9
Database and Search Handling

One early paradox that was discovered in the initial Web 2.0 buzz was that trying to be proactive and be more responsive by being prepared ahead of time ended up making applications *less* responsive, not more. The initial expectation that partial page updates would make more responsive applications met with a surprise when programmers made applications more like a desktop application and, initially at least, less responsive. The recommended best practice now is to be as lazy as possible.

In this chapter, we will:

- Start with a solution that demonstrates the easy AHAH (Asynchronous HTTP And HTML) technique, and plays to Django's strengths
- Demonstrate "graceful degradation" that allows our application to serve up read-only access with JavaScript turned off

Simply put, we will explore a solution with the responsiveness of an "as lazy as possible" solution. Behind the scenes, we will show how that solution showcases Django strengths.

Moving forward to an AHAH solution

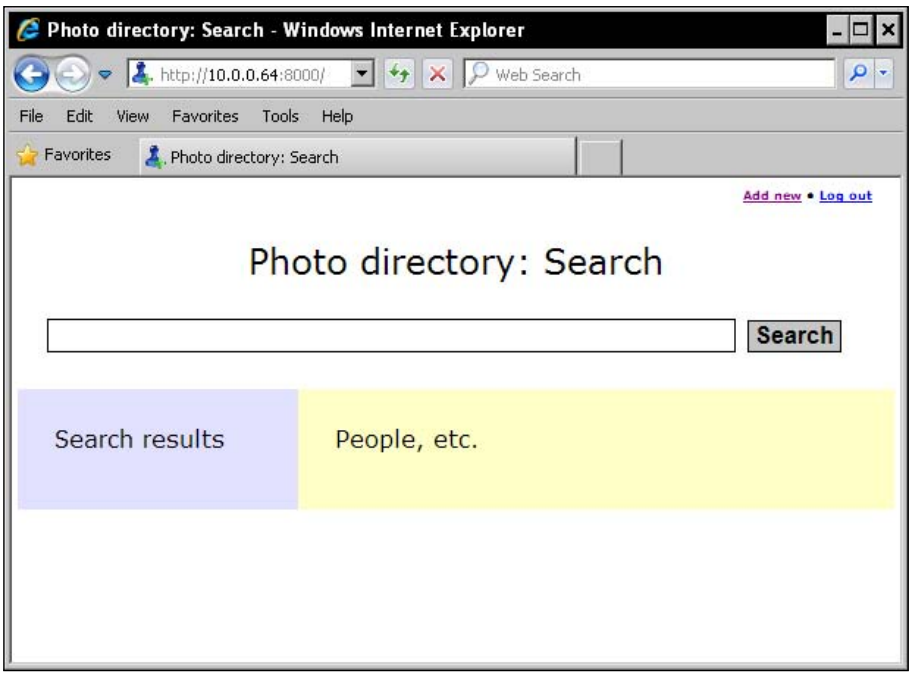
Let's start with a "fewer moving parts" solution that will play to Django's strengths reasonably well. We will move on to a more in-depth solution, but some have said, "Every complex system that works is found to have evolved from a simple system that works," and we can make a relatively simple system that works well before we start trying to push the envelope of what the system can do.

Part of "graceful degradation" means making a solution that works without JavaScript. In our case we will make a modest goal of "read-only" functionality accommodated without JavaScript; that is, we will not be attempting graceful degradation that can update the data without JavaScript on. That job can be done with the Django admin interface, by registering all relevant models as editable, and adding a link, possibly enclosed in `<noscript>` tags, to update the database. While our directory is intended to be easily updated, the main use is read only.

For purposes of illustration, in previous chapters we required an Ajax login before the user could see any data. Now we will attempt a more user-friendly, and realistic, approach in which users may search and see profiles without logging in, but must log in to do anything that makes a change to the data. For organizations that do not want all users to be able to change any data, this offers a slightly finer granularity of access, by creating accounts *only* for people who should be allowed to make changes to the database. Others will have full-fledged access to search and view profiles; they just won't be able to make changes.

We are also updating the model/database schema by making expanded internal use of Entities to serve as Locations. That is, Locations are no longer their own Model; their work is now done by Entities.

When the user first loads the screen, it should look like the following:



When the user has searched and loaded a profile, it should look like the following:

In both cases, the user's attention is drawn to three different areas of the page, each doing a different job. On a live computer, the left **Search results** area is a light blue, and the right **People, etc.** area is a light yellow. One accessibility note is to avoid using color to be the only way a piece of information is delivered, but we are following this guideline in what we are doing here because we are using color to offer additional emphasis to information that is delivered elsewhere. The bold red "✘", which could be replaced by a sprite in a production application for greater compatibility with Internet Explorer, is still a bold "✘" to the colorblind, and the light blue and light yellow differ slightly in value.

Non-colorblind designers wishing to see how something looks to the colorblind, may visit the web page filter at `http://colorfilter.wickline.org/`. In this case, one would expect people with different forms of colorblindness to perceive the slightly different values for the yellow and blue areas as slightly different shades of gray, if not slightly different colors. But even then, their job is to provide an additional underscore that says "Here on the left is one area of the screen doing one job, and here on the right is another area of the screen doing another job."

The breakdown we will use to accomplish this will be one parent template that hosts the search box at the top, and two child templates: a sidebar to the left that lists search results, and a larger template to the right that drills down to the details of one specific Entity. This interface allows users to retain their list of search results while they look at the specific result they have drilled down to at the moment.

Django templates for simple AHAH

"AHAH", or "Asynchronous HTML and HTTP", might be called Ajax lite and is a clean, lightweight variant of Ajax that fetches chunks of HTML and does not include intricate, surgical DOM manipulations.

We will be making three templates:

- A parent template for the whole page
- A template for search results
- A template to drill down to a specific result

The whole page template will be in `search.html` with the search results template in `search_internal.html` and the profile drill down template in `profile_internal.html`.

Templating for a list of search results

The **Search results** sidebar to the left, stored in `search_internal.html` is:

```
<h2>Search results</h2>
```

We also add to the CSS, to prevent wrapping in the middle of the `h2` tags, which we never intend to wrap:

```
h2
    {
    white-space: nowrap;
    }
```

In addition, the perceptive reader may notice that this does not begin by extending `base.html`, or any other template. The reason for this is the template named `search_internal.html`, is a template to render *partial* web pages, and `base.html` is a template to render *whole* web pages. For what we are doing, we don't want a `header`, or a second `body` tag, or various other elements. Partial web page templates will be named `*_internal.html` and will not need to be based on `base.html`.

```
{% if first_portion %}
```

The view that will be using this template will hand it two, possibly empty, lists: `first_portion` and `second_portion`. In lieu of a full paging system, we will display a limited number of results, ten by default and like several other features configurable in the beginning of `settings.py`, and make an Ajax link to show the rest.

In addition, we will make an Ajax feature: when the user "mouses over" a search result, that profile is loaded. This makes for faster, lighter browsing, although old-fashioned clicking on links still works.

If the first portion is nonempty, that is, if the search turned up any results, the template loops through results and populates the query.

```
{% for result in first_portion %}

    <div class="search_result"
      onmouseover="PHOTO_DIRECTORY.load_profile(
                                    {{ result.id }});">
```

The following link is a classic example of a Hijaxed link. There is a link pointing to a URL, in this case RESTful, that quotes the query (escaped in the view), and the ID. The overall application will be designed so that loading that URL will render the desired page. Its `onclick` attribute is set to load the profile in the larger right pane for drilling down to individual results and returns `false` to prevent the default behavior. (One JavaScript gotcha is that you need to specifically return `false` to prevent the default behavior; you cannot return `0` or other "falsy" values and get the same result.)

There is another point to be made about URLs. Django is designed to flexibly allow the creation of attractive URLs like `http://example.com/2/test` where we have an effective URL of `http://example.com/?query=test&id=2`. And this is a carefully chosen feature, since users are more likely to bookmark the former than the latter. However, what we have chosen more directly correlates to "graceful degradation" form submission, and is more straightforward, especially in the case of queries containing characters that would need escaping. If you think you can improve on the "graceful degradation" URL handling, especially in a way that showcases one of Django's deliberately chosen features, go for it.

```
<p><a href="/?query={{ query }}&id={{ result.id }}"
   onclick="PHOTO_DIRECTORY.load_profile({{ result.id }});
   return false;" >{{ result.name }}</a><br />
```

We will cover the CSS later on. The image class, `search_results`, is used to float the remainder of the paragraph that is the Entity's `description`, around the image.

```
        {% if result.image %}
            <img class="search_results"
                 src="/profile/images/{{ result.id }}" />
        {% endif %}
        {% if result.title %}
            {{ result.title }}{% if result.department.name %}
                              ,<br />{% endif %}
        {% endif %}
        {{ result.department.name }}
        </p>
        </div>

    {% endfor %}
```

Having looped through the first portion of the search results, we forestall user puzzlement when a search turns up no results, by explicitly treating them with a simple note telling them why they do not see search results:

```
{% else %}
    <p><em>There were no matches to your search.</em></p>
{% endif %}
```

