4

Server-side Database Search with Ajax

In this chapter we will cover some of Django's database and persistence basics, create a model, and create an Ajax-oriented search for that model. We will build on the observation that context switching between Python and SQL, especially if it is frequent, bears a "cognitive tax" and makes for buggier code. The way the Django developers have tried to offer an improvement is to create features so that most of the work addressed by writing SQL can now be managed without disrupting a programmer from "Python mode." We will both take an overview of the basics and then see a simple case of these basics in action.

More specifically, we will:

- Compare client and server roles and discuss reasons for a server-side home to search functionality
- Introduce Django models, and build an in-depth sample Django model
- Begin work on our intranet employee photo directory
- Discuss what kind of functionality is desirable in a search tool
- Create a Django view to serve as the JSON backend part of the search tool
- Tour the Django database search functionality, and discuss how it is similar to things we have already seen in jQuery selectors

Let's take a look at searching on the client or server side.

Searching on the client side and server side

There exist powerful options for an in-memory client-side JavaScript database. This much means that the difference between server-side and client-side options is not a choice between excellent and terrible. However, two features are worth considering.

Persistence: Much of the attraction and reason for using a server-side database is that it offers persistence. Having a database that does not support persistence defeats the point, like a car without any room for passengers or the driver. This is not to say that a client-side JavaScript database cannot have persistence. With Ajax, it is possible to create a persistence solution. However, it *is* to say that creating a server-side database is a solution to the persistence problem, and a client-side in-memory database is *not*.

Scaling: For the application we are using, and for personal and test use of that application, delivering everything in the database we might possibly need except the files themselves is not that terribly big a download. For some production purposes, "everything in the database we might possibly need" is still not that big a deal in terms of download speed or memory footprint. *However,* for many production purposes the database becomes large enough to make delivering an in-memory database problematic. In many cases the recommended best practice is to work hard to be as lazy as you can. In other words, the best practice is not to have the client download what might sometime be needed, but download what actually *is* needed on a "just-in-time" basis. A production database is designed so that if anticipated or unanticipated reasons cause the dataset to grow ten times bigger, performance will not necessarily take too much of a hit. But if it does, there are things that can be done to compensate. If you define your solution to deliver an in-memory database, the problems with a tenfold increase in the dataset are not so easily resolved.

Let us explore the server side first.

Handling databases through Django models

The power of Ajax is derived partly from storage of information on the server-side, quite often in a database. Django provides an excellent platform for handling the database side.

Django attempts to offer certain facilities and then let you go your own way if you want. It is not as opinionated a framework as **Ruby on Rails**. You can choose, if you want, to use other packages like **psycopg2** and **sqlite3** and bypass Django's persistence facilities as much as you want, and you can also handle raw SQL while

using Django's persistence facilities. However, Django database handling is very carefully thought out, and it is well worth taking the effort to understand.

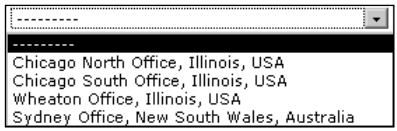
One Pythonic observation feeding into Django's approach is that frequent context switching bears a "cognitive tax," even if the templating is done by a developer who handles the models, rather than the web designers Django's templating is designed for. The Django approach is meant to spare the developer the cognitive strain of repeatedly switching between Python and HTML on a single screen of code. With SQL the cognitive tax is even more important. Repeated switching between Python and SQL introduces a strain that makes it easier to introduce bugs and then easier for the bugs to pass unnoticed. Django's approach is meant to remove this strain and allow for persistence to be handled without switching out of "Python mode."

Let us look at this more concretely.

