7 Using jQuery UI Autocomplete in Django Templates

In this chapter, we will cover ground in two different dimensions. First of all, we will use jQuery UI's autocomplete, with Django templates and views, covering both the server-side and client-side aspects and explore Django Ajax a little more deeply. Second, we will use something more like a real-world process of discovery while we are doing this. That is to say, instead of starting immediately with a finished solution, we will show what it is like to meet roadblocks along the way and still deliver a working project.

In this chapter we will cover:

- jQuery UI's autocomplete and themeroller
- A "progressive enhancement" combobox
- What needs to be done on the server-side and client-side to do the following:
    - Using Django templating to dynamically create elements
    - Client-side event handling to send autocomplete-based selections to the server
    - DOM Level 0 and iframe-based alternatives on the client-side
    - Extending server-side Django Ajax views to handle updates from the client
    - Refining the working solution
- An example of practical problem solving when issues arise

In previous chapters, we have explored how to get a solution working under optimal conditions. Here we'll look at what we can do when tools don't always work.

Adding autocomplete: first attempt

For further development, we will be using a jQuery theme. What specific theme can be used is customizable, but autocomplete and other plugins require *some* theme such as jQuery UI Themeroller provides. jQuery UI Themeroller, which lets you customize and tweak a theme (or just download a default), is available at:

`http://jqueryui.com/themeroller/`

When you have made any customizations and downloaded a theme, you can unpack it under your static content directory. In our `base.html` template, after our site-specific stylesheet, we have added an include to the jQuery custom stylesheet (note that you may download a different version number than we have used here).

```
{% block head_css_site %}<link rel="stylesheet" type="text/css"
href="/static/css/style.css" />
<link rel="stylesheet" type="text/css"
href="/static/css/smoothness/jquery-ui-1.8.2.custom.css" />
{% endblock head_css_site %}
```

We will be using the jQuery UI combobox, which offers a "progressive enhancement" strategy by building a page that will still work with JavaScript off and will be more accessible than building a solution with nothing but Ajax.

Progressive enhancement, a best practice

"Progressive enhancement," in a nutshell, means that as much as possible you build a system that works without JavaScript or CSS, with semantic markup and similar practices, then add appearance with CSS, and then customize behavior with JavaScript. A textbook example of customizing behavior is to make a sortable table which is originally sortable, in Web 1.0 fashion, by clicking on a link in a table header which will load a version of the table sorted by that link. Then the links are "hijaxed" by using JavaScript to sort the table purely by JavaScript manipulations of the page, so that if a user does not have JavaScript on, clicking on the links loads a fresh page with the table sorted by that column, and if the user does have JavaScript, the same end result is achieved without waiting on a network hit. In this case, we mark up and populate a dropdown menu of available entities, which the JavaScript will hide and replace with an autocomplete box. The `option` tags follow a naming convention of *fieldname.id*; all the autocomplete fields are for fields of an entity, and the naming convention is not directly for the benefit of server-side code, but so that an event listener knows which field it has been given a value for.

Here we follow the same basic formula for `department`, `location`, and `reports_to`. We produce a list of all available options. The first entry is for no selected `department`/`location`/`reports_to`; as a courtesy to the user we add `selected="selected"` to the presently selected value so that the form is smart enough to remember the last selected value, rather than defaulting to a (presumably unwanted) choice each time the user visits it.

This much of the code is run once and creates the beginning of the containing paragraph, sets a `strong` tag, and creates the entry for no `department` selected (and selects it if appropriate):

```
<p>Department:
    <strong>
        <select name="department" id="department"
                class="autocomplete">
            <option
            {% if not entity.department.id %}
                selected="selected"
            {% endif %}
            value="department.-1">&mdash; Select &mdash;</option>
```

Then we loop through the list of departments, creating an `option` that has a value of `"department."` followed by the id of the entity provided as a department. Remember that earlier we simply used a list of all entities for departments. If you are interested in tinkering, you could add a checkbox to indicate whether an entity should be considered a `department` or a `reports_to` candidate. (Locations are a different data type, so no paring should be obviously helpful for them.)

```
            {% for department in departments %}
            <option
            {% if department.id == entity.department.id %}
                selected="selected"
            {% endif %}
            value="department.{{ department.id }}">
                              {{ department.name}}
            </option>
          {% endfor %}
```

We then close the `select` and `strong`, and move on to the next line.

```
        </select>
    </strong>
</p>
```

The `location` and `reports_to` are handled similarly:

```
Location:
    <select name="location" id="location" class="autocomplete">
```

We add the default, unselected option:

```
        <option
        {% if not entity.location.id %}
            selected="selected"
        {% endif %}
        value="location.-1">&mdash; Select &mdash;</option>
```

Then we loop through available locations and build their options.

```
        {% for location in locations %}
            <option
            {% if locationi.id == entity.locationi.id %}
                selected="selected"
            {% endif %}
            value="location.{{ location.id }}">
                               {{ location.identifier }}</option>
        {% endfor %}
        </select>
    </p>
```

And likewise in `Reports to`:

```
<p>
Reports to:
    <select name="reports_to" id="reports_to" class="autocomplete">
        <option
        {% if not entity.reports_to.id %}
            selected="selected"
        {% endif %}
        value="reports_to_-1">&mdash; Select &mdash;</option>
        {% for reports_to in reports_to_candidates %}
            <option
            {% if reports_to.id == entity.reports_to.id %}
                selected="selected"
            {% endif %}
            value="reports_to_{{ reports_to.id }}">
                                 {{ reports_to.name }}</option>
        {% endfor %}
    </select>
</p>
```

