**Chapter 10 Non-spatial Queries**

**10.1 Introduction**

In Chapter [9](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#databases), we introduced the CARTO SQL API, and used it to display the result of a database query on a Leaflet map (Figure [9.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fig:example-09-01)). The query we used was constant: the same observations—namely the first 25 records in the plants table—were always loaded. In this chapter, we are going to extend example-09-01.html by adding a *user interface*, where one can choose which subset of the data gets loaded and displayed on the map. Specifically, we will use an input element, where the user selects a plant species, which we will use to dynamically generate an SQL query for filtering the plants table and getting the observations for that particular species only. That way, the user is able to load all observations of any given plant species, one at a time.

We will accomplish the above by going through several steps:

* Using SQL filtering based on column values (Section [10.2.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#filtering-based-on-attributes))
* Loading a subset of records from a database onto a Leaflet map (Section [10.2.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#displaying-on-a-map))
* Adding a <select> input for choosing a species (Section [10.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#creating-a-dropdown-menu))
* Collecting all unique species names, also using SQL (Section [10.4.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#finding-unique-values))
* Automatically adding all unique species names as <option> elements inside <select> (Section [10.4.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#adding-options))
* Associating the list with the map, so that changing the species choice will load new observations on the map (Sections [10.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#updating-the-map)–[10.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#rearranging-the-code))

The final result we aim at is shown in Figure [10.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fig:example-10-05).

**10.2 Subsetting with SQL**

**10.2.1 Filtering based on attributes.**In Chapter [9](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#databases), we used the following SQL query to load the first 25 records from the plants table on a Leaflet map (Figure [9.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#fig:example-09-01)).

**SELECT** name\_lat, the\_geom

**FROM** plants

**LIMIT** 25;

As mentioned in Section [9.6.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#non-spatial-queries), using SQL we can also *filter* the returned data, using the WHERE keyword combined with one or more criteria. For example, the name\_lat column specifies the Latin name of the observed species in the plants table. In case we want to select the observations of just one species, we can add the WHERE name\_lat = '...' expression to our SQL query, where ... is replaced with the species name of interest. For example, the following query returns all observations where name\_lat is equal to 'Iris mariae' (Figure [10.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fig:iris-mariae)):

**SELECT** name\_lat, the\_geom

**FROM** plants

**WHERE** name\_lat = 'Iris mariae';

For simplicity, let’s add LIMIT 1 to examine just the first observation for 'Iris mariae':

**SELECT** name\_lat, the\_geom

**FROM** plants

**WHERE** name\_lat = 'Iris mariae'

**LIMIT** 1;

When working with CARTO, we can execute this query with the corresponding SQL API call (Section [9.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#the-carto-sql-api)):

https://michaeldorman.carto.com/api/v2/sql?format=GeoJSON&q=

SELECT name\_lat, the\_geom FROM plants WHERE name\_lat = 'Iris mariae' LIMIT 1

Here is the returned GeoJSON string:

{

"type": "FeatureCollection",

"features": [

{

"type": "Feature",

"geometry": {

"type": "Point",

"coordinates": [34.412502, 31.019879]

},

"properties": {

"name\_lat": "Iris mariae"

}

}

]

}

Indeed, this is a "FeatureCollection" (Section [7.3.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#feature-collections)) with one feature and one non-spatial attribute (name\_lat), representing a single record for 'Iris mariae' as requested from the server.



FIGURE 10.1: *Iris mariae*, a rare iris species found in Israel. The image was taken in an ecological greenhouse experiment at the Ben-Gurion University in 2008.

**10.2.2 Displaying on a map.**For adding the observations of a single species on a Leaflet map, we can take example-09-01.html from Section [9.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#carto-and-leaflet) and modify the SQL query part. Instead of the query we used in example-09-01.html, we can use the new one that we have just constructed, where an individual species is being selected:

**SELECT** name\_lat, the\_geom

**FROM** plants

**WHERE** name\_lat = 'Iris mariae';

After creating the map object and loading a tile layer, the following code can be inserted to load the observations for 'Iris mariae' on the map[96](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fn96):

**let** urlGeoJSON = "https://michaeldorman.carto.com/api/v2/sql?" +

"format=GeoJSON&q=";

**let** sqlQuery1 = "SELECT name\_lat, the\_geom FROM plants " +

"WHERE name\_lat = 'Iris mariae'";

fetch(urlGeoJSON + sqlQuery1)

.then(**function**(response) {

**return** response.json();

})

.then(**function**(data) {

L.geoJSON(data).addTo(map);

});

This code section uses the fetch function for loading GeoJSON with Ajax (Section [7.7.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#making-ajax-requests-with-fetch)). As discussed in Section [9.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#carto-and-leaflet), we are requesting the GeoJSON content from the CARTO SQL API, using a URL for a specific SQL query, specified in sqlQuery1.

The resulting map (example-10-01.html), with marked 'Iris mariae' observations, is shown in Figure [10.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fig:example-10-01).

