# Location Search with Coveo for Sitecore and Google Maps

## Executive Summary

In my work within a project, I’ve faced a huge challenge when I have to build a Location Search Page integrating Sitecore, Coveo JavaScript Search API, Google Maps API and Google Geocode Maps API.

In a nutshell, what it was needed was a search page in which the end user could type a zip code, city name, state name and receive as result a set of locations based that search criteria. So, for instance, if the user searched for “Seattle”, it would receive a set of location results within Seattle and those results would be ordered per distance using the geolocation of the user.  
However, that’s not the end of the history because there was another type of search: the named searches. In this case, the user would type a location name and the page would return a set of location results based on that name ordered by relevancy but showing how far they are based on the current user’s geolocation.

This asset handles all those scenarios using the before mentioned technologies.

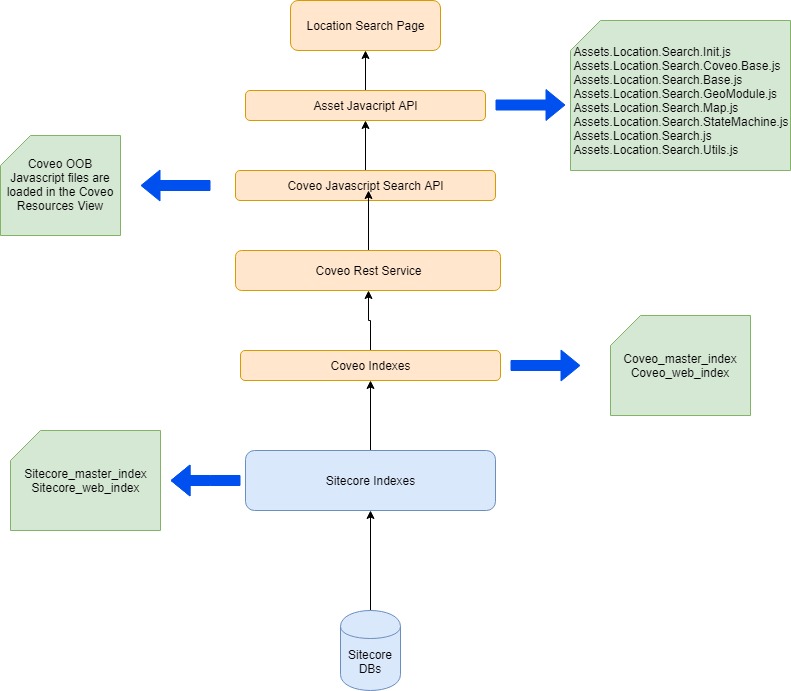
## Requirements

* 1. NET Framework 4.5 or above.
  2. ASP MVC 4 or above.
  3. Sitecore 8.2 or above working instance.
  4. MongoDB service running for Sitecore Analytics.
  5. Coveo for Sitecore 4 or above installed.
  6. Google Maps API key
     1. This key can be created thought the Google Developer Console using a Google Account for local development and testing purposes. The package contains a test-key which, theorically, cannot be used because the ownership of it its based on the google account from which is created.

On live environments, a full api key is needed based on the project need to not face request limits.

* 1. Optional: Bootstrap 3 or above
  2. jQuery 3 or above.

## Architecture



### **Asset JavaScript API**

This API is the core code of the asset, is a set of JavaScript files which are built using the JavaScript Module Pattern to accomplish the needs specified in the executive summary. Each file contains a self-contained module defined through JavaScript’s IIFE (Immediate Invoked-Function expression) that is, a JavaScript function that’s automatically executed when the browser finds it without the need of an explicit call.  
  