Our profile page is modified to provide more variables to the template. Later on, we might identify which entities can be an entity's departments and which entities can be reported to, thus improving the available options by paring away irrelevant entities, but for now we simply provide all entities.

```
@login_required
def profile(request, id):
    entity = directory.models.Entity.objects.get(pk = id)
    emails = directory.models.EntityEmail.objects.filter(
             entity__exact = id).all()
    all_entities = directory.models.Entity.objects.all()
    all_locations = directory.models.Location.objects.all()
    return HttpResponse(get_template(u'profile.html').render(Context(
      {
      u'entity': entity,
      u'emails': emails,
      u'departments': all_entities,
      u'reports_to_candidates': all_entities,
      u'locations': all_locations,
      })))
```

This provides nice-looking autocomplete functionality like:

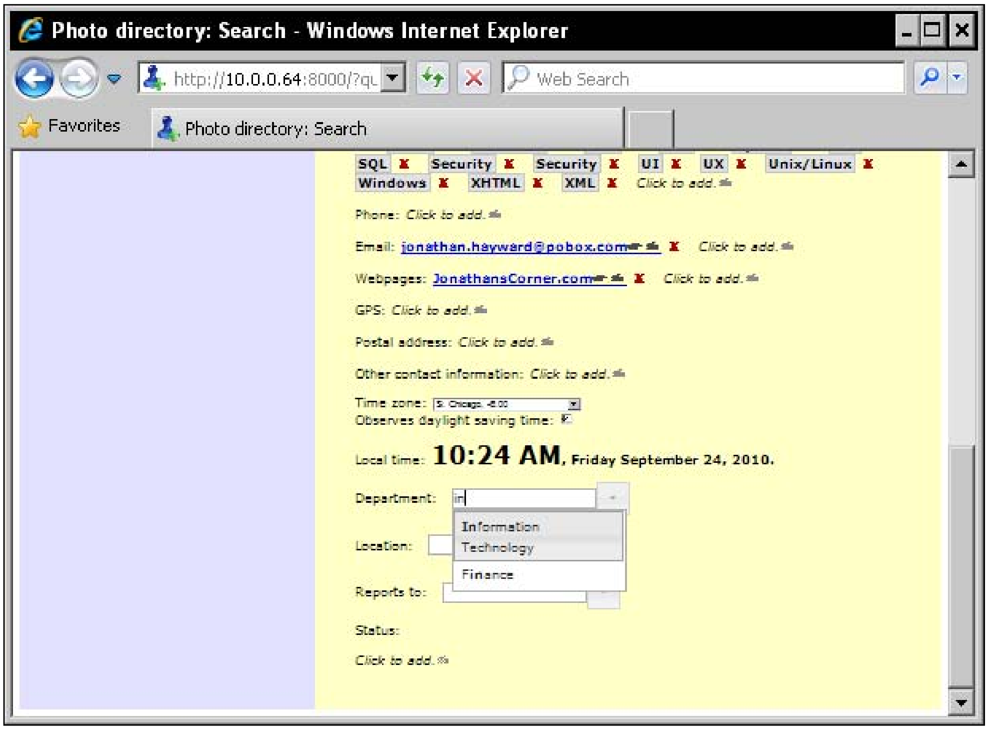
Photo directory: Search - Windows Internet Explorer
http://10.0.0.64:8000/?qu
Web Search
Favorites
Photo directory: Search

SQL Security Security UI UX Unix/Linux
Windows XHTML XML Click to add.

Phone: Click to add.

Email: jonathan.hayward@pobox.com Click to add.

Webpages: JonathansCorner.com Click to add.

GPS: Click to add.

Postal address: Click to add.

Other contact information: Click to add.

Time zone: S. Chicago -6:00
Observes daylight saving time:

Local time: 10:24 AM, Friday September 24, 2010.

Department: in
Information Technology
Finance

Location:

Reports to:

Status:

Click to add.

However, there's a bit of a quirk. When we use the code as implemented earlier, it does not seem to save our changes; they are not persistent across page views.

A real-world workaround

In the course of writing this book, a difficulty was encountered. We will take advantage of that opportunity to look at finding alternatives when things do not work. For those interested in jQuery UI complete specifically, the StackOverflow question at `http://stackoverflow.com/questions/188442/whats-a-good-ajax-autocomplete-plugin-for-jquery` is recommended, and a response with nine upvotes as of this writing said autocomplete doesn't work as of roughly jQuery UI 1.6. `http://pastie.org/362706` is reported as a working version. However, let's look at a workaround when things don't work—because even a recommended plugin for a good library may not work, or may not work for us. However, we will succeed later on working with the provided plugin.

The difficulty in question is as follows. jQuery UI includes autocomplete functionality. As downloaded in our case, the autocomplete functionality appeared to work from the user interface perspective, but did not appear to be saving data on the server. A callback function had been created and (attemptedly) registered, but it was not being called. Inserting `alert()` in the beginning of the function did not trigger any alert boxes, and no error appeared in the browser console. Consulting forums at `http://forum.jquery.com/` and `http://stackoverflow.com/`, which are excellent resources, did not seem to turn up results. A bug was filed at `http://dev.jqueryui.com/`, but that did not generate any response quickly. Supposing that we have deadlines looming, what should we do in a case like this?