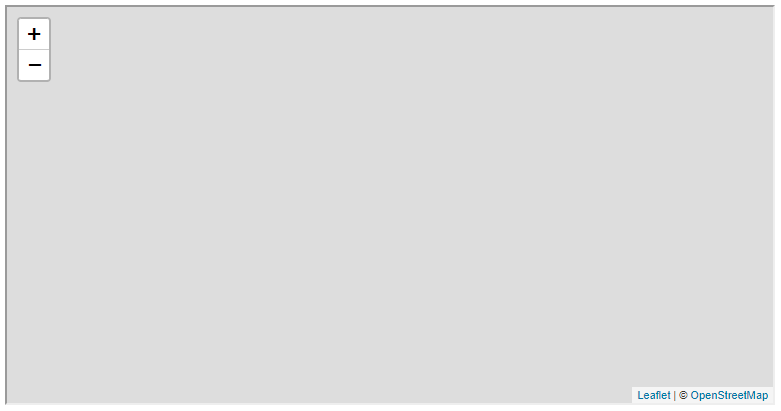


FIGURE 10.2: example-10-01.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\example-10-01.html)

**10.3 Creating a dropdown menu.**

Our ultimate goal in this chapter is for the user to be able to *choose* a species he/she wants to display, rather than seeing a predefined species such as 'Iris mariae' (Figure [10.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fig:example-10-05)). Therefore, our next step is to add an input element for selecting a species to load on the map. Since there are several hundreds of species in the plants table[97](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fn97), a bulleted list (Section [1.6.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\html.html#html-lists)) or a set of radio buttons (Section [1.6.13.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\html.html#radio-buttons)) will be too long to fit on screen and inconvenient to browse. A **dropdown menu** (Section [1.6.13.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\html.html#dropdown-lists)) is a good choice in case there are a lot of options to choose from. A dropdown menu can be opened to browse the species list and automatically closed again when selection is made, thus only temporarily obstructing other content on the web page.

As discussed in Section [1.6.13.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\html.html#dropdown-lists), a dropdown menu can be constructed using the <select> element, with an internal set of <option> elements for the various options. For example, suppose that we had just three plant species in the database, named "Abutilon indicum", "Acinos rotundifolius", and "Adonis aestivalis". In such case, the dropdown menu could be defined with HTML code such as follows:

**<select** id="species\_sel"**>**

**<option** value="Abutilon indicum"**>**Abutilon indicum**</option>**

**<option** value="Acinos rotundifolius"**>**Acinos rotundifolius**</option>**

**<option** value="Adonis aestivalis"**>**Adonis aestivalis**</option>**

**</select>**

Each <option> element has a value attribute, as well as text contents. Recall from Section [1.6.13](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\html.html#input-elements) that the value attribute specifies the ID sent to the server, or queried with the .value property (Section [4.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-interactivity.html#getting-and-setting-input-values)), while the text contents between the opening and closing tags (<option>...</option>) is actually displayed on screen. In this particular example, the value attribute and the text content are identical, but this does not necessarily have to be so (Section [1.6.13](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\html.html#input-elements)). In fact, when the value attribute is missing, the element contents are [treated as the values](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/option). Therefore, when text contents and value are identical, the value can be omitted, as in <option>Abutilon indicum</option>. The <select> element can thus be defined with more concise code, as follows:

**<select** id="species\_sel"**>**

**<option>**Abutilon indicum**</option>**

**<option>**Acinos rotundifolius**</option>**

**<option>**Adonis aestivalis**</option>**

**</select>**

To incorporate a dropdown menu—such as the one coded above—into our Leaflet map, we use the L.control function, which we already know how to use from Sections [6.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#adding-a-description), [8.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\symbology-and-interactivity.html#adding-a-legend), and [8.7.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\symbology-and-interactivity.html#dynamic-control-contents). The following code can be added to our <script> to build the dropdown menu:

**let** dropdown = L.control({position: "topleft"});

dropdown.onAdd = **function**() {

**let** div = L.DomUtil.create("div", "dropdown");

div.innerHTML =

'<select id="species\_sel">' +

'<option>Abutilon indicum</option>' +

'<option>Acinos rotundifolius</option>' +

'<option>Adonis aestivalis</option>' +

'</select>';

**return** div;

};

dropdown.addTo(map);

Like in previous examples of L.control, the above code is actually composed of three expressions:

* **Defining** a map control object, named dropdown, using L.control.
* **Setting** the contents of the control, assigning custom HTML code into the .innterHTML property. Note that the control itself is set as a <div> element with class "dropdown", and the <select> element inside the <div> is set with an ID attribute of "species\_sel".
* **Adding** the dropdown control to our map.

Next, we probably want to improve the default styling of the dropdown menu, by adding some CSS rules applied to #species\_sel, which refers to the <select> input element. For example, we can make the font slightly larger and add shadow effects around the input box. To do that, we add the following CSS code inside the <style> element within the <head>:

#species\_sel {

**padding**: 6px 8px;

**font**: 14px/16px Arial, Helvetica, sans-serif;

**background-color**: rgba(255,255,255,0.8);

**box-shadow**: 0 0 15px rgba(0,0,0,0.2);

**border-radius**: 5px;

}

Here is a short explanation of what each of the above styling rules does. You can go back to Chapter [2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\css.html#css) for more information on each of these properties.