If the second "overflow" portion is non-empty, we add a JavaScript link (here, one that does not degrade gracefully) to show the additional results, and open a `div` that will initially be hidden, but still deliver them. This means that there will be a very slightly longer initial download for the user, but if the **Show all** link is clicked, it should work immediately without requiring a network request.

```
{% if second_portion %}
    <a class="show_additional_results"
    href="JavaScript:show_additional('results')"
    ><span class="emphasized">+</span> Show all</a>
    <div id="additional_results">
```

Now we loop through the second portion, doing the same thing for each result as with the first:

```
    {% for result in second_portion %}

        <div class="search_result"
          onmouseover="PHOTO_DIRECTORY.load_profile(
                                       {{ result.id }});">
        <p><a href="/?query={{ query }}&id={{ result.id }}"
```

```
            onclick="PHOTO_DIRECTORY.load_profile({{ result.id }});
                     return false;">{{ result.name }}</a><br />
            {% if result.image %}
                <img class="search_results"
                     src="/profile/images/{{ result.id }}" />
            {% endif %}
            {% if result.title %}
                {{ result.title }}{% if result.department.name %}
                                   ,<br />{% endif %}
            {% endif %}
            {{ result.department.name }}
            </p>
            </div>

        {% endfor %}
```

We close the initially hidden `div`, and that's it for the template:

```
        </div>
    {% endif %}
```

Template for an individual profile

Now we will be looking at how we keep the right, "drilldown individual profile" pane up to date. It is possible to surgically update individual elements within the `div`, and our code supports this to some degree. Jeditable is intended to do just that with in-place edits. If we are going to spend a lot of time making a micro-optimized solution, the first choice response may be to replace as little of the page as possible.

However, there are distinct advantages to a "fewest moving parts" solution that simply replaces the "drilldown individual profile" with a fresh pane generated by the server. The generated pane can be generated in a way that more directly plays to Django's strengths. There are major advantages to debugging also, fewer moving parts means fewer places bugs can creep in, and fewer places for them to hide because when there is something wrong, it will often be more obviously wrong, and "more obviously wrong" is much easier to debug than "sometimes appears faintly wrong."

`profile_internal.html` contains the line:

```
    <h2>People, etc.</h2>
```

There is a link, which will be styled, to delete an Entity so that the interface supports full Create, Read, Update, and Delete operations.

```
<div class="deletion">
<span class="delete" id="Entity_{{ id }}">
Delete {{ entity.name }}</span>
</div>
```

Here we have a container, in this case an `h2`, that is marked up for in-line editing with Jeditable (`http://www.appelsiini.net/projects/jeditable`), by adding the class `edit`. It follows the naming convention: "Entity" and then the field ("name"), and then the entity's ID:

```
<h3 id="Entity_name_{{ id }}" title="Click to edit."
class="edit">{{ entity.name }}</h3>
```

The `div` uses the `ajaxFileUpload` plugin (`http://www.phpletter.com/Our-Projects/AjaxFileUpload/`). It displays an image, if there is one, and provides an Ajax file-uploading facility. It depends on the `ajaxFileUpload` code further down.

```
<div class="image">
    <form id="image_upload" name="image_upload"
      action="/ajax/saveimage/{{ id }}" method="POST"
      enctype="multipart/form-data">
        {% if entity.image_mimetype %}
            <img class="profile" src="/profile/images/{{ id }}" />
        {% else %}
            <div id="image_slot"></div>
            Upload image:<br />
        {% endif %}
        <button class="button" id="upload"
          onclick="return ajaxFileUpload({{ id }});">Upload</button>
    </form>
</div>
```

Now, with slight variations, we will go to Jeditable-based fields, those having CSS class `edit`, `edit_rightclick`, or `edit_textarea`, interspersed with deletion fields (`&#10008;` is the symbol "✘") handled by our own Ajax. First comes the title and description:

```
<p>Title: <strong id="Entity_title_{{ id }}" title="Click to edit."
  class="edit">{{ entity.title }}</strong></p>
<p>Description: <strong id="Entity_description_{{ id }}"
  title="Click to edit." class="edit_textarea">
{{ entity.description }}</strong></p>
```

After that come several similar, not particularly **DRY** (**Don't Repeat Yourself**, a Django mantra) sections. We iterate through all existing telephone numbers, displaying a **Click to edit** field with tooltip, and adding the deletion marker:

```
<p>Phone: <strong>
    {% for phone in phones %}
        <span id="Phone_{{ phone.id }}" class="edit"
          title="Click to edit.">{{ phone.number }}</span>
        <span class="delete"
        id="Phone_{{ phone.id }}">&#10008;</span>  
    {% endfor %}
```

Then, after the loop, we add a hook to add a new phone number:

```
    <span class="edit" title="Click to add."
          id="Phone_new_{{ id }}"></span>
</strong></p>
```

E-mails and URLs are handled basically the same way, but `Click to edit` the way we did before would be undesirable because it would block the e-mail addresses and URLs functioning as a live link (which is how we would normally want them to function). Instead of using a regular click, we use the right click as the hook for editing. The CSS is similar but noticeably different in the marker it appends:

```
<p>Email: <strong>
    {% for email in emails %}
    <a id="Email_{{ email.id }}" class="edit_rightclick"
      title="RIGHT click to edit." href="mailto:{{ email.email }}"
      >{{ email.email }}</a>
    <span class="delete" id="Email_{{ email.id }}">&#10008;
    </span>  
    {% endfor %}
    <span class="edit" title="Click to edit."
          id="Email_new_{{ id }}"></span>
</strong></p>

<p>Webpages: <strong>
    {% for url in urls %}
        <a id="URL_url_{{ url.id }}" class="edit_rightclick"
          title="RIGHT click to edit."
          href="{{ url.url }}">{{ url.url }}</a>
        <span class="delete" id="URL_{{ url.id }}">&#10008;
        </span>  
    {% endfor %}
    <span class="edit" title="Click to add."
          id="URL_new_{{ id }}"></span>
</strong></p>
```

The GPS coordinates are handled by a live link; Google maps will search by GPS coordinates, among many other kinds of queries, so this is a map link:

```
<p>GPS: <strong>
    {% if gps %}
        <a class="edit_rightclick" id="Entity_gps_{{ id }}"
          href="{{ gps_url }}">{{ gps }}</a>
    {% else %}
        <span class="edit" id="Entity_gps_{{ id }}"></span>
    {% endif %}
</strong></p>
```

Let's not forget the postal address:

```
<p>Postal address:
    <strong class="edit_textarea" title="Click to edit."
      id="Entity_postal_address_{{ id }}">
         {{ entity.postal_address }}</strong></p>
```

In terms of being future-proof and addressing business considerations, it can be very valuable to have a "catch-all" slot for other kinds of contact information. Right now there are existing forms of contact information that have not been listed, for instance a Skype ID, or IM addresses that are not identical to an e-mail address. It would be a very careless assumption to assume that the forms of contact information we have now are all we'll ever need. We can't guarantee a future-proof solution but listing a few forms of contact information that will probably stay around, and being sure to include a catch-all/wildcard slot, is almost certainly less wrong for the future than only listing what our organization uses today.

```
<p>Other contact information:
    <strong class="edit_textarea" title="Click to edit."
      id="Entity_other_contact_{{ id }}">{{ entity.other_contact }}
    </strong></p>
```

Tagging is a powerful value-added feature for a directory and is included in searches; it also provides an example of a many-to-many relationship. We iterate through an entity's tags in basically the same way we iterated through phone numbers, e-mail addresses, and URLs:

```
<p>Tags: <strong>
    {% for tag in tags %}
        <span class="tag">{{ tag.text }}</span>
        <span class="delete"
        id="tag_{{ id }}_{{ tag.id }}">&#10008;</span> 
    {% endfor %}
    <span class="edit" id="Entity_tag_new_{{ id }}"></span>
</strong></p>
```

For the `Department`, which is an Entity, we display its name as a hijaxed link to that Entity's profile:

```
<p>Department: <strong>
    <a href="/?query={{ query }}&id={{ entity.department.id }}"
      onclick="load_profile({{ entity.department.id }});
      return false;">{{ entity.department.name }}</a>  
```

We build up a `select`, and it is important here that the `select` defaults to the currently selected value and not whatever we happen to place first in our list. The `onchange` calls a function that saves the change and refreshes the profile:

```
    <select name="department" id="department" class="autocomplete"
      onchange="PHOTO_DIRECTORY.update_autocomplete(
            'Entity_department_{{ id }}', 'department');">
```

First we manually create the `option` for no selected value:

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

Then we loop through the available entities for the options.

We may note as a known issue, and possible area for future enhancements, that the list of Entities is not trimmed to only present departments, locations, or bosses when those `select` elements are populated. It would be possible to add a checkbox below the `department/location/reports_to` paragraph saying that this Entity is available to populate those fields, but for now we will leave that as future room for enhancement.