"Interest-based negotiation": a power tool for problem solving when plan A doesn't work

At this point we might discuss a tangent from *interest-based negotiation*.

Fisher and Ury's negotiation classic *Getting to Yes* discusses two basic kinds of negotiation: hard and soft. Hard negotiation tries to bend as little as possible from its stated position; soft negotiation, such as often occurs in informal and friendly settings, is much more flexible. But they both suffer from an Achilles heel: when both sides start by defining a position, and the only question is who is going to give how much from their initial positions, the result is almost guaranteed to be suboptimal. If you play that game in the first place, you lose because you are playing the wrong game.

The alternative, interest-based negotiation, which involves finding what interests exist on both sides and doing creative problem solving based on those interests, is much more likely to produce a winner for all involved. *Getting to Yes* discusses interest-based negotiation as a power tool for hostile negotiations where the other side has the upper hand, and perhaps it may be. But some of the best mileage you can get out of interest-based negotiations is in friendly negotiations.

One such situation that keeps coming up at work is when someone has basically figured out what they consider a solution to a problem, and then asks for you to attend to the implementation. For example, a manager might have come to a system administrator in the days when pagers were the hot new thing and said, "Our disk filled up last night! In order to prevent this from happening again, is there any way you can set up an automatic process to send the output of a `df` to a pager every five minutes?" And almost the worst response in that situation is, "Yes, I'll get right on it." In cases like these, if the solution was envisioned, architected, and designed by someone nontechnical, and the technical person is asked only to handle implementation details, the solution is almost guaranteed to be wrong.

The correct solution is not to negotiate on a level of positions, but of interests. This particular solution uses a program whose output is designed for a full terminal window, which would be quite painful to scroll through on an early pager even once. It is a "boy who cried wolf" solution that means to the system administrator, "Here's some spam you have to scroll through every five minutes to find out if there is interesting data." Not, necessarily, that it is wrong to send something to a pager. It might well be an appropriate solution to periodically check and send a brief message to the system administrators if the disk is fuller than some threshold percentage, or if the disk is being filled up beyond some threshold percentage per unit of time. But in many cases the correct response is to politely receive the stated position and then get on to identifying the interests involved and trying as best you can to craft a position.

In this case, we are applying the principle of interest-based negotiation to negotiation with the computer. Our initial position, "Follow the jQuery UI instructions" has not produced the desired results, at least not yet. So the next thing we can do is identify our interests. Our interest here, without degrading anything else, is to provide autocomplete functionality to the user interface. This allows at least four potential ways to get past the obstacle:

- Resolve the problem and complete the intended solution
- Work around the bug using the same framework
- Find another jQuery plugin to handle autocomplete
- Use a standalone solution, or another library

We don't need to stay stuck; in fact, we have several options. The first option cannot be ruled out, but right now we have not succeeded. Let's say that's not an option in our case. The third and fourth possibilities almost certainly have multiple options, and multiple live options, but in this case we can bend the rules a little bit, comment out our event-handling code, and give a little nudge to let our jQuery UI-based solution work with all of the niceties of using jQuery UI. We will go with this workaround, but we are not just a workaround away from being blocked by a brick wall. There are presumably several live options, and the more we think in terms of interests rather than positions, and identify interests to feed into problem solving, the more a brick wall fades into a cornucopia of possibilities.

A first workaround

The workaround, like many workarounds, is itself an example of interest-based negotiation with the computer. What we want to happen, that isn't happening yet, is for the data to be saved. The obvious way for us to save the data is by an `XMLHttpRequest` based call but it is not strictly an interest to say that we go through `XMLHttpRequest` or jQuery for the submission. It would also work to have a form that submitted the same data to an `iframe`. This is not a first choice solution, but we should be much more cautious about ruling out positions altogether than identifying our interests. Now we don't want a spurious `iframe` on the page, but we can set it to `display: none;`, and treat it as a bit bucket, or a Unix `/dev/null`. And the graceful degradation solution provided by jQuery UI uses a `select`, so we can specify an `onchange` for the `select` that will submit the form whenever the `select` is changed, that is whenever an autocomplete value is selected. jQuery UI allows us to display the select, and we will do this, both to allow a person to see available options, and to provide a choice about means of input.

Our revised code is as follows in `static/style.css`:

```
bitbucket, #bitbucket
    {
    display: none;
    }
```

In the top of the `body_main` block in `templates/profile.html` we add:

```
<iframe class="bitbucket" id="bitbucket" name="bitbucket"
       src="about:blank"></iframe>
```

One design consideration from our implementation is that it wraps each of the `select` tags (that populate an autocomplete) in its own form element. This means that semantically we cannot have all of them in different `br` separated lines of the same `p`, but this gives an opportunity to make the design better. If we put each field in its own paragraph, it will make a more readable use of whitespace. So we give each field its own paragraph, wrap the paragraphs with `select` tags in `form` elements which submit via `POST` to our Ajax-geared URL and have a target of `bitbucket`, add a hidden form element which will receive special treatment on the server side as will be discussed later, and specify an `onchange` form submission for `select`. This workaround uses what is informally called "DOM Level 0," and is not a first choice. It does, however, allow us to keep a "close to jQuery" solution, with a workaround that is readily replaced by a more preferable solution if a bug is fixed.

