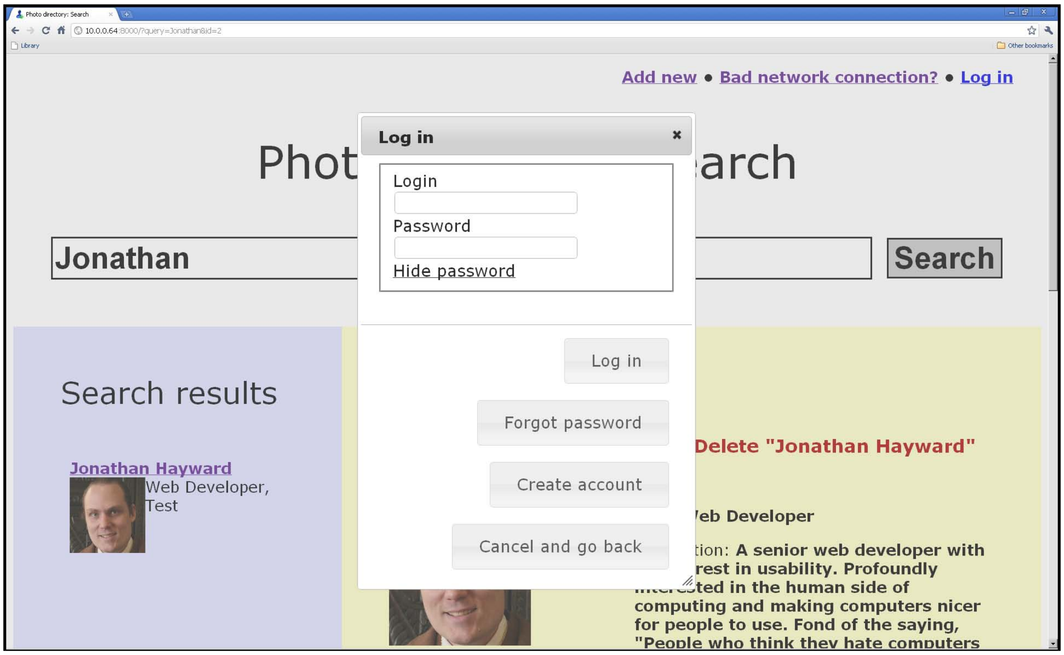
5
Signing Up and Logging into a Website Using Ajax

Authentication and logins using Django Ajax can be accomplished using the following:

- On the server-side, an `@ajax_login_required` decorator for views that is used similarly to Django's `@login_required`, and also server-side Ajax authentication handling.
- On the client-side, a function that will, if an Ajax call returns a specific message saying authentication is needed, will provide an appropriate client-side `div` requesting login credentials and other details, and allow the request to be resubmitted after authentication has been cleared.

A **decorator** is a function that encapsulates another function and provides some added or modified functionality compared to the bare original. It is one of the patterns defined in the "gang of four" patterns book, "*Patterns: Elements of Reusable Object-Oriented Software*" by *Erich Gamma*, *Richard Helm*, *Ralph Johnson*, and *John Vlissides*; *Addison-Wesley Professional*.

In this specific decorator, an inner function wrap is defined, which calls the original function if the user is authenticated and returns a JSON message that the user is not authenticated otherwise.



tion: A senior web developer with rest in usability. Profoundly interested in the human side of computing and making computers nicer for people to use. Fond of the saying, "People who think they hate computers

We will go over both server-side and client-side details to do this. The initial search page will present a regular search interface whether or not a user is logged in. If a user is not logged in, the screen should be dimmed and the user should see an Ajax modal dialog allowing the user to log in, as in the screenshot.

In this chapter we will look at both server-side and client-side code, covering:

- `admin.py`, used to have functionality called once and only once on the server-side.
- `functions.py`, with different utility functions, such as the `@ajax_login_required` decorator.
- `views.py`, which has functions that render web pages and the server's responses to Ajax requests. One view shows how we can slowly build a view by hand, and another shows an excellent Django shortcut.
- `style.css`, which has basic styling for utilitarian needs and usability.

