27

Recipe: Creating an Auto-Complete Text Box

This chapter covers

* Creating an auto-complete textbox in ASP.NET MVC
* Using a jQuery auto-complete plug-in

It’s not uncommon for text boxes to automatically suggest items based on what we type. The results are further filtered as we type to give us the option to select an available item with the mouse or keyboard. One of the first examples of this in the wild was Google Suggest as shown in figure 27.1.

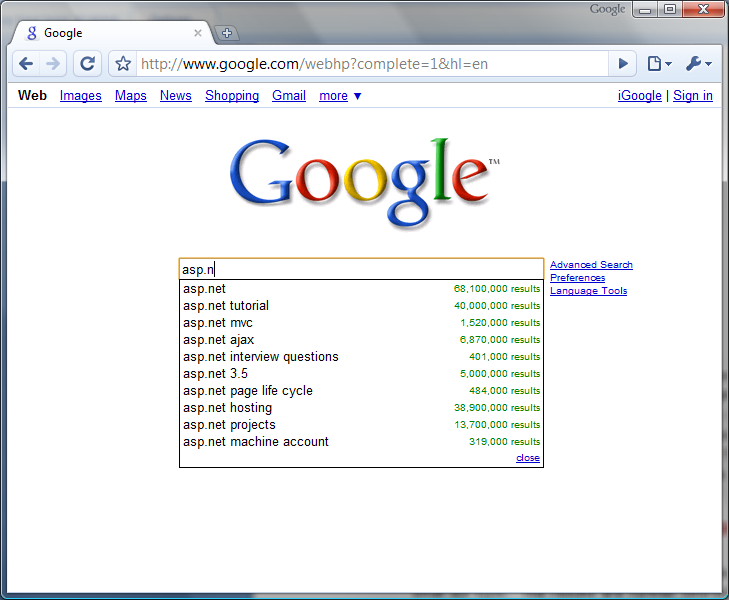


Figure 27.1 Google Suggest filters options as you type.

A rudimentary implementation of this automatic suggestion feature would be to monitor key presses and fire off AJAX requests for each one. Of course this means that a fast typist would trigger many requests, most of which would be immediately discarded for the next request coming in 5 milliseconds. An effective implementation will take into account a typing delay and also provide keyboard/mouse support for selecting the items.

Luckily jQuery has an extensive list of plugins available. One such plugin is Dylan Verheul’s autocomplete.

Dylan Verheul’s autocomplete

You can download the autocomplete plugin at [http://www.dyve.net/jquery/](http://www.dyve.net/jquery/?autocomplete) with a few others including googlemaps and listify.

Another, arguably equally popular plug-in exists from <http://bassistance.de/jquery-plugins/jquery-plugin-autocomplete/>. The plug-ins are similar, so most of what you read here will apply to that plug-in as well.

The basic idea is that you have a simple text box on your page. The jQuery plug-in adds the behavior necessary to handle key press events and fire the appropriate AJAX requests to a URL that will handle the request. The URL points to a controller action, and by convention the response is formatted so that the plug-in could handle the response.

Assume for our purposes that we want to filter U.S. cities in the text box. The first step is to add a controller, action, and view for displaying the UI for this example. Ensure that jQuery (in this case jquery-1.2.6.js) and jquery.autcomplete.js are referenced at the top of the view (or master page).

<script type="text/javascript" src="../../scripts/jquery-1.2.6.js"></script>

<script type="text/javascript" src="../../scripts/jquery.autocomplete.js"> </script>

Next, add the text box. In this example we’ll call it city.

<%= Html.TextBox("city") %>

Package this up with a simple controller (listing 27.1), and the result will be similar to that show in figure 27.2.

Listing 27.1 Controller and action for displaying our test page

public class HomeController : Controller

{

public ActionResult Index()

{

return View();

}

}

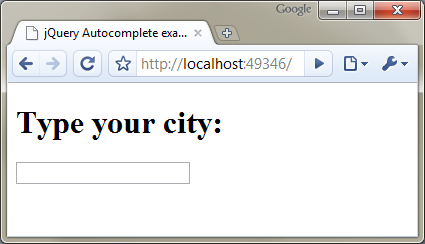


Figure 27.2 Our simple view with a text box

Now we add a little JavaScript to add the auto-complete behavior.

<script type="text/javascript">

$(document).ready(function() {

$("input#city").autocomplete('<%= Url.Action("Find", "City") %>');

});

</script>

Place the script in the <head> of the page. You can see that the URL for the auto-complete behavior is specified as Url.Action("Find", "City"). This will point to a Find() action on the CityController. We'll need to write this controller and action as shown in listing 27.2.

Local Data Mode

The autocomplete plug-in can also filter local data structures. This is useful when you have a limited set of data, and you want to minimize requests sent to the server. The autocomplete plug-in in local mode is also much faster, because there is no AJAX request being made behind the scenes. The only downside is that you must render the entire array onto the view as a javascript array.