All of these modules can be extended. The module structure for this specific case follows this approach:

* The **Assets.Location.Search.Init** file contains the JavaScript Namespace definition and a function to create new namespaces.
* The **Assets.Location.Search.Coveo.Base** file is the “parent” module in a virtual hierarchical structure, because it contains all the variables and methods that directly consumes the Coveo Search JavaScript API. By designing this, we have the ability to centralize the coveo-JavaScript-directly calls in one place as well flexibility, scalability and maintainability of the code, because if we need to access coveo from another page with the JavaScript API, the only thing we have to do is to include this file and consume it from a upper layer. This module tt contains several submodules within it, such as:
  + The Facet Module, which takes care of handing the DOM elements that represents each facet module and exposes several methods to get and set is values programmatically
  + The State Module, which takes care of handling the QueryStateModel object that coveo exposes to handle the state of the page based on event. For this Asset, I’ve defined several custom states to model the flow of the page; you’ll see this more in detail in the Page Flow section. Last but not least, each custom state is exposed as a query string in the URL.
  + The See More Module, which takes care of the see-more logic. Since this Asset doesn’t use Coveo OOB pager, I needed a custom logic to handle the see more clicks to fetch more results based on the page size defined by rendering parameter.
  + The Results Module is an simplified implementation of the factory pattern to create custom objects based on the Coveo Query Results object.
* The **Assets.Location.Search.Search.Base** file is the direct child of the Coveo.Base file but at the same time, it’s the parent module for the Location Search JavaScript API. This means that if we would need another type of location search, such as by “City and State” instead of a global one, we could plug in or custom logic to this module to consume its base functions. As you already seen in the code, the first parameter that receives is the Coveo.Base module; by doing this, I’m able to access to all the functions of that module as well as inject new functions from the current module.  
  The Search.Base module handles takes care, amongst several things:
  + To fetch the Latitude and Longitude Facets as well as exposing get and set methods.
  + To fetch any other kind of non-range facet, such as normal filters.
  + To fetch the distanceUnits from the Rendering Parameters as well as to build the distance query function as well as its corresponding unit conversion and a sort criteria function.
  + To handle the custom message presented to the user by leveraging its visibility.
  + To handle the SkipGoogleSearch logic based on the SkipGoogleSearch rendering parameter, this is a comma-separated string which values will pass the call to google geocode. This will be detailed in the Page Flow section.
* The **Assets.Location.Search.GeoModule** file is another module that takes care of all the logic related with latitude and longitude coordinates, the call to Google Geocode and the update of the Latitude & Longitude range facets based on the search input. Both current coordinates and the Sitecore’s GeoIP output coordinates are stored in custom object to be persisted while browsing the page in order to be accessed and used by other modules.
* The **Assets.Location.Search.Map** file is another module that encapsulates all the logic and calls to the Google Map API, such as:
  + Creation and Initialization of the Map
  + Creation and update of the pins (markers)
  + Map event listeners handlers
  + Map’s boundaries update and change.
  + Refreshing the map’s center based on the search input and its results.
* The **Assets.Location.Search.StateMachine** file is the module that contains all the custom and core logic for the Location Search Page. It’s a hybrid implementation of the State Pattern through JavaScript and takes care of the whole behavior of the page. This will be deeply detailed in the Page Flow section, but in a nutshell: the location search page is modeled as a set of states with its custom transitions and a set of instructions that are only executed if the State Machine is in the right state.
* The **Assets.Location.Search.Utils** file contains several utils functions such as a Javascript equivalent for C#’s String.Format and String.contains as well as Number.Round function and a Boolean function to detect the IEVersion of the browser.
* Finally, the **Assets.Location.Search**.**js** file contains the calls to all the above-mentioned modules within the coveo events handler in which the logic of the Location Search is constructed. This also means that if in the future, another type of coveo search is needed, such as a Location by City State page, the only thing that has to be done from the JavaScript side is to build a new JavaScript file calling the above modules as the same way as it being written in the Assets.Location.Search.js file.

### **Coveo Javascript API**

Coveo own documentation about Coveo JavaScript Search API is extensive enough to know how this API works and how to build apps against it; nonetheless, it’s important to point that:

* It exposes several event handlers to attach to based on the different states of the coveo query., such as when the query begins, during the building of its expression, before being sent to the REST service, during the processing in the server and when it comes back.
* It has its own jQuery helper class to handle DOM elements, which can be accessed through Coveo.$; to avoid referential issues, I’m sending the $ var as a parameter to all related coveo modules.
* It allows to modify the query on the fly before being sent to the server.