- `search.html`, which has basic client-side Ajax for our search. Normally Ajax is factored out into its own files; this could be done to some degree but we have chosen an implementation that means that settings changes are centralized to the top of `settings.py`. Part of our solution entails JavaScript files that are populated as Django templates rather than being completely static, separate files.
- The Django admin interface, which we can use to create test data.

Let's begin!

admin.py: administrative functions called once

In `admin.py`, we have certain things that we wish to have called once and only once. This sets up the admin interface to be able to handle Entities:

```
import django.contrib.admin
import directory.models

django.contrib.admin.autodiscover()
django.contrib.admin.site.register(directory.models.Entity)
django.contrib.admin.site.register(directory.models.Location)
```

functions.py: project-specific functions, including our @ajax_login_required decorator

This is our `functions.py` file, which incorporates a `StackOverflow` solution. Traditionally in Django the `@login_required` decorator is placed before a view to have Django check to see if a user is authenticated and request authentication the old-fashioned way if a user is not logged in. This `@ajax_login_required` decorator works along the same lines: it is intended for Ajax-oriented views that return JSON, should pass along the view's output if the user is authenticated, and should otherwise output JSON with `not_authenticated` as `TRUE`.

On the client side, the appropriate behavior is to take a returned JSON value, and see if it has `not_authenticated` defined as `TRUE`. If so, the client should behave in an appropriate fashion to the user not being authenticated. In our case, we will present a modal dialog form via jQuery UI, which will allow a user to log in and eventually register for an account. Note that we should allow an escape hatch, as sometimes the user would rather not register. However important the website may be to us, it may not be so important to all users, and we want to gracefully handle the case where the user isn't interested enough to log in or register.

```
from django.http import HttpResponse

import json

def ajax_login_required(view_function):
    def wrap(request, *arguments, **keywords):
        if request.user.is_authenticated():
            return view_function(request, *arguments, **keywords)
        output = json.dumps({ 'not_authenticated': True })
        return HttpResponse(output, mimetype = 'application/json')
    wrap.__doc__ = view_function.__doc__
    wrap.__dict__ = view_function.__dict__
    return wrap
```

This is a good working example of creating a decorator. Its docstring (`__doc__`) and dictionary (`__dict__`) are assigned to be those of the inner function, and the inner function is returned. We see this decorator used in the first function in `views.py`, as will be shown in the following section.

views.py: functions that render web pages

The pattern that has been perhaps the most popular, the most used, and the most misused is the MVC pattern. Here there is a separation of concerns between the Model, the View, and the Controller. Theoretically, the concerns are separated.

In Django, the most common pattern is the MTV pattern, where the three letters mean Model, Template, and View. We have edited models before, and we will be using a simple template later on in this chapter. Here we will be looking at two views. At least in their classical definitions, a Django MTV view is not quite the same thing as an MVC view. While MVC and MTV models may be doing basically the same thing, a Django MTV view has a job description that includes some responsibilities of an MVC controller. There are many similarities between MVC and MTV, and both make a careful separation of concerns, but the division of labor is different and the Django view has some of the responsibilities of both the view and controller under MVC.

```
#!/usr/bin/python

from django.contrib.auth import authenticate, login
from django.core import serializers
from django.http import HttpResponse
from django.shortcuts import render_to_response
from directory.functions import ajax_login_required

import directory.models
import json
import re

RESULTS_PER_PAGE = 10

def ajax_login_request(request):
    try:
        request.POST[u'login']
        dictionary = request.POST
    except:
        dictionary = request.GET
```

The preceding code is important for debugging purposes: it checks for a basic key first in `request.POST`, defaulting to `request.GET` if this is not found.

The benefit of this is not relevant to production, and more pointedly it should not be used in production, where credentials should be submitted via `POST` over SSL. In production Ajax calls to this function should always use `POST` rather than `GET`, but for debugging purposes it can be highly desirable to fall back to GET. The reason is to provide one way of dealing with an uncaught exception. If there is an uncaught exception and Django has `DEBUG` set to `True`, it will serve up a detailed and informative error page. Unfortunately, if jQuery is expecting JSON and gets a detailed HTML error page, it will offer a singularly uninformative account of the error, which is not useful for debugging.