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

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

```
<p>Location: <strong>
    <a href="/?query={{ query }}&id={{ entity.location.id }}"
            onclick="load_profile({{ entity.location.id }});
            return false;">{{ entity.location.name }}</a>  
    <select name="location" id="location" class="autocomplete"
      onchange="PHOTO_DIRECTORY.update_autocomplete(
               'Entity_location_{{ id }}', 'location');">
        <option
          {% if not entity.location %}
             selected="selected"
          {% endif %}
          value="location.-1">None</option>
        {% for location in entities %}
        <option
          {% if location.id == entity.location %}
              selected="selected"
          {% endif %}
          value="location.{{ location.id }}">
          {{ location.name }}</option>
        {% endfor %}
</select></strong></p>

<p>Reports to: <strong>
    <a href="/?query={{ query }}&id={{ entity.reports_to.id }}"
      onclick="load_profile({{ entity.reports_to.id }});
      return false;">{{ entity.reports_to.name }}</a>  
    <select name="reports_to" id="reports_to" class="autocomplete"
      onchange="PHOTO_DIRECTORY.update_autocomplete(
              'Entity_reports_to_{{ id }}', 'reports_to');">
        <option
          {% if not entity.reports_to %}
             selected="selected"
          {% endif %}
          value="reports_to.-1">None</option>
        {% for reports_to in entities %}
        <option
          {% if reports_to.id == entity.reports_to %}
              selected="selected"
          {% endif %}
          value="reports_to.{{ reports_to.id }}">
          {{ reports_to.name }}</option>
        {% endfor %}
</select></strong></p>
```

The `status` field is used as in Facebook or Twitter, as we have discussed earlier. This could be a very useful feature, but if it doesn't make sense in your corporate environment, then it should be disabled. It may be that this is the one field that consistently makes the database grow, and optimization concerns would eschew such fields, but we should recall Donald Knuth's observation, "Premature optimization is the root of all evil." And we should remember that our usual guesses before investigation and research, about why a system is having performance issues, are rarely correct. If this is an example of a feature that is not needed for performance reasons, this should be an observation corroborated by experiment.

We break the statuses up as we broke up search results, although with initial configuration we are only displaying the five most recent statuses:

```
<p>Status:</p>
<div class="edit_textarea" id="Status_new_{{ id }}"></div>
{% for status in first_statuses %}
<p>{{ status.text }}<br />
<span class="timestamp">{{ status.datetime }}</span></p>
{% endfor %}
{% if second_statuses %}
    <p><a class="show_additional_statuses"
    href="JavaScript:show_additional('statuses');"
    ><span class="emphasized">+</span> Show all</a></p>
    <div class="additional_statuses">
        {% for status in second_statuses %}
            <p>{{ status.text }}<br />
            <span class="timestamp">{{ status.datetime }}</span></p>
        {% endfor %}
    </div>
{% endif %}
```

Here we add CSS overrides, enclosed in a `noscript` tag. The CSS styling adds (dingbat) symbols after editable areas as a visual reminder that they can be edited by clicking; but if JavaScript is turned off, so is that functionality. So this little snippet removes the symbols:

```
<noscript>
    <style type="text/css">
        <!--

            .edit:after, .edit_textarea:after, .edit_rightclick:after
                {
                content: "";
                }

        // -->
    </style>
</noscript>
```

Views on the server side

We have the following views to service requests on the server side.

Telling if the user is logged in

We make a view to allow Ajax to test if the user is logged in. Because we have the `@ajax_login_required` decoator, we only need to write code to unconditionally state that the user is logged in, and then guard the view using the `@ajax_login_required` decorator:

```
@ajax_login_required
def ajax_check_login(request):
    output = json.dumps({ u'not_authenticated': False })
    return HttpResponse(output, mimetype = u'application/json')
```

A view to support deletion

The following view allows Ajax deletion of e-mail addresses, URLs, and so on, as well as Entities.

```
@ajax_login_required
def ajax_delete(request):
```

The many-to-many relationship in tagging requires special treatment:

```
    if request.POST[u'id'].lower().startswith(u'tag_'):
        search = re.search(ur'[Tt]ag_(\d+)_(\d+)',
        request.POST[u'id'])
        if search:
            entity = directory.models.Entity.objects.get(
              id = int(search.group(1)))
            entity.tags.remove(directory.models.Tag.objects.get(
              id = int(search.group(2))))
            entity.save()
```

Everything else, that is, every deletion of a model instance (including e-mail addresses and URLs as well as Entities), can be handled by the same method:

```
    else:
        search = re.search(ur'(.*)_(\d+)', request.POST[u'id'])
        if search:
            getattr(directory.models, search.group(1)).objects.get(
              id = int(search.group(2))).delete()
```

In particular here, we log who made a deletion:

```
directory.functions.log_message(u'Deleted: ' +
  request.POST[u'id'] + u' by ' + request.user.username + u'.')
```

The response, which may feed back into the slot in the page, is an empty string:

```
return HttpResponse(u'')
```

These changes require an addition to the `urlpatterns` in `urls.py`. The full `urlpatterns` example given for the hybrid solution will work for the AHAH solution as well.

The AHAH view to load profiles

The AHAH view to load a profile returns a rendered XHTML fragment, not JSON. It is fairly simple:

```
def ajax_profile(request, id):
```

First it looks up and/or calculates values needed by the template:

```
entity = directory.models.Entity.objects.filter(id = int(id))[0]
if entity.gps:
    gps = entity.gps
elif entity.location and entity.location.gps:
    gps = entity.location.gps
else:
    gps = u''
if gps:
    gps_url = \
      u'http://maps.google.com/maps?f=q&source=s_q&hl=en&q=' \
      + gps.replace(u' ', u'+') + "&iwloc=A&hl=en"
else:
    gps_url = u''
```

Then it feeds those values to the template and renders it to a returned response:

```
return render_to_response(u'profile_internal.html',
    {
    u'entities': directory.models.Entity.objects.all(),
    u'entity': entity,
    u'first_statuses':
      directory.models.Status.objects.filter(
        entity = id).order_by(
          u'-datetime')[:directory.settings.INITIAL_STATUSES],
    u'gps': gps,
```

```
            u'gps_url': gps_url,
            u'id': int(id),
            u'emails': directory.models.Email.objects.all(),
            u'phones': directory.models.Phone.objects.all(),
            u'second_statuses':
              directory.models.Status.objects.filter(
                entity = id).order_by(
                  u'-datetime')[directory.settings.INITIAL_STATUSES:],
            u'tags': entity.tags.all().order_by(u'text'),
            u'urls': directory.models.URL.objects.all(),
            })
```

Helper functions for the AHAH view for searching

First, we define a function to count, case-insensitive, the number of times a token appears in a string. The matching is whole word matching, so the string "they looked" would be matched by the query "they" or "looked" but not "look." We put this in `functions.py`, which now needs to have `import re` in its `import` section. The string to search for is the first argument, and the token to search for is the second argument:

```
def count_tokens(raw, query):
    result = 0
    try:
```

Note that this regular expression splits more or less on a non-word character, but a hyphen is treated as a word character. This means, in particular, that tagging, which uses this function, will count "customer-service" as one token, not two.

```
            tokens = re.split(ur'(?u)[^-\w]', raw)
        except TypeError:
            tokens = raw
        while u'' in tokens:
            tokens.remove(u'')
        try:
            matches = re.split(ur'(?u)[^-\w]', query)
        except TypeError:
            matches = query
        while u'' in matches:
            matches.remove(u'')
        for token in tokens:
            for match in matches:
                if token.lower() == match.lower():
                    result += 1
        return result
```

In addition, we define a weighted scoring function designed to estimate the relevance of an Entity to a query. If a search term appears in an Entity's name, there will be a very high score; if an Entity is tagged with the term, it will also be a high score, but lower; if it appears in the Entity's statuses, it will have some relevance but not nearly as high of a score.

The weighted scoring function is as follows. The scoring algorithm is simple: for the Entity provided, count how many times a keyword matches a relevant field associated with the Entity, and multiply it by a weight defined in `settings.py`. This provides a way to try to present the best matches first. It also resides in `functions.py`.

```
def score(entity, keywords):
    result = 0
    if entity.name:
```

In this case, long lines are broken up by adding a backslash (\) as the last character before a line is broken; the indentation by two spaces more is a matter of convention more than requirement, but makes for readability. There are some cases where a line may be broken without a backslash, such as where a parenthesis, curly brace, or square brace has been opened but not closed, in which case a backslash is permitted but unnecessary:

```
result += count_tokens(entity.name, keywords) * \
          directory.settings.NAME_WEIGHT
    if entity.description:
      result += count_tokens(entity.description, keywords) *\
          directory.settings.DESCRIPTION_WEIGHT
    if entity.tags:
        for tag in entity.tags.all():
            result += count_tokens(tag.text, keywords) * \
              directory.settings.TAG_WEIGHT
    if entity.title:
        result += count_tokens(entity.title, keywords) * \
          directory.settings.TITLE_WEIGHT
    if entity.department:
        result += count_tokens(entity.department.name, keywords) * \
          directory.settings.DEPARTMENT_WEIGHT
    if entity.location:
        result += count_tokens(entity.location.name, keywords) * \
          directory.settings.LOCATION_WEIGHT
    for status in directory.models.Status.objects.filter(
      entity = entity.id):
        result += count_tokens(status.text, keywords) * \
          directory.settings.STATUS_WEIGHT
    return result
```

We have added, towards the top of our `settings.py`, the scoring weights and other custom settings for our photo directory:

```
DEBUG = True
TEMPLATE_DEBUG = DEBUG

DIRNAME = os.path.dirname(__file__)

# These are constants used in the template.
DELAY_BETWEEN_RETRIES = 1
INITIAL_RESULTS = 10
INITIAL_STATUSES = 5
SHOULD_DOWNLOAD_DIRECTORY = 1 # 1 or 0, BUT NOT True or False
SHOULD_TURN_OFF_HIJAXING = 0 # 1 or 0, BUT NOT True or False
# These are weightings used to determine importance in searches.
# The values provided are integer clean, but the code should work for
the most
# part with floating point values.
DEPARTMENT_WEIGHT = 30
DESCRIPTION_WEIGHT = 30
LOCATION_WEIGHT = 10
NAME_WEIGHT = 70
STATUS_WEIGHT = 1
TAG_WEIGHT = 50
TITLE_WEIGHT = 50
```

This concludes what we add to `settings.py` to customize behavior of our application.

