>${basedir}/target/__PROJECTNAME__</
outputDirectory>
        <encoding>UTF-8</encoding>
        <resources>
            <resource>
                <directory>${basedir}/../dist</directory>
                <includes>
                    <include>dashboard-embedded.html</include>
                </includes>
                <excludes>
                    <exclude>MapStore2/*</exclude>
                    <exclude>MapStore2/**/*</exclude>
                </excludes>
            </resource>
        </resources>
    </configuration>
</execution>
<execution>
    <id>only geostory-embedded.html</id>
    <phase>process-classes</phase>
    <goals>
        <goal>copy-resources</goal>
    </goals>
    <configuration>
        <outputDirectory>${basedir}/target/__PROJECTNAME__</
outputDirectory>
        <encoding>UTF-8</encoding>
        <resources>
            <resource>
                <directory>${basedir}/../dist</directory>
                <includes>
                    <include>geostory-embedded.html</include>
                </includes>
                <excludes>
```

```
                    <exclude>MapStore2/*</exclude>
                    <exclude>MapStore2/**/*</exclude>
                </excludes>
            </resource>
        </resources>
    </configuration>
</execution>
```

Migration from 2020.02.00 to 2021.01.00

Update to webpack 5 - Module federation

MapStore migrated to webpack 5 and provided the extension system using "Webpack Module Federation". Here the steps to update the existing files in your project.

package.json:

- dev server scripts changed syntax. now you need to use `webpack serve` instead of `webpack-dev-server` . Replace also all `--colors` with `--color` in your scripts that use webpack / webpack-dev-server.
- Align `dependencies` and `devDependencies` with MapStore's one, reading the `package.json` , as usual.
- To support extensions in your project, you need to add `ModuleFederationPlugin` to your `prod-webpack.config.js` and `webpack.config.js`

```
const ModuleFederationPlugin = require('./MapStore/build/
moduleFederation').plugin; // <-- new line
module.exports = require('./buildConfig')(
    assign({
        "mapstore2": path.join(paths.code, "product", "app"),
        "embedded": path.join(paths.code, "product",
"embedded"),
        "ms2-api": path.join(paths.code, "product", "api")
    },
    require('./examples')
    ),
    themeEntries,
    paths,
```

```
    extractThemesPlugin,
    [extractThemesPlugin, ModuleFederationPlugin], // <-- this
parameter has been changed, now it accepts also array of the
plugins you want to add bot in prod and dev
```

Other the other changes required are applied automatically in `buildConfig.js`.

Eslint config

Now eslint configuration is shared in a separate npm module. To update your custom project you have to remove the following files:

- `.eslintignore`
- `.eslintconfig`

And add to `package.json` the following entry, in the root:

```
        "eslintConfig": {
            "extends": [
                "@mapstore/eslint-config-mapstore"
            ],
            "parserOptions": {
                "babelOptions": {
                    "configFile": "./MapStore2/build/
babel.config.js"
                }
            }
        },
```

If you have aproject that includes MapStore as a dependency, you can run `npm run updateDevDeps` to finalize the update. Otherwise make you sure to include:

- devDependencies:
- add `"@mapstore/eslint-config-mapstore": "1.0.1",`
- delete `babel-eslint`
- dependencies:
- update `"eslint": "7.8.1"

App structure review

From this version some base components of MapStore App (`StandardApp` , `StandardStore` ...) has been restructured and better organized. Here a list of the breaking change you can find in a depending project

- `web/client/product/main.jsx` has been updated to new `import` and `export` syntax (removed `require` and `exports.module`). So if you are importing it (usually in your `app.jsx`) you have to use the `import` syntax or use `require(...).default` in your project. The same for the other files.
- New structure of arguments in web/client/stores/StandardStore.js

```
const appStore = (
    {
        initialState = {
            defaultState: {},
            mobile: {}
        },
        appReducers = {},
        appEpics = {},
        rootReducerFunc = ({ state, action, allReducers }) =>
allReducers(state, action)
    },
    plugins = {},
    storeOpts = {}
) {
  ...
```

- Moved standard epics, standard reducers and standard rootReducer function from web/client/stores/StandardStore.js to a separated file web/client/stores/defaultOptions.js
- loading extensions functionalities inside StandardApp has been moved to an specific withExtensions HOC, so if you are not using `main.js` but directly `StandardApp` and you need extensions you need to add this HOC to your StandardApp

Migration from 2020.01.00 to 2020.02.00

New authentication rule for internal services

With this new version the support for uploading extensions has been introduced. A new entry point needs administration authorization to allow the upload of new plugins by the administrator. So:

- In `localConfig.json` add the following entry in the `authenticationRules` array:

```
{
    "urlPattern": ".*rest/config.*",
    "method": "bearer"
  }
```

the final entry should look like this

```
 "authenticationRules": [{
        "urlPattern": ".*geostore.*",
        "method": "bearer"
      }, {
        "urlPattern": ".*rest/config.*",
        "method": "bearer"
      }, ...],
```

Translation files

- The translations file extension has been changed into JSON. Now translation files has been renamed from `data.<locale>` to `data.<locale>.json`. This change makes the `.json` extension mandatory for all translation files. This means that depending projects with custom translation files should be renabled in the same name. E.g. `data.it-IT` have to be renamed as `data.it-IT.json`

Database Update

Database schema has changed. To update your database you need to apply this SQL scripts to your database

- Update the user schema run the script available here:

```
-- Update the geostore database from 1.4.2 model to 1.5.0
-- It adds fields to gs_security for external authorization

-- The script assumes that the tables are located into the
schema called "geostore"
--      if you put geostore in a different schema, please edit
the following search_path.
SET search_path TO geostore, public;

-- Tested only with postgres9.1

-- Run the script with an unprivileged application user allowed
to work on schema geostore

alter table gs_security add column username varchar(255);
alter table gs_security add column groupname varchar(255);

create index idx_security_username on gs_security (username);

create index idx_security_groupname on gs_security (groupname);
```

- Add new categories

```
-- New CONTEXT category
INSERT into geostore.gs_category (id ,name) values (
nextval('geostore.hibernate_sequence'),  'CONTEXT') ON CONFLICT
DO NOTHING;
-- New GEOSTORY category (introduced in 2020.01.00)
INSERT into geostore.gs_category (id ,name) values
(nextval('geostore.hibernate_sequence'),  'GEOSTORY') ON
CONFLICT DO NOTHING;
-- New TEMPLATE category
INSERT into geostore.gs_category (id ,name) values (
nextval('geostore.hibernate_sequence'),  'TEMPLATE') ON
CONFLICT DO NOTHING;
-- New USERSESSION category
INSERT into geostore.gs_category (id ,name) values (
```

```
  nextval('geostore.hibernate_sequence'),  'USERSESSION') ON
  CONFLICT DO NOTHING;
```

Backend update

For more details see this commit

new files have been added:

- `web/src/main/webapp/WEB-INF/dispatcher-servlet.xml`
- `web/src/main/resources/mapstore.properties`

some files has been changed:

- `web/src/main/webapp/WEB-INF/web.xml`
- `pom.xml`
- `web/pom.xml`

Migration from 2019.02.01 to 2020.01.00

With MapStore **2020.01.00** some dependencies that were previously hosted on github, have now been published on the npm registry, and package.json has been updated accordingly. Here is the PR that documents how to update local package.json and local webpack if not using the mapstore buildConfig/ testConfig common files.

After updating package.json run **npm install** Now you should be able to run locally with **npm start**

For more info see the related issue

Moreover a new category has been added for future features, called GEOSTORY.

It is not necessary for this release, but, to follow the update sequence, you can add it by executing the following line.

```
```

```
INSERT into geostore.gs_category (id ,name) values
(nextval('geostore.hibernate_sequence'),  'GEOSTORY') ON
CONFLICT DO NOTHING;
```

Migration from 2019.01.00 to 2019.01.01

MapStore **2019.01.01** changes the location of some of the build and test configuration files. This also affects projects using MapStore build files, sp if you update MapStore subproject to the **2019.01.01** version you also have to update some of the project configuration files. In particular:

- **webpack.config.js** and **prod-webpack.config.js**:
- update path to themes.js from ./MapStore2/themes.js to ./MapStore2/build/themes.js
- update path to buildConfig from ./MapStore2/buildConfig to ./MapStore2/build/buildConfig
- **karma.conf.continuous-test.js** and **karma.config.single-run.js**: update path to testConfig from ./MapStore2/testConfig to ./MapStore2/build/testConfig

Migration from 2017.05.00 to 2018.01.00

MapStore **2018.01.00** introduced theme and js and css versioning. This allows to auto-invalidates cache files for each version of your software. For custom projects you could choose to ignore this changes by setting version: "no-version" in your app.jsx `StandardRouter` selector:

```
//...
const routerSelector = createSelector(state => state.locale,
(locale) => ({
    locale: locale || {},
    version: "no-version",
    themeCfg: {
        theme: "mythheme"
    },
    pages
}));
```

```
const StandardRouter = connect(routerSelector)(require('../
MapStore2/web/client/components/app/StandardRouter'));
//...
```

Support js/theme versioning in your project

Take a look to this pull request as reference. Basically versioning is implemented in 2 different ways for css and js files :

- Add at build time the js files inclusion to the files, with proper hashes.
- Load theme css files appending to the URL the ?{version} where version is the current mapstore2 version The different kind of loading for css files is needed to continue supporting the theme switching capabilities. For the future we would like to unify these 2 systems. See this issue.

You have to:

- Add the version file to the root (`version.txt`).
- Create a template for each html file you have. These files will replace the html files when you build the final war file. These files are like the original ones but without the `[bundle].js` file inclusion and without theme css.
- Add `HtmlWebpackPlugin` for production only, one for each js file. This plugin will add to the template file the script inclusion (example).
- if you have to include the script in the head (e.g. `api.html` has some script that need the js to be loaded before executing the inline scripts), use the option `inject: 'head'`
- change each entry point (`app.jsx` , `api.jsx` , `embedded.jsx` , `yourcoustomentrypoint.jsx`) this way (example):
- `version` reducer in `StandardRouter`
- `loadVersion` action in `initialActions`
- `version` and `loadAfterTheme` selectors to `StandardRouter` state.

```
// Example
const {versionSelector} = require('../MapStore2/web/client/
selectors/version');
```

```
const {loadVersion} = require('../MapStore2/web/client/actions/
version');
const version = require('../MapStore2/web/client/actions/
version');
//...
StandardRouter = connect ( state => ({
    locale: state.locale || {},
        pages,
        version : versionSelector(state),
        loadAfterTheme: loadAfterThemeSelector(state)
    }))(require('../MapStore2/web/client/components/app/
StandardRouter'))
const appStore = require('../MapStore2/web/client/stores/
StandardStore').bind(null, initialState, {
    // ...
    version: version
});
// ...
const appConfig = {
    // ...
    initialActions: [loadVersion]
}
```

- Add to your `pom.xml` some execution steps to replace html files with the ones generated in 'dist' directory. (example). And copy `version.txt`
- Override the version file in your build process (e.g. you can use the commit hash)

Migration from 2017.05.00 to 2017.03.00 and previews

In **2017.03.00** the `createProject.js` script created only a custom project. From version 2017.04.00 we changed the script to generate 2 kind of projects:

- **custom**: the previous version
- **standard**: mapstore standard

Standard project wants to help to generate a project that is basically the MapStore product, where you can add your own plugins and customize your theme (before this you had to create a project similar to MapStore on your own)

Depending on our usage of custom project, this may introduce some breaking changes. If you previously included some file from `product` folder, now `app.jsx` has been changed to call `main.jsx`. Please take a look on how the main product uses this to migrate your changes inside your custom project.

Migration from 2017.01.00 to 2017.02.00

The version 2017.02.00 has many improvements and changes:

- introduced `redux-observable`
- updated `webpack` to version 2
- updated `react-intl` to version 2.x
- updated `react` to [version 15.4.2] (https://facebook.github.io/react/blog/2016/04/07/react-v15.html)
- updated `react-bootstrap` to version 0.30.7

We suggest you to:

- align your package.json with the latest version of 2017.02.00.
- update your webpack files (see below).
- update your tests to react 15 version. see upgrade guide
- Update your `react-bootstrap` custom components with the new one (see below).

Side Effect Management - Introduced redux-observable

To manage complex asynchronous operations the thunk middleware is not enough. When we started with MapStore there was no alternative to thunk. Now we have some options. After a spike (results available here) we chose to use redux-observable. For the future, we strongly recommend to use this library to perform asynchronous tasks.

Introducing this library will allow to :

- remove business logic from the components event handlers

- now all new `actionCreators` should return pure actions. All async stuff will be deferred to the epics.
- avoid bouncing between components and state to trigger side effect
- speed up development with `rxjs` functionalities
- Existing thunk integration will be maintained since all the thunks will be replaced.

If you are using the Plugin system and the StandardStore, you may have only to include the missing new dependencies in your package.json (redux-observable and an updated version of redux).

Check the current package.json to get he most recent versions. For testing we included also redux-mockup-store as a dependency, but you are free to test your epics as you want.

For more complex integrations check this pull request to see how to integrate redux-observable or follow the guide on the redux-observable site.

Webpack update to version 2

We updated webpack (old one is deprecated), check this pull request to find out how to update your webpack files. here a list of what we had to update:

- module.loaders are now module.rules
- update your package.json with latest versions of webpack, webpack plugins and karma libs and integrations (Take a look to the changes on package.json in the pull request if you want a detailed list of what to update in this case).
- change your test proxy configuration with the new one.

More details on the webpack site.

react-intl update to 2.x

See this pull request for the details. You should only have to update your package.json

react update to 15.4.2

Check this pull request to see how to:

- update your package.json
- update your tests

React Bootstrap update

The version we are using is not documented anymore, and not too much compatible with react 15 (too many warnings). So this update can not be postponed anymore. The bigger change in this case is that the Input component do not exists anymore. You will have to replace all your Input with the proper components, and update the `package.json`. See this pull request for details.

How to release

Below you can find the release procedure as a checklist. On each release it can be updated (if needed), copied and pasted into a GitHub issue, of course changing the release name. Then you can check each entry on the GitHub issue when done until the release is end.

Changelog generation

Add an entry in the changelog like this:

```
## [2018.02.00](https://github.com/geosolutions-it/MapStore2/
tree/v2018.02.00) (2018-09-11)

 - **[Full Changelog](https://github.com/geosolutions-it/
MapStore2/compare/v2018.01.00...v2018.02.00)**

 - **[Implemented enhancements](https://github.com/geosolutions-
it/MapStore2/issues?
q=is%3Aissue+milestone%3A%222018.02.00%22+is%3Aclosed+label%3Aenha

 - **[Fixed bugs](https://github.com/geosolutions-it/MapStore2/
issues?
q=is%3Aissue+milestone%3A%222018.02.00%22+is%3Aclosed+label%3Abug)

 - **[Closed issues](https://github.com/geosolutions-it/
MapStore2/issues?
q=is%3Aissue+milestone%3A%222018.02.00%22+is%3Aclosed)**
```

Replacing:

- replacing `2022.01.00` with branch name
- with current release tag name `v2018.02.00`
- `%22v2022.01.00%22` with the name of the milestone (`%22` are `"` in the URL to generate a filter like `milestone:"v2022.01.00"`)

Release Checklist

naming conventions

release and tag

- **vYYYY.XX.mm** name of the release and tag. (e.g. `v2022.01.01`)
- **YYYY** is the year,
- **XX** is the incremental number of the release for the current year (starting from 01)
- **mm** is an incremental value (starting from 00) to increment for minor releases

stable branch

- **YYYY.XX.xx** name of stable branch (e.g. `2022.01.xx`)
- **YYYY** is the year
- **XX** is the incremental number of the release for the current year (starting from 01)
- **xx** is the fixed text `xx`

Release procedure

```
- [ ] Create an issue with this checklist in the release
milestone.
- [ ] Verify if it is needed to release a new version of
http_proxy, mapfish print or geostore, and do it if necessary.
Instruction for GeoStore [here](https://github.com/geosolutions-
it/geostore/wiki/Release-Process) and MapFish Print [here]
(https://github.com/geosolutions-it/mapfish-print/wiki/How-to-
Release)
  - [ ] for geostore, check if [here](https://maven.geo-
solutions.it/it/geosolutions/geostore/geostore-webapp/) is
present the version specified in the [release calendar 2022]
(https://github.com/geosolutions-it/MapStore2/wiki/MapStore-
Releases-2022)
  - [ ] for http_proxy, check if [here](https://
mvnrepository.com/artifact/proxy/http_proxy) is present the
```

```
version specified in the [release calendar 2022](https://
github.com/geosolutions-it/MapStore2/wiki/MapStore-
Releases-2022)
- [ ] If major release (YYYY.XX.00), create a branch
`YYYY.XX.xx`  (`xx` is really `xx`, example: 2018.01.xx)
- [ ] If major release, update the default stable branch used
in createProject.js script , in particular the utility/projects/
projectLib.js file
- [ ] If major release, Change [QA Jenkins job](http://
build.geosolutionsgroup.com/view/MapStore/job/MapStore/view/
MapStore%20QA/job/MapStore2-QA-Build/) to build the new branch,
enable the job continuous deploy by updating the `branch`
parameter in the build configuration page to `YYYY.XX.xx`
- [ ] Check version in `package.json`. (as for semantic
versioning the major have to be 0 until the npm package has not
a stable API).
    - [ ] Take note of current version of mapstore in
`package.json` in master branch, it should be in the form 0.x.0
    - [ ] If major release, make pr and merge on master
**0.&lt;x-incremented&gt;.0**
    - [ ] if minor release, make pr and merge on stable
YYYY.XX.xx **0.x.&lt;number-of-minor-version&gt;**
- [ ] Create a milestone on GitHub with the same name of the
release (vYYYY.XX.xx)
    - [ ] assign the label "current-release" to all the issues
and Prs of the current zenhub release
    - [ ] use the label to filter the issues on github and
assign to all the issues and Prs the milestone created
    - [ ] remove assignments of "current-release"
- [ ] Prepare PR for updating `CHANGELOG.md` for **master** and
**stable** [Instructions](https://mapstore.readthedocs.io/en/
latest/developer-guide/release/#changelog-generation)
- [ ] Fix `pom.xml` dependencies stable versions ( no `-
SNAPSHOT` usage release).
- [ ] Update the version of java modules on the stable branch
to a stable, incremental version.
    - [ ] Run `mvn release:update-versions -
DdevelopmentVersion=<VERSION> -Pprinting,printingbundle,binary`
    to update package version, where <VERSION> is the version
of the java packages (e.g. `1.3.1`).
    - [ ] Manually update project pom templates to use
`mapstore-services` of `<VERSION>`
- [ ] Release a stable `mapstore-services`. (from `2022.01.xx`
also mapstore-webapp (java/web) should be deployed for new
project system).
  - [ ] Use `mvn clean install deploy -f java/pom.xml` to
deploy `mapstore-services` and `mapstore-webapp`.
- [ ] create on [ReadTheDocs](https://readthedocs.org/projects/
mapstore/) project the version build for `YYYY.XX.xx` (click on
```

```
"Versions" and activate the version of the branch)
- [ ] Test on QA [https://qa-mapstore.geosolutionsgroup.com/
mapstore/](https://qa-mapstore.geosolutionsgroup.com/mapstore/)
    * Any fix must be done on **YYYY.XX.xx**. The fixes will be
manually merged on master
    * Test **everything**, not only the new features
    * Test the creation of a standard project starting in from
the stable branch and with the internal backend, so `npm run
backend` and `npm start`, then check that an empty homepage
loads correctly
- [ ] Test [Binary](http://build.geosolutionsgroup.com/view/
MapStore/job/MapStore/view/MapStore%20QA/job/MapStore2-QA-
Build/) (take the mapstore2-<RELEASE_BRANCH>-qa-bin.zip, from
latest build)
- [ ] Lunch the [stable deploy](http://
build.geosolutionsgroup.com/view/MapStore/job/MapStore/view/
MapStore%20Stable/job/MapStore2-Stable-Deploy/) to install the
latest stable version on official demo, remember to change
version to **YYYY.XX.mm**
- [ ] Manually edit the `localConfig.json` on
mapstore.geosolutionsgroup.com to fit the authkey for
production (change from `authkey-qa` to `authkey-prod`)
  - [ ] `ssh geosolutions@mapstore.geosolutionsgroup.com`
  - [ ] `sudo su`
  - [ ] `vim /var/lib/tomcats8/mapstore2_release/webapps/
mapstore/configs/localConfig.json`
  - [ ] to test the change has been applied, login on
mapstore.geosolutionsgroup.com and verify that the layers from
`gs-stable` are visible without errors (typically
authentication errors that was caused by the wrong auth-key).
- [ ] Commit the changelog to the stable branch **YYYY.XX.xx**
- [ ] Create a [github draft release](https://github.com/
geosolutions-it/MapStore2/releases)
  - [ ] `branch` **YYYY.XX.xx**
  - [ ] `tag` **vYYYY.XX.mm** (create a new tag from UI after
entering this value)
  - [ ] `release` name equal to tag **vYYYY.XX.mm**
  - [ ] `description` describe the major changes and add links
of the Changelog paragraph.
- [ ] Launch [MapStore2-Releaser](http://
build.geosolutionsgroup.com/view/MapStore/job/MapStore/view/
MapStore%20Stable/job/MapStore2-Stable-Build/) Jenkins job with
**YYYY.XX.mm** for the version and **YYYY.XX.xx** for the
branch to build and  **wait the end**). **Note:** Using the
MapStore2 Releaser allows to write the correct version number
into the binary packages.
    - [ ] Get the [latest mapstore.war](http://
build.geosolutionsgroup.com/view/MapStore/job/MapStore/view/
MapStore%20Stable/job/MapStore2-Stable-Build/ws/product/target/
```

```
mapstore.war) from the Releaser Jenkins build
    - [ ] Get the [latest mapstore2-YYYY.XX.mm-bin.zip](http://
build.geosolutionsgroup.com/view/MapStore/job/MapStore/view/
MapStore%20Stable/job/MapStore2-Stable-Build/ws/binary/target/)
from the Releaser Jenkins build
    > from the job [configuration page](http://
build.geosolutionsgroup.com/view/MapStore/job/MapStore/view/
MapStore%20Stable/job/MapStore2-Stable-Build/ws/) there is a
link to access the job workspace to easily download the built
WAR and binary package
    - [ ] Download `mapstore-printing.zip` [here](http://
build.geosolutionsgroup.com/view/MapStore/job/MapStore/view/
MapStore%20Stable/job/MapStore2-Stable-Build/ws/java/printing/
target/mapstore-printing.zip) from the Releaser Jenkins build
workspace
    - [ ] Check that the printing plugin is missing in the
binary package to release
    - [ ] Remove manually from `localConfig.json` the entry for
authentication to gs-stable from binary and war packages.
    - [ ] Upload the updated binary, the war package and
`mapstore-printing.zip` on github release
    - [ ] Publish the release
- [ ] create on [ReadTheDocs](https://readthedocs.org/projects/
mapstore/) project the version build for `vYYYY.XX.mm` (click
on "Versions" and activate the version of the tag, created when
release was published)
- [ ] Prepare a PR towards master branch **YYYY.XX.xx** in
order to reset versions of java modules to `-SNAPSHOT`
    - `mvn versions:set -DnewVersion=<SNAPSHOT_VERSION> -
DprocessAllModules -DgenerateBackupPoms=false`
    where `<SNAPSHOT_VERSION>` is the version to set. (e.g. 1.2-
SNAPSHOT).
    make sure that only mapstore-services has changed

- [ ] [Create a draft release](https://github.com/geosolutions-
it/MapStoreExtension/releases/new) for https://github.com/
geosolutions-it/MapStoreExtension with the same name and tag
  - [ ] target of the release is **master** branch
  - [ ] tag is **vYYYY.XX.mm**
  - [ ] Update revision of mapstore to the release tag
**vYYYY.XX.mm**
  - [ ] [run the build](https://github.com/geosolutions-it/
MapStoreExtension#build-extension) locally and attach to the
release the file `SampleExtension.zip` from the `/dist` folder
  - [ ] create a PR for the changes of the revision to the
MapstoreExtension repo
  - [ ] Merge the PR
  - [ ] Publish the release
- [ ] Create a blog post
```

```
- [ ] Write to the mailing list about the current release news
and the next release major changes
- [ ] Optional - prepare a PR for updating release procedure
- [ ] Close this issue
- [ ] Close the related milestone **vYYYY.XX.mm**
```

Developer Generic Guidelines

This guide wants to provide general information and suggestions about how to write code for MapStore. It contains a practical guide to write plugins for MapStore, then some general information about writing redux actions, reducers and redux-observble epics, with some hints specific of MapStore.

Still to do:

- Writing enhancers
- Writing components
- Using JS API

Work in progress:

- Extensions

Creating a MapStore2 plugin

The MapStore2 plugins architecture allows building your own independent modules that will integrate seamlessly into your project.

Creating a plugin is like assembling and connecting several pieces together into an atomic module. This happens by writing a plugin module, a ReactJS JSX file exporting the plugin descriptor.

Introduction

During this tutorial, you will learn how to create and configure plugins in a MapStore project. If you don't know how to work with MapStore projects, please read the Projects Guide. For this tutorial, a "standard project" is used.

A plugin example

A plugin is a ReactJS *component with a name*. The chosen name is always suffixed with **Plugin**.

js/plugins/Sample.jsx

```
import React from 'react';

class SampleComponent extends React.Component {
    render() {
        const style = {position: "absolute", top: "100px",
left: "100px", zIndex: 10000000};
        return <div style={style}>Sample</div>;
    }
}

export const SamplePlugin = SampleComponent;
// the Plugin postfix is mandatory, avoid bugs by calling all
your descriptors
// <Something>Plugin
```

Being a component with a name (**Sample** in our case) you can include it in your project by creating a *plugins.js* file:

js/plugins.js

```
module.exports = {
    plugins: {
        ...
        SamplePlugin: require('./plugins/Sample'),
        ...
    }
};
```

Note that SamplePlugin in plugins.js must be called with the same name used when exporting it.

Include the plugin.js from your app.jsx either replacing the plugins import from the product or extending it:

js/app.jsx

```
...

const m2Plugins = require('@mapstore/product/plugins');
```

```
const customPlugins = require('./plugins');
const allPlugins = {...m2Plugins, plugins:
{...customPlugins.plugins, ...m2Plugins.plugins}};
require('@mapstore/product/main')(appConfig, allPlugins);
```

Then you have to configure it properly so that is enabled in one or more application modes / pages:

localConfig.json

```
{
    ...
    "plugins": {
        "desktop": ["Sample", ...],
        ...
    }
}
```

Note: to enable a plugin you need to do two things:

- require it in the plugins.js file
- configure it in localConfig.json (remove the Plugins suffix here)

If one is missing, the plugin won't appear. To globally remove a plugin from your project the preferred way is removing it from plugins.js, because this will reduce the global javascript size of your application.

You can also specify plugins properties in the configuration, using the **cfg** property:

localConfig.json (2)

```
{
    ...
    "plugins": {
        "desktop": [{
            "name": "Sample",
            "cfg": {
                "myproperty": "myvalue"
```

```
            }
        }, ...],
        ...
    }
}
```

A store connected plugin example

A plugin component is a **smart component** (connected to the Redux store) so that properties can be taken from the global state, as needed.

js/plugins/ConnectedSample.jsx (1)

```
import React from 'react';
import {connect} from 'react-redux';
import PropTypes from 'prop-types';
import {get} from 'lodash';

class SampleComponent extends React.Component {
    static propTypes = {
        zoom: PropTypes.number
    };

    render() {
        const style = {position: "absolute", top: "100px",
left: "100px", zIndex: 1000000};
        return <div style={style}>Zoom: {this.props.zoom}</div>;
    }
}

const ConnectedSample = connect((state) => {
    return {
        zoom: get(state, 'map.present.zoom') // connected
property
    };
})(SampleComponent);

export const ConnectedSamplePlugin = ConnectedSample;
```

A plugin can use actions to update the global state.

js/plugins/ConnectedSample.jsx (2)

```
import React from 'react';
import { connect } from 'react-redux';
import PropTypes from 'prop-types';
import {get} from 'lodash';

import { changeZoomLevel } from '../../MapStore2/web/client/
actions/map';

class SampleComponent extends React.Component {
    static propTypes = {
        zoom: PropTypes.number,
        onZoom: PropTypes.func
    };

    render() {
        const style = { position: "absolute", top: "100px",
left: "100px", zIndex: 1000000 };
        return <div style={style}>Zoom: {this.props.zoom}
<button onClick={() => this.props.onZoom(this.props.zoom + 1)}
>Increase</button></div >;
    }
}

const ConnectedSample = connect((state) => {
    return {
        zoom: get(state, 'map.present.zoom')
    };
}, {
        onZoom: changeZoomLevel // connected action
    })(SampleComponent);

export const ConnectedSamplePlugin = ConnectedSample;
```

A plugin can define its own state fragments and the related reducers. Obviously you will also be able to define your own actions.

js/actions/sample.js

```
export const UPDATE_SOMETHING = 'SAMPLE:UPDATE_SOMETHING';
export const updateSomething = (payload) => {
    return {
```

```
        type: UPDATE_SOMETHING,
        payload
    };
};
```

js/reducers/sample.js

```
import { UPDATE_SOMETHING } from '../actions/sample';
export default function(state = { text: 'Initial Text' },
action) {
    switch (action.type) {
        case UPDATE_SOMETHING:
            return {
                text: action.payload
            };
        default:
            return state;
    }
}
```

js/plugins/ConnectedSample.jsx (3)

```
import React from 'react';
import { connect } from 'react-redux';
import PropTypes from 'prop-types';
import {get} from 'lodash';

import { updateSomething } from '../actions/sample';
import sample from '../reducers/sample';

class SampleComponent extends React.Component {
    static propTypes = {
        text: PropTypes.string,
        onUpdate: PropTypes.func
    };

    render() {
        const style = { position: "absolute", top: "100px",
left: "100px", zIndex: 1000000 };
        return <div style={style}>Text: {this.props.text}
<button onClick={() => this.props.onUpdate('Updated Text')}
>Update</button></div >;
```

```
    }
}

const ConnectedSample = connect((state) => {
    return {
        text: get(state, 'sample.text')
    };
}, {
        onUpdate: updateSomething // connected action
    })(SampleComponent);

export const ConnectedSamplePlugin = ConnectedSample;
export const reducers = {sample};
```

Data fetching and side effects

Side effects should be limited as much as possible, but there are cases where a side effect cannot be avoided. In particular all asynchronous operations are side effects in Redux, but we obviously need to handle them, in particular we need to asynchronously load the data that we need from ore or more web services.

To handle data fetching a plugin can define Epics. To have more detail about epics look at the Epics developers guide section of this documentation.

js/actions/sample.js

```
// custom action
export const LOAD_DATA = 'SAMPLE:LOAD_DATA';
export const LOADED_DATA = 'SAMPLE:LOADED_DATA';
export const LOAD_ERROR = 'SAMPLE:LOAD_ERROR';
export const loadData = () => {
    return {
        type: LOAD_DATA
    };
};

export const loadedData = (payload) => {
    return {
        type: LOADED_DATA,
        payload
    };
};
```

```
export const loadError = (error) => {
    return {
        type: LOAD_ERROR,
        error
    };
};
```

js/reducers/sample.js