As a workaround for this phenomenon, you can manually make `GET` requests for at least some queries. If, for example, the base URL is `http://127.0.0.1:8000/ajax/login` for this view, you can add `GET` data at the end and manually visit a URL like `http://127.0.0.1:8000/ajax/login?login=mylogin&password=mypassword`.

If your system isn't working correctly, and your (POST) Ajax query is throwing an error, manually querying by GET in a browser window can be a good way to access a detailed error page designed to help pinpoint what exact exception is being thrown, and where.

```
        user = authenticate(username = dictionary[u'login'],
                            password = dictionary[u'password'])
        if user and user.is_active:
            login(request, user)
            result = True
        else:
            result = False
        response = HttpResponse(json.dumps(result),
                                mimetype = u'application/json')
        return response
```

The authenticate() and login() functions do different jobs. For Django's normal (non-Ajax) login procedures, inactive accounts cannot log in. But here we need to check for ourselves that authenticate() returns a valid user, and furthermore that the valid user is_active (that is, has not been marked inactive). This function both logs the user in (if appropriate) and then returns a Boolean value telling whether the login has been successful. It is recommended procedure not to delete user accounts, but set them to inactive. This best practice avoids creating holes in the database where something refers to a user who is no longer available.

The following function is protected by the @ajax_login_required decorator. It is relatively long; let us walk through it:

```
    @ajax_login_required
    def search(request):
        try:
            query = request.POST[u'query']
            dictionary = request.POST
        except:
            query = request.GET[u'query']
            dictionary = request.GET
```

Here we repeat what we have discussed before: besides the decorator, we look for POST but default to GET to make debugging easier. Note that the debugging information is exposed only if settings.py has DEBUG set to TRUE. In general, it is bad security practice to give information about internals in error messages. And if Django's settings.py has DEBUG set to FALSE, it will shut off detailed error messages and give generic messages intended to avoid exposing your program's internal structure. However, when we are developing with DEBUG set to TRUE, this gives us a chance to load an equivalent to a query that triggered an error by typing the equivalent `GET` query in our browser. This could be made a `DEBUG` only feature, but it may not be clear what the advantage or added security would be to that.

```
        split_query = re.split(ur'(?u)\W', query)
```

Here are a couple of notes on syntax for this regular expression operation. We have been using strings like `u'test'` to indicate a Unicode string. Here we want to give a raw string, as is usual for regular expressions. This means that (in this case) the backslash will not be interpreted, as `u'(?u)\W'` will be interpreted simply as the backslash escaping the W, which doesn't do anything particularly interesting, giving a string that will effectively be treated as `u'(?u)W'`. We could achieve the intended effect by giving `u'(?u)\\W'`, but manually keeping track of extra backslashes is a consolation prize as a solution compared to simply asking Python to interpret the string as raw. This is done by changing the `u` to `ur`; note that the `u` comes first: we want `ur'(?u)\W'`, not `ru'(?u)\W'`; the latter will cause a syntax error. It is generally recommended to make a practice and habit of using raw strings when dealing with regular expressions.

In the regular expression, we pass the Unicode flag with `(?u)` at the beginning of the string. This will cause the `\W`, the regular expression sequence for a non-word character, to recognize Unicode word versus non-word characters, rather than (for instance) recognizing only ASCII word characters and treating other alphabets, ideograms, and so on as non-word characters.

The possibility of consistently doing this in Python is why we are doing, in Python, the kind of work that Django developers often offload to the databases. The database backends offer inconsistent support for Unicode-sensitive word breaks.

```
        while u'' in split_query:
            split_query.remove(u'')
```

We are interested in blocks of word characters; the regular expression call may leave both the blocks of word characters we want, and empty strings. We remove the empty strings to get the sequences of word characters included in the search string.

This solution may not separate words for languages where one character represents one word rather than a sound or part of a word. If a workaround is needed, users can put a space between each word. (But if this is a significant concern, it would be better to adapt the solution so that people can give search terms in the way that is natural to them, and the computer seems to auto-magically do the right thing.)

```
        results = []
        for word in split_query:
            for entity in directory.models.Entity.objects.filter(name__
    icontains = word):
```

```
                if re.match(ur'(?ui)\b' + word + ur'\b', entity.name):
                    entry = {u'id': entity.id, u'name': entity.name,
                      u'description': entity.description}
                if not entry in results:
                    results.append(entry)
```

For each word in the split query, we search for objects containing the word, and then if the word matches, we add it to our list of results if it is not already there.