### **Coveo REST Service**

Since this asset is majority build in the client side of Coveo and I’m not doing any server-side query processing, there’s no point to specify too much about this subject. Following the same concept as the Javascript API, Coveo documentation is extensive enough about this. The most important thing to note here is that the Coveo REST service endpoint could be configured via rendering parameters and it’s syntax share is the same as the Sitecore ContentSearch API

### **Coveo Indexes**

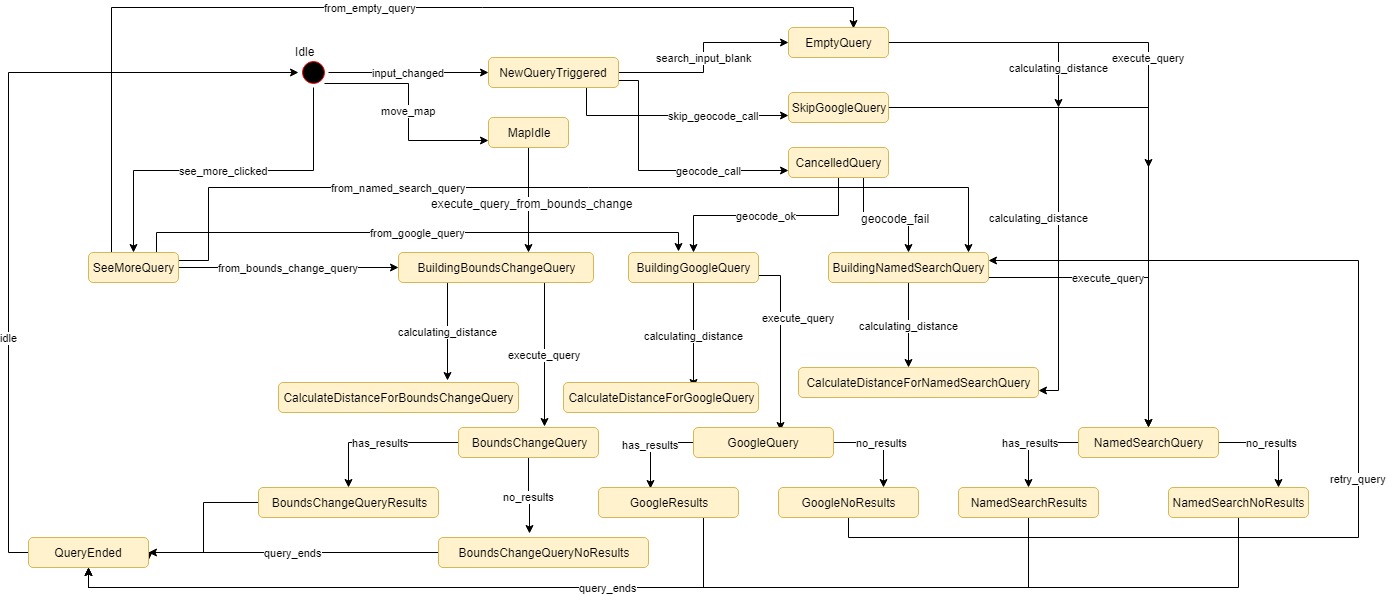
A fresh installation of Coveo, creates two important indexes in the Sitecore instance, that work against each Sitecore counterpart index; this means:

* Coveo\_master\_index is the coveo index that works against the sitecore\_master\_index and points to the Master Database
* Coveo\_web\_index is the coveo index that works against the sitecore\_web\_index and points to the data in the Web Database.

Every time that content or template fields are changed, new facets are created, or the index config is updated, a new rebuild of the indexes should be run in order to populate the results to the live site. This job could be configured using several strategies that are out of scope of this document.

### **Page Flow**

As I said in the Architecture section, the behavior of the Location Search Page is modeled using a State Pattern and there’s a hybrid State Machine implementation of it. This figure shows all the states and transitions that the page could have.