The previous code is largely the same, but is wrapped in a `form` tag, and preceded by a hidden `input` designed to ensure that the view has all the information it needs.

```
<form action="/ajax/save" method="POST" name="department_form"
      id="department_form" target="bitbucket">
    <input type="hidden" name="id"
           value="Entity_department_{{ entity.id }}" />
    <p>Department:
```

The `select` has an `onchange` set to submit the form wrapping it. This kind of approach, informally referred to as DOM Level 0 scripting, is not a first choice solution; it is not exactly semantic markup. But it has been said, "In theory, theory and practice are the same. In practice, theory and practice are different." Especially if you have a deadline, given a choice between purist semantic markup that doesn't work, and a less pure solution that works, choose the solution that works. (But, of course, if you have a choice between a purist solution that works and a less pure solution that works, choose the purist solution that works, even if it involves more learning or more work up front.)

```
    <select name="department" id="department" class="autocomplete"
            onchange="this.form.submit();">
        <option
           {% if not entity.department.id %}
                selected="selected"
            {% endif %}
```

We changed an option value of `&mdash; Select &mdash;`, such as is a common practice for `select` used for their own sake, to `None`. The reason has to do with why we are building this `select`: we wish to populate an autocomplete, and for autocomplete purposes, if someone wants to set a `value` to `None`, it is unlikely that they will start typing "Select".

But these are changes of relatively small details. The select is still essentially like it was before. And that is a benefit: if we are able to get things working in pure semantic markup using best practices, it helps to have a page that's still basically like a "best practices" implementation.

```
                value="department.-1">None</option>
           {% for department in departments %}
                <option
                   {% if department.id == entity.department.id %}
                       selected="selected"
                   {% endif %}
                   value="department.{{ department.id }}">
                          {{ department.name }}</option>
            {% endfor %}
        </select>
    </p>
</form>
```

The other fields are changed, if slightly. They are in their own paragraphs, which is semantically at least as good as having them one per line in a large paragraph, and lends itself to display with better use of whitespace—a major benefit if people will be using our directory a lot!

We define a link for the homepage:

```
<p>
    Homepage:
    {% if entity.homepage %}
        <a href="{{ entity.homepage }}">
    {% endif %}
```

Within the link we define `strong`, which is marked up for (right-click) editing, and display the URL inside the link:

```
    <strong class="edit_rightclick"
       id="entity_homepage_{{ entity.id }}">{{ entity.homepage }}
    </strong>
```

Then we close the link and paragraph:

```
    {% if entity.homepage %}
        </a>
    {% endif %}
</p>
```

We define similar, but not identical, handling for the e-mail:

```
<p>
    Email:
    <strong>
        {% for email in emails %}
            <a id="EntityEmail_email_{{ email.id }}"
               class="edit_rightclick"
               href="mailto:{{ email.email }}">
```

We eliminate some whitespace so that there will not be a space between the link and the separating comma:

```
                {{ email.email }}</a>{% if not forloop.last %},
            {% endif %}
        {% endfor %}
        <span class="edit" id="EntityEmail_new_{{ entity.id }}">
            Click to add email.
        </span>
    </strong>
</p>
```

The `location` fields, as the `reports_to` field in the following snippet, follow the same pattern as the `department` previously seen.

We define a `form`:

```
<form action="/ajax/save" method="POST" name="location_form"
      id="location_form" target="bitbucket">
```

We define a hidden `input` so the server-side code has a parsable identifier:

```
    <input type="hidden" name="id"
           value="Entity_location_{{ entity.id }}" />
    <p>
        Location:
```

We define a `select` with a DOM Level 0 submit:

```
       <select name="location" id="location" class="autocomplete"
          onchange="this.form.submit();">
```

We build the first option for when nothing has been selected:

```
        <option
        {% if not entity.location.id %}
           selected="selected"
        {% endif %}
        value="location.-1">None</option>
```

We populate the rest of the list:

```
            {% for location in locations %}
               <option
                {% if locationi.id == entity.locationi.id %}
                    selected="selected"
                {% endif %}
                value="location.{{ location.id }}">
                       {{ location.identifier }}</option>
            {% endfor %}
```

And we close tags that need to be closed:

```
            </select>
        </p>
    </form>
```

The `Phone` field is an in-place edit field. It works as a regular in-place edit field, not an autocomplete:

```
    <p>
        Phone:
        <strong class="edit" id="entity_phone_{{ entity.id }}">
            {% if entity.phone %}
                {{ entity.phone }}
            {% else %}
                Click to change.
            {% endif %}
        </strong>
    </p>
```

And finally, we have our third and last autocomplete, which works like the first two:

```
    <form action="/ajax/save" method="POST" name="reports_to_form"
          id="reports_to_form" target="bitbucket">
        <input type="hidden" name="id"
               value="Entity_reports_to_{{ entity.id }}" />
        <p>
            Reports to:
           <select name="reports_to" id="reports_to" class="autocomplete"
                   onchange="this.form.submit();">
           <option
           {% if not entity.reports_to.id %}
                selected="selected"
            {% endif %}
                value="reports_to_-1">None</option>
            {% for reports_to in reports_to_candidates %}
                <option
                {% if reports_to.id == entity.reports_to.id %}
                    selected="selected"
                {% endif %}
                value="reports_to_{{ reports_to.id }}">
                       {{ reports_to.name }}</option>
            {% endfor %}
            </select>
        </p>
    </form>
    <p>
        Start date:
        <strong>
            {{ entity.start_date }}
        </strong>
    </p>
```