The `directory.models.Entity.objects.filter(name__icontains = word)` is a brief example of the kind of workhorse one uses in Django's ORM; the usual correct solution would be based on that. Here we do a case-insensitive search for objects containing the word. So if the word is "ed", this will turn up results containing the names "Ed", "Ted", "Edna", and other similar names. For the purposes of this discussion, we only want exact matches, so we perform a regular expression check: `(ui)` requests both "Unicode aware" and "case-insensitive," and `\b` specifies a word boundary, so "Ed" will match "ed" while "Ted" and "Edna" will not.

```
        for entry in results:
            score = 0
            for word in split_query:
                if re.match(ur'(?ui)\b' + word + ur'\b', entry[u'name']):
                    score += 1
            entry[u'score'] = score
```

This is a bare-bones scoring algorithm, and one of many places where you should be able to play around with the provided starting point and make it better.

```
        def compare(a, b):
            if cmp(a[u'score'], b[u'score']) == 0:
                return cmp(a[u'name'], b[u'name'])
            else:
                return -cmp(a[u'score'], b[u'score'])
        results.sort(compare)
```

This defines a function that sorts first by score, and then by name in alphabetical order.

```
        try:
            start = int(dictionary[u'start'])
        except:
            start = 0
        try:
            results_per_page = int(dictionary[u'results_per_page'])
        except:
            results_per_page = RESULTS_PER_PAGE
        returned_results = results[start:start + results_per_page]
        response = HttpResponse(json.dumps([returned_results,
          len(results)]),
          mimetype = u'application/json')
        return response
```

Here we take a slice of the result set, and send a response with a JSON encoding of the appropriate slice of a result set (so the client can obtain page 2 at 10 results per page, or page 3 of 50 results per page, and so on and so forth), and the total number of results. Both of these are hooks meant to support pagination. The default behavior if the client does not specify our `POST`/`GET` fields for pagination is to return the first 10 results (or all results if there are less than 10) and the number of results total.

```
    def homepage(request):
        return render_to_response(u'search.html')
```

This two-line view is a Django shortcut in action, and an example of how Django can save us time.

style.css: basic styling for usability

The CSS we use is straightforward. There is little ornament, but there is styling to make text easier to read, such as requesting a font that was very carefully designed for readability on screen: one of the best fonts optimized for usability. Some of the styling is utilitarian: the form to log in via Ajax is not shown by default. There is a `notifications div`, which has certain styles, and the text input for the search query is prominent and large:

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

    #login_form
        {
        display: none;
        }

    #notifications
        {
        background-color: #ff8080;
        border: 3px solid #800000;
        display: none;
        padding: 20px;
        }
```