Taking a look to the figure, we could see that the **initial** state of this State Machine is the **Idle** state, which means that the user loaded the page and did nothing else. From this point of time, one of the following flows could occur:

#### Google Search

If the user enters a text and triggers a new search, the State Machine will pass from the ***Idle*** state to the ***NewQueryTriggered*** state. From here, the keyword will be evaluated and then sent to the geocode module to make an async call to the geocode function of Google Geoding API. Since the call is asynchronous, I need to find a way to await the page until the response comes back; that can be achieved by cancelling the coveo query in the ***NewQuery*** event handler and rethrowing it after the response returned. That’s why the ***CancelledQuery*** state exists; it will receive the response and based on it, trigger a new transition.

Let’s suppose the geocode response is ok, then the state machine will pass to the ***BuildingGoogleQuery*** state in which the code will handle the geocode response to update the map’s center coordinates and build the latitude and longitude facet-range values. In this point of time, the ***newQuery*** coveo event has enden and the ***buildingQuery*** event is triggered. A hander in this event will evaluate if Sitecore GeoIP module is working fine; if it is, it means that the user is geolocated, therefore will trigger a transition to calculate the distance between the coordinates received from the geocoder and the ones stored in the latitude & longitude indexed fields, build the distance, conversion and sorting query functions to be injected in the underlying query. In the case that the GeoIP module is not working, the code will not calculate distance and will not modify the underlying query. Either way, once that’s done, the ***buildingQuery*** event ends.

The next coveo event that is triggered is the ***doneBuildingQuery***, in which is used to calculate the number of results to be shown. Since the user started a new search, the see more state is zero, therefore the results count will be defaulted to the page size.

The next coveo event from which I had a handler to listen to is the ***querySuccess***, which is triggered once the query is done in the server and returned the returned the results set, if they are. In this handler, I’m evaluating the results count: if it’s greater than zero, then a has\_results transition is triggered leading the State Machine to the ***GoogleResults*** state, in which the coveo query results are converted to a custom-on-the-fly object that is going to be sent to the Map module to draw the pins in the map. If it’s the first time that the pins are drawed in the map, the code receives the collection of results and creates a marker per each result; if it’s not the first time, I’m using a pair of operations of the Set Theory such difference and intersection to calculate which pins to keep in the map and which to remove in order to refresh them in the map. Once that’s done, the ***query\_ends*** transition is triggered and the State Machine returns to the ***Idle*** state.

However, let’s suppose that the geocoder returned a correct response but it yielded zero results, in that particular scenario, the State Machine triggers a ***retry\_query*** transition to re-execute the query as a named search query. That’s the next flow I’ll talk about.

#### Named Search

The beginning of the flow is pretty much the same as the google query one, except that in this scenario, geocode response failed, which means that the geocoded didn’t yield any location. From the ***CancelledQuery*** state, the State Machine pass to the ***BuildingNamedSearch*** state during the ***buildingQuery*** event in which the value of the latitude & longitude facet range filter is reset if they were active because a Google Search was triggered before. I’m doing this because if they were active, the named search will narrow its results to the keyword match AND the latitude & longitude facet ranges. Once that’s done, the same GeoIP module evaluation is done. Nonetheless, if the module is working and the distance has to be calculated, this time it will calculated be between the center of the coordinates returned by GeoIP and the latitude & longitude indexed fields, converted to the proper unit but it won’t be used as sort criteria field; the sort criteria for the named search query is Coveo OOB relevancy. As the same as the Google Query flow, in case GeoIP is not working no distance calculation is done and the underlying query is not modified.

In the ***doneBuildingQuery*** event, the same calculation for the number of results is made.

In the ***querySuccess*** event, the same logic that is made in the Google Search flow is done. This is: to draw the pins in the map, refresh its center and boundaries.

#### Query from Bounds Change

This flow models the scenario when query is triggered when the user drags or zoom the map. It could happen in three different points of time:

* After the page load
* After a Google Query
* After a Named Search Query

Google Maps JavaScript API documentation is pretty extensive and exhaustive about the map events, so I won’t go to deep in the explanation of that; however, there three events in which should we be focused in: ***zoom****\_****change***, ***dragend*** and ***mapIdle***.