An updated model

Before we move on, our Entity model has evolved; now would be a good time to give the new Entity model. The changes are mostly straightforward, some fields have been deleted and others added, and we now have a `ManyToManyField`:

```
class Entity(models.Model):
    active = models.BooleanField(blank = True)
    department = models.ForeignKey(u'self', blank = True,
      null = True, related_name = u'member')
    description = models.TextField(blank = True)
    gps = GPSField()
    image_mimetype = models.TextField(blank = True, null = True)
    location = models.ForeignKey(u'self', blank = True, null = True,
      related_name = u'occupant')
    name = models.TextField(blank = True)
    other_contact = models.TextField(blank = True)
    postal_address = models.TextField(blank = True)
    publish_externally = models.BooleanField(blank = True)
    reports_to = models.ForeignKey(u'self', blank = True,
```

```
        null = True, related_name = u'subordinate')
    start_date = models.DateField(blank = True, null = True)
    tags = models.ManyToManyField(Tag)
    title = models.TextField(blank = True)
```

And that is the last thing before our AHAH search function.

An AHAH server-side search function

Without further ado, here is the Ajax search function. We determine the tokens in the query first:

```
def ajax_search(request):
    try:
        query = request.POST[u'query']
    except KeyError:
        try:
            query = request.GET[u'query']
        except KeyError:
            return HttpResponse(u'')
    tokens = re.split(ur'(?u)[^-\w]', query)
    while u'' in tokens:
        tokens.remove(u'')
```

Then we compile a list of candidates for inclusion in the search results. A candidate, in this context, is a two-item list containing an Entity and a starting relevance score of zero. Because we are trying to preserve Unicode sensitivity, and not all database backends play nice with Unicode case-insensitive regular expression matching, we handle all of this in Python, instead of dancing around database backend incompatibilities or needlessly restricting the reader's choices in database backends.

```
    candidates = []
    for candidate in directory.models.Entity.objects.all():
        candidates.append([candidate, 0])
```

Then, for each token, we weed out the candidates that don't match; that is, we have AND-based and not an OR-based handling of query terms. If a candidate does make the cut, its total score is increased by the total for that token.

```
    for token in tokens:
        new_candidates = []
        for candidate in candidates:
            if directory.functions.score(candidate[0], token) > 0:
                candidate[1] += directory.functions.score(
                      candidate[0], token)
                new_candidates.append(candidate)
        candidates = new_candidates
```

The candidates are sorted by score, from highest to lowest. We use an anonymous function, a `lambda` function, which returns the negative of `cmp` of the first item in an array: `cmp()` returns `1` if the first argument is greater (numerically or alphabetically), `0` if they are equal or equivalent, and `-1` if the second argument is greater. We pass a `sort` argument that requests sorting from highest to lowest score. `candidates.sort()` can be passed a named or anonymous function, as long as the function takes two arguments and returns a comparison such as can be used to sort from lowest to highest (or highest to lowest).

```
candidates.sort(lambda a, b: -cmp(a[1], b[1]))
```

The data structure we hand off to the template is not our raw list of candidates; rather, we extract the `description`, `id`, whether or not an image should be displayed, and the `name` of the candidate.

```
export = []
for candidate in candidates:
    if candidate[0].image_mimetype:
        image = True
    else:
        image = False
    export.append(
        {
        u'department': name,
        u'description': candidate[0].description,
        u'id': candidate[0].id,
        u'image': image,
        u'name': candidate[0].name,
        u'title': candidate[0].title,
        })
```

Without tackling a full pagination solution, we display a configurable number of initial results in one area of the template, and then put the others in a hidden `div` that is displayed if the user clicks **Show all**.

```
first_portion = export[:directory.settings.INITIAL_RESULTS]
second_portion = export[directory.settings.INITIAL_RESULTS:]
return render_to_response(u'search_internal.html',
    {
    u'first_portion': first_portion,
    u'second_portion': second_portion,
    u'query': urllib.quote(query),
    })
```

Handling the client-side: A template for the main page

This template extends `base.html`, which contains all the JavaScript includes. This page contains a small amount of boilerplate JavaScript that goes with jQuery plugins, but is mostly custom JavaScript for our application.

We limit ourselves to one incursion into the global namespace above jQuery, namely `PHOTO_DIRECTORY`. We enclose our jQuery-using code in:

```
jQuery(function($)
    {
    /* Code enclosed here. */
    });
```

This allows us to freely reference `jQuery` as `$`, but still create code that can coexist with other libraries that use the `$` identifier for their own purposes. Why are we doing this in a test setup where no other libraries want to use `$`? There is a martial arts maxim, which could just as well be taken from musical instrument performance: "The way you practice is the way you will fight." The only time a martial artist or a musician really has a live choice about how well to perform is in practice; in the heat of performance the decision has already been made. So we follow this best practice, now while we have a choice about it, and not in a project when we will confidently use `$` and incur the wrath of another library. While we will be placing everything under the `PHOTO_DIRECTORY` namespace, we will not be using closures to try to lock down unintended access. If other people want to incorporate this code and use it for other purposes without editing its internals, we are leaving the door open.

We will be following a Whitesmith brace style, meaning that in code written from scratch we will indent four spaces and indent braces along with the code they go with:

```
function log(message)
    {
    try
        {
        console.log(message);
        }
    catch(error)
        {
        alert(message);
        }
    }
```

We will, when possible, place functions and lists of variable assignment in alphabetical order. The more functions or objects you have in a file (or the more functions you have in an object), the easier it is to find things if they are arranged in alphabetical order, or as alphabetical an order as constraints allow.

One surprising feature of this application, and one that is not generally expected, is that in this version all top-level page renderings come from a single template, `search.html`. This is an unexpected simplicity; ordinarily one would be managing multiple templates for different top-level page renderings, and Django template inheritance would be a more potent advantage. However, we will still see an example of template inheritance.

The `base.html` template now reads as follows. We open by defining an XHTML 1.0 `doctype` and `html` tag:

```
{% block dtd %}
  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
{% endblock dtd %}
{% block html_tag %}
  <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-US"
  lang="en-US">
{% endblock html_tag %}
```

Then we open the `head`. In some cases we define actual tags within hooks; in other cases we define hooks, the goal being to have more hooks rather than less to override.

```
{% block head %}<head>
    <title>{% block head_title %}{{ page.title }}
    {% endblock head_title %}</title>
    {% block head_favicon %}<link rel="icon"
    href="/static/favicon.ico" type="x-icon" />
    <link rel="shortcut icon" href="/static/favicon.ico"
    type="x-icon" />{% endblock head_favicon %}
    {% block head_meta %}
        {% block head_meta_author%}
        {% endblock head_meta_author %}
```

We specify a charset of `UTF-8`; without it, many Unicode characters are displayed inappropriately, and user-provided Unicode content will be garbled:

```
        {% block head_meta_charset %}
          <meta http-equiv="Content-Type"
          content="text/html; charset=UTF-8" />
        {% endblock head_meta_charset %}
```

```
            {% block head_meta_contentlanguage %}
              <meta http-equiv="Content-Language" value="en-US" />
            {% endblock head_meta_contentlanguage %}
```

We then define several tags, the meta description, keywords, and possibly refresh and robots tags are important even if they are not defined in this template. If search engine friendliness is at all a concern, you will want pages to define the meta description tag and possibly others.

```
            {% block head_meta_description %}
            {% endblock head_meta_description %}
            {% block head_meta_keywords %}
            {% endblock head_meta_keywords %}
            {% block head_meta_othertags %}
            {% endblock head_meta_othertags %}
            {% block head_meta_refresh %}
            {% endblock head_meta_refresh %}
            {% block head_meta_robots %}
            {% endblock head_meta_robots %}
        {% endblock head_meta %}
        {% block head_rss %}{% endblock head_rss %}
```

The best practice is to have one HTML or XHTML page, with one CSS hit at the top, and one JavaScript hit at the bottom. For development purposes, we do not yet concatenate and minify down to one hit. We define section and page CSS files for development; the principle again is that it is better to have more hooks than less.

```
        {% block head_css %}
            {% 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 %}
            {% block head_css_section %}
            {% endblock head_css_section %}
            {% block head_css_page %}{% endblock head_css_page %}
        {% endblock head_css %}
        {% block head_section %}{% endblock head_section %}
        {% block head_page %}{% endblock head_page %}
    </head>{% endblock head %}
```