```
#search_form
    {
    margin-left: 40px;
    }

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

search.html: a template for client-side Ajax

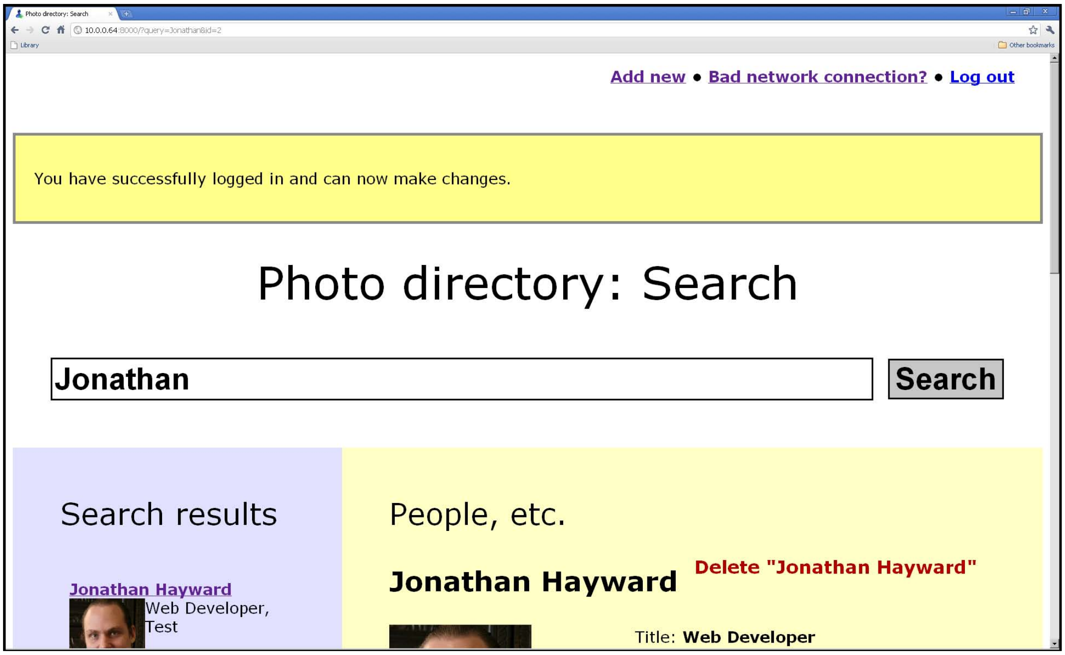
Next we will go through the template that defines the search page. We note in advance that while this template gets a lot done, it is not a showcase of Django templating features. The Django templating engine is intended for designers and allows designers to access variables, perform looping and conditionals, and other major features while not being given the power to execute arbitrary code. But even without harnessing most of the templating engine's features, we can still get quite a lot done.

```
{% extends "base.html" %}
{% block head_title %}Photo Directory: Search{% endblock head_title %}
```

The first line states which template it extends, and the second gives it a title. The code that follows contains HTML, and in many cases hooks for Ajax, to serve as a base for client-side work.

```
{% block body_main %}
<div id="notifications"></div>
```

The `notifications div` is meant to serve as a hook that will be available for Ajax manipulations as a message area. With a bit of Ajax shown later on, notifications will fade in, stay there for five seconds, and fade out. When a notification is fully faded in it can look like the following:


Add new • Bad network connection? • Log out

You have successfully logged in and can now make changes.

Photo directory: Search

Jonathan

Search

Search results

Jonathan Hayward
Web Developer,
Test

People, etc.

Jonathan Hayward

Delete "Jonathan Hayward"

Title: Web Developer

The HTML is:

```
<form name="search_form" id="search_form" action="/">
  <input type="text" name="query" id="query" /><br />
  <input type="submit" name="submit" id="submit" value="Search" />
</form>
```

This is the search form. In keeping with semantic markup, we define the form semantically, and then use jQuery to submit searches and do partial page updates with the results. Note that every HTML element has an ID. This is not required for semantic markup as such, but it helps for styling elements via CSS when desired, for customizing behavior via jQuery or other Ajax tools, and for graceful degradation.

Although we have not implemented server-side support for this, the form as it stands could work perfectly well for a graceful degradation strategy to let people search the old-fashioned way with JavaScript turned off. (Such support will be added later.)

```
<div id="results"></div>
```

Like the `notifications div` seen earlier, this is a hook for Ajax to manipulate and, in this case, populate with results.

```
<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 `div`, initially hidden, is used for Ajax login. For demonstration purposes, the directory is only searchable to users who are logged in, and this is the login form that is displayed when someone who is not logged in tries to search.

If you wish the search functionality to be available regardless of whether a user is logged in, simply comment or remove the `@ajax_login_required` decorator before `search()` in `views.py`.

In the next two lines, we indicate that we are done with content for the `body_main` hook, and are moving on to the page-specific JavaScript hook, `footer_javascript_page`.

```
{% endblock body_main %}
{% block footer_javascript_page %}
```

We include a JavaScript file, and then write functionality for this page. The main jQuery script has already been included in the `base.html` Django template; we don't need to do anything more for that here.

The following is a helper function. Right now, on both the server-side and client-side we have not discussed the several reasons a user might not be treated as logged in:

- No attempt has been made to log in
- The user has attempted to log in with an invalid account
- The user has attempted to log in to a valid account but got the password wrong
- The user has correctly authenticated to an account that does not have `is_valid` set

In terms of room for expansion, it might be helpful to distinguish which of these several options has occurred. Particularly in the Palaeolithic era, when programmers sat down on stone benches and logged in to monochrome terminals, it made sense to answer either an invalid login or password with "login incorrect" and avoid divulging whether the login or password is wrong. Now refusing to give any hint of an answer to "Did I get my login or password wrong?" is needless and usually doesn't do much to improve websites' security. One way this program could be improved is for the server and client to distinguish between possible reasons an attempted login did not succeed.