```
import { LOADED_DATA, LOAD_ERROR } from '../actions/sample';
export default function(state = { text: 'Initial Text' },
action) {
    switch (action.type) {
        case LOADED_DATA:
            return {
                text: action.payload
            };
        case LOAD_ERROR:
            return {
                error: action.error
            };
        default:
            return state;
    }
}
```

js/epics/sample.js

```
import * as Rx from 'rxjs';
import axios from 'axios';

import { LOAD_DATA, loadedData, loadError } from '../actions/
sample';

export const loadDataEpic = (action$) => {
    return action$.ofType(LOAD_DATA)
        .switchMap(() => {
            return Rx.Observable.defer(() =>
axios.get('version.txt'))
                .switchMap((response) =>
Rx.Observable.of(loadedData(response.data)))
```

```
                .catch(e =>
Rx.Observable.of(loadError(e.message)));
        });
};

export default {
    loadDataEpic
};
```

js/plugins/SideEffectComponent.jsx

```
import React from 'react';
import { connect } from 'react-redux';
import PropTypes from 'prop-types';
import {get} from 'lodash';

import { loadData } from '../actions/sample';
import sampleEpics from '../epics/sample';
import sample from '../reducers/sample';

class SideEffectComponent extends React.Component {
    static propTypes = {
        text: PropTypes.string,
        onLoad: PropTypes.func
    };

    render() {
        const style = { position: "absolute", top: "100px",
left: "100px", zIndex: 1000000 };
        return <div style={style}>Text: {this.props.text}
<button onClick={this.props.onLoad}>Load</button></div >;
    }
}

const ConnectedSideEffect = connect((state) => {
    return {
        text: get(state, 'sample.text')
    };
}, {
        onLoad: loadData // connected action
    })(SideEffectComponent);

export const SideEffectPlugin = ConnectedSideEffect;
export const reducers = {sample};
export const epics = sampleEpics;
```

Plugins that are containers of other plugins

It is possible to define **Container** plugins, that are able to receive a list of *items* from the plugins system automatically. Think of menus or toolbars that can dynamically configure their items / tools from the configuration.

In addition to those "user defined" containers, there is always a **root container**. When no container is specified for a plugin, it will be included in the root container.

js/plugins/ContainerComponent.jsx

```
import React from 'react';
import PropTypes from 'prop-types';

class SampleContainer extends React.Component {
    static propTypes = {
        items: PropTypes.array
    };
    renderItems = () => {
        return this.props.items.map(item => {
            const Item = item.plugin; // item.plugin is the
plugin ReactJS component
            return <Item id={item.id} name={item.name} />;
        });
    };

    render() {
        const style = { zIndex: 1000, border: "solid black
1px", width: "200px", height: "200px", position: "absolute",
top: "100px", left: "100px" };
        return <div style={style}>{this.renderItems()}</div>;
    }
}

export const ContainerPlugin = SampleContainer;
```

Plugins for other plugins

Since we have containers, we can build plugins that can be contained in one or more container plugins.

js/plugins/ContainedComponent.jsx

```
```

```
import React from 'react';
import { connect } from 'react-redux';
import PropTypes from 'prop-types';
import {get} from 'lodash';
import assign from 'object-assign';

import sample from '../reducers/sample';

class SampleComponent extends React.Component {
    static propTypes = {
        text: PropTypes.string
    };

    render() {
        const style = { position: "absolute", top: "100px",
left: "100px", zIndex: 1000000 };
        return <div style={style}>Text: {this.props.text}</div
>;
    }
}

const ConnectedSample = connect((state) => {
    return {
        text: get(state, 'sample.text')
    };
})(SampleComponent);

export const ContainedPlugin = assign(ConnectedSample, {
    // we support the previously defined Container Plugin as a
    // possible container for this plugin
    Container: {
        name: "Sample",
        id: "sample_tool",
        priority: 1
    }
});
export const reducers = {sample};
```

Each section defines a possible container for the plugin, as the name of another plugin (*Container* in the example). The properties in it define the plugin behaviour in relation to the container (e.g. id of the item).

Containers will receive a list of items similar to this:

```
```

```
items = [{plugin: ConnectedSample, name: "Sample", id:
"sample_tool", ...}]
```

Notice that also container related properties can be overridden in the application configuration, using the override property:

localConfig.json

```
{
    ...
    "plugins": {
        "desktop": [{
            "name": "Sample",
            "override": {
                "Container": {
                    "name": "Another Sample"
                }
            }
        }, ...],
        ...
    }
}
```

Plugins Configuration

We have already mentioned that plugins can be configured through the localConfig.json file. The simplest configuration is needed to include the plugin in a particular application mode, and is accomplished by listing the plugin name in the plugins array of the chosen mode:

localConfig.json

```
{
    ...
    "plugins": {
        "desktop": ["Sample", ...],
        ...
```

```
    }
}
```

To customize a plugin style and behaviour a JSON object can be used instead, specifying the plugin name in the **name** property, and the plugin configuration in the **cfg** property.

```
{
    ...
    "plugins": {
        "desktop": [{
            "name": "Sample",
            "cfg": {
                "text": "my text"
            }
            ...
        }, ...],
        ...
    }
}
```

Dynamic configuration

Configuration can also dynamically change when the application state changes. This is accomplished by using expressions in configuration values. An expression is a value of the following form:

```
"property: "{expression}"
```

The expression itself is javascript code (supported by the browser, babel transpiled code is not supported here) where you can use the following variables:

- *request*: **request URL** parsed by the url library
- *context*: anything defined in **plugins.js requires section**
- *state*: a function usable to extract values from the **application state** (e.g. state('map.present.zoom' to get current zoom))

Note that not all the application state is available through the state function, only the *monitored state* is. To add new fragments the monitored state, you can add the following to localConfig.json:

```
{
    ...,
    "monitorState": [
      {"name": "router", "path": "router.location.pathname"},
      {"name": "browser", "path": "browser"},
      {"name": "featuregridmode", "path": "featuregrid.mode"}],
    ...
}
```

The default monitored state is:

```
{name: "mapType", path: 'maptype.mapType'}, {name: "user",
path: 'security.user'}
```

Example

```
{
    ...,
    "plugins": {
        "desktop": [{
            "name": "Sample",
            "cfg": {
                "text": "{state('mapType') === 'leaflet' ?
'Leaflet Map' : 'OpenLayers Map'}"
            }
            ...
        }, ...],
        ...
    }
    ...
}
```

Container configuration

Each plugin can define a list of supported containers, but it's the plugin system that decides which ones will be used at runtime based on:

- container existance: if a container is not configured, it will not be used (obviously)
- between the existing ones, the ones with the highest priority property value will be chosen; note that a plugin can be included in more than one container if they have the same priority

Example

```
...

module.exports = {
    ContainedSamplePlugin: ConnectedSample,
    Container1: {
        name: "Sample",
        id: "sample_tool",
        priority: 1,
        ...
    },
    Container2: {
        name: "Sample",
        id: "sample_tool",
        priority: 2,
        ...
    },
    Container3: {
        name: "Sample",
        id: "sample_tool",
        priority: 3,
        ...
    }
};
```

If all the containers exist, Container3 will be chosen, the one with highest priority,if not Container2 will be used, and so on.

To explicitly configure plugins containment and introduce custom behaviours (overriding default properties), the **override** configuration property is available.

Using it, you can override the relation between a plugin and its supported containers.

We can change containers relation like this:

```
{
    ...,
    "plugins": {
        "desktop": [{
            "name": "Sample",
            "override": {
                "Container1": {
                    "name": "custom name",
                    "priority": 4
                }
            }
            ...
        }, ...],
        ...
    }
    ...
}
```

This will force the plugin system to choose Container1 instead of Container3, and will override the name property.

There is also a set of options to (dynamically) add/exclude containers:

- **showIn**: can be used to add a plugin to a container or more than one, in addition to the default one (it is an array of container plugin names)
- **hideFrom**: can be used to exclude a plugin from a given container or more than one (it is an array of container plugin names)
- **doNotHide**: can be used to show a plugin in the root container, in addition to the default one

Note that also these properties accept dynamic expressions.

js/plugins/Container.jsx

```
import React from 'react';
import PropTypes from 'prop-types';
```

```
class SampleContainer extends React.Component {
    static propTypes = {
        items: PropTypes.array
    };
    renderItems = () => {
        return this.props.items.map(item => {
            const Item = item.plugin; // item.plugin is the
plugin ReactJS component
            return <Item id={item.id} name={item.name} />;
        });
    };

    render() {
        const style = { zIndex: 1000, border: "solid black
1px", width: "200px", height: "200px", position: "absolute",
top: "100px", left: "100px" };
        return <div style={style}>{this.renderItems()}</div>;
    }
}

export const ContainerPlugin = SampleContainer;
```

js/plugins/ContainerOther.jsx

```
import React from 'react';
import PropTypes from 'prop-types';

class SampleContainer extends React.Component {
    static propTypes = {
        items: PropTypes.array
    };
    renderItems = () => {
        return this.props.items.map(item => {
            const Item = item.plugin; // item.plugin is the
plugin ReactJS component
            return <Item id={item.id} name={item.name} />;
        });
    };

    render() {
        const style = { zIndex: 1000, border: "solid red 1px",
width: "200px", height: "200px", position: "absolute", top:
"100px", left: "100px" };
        return <div style={style}>{this.renderItems()}</div>;
    }
```

```
}

export const ContainerOtherPlugin = SampleContainer;
```

js/plugins/Sample.jsx

```
import React from 'react';
import assign from 'object-assign';

class SampleComponent extends React.Component {
    render() {
        const style = { position: "absolute", top: "100px",
left: "100px", zIndex: 1000000 };
        return <div style={style}>Hello</div >;
    }
}

export const SamplePlugin = assign(SampleComponent, {
    Container: {
        name: "Sample",
        id: "sample_tool",
        priority: 1
    },
    ContainerOther: {
        name: "Sample",
        id: "sample_tool",
        priority: 1
    }
});
```

With this configuration the sample plugin will be shown in both Container and ContainerOther plugins (they have the same priority, so both are picked).

We can change this using `showIn` or `hideFrom` in `localConfig.json`:

localConfig.json

```
{
    ...,
    "plugins": {
        "desktop": [{
            "name": "Sample",
            "showIn": ["Container"]
```

```
            ...
        }, ...],
        ...
    }
    ...
}

or

{
    ...,
    "plugins": {
        "desktop": [{
            "name": "Sample",
            "hideFrom": ["ContainerOther"]
            ...
        }, ...],
        ...
    }
    ...
}
```

We can also add the plugin to the root container, using the doNotHide property (note that this is a container property, so we have to use an override for it):

localConfig.json

```
{
    ...,
    "plugins": {
        "desktop": [{
            "name": "Sample",
            "showIn": ["Container"],
            "override": {
                "Container": {
                    "doNotHide": true
                }
            }
            ...
        }, ...],
        ...
    }
    ...
}
```

Conditionally disabling plugins

Dynamic expression can also be used to enable a plugin only when a specific application state is met, using the **disablePluginIf** property.

```
{
    ...,
    "plugins": {
        "desktop": [{
            "name": "Sample",
            "cfg": {
                "disablePluginIf": "{state('mapType') ===
'cesium'}"
            }
            ...
        }, ...],
        ...
    }
    ...
}
```

The plugin will be disabled in 3D mode.

Lazy loading plugins

You can lazy load your plugins (load them on demand), but only if you define a loading mechanism for your plugin. This is expecially useful for plugins that include big external libraries.

A lazy loaded plugin is not defined by its component, but with a lazy descriptor with:

- a **loadPlugin** function that loads the plugin code and calls the given **resolve** when the plugin is loaded
- an **enabler** function that triggers plugin loading on a specific state change

```
module.exports = {
    LazySamplePlugin: {
        loadPlugin: (resolve) => {
```

```
            // require.ensure allows code splitting through
webpack and
            // creates a different js bundle for the plugin
            require.ensure(['./LazySampleComponent'], () => {
                const LazySamplePlugin = connect(...)
(require('./LazySampleComponent'));
                resolve(LazySamplePlugin);
            });
        },
        enabler: (state) => state.controls.lazyplugin &&
state.controls.lazyplugin.enabled || false
    }
};
```

Testing plugins

As we already mentioned a plugin is a collection of entities that should already have unit tests (components, reducers, actions, selectors, epics). We can limit plugins testing to testing the interactions between these different entities, for example:

- connection of the redux state to the plugins properties
- epics that are related to the plugin lifecycle
- containment relations between plugins

To ease writing a plugin unit test, an helper is available (pluginsTestUtils) that can be used to:

- create a plugin connected with a redux store (**getPluginForTest**), initialized with plugin's defined reducers and epics, and with a given initial state
- get access to the redux store
- get access to the list of actions dispatched to the store
- get access to the list of containers plugins supported by the plugin (you can limit this list by passing your plugins definitions to getPluginForTest)

Examples

js/tests/myplugin-test.js

```
import expect from 'expect';
import React from 'react';
import ReactDOM from 'react-dom';

import MyPlugin from '../MyPlugin';
import { getPluginForTest } from '../../MapStore2/web/client/
plugins/__tests__/pluginsTestUtils';

const initialState = {};

describe('MyPlugin Test', () => {
    beforeEach((done) => {
        document.body.innerHTML = '<div id="container"></div>';
        setTimeout(done);
    });

    afterEach((done) => {

ReactDOM.unmountComponentAtNode(document.getElementById("container
        document.body.innerHTML = '';
        setTimeout(done);
    });

    it('creates MyPlugin with default configuration', () => {
        const {Plugin, store, actions, containers } =
getPluginForTest(MyPlugin, initialState);
        ReactDOM.render(<Plugin />,
document.getElementById("container"));
        expect(document.getElementById('<my plugin
id>')).toExist();
        expect(...);
    });
    // use pluginCfg to override plugins properties
    it('creates MyPlugin with custom configuration', () => {
        const {Plugin, store, actions, containers } =
getPluginForTest(MyPlugin, initialState);
        ReactDOM.render(<Plugin pluginCfg={{
            property: 'value'
        }}/>, document.getElementById("container"));
        expect(document.getElementById('<my plugin
id>')).toExist();
        expect(...);
```

```
    });

    // test connected epics looking at the actions array
    it('test plugin epics', () => {
        const {Plugin, store, actions, containers } =
getPluginForTest(MyPlugin, initialState);
        ReactDOM.render(<Plugin/>,
document.getElementById("container"));
        store.dispatch({
            type: ACTION_CAPTURED_BY_AN_EPIC,
            payload
        });
        expect(actions.filter(action => action.type ===
ACTION_LAUNCHED_BY_AN_EPIC).length).toBe(1);
    });

    // test supported containers
    it('test containers', () => {
        const {Plugin, store, actions, containers } =
getPluginForTest(MyPlugin, initialState, {
            MyContainerPlugin: {}
        });
        ReactDOM.render(<Plugin/>,
document.getElementById("container"));

expect(Object.keys(containers)).toContain('MyContainer');
    });
});
```

General Guidelines

- Components
    - define the plugin component(s) into dedicated JSX file(s), so that they can be reused outside of the plugin
    - connect the component(s) in the plugin JSX file
- State
    - define your own state fragment (and related actions and reducers) to handle internal state, and use existing actions and state fragments from MapStore2 to interact with the framework

- Selectors
    - use existing selectors when possible to connect the state, eventually using reselect to compose them together or with your own selectors
- Avoid as much as possible direct interactions between different plugins; plugins are meant to be independent modules, so they should be able to work if other plugins appear / disappear from the application configuration
- interact with other plugins and the application itself using actions and state sharing
- creating side effects to make plugins interact in more strict ways should not be done at the plugin level, orchestrating different plugins should be delegated at the top (application) level
- use containers configuration to combine plugins in containers

Writing Epics

Most of the asynchronous operations we are doing in MapStore2 are implemented using epics. This guide gives the developer the base concepts about epics and provides the best practices to write and add your epics to a MapStore2 project.

Base Concepts

Epics are the core element of the redux middleware called redux-observable. **redux-observable** is based on **RxJS**.

RxJS is a library for reactive programming using **Observables**, to make it easier to compose asynchronous or callback-based code.

stream The concept of stream is *"sequence of data made available over time."*.

```
--a---b-c---d---X---|->

a, b, c, d are emitted values
X is an error
| is the 'completed' signal
---> is the timeline
```

Observable is the core entity of RxJS and, more generically, of the whole reactive programming paradigm. Basically it is an entity that emits events and can be subscribed to, so that subscribers can intercept the events emitted. This is the entity that implements the concept of **stream** (so *stream* and *Observable* are almost used as synonym).

Subscribing to observables can be hard, so RxJs provides a lot of **operators** to help manipulating and combining observables (so, *streams*). Here an example of how operators allow manipulating an event stream to count clicks:

```
```

```
clickStream:    ---c----c--c----c------c--> <-- Stream of clicks
                vvvvv map(c becomes 1) vvvv <-- operator that
transforms each event into a `1`
                ---1----1--1----1------1--> <-- new stream
returned by the operator
                vvvvvvvvv scan(+) vvvvvvvvv <-- operator that
does the sum
counterStream:  ---1----2--3----4------5--> <-- click count
stream returned by the operator
```

The final stream can be finally subscribed to update, for instance, a counter on the UI.

Versions

At the time of writing this documentation MapStore2 is using RxJS 5.1.1 and redux-observable 0.19.0. So make you sure to check the correct documentation about the current versions of these libraries.

What is an epic

An **epic** is basically nothing more than:

a function that returns a stream of redux actions to emit.

A simple epic in mapstore can be this one:

```
const fetchUserEpic = (action$, store) => action$
    .ofType(MAP_CONFIG_LOADED)
    .filter(() =>
isMapLoadConfigurationEnabled(store.getState()))
    .map({
        type: NOTIFICATION,
        message: "Map Loaded"
    });
);
```

The epic function has 2 arguments:

- `action$` : the stream of redux actions. Every time an action is triggered through redux, it is emitted as an event on the `action$` stream.
- `store` . A *small version* of the redux store, that contains essentially only one important method called `getState()` . This method returns the current redux state object.

This function **must return** a new stream that emits the actions we want to dispatch to redux. The **redux-observable** middleware subscribes to the action streams returned by the epics so the actions will be automatically triggered on redux.

> **NOTE**: **redux-observable** middleware is already added to the MapStore2's StandardStore and StandardApp, so a developer should only take care of creating his own epics and add them to MapStore.

Typically the stream returned by an epic is always listening for new actions and dispatches other actions:

> **actions in, actions out**.

Let's analyze the epic reported as first example:

It returns a stream (arrow function (`=>`) implicit return) manipulating the `action$` stream. It first filters out all the unwanted actions catching only the `MAP_CONFIG_LOADED` action types, then another filter checks the state to verify some condition (typically a `selector` can be used to check the state).

> **NOTE**: **redux-observable** adds an operator to rxjs called `ofType` that simply filters the actions of certain types, passed as argument, but it is not a part of standard RxJS operators.

The events that passed the 2 filters then hit the `map` operator. The map operator simply returns the (action) object:

```
{
    type: NOTIFICATION,
```

```
    message: "Map Loaded"
}
```

This object will be emitted on the returning stream and so the action will be triggered in redux.

Of course instead of emitting the plain object, you can use an action creator, like this:

```
const notifyMapLoaded = (action$, store) => action$
    .ofType(MAP_CONFIG_LOADED)
    .filter(() =>
isMapLoadConfigurationEnabled(store.getState()))
    .map(info({
        message: "Map Loaded"
    }))
);
```

Create complex data flows triggered by actions

Typical operators to start creating a complex data flow are:

- switchMap
- mergeMap
- exhaustMap, forkJoin and many others...

The base concept of all these solutions is to create one or more new streams (using a function passed as argument) and then emit the events on the final Observer.

> Note: Creating Higher order observables (that are basically streams of streams) and merging their events is a common pattern in RxJs, so, mergeMap and switchMap are simpler shortcuts to increase readability and maintainability of the code:
>
> - mergeMap() is just map() + mergeAll()
> - switchMap() is just map() + switch().

Example:

```
const countDown = action$ => action$
    .ofType(START_COUNTDOWN)
    .switchMap( ({seconds}) =>
        Rx.Observable.interval(1000)
            .map( (value) => updateTime(seconds - value))
            .takeUntil(Rx.Observable.timer(seconds * 1000))
    );
```

```
---{seconds: 5}------------------------> action in
vv switchMap vvvvvvvvvvvvvvvvvvvvvvvvvv
--------------{5}--{4}--{3}--{2}--{1}--> action out
```

In this epic, every time `START_COUNTDOWN` action is performed, the `switchMap` operator's argument function will be executed. The argument function of `switchMap` must return an Observable. Every value emitted on this stream will be projected on the main flow.

So on the first `START_COUNTDOWN` the timer starts (`Rx.Observable.interval(1000)`). The timer will emit an incremental value (0, 1, 2, ...) every 1000 milliseconds. This value is used to trigger another action to emit on redux (using an action creator called `updateTime` , for instance).

At the end the stream will be closed after the n seconds because of the takeUntil: `.takeUntil(Rx.Observable.timer(seconds * 1000))` unsubscribes the observable when the stream passed as function emits a value.

`switchMap` operator unsubscribes its observable (so stops getting events from it) even if another event comes on the main stream (so in this case another `START_COUNTDOWN` action).

```
---{seconds: 5}------{seconds: 5}----------------------->
action in
vvvvvvvvvvvvvvvv switchMap vvvvvvvvvvvvvvvvvvvvvvvvvvv
--------------{5}-{4}----------{5}--{4}--{3}--{2}--{1}-->
action out
```

If you don't want to stop listening you may need to use `mergeMap` instead.

Imagine to have to modify the epic above to manage many countdowns, identified by an `id`. In this case a START_COUNTDOWN event should not stop the ones already started. You can do it using `mergeMap`.

```
const countDown = action$ => action$
    .ofType(START_COUNTDOWN)
    // mergeMap do not stops the flows already created.
    .mergeMap( ({seconds, id}) => // the id of the new
countdown is in the action
        Rx.Observable.interval(1000)
            // emit an action that updates the coundown for the
specific id
            .map( (value) => updateTime(seconds - value, id))
            .takeUntil(Rx.Observable.timer(seconds * 1000))
    );
```

In this case the streams will look like this:

```
---{sec: 5,id: A}----{sec: 5, id:
B}----------------------------->   action in
vvvvvvvvvvvvvvvvvvvvvvvvv switchMap vvvvvvvvvvvvvvvvvvvvvvvvv
..............A....A....A....A...B...A...B....B....B....B........
id
--------------{5}--{4}--{3}--{2}-{5}-{1}-{4}--{3}--{2}--
{1}------->   value
```

Doing AJAX

Ajax calls in MapStore should all pass by `libs/ajax.js`. This is an `axios` instance that adds the support for using proxies or CORS.

Axios is a library that uses ES6 Promises to do ajax calls. Luckily RxJs allows to use Promises instead of streams in most of the cases. In the other cases, there is a specific operator called `defer` that you can use to wrap your Promise into a stream.

> NOTE: It is perfectly normal to consider the concept of Promise as a special case of a stream, that emit only one value, then closes.

So, every time you have to do an ajax call, you will need to use axios:

Example with `defer`:

```
const axios = require('../libs/ajax');
const fetchDataEpic = (action$, store) => action$
    .ofType(FETCH_DATA)
    .switchMap(
        Rx.Observable.defer(() => axios.get("MY_DATA")) //
defer gets a function
            map(response => dataFetched(response.data))
    );
```

Epic state: muted / unmuted

All the epics attached to plugins or extension will be registered once plugin is loaded.

Each registered epic can be in one of two possible states:

muted: no reaction to the actions that comes in
unmuted: reacting to the listed actions

- Whenever new epic is registered it will be in unmuted state by default.
- Epic will become muted whenever there is no plugin/extension on the page listing that specific epic in plugin definition. In other words, if there are `Extension1` and `Plugin2`, both are adding epic called `testEpic` and both plugin and extension are not added to the current page plugins in pluginsConfig, then epic will become muted.

Muted epics: how to mute internal streams

MapStore will mute all the epics whenever corresponding plugin or extension is not rendered on the page. Though, it might be the case that one of your epics will return internal stream, like in example below:

```
export const dummyEpic = (action$, store) =>
action$.ofType(ACTION)
    .switchMap(() => {
        return Rx.Observable.interval(1000)
            .switchMap(() => {
                console.log('TEST');
                return Rx.Observable.empty();
            });
    });
```

In this case, internal stream should be muted explicitly.

Each epic receives third argument type of object, having property called `pluginRenderStream$` . Combined with `semaphore` it allows to mute internal stream whenever epic is muted:

```
export const dummyEpic = (action$, store, {
pluginRenderStream$ }) => action$.ofType(ACTION)
    .switchMap(() => {
        return Rx.Observable.interval(1000)
            .let(semaphore(pluginRenderStream$.startWith(true)))
            .switchMap(() => {
                console.log('TEST');
                return Rx.Observable.empty();
            });
    });
```

Writing Actions and Reducers

> ┻┳|
> ┳┻| _
> ┻┳| •.•) 💬 *"Hey, Checkout this awesome documentation for actions and reducers!"*
> ┳┻|⊂ﾉ
> ┻┳|

What are actions?

Quoting the redux documentation they are:

> Actions are payloads of information that send data from your application to your store.

They are simply plain JavaScript objects

```
/* trigger the panning action of the map to a center point */
const center = [42.3, 36.5];
export const PAN_TO = 'MAP:PAN_TO';
{
    type: PAN_TO,
    center
}
```

They must have type property, typically a constant with a string value, but any other properties are optional

Why we use them

We need them to trigger changes to the application's store via reducers. To do that we use Action Creators

Action Creators

They are simply function that returns actions objects

```
const defaultValue = [42.3, 36.5];
/*
 * by convention, use an initial name (the action filename)
 * in order to describe better the action type, in this case MAP
 * separated by a colon : and the action constant name
*/
export const PAN_TO = 'MAP:PAN_TO';
export const panTo = (center = defaultValue) => ({
    type: PAN_TO,
    center
});
```

Note: Stick to es6 import/export module system and when possible provide a default value for the parameters

These action creators are used in the connected components or in MapStore2 plugins But actions by themselves are not enough we need Reducers that intercepts those actions and change the state accordingly.

Note: Remember to put all the actions .js files in the web/client/actions folder or in js/actions if you are working with custom plugins

Reducers

Again quoting redux documentation they are:

> Reducers specify how the application's state changes in response to actions sent to the store.

Reducers are pure functions that take the previous state and an action and return a new state

> (previousState, action) => newState

let's see an example:

```
// @mapstore is an alias for dir_name/web/client (see
webpack.config.js)
import {PAN_TO} from '@mapstore/actions/map';

export default function map(state, action) {
    switch (action.type) {
        case PAN_TO: {
            return {
                ...state,
                center: action.center
            };
        }
        default: return state;
    }
}
```

As you can see we are changing the center of the map that triggers the panning action of the mapping library

And that's it we have wrote an action and a reducers that make the map panning around.

Note: Remember to put all the reducers .js files in the web/client/reducers folder or in js/reducers if you are working with custom plugins

Advanced usage and tips

Sometimes you need to change a value of an item which is stored inside an array or in a nested object.

Let's imagine we have this object in the store:

```
layer: {
    features: [object_1, object_2, ...object_n]
}
```

And we have created an action that holds the id of the object to change and some properties

```
```

```
export const UPDATE_LAYER_FEATURE = 'LAYER:UPDATE_LAYER_FEATURE'
export const updateFeature = (id, props = {}) => ({type:
UPDATE_LAYER_FEATURE, id, props})
```

Then in the reducer we can have different implementations. Here we show the one using **arrayUpdate** from @mapstore/utils/ImmutableUtils for updating objects in array

```
import {UPDATE_LAYER_FEATURE} from '@mapstore/actions/layer';
import {find} from 'lodash';
const defaultState = {
    features: [{ id: 1, type: "Feature", geometry: { type :
"Point", coordinates: [1, 2]}}]
};

export default function layer(state = defaultState, action) {
    switch (action.type) {
        case UPDATE_LAYER_FEATURE: {
            // let's assume that action.props = {newProp:
"newValue"}
            const feature = find(state.features, {id:
action.id});
            // merging the old feature object with the new prop
while replacing the existing element in the array
            const newFeature = {...feature, ...action.props};
            return arrayUpdate("features", newFeature, {id:
action.id}, state);
            // after this you expect to find the new properties
in the feature specified by the id
        }
        default: return state;
    }
}
```

Testing

Tests in mapstore are stored in `__tests__` folder at the same level where actions/reducer are. The file name is the same of the action/reducer with a '-test' suffix

```
```

```
actions/map.js
actions/__tests__/map-test.js
or
reducers/map.js
reducers/__tests__/map-test.js
```

We use expect as testing library, therefore we suggest to have a look there.

How to test an action

Typically you want to test the type and the params return from the action creator

let's test the mapTo action:

```
// copyright section
import expect from 'expect';
import {panTo, PAN_TO} from '@mapstore/actions/map';
describe('Test correctness of the map actions', () => {
    it('testing panTo', () => {
        const center = [2, 3];
        const returnValue = panTo(center);
        expect(returnValue.type).toEqual(PAN_TO);
        expect(returnValue.center).toEqual(center);
    });
});
```

In order to speed up the unit test runner, you can:

- change the path in tests.webpack.js (custom/standard project) or build\tests.webpack.js (framework) to point to the folder parent of **tests** for example `'/js/actions'` for custom/standard project or `'../web/client/actions'` for framework
- then run this command: `npm run continuoustest`

This allows to run only the tests contained to the specified path. **Note:** When all tests are successfully passing remember to restore it to its original value.

How to test a reducer

Here things can become more complicated depending on your reducer but in general you want to test all cases

```
// copyright section
import expect from 'expect';
import {panTo} from '@mapstore/actions/map';
import map from '@js/reducers/map'; // the one created before
not the one present in @mapstore/reducers
describe('Test correctness of the map reducers', () => {
    it('testing PAN_TO', () => {
        const center = [2, 3];
        const state = map({}, panTo(center));