Models for an intranet employee photo directory

Let us look at the models for an employee intranet photo directory. For our directory we will attempt to cover routine information for employees, such as office, extension, tagged skills, and so on. In general, solving a problem correctly in Django means taking advantage of existing working parts, those that come standard with Django and those that are available. For the purposes of an intranet employee photo directory, it might make good sense to use a Pinax social network as a basis. Our reasons for not using Pinax are not mainly due to its deficiencies, but because we need a sample application that will let us tour Django and Ajax. Pinax may solve enough of the problem for us that we would not have as informative a tour. This is not a criticism of Pinax; it's just a statement about what makes a good sample application that will get the reader's hands dirty with the power of Django and Ajax.

The `OFFICE_CHOICES` list is only illustrative; we expect that it would be replaced for production code. This one uses a two-character code. If you were making a choice like this, you could use one, or three, or ten. The short form is stored in the database and the long form is displayed in the example dropdown menu:



The code defining this choice, and the beginning of our models.py source file, is:

```
#!/usr/bin/python

from django.db import models
import datetime

OFFICE_CHOICES = (
  (u'CN', u'Chicago North Office, Illinois, USA'),
  (u'CS', u'Chicago South Office, Illinois, USA'),
  (u'WH', u'Wheaton Office, Illinois, USA'),
  (u'SY', u'Sydney Office, New South Wales, Australia'),
  )
```

The `ExtensionField` in this case is implemented as a `TextField`, but this may be one case where a `VARCHAR`-based `CharField` makes sense to store, for example, a four or five digit extension field can be safely assumed, and is possibly deeply entrenched. The existing `ExtensionField` is:

```
class ExtensionField(models.TextField):
    pass
```

If you wanted to specify an extension of up to five digits, you could declare a field like the following:

```
import django.forms
import re

EXTENSION_LENGTH = 5

def is_extension(number):
    if len(str(number)) > EXTENSION_LENGTH:
        raise forms.ValidationError(u'This extension is too long.')
    #elif len(str(number)) < EXTENSION_LENGTH:
        #raise forms.ValidationError(u'This extension is too short.')
    else:
        return text

class ExtensionField(PositiveIntegerField):
    default_error_messages = {
      u'invalid': u'Enter a valid extension.',
      }
    default_validators = [is_extension]
```

(The commented-out lines, if uncommented, enforce that the extension is exactly `EXTENSION_LENGTH` digits long, instead of being at most that length.)

Returning to the code, we have a `Location` model, which has several fields. Note the `required = False`: Django's default behavior is to treat all fields as required, and in this case we want to support several kinds of information without requiring the user to fill them in before continuing. The `notes` field is for any particular notes on a location. The `office` field is a `CharField` populated from a choice, meaning that if it is filled in, it will be one of the remaining values in the choice above. If you want a choice for a U.S. state, you don't need to reinvent the wheel, just import and use `django.contrib.localflavor.us.forms.USStateField`. And if that sort of thing wasn't covered, it would make sense to make a `CharField` of length `2` populated by a straightforward choice letting people pick full state names and guaranteeing that any value stored in the database was one of the two that were entered. Besides that issue, companies around the world have a use for specifying a department, product line, or other item where there is a fixed list. It may be less of a real strength that one can save a few bytes by using an abbreviation for database storage and other internal use. However, a choice like this is a useful and powerful tool for our toolbox, and it's nice that Django allows us to decouple what is displayed from how it is represented internally.

Again, the `Location` object is designed to allow and encourage information to be filled, instead of requiring everything be given up front:

```
class Location(models.Model):
    notes = models.TextField(required = False)
    office = models.CharField(max_length = 2,
                              choices = OFFICE_CHOICES,
                              required = False)
    postal_address = models.TextField(required = False)
    room = models.TextField(required = False)
    coordinates = GPSCoordinate(required = False)

class TextEmailField(models.EmailField):
    entity = models.ForeignKey(Entity)
    def get_internal_type(self):
        return u'TextField'

class TextPhoneField(models.TextField):
    number = TextField()
    description = TextField()
    def __eq__(self, other):
        try:
            return self.remove_formatting() ==
                   other.remove_formatting()
        except:
            return False
```

```
        def remove_formatting(self):
            return re.sub(ur'\D', u'', str(self))

    class TextURLField(models.URLField):
        def get_internal_type(self):
            return u'TextField'
```

We have text-based fields that work similarly to Django's `EmailField` and `URLField`, but in the database are represented by `TEXT` fields instead of `VARCHAR`, which breaks the zero-one-infinity rule.