As their name implies, the ***zoom\_change*** event is triggered if the user zooms in / zoom out the map or when the zoom is changed programmatically. The ***dragend***event is triggered if the user drags the map the left/right and/or up/down or when the center of the map is changed programmatically. Last in the line is the ***mapIdle*** event which is triggered after all events in the map are done. This makes this event the key event for two reasons:

* Performance: it allows me to detect the point in time after the map was moved or zoomed without worry in triggering more than one coveo query. If I only detect every time the zoom is changed, and the user clicked ten times in the controls, I’d trigger nine additional queries that aren’t needed as well as will kill the server’s performance.
* To avoid infinite recursion scenario: in the map module file, there’s a subset of custom state to check whether a particular map event was triggered. Combining this with the mapIdle listener, I can avoid an infinite query execution caused because the dragend state was triggered when I programmatically updated the center of the map.

After this surfaced explanation about google maps events, the flow for the bounds change query works as follows:

I evaluate the flags of the map events in the mapIdle event listener and if they met the desired values, a transition of type move\_map is triggred making the State Machine to pass from Idle state to BuildingBoundsChangeQuery. From here, I evaluate the latitude / longitude facet range values and update them with the values of the current map boundaries. This is an important thing to clarify: the latitude & longitude facet-range values are proportionally matched with the boundaries of the radius built in the map.   
From a graphical standpoint, you can thing of the latitude & longitude facet ranges as a set of coordinates to build an imaginary rectangle in the map that wraps all the location results returned based on the center we’re using. This clarification is important because this so-called rectangle is not the same as the full viewport of the map, it’s smaller.

#### Skipped Google Search

If you take a look to the Rendering Parameters of the Asset Location Search Result View rendering, you’ll see that it has a parameter called “SkipGoogleSearch”. This parameter is a comma-separated-string that can contain several search terms that could hit Google Geocoding API and yield results. Since it may exist the case when the user may be looking for a location by its name and the name contains something that clashes against geocode, that location could not appear in the results list. This parameter exist to resolve this scenario. Every keyword entered in that field would be treated as a named search input although it could yield results from Google Geocoding API.

#### No Results

If the user searches for something that doesn’t not yield results from Coveo, the State Machine will trigger the no\_results transition, passing from its current state to the equivalent no result state based on the search type. This means that if the user searches for something and a named search is triggered, the State Machine will end in the NamedSearchNoResult state before ending the query; this concept also applies to the Bounds Change Query flow, however as it was briefly explained before, there’s a exceptional no results case the GoogleNoResults state has an special flow in which a new named search query is triggred after it.

#### See More Query

The see more query is triggered everytime the user clicks on the See More button. However, this button is only visible when the count of yielded results is greater than the page size. If this condition is not met, then the button is hided. This query could be triggered in four different points of time:

* When the user lands on the page and wants to see more results after the first load.
* When the user wants to see more results after a Google Query
* When the user wants to see more results after a Named Search Query
* When the user wants to see more results after a Bounds Change Query.

The logic of the see more is handled in the See More Module within the Coveo.Base file, as mentioned in the architecture section and exposes the DOM element as well as the logic to build how many results are being shown after the user clicks on the button.

Every time the user clicks on the button, the amount of clicks is saved in an object and used as a custom state value to be persisted across the subsequent see-more-clicks; this value is then fetched in the doneBuildingQuery to update on-the-fly the Coveo OOB number of results property, which is in charge to define how many results must be shown in the page. Nonetheless, every time the user triggers another query, the custom state that handles the see more is reset to 0. This means that the user will end seeing 1 to n-th results again (being “n” the page size). This allows me to keep the search results and count consistent throught the several scenarios.

#### Optional Flow: Search by Code

If you take a deep look to the JavaScript code of the Asset API, you’ll find a function that builds a 4-digit regex to be applied in the search input; This logic is for those scenarios when you want to offer to the user a way to search by location using a unique field, thus giving only one-single result. That’s why the code when it detects that the regex matches, it will create redirect to the page detail of that location item. The details about how the URL is build, how the item is resolved and how the page detail is rendered are out of the scope of this document because it depends in the Sitecore architecture that your Sitecore project is using, so it’s up to you on how to connect the pieces together.