        // here you have to check that state has changed
accordingly
        expect(state.center).toEqual(center);
    });
});
```

Here for speedup testing you can again modify the tests.webpack.js (custom/ standard project) or build\tests.webpack.js (framework) in order to point to the reducers folder and then running `npm run continuoustest`

Actions and epics

Actions are not only used by redux to update the store (through the reducers), but also for triggering side effects workflows managed by epics

For more details see Writing epics

Configuring MapStore

MapStore (and every application developed with MapStore) allows customization through configuration. To understand how to configure MapStore you have to know that the back-end and the front-end of MapStore have two different configuration systems.

This separation allows to:

- Make mapstore configuration system live also as a front-end only framework
- Keep the power of customization provided by spring on the back-end

Back-end Configuration Files

They are `.properties` files or `.xml` files, and they allow to configure the various parts of the back-end. They are located in `java/web/src/main/resources` and they will be copied in `MapStore.war` under the directory `/WEB-INF/classes`.

- `proxy.properties` : configuration for the internal proxy (for cross-origin requests). More information here.
- `geostore-datasource-ovr.properties` : provides settings for the database.
- `log4j.properties` : configuration for back-end logging
- `sample-categories.xml` : initial set of categories for back-end resources (MAP, DASHBOARD, GEOSTORY...)
- `mapstore.properties` : allow specific overrides to front-end files, See externalization system for more details

Except for `mapstore.properties` and `ldap.properties`, all these files are simply overrides of original configuration files coming from the included sub-applications part of the back-end. In `WEB-INF/classes` you will find also some other useful files coming from the original application:

Back-end security configuration files

Back-end security can be configured to use different authentication strategies. Maven profiles can be used to switch between these different strategies.

Depending on the chosen profile a different file will be copied from the `product/config` folder to override `WEB-INF/classes/geostore-spring-security.xml` in the final package. In particular:

- **default**: `db\geostore-spring-security-db.xml` (geostore database)
- **ldap**: `ldap\geostore-spring-security-ldap.xml` (LDAP source)

Specific configuration files are available to configure connection details for the chosen profile.

For example, if using LDAP, look at LDAP integration.

Front-end Configurations Files

They are JSON files that will be loaded via HTTP from the client, keeping most of the framework working also in an html-only context (when used with different back-ends or no-backend). These JSON files are located in `web/client/configs` directory and they will be copied in the `configs` of the war file.

Several configuration files (at development and / or run time) are available to configure all the different aspects of an application.

- `localConfig.json` : Dedicated to the application configuration. Defines all general settings of the front-end part, with all the plugins for all the pages. See Application Configuration for more information.
- `new.json` Can be customized to set-up the initial new map, setting the backgrounds, initial position .. See Maps configuration for more information.
- `pluginsConfig.json` : Allows to configure the context editor plugins list. See Context Editor Configuration for more information.

Externalize Configurations

Typically configuration customization should stay outside the effective application installation directory to simplify future updates. Updates in fact are usually replacement of the old application file package with the newer one. Changes applied directly inside the application package may be so removed on every update. For this reason MapStore provides a externalization system for both the configuration systems. See Externalize Configuration section to learn how to do this.

Application configuration

The application will load by default it will load the `localConfig.json` which is now stored in the configs\ folder

You can load a custom configuration by passing the `localConfig` argument in query string:

```
localhost:8081/?localConfig=myConfig#/viewer/openlayers/0
```

The **localConfig** file contains the main information about URLs to load and plugins to load in the various modes.

This is the main structure:

```
{
  // URL of geoStore
  "geoStoreUrl": "rest/geostore/",
  // printURL the url of the print service, if any
  "printUrl": "/geoserver-test/pdf/info.json",
  // a string or an object for the proxy URL.
  "proxyUrl": {
    // if it is an object, the url entry holds the url to the
proxy
    "url": "/MapStore2/proxy/?url=",
    // useCORS array contains a list of services that support
CORS and so do not need a proxy
    "useCORS": ["http://nominatim.openstreetmap.org", "https://
nominatim.openstreetmap.org"]
  },
  // JSON file where uploaded extensions are configured
  "extensionsRegistry": "extensions.json",
  // URL of the folder from where extensions bundles and other
assets are loaded
  "extensionsFolder": "",
  // API keys for bing and mapquest services
  "bingApiKey",
  // force dates to be in this specified format. use moment js
format pattern
  "forceDateFormat": "YYYY-MM-DD",
```

```
  // force time to be in this specified format. use moment js
format pattern
  "forceTimeFormat": "hh:mm A",
  "mapquestApiKey",
  // list of actions types that are available to be launched
dynamically from query param (#3817)
  "initialActionsWhiteList": ["ZOOM_TO_EXTENT",
"ADD_LAYER", ...],
  // path to the translation files directory (if different from
default)
  "translationsPath",
  // if true, every ajax and mapping request will be
authenticated with the configurations if match a rule (default:
true)
  "useAuthenticationRules": true
  // the authentication rules to match
  "authenticationRules": [
  { // every rule has a `urlPattern` regex to match
    "urlPattern": ".*geostore.*",
    // and a authentication `method` to use (basic, authkey,
browserWithCredentials)
    "method": "basic"
  }, {
    "urlPattern": "\\/geoserver.*",
    "method": "authkey"
  }],
  // flag for postponing mapstore 2 load time after theme
  "loadAfterTheme": false,
  // if defined, WMS layer styles localization will be added
  "localizedLayerStyles": {
      // name of the ENV parameter variable that is needed for
localization proposes
      "name": "mapstore_language"
  },
  // flag for abandon map edit confirm popup, by default is
enabled
  "unsavedMapChangesDialog": false,
  // optional flag to set default coordinate format (decimal,
aeronautical)
  "defaultCoordinateFormat": "aeronautical",
  // optionals misc settings
  "miscSettings": {
      // Use POST requests for each WMS length URL highter than
this value.
      "maxURLLength": 5000
  },
  // optional state initializer (it will override the one
defined in appConfig.js)
  "initialState": {
```

```
      // default initial state for every mode (will override
initialState imposed by plugins reducers)
      "defaultState": {
          ...
          // if you want to customize the supported locales put
here the languages you want and follow instruction linked below
          "locales": {
            "supportedLocales": {
              "it": {
                "code": "it-IT",
                "description": "Italiano"
              },
              "en": {
                "code": "en-US",
                "description": "English"
              }
          }
        }
      },
      // mobile override (defined properties will override
default in mobile mode)
      "mobile": {
          ...
      }
  },
  // allows to apply map options configuration to all the Map
plugins instances defined in the plugins configuration.
  // The mapOptions in the plugin configuration have priority
so they will overrides this global config
  "defaultMapOptions": {
    "openlayers": { ... },
    "leaflet": { ... },
    "cesium": { ... }
  },
  "plugins": {
      // plugins to load for the mobile mode
      "mobile": [...]
      // plugins to load for the desktop mode
      "desktop": [...]
      // plugins to load for the embedded mode
      "embedded": [...]
      // plugins to load for the myMode mode
      "myMode": [...]
  }
}
```

If you are building your own app, you can choose to create your custom modes or force one of them by passing the `mode` parameter in the query string.

For adding a new locale or configuring currently supported locales, go check this out.

For configuring plugins, see the Configuring Plugins Section and the plugin reference page

Explanation of some config properties

- **loadAfterTheme** is a flag that allows to load mapstore.js after the theme which can be versioned or not(default.css). default is false
- **initialState** is an object that will initialize the state with some default values and this WILL OVERRIDE the initialState imposed by plugins & reducers.
- **projectionDefs** is an array of objects that contain definitions for Coordinate Reference Systems

initialState configuration

It can contain:

1. a defaultState valid for every mode
2. a piece of state for each mode (mobile, desktop, embedded)

Catalog Tool configuration

Inside defaultState you can set default catalog services adding the following key

```
"catalog": {
  "default": {
    "newService": {
      "url": "",
      "type": "wms",
      "title": "",
      "isNew": true,
      "editable": true,
      "autoload": false
```

```
    },
    "selectedService": "Demo CSW Service",
    "services": {
      "Demo CSW Service": {
        "url": "https://demo.geo-solutions.it/geoserver/csw",
        "type": "csw",
        "title": "A title for Demo CSW Service",
        "autoload": true
      },
      "Demo WMS Service": {
        "url": "https://demo.geo-solutions.it/geoserver/wms",
        "layerOptions": {
          "tileSize": 512
          },
          "format": "image/png8"
        "type": "wms",
        "title": "A title for Demo WMS Service",
        "autoload": false
      },
      "Demo WMTS Service": {
        "url": "https://demo.geo-solutions.it/geoserver/gwc/
service/wmts",
        "type": "wmts",
        "title": "A title for Demo WMTS Service",
        "autoload": false
      }
    }
  }
}
```

Set `selectedService` value to one of the ID of the services object ("Demo CSW Service" for example).
This will become the default service opened and used in the catalog panel.
For each service set the key of the service as the ID.

```
"ID_CATALOG_SERVICE": {
  "url": "the url pointing to the catalog web service",
  "type": "the type of webservice used. (this need to be
consistent with the web service pointed by the url)",
  "title": "the label used for recognizing the catalog service",
  "autoload": "if true, when selected or when catalog panel is
opened it will trigger an automatic search of the layers. if
false, search must be manually performed."
  "readOnly": "if true, makes the service not editable from
catalog plugin"
```

```
  "titleMsgId": "optional, string used to localize the title of
the service, the string must be present in translations",
  "format": "image/png8" // the image format to use by default
for layers coming from this catalog (or tiles).
  "layerOptions": { // optional
      "format": "image/png8", // image format needs to be
configured also inside layerOptions
      "tileSize": 512 // determine the default tile size for
the catalog, valid for WMS and CSW catalogs
  },
  "filter": { // applicable only for CSW service
      "staticFilter": "filter is always applied, even when
search text is NOT PRESENT",
      "dynamicFilter": "filter is used when search text is
PRESENT and is applied in `AND` with staticFilter. The template
is used with ${searchText} placeholder to append search string"
  }
}
```

CSW service

`filter` - For both static and dynamic filter, input only xml element contained within (i.e. Do not enclose the filter value in)

Example:

```
{
    "filter": { // Default filter values
        "staticFilter":
"<ogc:Or><ogc:PropertyIsEqualTo><ogc:PropertyName>dc:type</
ogc:PropertyName><ogc:Literal>dataset</ogc:Literal></
ogc:PropertyIsEqualTo><ogc:PropertyIsEqualTo><ogc:PropertyName>dc:
ogc:PropertyName><ogc:Literal>http://purl.org/dc/dcmitype/
Dataset</ogc:Literal></ogc:PropertyIsEqualTo></ogc:Or>",
        "dynamicFilter": "<ogc:PropertyIsLike wildCard='%'
singleChar='_' escapeChar='\\'><ogc:PropertyName>csw:AnyText</
ogc:PropertyName><ogc:Literal>%${searchText}%</ogc:Literal></
ogc:PropertyIsLike>"
    }
}
```

Be careful to use unique IDs

Future implementations will try to detect the type from the url.
newService is used internally as the starting object for an empty service.

Annotations Editor configuration

Annotations editor can be configured by setting it's defaultState. It looks like this:

```
"defaultState": {
 ...
  "annotations": {
    "config": {
    "geometryEditorOptions": {
        "measureOptions": {
          "displayUom": "nm"
        }
      },
      "multiGeometry": true,
      "validationErrors": {},
      "defaultPointType": "symbol"
    },
    "format": "aeronautical",
    "defaultTextAnnotation": "New"
  },
```

- **format** - decimal or aeronautical degree for coordinates
- **defaultTextAnnotation** - default text value for text annotations
- **config.geometryEditorOptions** - properties to be passed to CoordinatesEditor of GeometryEditor. For more information refer to the documentation of CoordinatesEditor component
- **config.multiGeometry** - if set to true allows to add more then one geometry to annotations
- **config.defaultPointType** - default point type of marker geometry type. Can be 'marker' or 'symbol'

projectionDefs configuration

Custom CRS can be configured here, at root level of localConfig.json file. For example:

```
"projectionDefs": [{
  "code": "EPSG:3003",
  "def": "+proj=tmerc +lat_0=0 +lon_0=9 +k=0.9996 +x_0=1500000
+y_0=0
+ellps=intl+towgs84=-104.1,-49.1,-9.9,0.971,-2.917,0.714,-11.68
+units=m +no_defs",
  "extent": [1241482.0019, 973563.1609, 1830078.9331,
5215189.0853],
  "worldExtent": [6.6500, 8.8000, 12.0000, 47.0500]
}]
```

Explanation of these properties:

- **code** - a code string that will identify the added projection
- **def** - projection definition in PROJ.4 format
- **extent** - projected bounds of the projection
- **worldExtent** - bounds of the projection in WGS84

These parameters for a projection of interest can be found on epsg.io

CRS Selector configuration

CRS Selector is a plugin, that is configured in the plugins section. It should look like this:

```
"plugins": {
  ...
  "desktop": [
    ...}, {
      "name": "CRSSelector",
      "cfg": {
        "additionalCRS": {
          "EPSG:3003": {
            label: "Monte Mario"
```

```
            }
          },
          "filterAllowedCRS": [
            "EPSG:4326",
            "EPSG:3857"
          ],
          "allowedRoles": [
            "ADMIN"
          ]
        }
      }, {
    ]
  }
```

Configuration parameters are to be placed in the "cfg" object. These parameters are:

- **additionalCRS** - object, that contains additional Coordinate Reference Systems. This configuration parameter lets you specify which projections, defined in **projectionDefs**, should be displayed in the CRS Selector, alongside default projections. Every additional CRS is a property of **additionalCRS** object. The name of that property is a code of a corresponding projection definition in **projectionDefs**. The value of that property is an object with the following properties:
- **label** - a string, that will be displayed in the CRS Selector as a name of the projection
- **filterAllowedCRS** - which default projections are to be available in the selector. Default projections are:
- EPSG:3857
- EPSG:4326
- **allowedRoles** - CRS Selector will be accessible only to these roles. By default, CRS Selector will be available for any logged in user.

Search plugin configuration

The search plugin provides several configurations to customize the services behind the search bar in the map: - Allow to configure more many services to use

in parallel, in the `services` array. - Natively supports nominatim and WFS protcols - Allows to register **your own** custom services to develop and use in your custom project - Allows to configure services in cascade, typically when you have a hierarchical data structures (e.g. search for municipality, then for street name, than for house number, or search state,then region, then specific feature, and so on...)

Following you can find some examples of the various configurations. For more details about the properties, please check to plugin API documentation: https://mapstore.geosolutionsgroup.com/mapstore/docs/api/plugins#plugins.Search

Nominatim configuration:

```
{
   "type": "nominatim",
   "searchTextTemplate": "${properties.display_name}", // text
to use as searchText when an item is selected. Gets the result
properties.
   "options": {
     "polygon_geojson": 1,
     "limit": 3
}
```

WFS configuration:

```
"plugins": {
  ...
  "desktop": [
    ...}, {
      "name": "Search",
      "cfg": {
        "showCoordinatesSearchOption": false,
        "maxResults": 20,
        "searchOptions": {
          "services": [{
            "type": "wfs",
            "priority": 3,
            "displayName": "${properties.propToDisplay}",
            "subTitle": " (a subtitle for the results coming
from this service [ can contain expressions like $
{properties.propForSubtitle}])",
            "options": {
```

```
              "url": "/geoserver/wfs",
              "typeName": "workspace:layer",
              "queriableAttributes": ["attribute_to_query"],
              "sortBy": "id",
              "srsName": "EPSG:4326",
              "maxFeatures": 20,
              "blacklist": ["... an array of strings to exclude
from the final search filter "]
            }
        ]
        }
      }
    }, {
  ]
}
```

WFS configuration with nested services:

```
"plugins": {
  ...
  "desktop": [
    ...}, {
      "name": "Search",
      "cfg": {
        "showCoordinatesSearchOption": false,
        "maxResults": 20,
        "searchOptions": {
          "services": [{
            "type": "wfs",
            "priority": 3,
            "displayName": "${properties.propToDisplay}",
            "subTitle": " (a subtitle for the results coming
from this service [ can contain expressions like $
{properties.propForSubtitle}])",
            "options": {
              "url": "/geoserver/wfs",
              "typeName": "workspace:layer",
              "queriableAttributes": ["attribute_to_query"],
              "sortBy": "id",
              "srsName": "EPSG:4326",
              "maxFeatures": 20,
              "blacklist": ["... an array of strings to exclude
from the final search filter "]
            },

            "nestedPlaceholder": "the placeholder will be
```

```
displayed in the input text, after you have performed the first
search",
            "then": [{
              "type": "wfs",
              "priority": 1,
              "displayName": "${properties.propToDisplay} $
{properties.propToDisplay}",
              "subTitle": " (a subtitle for the results coming
from this service [ can contain expressions like $
{properties.propForSubtitle}])",
              "searchTextTemplate": "$
{properties.propToDisplay}",
              "options": {
                "staticFilter": " AND SOMEPROP = '$
{properties.OLDPROP}'", // will be appended to the original
filter, it gets the properties of the current selected item (of
the parent service)
                "url": "/geoserver/wfs",
                "typeName": "workspace:layer",
                "queriableAttributes": ["attribute_to_query"],
                "srsName": "EPSG:4326",
                  "maxFeatures": 10
                }
            }]
  }

        ]
        }
      }
    }, {
  ]
}
```

Custom services configuration:

```
{
  "type": "custom Service Name",
  "searchTextTemplate": "${properties.propToDisplay}",
  "displayName": "${properties.propToDisplay}",
  "subTitle": " (a subtitle for the results coming from this
service [ can contain expressions like $
{properties.propForSubtitle}])",
  "options": {
    "pathname": "/path/to/service",
    "idVia": "${properties.code}"
  },
```

```
    "priority": 2,
    "geomService" : {
      "type": "wfs",
      "options": {
        "url": "/geoserver/wfs",
        "typeName":  "workspace:layer",
        "srsName": "EPSG:4326",
        "staticFilter": "ID = ${properties.code}"
      }
    }
  }
```

Configuring plugins

To configure the plugins used by your application, a dedicated section is available in the **localConfig.json** configuration file:

```
"plugins": {
  ...
}
```

Inside the **plugins** section, several modes can be configured (e.g. desktop or mobile), each one with its own list of plugins:

```
"plugins": {
  "mobile": [...],
  "desktop": [...]
}
```

Each plugin can be simply listed (and the default configuration is used):

```
"plugins": {
  ...
  "desktop": ["Map", "MousePosition", "Toolbar", "TOC"]
}
```

or fully configured:

```
"plugins": {
  ...
  "desktop": [{
    "name": "Map",
       ...
    }
  },
  ...
  ]
}
```

Dynamic configuration

Configuration properties of plugins can use expressions, so that they are dynamically bound to the application state.

An expression is anything between curly brackets ({...}) that is a javascript expression, where the **monitored state** of the application is available as a set of variables.

To define the monitored state, you have to add a **monitorState** property in **localConfig.json**.

```
{
  ...
  "monitorState": [{"name": "mapType", "path":
"mapType.mapType"}]
  ...
}
```

Where: * **name** is the name of the variable that can be used in expressions * **path** is a javascript object path to the state fragment to be monitored (e.g. map.present.zoom)

When you have a monitored state, you can use it in configuration properties this way: Be sure to write a valid javascript expression.

```
```

```
"cfg": {
  ...
  "myProp": "{state('mapType') === 'openlayers' ? 1 : 2}"
  ...
}
```

Expressions are supported in **cfg** properties and in **hideFrom** and **showIn** sections.

In addition to monitored state also the **page request parameters** are available as variables to be used in expressions.

Look at the plugin reference page for a list of available configuration properties.

Map Configuration

By default MapStore is able to open maps with this path in the URL:

```
http://localhost:8081/#viewer/<maptype>/<mapId>
```

Where:

- `maptype` can be `leaflet` `openlayers` or `cesium`.
- `mapId` can be a number or a string.
- A **number** represents standard maps, stored on the database.
- A **string** instead represents a static json file in the root of the application.

The first case can be used to load a map from the maps database, using its id.

There is a special mapId, 0 (zero), that is used to load a basic OSM map for demo purposes.

```
http://localhost:8081/#viewer/openlayers/0
```

The configuration of this map is stored in the static `config.json` file in the root of the project.

The second case can be used to define standard map contexts.

This is used for the **new map**. If you're logged in and allowed to create maps, when you try to create a new map you will see the the application will bring you to the URL:

```
http://localhost:8081/#viewer/openlayers/new
```

This page uses the `new.json` file as a template configuration to start creating a new map. You can find this file in `web/client/configs` directory for standard

MapStore or in `configs/` folder for a custom projects. You can edit `new.json` to customize this initial template. It typically contains the map backgrounds you want to use for all the new maps (identified by the special property `"group": "background"`).

If you have enabled the datadir, then you can externalize the new.json or config.json files. (see here for more details)

`new.json` and `config.json` are special cases, but you can configure your own static map context creating these json files in the root of the project, for instance `mycontext.json` and accessing them at the URL:

```
http://localhost:8081/#viewer/openlayers/mycontext
```

important note: `new.json` and `config.json` are special files and don't require the version. For other map context, you **must** specify the version of the map file type in the root of the json file:

```
{
    "version": 2,
    // ...
}
```

These static map contexts are accessible by anyone. If you want to customize standard maps (that are listed in home page and where you can define) manually, you will have to edit the maps using the GeoStore REST API.

Map options

The following options define the map options (projection, position, layers):

- `projection: {string}` expressed in EPSG values
- `units: {string}` uom of the coordinates
- `center: [object]` center of the map with starting point in the bottom-left corner

- `zoom: {number}` level of zoom
- `resolutions: {number[]}` resolutions for each level of zoom
- `scales: {number[]}` scales used to compute the map resolutions
- `maxExtent: {number[]}` max bbox of the map expressed [minx, miny, maxx, maxy]
- `layers: {object[]}` list of layers to be loaded on the map
- `groups {object[]}` : contains information about the layer groups

i.e.


```
{
    "version": 2,
    "projection": "EPSG:900913",
    "units": "m",
    "center": {"x": 1000000.000000, "y": 5528000.000000, "crs":
"EPSG:900913"},
    "zoom": 15,
    "mapOptions": {
      "view": {
        "scales": [175000, 125000, 100000, 75000, 50000, 25000,
10000, 5000, 2500],
        "resolutions": [
          84666.66666666688,
          42333.33333333344,
          21166.66666666672,
          10583.33333333336,
          5291.66666666668,
          2645.83333333334,
          1322.91666666667,
          661.458333333335000,
          529.166666666668000,
          396.875000000001000,
          264.583333333334000,
          132.291666666667000,
          66.145833333333500,
          39.687500000000100,
          26.458333333333400,
          13.229166666666700,
          6.614583333333350,
          3.968750000000010,
          2.645833333333340,
          1.322916666666670,
          0.661458333333335
```

```
            ]
          }
      },
      "maxExtent": [
          -20037508.34, -20037508.34,
          20037508.34, 20037508.34
      ],
      "layers": [{...},{...}]
  }
```

Note

The option to configure a list of scale denominators allow to have them in human friendly format, and calculate the map resolutions from scales.

Warning

If the scales and resolutions property are declared, in the same json object, the scales have priority. In the array, the values have be in descending order.

Warning

Actually the custom resolution values are valid for one single CRS. It's therefore suggested to avoid to add this parameter when multiple CRSs in the same map configuration are needed.

Additional map configuration options

Map configuration also contains the following additional options:

- `catalogServices` object describing services configuration for Catalog
- `widgetsConfig` configuration of map widgets
- `mapInfoConfiguration` map info configuration options
- `dimensionData` contains map time information
  - `currentTime` currently selected time; the beginning of a time range if offsetTime is set
  - `offsetTime` the end of a time range

- `timelineData` timeline options
    - `selectedLayer` selected layer id; if not present time cursor will be unlocked

Layers options

Every layer has it's own properties. Anyway there are some options valid for every layer:

- `title` : `{object|string}` the title of the layer, can be an object to support i18n.
- `type` : `{string}` the type of the layer. Can be `wms` , `wmts` , `osm` ...
- `name` : `{string}` the name is used as general reference to the layer, or as title, if the title is not specified. Anyway, it's usage depends on the specific layer type.
- `group` : `{string}` : the group of the layer (in the TOC). Nested groups can be indicated using `/` . i.e. `Group/SubGroup` . A special group, `background` , is used to identify background layers. These layers will not be available in the TOC, but only in the background switcher, and only one layer of this group can be visible.
- `thumbURL` : `{string}` : the URL of the thumbnail for the layer, used in the background switcher (if the layer is a background layer)
- `visibility` : `{boolean}` : indicates if the layer is visible or not
- `queriable` : `{boolean}` : Indicates if the layer is queriable (e.g. getFeatureInfo). If not present the default is true for every layer that have some implementation available (WMS, WMTS). Usually used to set it explicitly to false, where the query service is not available.
- `hideLoading` : {boolean}. If true, loading events will be ignored (useful to hide loading with some layers that have problems or trigger errors loading some tiles or if they do not have any kind of loading.).
- `minResolution` : `{number}` : layer is visible if zoom resolution is greater or equal than this value (inclusive)

- `maxResolution` : `{number}` : layer is visible if zoom resolution is less than this value (exclusive)
- `disableResolutionLimits` : `{boolean}` : this property disables the effect of minResolution and maxResolution if set to true

i.e.

```
{
    "title": "Open Street Map",
    "name": "mapnik",
    "group": "background",
    "visibility": false,
    "hidden": true
}
```

Localized titles: In these configuration files you can localize titles using an object instead of a string in the `title` entry. In this case the `title` object has this shape:

```
title: {
      'default': 'Meteorite Landings from NASA Open Data
Portal', // default title, used in case the localized entry is
not present
      'it-IT': 'Atterraggi meteoriti', // one string for each
IETF language tag you want to support.
      'en-US': 'Meteorite Landings',
      'fr-FR': 'Débarquements de météorites'
    },
```

Layer types

- `wms` : WMS - Web Mapping Service layers
- `osm` : OpenStreetMap layers format
- `tileprovider` : Some other mixed specific tile providers
- `wmts` : WMTS: Web Map Tile Service layers
- `bing` : Bing Maps layers

- `google` : Google Maps layers
- `mapquest` : MapQuest layers
- `graticule` : Vector layer that shows a coordinates grid over the map, with optional labels
- `empty` : special type for empty background
- `3dtiles` : 3d tiles layers
- `terrain` : layers that define the elevation profile of the terrain

WMS

i.e.

```
{
    "type": "wms",
    "url": "http..." // URL of the WMS Service
    "name": "TEST:TEST", // The name of the layer
    "format": "image/png8" // format
    "title": "Open Street Map",
    "name": "mapnik",
    "group": "background",
    "visibility": false,
    "params": {}, // can be used to add parameters to the
request, or override the default ones
    "credits": { // optional
        "imageUrl": "somePic.png", // URL for the image to put
in attribution
        "link": "http://someURL.org", // URL where attribution
have to link to
        "title": "text to render" // title to show (as image
title or as text)
    }
}
```

You can also configure a WMS layer also as background, like this:

```
    {
        "format": "image/jpeg",
        "name": "workspace:layername",
        "params": {},
        "singleTile": false,
```

```
        "title": "My WMS Background",
        "type": "wms",
        "group": "background",
        "thumbURL": "http://some.wms.service/geoserver/ows?
SERVICE=WMS&REQUEST=GetMap&VERSION=1.3.0&LAYERS=rv%3Arv1&STYLES=&F
        "url": "http://some.wms.service/geoserver/ows",
        "visibility": false
    },
```

MULTIPLE URLS

This feature is not yet fully supported by all the plugins, but OpenLayers supports it so if you put an array of urls instead of a single string in the layer url. Some other feature will break, for example the layer properties will stop working, so it is safe to use only on background layers.

```
{
  "type": "wms",
  "url": [
    "https://a.maps.geo-solutions.it/geoserver/wms",
    "https://b.maps.geo-solutions.it/geoserver/wms",
    "https://c.maps.geo-solutions.it/geoserver/wms",
    "https://d.maps.geo-solutions.it/geoserver/wms",
    "https://e.maps.geo-solutions.it/geoserver/wms",
    "https://f.maps.geo-solutions.it/geoserver/wms"
  ],
  "visibility": true,
  "opacity": 1,
  "title": "OSM",
  "name": "osm:osm",
  "group": "Meteo",
  "format": "image/png8",
  "bbox": {
    "bounds": {"minx": -180, "miny": -90, "maxx": 180, "maxy":
90},
    "crs": "EPSG:4326"
  }
},
```

SPECIAL CASE - THE ELEVATION LAYER

> **Note**
>
> This type of layer configuration is still needed to show the elevation data inside the MousePosition plugin. The `terrain` layer section shows a more versatile way of handling elevation but it will work only as visualization in the 3D map viewer.

WMS layers can be configured to be used as a source for elevation related functions.

This requires:

- a GeoServer WMS service with the DDS/BIL plugin
- A WMS layer configured with **BIL 16 bit** output in **big endian mode** and **-9999 nodata value**
- a static layer in the Map plugin configuration (use the additionalLayers configuration option):

in `localConfig.json`

```
{
    "name": "Map",
    "cfg": {
        "additionalLayers": [{
            "url": "http...",
            "format": "application/bil16",
            "type": "wms",
            ...
            "name": "elevation",
            "littleendian": false,
            "visibility": true,
            "useForElevation": true
        }]
    }
}
```

The layer will be used for:

- showing elevation in the MousePosition plugin (requires showElevation: true in the plugin configuration)
- as a TerrainProvider if the maptype is Cesium

in `localConfig.json`

```
{
    "name": "MousePosition",
    "cfg": {
        "showElevation": true,
        ...
    }
}
```

WMTS

WMTS Layer require a source object in the `sources` object of the map configuration where to retrieve the `tileMatrixSet`. The source is identified by the `capabilitiesURL`. (if `capabilitiesURL` is not present it will use the `url`, in case of multiple URLs, the first one.).

A WMTS layer can have a `requestEncoding` that is RESTful or KVP. In case of RESTful the URL is a template where to place the request parameters (see the example below), while in the KVP the request parameters are in the query string. See the WMTS standard for more details.

e.g. (RESTful):

```
{
  "version": 2,
  // ...
  "map": {
    // ...
    "layers": [
        // WMTS layer sample
        {
        "id": "bmapoberflaeche__11",
        "name": "layer_name",
        // ...
```

```
        "type": "wmts",
        "url": [ // MULTIPLE URLS are allowed
            "https://maps1.sampleServer/{Style}/{TileMatrixSet}/
{TileMatrix}/{TileRow}/{TileCol}.jpeg",
            "https://maps2.sampleServer/{Style}/{TileMatrixSet}/
{TileMatrix}/{TileRow}/{TileCol}.jpeg",
            "https://maps3.sampleServer/{Style}/{TileMatrixSet}/
{TileMatrix}/{TileRow}/{TileCol}.jpeg",
            "https://maps4.sampleServer/{Style}/{TileMatrixSet}/
{TileMatrix}/{TileRow}/{TileCol}.jpeg",
            "https://maps.sampleServer/{Style}/{TileMatrixSet}/
{TileMatrix}/{TileRow}/{TileCol}.jpeg"
        ],
        "allowedSRS": {
          "EPSG:3857": true
        },
        "matrixIds": [
          "google3857",
          "EPSG:3857"
        ],
        "tileMatrixSet": true,
        // KVP (By default) or RESTful
        "requestEncoding": "RESTful",
        // identifier for the source
        "capabilitiesURL": "https://sampleServer.org/wmts/1.0.0/
WMTSCapabilities.xml",
      }
    ],
    "sources": {
      // source of the layer above
      "https://sampleServer.org/wmts/1.0.0/
WMTSCapabilities.xml": {
        "tileMatrixSet": {
          "google3857": {
            "ows:Identifier": "google3857",
            "ows:BoundingBox": {
              "$": {
                "crs": "urn:ogc:def:crs:EPSG:6.18.3:3857"
              },
              "ows:LowerCorner": "977650 5838030",
              "ows:UpperCorner": "1913530 6281290"
            },
            "ows:SupportedCRS": "urn:ogc:def:crs:EPSG:
6.18.3:3857",
            "WellKnownScaleSet": "urn:ogc:def:wkss:OGC:
1.0:GoogleMapsCompatible",
            "TileMatrix": [
              {
                "ows:Identifier": "0",
```

```
                "ScaleDenominator": "559082264.029",
                "TopLeftCorner": "-20037508.3428 20037508.3428",
                "TileWidth": "256",
                "TileHeight": "256",
                "MatrixWidth": "1",
                "MatrixHeight": "1"
              },
              {
                "ows:Identifier": "1",
                "ScaleDenominator": "279541132.015",
                "TopLeftCorner": "-20037508.3428 20037508.3428",
                "TileWidth": "256",
                "TileHeight": "256",
                "MatrixWidth": "2",
                "MatrixHeight": "2"
              },
              // ...more levels
            ]
          }
        }
      }
    }
  }
```

e.g. (KVP)

```
{
  "version": 2,
  "map": {
    // ...
    "projection": "EPSG:900913",
    "layers": [
      // ...
      {
        // requestEncoding is KVP by default
        "id": "EMSA:S52 Standard__6",
        "name": "EMSA:S52 Standard",
        "description": "S52 Standard",
        "title": "S52 Standard",
        "type": "wmts",
        // if the capabilitiesURL is not present, the `url`
will be used to identify the source.
        // (for retro-compatibility with existing layers)
        "url": "http://some.domain/geoserver/gwc/service/wmts",
        "bbox": {
```

```
            "crs": "EPSG:4326",
            "bounds": {
              "minx": "-180.0",
              "miny": "-79.99999999999945",
              "maxx": "180.0",
              "maxy": "83.99999999999999"
            }
          },
          // list of allowed SRS
          "allowedSRS": {
            "EPSG:4326": true,
            "EPSG:3857": true,
            "EPSG:900913": true
          },
          // list of the available matrixes for the layer
          "matrixIds": [
            "EPSG:3395",
            "EPSG:32761",
            "EPSG:3857",
            "EPSG:4326",
            "EPSG:900913",
            "EPSG:32661"
          ],
          "tileMatrixSet": true
        }
      ],
      // ...
      "sources": {
        "http://some.domain/geoserver/gwc/service/wmts": {
          "tileMatrixSet": {
            "EPSG:32761": {/*...*/},
            "EPSG:3857": {/*...*/},
            "EPSG:4326": {/*...*/},
            "EPSG:32661": {/*...*/},
            "EPSG:3395": {/*...*/},
            "EPSG:900913": {
              "ows:Identifier": "EPSG:900913",
              // the supported CRS
              "ows:SupportedCRS": "urn:ogc:def:crs:EPSG::900913",
              "TileMatrix": [
                {
                  "ows:Identifier": "EPSG:900913:0",
                  "ScaleDenominator": "5.590822639508929E8",
                  "TopLeftCorner": "-2.003750834E7 2.0037508E7",
                  "TileWidth": "256",
                  "TileHeight": "256",
                  "MatrixWidth": "1",
                  "MatrixHeight": "1",
```

```
                "ranges": {
                  "cols": {
                    "min": "0",
                    "max": "0"
                  },
                  "rows": {
                    "min": "0",
                    "max": "0"
                  }
                }
              },
              {
                "ows:Identifier": "EPSG:900913:1",
                "ScaleDenominator": "2.7954113197544646E8",
                "TopLeftCorner": "-2.003750834E7 2.0037508E7",
                "TileWidth": "256",
                "TileHeight": "256",
                "MatrixWidth": "2",
                "MatrixHeight": "2",
                // these ranges limit the tiles available for
the grid level
                "ranges": {
                  "cols": {
                    "min": "0",
                    "max": "1"
                  },
                  "rows": {
                    "min": "0",
                    "max": "1"
                  }
                }
              },
              {
                "ows:Identifier": "EPSG:900913:2",
                "ScaleDenominator": "1.3977056598772323E8",
                "TopLeftCorner": "-2.003750834E7 2.0037508E7",
                "TileWidth": "256",
                "TileHeight": "256",
                "MatrixWidth": "4",
                "MatrixHeight": "4",
                "ranges": {
                  "cols": {
                    "min": "0",
                    "max": "3"
                  },
                  "rows": {
                    "min": "0",
                    "max": "3"
                  }
```

```
                    }
                }
            ]
        }
    }
}
```

Bing

TODO

Google

> **Note**
>
> The use of Google maps tiles in MapStore is not enabled and maintained due to licensing reasons. If your usage conditions respect the google license, you can enable the google layers by:
>
> - Adding `<script src="https://maps.google.com/maps/api/js?v=3"></script>` to all `html` files you need it.
> - Add your API-KEY to the request
> - Fix the code, if needed.

example:

```
{
    "type": "google",
    "title": "Google HYBRID",
    "name": "HYBRID",
    "source": "google",
    "group": "background",
    "visibility": false
}
```

OSM

example:

```
{
    "type": "osm",
    "title": "Open Street Map",
    "name": "mapnik",
    "source": "osm",
    "group": "background",
    "visibility": true
}
```

TileProvider

TileProvider is a shortcut to easily configure many different layer sources. It's enough to add `provider` property and 'tileprovider' as type property to the layer configuration object. `provider` should be in the form of `ProviderName.VariantName`.

i.e.