* padding—The padding clears an area around the contents (and inside the border) of an element. When two values are given, the first one (6px) refers to padding-top and padding-bottom and the second one (8px) refers to padding-right and padding-left (Section [2.8.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\css.html#boxes)).
* font—Specifies the font properties. When two sizes are given, the first one (14px) refers to font-size, and the second one (16px) refers to line-height (the total height of text plus the distance to the next line). The font specification first gives specific fonts (Arial, Helvetica), then a general font family (sans-serif) to fall back on if those fonts are not installed on the system (Section [2.8.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\css.html#text)).
* background-color—The box background color is given in the RGBA format: intensity of red, green, blue, and transparency (alpha). In this case, we have white at 80% opacity (rgba(255,255,255,0.8)) (Section [2.8.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\css.html#color)).
* box-shadow—Adds a shadow around the element’s box. The sizes refer to horizontal offset (0) and vertical offset (0), where zero means the shadow is symmetrical around all sides of the box. Blur distance (15 px) determines how far the shadow extends. Color specification comes next, in this case the shadow color is black at 20% opacity (rgba(0,0,0,0.2)).
* border-radius—Adds rounded corners to an element, the size (5px) sets the radius (Section [2.8.4.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\css.html#border)).

The reason we do not need any styling rules for the entire control <div> (using the class selector .dropdown), just for the internal <select> element (using the ID selector #species\_sel), is that the control contains nothing but the dropdown menu. The exercise at the end of this chapter (Section [10.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#exercise-10)) will require creating a more complex control, with two dropdown menus. In that case, styling the entire control <div> is necessary too—for example, setting its external border style and the style of text labels for the separate dropdown menus (Figure [10.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fig:solution-10)).

* Open example-10-02.html.
* Uncheck each of the dropdown menu CSS styling rules in the developer tools, or delete them from the source code, to see their effect on dropdown menu appearance.

The resulting map example-10-02.html is shown in Figure [10.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fig:example-10-02). Note the newly added dropdown menu in the top-left corner of the map.

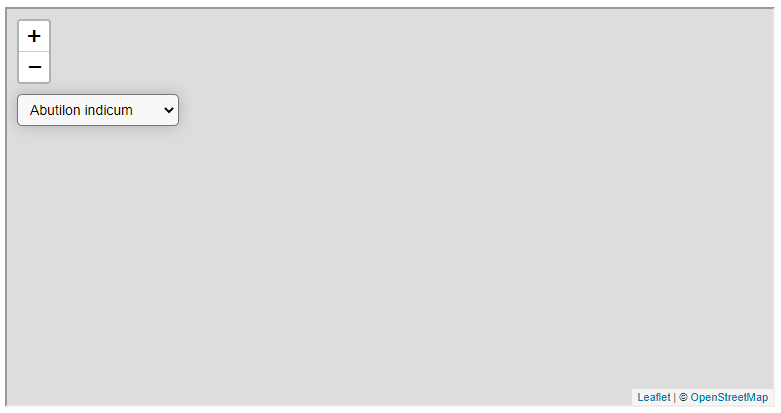


FIGURE 10.3: example-10-02.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\example-10-02.html)

The dropdown menu with the three individual species—which we see in the top-right corner in Figure [10.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fig:example-10-02)—is not yet functional. We can make a selection of a different species, but the map will still show 'Iris mariae' nevertheless, since it is hard-coded into our SQL query sqlQuery1 (open example-10-02.html and try it).

Before making the necessary adjustments for linking our dropdown menu with the marker layer on the map, we would like to improve the way that we create the dropdown menu in the first place. Instead of using pre-defined HTML code with just three species (as in example-10-02.html), we would like to build the options dynamically, with JavaScript code. Remember example-04-06.html (Section [4.11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-interactivity.html#modifying-page-based-on-data)), where we used the forEach iteration (Section [3.10.3.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#the-foreach-array-method)) for dynamically building a list of species names based on an array? This is exactly the technique we will use for dynamically populating our species dropdown menu with all of the unique species names.

**10.4 Populating dropdown options**

**10.4.1 How many species do we have?**Before looking into the dropdown menu construction, let’s check how many different plant species we actually have in the plants table. The following SQL query can be used to do that:

**SELECT** COUNT(**DISTINCT** name\_lat) **AS** n

**FROM** plants;

We are using the COUNT function along with the DISTINCT keyword to get the number of unique values in the name\_lat column, putting the result into a new column named n. The result is a small table with one column (n) and one row, as follows:

n

-----

417

(1 row)

We can get this result with the CARTO SQL API too, as follows:

https://michaeldorman.carto.com/api/v2/sql?q=

SELECT COUNT(DISTINCT name\_lat) AS n FROM plants

Here is the response:

{

"rows": [

{"n": 417}

],

"time": 0.04,

"fields": {

"n": {

"type": "number"

}

},

"total\_rows": 1

}

Omitting the format parameter implies the default format=JSON, thus our result is a JSON object (Section [9.7.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#query-example)). In JSON results coming from the CARTO SQL API, the "rows" property has the actual tabular content. The other properties in the JSON object are *metadata*, with information regarding processing time ("time"), the column data types ("fields"), and the total number of rows ("total\_rows")[98](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fn98).