One specific design decision that needs to be explained is why the following class is not called the `Person` or `Employee` class, but the `Entity` class. Scope creep is one of the facts of life in the programming world: if something is working, people want it to do more. The same insight lies behind the saying, "A successful tool is one that is used to do something its original creator never imagined." The baseline of success for this project is whether it works as an intranet employee photo directory, and if it doesn't, it fails, period. If it succeeds, people are going to want to put information besides specific employee details into the database, *guaranteed.* While no amount of foresight will get us a product that is officially certified to be future-proof, if we design for flexibility from the beginning, we will less certainly have painted ourselves into a corner.

Calling the class `Entity` by itself sounds like a move Dilbert's boss would encourage. But the class is designed for flexibility, and the name `Entity` is a carefully chosen reminder that the directory may include not only employees, but potentially departments, customers, events, or other things we cannot foresee. Perhaps some of these are more likely to happen than others. It may be unlikely that a company with any kind of serious CRM solution would want to be duplicating that kind of information in their employee directory.

If the people using our software—who will *rarely* do what we expect, and almost *never* do what we intend them to do—decide to store customer contact information in our directory, we want our solution to behave as gracefully as we can manage. Calling the class `Entity` is a reminder that we are trying to make a flexible solution: a solution that behaves gracefully when users do things that go against our assumptions. And while we cannot guarantee something future proof, at least we can minimize imposing the kind of assumptions that make for gratuitous failure to be future proof. (We will give our users enough pain even if we try not to.)

```
    # This class is basically the "Person" class; however, it is called
    # "Entity" to emphasize that it is intended to accommodate people,
    # offices, organizational units, and possibly other areas.
```

```
class Entity(models.Model):
    active = models.BooleanField(required = False)
    department = models.ForeignKey(Entity, required = False)
```

Note that at this point we are trying to make, and internally "dog food," one entity class that should be flexible enough to serve as a department. What this last line says, in slightly opaque form, is that the department, *if one is specified*, is another entity:

```
    description = models.TextField(required = False)
    email = TextEmailField(required = False)
    extension = ExtensionField(required = False)
    homepage = TextURLField(required = False)
```

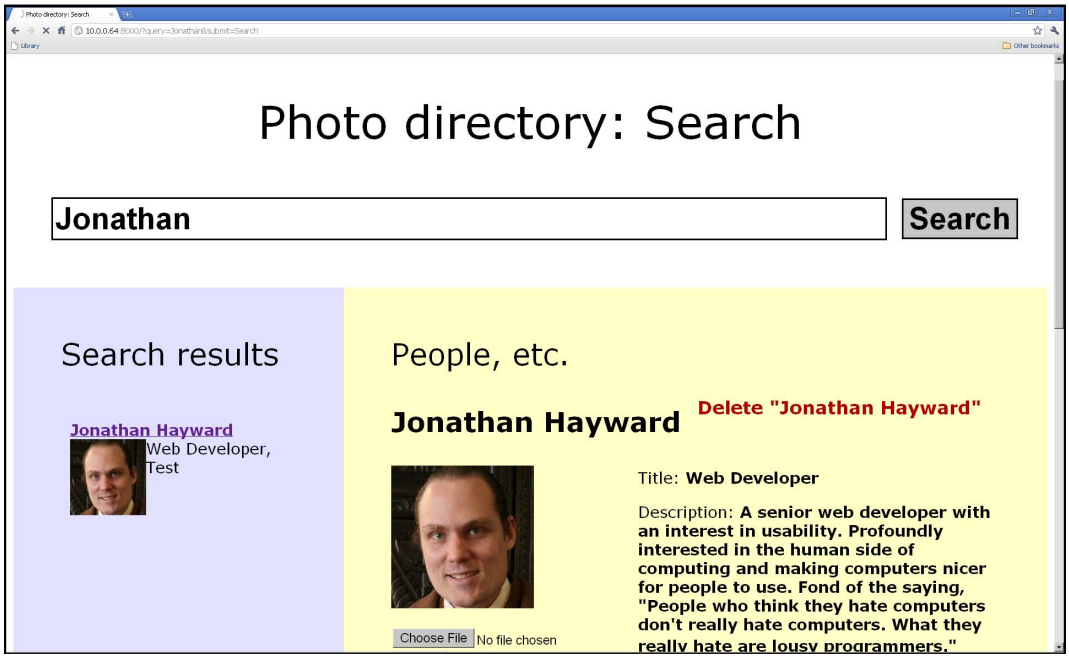
One remark here about not being control freaks and the last line: many corporations, at least the larger ones, have enough intranets with varying degrees of official support that "How many intranet sites do we have?" is the sort of question that no one likely knows the answer to. If you are in charge of a project on an officially sponsored intranet site, wonderful. But do not think in terms of the *real* intranet and then the unimportant, unofficial fluff.