```
{
    "type": "tileprovider",
    "title": "Title",
    "provider": "Stamen.Toner", // "ProviderName.VariantName"
    "name": "Name",
    "group": "GroupName",
    "visibility": false
}
```

Options passed in configuration object, if already configured by TileProvider, will be overridden.

Openlayers' TileProvider at the moment doesn't support `minZoom` configuration property and high resolution map.

In case of missing `provider` or if `provider: "custom"`, the tile provider can be customized and configured internally. You can configure the `url` as a template, than you can configure options add specific options (`maxNativeZoom`, `subdomains`).

```
{
    "type": "tileprovider",
```

```
    "title": "Title",
    "provider": "custom", // or undefined
    "name": "Name",
    "group": "GroupName",
    "visibility": false,
    "url": "https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png",
    "options": {
        "subdomains": [ "a", "b"]
    }
}
```

PROVIDERS AND VARIANTS

This is a *not maintained* list of providers and variants. For the most updated list check the code here

Some of them may need some additional configuration or API keys.

```
OpenStreetMap.Mapnik
OpenStreetMap.BlackAndWhite
OpenStreetMap.DE
OpenStreetMap.France
OpenStreetMap.HOT
Thunderforest.OpenCycleMap
Thunderforest.Transport
Thunderforest.TransportDark
Thunderforest.Landscape
Thunderforest.Outdoors
OpenMapSurfer.Roads
OpenMapSurfer.AdminBounds
OpenMapSurfer.Grayscale
Hydda.Full
Hydda.Base
Hydda.RoadsAndLabels
MapQuestOpen.OSM
MapQuestOpen.Aerial
MapQuestOpen.HybridOverlay
Stamen.Toner
Stamen.TonerBackground
Stamen.TonerHybrid
Stamen.TonerLines
Stamen.TonerLabels
Stamen.TonerLite
Stamen.Watercolor
Stamen.Terrain
Stamen.TerrainBackground
```

```
  Stamen.TopOSMRelief
  Stamen.TopOSMFeatures
  Esri.WorldStreetMap
  Esri.DeLorme
  Esri.WorldTopoMap
  Esri.WorldImagery
  Esri.WorldTerrain
  Esri.WorldShadedRelief
  Esri.WorldPhysical
  Esri.OceanBasemap
  Esri.NatGeoWorldMap
  Esri.WorldGrayCanvas
  OpenWeatherMap.Clouds
  OpenWeatherMap.CloudsClassic
  OpenWeatherMap.Precipitation
  OpenWeatherMap.PrecipitationClassic
  OpenWeatherMap.Rain
  OpenWeatherMap.RainClassic
  OpenWeatherMap.Pressure
  OpenWeatherMap.PressureContour
  OpenWeatherMap.Wind
  OpenWeatherMap.Temperature
  OpenWeatherMap.Snow
  HERE.normalDay
  HERE.normalDayCustom
  HERE.normalDayGrey
  HERE.normalDayMobile
  HERE.normalDayGreyMobile
  HERE.normalDayTransit
  HERE.normalDayTransitMobile
  HERE.normalNight
  HERE.normalNightMobile
  HERE.normalNightGrey
  HERE.normalNightGreyMobile
  HERE.carnavDayGrey
  HERE.hybridDay
  HERE.hybridDayMobile
  HERE.pedestrianDay
  HERE.pedestrianNight
  HERE.satelliteDay
  HERE.terrainDay
  HERE.terrainDayMobile
  Acetate.basemap
  Acetate.terrain
  Acetate.all
  Acetate.foreground
  Acetate.roads
  Acetate.labels
  Acetate.hillshading
```

```
CartoDB.Positron
CartoDB.PositronNoLabels
CartoDB.PositronOnlyLabels
CartoDB.DarkMatter
CartoDB.DarkMatterNoLabels
CartoDB.DarkMatterOnlyLabels
HikeBike.HikeBike
HikeBike.HillShading
BasemapAT.basemap
BasemapAT.grau
BasemapAT.overlay
BasemapAT.highdpi
BasemapAT.orthofoto
NASAGIBS.ModisTerraTrueColorCR
NASAGIBS.ModisTerraBands367CR
NASAGIBS.ViirsEarthAtNight2012
NASAGIBS.ModisTerraLSTDay
NASAGIBS.ModisTerraSnowCover
NASAGIBS.ModisTerraAOD
NASAGIBS.ModisTerraChlorophyll
NLS.OS_1900
NLS.OS_1920
NLS.OS_opendata
NLS.OS_6inch_1st
NLS.OS_6inch
NLS.OS_25k
NLS.OS_npe
NLS.OS_7th
NLS.OS_London
NLS.GSGS_Ireland
PDOK.brtachtergrondkaart
PDOK.brtachtergrondkaartgrijs
PDOK.brtachtergrondkaartpastel
PDOK.brtachtergrondkaartwater
PDOK.luchtfotoRGB
PDOK.luchtfotoIR
```

Vector

The layer type vector is the type used for imported data (geojson, shapefile) or for annotations. Generally speaking, any vector data added directly to the map. This is the typical fields of a vector layer

```
{
    "type":"vector",
```

```
    "features":[
        {
            "type":"Feature",
            "geometry":{
                "type":"Point",
                "coordinates":[
                12.516431808471681,
                41.89817370656741
                ]
            },
            "properties":{
            },
            "id":0
        }
    ],
    "style":{
        "weight":5,
        "radius":10,
        "opacity":1,
        "fillOpacity":0.1,
        "color":"rgba(0, 0, 255, 1)",
        "fillColor":"rgba(0, 0, 255, 0.1)"
    },
    "hideLoading":true
}
```

- `features` : features in GeoJSON format.
- `style` : the style object.
- `styleName` : name of a style to use (e.g. "marker").
- `hideLoading` : boolean. if true, the loading will not be taken into account.

WFS Layer

A vector layer, whose data source is a WFS service. The configuration has properties in common with both WMS and vector layers. it contains the search entry that allows to browse the data on the server side. The styling system is the same of the vector layer.

This layer differs from the "vector" because all the loading/filtering/querying operations are applied directly using the WFS service, without storing anything locally.

```
```

```
{
    "type":"wfs",
    "search":{
        "url":"https://myserver.org/geoserver/wfs",
        "type":"wfs"
    },
    "name":"workspace:layer",
    "styleName":"marker",
    "url":"https://myserver.org/geoserver/wfs"
}
```

Vector Style

The `vector` and `wfs` layer types are rendered by the client as GeoJSON features and it possible to apply specific symbolizer using the `style` property available in the layer options. The style object is composed by these properties

- `format` the format encoding used by style body
- `body` the actual style rules and symbolizers

example:

```
{
  "type": "vector",
  "features": [],
  "style": {
    "format": "geostyler",
    "body": {
      "name": "My Style",
      "rules": [
        {
          "name": "My Rule",
          "symbolizers": [
            {
              "kind": "Line",
              "color": "#3075e9",
              "opacity": 1,
              "width": 2
            }
          ]
        }
      ]
    }
```

```
    }
  }
```

The default format used by MapStore is "geostyler" that is an encoding based on the geostyler-style specification that could include some variations or limitations related to the map libraries used by MapStore app. We suggest to refer to following doc for the rule/symbolizer properties available in MapStore.

Ths style `body` is composed by following properties:

- `name` style name
- `rules` list of rule object that describe the style

A `rule` object is composed by following properties:

- `name` rule name that could be used to generate a legend
- `filter` filter expression
- `symbolizers` list of symbolizer object that describe the rule (usually one per rule)

The `filter` expression define with features should be rendered with the symbolizers listed in the rule

example:

```
// simple comparison condition structure
// [operator, property key, value]
{
  "filter": ["==", "count", 10]
}

// mulitple condition with logical operato
// [logical operator, [condition], [condition]]
{
  "filter": [
    "||",
    [">", "height", 10],
    ["==", "category", "building"]
  ]
}
```

Available logical operators:

- `||` OR operator
- `&&` AND operator

Available comparison operators:

- `==` equal to
- `*=` like (for string type)
- `!=` is not
- `<` less than
- `<=` less and equal than
- `>` grater than
- `>=` grater and equal than

The `symbolizer` could be of following `kinds`:

- `Mark` symbolizer properties
- `kind` must be equal to `Mark`
- `color` fill color of the mark
- `fillOpacity` fill opacity of the mark
- `strokeColor` stroke color of the mark
- `strokeOpacity` stroke opacity of the mark
- `strokeWidth` stroke width of the mark
- `radius` radius size in px of the mark
- `msBringToFront` this boolean will allow setting the `disableDepthTestDistance` value for the feature. This would only apply on Cesium maps.
- `wellKnownName` rendered shape, one of Circle, Square, Triangle, Star, Cross, X, shape://vertline, shape://horline, shape://slash, shape://backslash, shape://dot, shape://plus, shape://times, shape://oarrow or shape://carrow
- `Icon` symbolizer properties

- `kind` must be equal to `Icon`
- `image` url of the image to use as icon
- `size` size of the icon
- `opacity` opacity of the icon
- `rotate` rotation of the icon
- `msBringToFront` this boolean will allow setting the `disableDepthTestDistance` value for the feature. This would only apply on Cesium maps.
- `Line` symbolizer properties
- `kind` must be equal to `Line`
- `color` stroke color of the line
- `opacity` stroke opacity of the line
- `width` stroke width of the line
- `dasharray` array that represent the dashed line intervals
- `msClampToGround` this boolean will allow setting the `clampToGround` value for the feature. This would only apply on Cesium maps.
- `Fill` symbolizer properties
- `kind` must be equal to `Fill`
- `color` fill color of the polygon
- `fillOpacity` fill opacity of the polygon
- `outlineColor` outline color of the polygon
- `outlineOpacity` outline opacity of the polygon
- `outlineWidth` outline width of the polygon
- `msClassificationType` allow setting `classificationType` value for the feature. This would only apply on polygon graphics in Cesium maps.
- `msClampToGround` this boolean will allow setting the `clampToGround` value for the feature. This would only apply on Cesium maps.
- `Text` symbolizer properties
- `kind` must be equal to `Text`

- `label` text to show in the label, the {{propertyKey}} notetion allow to access feature properties (eg. 'feature name is {{name}}')
- `font` array of font family names
- `size` font size of the label
- `fontStyle` font style of the label: normal or italic
- `fontWeight` font style of the label: normal or bold
- `color` font color of the label
- `haloColor` halo color of the label
- `haloWidth` halo width of the label
- `offset` array of x and y values offset of the label

Legacy Vector Style (deprecated)

The `style` or `styleName` properties of vector layers (wfs, vector...) allow to apply a style to the local data on the map.

- `style` : a style object/array. It can have different formats. In the simplest case it is an object that uses some leaflet-like style properties:
- `weight` : width in pixel of the border / line.
- `radius` : radius of the circle (valid only for Point types)
- `opacity` : opacity of the border / line.
- `color` : color of the border / line.
- `fillOpacity` : opacity of the fill if any. (Polygons, Point)
- `fillColor` : color of the fill, if any. (Polygons, Point)
- `styleName` : if set to `marker` , the `style` object will be ignored and it will use the default marker.

In case of `vector` layer, style can be added also to the specific features. Other ways of defining the style for a vector layer have to be documented.

Advanced Vector Styles (deprecated)

To support advanced styles (like multiple rules, symbols, dashed lines, start point, end point) the style can be configured also in a different format, as an

array of objects and you can define them feature by feature, adding a "style" property.

> **Warning**
>
> This advanced style functionality has been implemented to support annotations, at the moment this kind of advanced style options is supported **only** as a property of the single feature object, not as global style.

SVG SYMBOL (DEPRECATED)

The following options are available for a SVG symbol.

- `symbolUrl` : a URL (also a data URL is ok) for the symbol to use (SVG format). You can anchor the symbol using:
- `iconAnchor` : array of x,y position of the offset of the symbol from top left corner.
- `anchorXUnits` , `anchorYUnits` unit of the `iconAnchor` (`fraction` or `pixels`).
- `size` : the size in pixel of the square that contains the symbol to draw. The size is used to center and to cut the original svg, so it must fit the svg.
- `dashArray` : Array of line, space size, in pixels. ["6","6"] Will draw the border of the symbol dashed. It is applied also to a generic line or polygon geometry.

MARKERS AND GLYPHS (DEPRECATED)

These are the available options for makers. These are specific of annotations for now, so allowed values have to be documented.

- `iconGlyph` : e.g. "shopping-cart"
- `iconShape` : e.g. "circle"
- `iconColor` : e.g. "red"
- `iconAnchor` : [0.5,0.5]

MULTIPLE RULES AND FILTERING (DEPRECATED)

In order to support start point and end point symbols, you could find in the style these entries:

- `geometry` : "endPoint"|"startPoint", identify how to get the geometry from
- `filtering` : if true, the geometry filter is applied.

Example (deprecated)

Here an example of a layer with:

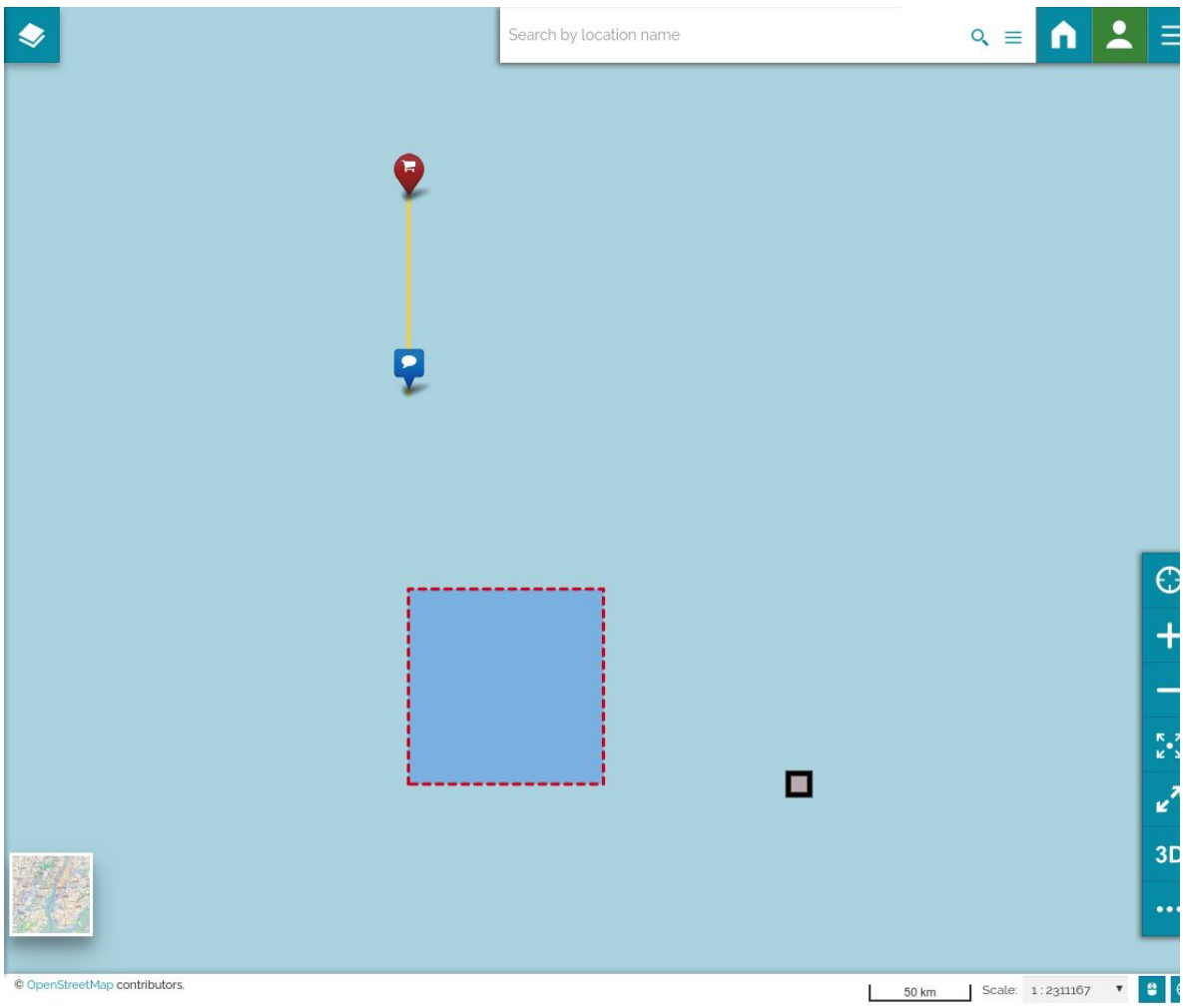
- a point styled with SVG symbol,
- a polygon with dashed style
- a line with start-end point styles as markers with icons

```
{
        "type": "vector",
        "visibility": true,
        "id": "styled-vector-layer",
        "name": "styled-vector-layer",
        "hideLoading": true,
        "features": [
          {
            "type": "Feature",
            "geometry": {
              "type": "Point",
              "coordinates": [2,0]
            },
            "properties": {},
            "style": [
              {
                "iconAnchor": [0.5,0.5],
                "anchorXUnits": "fraction",
                "anchorYUnits": "fraction",
                "opacity": 1,
                "size": 30,
                "symbolUrl": "data:image/svg+xml,%3Csvg
xmlns='http://www.w3.org/2000/svg' width='30'
height='30'%3E%3Crect  x='5' y='5' width='20' height='20'
style='fill:rgb(255,0,0);stroke-width:5;stroke:rgb(0,0,0)' /
%3E%3C/svg%3E",
                "shape": "triangle",
                "id": "c65cadc0-9b46-11ea-a138-dd5f1faf9a0d",
```

```
                "highlight": false,
                "weight": 4
              }
            ]
          },{
            "type": "Feature",
            "geometry": {
              "type": "Polygon",
              "coordinates": [[[0, 0],[1, 0],[1, 1],[0,1],[ 0,
0]]]
            },
            "properties": {},
            "style": [
              {
                "color": "#d0021b",
                "opacity": 1,
                "weight": 3,
                "fillColor": "#4a90e2",
                "fillOpacity": 0.5,
                "highlight": false,
                "dashArray": ["6","6"]
              }
            ]
          },{
            "type": "Feature",
            "geometry": {
              "coordinates": [[0, 2],[ 0,3]],
              "type": "LineString"
            },
            "properties": {},
            "style": [
              {
                "color": "#ffcc33",
                "opacity": 1,
                "weight": 3,
                "editing": {
                  "fill": 1
                },
                "highlight": false
              },
              {
                "iconGlyph": "comment",
                "iconShape": "square",
                "iconColor": "blue",
                "highlight": false,
                "iconAnchor": [ 0.5,0.5],
                "type": "Point",
                "title": "StartPoint Style",
                "geometry": "startPoint",
```

```
            "filtering": true
          },
          {
            "iconGlyph": "shopping-cart",
            "iconShape": "circle",
            "iconColor": "red",
            "highlight": false,
            "iconAnchor": [ 0.5,0.5 ],
            "type": "Point",
            "title": "EndPoint Style",
            "geometry": "endPoint",
            "filtering": true
          }
        ]
      }
    ]
  }
```

Result:



Graticule

i.e.

```
{
    "type": "graticule",
    "labels": true,
    "frame": true, // adds a frame to the map, to better
highlight labels
    "frameRatio": 0.07, // frame percentage size (7%)
    "style": { // style for the grid lines
        "color": "#000000",
        "weight": 1,
        "lineDash": [0.5, 4],
        "opacity": 0.5
    },
    "frameStyle": { // style for the optional frame
        "color": "#000000",
        "weight": 1,
        "fillColor": "#FFFFFF"
    },
    "labelXStyle": { // style for X coordinates labels
        "color": "#000000",
        "font": "sans-serif",
        "fontWeight": "bold",
        "fontSize": "20",
        "labelOutlineColor": "#FFFFFF",
        "labelOutlineWidth": 2
    },
    "labelYStyle": { // style for Y coordinates labels
        "color": "#000000",
        "font": "sans-serif",
        "fontWeight": "bold",
        "fontSize": "20",
        "labelOutlineColor": "#FFFFFF",
        "labelOutlineWidth": 2,
        "rotation": 90,
        "verticalAlign": "top",
        "textAlign": "center"
    }
}
```

3D tiles

This type of layer shows 3d tiles version 1.0 inside the Cesium viewer. This layer will not be visible inside 2d map viewer types: openlayer or leaflet. See specification for more info about 3d tiles here.

i.e.

```
{
    "type": "3dtiles",
    "url": "http..." // URL of tileset.json file
    "title": "3D tiles layer",
    "visibility": true,
    // optional
    "heightOffset": 0, // height offest applied to the complete
tileset
    "style": {
      "format": "3dtiles",
      "body": { // 3d tiles style
        "color": "color('#43a2ca', 1)"
      }
    }
}
```

The style body object for the format 3dtiles accepts rules described in the 3d tiles styling specification version 1.0 available here.

Terrain

`terrain` layer allows the customization of the elevation profile of the globe mesh in the Cesium 3d viewer. Currently Mapstore supports three different types of terrain providers. If no `terrain` layer is defined the default elevation profile for the globe would be the ellipsoid that provides a rather flat profile.

The other two available terrain providers are the `wms` (that supports `DDL/BIL` types of assets) and the `cesium` (that support resources compliant with the Cesium terrain format).

In order to create a `wms` based mesh there are some requirements that need to be fulfilled:

- a GeoServer WMS service with the DDS/BIL plugin
- A WMS layer configured with **BIL 16 bit** output in **big endian mode** and **-9999 nodata value**
- BILTerrainProvider is used to parse `wms` based mesh. Supports three ways in parsing the metadata of the layer

  a. Layer configuration with **sufficient metadata** of the layer. This prevents a call to `getCapabilities` eventually improving performance of the parsing of the layer. Mandatory fields are `url`, `name`, `crs`.

```
{
  "type": "terrain",
  "provider": "wms",
  "url": "http://hot-sample/geoserver/wms",
  "name": "workspace:layername",
  "littleEndian": false,
  "visibility": true,
  "crs": "CRS:84" // Supports only CRS:84 | EPSG:4326 |
EPSG:3857 | OSGEO:41001
}
```

  b. Layer configuration of `geoserver` layer with layer name *prefixed with workspace*, then the `getCapabilities` is requested only for that layer

```
{
"type": "terrain",
"provider": "wms",
"url": "https://host-sample/geoserver/wms", //
'geoserver' url
"name": "workspace:layername", // name of the geoserver
resource with workspace prefixed
"littleEndian": false
}
```

  c. Layer configuration of geoserver layer with layer name *not prefixed with workspace* then `getCapabilities` is requested in global scope.

```
{
  "type": "terrain",
  "provider": "wms",
  "url": "https://host-sample/geoserver/wms",
  "name": "layername",
  "littleEndian": false
}
```

> **Note**
>
> With `wms` as provider, the format option is not needed, as Mapstore supports only `image/bil` format and is used by default

Generic layer configuration of type `terrain` and provide `wms` as follows. The layer configuration needs to point to the geoserver resource and define the type of layer and the type of provider, here all available properties:

```
{
  "type": "terrain",
  "provider": "wms",
  "url": "https://host-sample/geoserver/wms",
  "name": "workspace:layername", // name of the geoserver
resource
  "littleEndian": false, // defines whether buffer is in little
or big endian
  "visibility": true,
  // optional properties
  "crs": "CRS:84", // projection of the layer, support only CRS:
84 | EPSG:4326 | EPSG:3857 | OSGEO:41001
  "version": "1.3.0", // version used for the WMS request
  "heightMapWidth": 65, // width  of a tile in pixels, default
value 65
  "heightMapHeight": 65, // height of a tile in pixels, default
value 65
  "waterMask": false,
  "offset": 0, // offset of the tiles (in meters)
  "highest": 12000, // highest altitude in the tiles (in meters)
  "lowest": -500, // lowest altitude in the tiles
  "sampleTerrainZoomLevel": 18 // zoom level used to perform
sampleTerrain and get the height value given a point, used by
measure components

}
```

The `terrain` layer of `cesium` type allows using Cesium terrain format compliant services (like Cesium Ion resources or MapTiler meshes). The options attributte allows for further customization of the terrain properties (see available options on the Cesium documentation for the cesium terrain provider)

```
{
  "type": "terrain",
  "provider": "cesium",
  "url": "https://terrain-provider-service-url/?key={apiKey}",
  "visibility": true,
  "options": {
    // requestVertexNormals, requestWatermask, credit...
  }
}
```

In order to use these layers they need to be added to the `additionalLayers` in `localConfig.json`. The globe only accepts one terrain provider so in case of adding more than one the last one will take precedence and be used to create the elevation profile.

```
{
    "name": "Map",
    "cfg": {
        "additionalLayers": [{
            "type": "terrain",
            "provider": "wms",
            "url": "https://host-sample/geoserver/wms",
            "name": "workspace:layername",  // name of the
geoserver resource
            "littleEndian": false,
            "visibility": true,
            "crs": "CRS:84"
        }]
    }
}
```

Layer groups

Inside the map configuration, near the `layers` entry, you can find also the `groups` entry. This array contains information about the groups in the TOC. A group entry has this shape:

- `id`: the id of the group.
- `expanded`: boolean that keeps the status (expanded/collapsed) of the group.
- `title`: a string or an object (for i18n) with the title of the group. i18n object format is the same of layer's title.

```
"title": {
        "default": "Root Group",
        "it-IT": "Gruppo radice",
        "en-US": "Root Group",
        "de-DE": "Wurzelgruppe",
        "fr-FR": "Groupe Racine",
```

```
          "es-ES": "Grupo Raíz"
        },
```

i.e.