The "rows" property is structured as an array, with array elements representing table rows. Each element is an object, representing all of the column values for that table row. In this case, there is just one row and one column, which is why the array has just one element, and that element is an object with just one property:

[{"n": 417}]

The response tells us that there are 417 species in the plants table, which we would like to include as dropdown menu options. Manually writing the <option> elements code for each and every one of the species is obviously not a good idea (Section [4.11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-interactivity.html#modifying-page-based-on-data)). Automating the construction of HTML code with 417 <option> element will be more convenient. What is more important, however, is the fact that our dynamically generated dropdown menu will be based on the exact species present in the database, in real-time. In other words, with the dynamic approach we no longer need to worry about updating the HTML code in case the database changes, since the HTML code is dynamically constructed based on the database itself and therefore always up-to-date. Remember that unlike a local file, the CARTO database, just like any other database, may be accessed and modified by different users on different platforms (Section [9.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#introduction-databases)). For example, a database with a list of rare and protected species may be periodically updated by the GIS department of the Nature and Parks authority, adding new observations, removing species that are no longer considered rare, and so on. If our web page connects to the database to dynamically generate its contents on page load, we do not need to worry about changing the code to reflect database modifications; the up-to-date version will be automatically loaded each time[99](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fn99).

* Removing the DISTINCT keyword from the above query gives the overall number of values in the name\_lat column, that is, the total number of records in the table.
* Execute the query through the CARTO SQL API to find out how many individual plant observations are there in the plants table[100](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fn100) (also see Section [12.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#heatmaps-with-leafletheat)).

**10.4.2 Dropdown menu placeholder.**We now begin with populating the dropdown options. As a first step, it is convenient to create an empty placeholder, like we did in the species list example from Section [4.11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-interactivity.html#modifying-page-based-on-data). That way, the placement of the contents on our page is easier to manage. To create the empty placeholder, instead of the HTML code for a <select> element with three pre-defined <option> elements (Section [10.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#creating-a-dropdown-menu)), we place the following HTML code which creates an *empty* <select> element, with no internal <option> elements at all:

**let** dropdown = L.control({position: "topright"});

dropdown.onAdd = **function**() {

**let** div = L.DomUtil.create("div", "dropdown");

div.innerHTML = '<select id="species\_sel"></select>';

**return** div;

};

dropdown.addTo(map);

Just like in example-10-02.html (Section [10.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#creating-a-dropdown-menu)), we hereby set an ID "species\_sel" for the <select> element. We already used the ID for styling the dropdown menu with CSS (Section [10.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#creating-a-dropdown-menu)). Now, in example-10-03.html (Section [10.4.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#adding-options)), we will also use the ID in our <script> to dynamically populate the dropdown <option> elements. To do that, we need to:

* Get the list of all unique species in the database (Section [10.4.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#finding-unique-values))
* Iterate over the species, adding an <option> element inside the <select> for each one of them (Section [10.4.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#adding-options))

**10.4.3 Finding unique values.**The following SQL query—which we already mentioned in the last exercise (Section [9.9](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#exercise-09))—can be used to get the list of all unique values in the name\_lat column of the plants table:

**SELECT** **DISTINCT** name\_lat

**FROM** plants

**ORDER** **BY** name\_lat;

The SELECT DISTINCT keyword combination before the name\_lat column name ensures that we only get the unique values in that column. We are also using the ORDER BY keyword to have the resulting table of species names **sorted** by alphabetical order. It makes sense that the options in the dropdown menu are alphabetically ordered: that way the user can easily locate the species he/she is interested in, by scrolling towards that species and/or by typing its first letter with the keyboard. Also, all species of the same *genus* will be listed together, since they start with the same word.

You can test the above query with the CARTO API:

https://michaeldorman.carto.com/api/v2/sql?q=

SELECT DISTINCT name\_lat FROM plants ORDER BY name\_lat

The JSON response should look as follows[101](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fn101):

{

"rows": [

{"name\_lat":"Abutilon indicum"},

{"name\_lat":"Acinos rotundifolius"},

{"name\_lat":"Adonis aestivalis"},

**...**

]

"time"**:** 0.032,

"fields": {

"name\_lat": {

"type": "string"

}

},

"total\_rows": 417

}

Next (Section [10.4.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#adding-options)), we will use this JSON response to fill up the dropdown menu where the user selects a species to display on the map.

**10.4.4 Adding the options.**To populate the dropdown menu, we need to incorporate the above SELECT DISTINCT SQL query (Section [10.4.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#finding-unique-values)) into our script. First, we define a variable named sqlQuery2 for the latter SQL query, listing the distinct species. At this point, we thus have two different SQL queries in our script:

* sqlQuery1 for loading observations of *one* species (Section [10.2.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#displaying-on-a-map))
* sqlQuery2 for finding all *unique* species names

For now, in sqlQuery1 we manually set the first displayed species to 'Abutilon indicum', since it is the first species in terms of alphabetic ordering, as shown in the above SQL API response (Section [10.4.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#finding-unique-values)). The species named 'Abutilon indicum' therefore comprises the first option in our dropdown menu. Later on, we will replace this *manual* way of selecting the first species with an automatic one, without hard-coding its name into the script (Section [10.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#rearranging-the-code)).

