6
jQuery In-place Editing Using Ajax

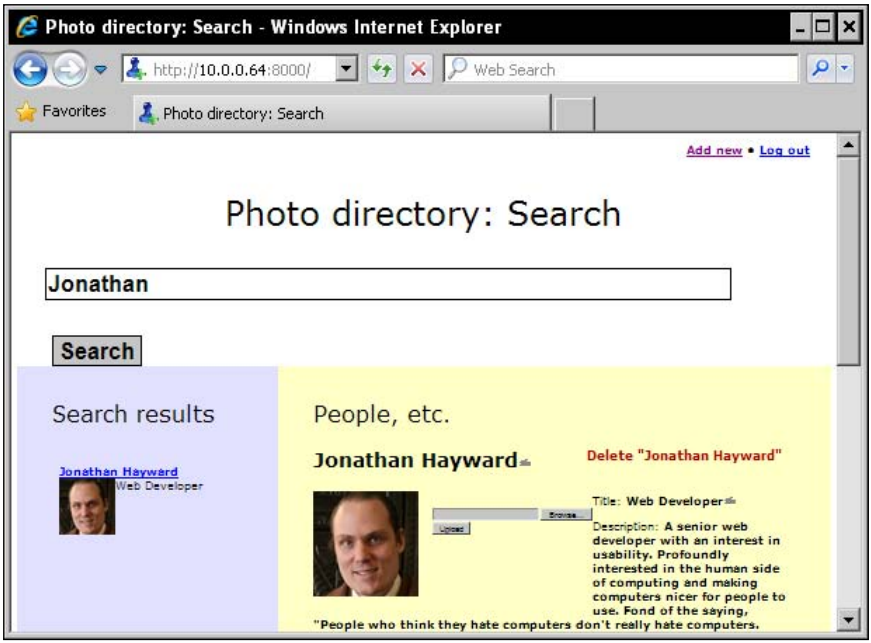
jQuery as a library is intended to have a good, solid, lightweight core that invites an ecosystem of plugins. Ordinary JavaScript programmers have been known to learn jQuery and start writing new plugins on their first day. In this chapter, we will take advantage of the **Jeditable – Edit In Place Plugin For jQuery**, with homepage at `http://www.appelsiini.net/projects/jeditable`. Jeditable is not the only plugin out there, nor the only good one; it is one of a number of interesting and useful plugins that are available in the jQuery plugin ecosystem. If you would like to find or explore jQuery plugins, `http://plugins.jquery.com/` is a good place to start.

In this chapter, we will cover:

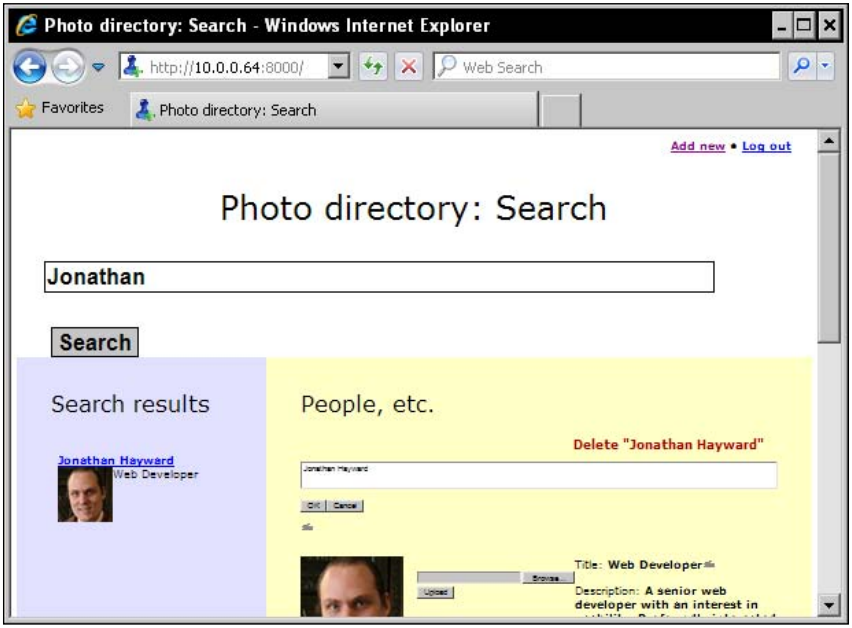
- How to include a plugin on a page
- How to use jQuery with the Jeditable plugin to add "edit-in-place" functionality
- How to use Django to keep track of the server-side responsibilities
- How to make a detailed profile page that supports in-place editing, as well as adding the same functionality to the search results page