```
{
  "id": "GROUP_ID",
  "title": "Some default title"
  "expanded": true
}
```

Other supported formats

The JSON format above is the standard MapStore format. Anyway MapStore allows to import/export different kinds of formats for maps.

Web Map Context

MapStore provides support for OGC Web Map Context(WMC) files. They can be imported either using Import plugin functionality, or from within a context using Map Templates plugin. MapStore maps can also be exported in WMC format through Export plugin.

The important thing to remember when exporting MapStore maps to WMC format is that it only supports WMS layers, meaning any non-WMS layers(such as tiled OSM backgrounds for example) will not be preserved in the resulting WMC file. The exact way in which the conversion happens is described in further detail throughout this document.

WMC File Structure

WMC context file generated by MapStore is an XML file with the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<ViewContext xmlns="http://www.opengis.net/context"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
version="1.1.0" xsi:schemaLocation="http://www.opengis.net/
context http://schemas.opengis.net/context/1.1.0/context.xsd">
    <General>
        <Title>MapStore Context</Title>
        <Abstract>This is a map exported from MapStore2.</
Abstract>
        <BoundingBox minx="-20037508.34" miny="-20037508.34"
maxx="20037508.34" maxy="20037508.34" SRS="EPSG:900913"/>
        <Extension>
            <!--general extensions go here-->
        </Extension>
    </General>
    <LayerList>
        <Layer queryable="0" hidden="0">
        <Name>topp:states</Name>
        <Title>USA Population</Title>
        <Server service="OGC:WMS" version="1.3.0">
            <OnlineResource xmlns:xlink="http://www.w3.org/1999/
xlink" xlink:type="simple" xlink:href="https://demo.geo-
solutions.it/geoserver/wms"/>
        </Server>
        <DimensionList>
            <Dimension name="elevation" units="EPSG:5030"
unitSymbol="m" default="0.0"
multipleValues="1">0.0,200.0,400.0,600.0</Dimension>
            <Dimension name="time" units="ISO8601"
default="current" multipleValues="1">2016-02-23T03:00:00.000Z,
2016-02-23T06:00:00.000Z</Dimension>
            <!--...other dimensions-->
        </DimensionList>
        <FormatList>
            <Format current="1">image/png</Format>
            <!--...other formats-->
        </FormatList>
        <StyleList>
            <Style>
                <Name>population</Name>
                <Title>Population in the United States</Title>
                <LegendURL width="81" height="80" format="image/
png">
                    <OnlineResource xmlns:xlink="http://
www.w3.org/1999/xlink" xlink:type="simple" xlink:href="https://
demo.geo-solutions.it:443/geoserver/topp/states/ows?
service=WMS&amp;request=GetLegendGraphic&amp;format=image%2Fpng&am
>
                </LegendURL>
            </Style>
            <!--...other styles-->
        </StyleList>
```

```
        <Extension>
            <!--layer extensions go here-->
        </Extension>
        <!--...other layers-->
    </LayerList>
</ViewContext>
```

More information about each of the elements in the example above can be looked up in OGC WMC implementation specification

Apart from standard WMC XML elements, MapStore provides support for various extensions. These are placed inside `Extension` tag, and are not gueranteed to be supported outside MapStore, as they are not a part of OGC Web Map Context specification. MapStore provides two types of extensions: openlayers and mapstore-specific elements. WMC can have an `Extension` element inside `General`, and each of the `Layer` elements. Supported extensions in `General` are:

Openlayers:

- `maxExtent` if present, it's attributes are used as map's bounding box, instead of the values specified in `BoundingBox` tag. The values are assumed to be in a projection, specified in SRS attribute of `BoundingBox`

```
<ol:maxExtent xmlns:ol="http://openlayers.org/context"
minx="-20037508.34" miny="-20037508.34" maxx="20037508.34"
maxy="20037508.34"/>
```

MapStore specific:

- `GroupList` defines a mapstore group list. Contains `Group` elements that describe a particular layer group:

```
<ms:GroupList xmlns:ms="http://geo-solutions.it/mapstore/
context">
    <ms:Group xmlns:ms="http://geo-solutions.it/mapstore/
```

```
context" id="Default" title="Default" expanded="true"/>
</ms:GroupList>
```

- `center` defines a center of map view

```
<ms:center xmlns:ms="http://geo-solutions.it/mapstore/context"
x="1.5" y="2.5" crs="EPSG:3857"/>
```

- `zoom` map zoom level

```
<ms:zoom xmlns:ms="http://geo-solutions.it/mapstore/context">7</
ms:zoom>
```

Supported extensions for each `Layer` element are:

Openlayers:

- `maxExtent` if present, used for the value of layer's bbox. Values are assumed to be in a projection, specified in SRS attribute of "BoundingBox"
- `singleTile` specifies layer's "singleTile" property value
- `transparent` is layer transparent or not, true by default
- `isBaseLayer` if true, the layer is put into "backgrounds" group
- `opacity` layer's opacity value

```
<ol:maxExtent xmlns:ol="http://openlayers.org/context"
minx="-13885038.382960921" miny="2870337.130793682"
maxx="-7455049.489182421" maxy="6338174.0557576185"/>
<ol:singleTile xmlns:ol="http://openlayers.org/context">false</
ol:singleTile>
<ol:transparent xmlns:ol="http://openlayers.org/context">true</
ol:transparent>
<ol:isBaseLayer xmlns:ol="http://openlayers.org/context">false</
ol:isBaseLayer>
<ol:opacity xmlns:ol="http://openlayers.org/context">1</
ol:opacity>
```

Cesium:

- `tileDiscardPolicy` sets a policy for discarding (missing/broken) tiles (https://cesium.com/learn/cesiumjs/ref-doc/TileDiscardPolicy.html). If it is not specified the NeverTileDiscardPolicy will be used. If "none" is specified, no policy at all will be set.

MapStore specific:

- `group` specifies to which group, among listed in "GroupList" element, the layer belongs to
- `search` JSON describing a filter that is applied to the layer
- `DimensionList` contains `Dimension` elements that describe dimensions that cannot be described using standard "Dimension" tag. Currently supports dimensions of ***multidim-extension*** type:
- `CatalogServices` contains `Service` elements that describe services available for use in Catalog.

```
<ms:DimensionList xmlns:ms="http://geo-solutions.it/mapstore/
context">
    <ms:Dimension xmlns:ms="http://geo-solutions.it/mapstore/
context" xmlns:xlink="http://www.w3.org/1999/xlink" name="time"
type="multidim-extension" xlink:type="simple"
xlink:href="https://cloudsdi.geo-solutions.it/geoserver/gwc/
service/wmts"/>
</ms:DimensionList>
<ms:CatalogServices selectedService="gs_stable_csw">
    <ms:Service serviceName="gs_stable_csw">
        <ms:Attribute name="url" type="string">https://gs-
stable.geo-solutions.it/geoserver/csw</ms:Attribute>
        <ms:Attribute name="type" type="string">csw</
ms:Attribute>
        <ms:Attribute name="title" type="string">GeoSolutions
GeoServer CSW</ms:Attribute>
        <ms:Attribute name="autoload" type="boolean">true</
ms:Attribute>
    </ms:Service>
    <ms:Service serviceName="gs_stable_wms">
        <ms:Attribute name="url" type="string">https://gs-
stable.geo-solutions.it/geoserver/wms</ms:Attribute>
        <ms:Attribute name="type" type="string">wms</
ms:Attribute>
```

```
        <ms:Attribute name="title" type="string">GeoSolutions
GeoServer WMS</ms:Attribute>
        <ms:Attribute name="autoload" type="boolean">false</
ms:Attribute>
    </ms:Service>
    <ms:Service serviceName="gs_stable_wmts">
        <ms:Attribute name="url" type="string">https://gs-
stable.geo-solutions.it/geoserver/gwc/service/wmts</
ms:Attribute>
        <ms:Attribute name="type" type="string">wmts</
ms:Attribute>
        <ms:Attribute name="title" type="string">GeoSolutions
GeoServer WMTS</ms:Attribute>
        <ms:Attribute name="autoload" type="boolean">false</
ms:Attribute>
    </ms:Service>
</ms:CatalogServices>
```

Note, that during the exporting process, some sort of fallback for dimensions, listed as extensions, is provided inside the standard `DimensionList` tag whenever possible, to ensure interoperability with other geospatial software. When such a context is imported back into MapStore, the values of dimensions inside extensions will override the ones specified inside the standard `DimensionList` tag.

Also note, that the extension elements would be read correctly only if they belong to appropriate XML namespaces:

- `http://openlayers.org/context` for openlayers extensions
- `http://geo-solutions.it/mapstore/context` for mapstore specific extensions

Usage inside MapTemplates plugin

As stated previously, WMC files can be used as map templates inside contexts. New WMC templates can be uploaded in context creation tool, after enabling the MapTemplates plugin for a context. When the context is loaded, for every template inside MapTemplates there are two options available:

- `Replace map with this template` replace the currently loaded map with the one described by the template. Upon loading, the map will zoom to the

extent specified in `maxExtent` extension or in `BoundingBox` tag. If the template has no visible background layers available, the default empty background will be added and set to be visible automatically.

- `Add this template to map` merges layers and groups inside the template with the current map configuration. If the WMC template does not contain `GroupList` extension, a new group with the name extracted from `Title` tag of the template will be created and will contain all the layers of the template. Zoom and projection will remain unchanged.

Other considerations

Due to the limitations posed by WMC format the conversion process will not preserve the map state in it's entirety. The only supported way to do this is to export to MapStore JSON format. The WMC export option presumably should be used in cases when the WMS layers inside a MapStore map need to be used in some way with a different geospatial software suite, or to import such layers from outside MapStore or if you already have WMC context files that you want to use.

Externalized Configuration

The **data directory** is a directory on the file-system, configured for an instance of MapStore, that will be used to externalize configuration of MapStore.

Configuring this directory you will be able to:

- Externalize database configuration
- Externalize proxy configuration
- Externalize JSON configs files for the application (`localConfig.json` , `new.json`)
- Apply patches to default JSON config files (e.g. to store only the differences)
- Store extensions installed

All the configuration stored here will persist across MapSore updates.

Using a data directory

To use a data directory, this must be configured through a specific JVM system property: `datadir.location`

```
java -Ddatadir.location=/etc/mapstore/datadir
```

The data-directory must exist, but all the files inside it are optional. Due to some particular operations (e.g. installation of extensions), some files may be stored in data-dir by the application itself.

The structure of the data-dir is the following:

```
.
├── configs                   (JSON configs)
│   └── pluginsConfig.json.patch
├── extensions
│   ├── extensions.json       (extensions index)
```

```
|   └── SampleExtension      (One directory for each extension
installed)
|       ├── index.js
|       ├── assets
|       └── translations
├── geostore-datasource-ovr.properties (database configuration)
├── ldap.properties
├── mapstore-ovr.properties
└── mapstore.properties
```

- `configs` : files in this folder can override the files in `configs` file of the application (`pluginsConfig.json` , `localConfig.json`).
- If a file with the same name is present, it will be provided instead of the original one
- If a patch file is present,(e.g. `localConfig.json.patch`) the patch will be applied to the JSON (original or overridden) and provided patched to the client
- `extensions` : this folder contains all the files for the installed extensions, one folder for each installed extension
- `extensions.json` : the index of the current extensions installed.

Multiple data directory locations

It is possible to specify more than one datadir location path, separated by commas. This can be useful if you need to have different places for static configuration and dynamic one. A dynamic configuration file is one that is updated by MapStore using the UI, while static ones can only updated manually by an administrator of the server. An example are uploaded extensions, and their configuration files.

MapStore looks for configuration resources in these places in order:

- the first datadir.location path other datadir.location paths, if any, in order
- the application root folder

Dynamic files will always be written by the UI in the first location in the list, so the first path is for dynamic stuff.

Example:

```
-Ddatadir.location=/etc/mapstore_extensions,/etc/
mapstore_static_config
```

Logging

Logging has not been externalized yet, You can manually do this change in `WEB-INF/web.xml` file to externalize also this file:

```
<context-param>
    <param-name>log4jConfigLocation</param-name>
    <param-value>file:${datadir.location}/log4j.properties</
param-value>
</context-param>
```

Database Connection

If you create a file in the datadir called `geostore-datasource-ovr.properties` , it will be used and override the current

Example:

```
geostoreDataSource.driverClassName=org.postgresql.Driver
geostoreDataSource.url=jdbc:postgresql://localhost:5432/geostore
geostoreDataSource.username=geostore
geostoreDataSource.password=geostore
geostoreVendorAdapter.databasePlatform=org.hibernate.dialect.Postg
geostoreEntityManagerFactory.jpaPropertyMap[hibernate.hbm2ddl.auto
geostoreEntityManagerFactory.jpaPropertyMap[hibernate.default_sche
geostoreVendorAdapter.generateDdl=true
geostoreVendorAdapter.showSql=false
```

NOTE: this file simply overrides the values in `geostore-datasource-ovr.properties` in the web-application, it will not replace it usually it is configured by default to use h2 database, so configuring the database (h2, postgreSQL or oracle) will override all the properties. Anyway if you changed this file in your project, you may need to override more variables to make it work

Overriding front-end configuration

Externalizing the whole `localConfig.json` file allows to keep your configurations during the various updates. Anyway keeping this long file in sync can become hard. You can use patch files, and this is the first suggested option.

Anyway if you need to specify something in `localConfig.json` that comes from your Java application, MapStore gives you the possibility to override only some specific properties of this big file and keep these changes separated from the application, allowing an easier updates. This is particularly useful for example when you have to change only a bunch of settings on a specific instance, and use the standard configuration for everything else.

You can override one or more properties in the file using the following JVM flags:

- `overrides.config` : the path of a properties file (**relative to the datadir**) where override values are stored
- `overrides.mappings` : comma limited list of JSONPath=property values to override

An example of overrides that will replace the default WMS service url:

In `mapstore.properties` :

```
overrides.config=env.properties
overrides.mappings=initialState.defaultState.catalog.default.servi
```

In `datadir_path/env.properties` :

```
geoserverUrl=https://demo.geo-solutions.it/geoserver/wms
```

This allows to have in `datadir_path/env.properties` a set of variables that can be used in overrides (even in different places) that are indicated by `overrides.mappings`.

Note: `env.properties` should **not be placed** in classpath folder

Patching front-end configuration

Another option is to patch the frontend configuration files, instead of overriding them completely, using a patch file in json-patch format.

To patch one of the allowed resources you can put a file with a **.patch** extension in the datadir folder (e.g. localConfig.json.patch) and that file will be merged with the main localConfig.json to produce the final resource.

This allows easier migration to a new MapStore version. Please notice that when you use a patch file, any new configuration from the newer version will be applied automatically. This can be good or bad: the good part is that new plugins and features will be available without further configuration after the migration, the bad part is that you won't be aware that new plugins and features will be automatically exposed to the final user.

Example: adding a plugin to the localConfig.json configuration file:

```
[{"op": "add", "path": "/plugins/desktop/-", "value":
"MyAwesomePlugin"}]
```

Externalize front-end Configurations

From version 2021.02.xx, the externalization of the front-end files is automatic on the back-end, as well as you configure the data-directory. Anyway for your custom application you may want to customize the following paths to point your own services for configuration, extensions, and so on. The paths can be customized by adding the relative line in the `app.jsx` :

- *Application* (`localConfig.json`):

```
ConfigUtils.setLocalConfigurationFile("configs/
localConfig.json");
```

- *Static maps* (`new.json` and `config.json`):

```
ConfigUtils.setConfigProp("configurationFolder", "configs/");
```

- *Extensions configuration* (`extensions.json`):

```
ConfigUtils.setConfigProp("extensionsRegistry", "extensions/
extensions.json");
```

- *Context Editor* (`pluginsConfig.json`):

```
ConfigUtils.setConfigProp("contextPluginsConfiguration",
"configs/pluginsConfig.json");
```

- *Extensions folder* (folder where to get the extensions found in `extensions.json`):

```
ConfigUtils.setConfigProp("extensionsFolder", "extensions/");
```

> **Note**
>
> Because in this case we are modifying the `app.jsx` file, these changes can be applied only at build time in a custom project. Future improvements will allow to externalize these files also in the main product, without any need to rebuild the application.

Configuration of Application Context Manager

The Application Context Manager can be configured editing the `configs/pluginsConfig.json` file.

The configuration file has this shape:

```
{
 "plugins": [
    {
        "name": "Map",
        "mandatory": true, // <-- mandatory should not be shown
in editor OR not movable and directly added to the right list.
    }, {
        "name": "Notifications",
        "mandatory": true, // <-- mandatory should not be shown
in editor OR not movable and directly added to the right list.
             "hidden": true, // some plugins are only support,
so maybe showing them in the UI is superfluous.
    }, {
        "name": "TOC",
       "symbol": "layers",
        "title": "plugins.TOC.title",
        "description": "plugins.TOC.description",
        "defaultConfig": {},
        "children": ["TOCItemSettings", "FeatureEditor"]
    }, {
        "name": "FeatureEditor",
        "defaultConfig":  {}
    }, {
        "name": "TOCItemSettings",
        "...": "..."
    }, {
        "name": "MyPlugin", // <-- this is typically an
extension,
        "docUrl": "https://domain.com/documentation"  // <--
custom documentation url
    }, {
       "name": "Footer",
       "children": ["MousePosition", "CRSSelector",  "ScaleBox"]
    }, {
```

```
        "name": "Widgets",
      "children": ["WidgetsBuilderPlugin", "WidgetsTrayPlugin"],
      "dependencies": ["WidgetsBuilderPlugin"], // some plugins
may be mandatory only if parent is added.
    }, {
      "name": "WidgetsTrayPlugin"
    },  {
      "name": "WidgetsBuilderPlugin",
      "hidden": true // <-- This is a child. In this case it
will be added automatically,
                    // without showing if the parent is added
}]
}
```

The configuration contains the list of available plugins to display in the plugins selector. Each entry of `plugins` array is an object that describes the plugin, it's dependencies and it's properties. These are the properties allowed for the plugin entry object:

- `name` : `{string}` the name (ID) of the plugin
- `title` : `{string}` the title string OR messageId (from localization file)
- `description` : `{string}` : the description string OR messageId (from localization file)
- `docUrl` : `{string}` : the plugin/extension specific documentation url
- `symbol` : {string}`: icon (or image) symbol for the plugin
- `defaultConfig` `{object}` : optional object containing the default configuration to pass to the context-creator.
- `mandatory` `{boolean}` : if true, the plugin must be added to the map, so not possible to remove (but can be customized)
- `hidden` `{boolean}` : if true, the plugin should not be shown in UI. If mandatory, is added without showing.
- `children` `{string[]}` : list of the plugins names (ID) that should be shown as children in the UI
- `dependencies` : The difference between mandatory and dependencies is the "if the parent is present" condition.). Plugins that can not be disabled (or if are hidden, added by default) and are added ONLY if the parent plugin is

added. (e.g. containers like toolbar, omnibar, footer or DrawerMenu, and other dependencies like Widgets that must contain WidgetsBuilder and so on)

Database Setup

MapStore can use 3 types of database:

- H2
- PostgreSQL
- Oracle

MapStore uses an H2 in-memory DB as the default DBMS to persist the data. This configuration is useful for development and test purposes, or to evaluate the project but it is obviously NOT RECOMMENDED for production usage; moreover the H2 DB cannot be used for the integration with GeoServer.

In the following guide you will learn how to configure MapStore to use an external database.

Externalize properties files

MapStore has a file called `geostore-datasource-ovr.properties`. This file is on the repository in the folder `java/web/src/main/resources`, in the final `mapstore.war` package it will be copied into `WEB-INF/classes` path. It contains the set-up for the database connection. Anyway if you edit the file in `WEB-INF/classes` this file will be overridden on the next re-deploy. To preserve your configuration on every deploy you can use an environment variable, `geostore-ovr`, to configure the path to an override file in a different, external directory. In this file the user can re-define the default configuration and so set-up the database configuration.

For instance using tomcat on linux you will have to do something like this to add the environment variable to the JAVA_OPTS

> where to add your JAVA_OPTS depends on your operating system. For instance the file could be `/etc/default/tomcat8`, or similar, in linux debian

```
```

```
# here the path to the ovr file
GEOSTORE_OVR_FILE=file:///var/lib/tomcat/conf/geostore-
ovr.properties

# add the env. variable 'geostore-ovr' to JAVA_OPTS
JAVA_OPTS="-Dgeostore-ovr=$GEOSTORE_OVR_FILE [other opts]"
```

So your file `/var/lib/tomcat/conf/geostore-ovr.properties` will contain the overrides to the database set-up.

Database creation Mode

By default MapStore automatically populates the database on it's own. If you want to disable this functionality (e.g. if you don't want to allow the database user to have permission to create tables) then you have to set-up the following property in the ovr file to 'validate'

```
geostoreEntityManagerFactory.jpaPropertyMap[hibernate.hbm2ddl.auto
```

> Options are:
>
> - `validate` : validate the schema, makes no changes to the database.
> - `update` : update the schema.
> - `create` : creates the schema, destroying previous data.
> - `create-drop` : drop the schema when the SessionFactory is closed explicitly, typically when the > application is stopped.

In this case it is necessary to manually create the required tables using the scripts available here for the needed DBMS.

The `update` mode is usually discouraged in production. On production servers you should always use `validate` mode and apply SQL scripts and/or patches manually. Anyway before every update a database backup is strongly suggested.

H2

If you download or build mapStore.war, it's default configuration will be this one:

```
geostoreDataSource.url=jdbc:h2:./webapps/mapstore/geostore
geostoreEntityManagerFactory.jpaPropertyMap[hibernate.hbm2ddl.auto
```

This configuration creates a file called `geostore` in the webapp folder. You can change the `geostoreDataSource.url` to set the path to the database you want to use. Make you sure that the user of the project that executes Tomcat has write permissions on the folder where you want to create the database.

PostgreSQL

All the following configurations will use `geostore` as password of the user `geostore`. Of course you can change it according to your needings.

Database Creation and Setup

To use postgreSQL DBMS as MapStore you have to create the "geostore" DB.

- Log in as user postgres
- Create the geostore DB:

```
createdb geostore
```

Create users and schemas:

```
psql geostore < 001_setup_db.sql
```

Here below the required part of the file `001_setup_db.sql`, available here (creation of test user and schema for `geostore_test` in the original file is not strictly required for MapStore)

> Write the password you prefer instead of 'geostore'

```
-- CREATE SCHEMA geostore (set the password you prefer)
CREATE user geostore LOGIN PASSWORD 'geostore' NOSUPERUSER
INHERIT NOCREATEDB NOCREATEROLE;

CREATE SCHEMA geostore;

GRANT USAGE ON SCHEMA geostore TO geostore ;
GRANT ALL ON SCHEMA geostore TO geostore ;

alter user geostore set search_path to geostore , public;
```

If you need to create the database schema manually (validate mode), you have also this script.

At the end, **make you sure that the user `geostore` has access to the database** from the address of MapStore application. You can give permission by editing pg_hba.conf

Connection to the Database

To configure MapStore to connect it to the new created database you have to edit your override file like below (change the connection parameters accordingly):

```
# Setup driver and dialect for PostgreSQL database
geostoreDataSource.driverClassName=org.postgresql.Driver
geostoreVendorAdapter.databasePlatform=org.hibernate.dialect.Postg

# Connection parameters
geostoreDataSource.url=jdbc:postgresql://localhost:5432/geostore
geostoreDataSource.username=geostore
geostoreDataSource.password=geostore
geostoreEntityManagerFactory.jpaPropertyMap[hibernate.default_sche

# Automatic create-update database mode
geostoreEntityManagerFactory.jpaPropertyMap[hibernate.hbm2ddl.auto

# Other options
geostoreVendorAdapter.generateDdl=true
geostoreVendorAdapter.showSql=false
```

Migrate an existing H2 database to PostgreSQL

If you used an H2 database during development, and you want to deploy the application in production, migrating the database to PostgreSQL is not that easy.

For this reason we have created a specific tool for this task, called **H2ToPgSQLExport** that is part of the GeoStore CLI.

More information on the migration tool is available in the GeoStore CLI documentation page.

Oracle

Database Creation and Setup

Create a database geostore, a schema called GEOSTORE and a user `geostore` that has write access to them.

Use this SQL script to create the DB schema.

Connection to the Database

To configure MapStore to connect to the new created database you have to edit your override file like reported below:

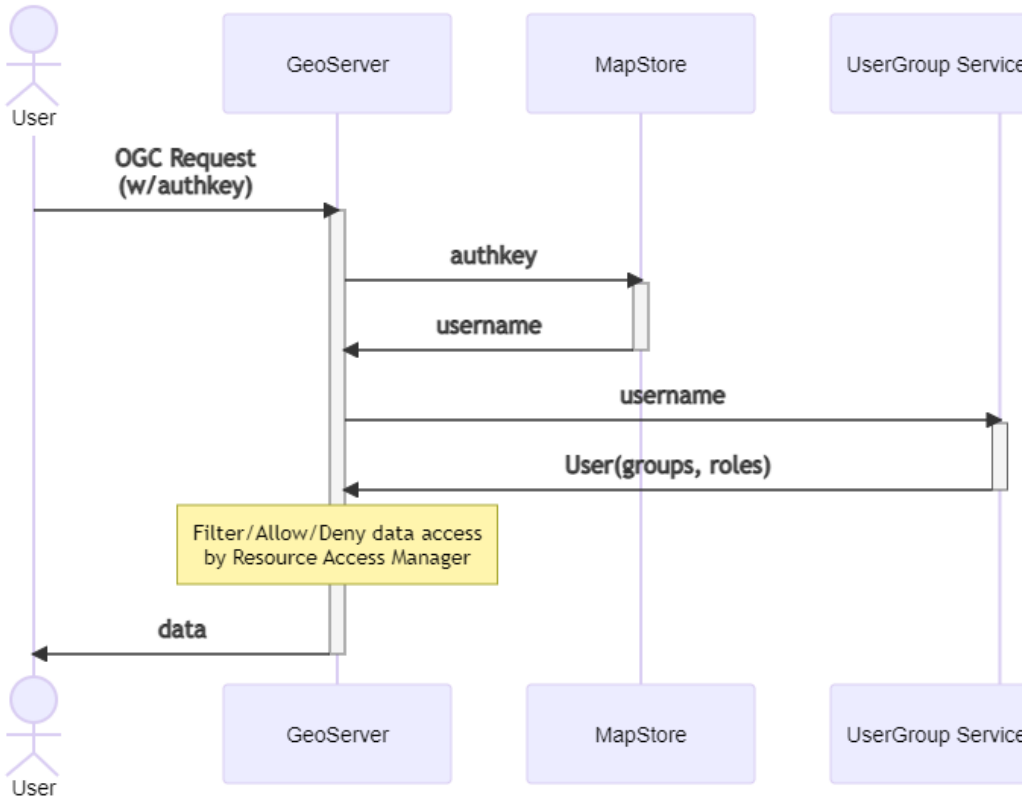
```
# Setup driver and dialect for Oracle Database
geostoreDataSource.driverClassName=oracle.jdbc.OracleDriver
geostoreVendorAdapter.databasePlatform=org.hibernate.dialect.Oracl

# Connection parameters
geostoreDataSource.url=jdbc:oracle:thin:@localhost:1521/ORCL
geostoreEntityManagerFactory.jpaPropertyMap[hibernate.default_sche
geostoreDataSource.username=geostore
geostoreDataSource.password=geostore

# Automatic create-update database mode
geostoreEntityManagerFactory.jpaPropertyMap[hibernate.hbm2ddl.auto

# Other options
geostoreVendorAdapter.generateDdl=true
geostoreVendorAdapter.showSql=false
```

GeoServer integrations

MapStore/GeoServer users integration

MapStore can share users, groups an roles with GeoServer. This type of integration allows to setup a fine grained access to the data and the services based on MapStore groups and roles.

This guide explains how to share users, groups and roles between MapStore and GeoServer. Applying this configurations will allow users logged in MapStore to be recognized by GeoServer. So security rules about restrictions on services, layers and so on can be correctly applied to MapStore users (also using GeoFence).



Note

UserGroup Service/Role Service can be *MapStore database* or *LDAP* depending on the setup you prefer.

With the suggested implementation the MapStore database will be also a UserGroupService and a RoleService for GeoServer. This means that every user of MapStore will be also a user in GeoServer, with the same attributes, the same roles (ADMIN, USER) and the same user groups.

For every user-group assigned to a user GeoServer will see also **a role** of the same name, from the role service, assigned to the members of the user-group (as user-group derived roles).

Permission on GeoServer can be assigned using these roles or with more detailed granularity using a custom Resource Access Manager (like GeoFence).

Limits of this solution

This solution partially degradates the functionalities of user management UI of GeoServer (for users, groups and roles that belong to MapStore). If you want to use this solution, you should use the MapStore's user manager and avoid the GeoServer's one.

Requirements

1. GeoServer must have the Authkey Plugin Community Module installed
2. MapStore2 Database must be reachable by GeoServer (H2 will not work, use PostgreSQL or Oracle)
3. MapStore2 must be reachable by GeoServer via HTTP

This example will focus on **PostgreSQL** database type I am assuming this is a new installation, so no existing user or map will be preserved

Database preparation

1. Follow Geostore wiki to setup a postgresql database (ignore the geostore_test part)

2. Start your Tomcat at least once, so `mapstore.war` will be extracted in the `webapps` directory of tomcat instance.
3. Stop Tomcat.
4. Copy from the extracted folder (`<TOMCAT_DIR>/webapps/mapstore`) the file located at `WEB-INF/classes/db-conf/postgres.properties` to replace the file `WEB-INF/classes/geostore-database-ovr.properties` .
5. Edit the new `WEB-INF/classes/geostore-database-ovr.properties` file with your DB URL and credentials.
6. Start Tomcat

Default user password couples are

- admin:admin
- user:user

GeoServer Setup

Follow this guide

Create the empty GeoStore database using scripts as described in GeoStore WIKI.

The following procedure will make GeoServer accessible to users stored in the MapStore database. In case of the users on MapStore and GeoServer have the same name, the users of MapStore will have precedence. At the end of the procedure, if you access with the user `admin` , you will have to use the password of the `admin` user of MapStore (`admin` by default).

User Groups and Roles

Steps below reference user, group and role service configuration files, as needed download the files from the geostore repository.

Setup User Group Service

- In GeoServer, Open the page "Security" --> "User Groups Roles" (from the left menu)
- In the section "User Group Services" click on "add new" to a new user group service
- Select `JDBC`
- name: `geostore`
- Password encryption : `Digest`
- password policy `default`
- Driver `org.postgresql.Driver` (or `JNDI`)
- connection url `jdbc:postgresql://localhost:5432/geostore` (or the one for your setup)
- JNDI only: the JNDI resource name should look like this `java:comp/env/jdbc/geostore`
- set username and password for the db (user `geostore` with password `geostore`)
- click on "Save" button
- Then, in order to adapt the standard JDBC service to MapStore database, you must place the provided files in the new directory (created by GeoServer for this new user group service) inside the data directory at the following path `<gs_datadir>/security/usergroup/geostore`. (`geostore` is the name of the new user group service)
- Then go back to `geostore` user group service page in GeoServer (the `ddl` and `dml` path should have values in them)
- click on "Save" button again

Setup Role Service

- In GeoServer Open the page "Security" --> "User Groups Roles" (from the left menu)
- In the section "Role Services" click on "add new" to a new role service

- select `JDBC`
- name `geostore`
- db `org.postgresql.Driver`
- connection url: `jdbc:postgresql://localhost:5432/geostore` (or JNDI, same as above)
- set user and password (user `geostore` with password `geostore`)
- click on "Save" button
- add the provided files to the geostore directory under `/<gs_datadir>/security/role/geostore`
- click on "Save" button again
- go Again in JDBC Role Service `geostore`
- select Administrator role to `ADMIN`
- select Group Administrator Role to `ADMIN`

Use these services as default

- In GeoServer "Security" --> "Settings" section (from the left menu)
- Set the **Active role service** to `geostore`
- go to Authentication Section, scroll to Authentication Providers and Add a new one.
- select 'Username Password'
- name it “geostore”
- select “geostore” from the select box
- Save.
- Then go to "Provider chain" and move geostore **on top** in the **right list**.
- Save again

Use the Auth key Module with GeoStore/GeoServer

These last steps are required to allow users logged in MapStore to be authenticated correctly by GeoServer.