Also note that we have two different URL prefixes:

* urlJSON for getting sqlQuery2 in JSON format, using the format=JSON option (unspecified, since it is the default)
* urlGeoJSON for getting sqlQuery1 in GeoJSON format, using the format=GeoJSON option

Here are the definitions of both URL prefixes and queries:

*// URL prefixes*

**let** url = "https://michaeldorman.carto.com/api/v2/sql?";

**let** urlJSON = url + "q=";

**let** urlGeoJSON = url + "format=GeoJSON&q=";

*// Queries*

**let** sqlQuery1 = "SELECT name\_lat, the\_geom FROM plants " +

"WHERE name\_lat = 'Abutilon indicum'";

**let** sqlQuery2 = "SELECT DISTINCT name\_lat FROM plants ORDER BY name\_lat";

Then, the following expression fills the dropdown menu with all unique species names in the plants table, using sqlQuery2:

fetch(urlJSON + sqlQuery2)

.then(**function**(response) {

**return** response.json();

})

.then(**function**(data) {

**let** html = "";

data.rows.forEach(**function**(element) {

html += "<option>" + element.name\_lat + "</option>";

});

document.getElementById("species\_sel").innerHTML = html;

});

This is an Ajax request using fetch. What does it do? Let’s go over the code step by step.

* The fetch function is used to load the JSON response with the unique species names from the database, using urlJSON+sqlQuery2, eventually passed as an object named data. The contents of data is the same as shown in the JSON printout from Section [10.4.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#finding-unique-values).
* Then, the HTML code for all <option> elements is constructed using an iteration:
  + An empty string, named html, is initialized to hold the HTML code
  + The .forEach method (Section [3.10.3.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#the-foreach-array-method)) is used to iterate over each element in the rows property of data (i.e., the returned table rows), consecutively appending the <option> elements for all species
  + The resulting HTML code in html is assigned to the <select> element (id="species\_sel"), i.e., the dropdown menu is filled with the species options

As a result of adding the last code section, we now have a map (example-10-03.html) with a dropdown menu listing all available species. The observations of the first species on the list are shown on the map on page load (Figure [10.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fig:example-10-03)).

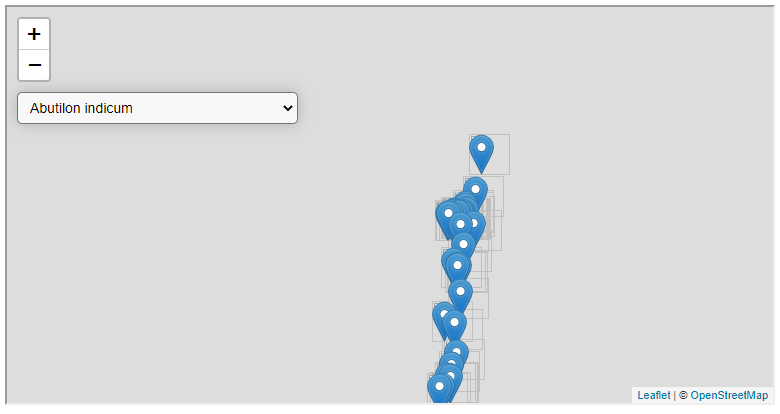


FIGURE 10.4: example-10-03.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\example-10-03.html)

* Insert a console.log expression inside the <script> of example-10-03.html so that the name of each species is printed in the console as it is being added to the list of options.
* You should see a printout with species named (Figure [10.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fig:species-console-log)).

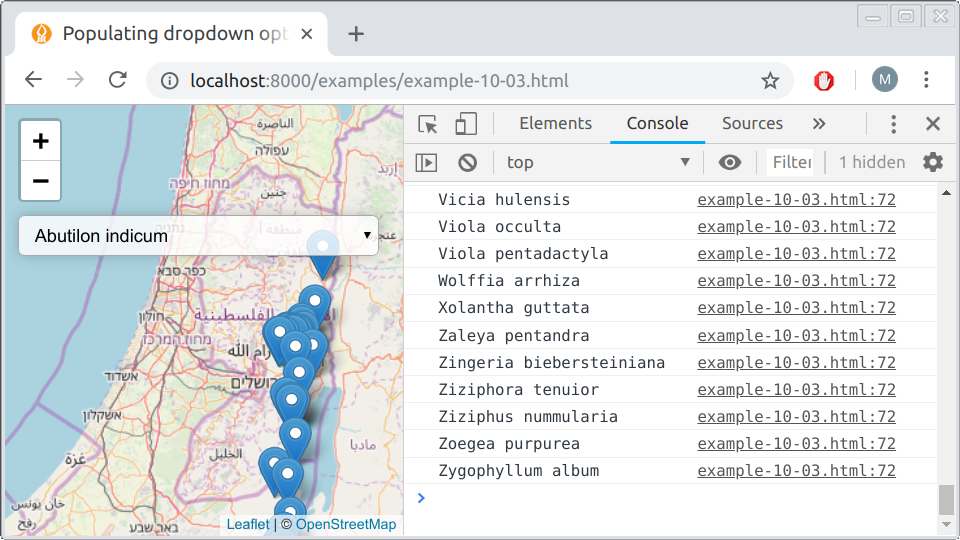


FIGURE 10.5: Species names printed with console.log

**10.5 Updating the map**

**10.5.1 Overview.**In example-10-03.html, we now have a web map showing observations of the *first* species from the plants table, as well as a dropdown menu with *all* species that are present in the plants table (Figure [10.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fig:example-10-03)). Try clicking on the dropdown menu in example-10-03.html and selecting a different species. Nothing happens! What is still missing is an event listener, triggering an update of the plant observations layer on selection change. In our case, whenever the species selection changes, the event listener function should do the following things:

* **Clear** the map of all previously loaded observations
* **Define** a new SQL query for getting the observations of the currently selected plant species
* **Send** the SQL query to the database and get the GeoJSON with the new observations
* **Display** the new observations on the map