In our next chapter, we will complete the profile pages, building on autocomplete features to allow an employee's supervisor to be specified without a long dropdown menu, as well as exploring autocomplete for search.

This will allow us to create a results page as shown:



If someone clicks on the name, for instance, it becomes:



When someone clicks **OK**, the data is saved on the server, and also shown on the page.

Let's get started on how this works.

Including a plugin

We include a jQuery plugin on a page by including jQuery, then including the plugin (or plugins, if we have more than one). In our `base.html`, we update:

```
{% 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>
{% endblock footer_javascript_site %}
```

This is followed by the `footer_javascript_section` and `footer_javascript_page` blocks. This means that if we don't want the plugin ,which is the last inclusion, to be downloaded for each page, we could put it in overridden section and page blocks. This would render as including the plugin after jQuery.

How to make pages more responsive

We would also note that the setup, with three JavaScript downloads, is appropriate for development purposes but not for deployment. In terms of YSlow client-side performance optimization, the recommended best practice is to have one HTML/XHTML hit, one CSS hit at the top, and one JavaScript hit at the bottom. One of the basic principles of client-side optimization, discussed by Steve Souders (see `http://developer.yahoo.com/yslow/`) is, since HTTP requests slow the page down, the recommended best practice is to have one (preferably minified) CSS inclusion at the top of the page, and then one (preferably minified) JavaScript inclusion at the bottom of each page. Each HTTP request beyond this makes things slower, so combining CSS and/or JavaScript requests into a single concatenated file is low-hanging fruit to improve how quick and responsive your web pages appear to users.

For deployment, we should minify and combine the JavaScript. As we are developing, we also have JavaScript included in templates and rendered into the delivered XHTML; this may be appropriate for development purposes. For deployment though, as much shared functionality as possible should be factored out into an included JavaScript file. For content that can be delivered statically, such as CSS, JavaScript, and even non-dynamic images, setting far-future Expires/Cache-Control headers is desirable. (One practice is to never change the content of a published URL for the kind of content that has a far-future expiration set, and then if it needs updating, instead of changing the content at the same location, leave the content where it is, publish at a new location possibly including a version number, and reference the new location.)

A template handling the client-side requirements

Here's the template. Its view will render it with an entity and other information. At present it extends the base directly; it is desirable in many cases to have the templates that are rendered extend section templates, which in turn extend the base. In our simple application, we have two templates which are directly rendered to web pages. One is the page that handles both search and search results—dealt with earlier—and the other, the page that handles a profile, from the following template:

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

Following earlier discussion, we include honorifics before the name, and post-nominals after. At this point we do not do anything to make it editable.

```
{% block head_title %}
{{ entity.honorifics }} {{ entity.name }} {{ entity.post_nominals }}
{% endblock head_title %}
{% block body_main %}
```

There is one important point about Django and the title block. The Django developers do not find it acceptable to write a templating engine that produces errors in production if someone attempts to access an undefined value (by typos, for instance). As a result of this design decision, if you attempt to access an undefined value, the templating engine will silently insert an empty string and move on. This means that it is safe to include a value that may or may not exist, although there are ways to test if a value exists and is nonempty, and display another default value in that case. We will see how to do this soon. Let's move on to the main block, defined by the last line of code.