The handler follows the signature for event handlers registered, as we wish to register them. While we do not use the `event` object, we keep it in the signature. `context.item` should be the `option` that was selected. Its value will look like `department.1` and is meant to carry all the information needed for this function. The data submitted follows the same `id-value` structure as we have already used for in-place editing in the previous chapter, and submits it to the same view. We will in fact have different server-side code handling these selections. The code that stores text in a field will not take an ID and look up the right instance of the intended field, at least not without significant refactoring. However, we are developing with an analogous interface in mind: submitted data in the `id-value` format, with the ID following the *ModelName_fieldname_instanceID* naming convention.

```
function update_autocomplete(event, context)
    {
    var split_value = context.item.value.split(".");
    if (split_value.length == 2 && !isNaN(split_value[1]))
        {
        var field = split_value[0];
        var id = split_value[1];
        $.ajax({
            data:
                {
                id: "Entity_" + field + "_" + {{ entity.id }},
                value: id,
                },
            url: "/ajax/save",
            });
        };
    }
```

Boilerplate code from jQuery UI documentation

It is commonplace when using software to include adding boilerplate code. Here is an example. We insert boilerplate code from the documentation pages for jQuery UI at `http://jqueryui.com/demos/autocomplete/#combobox`:

```
    (function($) {
           $.widget("ui.combobox", {
               _create: function() {
                   var self = this;
                   var select = this.element.hide();
                   var input = $("<input>")
                       .insertAfter(select)
                       .autocomplete({
                           source: function(request, response) {
                               var matcher = new RegExp(request.term,
                                                        "i");
   response(select.children("option").map(function() {
                                   var text = $(this).text();
                                   if (this.value && (!request.term ||
                                            matcher.test(text)))
                                       return {
                                           id: this.value,
                                           label: text.replace(new
   RegExp("(?![^&;]+;)(?!<[^<>]*)(" +
   $.ui.autocomplete.escapeRegex(request.term) +
   ")(?![^<>]*>)(?![^&;]+;)", "gi"), "<strong>$1</strong>"),
                                           value: text
                                       };
                               }));
                           },
                           delay: 0,
                           change: function(event, ui) {
                               if (!ui.item) {
                                   // remove invalid value,
                                   // as it didn't match anything
                                   $(this).val("");
                                   return false;
```

```
                        }
                        select.val(ui.item.id);
                        self._trigger("selected", event, {
                            item: select.find("[value='" +
                                           ui.item.id + "']")
                        });

                    },
                    minLength: 0
                })
                .addClass(
                 "ui-widget ui-widget-content ui-corner-left");
            $("<button>&nbsp;</button>")
            .attr("tabIndex", -1)
             .attr("title", "Show All Items")
            .insertAfter(input)
            .button({
                icons: {
                    primary: "ui-icon-triangle-1-s"
                },
                text: false
            }).removeClass("ui-corner-all")
            .addClass("ui-corner-right ui-button-icon")
            .click(function() {
                // close if already visible
                if (input.autocomplete("widget").is(":visible")) {
                    input.autocomplete("close");
                    return;
                }
                // pass empty string as value to search for,
                // displaying all results
                input.autocomplete("search", "");
                input.focus();
            });
        }
    });

})(jQuery);
```

Turning on Ajax behavior (or trying to)

We make autocompletes out of the relevant selects, and also display the selects, which the `combobox()` call hides by default. The user now has a choice between the select and an autocomplete box.

```
$(function()
    {
    $(".autocomplete").combobox();
    $(".autocomplete").toggle();
```

Here we have code which, from the documentation, might be expected to call the `update_autocomplete()` event handler when a selection or change is made. However, at this point we encounter a bend in the road.

The bend in the road is this: the following commented code, when uncommented, doesn't seem to be able to trigger the handler being called. When it was uncommented, and an `alert()` placed at the beginning of `update_autocomplete()`, the `alert()` was not triggered even once. And the usual suspects in terms of forums and even filing a bug did not succeed in getting the handler to be called. The `alert()`was still not called.

After this code, let's look at our updated code on the server side, and then see how this bend in the road can be addressed.

```
    /*
    $(".autocomplete").autocomplete({select: update_autocomplete});
    $(".autocomplete").bind({"autocompleteselect": update_
autocomplete});
    $(".autocomplete").bind({"autocompletechange": update_
autocomplete});
    */
    });
```

Now let us turn our attention to the server side.

Code on the server side

Here we have the updated `save()` view which has been expanded to address its broader scope. We accept either `GET` or `POST` requests, although requests that alter data should only be made by `POST` for production purposes, and save the dictionary for exploration.

```
@ajax_login_required
def save(request):
    try:
        html_id = request.POST[u'id']
```

```
            dictionary = request.POST
        except:
            html_id = request.GET[u'id']
            dictionary = request.GET
```

If we have one of the autocomplete values, we have a hidden field named `id` which guarantees that any submission will have that field in its dictionary, either `request.POST` or `request.GET`. However, the `department`, `location`, and `reports_to` fields are not all named `value`, and we manually check for them and use `value` as a default:

```
        if html_id.startswith(u'Entity_department_'):
            value = dictionary[u'department']
        elif html_id.startswith(u'Entity_location_'):
            value = dictionary[u'location']
        elif html_id.startswith(u'Entity_reports_to_'):
            value = dictionary[u'reports_to']
        else:
            value = dictionary[u'value']
```

We perform some basic validation. Our code's HTML ID should only consist of word characters:

```
        if not re.match(ur'^\w+$', html_id):
            raise Exception("Invalid HTML id.")
```