After adding more `header` hooks, we define multiple hooks in the main body of the page:

```
{% block body %}<body>
    {% block body_preamble_site %}
    {% endblock body_preamble_site %}
    {% block body_preamble_section %}
    {% endblock body_preamble_section %}
    {% block body_preamble_page %}
    {% endblock body_preamble_page %}
    <div id="sidebar">
    {% block body_sidebar %}{% endblock body_sidebar %}
    </div>
    <div id="content">
    {% block body_content %}
        <div id="header">
        {% block body_header %}
            {% block body_header_banner %}
            {% endblock body_header_banner %}
            {% block body_header_title %}<h1>
              {{ page.title }}</h1>
            {% endblock body_header_title %}
            {% block body_header_breadcrumb %}
              {{ page.breadcrumb }}
            {% endblock body_header_breadcrumb %}
        {% endblock body_header %}
        </div>
        {% block body_site_announcements %}
        {% endblock body_site_announcements %}
        {% block body_notifications %}
          <div id="notifications"></div>
        {% endblock body_notifications %}
        {% block body_main %}{% endblock body_main %}
    {% endblock body_content %}
```

Having placed those hooks in the `body` content, we define a `footer`. We leave room for a `breadcrumb` in two places, at the top and the bottom of the page; from a usability standpoint a `breadcrumb` is a minor feature but if it makes sense to provide one, you should at least leave the door open to one.

```
    </div>
    <div id="footer">
    {% block body_footer %}
        {% block body_footer_breadcrumb %}
          {{ page.breadcrumb }}
```

```
            {% endblock body_footer_breadcrumb %}
            {% block body_footer_legal %}
            {% endblock body_footer_legal %}
        {% endblock body_footer %}
        </div>
    </body>{% endblock body %}
```

We define a logical `footer`. In development we leave separate files for what we will concatenate and minify for development.

```
    {% block footer %}
        {% block footer_javascript %}
            {% block footer_javascript_site %}
              <script language="JavaScript" type="text/javascript"
                src="/static/js/jquery.js"></script>
             <script language="JavaScript" type="text/javascript"
               src="/static/js/jquery-ui.js"></script>
             <script language="JavaScript" type="text/javascript"
               src="/static/js/jquery.jeditable.js"></script>
             <script language="JavaScript" type="text/javascript"
               src="/static/js/ajaxfileupload.js"></script>
             <script language="JavaScript" type="text/javascript"
               src="/static/js/jquery.effects.fade.js"></script>
            {% endblock footer_javascript_site %}
            {% block footer_javascript_section %}
            {% endblock footer_javascript_section %}
            {% block footer_javascript_page %}
            {% endblock footer_javascript_page %}
        {% endblock footer_javascript %}
    {% endblock footer %}
</html>
```

We open the template we are working on by declaring it to extend `base.html`:

```
{% extends "base.html" %}
```

We define the block that populates the HTML title:

```
{% block head_title %}Photo directory: Search
{% endblock head_title %}
```

We override a (new) block in `base.html` to put something at the beginning of a page, and put a link to add a new Entity. This will be styled to appear at the top right of the page.

```
{% block body_preamble_page %}
<div id="body_preamble_page">
    <strong>
        <a href="/" onclick="PHOTO_DIRECTORY.add_new();
                    return false;">Add new</a>
    </strong>
</div>
{% endblock body_preamble_page %}
```

Now we will populate the main part of the page body.

```
{% block body_main %}
```

First, we define a semantically marked-up search form with a **query** and **submit** button. These will be styled to be fairly striking, and centered (with a little help from presentationally used jQuery to adjust things).

```
<h1>Photo directory: Search</h1>
<form name="search_form" id="search_form" action="/">
    <input type="text" name="query" id="query" value="{{ query }}" />
    <input type="submit" name="submit" id="submit" value="Search" />
</form>
```

We wrap the search results and Entity drilldown portions in two containing `div` elements, in order to facilitate styling:

```
<div class="outer_outer">
    <div class="outer">
```

If there are search results, we display them, but if there are no search results, we still populate the `div` with a heading. This means that when a user first visits a page and is getting acquainted, it is easier to recognize what the search results portion of the page is there for.

```
<div id="search_results">
    {% if search_results %}
        {{ search_results }}
    {% else %}
        <h2>Search results</h2>
    {% endif %}
</div>
```

We do the same thing with the Entity drilldown page. However, we have chosen a different label. For the purposes of development, there are benefits to using an abstract term: not "Person" but "Entity". The directory is first and foremost a directory to keep track of people, but a successful tool is one that is used in a way that its maker never imagined, and if this succeeds people will keep track of more than people. We both identify that this section of the page is for people, and that it may be used for other things, with an `h2` labeling it as `People, etc.`:

```
            <div id="profile">
                {% if profile %}
                    {{ profile }}
                {% else %}
                    <h2>People, etc.</h2>
                {% endif %}
            </div>
        </div>
    </div>
```

This `div` is not displayed initially, but is used by jQuery UI to support Ajax logins.

```
    <div id="login_form" title="Log in">
        <form>
            <fieldset>
                <label for="login">Login</label>
                <input type="text" name="login" id="login"
                  class="text ui-widget-content ui-corner-all" /><br />
                <label for="password">Password</label>
                <input type="password" name="password" id="password"
                  class="text ui-widget-content ui-corner-all" /><br />
            </fieldset>
        </form>
    </div>
```

This is all that we will be putting in the main body area. We close this template block, and move on to the page-specific JavaScript block:

```
    {% endblock body_main %}
    {% block footer_javascript_page %}
    <script language="JavaScript" type="text/javascript">
    <!--
```

We enclose virtually all of our code in a `jQuery(function($){ ... });` wrapper, but the `PHOTO_DIRECTORY` namespace is declared outside so that it will be available in the global namespace and therefore visible to event handlers and the like.

```
var PHOTO_DIRECTORY = new Object();

jQuery(function($)
    {
```

Firefox's Firebug plugin is a powerful debugging tool, and it provides a `console.log()` function that is much better for diagnostic purposes than `alert()`. However, if your code tries to call `console.log()` when Firebug is not open, an error is thrown that may stop some of your JavaScript.

This means that if you forget and leave in a `console.log()` call that you were using when you were debugging, you can have code that will work for your debugging setup and break badly when a visitor tries to see it. It would be best of all not to leave in a `console.log()` call, but the following code provides a safety net, that is, if `console.log()` does not exist, we define it as a function that does nothing.

```
try
    {
    console.log("Starting...");
    }
catch(error)
    {
    console = function()
        {
        }
    console.log = console;
    }
```

We request that when the DOM is ready, the default values in jQuery's `.ajax()` call be set. Besides a default returned data type and form submission type, the global error handler does one other major service, namely, if the settings are set to debug mode, it extracts the `responseText` from the `XMLHttpRequest` and sends it in a notification.

This is important because in Ajax development, we will not necessarily see Django's error pages when there has been a server error. This is unfortunate because they often provide information that is helpful for debugging. This is a way that Ajax development can have access to a helpful Django development feature.

We only do this if `settings.DEBUG` is true. However helpful a good error page may be to developers, including one in this way is needlessly confusing and intimidating to users. If debug mode is off, a short and simple message is sent:

```
$(function()
    {
    $.ajaxSetup(
        {
        datatype: "json",
        error: function(XMLHttpRequest, textStatus, errorThrown)
            {
            if (XMLHttpRequest.responseText)
                {
                {% if settings.DEBUG %}
                    PHOTO_DIRECTORY.send_notification(
                      XMLHttpRequest.responseText);
                {% else %}
                    PHOTO_DIRECTORY.send_notification(
                    "There was error handling your request.");
                {% endif %}
                }
            },
        type: "POST",
        });
    });
```

The following is boilerplate code provided for use with jQuery UI's autocomplete:

```
(function( $ ) {
    $.widget( "ui.combobox", {
        _create: function() {
            var self = this;
            var select = this.element.hide(),
                selected = select.children( ":selected" ),
                value = selected.val() ? selected.text() : "";
            var input = $( "<input>" )
                .insertAfter( select )
                .val( value )
                .autocomplete({
                    delay: 0,
                    minLength: 0,
                    source: function( request, response ) {
                        var matcher = new RegExp(
                          $.ui.autocomplete.escapeRegex(
                            request.term), "i" );
                        response( select.children(
```

```
        "option" ).map(function() {
           var text = $( this ).text();
           if ( this.value && ( !request.term ||
             matcher.test(text) ) )
               return {
                  label: text.replace(
                  new RegExp(
                  "(?![^&;]+;)(?!<[^<>]*)(" +
                  $.ui.autocomplete.escapeRegex(
                  request.term) +
                  ")(?![^<>]*>)(?![^&;]+;)",
                  "gi"), "<strong>$1</strong>"
                   ),
                   value: text,
                   option: this
               };
    }) );
},
select: function( event, ui ) {
PHOTO_DIRECTORY.update_autocomplete_handler(
    event, ui);
    ui.item.option.selected = true;
    //select.val( ui.item.option.value );
    self._trigger( "selected", event, {
        item: ui.item.option
    });
},
change: function( event, ui ) {
 PHOTO_DIRECTORY.update_autocomplete_handler(
    event, ui);
    if ( !ui.item ) {
        var matcher = new RegExp( "^" +
        $.ui.autocomplete.escapeRegex(
            $(this).val() ) + "$", "i" ),
            valid = false;
        select.children(
        "option" ).each(function() {
            if ( this.value.match(
              matcher ) ) {
                this.selected = valid = true;
                return false;
            }
        });
        if ( !valid ) {
           // remove invalid value,
           // as it didn't match anything
            $( this ).val( "" );
```

```
                                select.val( "" );
                                return false;
                            }
                        }
                    }
                })
                .addClass(
                 "ui-widget ui-widget-content ui-corner-left" );

            input.data( "autocomplete" )._renderItem = function(
            ul, item ) {
                return $( "<li></li>" )
                    .data( "item.autocomplete", item )
                    .append( "<a>" + item.label + "</a>" )
                    .appendTo( ul );
            };

            $( "<button> </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);
```