Once we are in the main block, we have an `h1` which is almost identical to the title block, but this time it is marked up to support editing in place. Let us look at the `honorifics` span; the `name` and `post_nominals` spans work the same way:

```
<h1>
    <span id="Entity_honorifics_{{ entity.id }}" class="edit">
        {% if entity.honorifics %}
            {{ entity.honorifics }}
        {% else %}
            Click to edit.
        {% endif %}
    </span>
```

The class `edit` is used to give all `$(".edit")` items some basic special treatment with Jeditable; there is nothing magical about the class name, which could have been replaced by `user-may-change-this` or something else. `edit` merely happens to be a good name choice, like almost any good variable/function/object name.

We create a naming convention in the span's HTML ID which will enable the server side to know which—of a long and possibly open-ended number of things we could intend to change—is the one we want. In a nutshell, the convention is *modelname_fieldname_instanceID*. The first token is the model name, and is everything up to the first underscore. (Even if we were only interested in one model now, it is more future proof to design so that we can accommodate changes that introduce more models.)

The last token is the instance ID, an integer. The middle token, which may contain underscores (for example `post_nominals` in the following code), is the field name. There is no specific requirement to follow a naming convention, but it allows us to specify an HTML ID that the server-side view can parse for information about which field on which instance of which model is being edited.

We also provide a default value, in this case `Click to edit`, intended not only to serve as a placeholder, but to give users a sense on how this information can be updated.

We might also observe that here and in the following code, we do not presently have checks against race conditions in place. So nothing here or in the following code will stop users from overwriting each others' changes. This may be taken as a challenge to refine and extend the solution to either prevent race conditions or mitigate their damage.

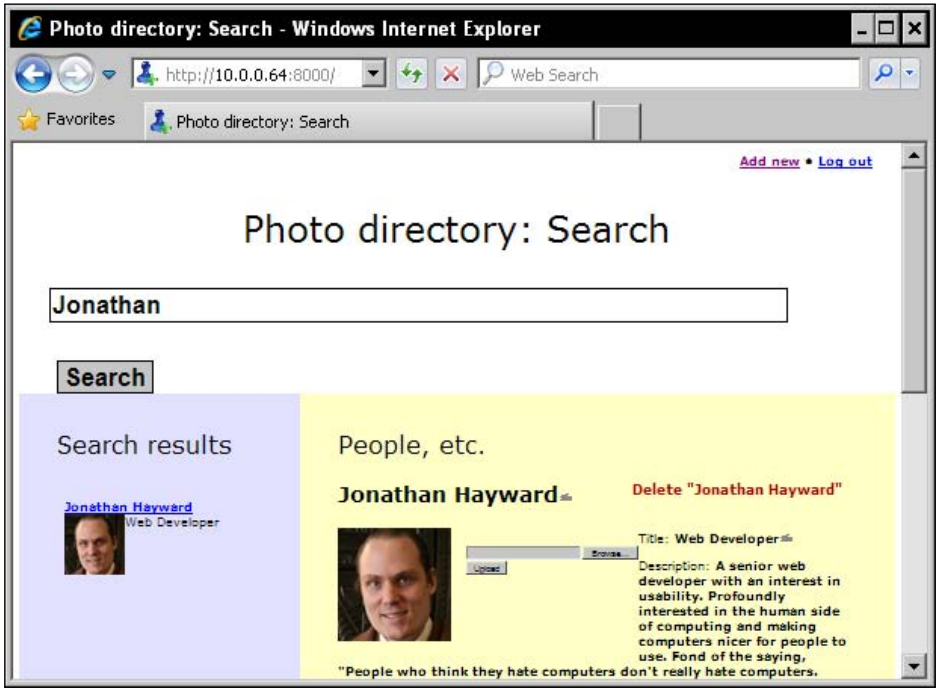
```
    <span id="Entity_name_{{ entity.id }}" class="edit">
        {% if entity.name %}
            {{ entity.name }}
```

```
            {% else %}
                Click to edit.
            {% endif %}
        </span>
        <span id="Entity_post_nominals_{{ entity.id }}" class="edit">
            {% if entity.post_nominals %}
                {{ entity.post_nominals }}
            {% else %}
                Click to edit.
            {% endif %}
        </span>
    </h1>
```