Then we handle several special cases before the general-purpose code that handles most `/ajax/save` requests. If there is a new `EntityEmail`, we create it and save it:

```
        match = re.match(ur'EntityEmail_new_(\d+)', html_id)
        if match:
            model = int(match.group(1))
            email = directory.models.EntityEmail(email = value, entity =
              directory.models.Entity.objects.get(pk = model))
            email.save()
            directory.functions.log_message(u'EntityEmail for Entity ' +
              str(model) + u' added by: ' + request.user.username + u',
              value: ' + value + u'\n')
            return HttpResponse(
              u'<a class="edit_rightclick"
                   id="EntityEmail_email_' + str(email.id)
              + u'" href="mailto:' + value + u'">' + value + u'</a>' +
              u'''<span class="edit" id="EntityEmail_new_%s">
                  Click to add email.</span>''' % str(email.id))
```

We have the added code to look up the entity, look up the appropriate `department` (if any), assign the updated `department`, and save the entity. This technique is repeated, with slight variation, for the `location` and `reports_to` fields. We still need to return an `HttpResponse`, even if the value is ignored. On the client-side, the following code we develop will report a Django error page for development purposes but even then will discard the reported value if the update runs without error.

Here we have the special case of an Entity's `departmen`t being set. While we create the illusion on the client-side that this is just the same sort of submission as (say) an Entity's description, we need to handle a few things on the server side to give the client-side a simple appearance of "Mark it up right and save it, and it will be saved."

```
    elif html_id.startswith(u'Entity_department_'):
        entity_id = int(html_id[len(u'Entity_department_'):])
        department_id = int(value[len(u'department.'):])
        entity = directory.models.Entity.objects.get(pk = entity_id)
        if department_id == -1:
            entity.department = None
        else:
            entity.department = directory.models.Entity.objects.get(
                                    pk = department_id)
        entity.save()
        return HttpResponse(value)
```

The `location` code works the same way as `reports_to` and `department`:

```
    elif html_id.startswith(u'Entity_location_'):
        entity_id = int(html_id[len(u'Entity_location_'):])
        location_id = int(value[len(u'location.'):])
        if location_id == -1:
            entity.location = None
        else:
            entity.location = directory.models.Location.objects.get(
                                  pk == location_id)
        entity.save()
        return HttpResponse(value)
    elif html_id.startswith(u'Entity_reports_to_'):
        entity_id = int(html_id[len(u'Entity_reports_to'):])
        reports_to_id = int(value[len(u'reports_to.'):])
        entity = directory.models.Entity.object.get(pk = entity_id)
        if reports_to_id == -1:
            entity.reports_to = None
        else:
            entity.reports_to = directory.models.Entity.objects.get(
                                  pk == reports_to_id)
        entity.save()
        return HttpResponse(value)
```

Although it is last, this is the mainstream and most generic handler, handling the usual cases of text fields supporting in-place editing:

```
    else:
        match = re.match(ur'^(.*?)_(.*)_(\d+)$', html_id)
        model = match.group(1)
        field = match.group(2)
        id = int(match.group(3))
        selected_model = get_model(u'directory', model)
        instance = selected_model.objects.get(pk = id)
        setattr(instance, field, value)
        instance.save()
        directory.functions.log_message(model + u'.' + field +
             u'(' + str(id) + u') changed by: ' +
             request.user.username + u' to: ' + value + u'\n')
        return HttpResponse(escape(value))
```

Refining our solution further

This approach works, but there is room to further clean it up. First of all, we can set things up in the base template so that, for development, Django's informative error pages are displayed; we also set form submissions to `POST` by default. We make a target `div` for the notifications:

```
            {% block body_site_announcements %}
            {% endblock body_site_announcements %}
            {% block body_notifications %}<div
                  id="notifications"></div>
            {% endblock body_notifications %}
```

Then we add, to the `footer_javascript_site` block, a slightly tweaked `send_notifications()`. Compared to our earlier code, instead of delaying five seconds, it delays five seconds plus two milliseconds per character of the message. This does not have a noticeably different effect for normal, short notifications, but it means that if a 60k Django error page is served up, you have more time to inspect the error. We could tweak it further so that above a threshold length, or on some other conditions, the notification is only dismissed by explicitly pressing a button, but we will stop here.

```
<script language="JavaScript" type="text/javascript">
function send_notification(message)
    {
    $("#notifications").html("<p>" + message + "</p>");
    setTimeout("$('#notifications').show('slow').delay(" + (5000 +
      message.length * 2) + ").hide('slow');", 0);
    }
```

Our notifications area has several different messages, not all of which need to be visually labeled as errors, so we move from a red-based styling to one that is silver and grey in `static/css/style.css`:

```
    #notifications
        {
        background-color: #c0c0c0;
        border: 3px solid #808080;
        display: none;
        padding: 20px;
        }
```

We call, on page load, `$.ajaxSetup()` to specify a default error handler, and also specify form submission via `POST`. This will need to be changed for deployment, but together this means that a valuable Django error page is displayed as a notification whenever an Ajax error occurs, which is a kind of "best of both worlds" solution for development.

```
    $(function()
        {
        $.ajaxSetup(
            {
            error: function(XMLHttpRequest, textStatus, errorThrown)
                {
                send_notification(XMLHttpRequest.responseText);
                },
            type: "POST",
            });
        });
    </script>
```

In the profile template, we will remove the containing `form` elements and the hidden inputs, and replace the contents of the `onchange` attributes. We will also rename the original `update_autocomplete()` to `update_autocomplete_handler()`, leaving it available should the originally intended approach work. The new `update_autocomplete()` will make the Ajax call, submitting the same information as the (now) `update_autocomplete_handler()`. We will remove the bit bucket `iframe`, although we leave the CSS in, in case a bit bucket is desired later on.