If it makes sense in terms of your company to use a `homepage` field for personal homepages, Twitter feeds, and the like, that is one valid use of a `homepage` field. But even if personal links are discouraged, it makes excellent sense for the "official" intranet to have at least a little bit of a roadmap to connect people to "spontaneous intranet" sites. Perhaps a `homepage` field is not enough, and such functionality may or may not belong in a directory. Although, there would seem to be real added value in an official directory that had data on departments, organizations, and groups, *including* a link to the unofficial intranet sites that not only are not going away, but probably should not go away.

We will cover ways to associate more than one of X with a model. There could be a `Website` class created, with a URL and a displayed name, and then entities could list zero, one, ten, or more than twenty URLs. That would not be a bad idea; it is just something not included in this implementation but left as room for the reader to expand and tinker with the starting point provided here:

```
    image = models.FileField(required = False)
```

Since we are working on an employee intranet *photo* directory, this is an important field, and one comment is apropos here: we have chosen to allow at most *one* image. This is not the only option; it is a defensible choice to design to allow for a gallery with arbitrarily many photos, and one selected to be displayed first. In our case, it makes a page like:



The point underscored here is not that there is anything wrong with photo galleries (even if it may not be obvious that a mere photo directory needs to give Facebook's multiple galleries a run for their money). But in terms of the zero-one-infinity rule, both "at most one" and "as many as you want" are sensible design choices. The iPhone and iPad interface, with one app at a time, and the Droid interface allowing multiple apps, are *both* sound design decisions and are *both* part of a carefully designed interface, partly because they *both* respect the zero-one-infinity rule. We will not try to resolve the question of which is better, but we would pointedly suggest that *both* make sense.

If you want to make a version of the directory that allows arbitrarily many photographs, and more power to you, then this book is intended to put you in a position to hack and tinker. If your first action is to explore how you can change this design decision, that's excellent. We'll be glad that we helped get you tinkering.

```
location = LocationField(required = False)
honorifics = models.TextField(required = False)
name = models.TextField(required = False)
post_nominals = models.TextField(required = False)
```

In these fields we incorporate a name and how a person should be addressed, as discussed earlier. The `name` is a `TextField`, and there is an optional field for any honorifics, and an optional field for any letters after a person's name, whether for academic degrees or any other reason. (In this case `name` is also optional, as we are trying to let people create an empty initial entry and fill it in later, but of all the fields on the form, this may have the strongest case to be required.)

```
        publish_externally = models.BooleanField(required = False)
```

This is a field that is placed as room to grow. At present, the conceived use of the directory is one that will live behind a firewall and allow anyone on the trusted network area to read information, and appropriately authorized users to write. If it is additionally used to present information externally, this is used for an entity to "opt-in" to external placement. Note that in that case it may make sense for whitelisted fields only to be displayed.