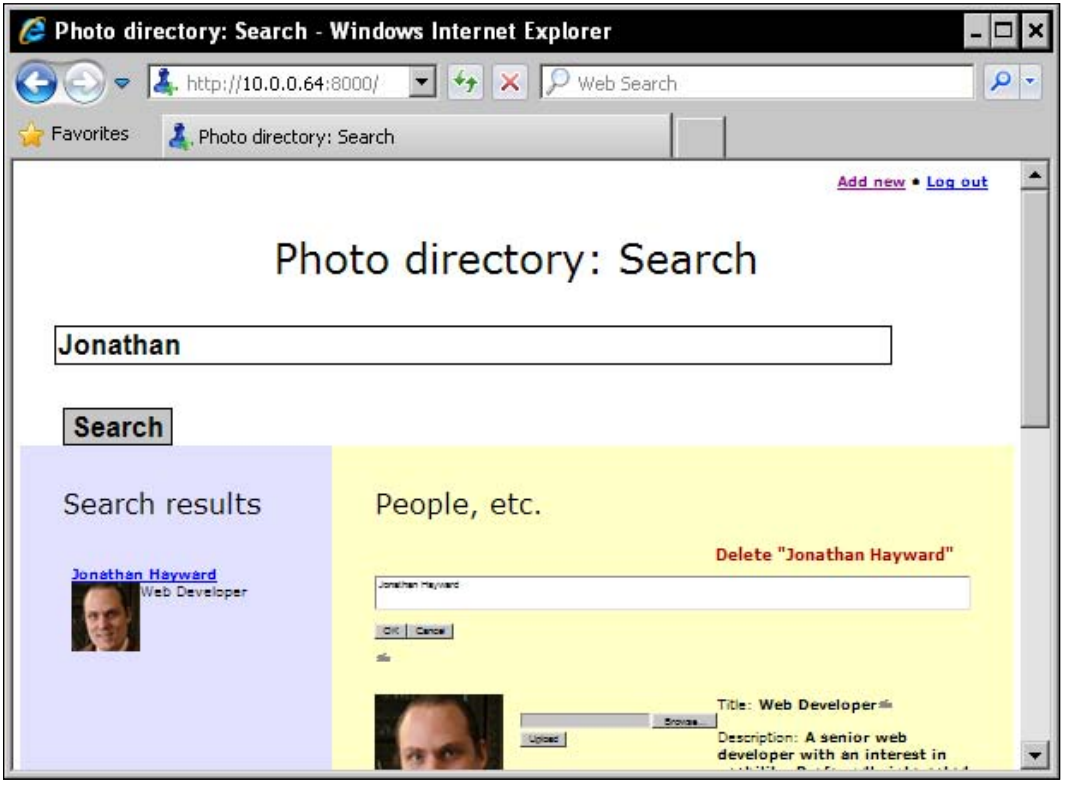
This approach is an excellent first approach but in practice is an `h1` with three slots that say `Click to edit` on a profile, creating needless confusion. We move to a simplified:

```
<h1 class="edit" id="Entity_name_{{ entity.id }}">
    {{ entity.name }}
</h1>
```

Taken together, the three statements form the heading in this screenshot:



If we click on the name (for instance) it becomes:



The image is presently a placeholder; this should be expanded to allow an image to be uploaded if the user clicks on the picture (implementing consistent-feeling behavior whether or not we do so via the same plugin). We also need the `view` and `urlpattern` on the backend:

```
{% if entity.image %}
<img src="/images/{{ entity.id }}" alt="{{ entity.name }}">
{% endif %}
```

The bulk of the profile

For small bits of text, we use the `edit` CSS class, which will be transformed to an input of type text on click (or double-click or mouseover, if we were using Jeditable differently). The description is an example of something that would more naturally lend itself to a textarea, so we will use the `edit_textarea` CSS class, which will be configured to use a textarea.

```
<p>
        Description
    <strong id="Entity_description_{{ entity.id }}"
            class="edit_textarea">
        {{ entity.description }}
    </strong>
</p>
```

The **Department**, as well as **Reports to** field, are not arbitrary text in our implementation; they are another entity (if one is specified). This could appropriately enough be implemented as a dropdown menu, but even a carefully pruned dropdown menu could be long and unwieldy for a large company. We will, in our next chapter, use an autocomplete plugin for this job.