Here we define several variables that will be used. The `id` provided by the view may be empty if no page has been specified; we have an example of giving a default value in the templating engine. The variable will be initialized to a JavaScript `null` value if no (nonempty) `id` has been provided.

```
PHOTO_DIRECTORY.current_profile = {{ id|default:"null" }} ;
PHOTO_DIRECTORY.database_loaded = false;
PHOTO_DIRECTORY.last_attempted_function = null;
PHOTO_DIRECTORY.logged_in = false;
PHOTO_DIRECTORY.DELAY_BETWEEN_RETRIES =
  {{ settings.DELAY_BETWEEN_RETRIES }};
PHOTO_DIRECTORY.SHOULD_DOWNLOAD_DIRECTORY =
  {{ settings.SHOULD_DOWNLOAD_DIRECTORY }};
PHOTO_DIRECTORY.SHOULD_TURN_OFF_HIJAXING =
  {{ settings.SHOULD_TURN_OFF_HIJAXING }};
```

Here is the function that requests a new Entity instance and pulls up its profile for editing:

```
PHOTO_DIRECTORY.add_new = function()
    {
    $.ajax({
        success: function(data, textStatus, XMLHttpRequest)
            {
            if (PHOTO_DIRECTORY.check_authentication(data))
                {
                PHOTO_DIRECTORY.load_profile(data[0].pk);
                }
            else
                {
```

While this functionality is not available everywhere across the program, `PHOTO_DIRECTORY.last_attempted_function` is used for a callback on successful login. On successful login, it will be called if non-null.

```
                PHOTO_DIRECTORY.last_attempted_function =
                  PHOTO_DIRECTORY.add_new;
                PHOTO_DIRECTORY.offer_login();
                }
            },
        url: "/ajax/new/Entity",
        });
    }
```

The following is boilerplate code provided for the `ajaxFileUpload` plugin, homepage at `http://www.phpletter.com/Our-Projects/AjaxFileUpload/`:

```
PHOTO_DIRECTORY.ajax_file_upload = function()
    {
    //starting setting some animation when the ajax starts
    // and completes
    $("#loading")
    .ajaxStart(function()
        {
        $(this).show();
        })
    .ajaxComplete(function()
        {
        $(this).hide();
        });
        /*  preparing ajax file upload
            url: the url of script file handling the
                 uploaded files
            fileElementId: the file type of input
                           element id and it will be the index of
                           $_FILES Array()
            dataType: it support json, xml
            secureuri:use secure protocol
            success: call back function when the ajax complete
            error: callback function when the ajax failed

                */
    $.ajaxFileUpload(
        {
        url: '/ajax/saveimage/' +
             PHOTO_DIRECTORY.current_profile,
        secureuri: false,
        fileElementId: 'image',
        dataType: 'json',
        success: function(data, status)
            {
            if (!PHOTO_DIRECTORY.check_authentication(data))
                {
                PHOTO_DIRECTORY.offer_login();
                }
            },
        });
    return false;
    }
```

The following function is intended to check authentication on data returned by the server. On the server the `@ajax_login_required` decorator intercepts the request and, if the user is not logged in, returns JSON of `{"not_authenticated": true}`. This function and `check_login()` both save their findings in `PHOTO_DIRECTORY.logged_in`.

```
PHOTO_DIRECTORY.check_authentication = function(parsed_json)
    {
    if (parsed_json == '{"not_authenticated": true}')
        {
        PHOTO_DIRECTORY.logged_in = false;
        return false;
        }
    try
        {
        if (parsed_json.not_authenticated)
            {
            PHOTO_DIRECTORY.logged_in = false;
            return false;
            }
        else
            {
            PHOTO_DIRECTORY.logged_in = true;
            return true;
            }
        PHOTO_DIRECTORY.logged_in = true;
        return true;
        }
    catch(error)
        {
        PHOTO_DIRECTORY.logged_in = false;
        return false;
        }
    }
```

`check_login()` serves a similar purpose but does not need to be supplied with any data or arguments:

```
PHOTO_DIRECTORY.check_login = function()
    {
    var result = $.ajax({
        success: function(data, textStatus, XMLHttpRequest)
            {
            if (PHOTO_DIRECTORY.check_authentication(data))
                {
```

```
                PHOTO_DIRECTORY.logged_in = true;
                }
            else
                {
                PHOTO_DIRECTORY.logged_in = false;
                }
            },
        url: "/ajax/check_login",
        });
    }
```

A function to hide additional search results or statuses, beyond the limited number that are initially displayed, would go something like the following:

```
PHOTO_DIRECTORY.hide_additional = function(name)
    {
    $(".show_additional_" + name).hide();
    $("#additional_" + name).show("slow");
    }
```

The following is a function to load the currently selected profile, if there is one:

```
PHOTO_DIRECTORY.load_current_profile = function()
    {
    if (PHOTO_DIRECTORY.current_profile)
       {
       $("#profile").load("/ajax/profile/" +
          PHOTO_DIRECTORY.current_profile,
          PHOTO_DIRECTORY.register_update);
        }
    else
        {
        $("#profile").load("");
        }
    }
```

A function to load an Entity's profile with a specified ID is shown below:

```
PHOTO_DIRECTORY.load_profile = function(id)
    {
    PHOTO_DIRECTORY.current_profile = id;
    if (PHOTO_DIRECTORY.current_profile)
        {
        $("#profile").load("/ajax/profile/" +
          PHOTO_DIRECTORY.current_profile,
          PHOTO_DIRECTORY.register_update);
```

```
            }
        else
            {
            $("#profile").html("");
            }
        }
```

If people upload their own photos, some of them will likely be rather large. This proportionally scales down all images belonging to, say, `img.profile` that are above a maximum width in pixels, to that maximum width.

The following function may be called before an image has loaded; in that case, its `width()` may be 0, in which case the function sets a timeout to wait a configurable number of milliseconds and retry:

```
PHOTO_DIRECTORY.limit_width = function(css_class, limit)
        {
        $(css_class).each(function(index, element)
            {
            if ($(element).width() == 0)
                {
                setTimeout("PHOTO_DIRECTORY.limit_width('" +
                           css_class + "', " + limit + ");",
                           PHOTO_DIRECTORY.DELAY_BETWEEN_RETRIES);
                }
            if ($(element).width() > limit)
                {
                var height = Math.ceil($(element).height() * limit /
                  $(element).width());
                $(element).width(limit);
                $(element).height(height);
                }
            });
        }
```

We place a wrapper around the code to display the login screen:

```
    PHOTO_DIRECTORY.offer_login = function()
        {
        $("#login_form").dialog("open");
        }
```

The following code, when run, will hide the `select` elements of class `autocomplete`, displaying an autocomplete text field. The commented-out line shows the hidden `select` elements:

```
PHOTO_DIRECTORY.register_autocomplete = function()
    {
    $(".autocomplete").combobox();
    /*
    $(".autocomplete").toggle();
    */
    }
```

This function enables inline editing and deletion for fields where in-place editing is desired. We can prepare a field for this by following the naming convention where an HTML ID of `Entity_name_1` means "on the Entity of primary key `1`, the field `name`," and adding `edit` or another CSS class to enable editing, and making sure this function is called.