Configure GeoServer

- Install the authkey module in GeoServer if needed (most recent versions of GeoServer already include it).
- Go to the authentication page and scroll into the 'Authentication Filters' section
- Click 'Add new'.
- Inside the 'New authentication Filter' page click on authkey module.
- Insert the name (i.e. 'geostore').
- Leave authkey as parameter name.
- Select the **Web Service** as 'Authentication key user mapper'.
- Select the created geostore's 'User/Group Service'.
- Input the mapstore2 url: `http://<your_hostname>:<mapstore2_port>/mapstore/rest/geostore/session/username/{key}` . Examples:

```
http://localhost:36728/mapstore/rest/geostore/session/username/
{key}
http://localhost/mapstore2/rest/geostore/session/username/{key}
http://mapstore.geosolutionsgroup.com/mapstore/rest/geostore/
session/username/{key}
```

- Save.
- Go into the authentication page and open default filter chain.
- Add 'geostore' into the 'Selected' filters and put it on top, and save.

> **Note**
>
> in the User Groups and Roles Services available options there are "AuthKEY WebService Body Response - UserGroup Service from WebService Response Body" and "AuthKEY REST - Role service from REST endpoint". Ignore them as they are not supported from MapStore2.

Configure MapStore

The last step is to configure MapStore to use the authkey with the configured instance of GeoServer. You can do it by adding to `localConfig.json` like this:

```
//...
"useAuthenticationRules": true,
  "authenticationRules": [{
    "urlPattern": ".*geostore.*",
    "method": "bearer"
  }, {
    "urlPattern": "\\/geoserver/.*",
    "authkeyParamName": "authkey",
    "method": "authkey"
  }],
//...
```

- Verify that "useAuthenticationRules" is set to `true`
- `authenticationRules` array should contain 2 rules:
  - The first rule should already be present, and defines the authentication method used internally in mapstore
  - The second rule (the one you need to add) should be added and defines how to authenticate to GeoServer:
    - `urlPattern` : is a regular expression that identifies the request url where to apply the rule
    - `method` : set it to `authkey` to use the authentication filter you just created in Geoserver.
    - `authkeyParamName` : is the name of the authkey parameter defined in GeoServer (set to `authkey` by default)

Advantages of user integration

Integrating the user/groups database with GeoServer you can allow some users to:

- Execute some processes (via WPS security)
- Download data (setting up the WPS download extension to allow/deny certain users to download data)
- Edit Styles (by default allowed only to administrators, but you can change it acting on `/rest/` Filter Chains).
- Access to layers based on users (using the standard GeoServer security)
- Filter layers data based on users (GeoFence), see here
- Allow editing of layers to certain MapStore users (GeoServer Security). The editing can be enabled in the plugin settings of MapStore

GeoServer Plugins and Extensions

MapStore supports several plugins for GeoServer. Installing them will expand the functionalities of MapStore, allowing to navigate data with time dimension, styling layers and so on.

Here a list of the extensions that MapStore can use:

- WMTS Multidimensional despite the name, this service provides multidimensional discovery services for GeoServer in general, not only for WMTS, and it is **required** to use the timeline plugin of MapStore.
- SLD Rest Service: This extension can be used by the MapStore styler to classify Vector and Raster data. It can inspect the real layer data to apply classification based on values contained in it. It allows to select various classification types (quantile, equalInterval, standardDeviation…) and to customize the color scales based on parameters
- CSS Extension: With this extension the MapStore styler allows to edit styles also in CSS format, in addition to the standard SLD format

- WPS Extension: Provides several process that can be executed using the OGC WPS Standard. IT contains some default services very useful for MapStore:
    - **gs:PagedUnique**: Provide a way to query layer attribute values with pagination and filtering by unique values. It enables autocomplete of attribute values for feature grid, attribute table, filter layer and other plugins.
    - **gs:Aggregate**: Allows aggregation operation on vector layers. This can be used by the charts (widgets, dashboards) to catch data
    - **gs:Bounds**: allows to calculate bounds of a filtered layer, used to dynamically zoom in dashboards map, when filtering is active.
- WPS download community module: This additional module allows to improve the default download plugin, based on WFS, with more functionalities. The advanced Download, activated when GeoServer provides the WPS service above, allows to
    - Download also the raster data
    - Schedule download processes in a download list (and download them later, when post processing is finished).
    - Select Spatial reference system
    - Crop dataset to current viewport
    - For vector layers:
        - Filter the dataset (based on MapStore filter)
    - For raster layers:
        - Select Compression type and quality
        - Define width and height of internal tiles
- CSW Extension: Activating this extension, MapStore can browse data of GeoServer using the CSW protocol. This is particularly useful when GeoServer contains hundreds or thousands of layers, so the WMS capabilities services can be too slow.
- Query Layer Plugin: This plugin allows the possibility to do cross-layer filtering. Cross layer filtering is the mechanism of Filtering a layer using

geometries coming from another layer. The plugin allows this filtering to be performed on the server side in an efficient way.

- DDS/BIL Plugin: this plugin add to geoserver the possibility to publish raster data in DDS/BIL format (World Wind). This particular plugin is useful if we want to use a raster data as elevation model for MapStore. This elevation model will be used in 3D mode or with the mouse coordinates plugin (displaying the elevation of a point on the map, together with the coordinates).

LDAP integration with MapStore

The purpose of this guide is to explain how to configure MapStore to use an LDAP repository for authentication and accounting (users, roles and user-groups) instead of the standard database.

Overview

By default the MapStore backend users service (also known as GeoStore), uses a relational database to store and fetch users details, implement authentication and assign resource access rights to users and groups (for maps, dashboards, etc.).

If you already have your users on an LDAP repository you can anyway configure MapStore to connect to your service and use it to authenticate users and associate user groups and roles, instead of using the default database. In this case the relational database will store only resources and accessory data (permissions, attributes ...) referring the users of your service.

Notice that the LDAP storage is read-only. This means that the MapStore User/Groups management UI cannot be used to manage users and groups. This makes sense because an LDAP repository is considered an external source and should be managed externally.

If this can create confusion, you can eventually fully disable the UI when using LDAP, by removing the corresponding plugin from the MapStore configuration.

The LDAP storage can be configured in two different ways:

- *synchronized* mode
- *direct connection* mode (experimental)

Synchronized mode

In *synchronized mode*, user data (users, groups, roles) is read from LDAP on every login and copied on the internal database.

Any other operation, for example getting user permissions on maps, always uses the internal database.

Synchronized mode is faster for normal use, but data may disalign when users are removed from the LDAP repository.

In general we suggest to use synchronized mode, since it is the most stable and tested one.

Direct connection mode (experimental)

In direct connection mode, user data is always read from LDAP, for any operation, so there is no risk of misaligned data.

Direct connection is still experimental and not tested in all the possible scenarios, but will hopefully become the standard mode in an early future, because the approach is simpler and avoids most the synchronized mode defects (e.g. misalignments).

Configuration

Configuring MapStore to use the LDAP storage requires:

- filling out the LDAP configuration properties in the java/web/src/main/resources/ldap.properties file to match your LDAP repository structure
- invoking the build with the **ldap** profile

```
./build.sh <version> ldap
```

Configuration properties

Configurable properties in the ldap.properties file include the following:

```
## name of the LDAP server host
ldap.host=localhost
## port of the LDAP server
ldap.port=10389
## root path for all searches
ldap.root=dc=acme,dc=org
## complete DN of an LDAP user, with browse permissions on the
used LDAP tree (optional, if browse is available to anoymous
users)
ldap.userDn=
## password of the userDn LDAP user (optional, if browse is
available to anoymous users)
ldap.password=
## root path for seaching users
ldap.userBase=ou=people
## root path for seaching groups
ldap.groupBase=ou=groups
## root path for seaching roles
ldap.roleBase=ou=groups
## LDAP filter used to search for a given username ({0} is the
username to search for)
ldap.userFilter=(uid={0})
## LDAP filter used to search for groups membership of a given
user ({0} is the full user DN)
ldap.groupFilter=(member={0})
## LDAP filter used to search for role membership of a given
user ({0} is the full user DN)
ldap.roleFilter=(member={0})

## enables / disables support for nested (hierarchical) groups;
when true, a user is assigned groups recursively if its groups
belong to other groups
ldap.hierachicalGroups=false
## LDAP filter used to search for groups membership of a given
group ({0} is the full group DN)
ldap.nestedGroupFilter=(member={0})
## max number of nested groups that are used
ldap.nestedGroupLevels=3

## if true, all the searches are recursive from the relative
root path
ldap.searchSubtree=true
```

```
## if true, all users, groups and roles names are transformed
to uppercase in MapStore
ldap.convertToUpperCase=true
```

Enabling direct connection mode

The default configuration enables the synchronized mode. To switch to direct connection mode you have to manually edit the final `geostore-spring-security.xml` to uncomment the related section at the end of the file:

```
<!-- enable direct connection mode -->
<bean id="ldapUserDAO" [...]>
        [...]
    </bean>
    <bean id="ldapUserGroupDAO" [...]>
        [...]
    </bean>
    <!-- -->
```

Testing LDAP support

If you don't have an LDAP repository at hand, a very light solution for testing is the acme-ldap java server included in the GeoServer LDAP documentation here.

You can easily customize the sample data tree, editing the java code.

The sample MapStore LDAP configuration in the default `ldap.properties` file works seamlessly with acme-ldap.

Advanced Configuration

More information about the MapStore backend storage and security service, GeoStore, is available here.

In particular, more information about LDAP usage with GeoStore is in the following Wiki page.

Integration with OpenID connect

MapStore allows to integrate and login using some common OpenID connect services. Having this support properly configured, you can make MapStore users able to login with the given OpenID service.

Customizing logo an text in Login Form

For details about the configuration for a specific service, please refer to the specific section below. For details about `authenticationProviders` optional values (e.g. to customize icon and/or text to show), refer to the documentation of the LoginPlugin.

By default `authenticationProviders` is `{"type": "basic", "provider": "geostore"}`, that represents the standard login on **MapStore** with username and password. With the default configuration, when the user try to login, MapStore will show the classic login form.

It is possible to add other providers to the list (e.g. `openid`) and they will be added as options to the login window. You can remove the `geostore` entry from `authenticationProviders` list to remove the login form from the possible login systems.

> **Note**
>
> If only one OpenID entry is present in `authenticationProviders` (and no `geostore` entry available), clicking on login in the login menu will not show any intermediate window and you will be redirected directly to the OpenID provider configured. If more than one entry is present in `authenticationProviders` list, the user will have to choose one of them to be authenticated.

Supported OpenID services

MapStore allows to integrate with the following OpenID providers.

- Google

- Keycloak

For each service you want to add you have to:

- properly configure the backend
- modify `localConfig.json` adding a proper entry to the `authenticationProviders`.

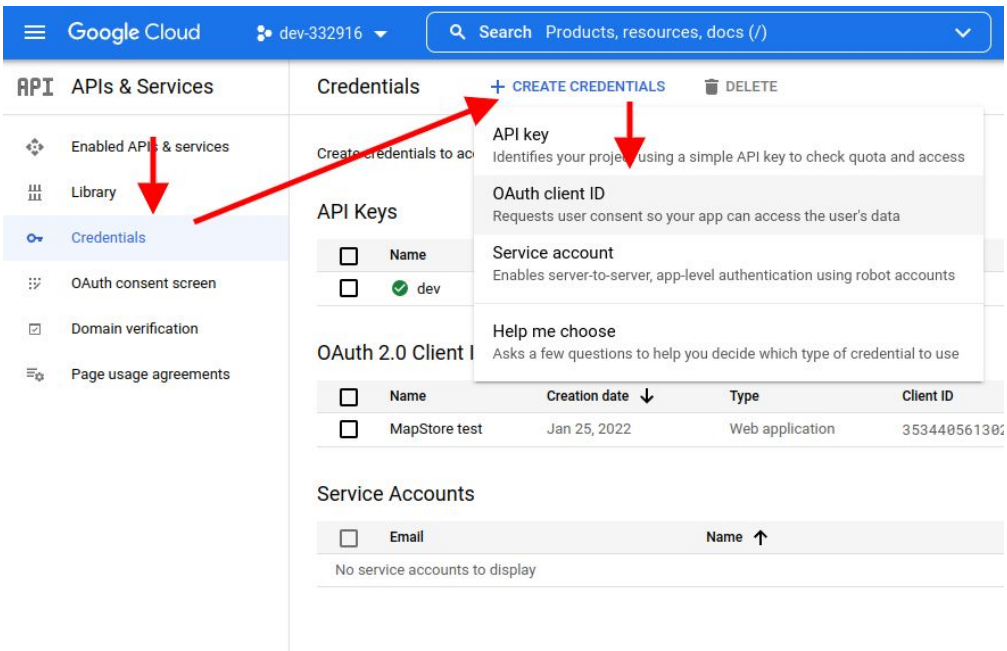
> **Note**
>
> For the moment we can configure only one authentication per service type (only one for google, only one for keycloak ...).

Google

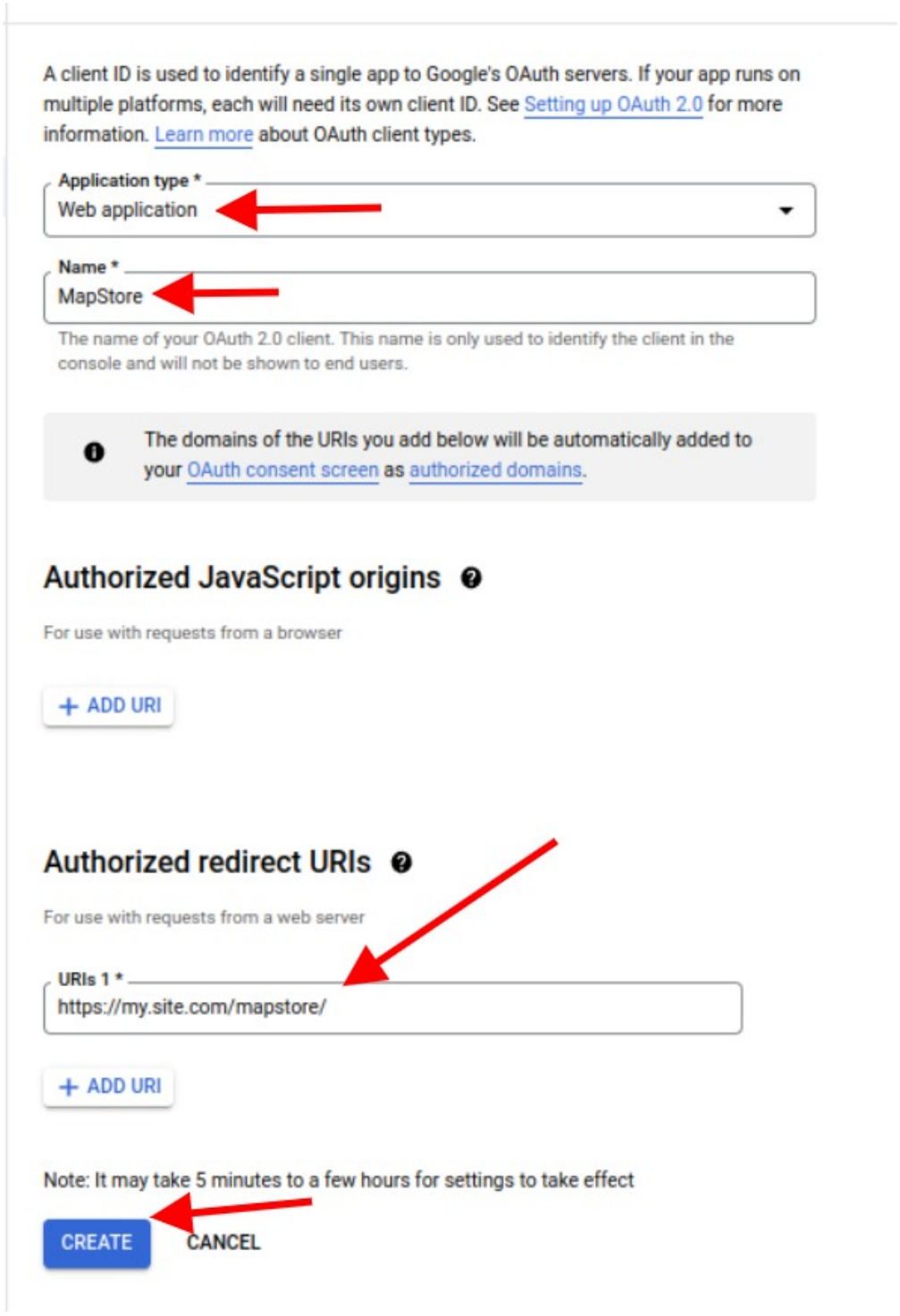
Create Oauth 2.0 credentials on Google Console

In order to setup the openID connection you have to setup a project in Google API Console to obtain Oauth 2.0 credentials and configure them.

- Open Google developer console and, from credentials section, create a new credential of type **Oauth client ID**



- Set the **Application Type** to **Web Application**, name it as you prefer and configure the root of the application as an authorized redirect URI. Then click on **Create**



- After creation you will obtain **ClientID** and **Client Secret** to use to configure MapStore.

Please follow the Google documentation for any detail or additional configuration.

Configure MapStore back-end for Google OpenID

After the setup, you will have to:

- create/edit `mapstore-ovr.properties` file (in data-dir or class path) to configure the google provider this way:

```
# enables the google OpenID Connect filter
googleOAuth2Config.enabled=true

#clientId and clientSecret
googleOAuth2Config.clientId=<the_client_id_from_google_dev_console
googleOAuth2Config.clientSecret=<the_client_secret_from_google_dev

# create the user if not present
googleOAuth2Config.autoCreateUser=true

# Redirect URL
googleOAuth2Config.redirectUri=https://<your-appliction-domain>/
mapstore/rest/geostore/openid/google/callback
# Internal redirect URI (you can set it to relative path like
this `../../..` to make this config work across domain)
googleOAuth2Config.internalRedirectUri=https://<your-appliction-
domain>/mapstore/

## discoveryUrl: contains all the information for the specific
service.
googleOAuth2Config.discoveryUrl=https://
accounts.google.com/.well-known/openid-configuration
```

Configure MapStore front-end for Google OpenID

- Add an entry for `google` in `authenticationProviders` inside `localConfig.json` file.

```
{
    "authenticationProviders": [
      {
        "type": "openID",
        "provider": "google"
      },
      {
        "type": "basic",
```

```
            "provider": "geostore"
        }
    ]
}
```

Keycloak

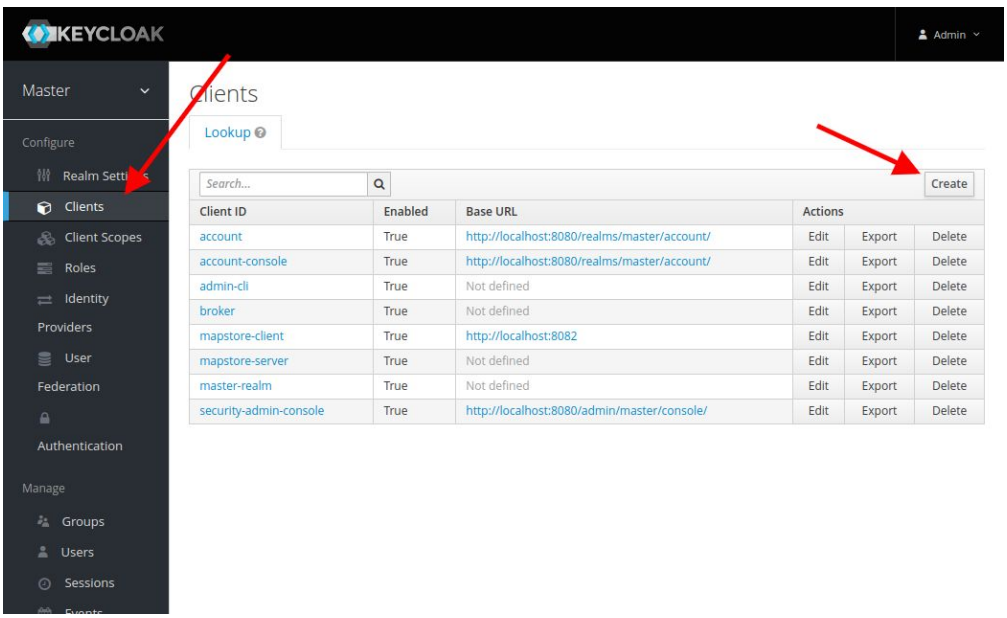
Keycloak is an open source identity and access management application widely used. MapStore has the ability to integrate with keycloak:

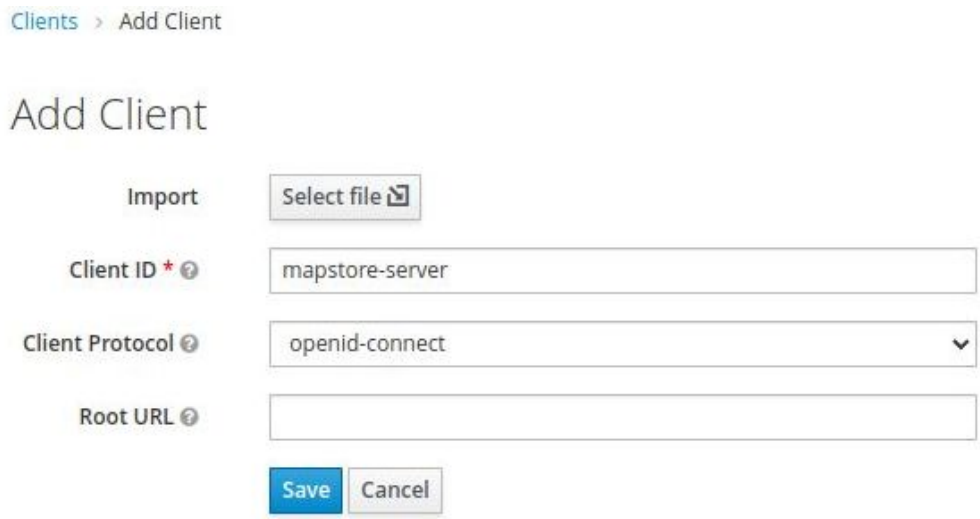
- Using the standard OpenID Connect (OIDC) protocol to login/logout in MapStore
- Supporting Single Sign On (SSO) with other applications.
- Mapping keycloak roles to MapStore groups, as well as for ldap.

In this section you can see how to configure keycloak as a standard OpenID provider. For other advanced functionalities, you can see the dedicated section of the documentation

Configure keycloak Client

Create a new Client on keycloak. In this guide we will name it `mapstore-server` (because if you need to configure SSO, we may need another key to call `mapstore-client`)





- Configure it as `Confidential` setting the Redirect-URL with your MapStore base root, with a `*` at the end (e.g. `https://my.mapstore.site.com/mapstore/*`)

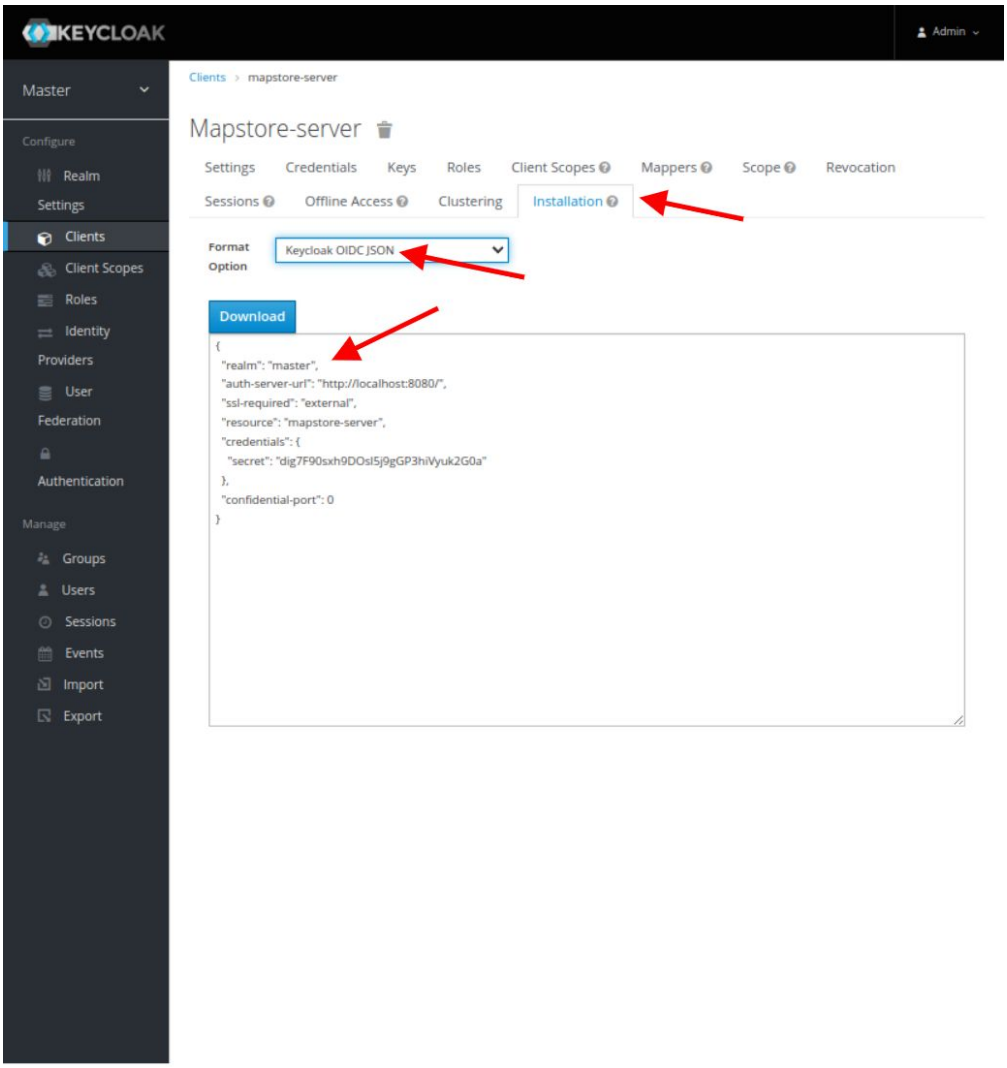
Clients › mapstore-server

Mapstore-server

Settings | Keys | Roles | Client Scopes | Mappers | Scope | Offline Access | Installation

Client ID	mapstore-server
Name	
Description	
Enabled	ON
Always Display in Console	OFF
Consent Required	OFF
Login Theme	
Client Protocol	openid-connect
Access Type	confidential
Standard Flow Enabled	ON
Implicit Flow Enabled	OFF
Direct Access Grants Enabled	ON
Service Accounts Enabled	OFF
OAuth 2.0 Device Authorization Grant Enabled	OFF
OIDC CIBA Grant Enabled	OFF
Authorization Enabled	OFF
Front Channel	OFF

- Click on Save button, then open the *Installation* tab, select the `Keycloak OIDC JSON` format, and copy the JSON displayed below.



Configure MapStore back-end for Keycloak OpenID

Create/edit `mapstore-ovr.properties` file (in data-dir or class path) to configure the keycloak provider this way:

```
# enables the keycloak OpenID Connect filter
keycloakOAuth2Config.enabled=false

# Configuration
keycloakOAuth2Config.jsonConfig=<copy-here-the-json-config-from-keycloak-removing-all-the-spaces>
```

```
# Redirect URLs
# - Redirect URL: need to be configured to point to your
application at the path <base-app-url>/rest/geostore/openid/
keycloak/callback
# e.g. `https://my.mapstore.site.com/mapstore/mapstore/rest/
geostore/openid/keycloak/callback`
keycloakOAuth2Config.redirectUri=https://my.mapstore.site.com/
mapstore/rest/geostore/openid/keycloak/callback
# - Internal redirect URL when logged in (typically the home
page of MapStore, can be relative)
keycloakOAuth2Config.internalRedirectUri=https://
my.mapstore.site.com/mapstore/

# Create user (if you are using local database, this should be
set to true)
keycloakOAuth2Config.autoCreateUser=true

# Comma separated list of <keycloak-role>:<geostore-role>
keycloakOAuth2Config.roleMappings=admin:ADMIN,user:USER

# Comma separated list of <keycloak-role>:<geostore-group>
keycloakOAuth2Config.roleMappings=MY_KEYCLOAK_ROLE:MY_MAPSTORE_GRO

# Default role, when no mapping has matched
keycloakOAuth2Config.authenticatedDefaultRole=USER
```

- `keycloakOAuth2Config.jsonConfig` : insert the JSON copied, removing all the spaces
- `keycloakOAuth2Config.redirectUri` : need to be configured to point to your application at the path `<base-app-url>/rest/geostore/openid/keycloak/callback` , e.g. `https://my.mapstore.site.com/mapstore/rest/geostore/openid/keycloak/callback`
- `keycloakOAuth2Config.internalRedirectUri` can be set to your application root, e.g. `https://my.mapstore.site.com/mapstore/`
- `keycloakOAuth2Config.autoCreateUser` : true if you want MapStore to insert a Keycloak authenticated user on the DB. UserGroups will be inserted as well and kept in synch with the roles defined for the user in Keycloak. The option **must be set to false if MapStore is using a read-only external service for users and groups** (i.e. Keycloak or LDAP).

- `keycloakOAuth2Config.forceConfiguredRedirectURI` : optional, if `true` , forces the redirect URI for callback to be equal to teh redirect URI. This is useful if you have problems logging in behind a proxy, or in dev mode.
- `keycloakOAuth2Config.roleMappings` : comma separated list of mappings with the following format `keycloak_admin_role:ADMIN,keycloak_user_role:USER` . These mappings will be used to map Keycloak roles to MapStore roles. Allowed values `USER` or `ADMIN` .
- `keycloakOAuth2Config.authenticatedDefaultRole` : where the role has not been assigned by the mappings above, the role here will be used. Allowed values `USER` or `ADMIN` .
- `keycloakOAuth2Config.groupMappings` : comma separated list of mappings with the following format `keycloak_role_name:mapstore_group_name,keycloak_role_name2:mapstore_` These mappings will be used to map Keycloak roles to MapStore groups.
- `keycloakOAuth2Config.dropUnmapped` : when set to false, MapStore will drop Keycloak roles that are not matched by any mapping role and group mapping. When set to true all the unmatched Keycloak roles will be added as MapStore UserGroups.

Configure MapStore front-end for Keycloak OpenID

- Add an entry for `keycloak` in `authenticationProviders` inside `localConfig.json` file.

```
{
    "authenticationProviders": [
      {
        "type": "openID",
        "provider": "keycloak"
      },
      {
        "type": "basic",
        "provider": "geostore"
      }
    ]
}
```

Keycloak Integrations

General

MapStore supports various Keycloak integration features:

- **OpenID support**: Allows to login to MapStore using a keycloak account.
- **Single sign on**: Enhances the OpenID support by detecting a session in the keycloak realm and automatically login/logout from MapStore
- **Direct user integration**: Enhances the OpenID support making MapStore use keycloak as unique Identity Manager System (IdM), replacing the MapStore DB with Keycloak REST API.

OpenID

Keycloak OpenID support allows to use a keycloak instance as Identity Provider (IdP) via OpenID Connect (OIDC), so that the user can login to MapStore using an existing account in keycloak.

You can find details about how to configure it in the dedicated **"OpenID Connect" page section dedicated to keycloak**

Single sign on integration

MapStore provides an integration with the keycloak **Single Sign On** (SSO) system, that allows to **automatically login/logout** in MapStore when you login/ logout from another application in the same keycloak realm, an vice-versa.