In terms of the Agile "You ain't gonna need it," it would be entirely defensible to delete this field if you do not need it now. If you find you need it later, it can be added when it is needed, and omitting such features now does not paint any Pythonist into a corner if they turn out to be needed later on.

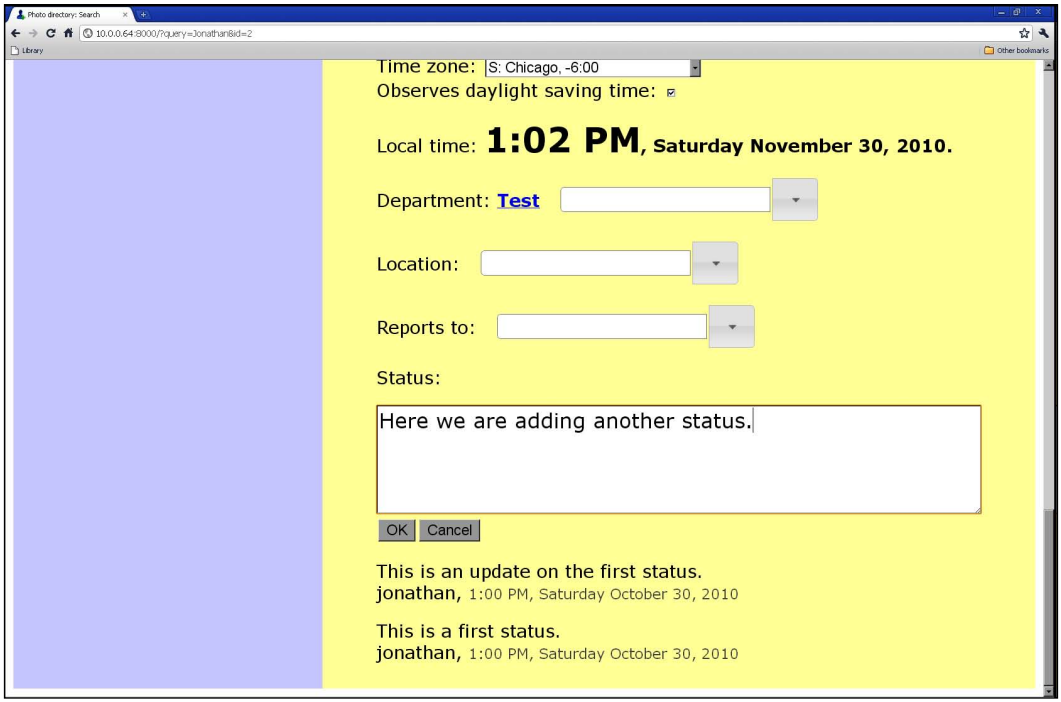
```
        reports_to = models.ForeignKey(Entity, required = False)
        start_date = models.DateField(required = False)

    # Tagging is intended at least initially to locate areas of expertise
    # tagging.register(Entity)
```

In Django-tagging, the way you make a model subject to tagging is not to add a field, but by calling `tagging.register()` on the model.

```
    class TextStatus(models.Model):
        datetime = models.DateTimeField(default=datetime.now)
        entity = models.ForeignKey(Entity)
        text = models.TextField()
```

An Entity may have many `TextStatus` tags; the intent in this field is to allow status within the company to be published. While not all corporate cultures would really have a use for such things, and certain corporate cultures could squelch any really useful functionality, having a Twitter-like status for current projects and the like could render very valuable internal communication services.

That could look like the following:



Note that no specific character limit is given, and it may make sense to post updates that are copies of the previous ones, updated where a change is warranted.

You are welcome to disable or delete this feature if it does not make enough sense in your organization.

Other many-to-one fields could be given. A many-to-one phone field is:

```
class EntityPhoneField(TextPhoneField):
    entity = models.ForeignKey(Entity)
```

If you want to support multiple images, the database backend could be handled by a model containing a `models.FileField()` and a foreign key as shown. You would presumably want to delete the single `image` field. Or, alternatively, the existing could be kept as a main image field. In addition, there could be galleries supporting as many images as desired.