**10.5.2 Manual example.**Before setting up the event listener, it is often convenient to start with a manually defined, specific case. Let us suppose the user chose a different species instead of 'Abutilon indicum', say 'Iris mariae', in the dropdown menu. The code for adding 'Iris mariae' observations on the map is exactly the same as the one we used for 'Abutilon indicum'. The only thing that’s changed is the species name in the SQL query, hereby named sqlQuery3. Try entering the following code section into the console of example-10-03.html (Figure [10.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fig:example-10-03)). This should load new observations, for 'Iris mariae', on top of the existing ones for 'Abutilon indicum':

**let** valueSelected = "Iris mariae";

**let** sqlQuery3 = "SELECT name\_lat, the\_geom FROM plants " +

"WHERE name\_lat = '" + valueSelected + "'";

fetch(urlGeoJSON + sqlQuery3)

.then(**function**(response) {

**return** response.json();

})

.then(**function**(data) {

L.geoJSON(data).addTo(map);

});

* Change the value of valueSelected in the above code section to a different species name, such as "Iris atropurpurea", then execute the modified code section in the console of example-10-03.html.
* You should see observations of the species you chose added on the map.

**10.5.3 Automatic updating.**In the last code section, we manually added new observations of another species from the database. What we now need is to associate this code section with dropdown menu *changes*. In other words, we want the selection of a new species in the menu to trigger the automatic removal of the old observations and loading the new ones. This is where the event listener comes into play.

The first thing we do is define a layerGroup named plantsLayer and add it on the map:

**let** plantsLayer = L.layerGroup().addTo(map);

The named layer group can be referenced later on, to clear it or add layers into it. This will make it easier for us to clear any previously loaded species and add markers for the currently selected one. Recall the GeoJSON viewer in example-07-02.html (Section [7.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#geojson-viewer)), where we used a layer group for clearing old GeoJSON layers whenever a new one is submitted from the text area input.

Accordingly, we replace the .addTo(map) part with .addTo(plantsLayer) when loading the initial species on the map. So, the following expression from example-10-03.html:

fetch(urlGeoJSON + sqlQuery1)

.then(**function**(response) {

**return** response.json();

})

.then(**function**(data) {

L.geoJSON(data).addTo(map);

});

is replaced with this one in example-10-04.html:

fetch(urlGeoJSON + sqlQuery1)

.then(**function**(response) {

**return** response.json();

})

.then(**function**(data) {

L.geoJSON(data).addTo(plantsLayer);

});

The second thing we do is “wrap” the manual example from the previous section, which we used to load "Iris mariae" observations, so that:

* The code for loading observations on the map is contained inside an event listener function, responding to changes in the dropdown menu (id="species\_sel"). The appropriate event type in this case is "change" (Section [4.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-interactivity.html#event-listeners)), which means that any *change* in the input element triggers the event listener.
* The species to load from the database is determined based on the currently selected value in the dropdown menu, using document.getElementById("species\_sel").value, rather than hard coded as "Iris mariae".
* The old observations are cleared before adding new ones, using .clearLayers().

The following code section does all of those things:

document

.getElementById("species\_sel")

.addEventListener("change", **function**() {

**let** valueSelected = document.getElementById("species\_sel").value;

**let** sqlQuery3 =

"SELECT name\_lat, the\_geom FROM plants WHERE name\_lat = '" +

valueSelected + "'";

fetch(urlGeoJSON + sqlQuery3)

.then(**function**(response) {

**return** response.json();

})

.then(**function**(data) {

plantsLayer.clearLayers();

L.geoJSON(data).addTo(plantsLayer);

});

});

Let’s go over the code, step by step. In the outermost part of the expression, an event listener is being binded to a DOM selection:

document

.getElementById("species\_sel")

.addEventListener("change", **function**() {

...

});

More specifically, an anonymous function is being set to execute every time the input element "#species\_sel" changes. Let’s now review the body of that anonymous function. First thing, the function defines a local variable named valueSelected:

**let** valueSelected = document.getElementById("species\_sel").value;

The value of valueSelected is the name of the currently selected species in the dropdown menu. The current value is obtained through the .value property (Section [4.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-interactivity.html#getting-and-setting-input-values)). This is conceptually similar to the calculator example example-04-07.html (Figure [4.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-interactivity.html#fig:example-04-07)), where we used the .value property to get the currently entered numbers, for displaying their multiplication product (Section [4.12](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-interactivity.html#working-with-user-input)).

Then, valueSelected is being used to construct a new SQL query string sqlQuery3. This query will be used to request the observations of the newly selected species from the database:

**let** sqlQuery3 = "SELECT name\_lat, the\_geom FROM plants " +

"WHERE name\_lat = '" + valueSelected + "'";

Finally, the new GeoJSON content is being requested with urlGeoJSON+sqlQuery3 using the fetch. Once the request finishes, the returned GeoJSON object is added on the map using L.geoJSON and .addTo(plantsLayer), but not before the old observations are cleared with plantsLayer.clearLayers():

fetch(urlGeoJSON + sqlQuery3)

.then(**function**(response) {

**return** response.json();

})

.then(**function**(data) {

plantsLayer.clearLayers();

L.geoJSON(data).addTo(plantsLayer);

});

The resulting map (example-10-04.html) is shown in Figure [10.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fig:example-10-04). Thanks to the event listener, the dropdown menu is now functional, and the map is responsive to the current species selection.