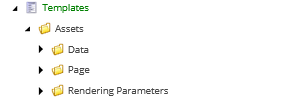
## Sitecore Package Content

Inside the whole Asset folder, you will find two zips files. One is the web project’s source code and the other is the sitecore package containing the compiled dlls of that project as well as the sitecore items. You could use this package to install it into your Sitecore instance as way to present the asset as a PoC to see if it matches your needs.

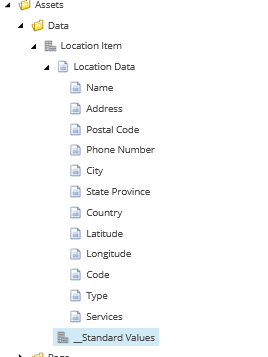
This section details what’s inside of the Sitecore package.

### **Templates**

As shown in the following image, there is an Assets folder which contains three subfolders.



The **Data** folder contains the Location Item template, which is used to model the location items that the Location Search Page will look for. It contains the following fields:



All of these fields are being used in the Location Search Page in one way or the other such as item presentation such as the name, postal code, phone number, address, city and state information; in the other hand, other fields are used as core field for the location search code, such as the Latitude and Longitude fields that are used for the Latitude and Longitude facets, respectively; or the Types and Services fields, which are used as the source for normal facets from which you can filter the results.

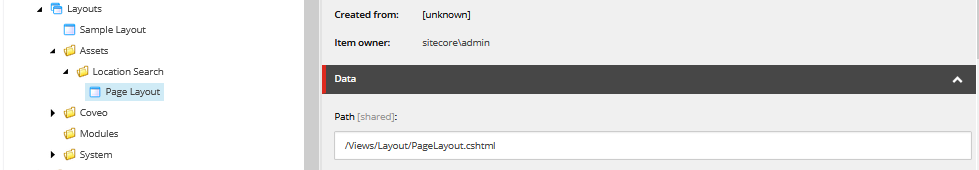


The second folder Page, contains the Location Search Page Template which inherits from Coveo MVC Search Page Template.



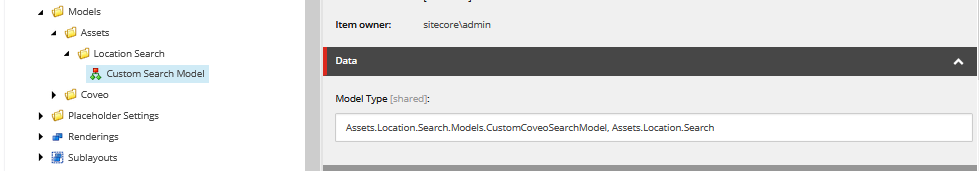
The third folder Rendering Parameters, contains a Asset Search Renderings Parameters template which inherits from the Coveo Search Parameters Template; thought this, I can acess to all the coveo OOB parameters plus I can extend the renderings to add the parameters I need, such as the Distance Unit and the Skip Google Search.

### **Layout**



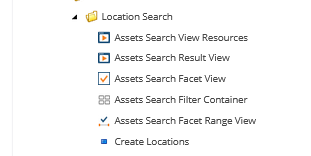
In the Layouts folder, there’s an Assets folder, as the picture shows. Based on the architecture of your project, you can use this Layout item for only the Location Search Page to have a segregate model of responsibilities or you can assign the page item to your own layout updating the Sitecore placeholders accordingly.

### Model



As the image shows, I’m defining a custom model to be used as the search model received for the Asset Location Search Result View rendering. By default, this rendering uses a pre-defined model in the Coveo folder, which is used to expose several API methods and properties such as all the HTML controls of the different elements that composes the search page. As you can see, all you need to define a custom model is the definition item, the full qualified name of the class and the assembly in which is located. If you want to define a custom model, it has to inherit from Coveo.UI.Mvc.Models.SearchModel. Once you do that and another trick, you can extend or customize the search model as you desire based on your needs.

### **Renderings**



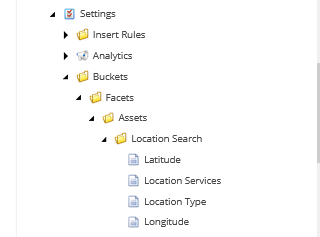
As the image shows, in the renderings folder you will finde an Assets folder that contains all those renderings. Except for the “Create Location” rendering, all the other ones are customized versions of the Coveo OOB renderings.