One additional note on usability: When displaying "label: value" information on pages, particularly heavily used pages, the most basic option is not to use any emphasis:

Name: J. Smith

To help people's eyes find what they want, one obvious solution is to emphasize the label, as in:

Name: J. Smith

This works well the first time. However, if people are looking at the same set of fields, in the same order, on a web page they visit repeatedly, it is no longer best to emphasize the labels. Regular visitors already know what the labels are, and the motive for even looking at the labels is to see the value. Therefore, for our directory, we will be using bold for the value rather than the label:

Name: **J. Smith**

```
<p>Department:
    <strong>
        {{ entity.department.name }}
    </strong>

<p>Homepage:
    {% if entity.homepage %}
```

```
            <a href="{{ entity.homepage }}">
        {% endif %}
        <strong class="edit_rightclick"
                id="Entity_homepage_{{ entity.id }}">
            {% if entity.homepage %}
                {{ entity.homepage }}
            {% else %}
                Right click to change.
            {% endif %}
        </strong>
        {% if entity.homepage %}
            </a>
        {% endif %}
    </p>
```

If a homepage is defined, we give the URL, wrapped in a link and a `strong` that makes the link editable by right clicking. If the link were just editable by a regular click, Jeditable would short-circuit the usual and expected behavior of clicking on a link taking you to the corresponding page or opening the corresponding e-mail. To allow editing while also allowing normal use of links on a profile page, we assign the right click rather than click event to be the way to allow editing. From a UI consistency perspective, it might be desirable to additionally always allow a right click to trigger any (possible) editing. However, we will leave that on our wishlist for now. We will define JavaScript later on in this chapter that will add desired behavior.

Whitespace and delivery

The formatting used above is preferable for development; for actual delivery, we may wish to strip out all whitespace that can be stripped out, for this page:

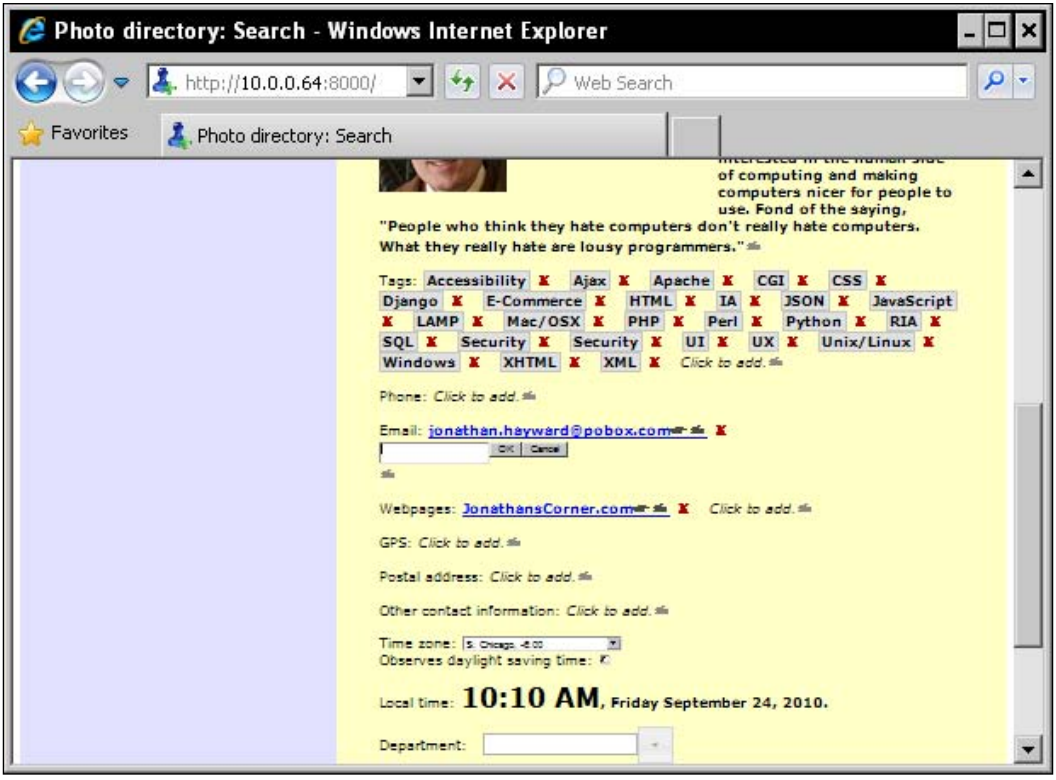
```
{% if entity.homepage %}<a href="{{ entity.homepage }}">{% endif
%}<strong class="edit_rightclick" id="entity_homepage_{{ entity.id
}}">{% if entity.homepage %}{{ entity.homepage }}{% else %}Right click
to change.{% endif %}</strong>{% if entity.homepage %}</a>{% endif
%}<br />
```