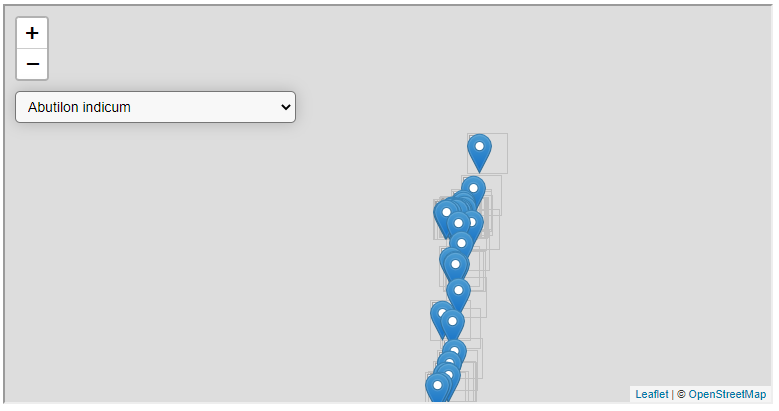


FIGURE 10.6: example-10-04.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\example-10-04.html)

**10.6 Refactoring the code**

Going over the complete code of example-10-04.html (Section [10.5.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#automatic-updating)), you may notice one important drawback: there are two code sections doing practically the same thing. Namely, we have the following code section for loading the initial query sqlQuery1, displaying the *initial* species 'Abutilon indicum' on page load:

**let** sqlQuery1 = "SELECT name\_lat, the\_geom FROM plants" +

"WHERE name\_lat = 'Abutilon indicum'";

fetch(urlGeoJSON + sqlQuery1)

.then(**function**(response) {

**return** response.json();

})

.then(**function**(data) {

L.geoJSON(data).addTo(plantsLayer);

});

and we also have the event listener, which does the same thing for each *subsequently* selected species, using a dynamically generated query sqlQuery3[102](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fn102):

document

.getElementById("species\_sel")

.addEventListener("change", **function**() {

**let** valueSelected = document.getElementById("species\_sel").value;

**let** sqlQuery3 =

"SELECT name\_lat, the\_geom FROM plants WHERE name\_lat = '" +

valueSelected + "'";

fetch(urlGeoJSON + sqlQuery3)

.then(**function**(response) {

**return** response.json();

})

.then(**function**(data) {

plantsLayer.clearLayers();

L.geoJSON(data).addTo(plantsLayer);

});

});

This duplication contradicts the [**Don’t Repeat Yourself**](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself) (DRY) principle—an essential guideline in computer programming. Basically, unnecessary duplication makes our code more difficult to manage, since we need to keep track of all duplicated parts and make sure they are synchronized every time we make changes in our code. For example, suppose we decide to add popups for the plant observation markers, say, displaying the observation date as well as the species name. In the repeated code structure from example-10-04.html, we would have to change L.geoJSON options in two places: the L.geoJSON function call that loads the initial species and the L.geoJSON function call that loads any subsequently selected species. If we forget to change one of those places, then popup content will look different depending on whether we changed the initial dropdown menu selection, which is clearly undesired.

A second issue in the present code configuration is that the initial species, the one displayed on map load, is hard-coded in sqlQuery1 (it is set to 'Abutilon indicum'). Like we said previously (Section [10.4.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#how-many-species-do-we-have)), the database may change in the future, so that 'Abutilon indicum' may be removed or a new species with alphabetical precedence may be added. In such a case, the initial view will be incorrect: the species initially loaded on the map will no longer match the first species inside the dropdown menu. Therefore, it is beneficial to determine the first species to load according to the real-time version of the database, rather than based on a constant string.

The solution to the duplication issue is to use a **function**, which also makes our code slightly shorter and more elegant. Instead of repeating the code section for loading plant observations in two places, we wrap the second code section—the one for loading the currently selected species—in a function called displaySpecies. The displaySpecies function loads the currently selected species on the map:

**function** displaySpecies() {

**let** valueSelected = document.getElementById("species\_sel").value;

**let** sqlQuery3 =

"SELECT name\_lat, the\_geom FROM plants WHERE name\_lat = '" +

valueSelected + "'";

fetch(urlGeoJSON + sqlQuery3)

.then(**function**(response) {

**return** response.json();

})

.then(**function**(data) {

plantsLayer.clearLayers();

L.geoJSON(data).addTo(plantsLayer);

});

}

The displaySpecies function can be used in the .addEventListener("change", ...) event listener, just like the anonymous function was in the previous version. The event listener definition from example-10-04.html:

document

.getElementById("species\_sel")

.addEventListener("change", **function**() {...});

is thus replaced with the following version in example-10-05.html:

document

.getElementById("species\_sel")

.addEventListener("change", displaySpecies);

This change takes care of clearing the map and loading a new species whenever the dropdown menu selection changes. However, how will the initial species be loaded and displayed on map load? To load the initial species, we simply need to call the function one more time in our script, *outside* of the event listener. The appropriate place to call the function is inside the Ajax request that populates the dropdown menu. That way, the first species is displayed right after the dropdown menu is filled with all of the unique species names:

fetch(urlJSON + sqlQuery2)

.then(**function**(response) {

**return** response.json();

})

.then(**function**(data) {

**let** html = "";

data.rows.forEach(**function**(element) {

html += "<option>" + element.name\_lat + "</option>";

});

document.getElementById("species\_sel").innerHTML = html;

displaySpecies(); *// Display initial species*

});

Since displaySpecies is being called after the dropdown menu was already populated, we can be sure that the initially loaded species corresponds to the first selected <option>, even if the database has changed. We no longer need to worry that a particular fixed species name in our code ('Abutilon indicum') still matches the first one in the alphabetically ordered species list from the database.