The following function registers editables and elements otherwise needing to have Ajax functionality added:

```
PHOTO_DIRECTORY.register_editables = function()
    {
```

Items of class `delete` are set to send a request for the server to delete that item, and then make a request to reload a fresh profile. This reloading the profile as a unit, instead of trying to keep track of surgical updates, makes for fewer moving parts with less complex debugging.

```
    $(".delete").each(function(index, item)
        {
        var id = item.id;
        $(item).click(function()
            {
            $.ajax(
                {
                data:
                    {
                    id: id,
                    },
                datatype: "html",
                success: function(data)
                    {
```

If the user comes back as logged in, the profile is reloaded; if the user is not logged in, a "reload the profile next" note is set, and a login screen is offered. (The login screen offers an escape hatch in case the user is not interested enough to want to log in, and we should not assume that our application is important enough to all of our users that it will be worth logging in.)

```
                    if (PHOTO_DIRECTORY.check_authentication(
                      data))
                        {
                        PHOTO_DIRECTORY.reload_profile();
                        }
                    else
                        {
                        PHOTO_DIRECTORY.last_function_called =
                          PHOTO_DIRECTORY.reload_profile;
                        PHOTO_DIRECTORY.offer_login();
                        }
                    },
                url: "/ajax/delete",
                });
            });
        });
```

The `.edit` class, as compared to `delete`, has a similar overall structure. Its URL is different (`/ajax/save` instead of `/ajax/delete`), and it is based on Jeditable, filling out the details of a Jeditable call instead of doing everything from scratch.

```
$(".edit").editable("/ajax/save",
    {
    callback: function(data)
        {
        if (PHOTO_DIRECTORY.check_authentication(data))
            {
            PHOTO_DIRECTORY.reload_profile();
            }
        else
            {
            PHOTO_DIRECTORY.last_function_called =
              PHOTO_DIRECTORY.reload_profile;
            PHOTO_DIRECTORY.offer_login();
            }
        },
    cancel: "Cancel",
    submit: "OK",
    });
```

The `.edit_rightclick` class is basically the same as `.edit`, and works in basically the same way. The chief difference is that it uses the `contextmenu`/`right-click` event instead of the regular (usually left) click for e-mail addresses and URLs where the desired behavior is not to initiate editing on a click as `.edit` is set to do, but to let the normal click behavior occur when the user clicks, and then intercept the `contextmenu` event (specified explicitly) to allow editing on a right-click.

```
$(".edit_rightclick").editable("/ajax/save",
    {
    cancel: "Cancel",
    callback: function(data)
        {
        if (PHOTO_DIRECTORY.check_authentication(data))
            {
            PHOTO_DIRECTORY.reload_profile();
            }
        else
            {
            PHOTO_DIRECTORY.last_function_called =
              PHOTO_DIRECTORY.reload_profile;
            PHOTO_DIRECTORY.offer_login();
            }
        },
    event: "contextmenu",
    submit: "OK",
    tooltip: "Right click to edit.",
    });
```

Here we have another variation on the theme: instead of a text input for ordinarily small fields, we allow a `textarea` for what may often be a much larger chunk of data:

```
$(".edit_textarea").editable("/ajax/save",
    {
    cancel: "Cancel",
    callback: function(data)
        {
        if (PHOTO_DIRECTORY.check_authentication(data))
            {
            PHOTO_DIRECTORY.reload_profile();
            }
        else
            {
            PHOTO_DIRECTORY.last_function_called =
              PHOTO_DIRECTORY.reload_profile;
```

```
                PHOTO_DIRECTORY.offer_login();
                }
            },
        rows: 5,
        submit: "OK",
        tooltip: "Click to edit.",
        type: "textarea",
        });
    }
```

This is a convenience method that calls several things that should be called after the page has been updated. If the `settings.py` says that link Hijaxing should be turned off, the "graceful degradation" option will become the default.

```
PHOTO_DIRECTORY.register_update = function()
    {
    PHOTO_DIRECTORY.limit_width("img.profile", 150);
    PHOTO_DIRECTORY.limit_width("img.search_results", 80);
    PHOTO_DIRECTORY.register_editables();
    PHOTO_DIRECTORY.register_autocomplete();
    if (PHOTO_DIRECTORY.SHOULD_TURN_OFF_HIJAXING)
        {
        $("a").removeAttr("onclick");
        // This link needs to be hijaxed:
        $("#add_new").click(function()
            {
            PHOTO_DIRECTORY.add_new();
            return false;
            });
        }
    }
```

The following function reloads the current profile. This can be called after an edit to ensure a fresh and correct page.

```
PHOTO_DIRECTORY.reload_profile = function()
    {
    PHOTO_DIRECTORY.tables_loaded = 0;
    PHOTO_DIRECTORY.load_current_profile();
    }
```

The following function hijaxes the search function:

```
PHOTO_DIRECTORY.search = function()
    {
    $("#search_results").load("/ajax/search?query=" +
```

```
        escape(document.search_form.query.value),
        PHOTO_DIRECTORY.register_update);
    }
```

The following function displays notifications to the user. The `setTimeout()` call is used so that the function will return immediately, and jQuery calls are made to fade it in, wait, and fade it out. The delay involved is five seconds plus two milliseconds per character in the message; in production, this should make very little difference, but if a detailed Django error page is served up, it will stay around for much longer.

```
PHOTO_DIRECTORY.send_notification = function(message)
    {
    $("#notifications").html("<p>" + message + "</p>");
     clearTimeout($('#notifications').data('showTimeout'));
    $('#notifications').data('showTimeout',
      setTimeout((function(message)
        {
        return function ()
            {
            $('#notifications').show('slow').delay((5000 +
              (message.length * 2))).hide('slow');
            };
        })(message), 0);
    }
```

The following is a function to show additional search results or statuses. It hides the link that says **Show all**:

```
PHOTO_DIRECTORY.show_additional = function(name)
    {
    $(".show_additional_" + name).hide();
    $("#additional_" + name).show("slow");
    }
```

The following function is intended to be called with preset arguments by the event handler for a select used in autocompletes:

```
PHOTO_DIRECTORY.update_autocomplete = function(id, html_id)
    {
    var value = $("#" + html_id).val();
    $.ajax({
        data:
            {
            id: id,
            value: value,
            },
```

```
            url: "/ajax/save",
            });
        PHOTO_DIRECTORY.reload_profile();
        }
```

The following function is intended to serve as an event handler by the autocomplete fields themselves:

```
PHOTO_DIRECTORY.update_autocomplete_handler = function(event, ui)
    {
    var split_value = ui.item.value.split(".");
    var field = split_value[0];
    var id = split_value[1];
    $.ajax({
        data:
            {
            id: "Entity_" + id + "_" +
                PHOTO_DIRECTORY.current_profile,
            value: field,
            },
        url: "/ajax/save",
        });
    PHOTO_DIRECTORY.reload_profile();
    }
```

The following setup function should be called as soon as the document's DOM is ready.

```
$(function()
    {
    if (!PHOTO_DIRECTORY.SHOULD_TURN_OFF_HIJAXING)
        {
        $("#search_form").submit(function(event)
            {
            PHOTO_DIRECTORY.search();
            return false;
            });
        }
    $("#query").width($(window).width() - 240);
```

A dialog is created for the login form; it is not initially used, but it is available for `offer_login()`. The button makes an Ajax login call.

```
        $("#login_form").dialog({
            autoOpen: false,
            height: 300,
```

```
width: 350,
modal: true,
buttons:
    {
    'Log in': function()
        {
        $.ajax({
            data:
                {
                "login": document.getElementById(
                          "login").value,
                "password": document.getElementById(
                            "password").value,
                },
            datatype: 'text',
```

On a successful login attempt, we notify the user, and if a function has been registered as the last attempted function before attempting login, that function was called. This provides for some continuity after the interruption of a login request to resume the prior workflow.

```
            success: function(data, textStatus,
              XMLHttpRequest)
               {
```

On login, we notify the user, load the database, and call any function registered as what the user was attempting before login.

```
               if (data)
                 {
                 PHOTO_DIRECTORY.send_notification(
                   "You have successfully logged " +
                   "in and can now make changes.");
                 PHOTO_DIRECTORY.load_database();

                 $(".ui-dialog").hide();
                 $(".ui-widget-overlay").hide();
                 PHOTO_DIRECTORY.register_update();
                 if (
                 PHOTO_DIRECTORY.last_attempted_function)
                   {
               PHOTO_DIRECTORY.last_attempted_function();
                    }
                 }
               else
```

```
                                {
                                 PHOTO_DIRECTORY.send_notification(
                                  "Your login was not successful.");
                                 }
                                },
                          url: "/ajax/login",
```

That takes us to the end of the setup. We perform a couple of housekeeping calls and wind down:

```
                              close: function(){},
                              });
                          },
                      },
                  });
              PHOTO_DIRECTORY.check_login();
              PHOTO_DIRECTORY.register_update();
              });


          });
      // -->
      </script>
      {% endblock footer_javascript_page %}
```

And that's it.

Let's take a look at some of the CSS referred to earlier.

CSS for styling the directory

This is the stylesheet we use to style things, adding emphasis to some and de-emphasizing other things. If we want to deliver a good, working system that is graceful to use, saying "There is no 'CSS' in 'Asynchronous JavaScript and XML'" is the same kind of legalistic error as saying, "There is no 'JSON' in 'Asynchronous JavaScript and XML .'" CSS can be used for aesthetics that do not correlate to the functionality of the core system, but the difference to usability and usefulness between practical use of CSS and impractical (or no) CSS is substantial. CSS will be treated here like JavaScript and other technologies as ways to deliver practical usefulness.

The following links show initially hidden search results or statuses that are important and are emphasize them:

```
a.hide_additional_statuses, a.hide_additional_results
    {
    font-weight: bold;
    }

a.show_additional_statuses, a.show_additional_results
    {
    font-weight: bold;
    }
```

Not presently in use, this is intended for a non-displayed `iframe` to send unwanted submissions to:

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

This is a usability-related choice of fonts; Verdana in particular reads very well on the screen, and it looks larger.