In order to enable the SSO in keycloak you have to:

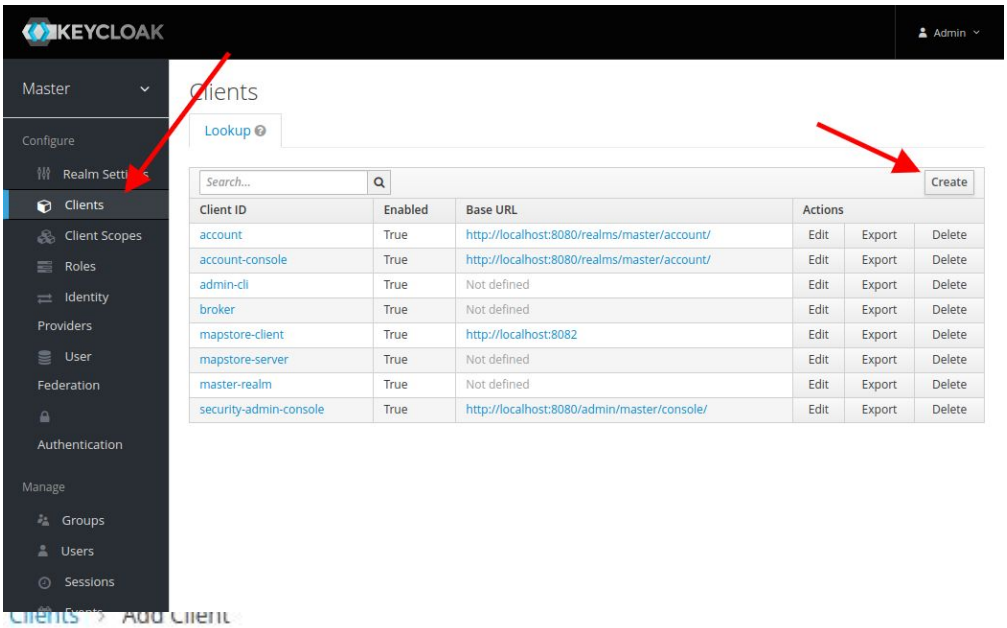
- Have already configured the openID for keycloak.
- Create a keycloak client in the same realm of openID integration above.
- Configure SSO in MapStore's `localConfig.json`

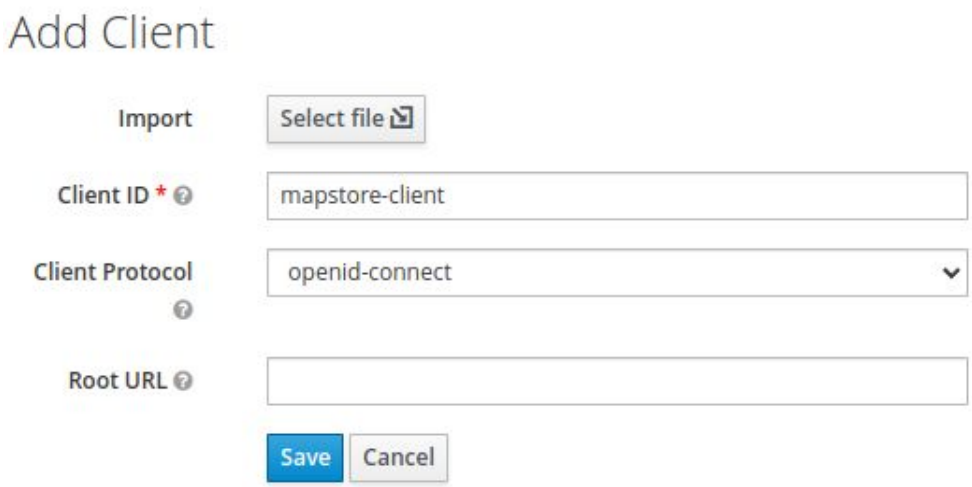
Configure the OpenID integration

- See here openID integration.

Configure keycloak client

After configuring the open openID integration, you will have a keycloak client called `mapstore-server`. In order to enable SSO you have to create **another** new Client on keycloak. In this guide we will name it `mapstore-client`.





- Configure it as `Public`

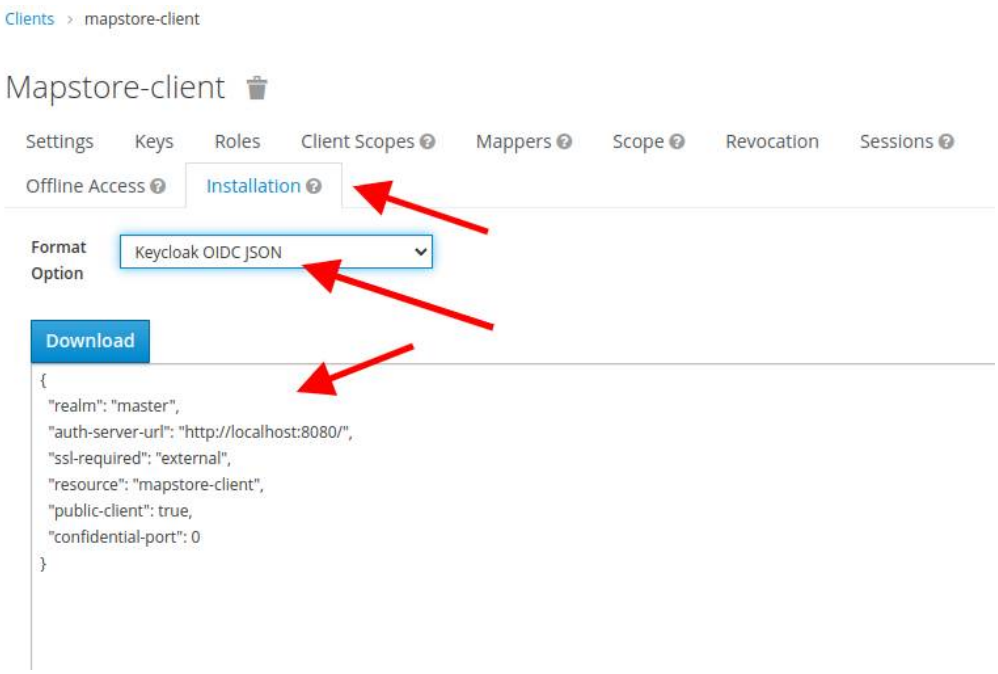
- Insert in "**Valid Redirect URIs**" your MapStore base root, with a `*` at the end (e.g. `https://my.mapstore.site.com/mapstore/*`)
- Insert in "**Web Origins**" your MapStore base domain name. (e.g. `https://my.mapstore.site.com`)

Mapstore-client

Settings | Keys | Roles | Client Scopes | Mappers | Scope | Offline Access | Installation

Client ID	mapstore-client
Name	
Description	
Enabled	ON
Always Display in Console	OFF
Consent Required	OFF
Login Theme	
Client Protocol	openid-connect
Access Type	public
Standard Flow Enabled	ON
Implicit Flow Enabled	ON
Direct Access Grants Enabled	ON
OAuth 2.0 Device Authorization Grant Enabled	OFF
Front Channel Logout	OFF
Root URL	
* Valid Redirect URIs	http://localhost:8081*
	http://localhost:8082/*
Base URL	
Admin URL	
Logo URL	

- Click on Save button, then open the *Installation* tab, select the `Keycloak OIDC JSON` format, and copy the JSON displayed below.



Configure SSO in MapStore

After configuring the open openID integration, you will have an entry named `keycloak` in `authenticationProviders`. In this entry, you will have to add `"sso":{"type":"keycloak"}` and `config: "<configuration coped from keycloak>"`.

e.g.

```
{
       "authenticationProviders": [
      {
        "type": "openID",
        "provider": "keycloak",
        "config": {
          "realm": "master",
          "auth-server-url": "http://localhost:8080/",
          "ssl-required": "external",
          "resource": "mapstore-client",
          "public-client": true,
          "confidential-port": 0
        },
        "sso": {
```

```
            "type": "keycloak"
          }
        }
      ],
  }
```

Here implementation details about keycloak login workflow.

Direct user integration

By default MapStore can integrate openID login with Keycloak and also supports integration with Keycloak SSO.

By default users that login with Keycloak are created on the database and their Keycloak roles inserted as MapStore UserGroup. Anyway MapStore can interact with Keycloak REST API to provide a direct integration without persisting anything on the MapStore's database. This provides a stricter integration between the applications, allowing the assignment of roles and groups directly from keycloak, and avoiding any synchronization issue.

In this scenario the integration MapStore replaces the user and user-group database tables with the keycloak REST API.

> **Note**
>
> This integration disables reading and writing to the users' and groups' database and replaces it with the Keycloak REST API, with read-only support. For this reason we suggest to disable the `UserManager`, `GroupManager` plugins, and remove the `authenticationProviders` entry of type `geostore`, if any, because the standard login with username and password is not allowed for the db users. In case of integration with GeoServer, also GeoServer should be connected to Keycloak for users, and not to the MapStore database.

Configure direct integration with keycloak

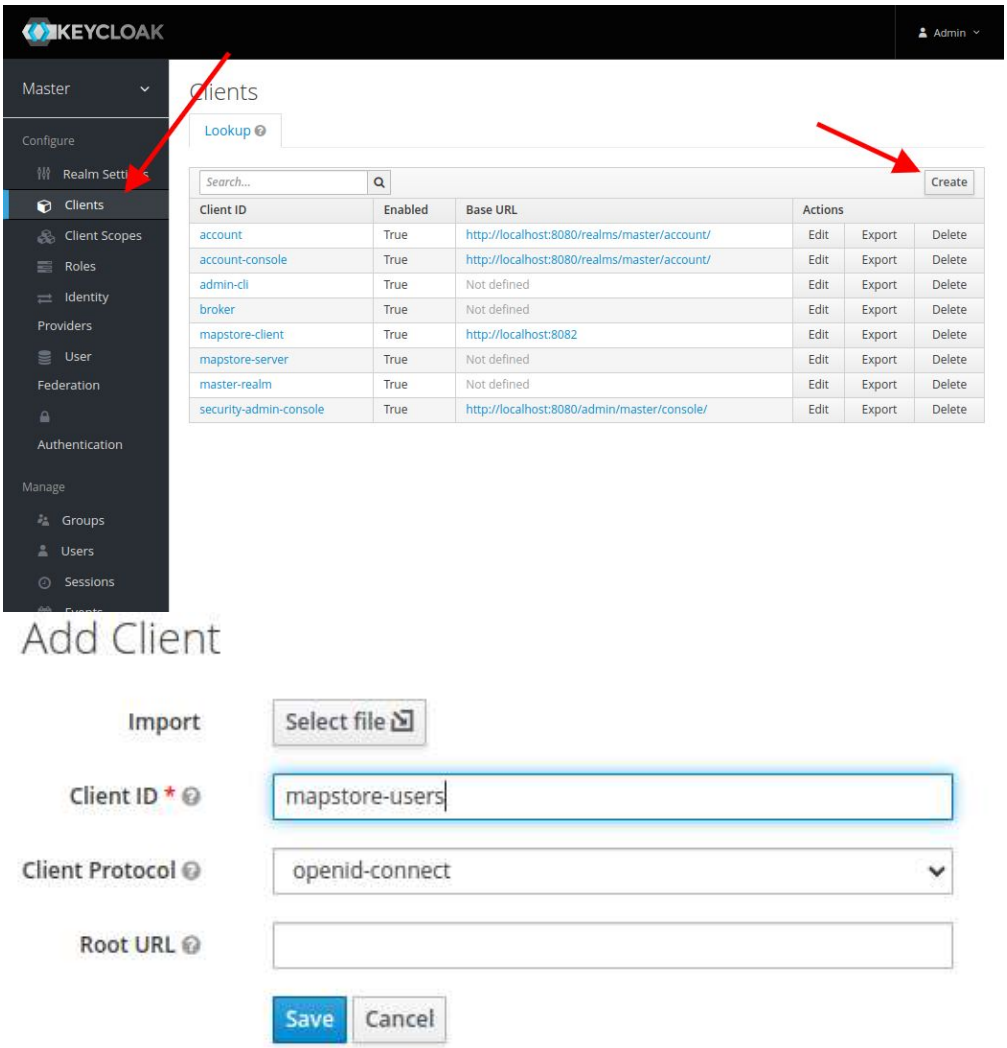
To enable the direct integration with keycloak you will have to:

1. Create a dedicated client for keycloak.
2. Configure `mapstore-ovr.properties`

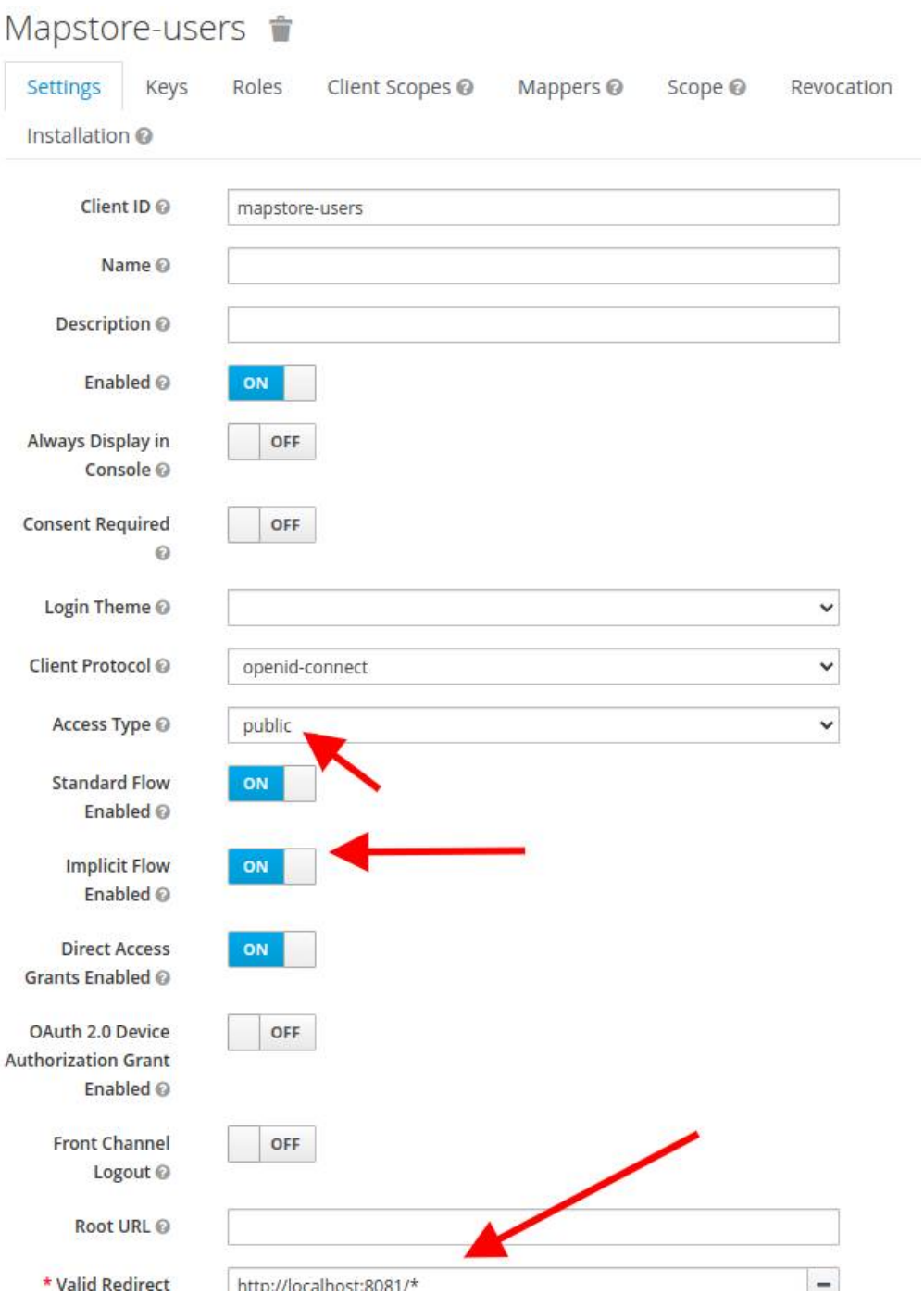
3. Activate the functionality via system property

1. Create a dedicated client for keycloak

- Create **another** client on keycloak, in the same realm of `mapstore-server` and `mapstore-client` (where present) called `mapstore-users`:



- Configure it with:
- **Access Type**: `public`
- **Implicit Flow Enabled** Set to on **On**
- **Valid Redirect URIs** with your app base URL, with an ending `*`, e.g. `http://localhost:8080/*`.



And click on Save.

2. Configure `mapstore-ovr.properties`

The `autoCreateUser` option must be set to false in `mapstore-ovr.properties`.

```
keycloakOAuth2Config.autoCreateUser=false
```

Moreover in `mapstore-ovr.properties` you have to add the following information (replacing `<keycloak-base-url>` with your base keycloak base url):

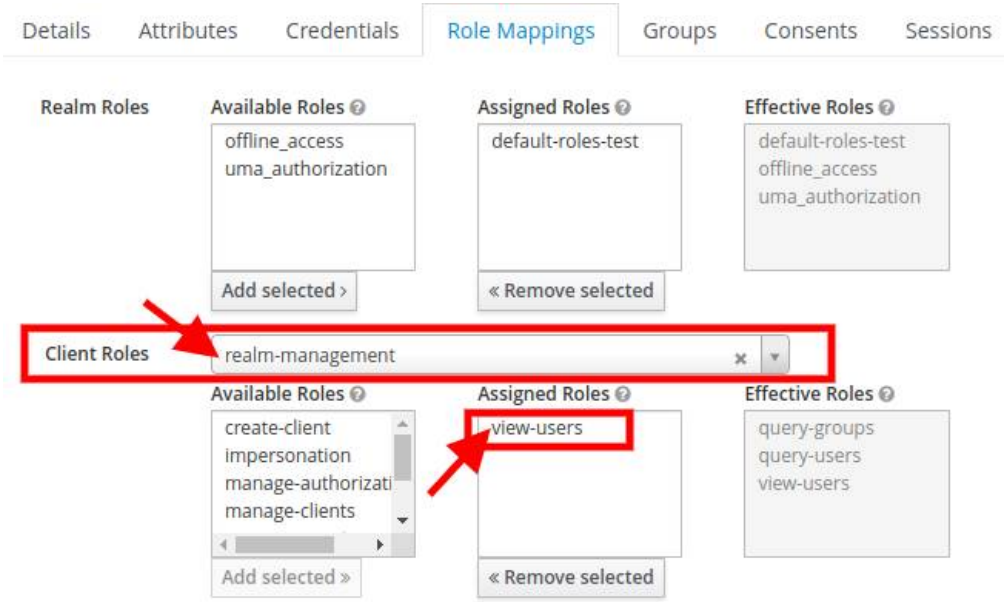
```
## Keycloak as User and UserGroup repository
keycloakRESTClient.serverUrl=<keycloak-base-url>
keycloakRESTClient.realm=master
keycloakRESTClient.username=admin
keycloakRESTClient.password=admin
keycloakRESTClient.clientId=mapstore-users
```

Where:

- `serverUrl` : URL of keycloak, (e.g. `http://localhost:8080` or `https://mysite.com/`)
- `realm` : the realm where the client has been created
- `username` , `password` : credentials of a user with the role to `view-users` .[1]

Note

[1] In order to query the keycloak REST API, you need to have in your realm at least one user with `realm-admin` role permission. Usually the administrator of the realm has these permission. To associate these permissions to a new user dedicated to this purpose, you have to open "Role Mappings" tab of keycloak and in "Client Roles" select `realm-management` (or in master realm select `master-realm`) and add to selected `realm-admin` .



3. Activate the functionality via system property

In order to activate the integration in your instance, you will need to set the Java System Property `security.integration` with the value `keycloak-direct`.

One easy and usual way to configure this system property in Tomcat is using the `JAVA_OPTS`. Like you do with `datadir.location`, you can set it by adding to `JAVA_OPTS` variable the entry `-Dsecurity.integration=keycloak-direct`.

> **Note**
>
> For old projects or in case you can not set the system property, you can anyway configure it by adding this section to your `geostore-spring-security.xml` file.

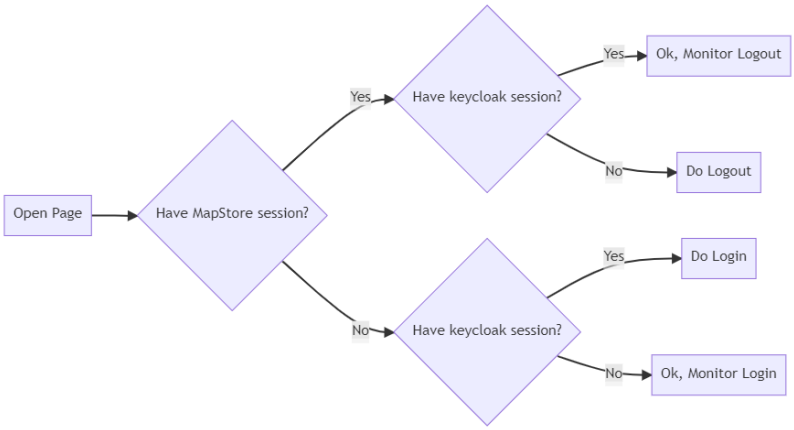
```
    <bean
        id="keycloakUserGroupDAO"

class="it.geosolutions.geostore.services.rest.security.keycloak.KeycloakUserGr
        <constructor-arg ref="keycloakRESTClient"/>
        <property name="addEveryOneGroup" value="true"/>
    </bean>
    <alias name="keycloakUserDAO" alias="userDAO"/>
    <bean
        id="keycloakUserGroupDAO"

class="it.geosolutions.geostore.services.rest.security.keycloak.KeycloakUserGr
        <constructor-arg ref="keycloakRESTClient"/>
    </bean>
    <alias name="keycloakUserGroupDAO" alias="userGroupDAO"
addEveryOneGroup="true"/>
    <alias name="externalSecurityDAO" alias="securityDAO"/>
```

SSO Workflow in Keycloak

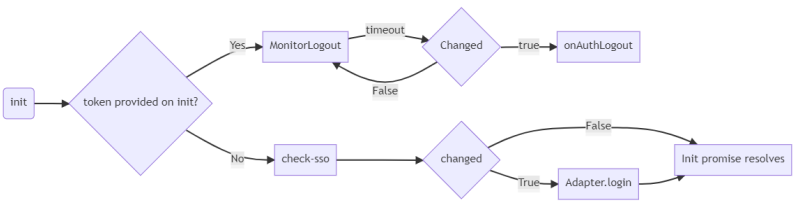
Here in this section some details about keycloak SSO integration

Desired workflow

If keycloak SSO is configured, we want to implement the following workflow.



The keycloakJS library implements the following workflow:



MapStore can:

- Re-run `init`
- Intercept `onAuthLogout`
- Implement adapter methods `login`, `logout`.
- Intercept `init` promise resolve with `.then`

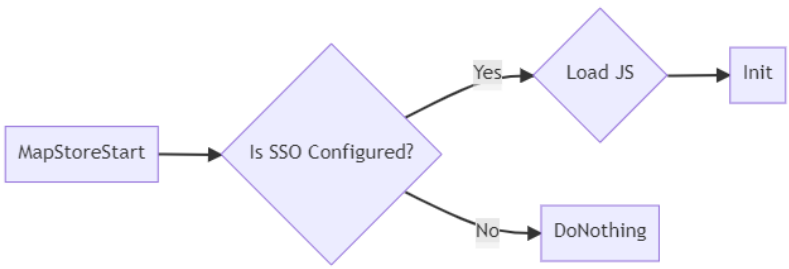
> **Note**
>
> `changed` is the variable emitted by an internal iframe managed by the keycloak JS API. This technique allows to intercept logout events, anyway refreshing tokens or intercepting login, after first attempt doesn't seem to work well and has some limitations because of security reasons. In particular in the current implementation with `openID` sync with GeoStore we need to workaround partially the logic of the library to make the tokens work in sync.

Implementation

The SSO integration in MapStore will reuse the entry points of the JS lib together with the existing openID integration in keycloak, implementing the following workflows:

Initialization

At the initial page load, we check if the `authenticationProviders` contains a `sso` entry (only keycloak)



- `LoadJS` : loads `keycloak.js` , that includes the JS support to keycloak, from keycloak instance (only once)
- `Init` is initialized by MapStore with the current config, adding MapStore's `access_token` and `refresh_token` , if present, from openID login.

Monitoring phase

After initialization, we may receive different events or cases. These are the possible cases:

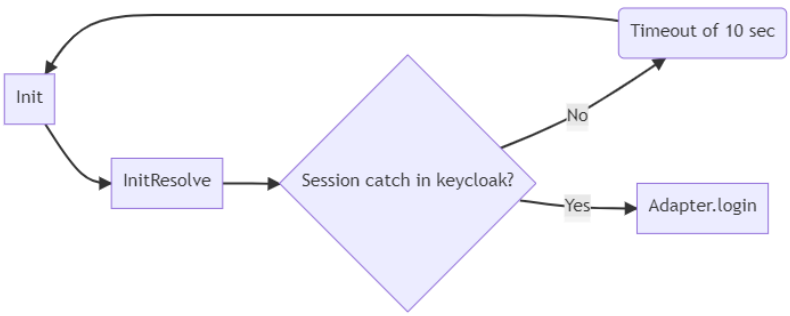
Case 1 - Login From MapStore

If MapStore is not logged in, the user can click on login button and be redirected to keycloak login form. After that, the init flow will pass the MapStore tokens to the JS interface. They will be used to check session logout.

> **Note**
>
> If MapStore user is logged in, the `init`, we may not initially have the token ready. For this reason, on LOGIN_SUCCESS, we re-init the application, or sync operation is triggered from `Adapter.login` to refresh the tokens.

Case 2 - Login from keycloak

If MapStore is not logged in, the `init` function do a `check-sso` operation and finish. In order to monitor the login on MapStore, we implemented a timer to re-init trigger anytime the `check-sso` resolves with not authenticated.



> **Note**

Implementation is using `messageReceiveTimeout` as timeout, the same timeout variable of the keycloak JS library for monitoring logout

Case 3 - Logout from keycloak

In this case the library that receives a valid keycloak token monitors the logout autonomously.

Case 4 - Logout from MapStore

Logout from MapStore, a bug in keycloak API doesn't correctly check the internal iframe (`changed` option event), and there is no possibility to trigger it, until you visit the keycloak page. This condition after logout can not be distinguished from a external login (from keycloak) detection. So refreshing the page before the token on client is naturally expired will cause a redirect to Login page, because MapStore find there is an active session on keycloak. In order to avoid this, an hack is necessary. MapStore loads an iframe immediately after logout to allow the cookie session to be catch and to apply the proper reset.

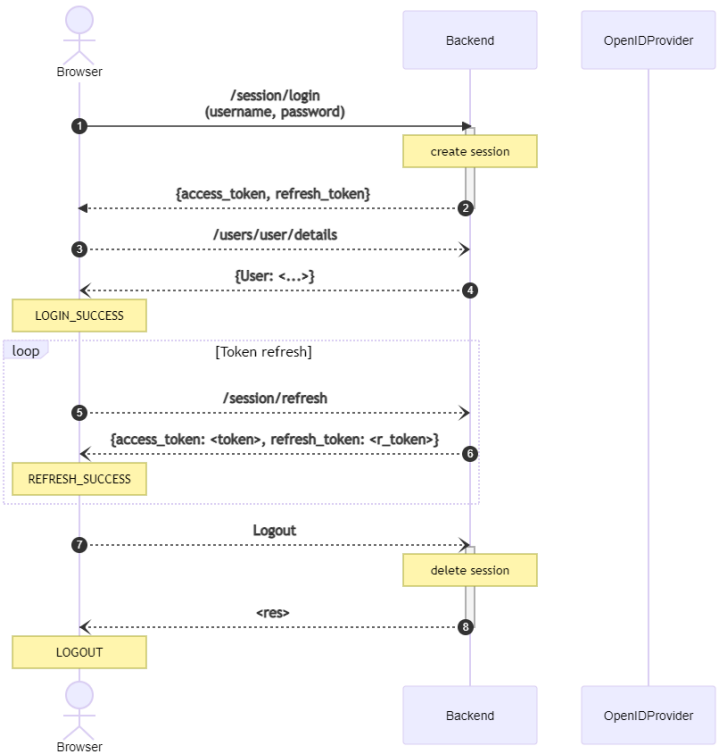
Refresh token

By default keycloak has 5 minutes long lifetime for token, 30 minutes for refresh token. Anyway this can be configured. For this reason, the keycloak support schedules a refresh based on the current token expiration, restarting from `init`, scheduling a refresh as half of time between expiring time and now. (e.g The token expires 2 minutes from now, a refresh is scheduled in 1 minute).

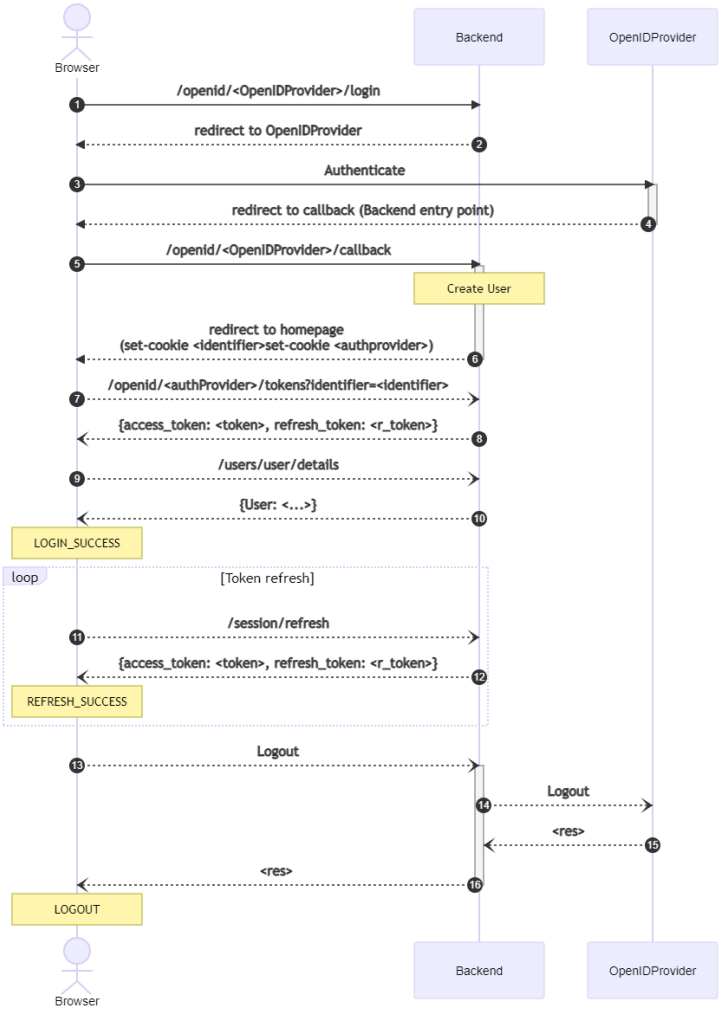
MapStore Authentication - Implementation Details

In this section you can see the implementation details about the login / logout workflow implemented by MapStore.