Some browsers now are better about this, but it has happened in the past that if you have whitespace such as a line break between the intended text of a link and the `</a>` tag, you could get unwanted trailing whitespace with a visible underline on the rendered link. In addition, pages load faster if minified. For development purposes, though, we will add whitespace for clarity. In the next code, we will have a spurious space before rendered commas because we are not stripping out unnecessary whitespace:

```
<p>Email:
    <strong>
        {% for email in emails %}
```

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

This allows e-mails to be added, like so:



For the **Location** field, we are deferring an intelligent way to let people choose an existing location, or create a new one, until the next chapter. For now we simply display a location's identifier, which is meant as a human-readable identifier rather than a machine-readable primary key or other identifier:

```
<p>Location:
    <strong>
        {{ entity.location.identifier }}
    </strong>
</p>
```

This entails a change to the `Location` model, to allow:

```
class Location(models.Model):
    identifier = models.TextField(blank = True)
    description = models.TextField(blank = True)
    office = models.CharField(max_length = 2,
                              choices = OFFICE_CHOICES,
                              blank = True)
    postal_address = models.TextField(blank = True)
    room = models.TextField(blank = True)
    coordinates = GPSField(blank = True)
```

The **Phone** field is the last one that is user editable.

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

The following fields are presently only displayed. The **Reports to** field should be autocomplete based. The **Start date** field might well enough be left alone as a field that should not need to be updated, or for demonstration purposes it could be set to a jQuery UI datepicker, which would presumably need to have Ajax saving functionality added.

```
<p>Reports to:
    <strong>
        {{ entity.reports_to.name }}
    </strong>
</p>
```

```
<p>Start date:
    <strong>
        {{ entity.start_date }}
    </strong>
</p>
{% endblock body_main %}
```

Page-specific JavaScript

The page-specific JavaScript follows. The first few lines enable the `edit`, `edit_rightclick`, and `edit_textarea` CSS classes to have in-place editing:

```
{% block footer_javascript_page %}
<script language="JavaScript" type="text/javascript">
<!--
function register_editables()
    {
    $(".edit").editable("/ajax/save",
        {
        cancel: "Cancel",
        submit: "OK",
        tooltip: "Click to edit.",
        });
    $(".edit_rightclick").editable("/ajax/save",
        {
        cancel: "Cancel",
        submit: "OK",
        tooltip: "Right click to edit.",
        event: "contextmenu",
        });
    $(".edit_textarea").editable("/ajax/save",
        {
        cancel: "Cancel",
        submit: "OK",
        tooltip: "Click to edit.",
        type: "textarea",
        });
    }

$(function()
    {
    register_editables();
    });
// -->
</script>
{% endblock footer_javascript_page %}
```