The resulting map example-10-05.html is shown in Figure [10.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fig:example-10-05). Visually and functionally it is exactly the same as example-10-04.html. However, the underlying code is rearranged and improved, which is also known as [**code refactoring**](https://en.wikipedia.org/wiki/Code_refactoring).

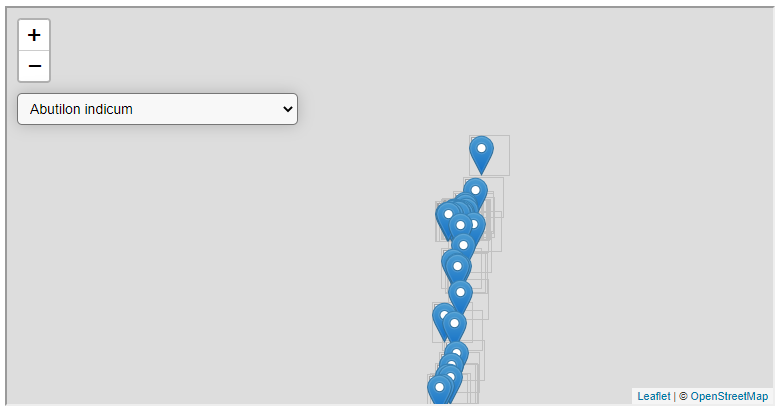


FIGURE 10.7: example-10-05.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\example-10-05.html)

**10.7 Exercise**

* Extend example-10-05.html into a web map with two dropdown menus—one for choosing the *genus* and another one for choosing the *species* (Figure [10.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fig:solution-10)).
* When the user selects a genus, the second dropdown should automatically update to list the species in that genus. The user then selects a species to show it on the map. If the user made no selection yet, the first species in each genus should be automatically displayed on the map.
* This is a challenging exercise, so here is a hint on how to start. Run the code section shown below in the console of example-10-05.html. This code constructs an object named species, whose properties are genus names and values are arrays with all species in that genus. Use the species object to populate the dropdown menus.

**let** url = "https://michaeldorman.carto.com/api/v2/sql?";

**let** urlJSON = url + "q=";

**let** sqlQuery = "SELECT DISTINCT name\_lat FROM plants ORDER BY name\_lat";

**let** species = {};

fetch(urlJSON + sqlQuery)

.then(**function**(response) {

**return** response.json();

})

.then(**function**(data) {

data.rows.forEach(**function**(element) {

**let** tmp = element.name\_lat;

**let** tmp\_split = tmp.split(" ");

**if**(species[tmp\_split[0]] === **undefined**) {

species[tmp\_split[0]] = [tmp];

} **else** {

species[tmp\_split[0]].push(tmp);

}

});

});

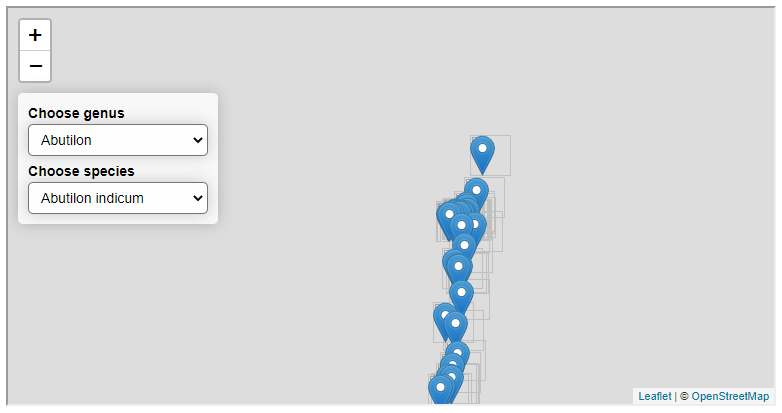


FIGURE 10.8: solution-10.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\solution-10.html)

1. For simplicity, this code lacks the onEachFeature option for displaying popups (which was used in example-09-01.html).[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fnref96)
2. There are 417 species in the plants table, to be exact. See how to find that out in Section [10.4.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#how-many-species-do-we-have).[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fnref97)
3. Remember that, as mentioned in Section [9.7.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#query-example), applying a non-spatial query (such as the unique species count) combined with the format=GeoJSON option will result in an error, because in order to produce a GeoJSON response the query must include the geometry column.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fnref98)
4. Note that the particular CARTO database used in this example is for demonstration, not directly related to the Nature and Parks authority, or any other users, and thus remains unchanged, but the principle still holds.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fnref99)
5. The answer is 23,827.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fnref100)
6. The "rows" property, which contains the returned table, is truncated here to save space, with the ... symbol representing further objects for the other 414 species not shown.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#fnref101)
7. Again, note that the arguments passed to L.geoJSON were replaced with ... to save space.