We are in a position technically to make one big paragraph out of several fields as we did originally, but breaking them into their own paragraphs was a fortunate change, and we will retain it.

The `Department` paragraph now looks like a cross between the previous two entries. It is its own paragraph, but the form is gone. There is an `onchange` attribute set, although its contents are different from earlier. It calls `update_autocomplete()` with an ID following the name convention and the present value of the `select`.

```
<p>Department:
     <select name="department" id="department" class="autocomplete"
      onchange="update_autocomplete(
             'Entity_department_{{ entity.id}}', this.value);">
        <option
        {% if not entity.department.id %}
              selected="selected"
        {% endif %}
         value="department.-1">None</option>
        {% for department in departments %}
            <option
                {% if department.id == entity.department.id %}
                    selected="selected"
                {% endif %}
                value="department.{{ department.id }}">
                       {{ department.name }}</option>
                {% endfor %}
    </select>
</p>
```

The `location` and `reports_to` areas follow suit:

```
<p>Location:
    <select name="location" id="location" class="autocomplete"
      onchange="update_autocomplete('Entity_location_{{ entity.id}}',
                                  this.value);">
    <option
    {% if not entity.location.id %}
        selected="selected"
    {% endif %}
    value="location.-1">None</option>
   {% for location in locations %}
       <option
       {% if location.id == entity.location.id %}
            selected="selected"
       {% endif %}
            value="location.{{ location.id }}">
                  {{ location.identifier }}</option>
   {% endfor %}
    </select>
</p>
```

The `Reports to` field also follows suit:

```
<p>Reports to:
    <select name="reports_to" id="reports_to" class="autocomplete"
      onchange="update_autocomplete(
               'Entity_reports_to_{{ entity.id}}', this.value);">
   <option
   {% if not entity.reports_to.id %}
        selected="selected"
    {% endif %}
    value="reports_to_-1">None</option>
    {% for reports_to in reports_to_candidates %}
        <option
        {% if reports_to.id == entity.reports_to.id %}
            selected="selected"
        {% endif %}
        value="reports_to_{{ reports_to.id }}">
               {{ reports_to.name }}</option>
    {% endfor %}
     </select>
</p>
```

And that's it. We now have a working, and slightly more polished internally, implementation that supports autocomplete and in-place editing like the following.

For the autocomplete:

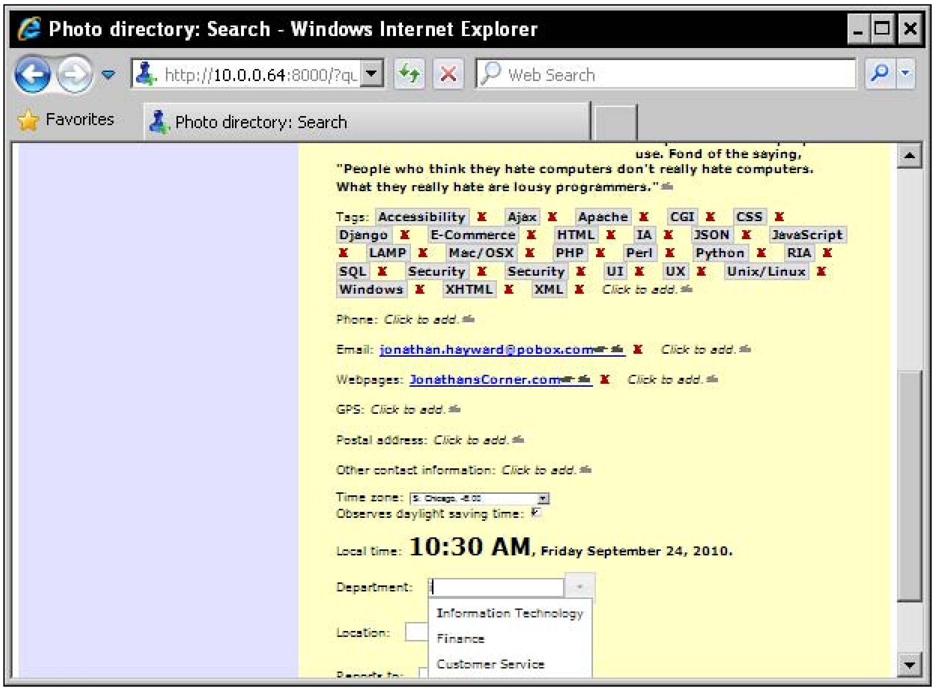
Photo directory: Search - Windows Internet Explorer
http://10.0.0.64:8000/?qu
Web Search
Favorites
Photo directory: Search

use. Fond of the saying, "People who think they hate computers don't really hate computers. What they really hate are lousy programmers."

Tags: Accessibility Ajax Apache CGI CSS Django E-Commerce HTML IA JSON JavaScript LAMP Mac/OSX PHP Perl Python RIA SQL Security Security UI UX Unix/Linux Windows XHTML XML *Click to add.*

Phone: *Click to add.*

Email: jonathan.hayward@pobox.com *Click to add.*

Webpages: JonathansCorner.com *Click to add.*

GPS: *Click to add.*

Postal address: *Click to add.*

Other contact information: *Click to add.*

Time zone: S: Chicago, -6:00

Observes daylight saving time: ✓

Local time: **10:30 AM**, Friday September 24, 2010.