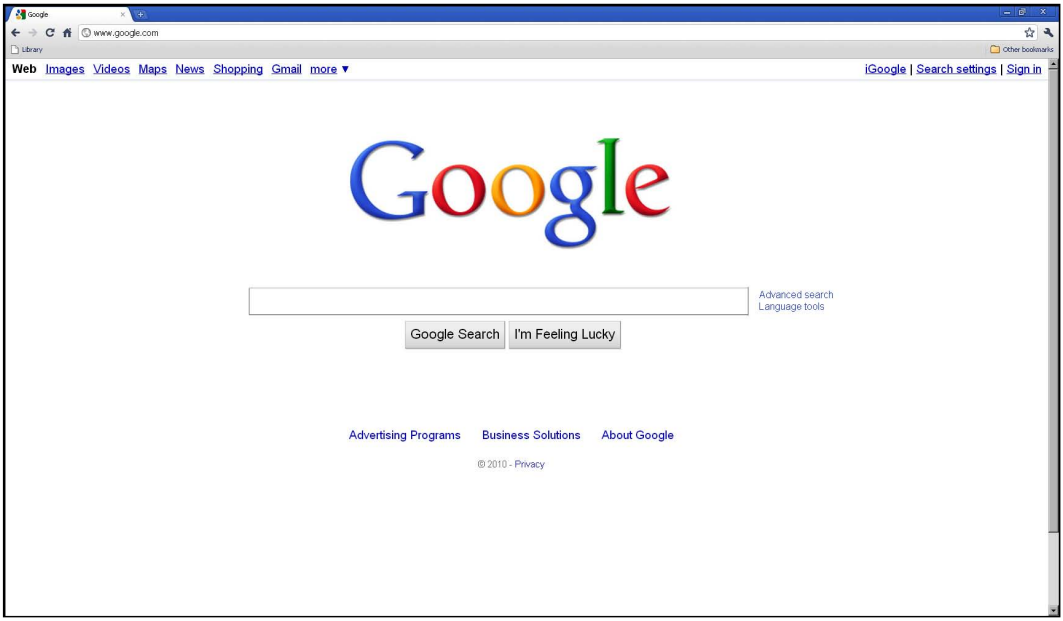
We might comment that making this change correctly would involve appropriately adapting the user interface. The bulk of the work in doing it right would be user interface work rather than making a couple of changes on the backend.

Searching our database

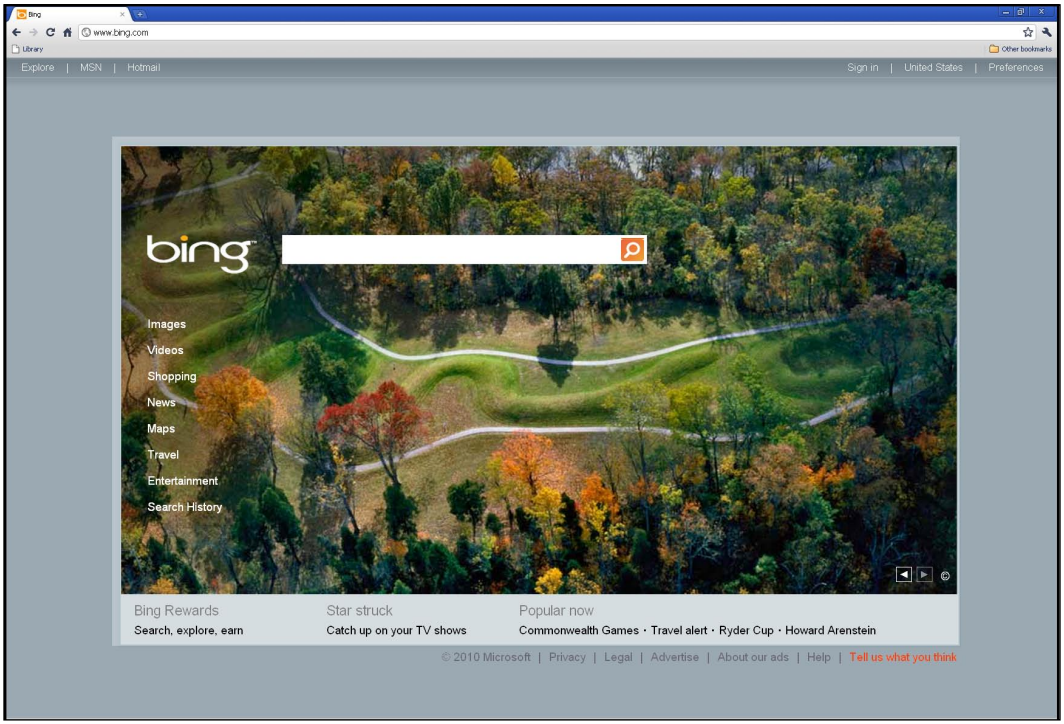
We will be making a deliberately limited basic search field, but one that is carefully chosen.