```
<script language="JavaScript" type="text/javascript"
        src="/static/js/jquery.effects.fade.js"></script>
<script language="JavaScript" type="text/javascript">
<!--
function check_authentication(parsed_json)
    {
    if (parsed_json.not_authenticated)
        {
        return false;
        }
    else
        {
        return true;
        }
    }
```

The following function is a less generic helper function. It mostly draws on functionality that is defined elsewhere:

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

The following function kicks off a notification appearing and disappearing—with animation—in the notifications area. `setTimeout()` is called so that `send_notification()` will return control to its caller quickly, instead of slowing down any caller by five seconds plus the duration of two animations:

```
function send_notification(message)
    {
    $("#notifications").html("<p>" + message + "</p>");
    setTimeout(function()
        {
        $('#notifications').show('slow').delay(5000).hide('slow');
        }, 0);
    }
```

The following function encapsulates the search submission. Its `success` function does not assume that the user is logged in, because a successful `XMLHttpRequest` call could happen whether or not the user is logged in, and could be search results, but could only be a message saying that the user is not authenticated.

If the user is not authenticated, it offers the *option* of logging in. That part is indirectly recursive in that a successful login from the `offer_login()` call will resubmit the search and should pull the results. Whether or not the implementation details could be done differently, this is the correct user interface behavior if login is required to do a search:

1. The search form is presented.
2. If the user is not authenticated, the user is given the option of logging in, but is also free to decide that this isn't worth logging in.
3. If the user authenticates successfully, the flow of activity continues from where it was left off. The user doesn't have to hit **Search** again, retype the query, or otherwise start over on the request. Instead, on successful authentication, the flow of activity continues without disruption as if the user had already been authenticated.

The function encapsulates a call to jQuery's low-level `$.ajax()`. This is an option worth considering in this case, but in many cases higher-level functions will be called, with a signature like `$("#messages").load("/messages/current")`.

```
function submit_search()
    {
    $.ajax(
        {
        data:
            {
            query: document.search_form.query.value
            },
        datatype: 'json',
        success: function(data, textStatus, XMLHttpRequest)
            {
            if (data)
                {
                if (check_authentication(data))
                    {
```

If `data` is "real" data, instead of a "not authenticated" message, we use jQuery to display the results. The system is designed with an eye to allowing pagination, but for now we simply display all results. The following code illustrates how one may manipulate the DOM using Ajax. Another approach would be to build the HTML constructed here on the server, perhaps make the HTML as a string stored in `data.html`, and simply call something like `$("#results").html(data.html);`. Both approaches can work, although we will focus on Ajax DOM manipulation:

```
                    $("#results").html("");
                    var results = data[0];
                    var length = data[1];
                    for (var index = 0; index < results.length;
                      ++index)
                        {
                        var result = results[index];
                        $("#results").append("<p><a href='/entities/"
                          + result["id"] + "'>" + result["name"] +
                          "</a><br />" + result["description"] +
                          "</p>");
                        }
                    }
                else
                    {
                    offer_login();
                    }
                }
            },
        type: 'POST',
        url: '/ajax/search',
        });
    }
```

The `$(function(){...});` in the following code specifies functionality to be executed when the DOM is ready. Note that this is not identical to the `onload` event because the DOM can be ready before images are loaded, and this will be executed sooner. An arbitrary number of functions can be requested this way; we have wrapped the function calls we want in one anonymous outer function, but it would have been equivalent to use two `$(function(){...});` calls, one per wrapped call. However, doing things this way and specifying a `body onload` do not work together; we should do things this way when we want something executed when the DOM is available and ready.

The following `function()` call is a bare bones example of **hijaxing**: replacing an element's default behavior on a given event with an Ajax effect. In this case, the **submit** button, which will perform a Web 1.0-style form submission and whole page replacement on a click, is hijaxed with a function that, in our case, calls a function, and then returns `false`, which prevents the default event handling from taking place. Unlike other facets of JavaScript, we cannot return `0`, `''`, or other non-true values besides `false` and achieve the same effect; we specifically need to return `false`.