Standard MapStore login



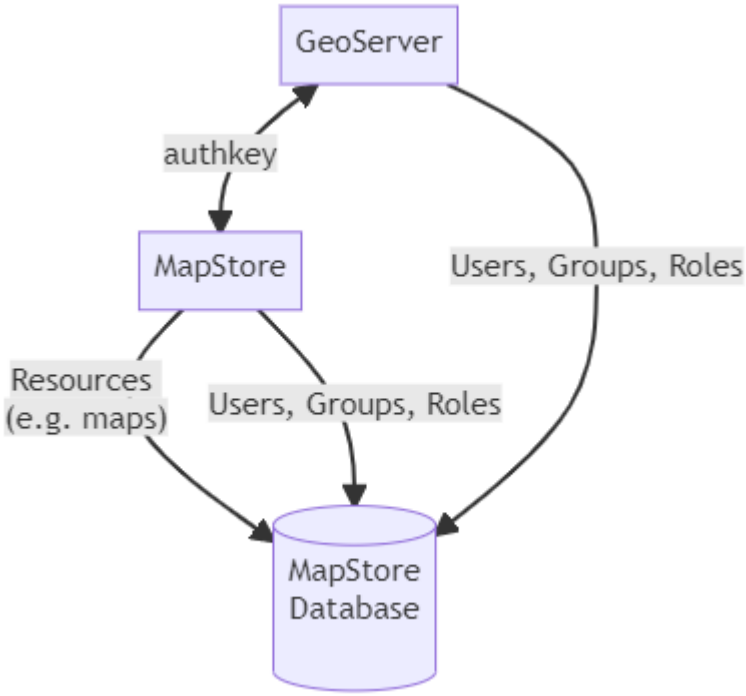
OpenID MapStore Login



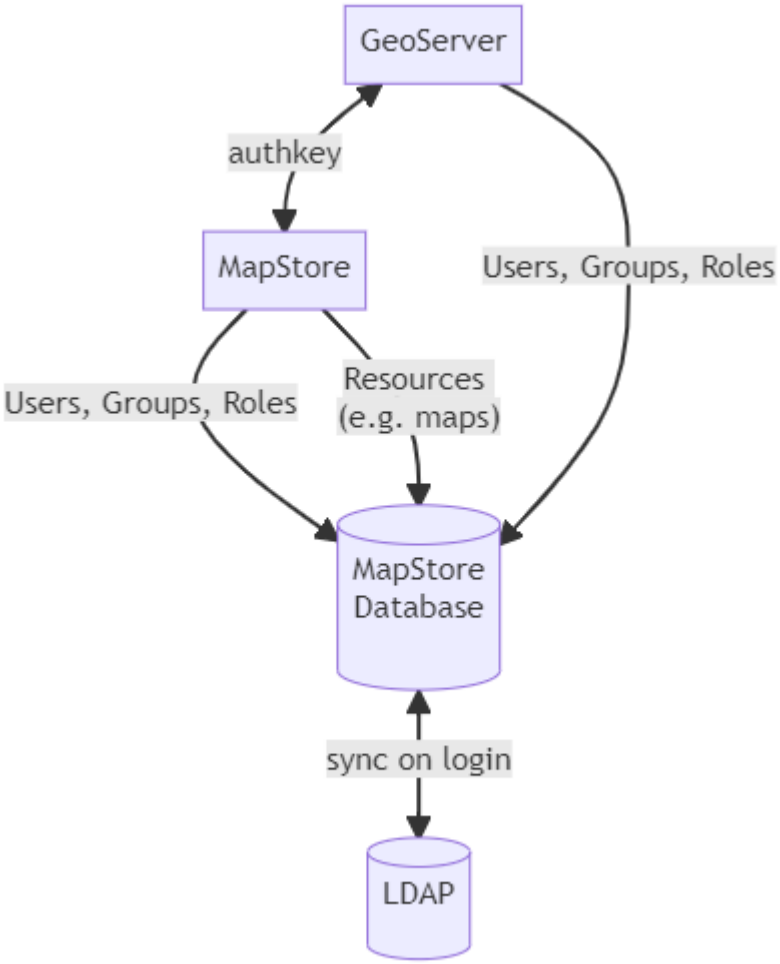
Possible setups

Accordingly with your infrastructure, there are several setups you can imagine with MapStore and GeoServer.

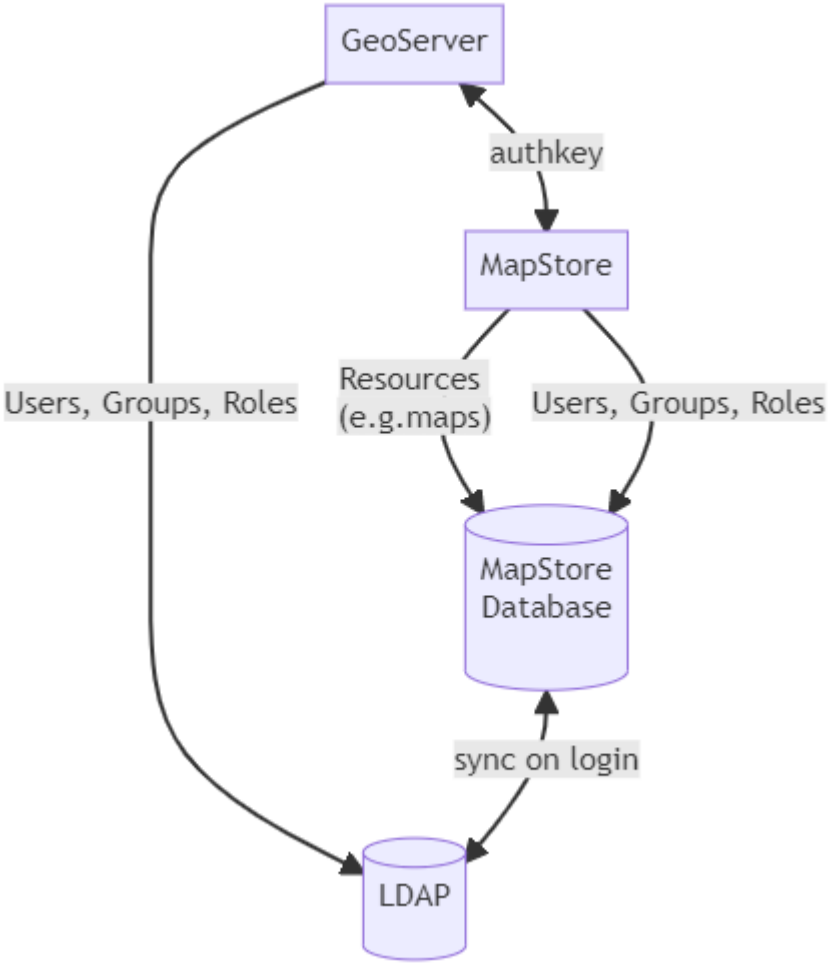
MapStore-GeoServer integration



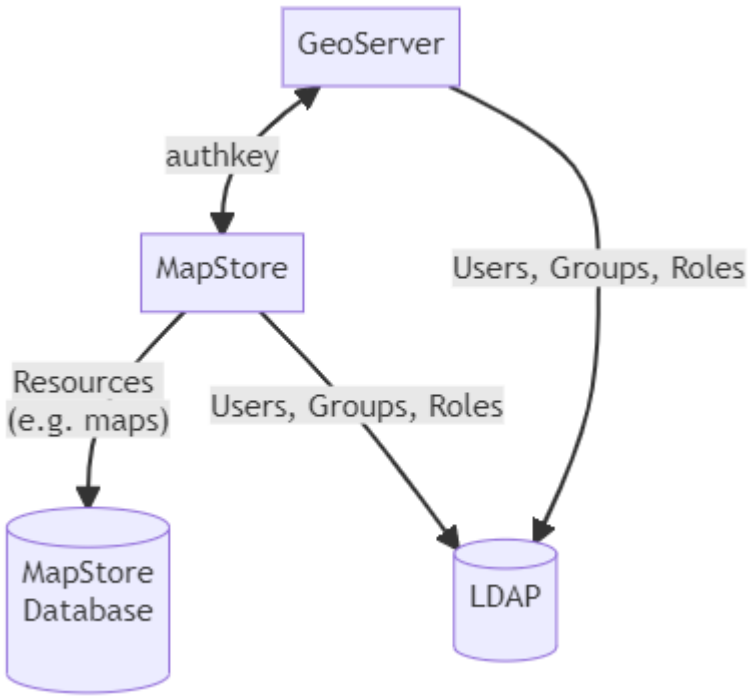
MapStore-LDAP + MapStore-GeoServer



MapStore-GeoServer + MapStore-LDAP + GeoServer-LDAP



MapStore-GeoServer + MapStore-LDAP (direct) + GeoServer-LDAP



MapStore Projects

MapStore projects can be created using the Project Creation Script.

A MapStore project is a custom WebGis application that uses MapStore as a framework.

The MapStore framework is linked as a git submodule in the MapStore2 project subfolder.

> **Note**
>
> Since MapStore is linked as a submodule, every project custom file should be created outside of it. This allows updating MapStore to a newer version easily, without conflicts. The general rule is: **never add / update / modify files directly in the MapStore2 subfolder**.

Standard Projects

A **Standard** MapStore project is a project that is, initially, a perfect copy of the standard MapStore product.

To create custom application using the standard projects template, you will start from **js/app.jsx** that is the project entry point.

Editing **app.jsx** you can start using your own configuration files and add custom behaviours and look and feel to your project, in particular:

- You can **add your own translation files**. Setting an array of paths in the `translationsPath` , the resources will be loaded in cascade from every directory of the array. So you can keep all the original translations from MapStore (first element of the array) and add your own files in the directory `translations` , overriding original values of the json or adding new ones (for instance, for your custom plugins). The files in the new directory must follow the same naming convention of the files in the oridinal directory.

```
ConfigUtils.setConfigProp('translationsPath', ['./MapStore2/web/
client/translations', './translations']);
```

- **Use your own configuration file** for plugins and other configurations. You can copy the original `localConfig.json` in the root of the project and configure the application to load it (instead of the default one, located in `MapStore2/web/client/localConfig.json`).

```
ConfigUtils.setLocalConfigurationFile('localConfig.json');
```

or you can apply some patch files defining an array of configurations, where the first is the main json file, and the rest are the patch files which must end with "patch.json" in the filename

```
ConfigUtils.setLocalConfigurationFile(['localConfig.json',
'production.patch.json']);
```

the patch will be applied using this package

- Configure your own pages:

```
const appConfig = assign({}, require('../MapStore2/web/client/
product/appConfig'), {
     pages: [{
         name: "mapviewer",
         path: "/",
         component: require('../MapStore2/web/client/product/
pages/MapViewer')
     }]
});
```

- Include the plugins you want in the app (either MapStore plugins or your own):

```
const plugins = require('./plugins');
```

Organizing your code

Our convention is to use the **js** folder to store your project code. You should recreate inside it the usual folders to organize your code based on the source code type:

- components
- actions
- reducers
- epics
- plugins

Images and other static assets should be located in the **assets** folder instead.

Create your own MapStore project

> **Note**
>
> From version 2021.02.xx MapStore introduced a new project system. Take a look here to learn more about the new project system.

To create a new MapStore based project you can use the createProject script. First of all, if you don't have done it before, clone the MapStore2 repository master branch into a local folder:

```
git clone https://github.com/geosolutions-it/MapStore2
```

Then, move into the folder that has just been created, containing MapStore2:

```
cd MapStore2
```

Choose from which branch you want the mapstore revision to be aligned, we suggest to use latest release or latest stable available (if you know which is)

```
git checkout <stable-branch>
```

or

```
git checkout v2022.01.02
```

Install dependencies for MapStore:

```
npm install
```

Finally, to create the project, use the following command:

```
node ./createProject.js <projectType> <projectName>
<projectVersion> <projectDescription> <gitRepositoryUrl>
<outputFolder>
```

Note that projectName and outputFolder are mandatory:

- **projectName**: short project name that will be used as the repository name on github, webapp path and name in package.json
- **projectType**: type of project to create, currently one type of projects is supported:
- **standard**: is a copy of the standard MapStore project, ready to be used and customized
- **projectVersion**: project version in package.json (X.Y.Z)
- **projectDescription**: project description, used in sample index page and as description in package.json
- **gitRepositoryUrl**: full url to the github repository where the project will be published
- **outputFolder**: folder where the project will be created

Usage:

```
node ./createProject.js standard MyProject "1.0.0" "this is my
awesome project" "" ../MY_PROJECT_NAME
```

At the end of the script execution, the given outputFolder will be populated by all the configuration files needed to start working on the project. Moreover, the local git repository will be initialized and the MapStore sub-module added and downloaded.

If you create a *standard* project, you can customize it editing **js/app.jsx**: look at the comments for hints and the MapStore documentation for more details.

The following steps are:

- `npm install` to download dependencies

- `npm start` to test the project
- `./build.sh` to build the full .war

Create a new project type

If you are not happy with the available project types (*standard*), you can extend them adding a new folder in **project**.

The folder will contain two sub-folders:

- **static**: for static content, to be copied as is to the project folder
- **templates**: for template files, containing project-dependent variables that will be replaced by the createProject script. You can use the following variables:
- **__PROJECTNAME__**: \<projectName> parameter value
- **__PROJECTDESCRIPTION__**: \<projectDescription> parameter value
- **__PROJECTVERSION__**: \<projectVersion> parameter value
- **__REPOURL__**: \<gitRepositoryUrl> parameter value

In addition to static and templates, the following files from the root MapStore folder will be copied:

- .babelrc
- .editorconfig
- LICENSE.txt

Update MapStore2 version in a project

To update MapStore2 version enter the MapStore2 folder and pull desired git version. If MapStore2 devDependencies have been changed you can manually update these in the project package.json file or run the script updateDevDeps

```
npm run updateDevDeps
```

The script will automatically copy the devDependencies from MapStore2 package.json to the project package.json file. All the project existing devDependencies will be overwritten.

To sync MapStore2 dependencies just run npm install from project root folder.

```
npm install
```

Also make sure to follow the migration guidelines here.

MapStore API usage

You can include MapStore in your application and interact with it via its JavaScript API

How to use

1. Create a map using the standard installation
2. Go to Share -> Embed
3. Copy the API html code and paste it in your application page

The map will now load inside your application

```
NOTE: If the map is using a Google Maps background you will
have to provide your own API key.
Add `&key=YOUR_API_KEY` in the <script> src value
```

MapViewer query parameters

In this section we will describe the available MapViewer query parameters that can be used when the map is loaded.

MapStore allows to manipulate the map by passing some parameters. This allows external application to open a customized viewer generating these parameters externally. With this functionality you can modify for instance the initial position of the map, the entire map and even trigger some actions.

Passing parameters to the map

Get Request

The parameters can be passed in a query-string-like section, after the `#<path>?` of the request.

Example:

```
#/viewer/openlayers/new?center=0,0&zoom=5
```

> **Note**
>
> The parameters in the request should be URL encoded. In order to make them more readable, the examples in this page will now apply the URL encoding.

POST Request

Sometimes the request parameters can be too big to be passed in the URL, for instance when dealing with an entire map, or complex data. To overcome this kind of situations, an adhoc `POST` service available at `<mapstore-base-path>/rest/config/setParams` allows to pass the parameters in the request payload `application/x-www-form-urlencoded`. The parameters will be then passed to

the client (using a temporary `queryParams-{random-UUID}` variable in `sessionStorage`). Near the parameters, an additional `page` value can be passed together with the params to specify to which url be redirect. If no page attribute is specified by default redirection happens to `#/viewer/openlayers/config`. The UUID used in the `queryParams-{random-UUID}` variable name is being added to the redirect URL in a query parameter named `queryParamsID=`. Assuming to use the default redirect value, the url will then look like the following: `#/viewer/openlayers/config?queryParamsID={random-UUID}`.

Example `application/x-www-form-urlencoded` request payload (URL encoded):

```
page=..%2F..
%2F%23%2Fviewer%2Fopenlayers%2Fnew&featureinfo=&bbox=&center=1%2C1
```

Here a sample page you can create to test the service:

```
<html><head><meta charset="UTF-8">
    <script>
        const POST_PATH = "rest/config/setParams";
        const queryParameters = {
            "page": '../../#/viewer/openlayers/config',
            "map": {"version":2,"map":{"projection":"EPSG:
900913","units":"m","center":{"x":1250000,"y":
5370000,"crs":"EPSG:900913"},"zoom":5,"maxExtent":
[-20037508.34,-20037508.34,20037508.34,20037508.34],"layers":
[{"type":"osm","title":"Open Street
Map","name":"mapnik","source":"osm","group":"background","visibili
            "featureinfo": '',
            "bbox": '',
            "center": '',
            "zoom": 4,
            "actions": [],
        };
        let i = 0;
        function createIframe() {
            i++;
            const iframe = document.createElement('iframe');
            iframe.name = `_iframe-${i}`;
            iframe.id = `_iframe-${i}`;
            iframe.style.width = "100%";
            iframe.style.height = "400px";
```

```
            document.body.appendChild(iframe);
            return iframe.name;
        }
        window.onload = function(){
            Object.keys(queryParameters).forEach(function (key)
{
                const element = document.getElementById(key);
                if (element) element.value = typeof
queryParameters[key] === "object" ||
Array.isArray(queryParameters[key]) ?
JSON.stringify(queryParameters[key]) : queryParameters[key];
            });
            const form = document.getElementById("post-form");
            form.addEventListener('submit', function() {
                const base_url =
document.getElementById('mapstore-base').value.replace(/\/?$/,
'/');
                const method =
document.getElementById("method").value;
                // handle GET URL
                if(method === "GET") {
                    event.preventDefault();
                    const page =
document.getElementById("page")?.value;
                    const data = new FormData(event.target);
                    const values = Array.from(data.entries());
                    const queryString = values
                        .filter(([k, v]) => !!v)
                        .reduce((qs = "", [k, v]) => `${qs}&${k}
=${encodeURIComponent(v)}`, "");
                    window.open(`${base_url}${page}?$
{queryString}`, "_blank");
                    return false;
                } else if (method === "GET_IFRAME") {
                    event.preventDefault();
                    const page =
document.getElementById("page")?.value;
                    const data = new FormData(event.target);
                    const values = Array.from(data.entries());
                    const queryString = values
                        .filter(([k, v]) => !!v)
                        .reduce((qs = "", [k, v]) => `${qs}&${k}
=${encodeURIComponent(v)}`, "");
                    const iframeName = createIframe();
                    const iframe =
document.getElementById(iframeName);
                    iframe.src = `${base_url}${page}?$
{queryString}`;
                    return false;
```

```
                }
                // handle POST and POST_IFRAME
                if(method === "POST_IFRAME") {
                    const iframeName = createIframe();
                    form.target = iframeName;
                } else if(method === "POST") {
                    form.target = "_blank";
                }
                form.action = base_url + POST_PATH;
                return true;
            })
        }
    </script>
</head><body>
    <fieldset>
        <legend>Options:</legend>
        <label>method:</label><select id="method">
            <option value="POST">POST</option>
            <option value="GET">GET</option>
            <option value="GET_IFRAME">GET_IFRAME</option>
            <option value="POST_IFRAME">POST_IFRAME</option>
        </select>
    <br/>
    <label>MapStore Base URL:</label><input type="text"
id="mapstore-base" value="http://localhost:8080/mapstore/">
</input><br/>
</fieldset>
<!-- Place the URL of your MapStore in "action" -->
<form id="post-form" action="http://localhost:8080/mapstore/
rest/config/setParams" method="POST" target="_blank">
    <fieldset>
        <legend>Params:</legend>
        <label for="map">map:</label><br/><textarea id="map"
name="map"></textarea><br/>
        <label for="page">page:</label><br/><input type="text"
id="page" name="page" value="../../#/viewer/openlayers/
config"></input><br/>
        <label for="featureinfo">featureinfo:</label><br/
><textarea id="featureinfo" name="featureinfo"></textarea><br/>
        <label for="bbox">bbox:</label><br/><input type="text"
id="bbox" name="bbox"></input><br/>
        <label for="center">center:</label><br/><input
type="text" id="center" name="center"></input><br/>
        <label for="zoom">zoom:</label><br/><input type="text"
id="zoom" name="zoom"></input><br/>
        <label for="marker">marker:</label><br/><input
type="text" id="marker" name="marker"></input><br/>
        <label for="actions">actions:</label><br/><textarea
id="actions" name="actions"></textarea><br/>
```

```
    </fieldset>
    <br/>
    <input id="submit-form" value="Submit" type="submit"><br/>
</form>
</body></html>
```

Available Parameters

Feature Info

Allows to trigger identify tool for the coordinates passed in "lat"/"lng" parameters.

Optional parameter "filterNameList" allows limiting request to the specific layer names. It will be effectively used only if it's passed as non-empty array of layer names. Omitting or passing an empty array will have the same effect.

GET: `#/viewer/openlayers/config?featureinfo={"lat": 43.077, "lng": 12.656, "filterNameList": []}`

GET: `#/viewer/openlayers/config?featureinfo={"lat": 43.077, "lng": 12.656, "filterNameList": ["layerName1", "layerName2"]}`

Simplified syntax:

GET: `#/viewer/openlayers/config?featureInfo=38.72,-95.625`

Where lat,lng values are comma-separated respecting order.

Map

Allows to pass the entire map JSON definition (see the map configuration format of MapStore).

GET:

```
#/viewer/openlayers/config?map={"version":2,"map":
{"projection":"EPSG:900913","units":"m","center":{"x":
1250000,"y":5370000,"crs":"EPSG:900913"},"zoom":5,"maxExtent":
```

```
[-20037508.34,-20037508.34,20037508.34,20037508.34],"layers":
[{"type":"osm","title":"Open Street
Map","name":"mapnik","source":"osm","group":"background","visibili
```

It also allows partial overriding of existing map configuration by passing only specific properties of the root object and/or the internal "map" object.

Following example will override "catalogServices" and "mapInfoConfiguration":

```
#/viewer/openlayers/config?map={"mapInfoConfiguration":
{"trigger":"click","infoFormat":"text/html"},"catalogServices":
{"services": {"wms": {"url": "http://example.com/geoserver/
wms","type": "wms","title": "WMS","autoload":
true}},"selectedService": "wms"}}
```

Center / Zoom

GET: `#/viewer/openlayers/config?center=0,0&zoom=5`

Marker / Zoom

GET: `#/viewer/openlayers/config?marker=0,0&zoom=5`

Bbox

GET: `#/viewer/openlayers/config?bbox=8,8,53,53`

AddLayers

This is a shortened syntax for `CATALOG:ADD_LAYERS_FROM_CATALOGS` action described down below.

GET: `#/viewer/openlayers/config?`
`addLayers=layer1;service,layer2&layerFilters=attributeLayer1='value';att`

`addLayers` parameter is a comma separated list of `<layerName>;<service>` (`service` is optional, and if present is separated from the layerName by a `;` .

In the example above:

- `layer1` and `layer2` are layer names;
- `service` is the service identifier of the catalog. If no service is provided, the default service will be used.
- `layerFilters` is a list of cql filters to apply to the corresponding layer in the same position of the `addLayers` parameter, separated by `;`

MapInfo

This is a shortened syntax for `SEARCH:SEARCH_WITH_FILTER` action described down below. In opposite to direct usage of action, `mapInfo` parameter can work with layers added by `addLayers` parameter. In this case search execution will be postponed up to the moment when layer is added to the map.

`mapInfo` handler will check if `addLayers` parameter is present and if it lists layer name used in `mapInfo` parameter. If so, it will postpone search to ensure that layer is added to the map. Otherwise, in case of no matches, search will execute immediately.

GET: `#/viewer/openlayers/new?addLayers=layer1;service&mapinfo=layer1&mapInfoFilter=BB='cc'`

Where:

- `layer1` is layer name.
- `service` is the service name providing layer data. Service name is optional. If no service is provided, the default service of the catalog will be used.
- `mapInfoFilter` is a cql filter applied to the layer.

Background

Allows to dynamically add background to the map and activate it. Supports default backgrounds provided by static service defined in `localConfig.json` (`default_map_backgrounds`) as well as other layers:

`#/viewer/openlayers/new?background=Sentinel;default_map_backgrounds`

`#/viewer/openlayers/new?background=layer1;service`

`#/viewer/openlayers/new?background=layer2`

Where:

- `Sentinel`, `layer1`, `layer2` are layer names,
- `service`, `default_map_backgrounds` are the service names providing layer data. Service name is optional. If no service is provided, the default service of the catalog will be used.

According to the implementation of `default_map_backgrounds` service, it is enough to pass desired layer name even partially, e.g. `background=Sen;default_map_backgrounds`, it will use the closest layer name match in this case.

Actions

To dispatch additional actions when the map viewer is started, the **actions** query parameter can be used. Only actions from a configured whitelist can be dispatched in this way (see the configuration section for more details).

```
// list of actions types that are available to be launched
dynamically from query param (#3817)
  "initialActionsWhiteList": ["ZOOM_TO_EXTENT",
"ADD_LAYER", ...]
```

The value of this parameter is a JSON string containing an array with an object per action. The structure of the object consist of a property type and a bunch of other properties depending on the action.

Available actions

Only the following actions can be used in the **actions** json string.

Zoom to extent

It zooms the map to the defined extent.

Example:

```
{
    "type": "ZOOM_TO_EXTENT",
    "extent": [1,2,3,4],
    "crs": "EPSG:4326",
    "maxZoom": 8
}
```

GET: `#/viewer/openlayers/config?actions=[{"type": "ZOOM_TO_EXTENT","extent": [1,2,3,4],"crs": "EPSG:4326","maxZoom": 8}]`

For more details check out the zoomToExtent in the framework documentation.

Map info

It performs a GetFeature request on the specified layer and then a GetFeatureInfo by taking a point from the retrieved features's geometry. This action can be used only for existing maps (map previously created).

With the GetFeature request it takes the first coordinate of the geometry of the first retrieved feature; that coordinates are then used for an usual GFI (WMS GetFeatureInfo) request by limiting it to the specified layer.

A **cql_filter** is also **mandatory** for that action to properly filter required data: that filter will be used in both request (GetFeature and GFI). If you don't need to apply a filter, you can use the standard INCLUDE clause (cql_filter=INCLUDE) so the whole dataset will be queried.

Requirements:

- The layer specified must be visible in the map
- There must be a geometry that can be retrieved from the GetFeature request

Example:

```
{
    "type": "SEARCH:SEARCH_WITH_FILTER",
    "cql_filter": "ID=75",
```

```
    "layer": "WORKSPACE:LAYER_NAME"
}
```

GET: `#/viewer/openlayers/config?`

`actions=[{"type":"SEARCH:SEARCH_WITH_FILTER","cql_filter":"ID=75","layer`

The sample request below illustrates how two actions can be concatenated:

```
https://dev-mapstore.geosolutionsgroup.com/mapstore/#/viewer/
openlayers/4093?
actions=[{"type":"SEARCH:SEARCH_WITH_FILTER","cql_filter":"STATE_F
{"type":"ZOOM_TO_EXTENT","extent":
[-77.48202256347649,38.74612266051003,-72.20858506347648,40.666647
4326","maxZoom":8}]
```

The MapStore invocation URL above executes the following operations:

- Execution of a search request filtering by **STATE_FIPS** with value 34 on the **topp:states** layer
- Execution of a map zoom to the provided extent

For more details check out the searchLayerWithFilter in the framework documentation

Scheduled Map Info

It works similarly to the `Map Info` action, but supports delaying of the search execution up to the moment when layer is added to the map. This behavior is used when search should be applied to the dynamically added layer (e.g. using `addLayer` parameter) :

Example:

```
{
    "type": "SEARCH:SCHEDULE_SEARCH_WITH_FILTER",
    "cql_filter": "ID=75",
    "layer": "WORKSPACE:LAYER_NAME"
}
```

GET: `#/viewer/openlayers/config?actions=[{"type":"SEARCH:SCHEDULE_SEARCH_WITH_FILTER","cql_filter":"ID=7{"type":"CATALOG:ADD_LAYERS_FROM_CATALOGS","layers":["WORKSPACE:LAYER_NAME"],"sources":["catalog1"]}]`

Add Layers

This action allows to add layers directly to the map by taking them from the catalogs configured, or passed.

Requirements:

- The number of layers should match the number of sources
- The source name can be a string that must match a catalog service name present in the map or an object that defines an external catalog (see example)

Supported layer types are WMS, WMTS and WFS.

Example:

```
{
    "type": "CATALOG:ADD_LAYERS_FROM_CATALOGS",
    "layers": ["workspace1:layer1", "workspace2:layer2",
"workspace:externallayername"],
    "sources": ["catalog1", "catalog2",
{"type":"WMS","url":"https://example.com/wms"}]
}
```

GET: `#/viewer/openlayers/config?actions=[{"type":"CATALOG:ADD_LAYERS_FROM_CATALOGS","layers":["layer1", "layer2", "workspace:externallayername"],"sources":["catalog1", "catalog2", {"type":"WMS","url":"https://example.com/wms"}]}]`

Data of resulting layer can be additionally filtered by passing "CQL_FILTER" into the options array. Each element of array corresponds to the layer defined in action:

```
{
    "type": "CATALOG:ADD_LAYERS_FROM_CATALOGS",
    "layers": ["workspace1:layer1", "workspace2:layer2",
"workspace:externallayername"],
    "sources": ["catalog1", "catalog2",
{"type":"WMS","url":"https://example.com/wms"}],
    "options": [{"params":{"CQL_FILTER":"NAME='value'"}}, {},
{"params":{"CQL_FILTER":"NAME='value2'"}}]
}
```

GET `#/viewer/openlayers/config?actions=[{"type":"CATALOG:ADD_LAYERS_FROM_CATALOGS","layers":["layer1","layer2","workspace:externallayername"],"sources":["catalog1","catalog2",{"type":"WMS","url":"https://example.com/wms"}],"options": [{"params":{"CQL_FILTER":"NAME='value'"}}, {},{"params":{"CQL_FILTER":"NAME='value2'"}}]}]`

Number of objects passed to the options can be different to the number of layers, in this case options will be applied to the first X layers, where X is the length of options array.

Quick Start

You can either choose to download a standalone binary package or a WAR file to quickly start playing with MapStore.

Binary package

The easiest way to try out MapStore is to download and extract the binary package available on MapStore release page. Here you can find some preconfigured maps as well users and groups. The goal for this package is to ease all the requirements needed for you to take MapStore for a test-drive.

We hope you enjoy MapStore!

How to run

Go to the location where you saved the zip file, unzip the contents and run:

Windows: `mapstore2_startup.bat`

Linux: `./mapstore2_startup.sh`

Point your browser to: http://localhost:8082/mapstore

To stop MapStore simply do:

Windows: `mapstore2_shutdown.bat`

Linux: `./mapstore2_shutdown.sh`

Package Contents

- MapStore
- Tomcat8

- Java JRE (Win and Linux)

Demo Maps

- **Aerial Imagery** - Simple map demo showing some aerial imagery data
- **WFS Query Map** - Demo map configured with MapStore built-in ability to query feature over WFS
- **User Map and User1 Map** - Map only visible to *user* and *user1* respectively, to demonstrate MapStore capabilities on user/group management and permissions.

Demo accounts/groups

Users	Groups
admin/admin	MyGroupAdmin,everyone
guest	everyone
user/user	everyone
user1/user1	everyone, MyGroup

WAR file

Download the WAR file from the latest release here.

All the releases

After downloading the MapStore war file, install it in your java web container (e.g. Tomcat), with usual procedures for the container (normally you only need to copy the war file in the webapps subfolder).

If you don't have a java web container you can download Apache Tomcat from here and install it. You will also need a Java7 JRE.

Then you can access MapStore using the following URL (assuming the web container is on the standard 8080 port):

http://localhost:8080/mapstore

Use the default credentials (admin / admin) to login and start creating your maps!