One basic user interface principle is that if there is a standard user interface, it is almost always the right decision to go with the standard interface rather than make something completely different, however clever. In terms of office software, that means the correct solution will be one that "feels like" Microsoft Office, and it is not an accident that at least before the ribbon, OpenOffice looked a lot like Microsoft Office. If some feature of the standard interface is patented, that throws in a monkey wrench, but it doesn't change the basic principle that the optimal interface is the one people expect.

In terms of searching, Bing has made inroads but Google is still the standard, and that rather cleanly answers the question of what we should be aiming for: one text input that people enter search terms into, no **Advanced search** to fight with, a **Submit** button you press to search, and then you get results, beginning with the ones that are probably most relevant. (Even if Bing's interface departs from Google's at points, they still have one query input field front and center, one button to press, and serve up a result page with the most relevant results first. Even if you're competing with Google, you will probably want to imitate Google on the all-important basics.) Compare Google with Bing:



And the Bing search page:



We will be aiming to achieve Google-like basics with Ajax. What this means is that while we will be implementing our search with partial page refreshes and Ajax internals, we will still aim for that familiar feel. Of course we might not be quite as good as Google in our implementation, but we have a very clear reference about what to aim for. If we shoot for a Google quality interface and miss, we might still land among the stars.

In terms of backend functionality we will take a deliberately restrained search and only search in one text field, the entity's name. A full-text search might be highly desirable but would be very easy to get wrong. For instance, a search for "Alice" could bury all employees named Alice under a mountain of employees who have ever registered a status of, for example, "Worked with Alice to investigate bug 1234. Marked RESOLVED INVALID." An appropriately done full-text search might include entities with a status containing the string "Alice", but it should presumably prioritize entities named "Alice" and place them earlier in the results.

If the tagging is used to denote employee areas of expertise, it would be very desirable to be able to search by tags, and we note as an area for improvement that our implementation does not address that desirable feature. Another desirable feature might be to make searches tolerant of spelling errors. The `py-editdist` package (`http://www.mindrot.org/projects/py-editdist`) is one package used to calculate the Levenstein edit distance between two strings, which can be used to guess a near-miss spelling match. Completing some of these features may involve doing work in Python beyond what is done through the Django database functionality, and in fact we will do this in our implementation. There are functions to match by regular expressions, but backend functionality varies from database to database, and an implementation that used the Django database regular expression functionality may work reasonably well with some database backends and break completely for non-ASCII Unicode names with others. We will be taking advantage of the features provided, before providing a discussion of other facilities Django offers, but we will also be writing our own code to provide a consistent and correct behavior across database backends.

What we want to do first is break the search string into words. This will be done by a Unicode-sensitive regular expression split into non-word characters, although this is not globalization-wise a perfect solution. Some languages have more or less one character per word, instead of one character per sound, and so separating words by non-word characters is not a universally assumed pattern. We will mark this as a known issue and move forward with this solution for lack of knowing a better obvious *and straightforward* solution.