```
$(function()
    {
    $("#submit").click(function()
        {
        submit_search();
        return false;
        });
```

The text input field is shrunk to make room for the **submit** button.

This is an example of using jQuery to get pixel-perfect CSS-style effects. The intended effect is that the query field and the **submit** button make up one line, and that one line is centered. This may degrade if the user has selected a larger or smaller font, but the basic effect of two items, with the query text input sized so that together with the **search** button it is centered, is an example of presentation that is more easily achieved with jQuery than CSS.

```
        $("#query").width($(window).width() - 240);
```

The following call is based on jQuery UI rather than jQuery core alone, and uses the `login_form div` as the basis for a modal dialog. It adds a **Log in** button, which will attempt to log in and submit the search on being clicked, and presently unimplemented **Forgot password** and **Create account** buttons, which should be created for production purposes and should be doable by the same principles as the server-side and client-side code enabling Ajax login.

```
    $("#login_form").dialog({
        autoOpen: false,
        height: 150,
        width: 350,
        modal: true,
        buttons:
            {
            'Log in': function()
                {
                $.ajax({
                    data:
                        {
```

```
                        login: $("#login").val(),
                        password: $("#password").val(),
                        },
                    datatype: 'text',
                    success: function(data, textStatus,
                                      XMLHttpRequest)
                        {
                        if (data)
                            {
                            send_notification(
                                 "You have successfully logged in.");
                            $(this).dialog('close');
                            submit_search();
                            }
                        else
                            {
                            send_notification(
                                 "Your login was not successful.");
                            }
                        },
                    type: 'POST',
                    url: '/ajax/login',
                    });
                $("#password").val("");
                },
            'Forgot password': function()
                {
                send_notification(
                     "This feature has not been implemented.");
                },
            'Create account': function()
                {
                send_notification(
                     "This feature has not been implemented.");
                },
            },
        });
```

The following call registers a global error handler that will be called by default by all Ajax operations. At present this is an error handler for developers rather than end users. `textStatus` and `errorThrown` are the sort of innards that can be helpful for a developer but are not appropriate to inflict on innocent users. Depending on the browser, the `textStatus` and `errorThrown` may not even be the sort of thing that would help a developer. In my case they were, respectively, "error" and "undefined" in every case, although much more useful debugging information was presumably available from digging through the `XMLHttpRequest` argument's data.

In terms of a production-ready message, neither "error"/"undefined" nor digging up the response received by the `XMLHttpRequest` are appropriate. It would be better to send a notification of **There was an error handling your request. We apologize for the inconvenience**, or if you want to make a more sophisticated error handler, see what information can be obtained from the `XMLHttpRequest`, and then give a more customized user-friendly error like **We're sorry, but we had an error trying to save your information. The development team has been alerted and will try to address the root problem. If you need help, please contact [email/phone/contact link]**.