Note that we are not making a common web design mistake of choosing Verdana for its usability, and then cancelling out some of its usability benefits by shrinking it to free up more screen real estate.

```
body
    {
    font-family: Verdana, Arial, sans;
    }
```

The following is for the links at the upper right-hand corner of the page:

```
#body_preamble_page
    {
    text-align: right;
    margin-right: 30px;
    margin-top: 15px;
    }
```

These are the initially hidden `div` elements with "extra" search results or statuses if there are more than will be displayed initially. (The parameters for how many search results or statuses are displayed are modifiable in `settings.py`.)

```
div#additional_statuses, div#additional_results
    {
    display: none;
    }
```

The the following `div` is for the link to delete a whole Entity:

```
div.deletion
    {
    float: right;
    margin-right: 20px;
    }
```

An entity's description, postal address, status, or other editable fields are specified as `textarea`. Here and with other fields, we do the opposite of having a bold label and plaintext content: we leave the label with the default appearance and font weight, and make the content itself bold:

```
div.edit_textarea
    {
    font-weight: bold;
    }
```

The `:after` pseudo-class will be used to display Unicode symbols as a cue that fields are user editable. This specifies that they will not be in bold.

```
div.edit_textarea:after
    {
    font-weight: normal;
    }

div.image
    {
    float: left;
    }
```

We use semantic markup and style with CSS even when we want a table-like display.

```
div.outer
    {
    display: table-row;
    }
```

```
div.outer_outer
    {
    display: table;
    width: 100%;
    }

div.search_result
    {
    display: table;
    margin-bottom: 20px;
    padding-right: 50px;
    width: 100%;
    }

div.search_result:hover
    {
    background-color: #ffffc0;
    }
```

Links in a search result `div` will be Entities' names, and will pull up their profiles.

```
div.search_result a
    {
    font-weight: bold;
    }
```

The following is used for the **Add new** link, and we will add below a **Bad network connection** link:

```
div.standard_links
    {
    font-weight: bold;
    margin-right: 15px;
    margin-top: 15px;
    text-align: right;
    }
```

We add a symbol showing a hand with a pen, grayed to lighten its visual effect, but larger to be easier to see. This will appear as HTML entities belonging to the relevant CSS classes.

```
.edit:after, .edit_textarea:after
    {
    color: #808080;
    content: "✍"; /* The symbol, not ASCII encoding, of &#9997; */
    font-size: larger;
    }
```

We add a similar, but not identical, markup to the end of links, which are edited by right-clicking. There are two symbols as we added a finger pointing right, and the symbol is a darker shade of grey. These do not make a perfect solution, and it would be wrong to expect users to automatically infer what the tooltip says: **Right-click to edit**. However, they provide a distinct and distinctive visual cue, and even if they are not perfect, they make the interface easier to remember and use.

```
.edit_rightclick:after
    {
    color: #404040;
    content: "☞✍"; /* The symbols, not ASCII encodings, of &#9755;
                      and then &#9997; */
    text-decoration: none;
    font-size: larger;
    }

img.profile
    {
    float: left;
    margin-bottom: 20px;
    margin-right: 20px;
    }
```

`h1` and `h2` tags are made to be large, but the emphasis of being bold is taken away, to be given instead to the search query, profile data, and similar tags. This develops a theme introduced earlier: labels are not emphasized; real content is made bold.

```
h1
    {
    font-size: 3em;
    font-weight: normal;
    margin-bottom: 50px;
    text-align: center;
    }

h2
    {
    font-size: 2em;
    font-weight: normal;
    }

h3
    {
    font-size: 1.8em;
    }
```

```
img.search_results
    {
    float: left;
    }
```

The markup used for the Ajax login is hidden immediately.

```
#login_form
    {
    display: none;
    }
```

Notifications have a yellow background and grey border. This is striking and catches the eye, and we intentionally are not using red as not all notifications are errors and we do not want to imitate the error dialog that says, "Error: The operation completed successfully." (Not to mention that computers make some people nervous and a bright red error message has needless force.)

```
#notifications
    {
    background-color: #ffff80;
    border: 3px solid #808080;
    display: none;
    padding: 20px;
    }
```

The profile has a light blue background, while the search results have a light yellow background and the search bar itself has a plain white background. These backgrounds are not specifically intended for aesthetics, but to visually mark out how the screen is divided into different areas doing different jobs. The purpose is to add (helpful) redundancy in the signals provided about where the user is and what things the user can do.

```
#profile
    {
    background-color: #ffffc0;
    display: table-cell;
    height: 100%;
    padding: 50px;
    width: 100%;
    }
```

Especially for an empty profile, it is nice to have a little more padding at the bottom than the top. This accomplishes that slight aesthetic touch.

```
#profile h2
    {
    margin-top: 0;
    }

.search_result
    {
    padding: 10px;
    padding-right: 0px;
    }

.search_result:hover
    {
    background-color: #ccccff;
    }

#search_results
    {
    background-color: #ddddff;
    display: table-cell;
    height: 100%;
    padding: 50px;
    padding-right: 0px;
    width: 32%;
    }

#search_results h2
    {
    margin-top: 0;
    }
```

The following is the `span` that is used to delete an added URL, e-mail address, and so on from an Entity:

```
span.delete
    {
    color: #aa0000;
    font-size: larger;
    font-weight: bold;
    }
```

```
span.emphasized
    {
    font-size: larger;
    }

span.placeholder
    {
    font-style: italic;
    font-weight: normal;
    }
```

We style `tag` elements to look slightly distinctive compared to the rest of the application:

```
span.tag
    {
    background-color: #dddddd;
    border: 1px solid silver;
    padding-left: 3px;
    padding-right: 3px;
    }
```

The timestamp below a status is displayed, but it is visually downplayed, as usually the status is the important thing to read.

```
span.timestamp
    {
    font-size: smaller;
    color: #808080;
    }
```

We also use more usable fonts in a `textarea`. We style `textarea` elements to have a reasonable height (CSS requires a hack if it is to be used to allow for direct specification of a number of rows in a `textarea`), and to be as wide as they can.

```
textarea
    {
    font-family: Verdana, Arial, sans;
    font-size: larger;
    height: 5em;
    width: 100%;
    }
```

The input for the user query is large, and is given bold emphasis:

```
#query
    {
    border: 2px solid black;
    font-size: 2em;
    font-weight: bold;
    margin-bottom: 50px;
    padding: 2px;
    width: 80%;
    }

#search_form
    {
    margin-left: 40px;
    }

#submit
    {
    border: 2px solid black;
    font-size: 2em;
    font-weight: bold;
    margin-left: 10px;
    }
```

The following is boilerplate CSS for a `div` used to dim the screen in Ajax modal forms, including our Ajax login functionality:

```
.ui-widget-overlay
    {
    background: #000000;
    -ms-filter:"progid:DXImageTransform.Microsoft.Alpha(Opacity=50)";
    filter: alpha(opacity=50);
    opacity: .5;
    position: absolute;
    top: 0;
    bottom: 0;
    left: 0;
    right: 0;
    z-index: 1000;
    }
```

Our updated urlpatterns

Our `urlpatterns`, for the example AHAH solution above and for the fuller client-side database example solution below, are as follows:

```
urlpatterns = patterns(u'',
    (ur'^$', directory.views.homepage),
    (ur'^accounts/login/$', u'django.contrib.auth.views.login'),
    (ur'^admin/', include(admin.site.urls)),
    (ur'^ajax/check_login', directory.views.ajax_check_login),
    (ur'^ajax/delete', directory.views.ajax_delete),
    (ur'^ajax/download/(Email|Entity|Phone|Status|Tag|URL)',
      directory.views.ajax_download_model),
    (ur'^ajax/login', directory.views.ajax_login_request),
    (ur'^ajax/new/Entity', directory.views.new_Entity),
    (ur'^ajax/profile/(\d+)', directory.views.ajax_profile),
    (ur'^ajax/saveimage/(\d+)', directory.views.saveimage),
    (ur'^ajax/save', directory.views.save),
    (ur'^ajax/search', directory.views.ajax_search),
    (ur'^(create/Entity)', directory.views.redirect),
    (ur'^(create/Location)', directory.views.redirect),
    (ur'^manage/Entity/?(\d*)', directory.views.modelform_Entity),
    (ur'^manage/Location/?(\d*)',
      directory.views.modelform_Location),
    (ur'^profile/images/(\d+)', directory.views.image),
    (ur'^profile/(new)$', directory.views.profile_new),
    (ur'^profile/(\d+)$', directory.views.profile_existing),
   )
```

And that is the last of the code we need!

We now have, on the server-side and client-side, all the pieces in place for a complete system that includes some best practices. We can tinker and extend it, but we have a complete working system now.

Summary

We have now covered all the bases for a simple, AHAH solution, and most of the time this will be best. However, there is more that we can do. In general, lazy programming and lazy network access that pulls as little as possible and does so as late as possible is the solution of choice and works better than a premature proactive solution that seems to be the root of all evil.

In our next chapter we will cover further customization and development. Let's go!