Support on the server side

This function provides a rather unadorned logging of changes. This could be expanded to logging in a form intended for machine parsing, display in views, and so on in `functions.py`:

```
def log_message(message):
    log_file = os.path.join(os.path.dirname(__file__),
                            directory.settings.LOGFILE)
    open(log_file, u'a').write(u"%s: %s\n" % (time.asctime(),
                               message)
```

In `settings.py`, after the `DATABASE_PORT` is set:

```
# Relative pathname for user changes logfile for directory
LOGFILE = u'log'
```

In the `urlpattern` in `urls.py`:

```
    (ur'^ajax/save', views.save),
    (ur'^profile/(\d+)$', views.profile),
```

In `views.py`, our import section has grown to the following:

```
from django.contrib.auth import authenticate, login
from django.contrib.auth.decorators import login_required
from django.core import serializers
from django.db.models import get_model
from django.http import HttpResponse
from django.shortcuts import render_to_response
from django.template import Context, Template
from django.template.defaultfilters import escape
from django.template.loader import get_template
from directory.functions import ajax_login_required

import directory.models
import json
import re
```

In `views.py` proper, we define a profile view, with the regular `@login_required` decorator. (We use `@ajax_login_required` for views that return JSON or other data for Ajax requests, and `@login_required` for views that return a full web page.)

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

The following view saves changes made via in-place edits:

```
@ajax_login_required
def save(request):
    try:
        html_id = request.POST[u'id']
        value = request.POST[u'value']
    except:
        html_id = request.GET[u'id']
        value = request.GET[u'value']
    if not re.match(ur'^\w+$', html_id):
        raise Exception(u'Invalid HTML id.')
```

First we test, specifically, for whether a new e-mail is being added. The last parsed token in that case will be the ID of the Entity the e-mail address is for.

```
    match = re.match(ur'EntityEmail_new_(\d+)', html_id)
    if match:
        model = int(match.group(1))
```

We create and save the new `EntityEmail` instance:

```
        email = directory.models.EntityEmail(email = value,
                entity = directory.models.Entity.objects.get(
                        pk = model))
        email.save()
```

We log what we have done, and for a view servicing Jeditable Ajax requests, return the HTML that is to be displayed. In this case we return a new link, and re-run the script that applies in-place edit functionality to all appropriate classes, as dynamically added content will not have this happen automatically. Our motive is that people will sometimes hit **Save** and then realize they made a mistake they want to correct, and we need to handle this as gracefully as the case where the in-place edit is perfect on the first try. We escape for display:

```
        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))
```

The `else` clause is the normal case. First it parses the `model`, `field`, and `id`:

```
else:
    match = re.match(ur'^(.*?)_(.*)_(\d+)$', html_id)
    model = match.group(1)
    field = match.group(2).lower()
    id = int(match.group(3))
```

Then it looks up the selected model (under the `directory` module, rather than anywhere), finds the instance having this ID, sets the instance's `field` value, and saves the instance. The solution is generic, and does the usual job that would be done by code like `entity.name = new_name`.

```
    selected_model = get_model(u'directory', model)
    instance = selected_model.objects.get(pk = id)
    setattr(instance, field, value)
    instance.save()
```

Finally, we log the change and return the HTML to display, in this case simply the value. As with previous examples, we escape the output against injection attacks:

```
    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))
```

Summary

We have now gone from a basic foundation to continuing practical application. We have seen how to divide the labor between the client side and server side. We used this to make a profile page in an employee directory where clicking on text that can be edited enables in-place editing, and we have started to look at usability concerns.

More specifically, we have covered how to use a jQuery plugin, in our case Jeditable, in a solution for Ajax in-place editing. We saw how to use Jeditable in slightly different ways to more appropriately accommodate editable plain text and editable e-mail/URL links. We discussed the server-side responsibilities, including both a generic solution for when a naming convention is required. We looked at an example of customizing behavior when we want something more closely tailored to specific cases (which often is part of solving usability problems well), and also how a detailed profile page can be put together.

In the next chapter, we will address part of the profile page not solved here, namely, how to use autocomplete-style functionality to provide a well-scaling alternative to let people choose Entities for the `department` and `reports_to` field. Let's look at that more closely.