```
        $.ajaxSetup(
            {
            error: function(XMLHttpRequest, textStatus, errorThrown)
                {
                send_notification(textStatus + "</p><p>" +
                                  errorThrown);
                },
            });
        });
    // -->
    </script>
    {% endblock footer_javascript_page %}
```

The Django admin interface

The Django admin interface is a full Create, Read, Update, and Delete (**CRUD**) capable interface. The basic philosophy behind the interface is that some forms of Create, Read, Update, and Delete operations need to be done in project-specific ways, which are often interesting challenges, and the Django admin interface is not intended for that purpose. What the admin interface is intended for is to address certain routine boilerplate Create, Read, Update, and Delete functionality, so that the more chore-like features are taken care of for you well.

The steps to make the admin site in Django 1.2.x available are:

1. Go to your `settings.py` and ensure that `INSTALLED_APPS` contains `django.contrib.admin`, `django.contrib.auth`, and `django.contrib.contenttypes`.
2. Put a line like the following in your `admin.py` file for every model that you want to be available via the admin interface:

   ```
   django.contrib.admin.site.register(directory.models.Entity)
   ```

3. Include in your `urls.py` the following lines, firstly at the top:

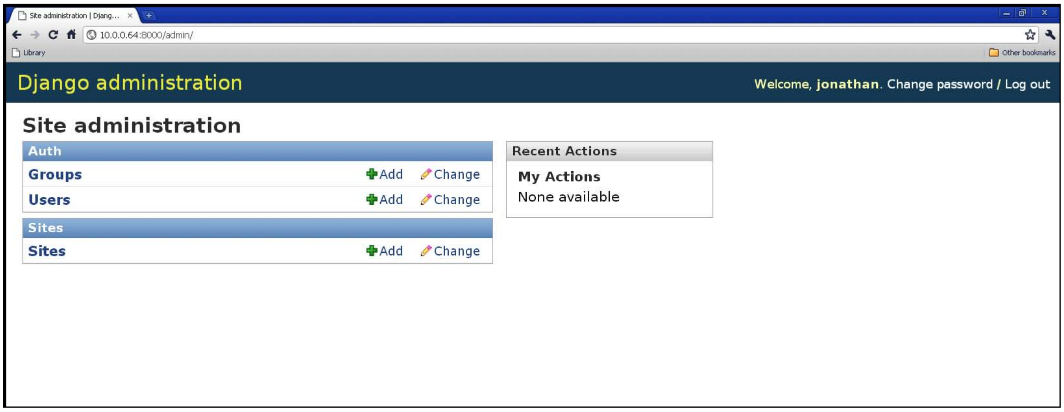
   ```
   import admin
   ```

   And in your `urlpatterns` include the following line or anything of its equivalent:

   ```
   (r'^admin/', include(admin.site.urls))
   ```

4. Restart the server.

After that, we should see a screen like the following once we have logged in with the user we've created:



Django administration

Welcome, **jonathan**. Change password / Log out

Site administration

Auth		
Groups	Add	Change
Users	Add	Change

Sites		
Sites	Add	Change

Recent Actions

My Actions

None available

We can then click on **Entity** (or **Locations**), and add an entity. We have set things up so all fields are optional.

We will be working later on to make a more responsive, Ajax-based interface with in-place editing. It has been said, "Where there is output, let there be input." Ideally, if someone wants to change information that is displayed, and that person has privileges to do so, it should be possible to click on that information and update it there, rather than scout out what other part of the site is responsible for giving input. However, the Django admin interface is decent, and we can use it for test data in particular.

Summary

With all the pieces put together, you should be able to use the Django admin interface to create test data, and then search and see a basic results page. There is room to expand and improve in almost all of what we have done, and we invite you to tinker around and make something better. But this is a substantial working system, and we have dealt with real and serious code.

We have covered `admin.py`, for functions that we want called exactly once when the server starts. We have also covered `functions.py`, a file we have created for utility functions and in particular an `@ajax_login_required` decorator that requires users to authenticate before seeing a view, but unlike the standard `@login_required` workhorse is optimized to respond appropriately to Ajax clients expecting JSON rather than web browsers rendering whole HTML pages. We have covered `views.py`, which defines two views. Again, these are views under Django's MTV Model-Template-View division of labor and separation of concerns, and not the classic MVC Model-View-Controller separation of concerns. We have covered `search.html`, a template that renders the desired web page without even needing to use many Django templating engine features. We have also covered the Django admin interface, which lets us handle routine Create, Read, Update, and Delete operations almost for free. We can (and in this case will) work on another interface, but for now we have a powerful and friendly way to create test data and one that would serve us well for production purposes if we wanted.

jQuery is a library designed to allow an ecosystem of plugins. In our next chapter, we will look at a snazzier interface that heeds the words, "Where there is output, let there be input." We will use a jQuery plugin to allow in-place Ajax editing, giving a Web 2.0 shine. Furthermore, we will log changes: in terms of successful intranets, companies that turn on anonymity tend to turn it off very quickly. Making the directory more like a wiki can result in better and more up-to-date information. This isn't the only option, but it is the one we will explore next.

On to in-place editing!