Department:

Information Technology

Finance

Customer Service

Location:

Reports to:

Or, for another of several examples of user input that was allowed, here is an in-place edit used to add a new email address.

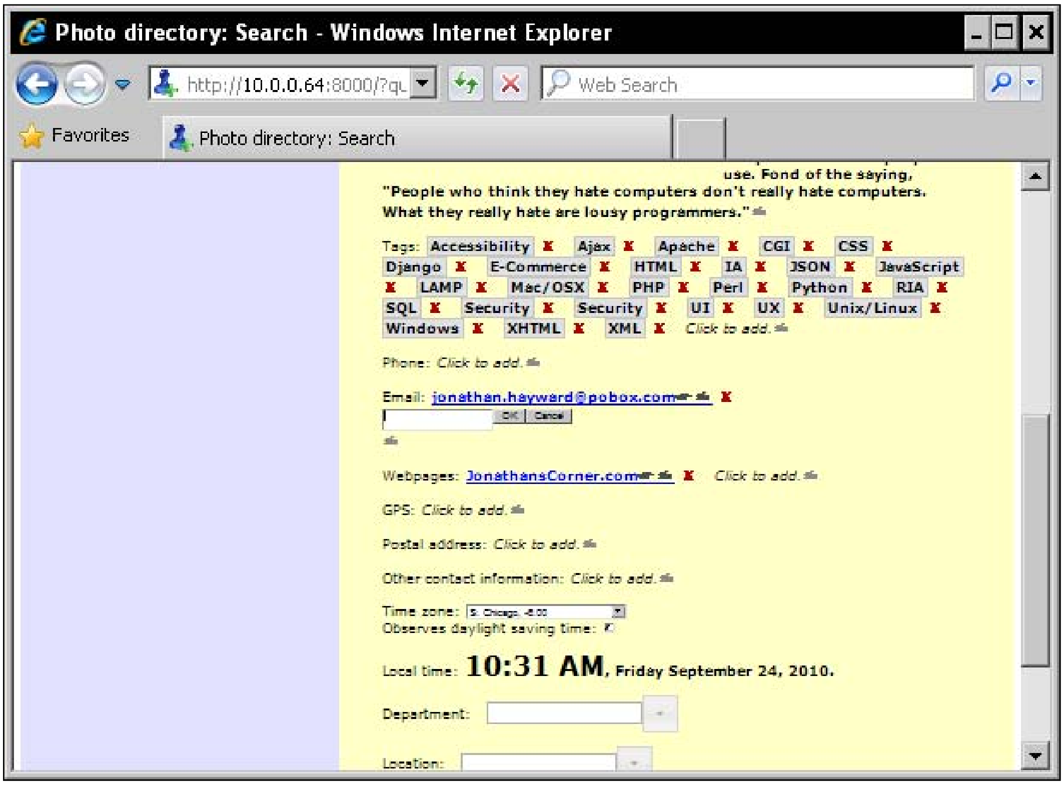
Photo directory: Search - Windows Internet Explorer
http://10.0.0.64:8000/?qu
Web Search
Favorites
Photo directory: Search

use. Fond of the saying,
"People who think they hate computers don't really hate computers.
What they really hate are lousy programmers."

Tags: Accessibility Ajax Apache CGI CSS Django E-Commerce HTML IA JSON JavaScript LAMP Mac/OSX PHP Perl Python RIA SQL Security Security UI UX Unix/Linux Windows XHTML XML *Click to add.*

Phone: *Click to add.*

Email: jonathan.hayward@pobox.com

OK | Cancel

Webpages: JonathansCorner.com *Click to add.*

GPS: *Click to add.*

Postal address: *Click to add.*

Other contact information: *Click to add.*

Time zone: S. Chicago -6.00
Observes daylight saving time:

Local time: **10:31 AM**, Friday September 24, 2010.

Department:

Location:

Summary

Again, we have moved in two different dimensions in this chapter. The first dimension is the obvious one: what we need on the client-side and server-side to get autocomplete working with jQuery UI. The second dimension has to do with creative problem solving when something goes wrong.

We have covered the nuts and bolts of jQuery UI's autocomplete, and where plugins can be obtained. We continue with the concept of "progressive enhancement," and a concrete example. We looked at what tools we have on the server-side and client-side to do this. We have continued to get our hands dirty with Django templating to build the desired pages.

We have covered the ideal case of setting up an event listener that will communicate with the server and keep it updated with pure Ajax, looked at DOM Level 0 and `iframe`-based alternatives on the client side, and further expanded the Django Ajax view on the server side so that it will accommodate autocomplete requests as well as the original edit-in-place requests. Once we had a working solution, we looked at how we could make it work better. We have also taken a cue from best practices in negotiation to look at getting the best from a computer when we can't get what we were first looking for.

In our next chapter, we will look at Django ModelForm or how to easily build forms from Django models. We have a great deal of power and control if we are going to build interfaces, but Django provides some built-in features that can get results quickly. Perhaps, for instance, your organization only has a few locations and it is not the best use of resources to build a fancy in-place edit/autocomplete page for data that is updated maybe once a month? Or perhaps you have one or two main models you need updated, and several more that don't need to be updated that often but do need to be available for editing?

Django ModelForms to the rescue!