Listing 27.2 Action to find cities from an auto-complete AJAX request

public class CityController : Controller

{

private readonly ICityRepository \_repository;

public CityController()

{

string csvPath = |#A

System.Web.HttpContext.Current |#A

.Server.MapPath("~/App\_Data/cities.csv"); |#A

\_repository = new CityRepository(csvPath); #B

}

public CityController(ICityRepository repository) #C

{

\_repository = repository;

}

public ActionResult Find(string q) #D

{

string[] cities = \_repository.FindCities(q);

return Content(string.Join("\n", cities)); #E

}

}

#A Load CSV file containing citites

#B Load CSV into repository

#C Testable constructor

#D Autocomplete sends parameter ‘q’

#E Return raw text

The details of the CityRepository can be found in the code samples provided with the book. For now, we’ll focus on the new Find(string q) action. Because this is a standard action, you can navigate to it in your browser and test it out. Figure 27.3 shows a quick test.

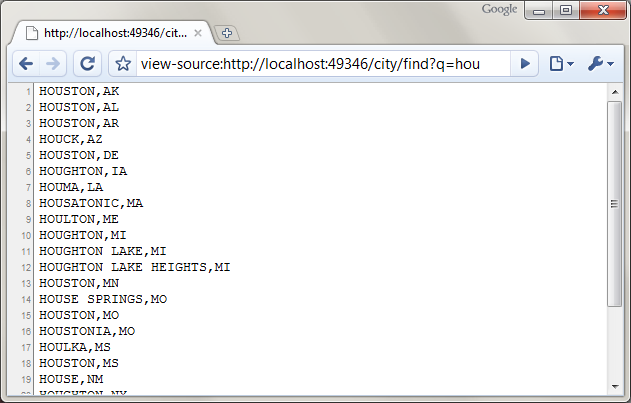


Figure 27.3 A simple HTTP GET for the action with a filter of "hou" yields the expected results.

Now that we are sure that the action is returning the correct results, we can test the text box. The JavaScript we added earlier hooks up to the key press events on the text box and should issue queries to the server. Figure 27.4 shows this in action.

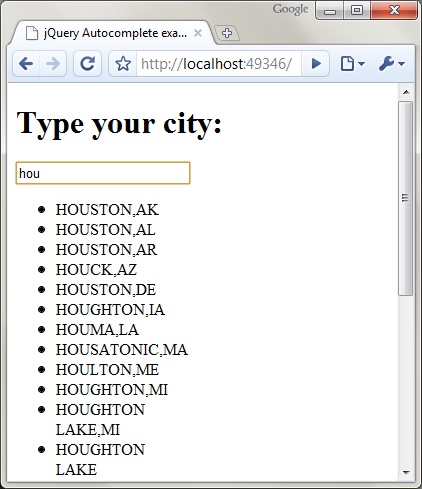


Figure 27.4 The results are displayed in a <ul> tag. We can apply CSS to make it look nicer.

The drop-down selections are unformatted by default, which makes them a little ugly. CSS magic will make it look nicer. Listing 27.3 shows sample CSS for this transformation.

Listing 27.3 CSS used to style the auto-complete results

<style type="text/css">

div.ac\_results ul

{

margin:0;

padding:0;

list-style-type:none;

border: solid 1px #ccc;

}

div.ac\_results ul li

{

font-family: Arial, Verdana, Sans-Serif;

font-size: 12px;

margin: 1px;

padding: 3px;

cursor: pointer;

}

div.ac\_results ul li.ac\_over

{

background-color: #acf;

}

</style>

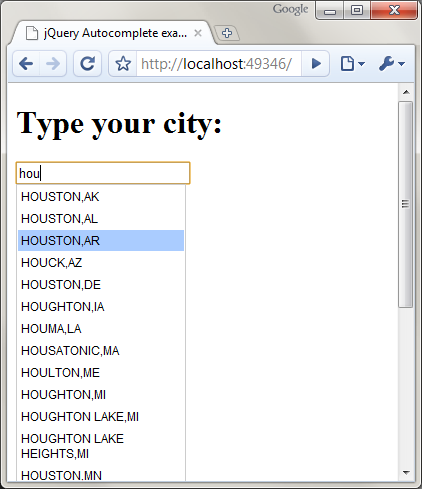


Figure 27.5 The styled drop down results look much nicer. The selected item is highlighted and can be chosen with the keyboard or mouse.

The options of autocomplete plug-in enable you to configure it to your needs. For the case that we've shown here, it's as simple as this:

$(your\_textbox).autocomplete('your/url/here');

No column headers on table 27.1, according to author

Table 27.1 Common options for the plug-in

|  |  |
| --- | --- |
| inputClass | This class will be added to the input box. |
| resultsClass | The class to apply to the results’ container. Default value is "ac\_results" |
| loadingClass | The class to apply to the input box while results are being fetched from the server. Default is “ac\_loading.” |
| lineSeparator | The character used to separate the results. The default is \n |
| minChars | The minimum # number of characters before sending a request to the server. Default is 1. |
| delay | The delay after typing when the request will be sent. Default is 400 ms. |
|  |  |

To set these options, include them in a dictionary as the second argument to the autocomplete method like this:

$("input#city").autocomplete('<%= Url.Action("Find", "City") %>', {

minChars : 3,

delay : 300

});

This type of functionality is immensely useful for selecting from large lists. It keeps your initial page size down by not loading all of these items at once and is user-friendly.

Summary

In this chapter, you learned how to leverage a common jQuery plug-in to add auto-complete behavior to your view. You learned how to respond to ajax requests and created a formatted response that the plug-in can consume. You should now be able to apply this technique to make your applications more responsive and helpful to your users.