* The **Assets Search View Resources renderings** only loads all the static files needed for coveo to work, this includes its own css files and core javascripts. By this reason, this rendering should be assigned in the top position inside the presentation details of the search page, in order to not have console errors in the page.
* The **Assets Search Results View** rendering is the key rendering because it renders the whole HTML of the search page, such as the search box, its properties, the result template and the javascript files. It is also the rendering where you can define the model in the definition item and, once assigned in the presentation details, all the related parameters to the search page, such as the page size, the placeholder of the search box, the URL of the REST service, the visibility of the facets, breadcrumb, etc.
* The **Assets Search Facet View**, as its name implies, renders the all the visible facets that can be used to filter the results in the page. Coveo own documentation specifies all the types of facets that can be used, too.
* The **Assets Search Filter Container** is a custom rendering that works a container for both visible facets rendering as well as hidden ones. It only exposes two placeholders, one for the visible and one for the hidden. As you maybe saw looking through Sitecore, I’m setting the Latitude & Longitude facet-ranges as hidden facets; this is design decision because the meaning of the facet-ranges is to be updated programatically only, instead of showing them to the user.
* The **Assets Search Facet Range View** is the rendering that is in charge of constructing the latitude & longitude ranges, respectively. As the same of the Facet View rendering, in its rendering parameters you can define which index field can consume.

Finally, the **Create Location** rendering is a controller rendering that I’ve build to let the developers test the search location page through the bulk creation of location items. It can be assigned to a test-blank page and it will present two forms: one that receives an IP to consume a web api called IP STACK and returns a json with the geographical details of that IP such as: city, country, state, latitude and longitude. The second form will present a number control to specify how many Sitecore items you will want to create based on the data returned by IP STACK.

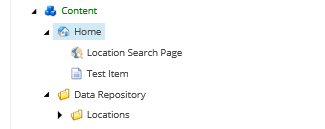
As a side note, in the config files there’s also an endpoint to google geocode’s web api. This will return the same json as its JavaScript equivalent. To use it, you need to have a full api key, because it limits the requests to one per day for ree api keys.

### **Facets**



The package will also install the definition items for each facet that the location search page uses. As you can see, there are the Latitud & Longitude definition items for each facet-range as well as the location type and location services for the normal-facets that will act as filters. This items will also need be defined at config level, which is covered in the Configuration Files section.

### **Data Items**



I’m only mentioning the data-items for two reasons:

* Be careful when you install the package. If you choose to overwrite all when installing it, it will replace your sitecore tree with this structure. I’d recommend to either click on the “Merge” option or to move the nodes outside the destination path.
* The location search page and the test-item page have presentation details, which makes them browseable items via URL; this doesn’t apply to the location items.

### **Configuration Files**

This asset has two sitecore patch include configuration file. They are: *webroot/<instance\_name>/Website/App\_Config/include/Assets.Search.Location.config*

*webroot/<instance\_name>/Website/App\_Config/include/Coveo/Custom.Asset.Coveo.Fields.config*

The first one contains the endpoints to IPSTACK and Google Geocoding web apis, the setting that defines the google api key to be used in the code.

The second one defines several important things:

* The item from which the Sitecore will crawl to both coveo\_master\_index and coveo\_web\_index
* The indexed fields definition for latitude, longitude, type and services; these are the sources fields for the facets and they have to match with the items defined in the “Facets” section.
* The multivalue separator character; when you use multivalue facets, coveo defines the semicolon “;” as the separator. I’m overriding that setting by putting the comma as the separator character.

Last but not least, I could have defined this config file in the include folder directly, but doing so will give you a runtime error this file has to be loaded after the coveo configuration files; take in mind the Sitecore loads all the files within the include folder in alphabetical order.

### View Files

The package will also install the corresponding views to all the renderings mentioned in the Renderings section in the following paths:

*webroot/<instance\_name>/Website/Views/layout*

*webroot/<instance\_name>/Website/Views/LocationSearch*