We define a view:

```
#!/usr/bin/python

from django.core import serializers

RESULTS_PER_PAGE = 10

def search(request):
    query = request.POST[u'query']
    split_query = re.split(ur'(?u)\W', query)
    while u'' in split_query:
        split_query.remove(u'')
```

The `(?u)` passes a flag to be Unicode sensitive. With it, any letter/word character in any language is recognized as a word character. Without it, only some ASCII characters would be recognized, so the split query would fail to preserve non-ASCII letters.

Having found the query, we will do an OR search by searching for all the query words. However, this is one point where we will depart from standard use of the database. We will be doing a case-insensitive search for exact words, so that people with short names or parts of names don't spuriously match. We do not want a search for "ed"/"Ed" to register as an exact match for "Ted", "Edna", "Edward", and similar names. And Django's database facilities allow a case-insensitive search for a substring, but matching word boundaries correctly in Unicode appears to be inconsistently supported by database backends. We will be using Django's database facilities to pull results containing query terms (so "ed" will pull "Ed", "Ted", "Edna" and "Edward", among other possibilities), but we will enter Python territory by doing the equivalent of the above code. Out of *all* the database hits, *some* are ones we will want.

```
results = []
for word in split_query:
    for entity in Entity.objects.filter(name__icontains = word):
        if re.match(ur'(?ui)\b' + word + ur'\b'):
            entry = {u'id': entity.id, u'name': entity.name,
                     u'description': entity.description}
        if not entry in results:
            results.append(entry)
```

This is creating a list containing select fields, and this much (namely `Entity.objects.filter()`) is working with Django's persistence facilities normally. The next few lines of code are working with Python objects in memory, not Django persistence facilities, for reasons partly discussed, and also partly discussed in the following.

Later in this chapter, we will provide a more direct look at Django persistence facilities.

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

Two remarks before we continue with the function. What we have done has neutralized one of the major benefits of working with a database, a benefit that Django preserves: laziness, meaning that data are loaded on an as-needed basis. In the days of ancient computers where a 64 megabyte memory was an unimaginably vast dream, it meant a major strength: you could deal with information too big to fit in memory. We have lost that benefit here, but if your company is large enough that loading the information stored above is taking too much memory on the server, then your company should be able to afford a dedicated server with maxed-out memory.

Second, the scoring, which we have handled by an inner function, ranks first by number of query terms that are matched (the `-cmp` is to ensure highest scores first, not lowest), and then as a tiebreaker sorts by alphabetical order. This is a basic scoring system that should be adequate, but it is one of many parts intended to allow the reader to tinker and see if it can be improved. The goal is to put the most interesting results here, and the specific criterion is intended as a sort of "adequate placeholder" that can be used as is but also invites tinkering and customization to your company's needs.

```
try:
    start = int(request.POST[u'start'])
except:
    start = 0
try:
    results_per_page = int(request.POST[u'results_per_page'])
except:
    results_per_page = RESULTS_PER_PAGE
returned_results = results[start:start + results_per_page]
json_serializer = serializers.get_serialized(u'json')()
response = HttpResponse()
response[u'Content-type'] = u'text/json'
json_serializer.serialize([returned_results, len(results)],
                    ensure_ascii = False, stream = response)
return response
```

In the last few lines, we look at some CGI variables, create an object with one page's worth of results plus the total count of results found, and put it in a JSON response. Once parsed on the client end, the client will have one page's worth of results, including the names of the entities, their descriptions, and their Django-assigned ID numbers, which can be used as a basis to link them to this person's page. Here again there is room to tinker and expand. For instance, since we are working on a photo directory, we might search for a thumbnail creator (`sorl-thumbnail` turns up quickly as a Django image thumbnailer) and work on a way to deliver a client-side page that includes thumbnail images.

Let us look at another point, beginning with an earlier remark about Django philosophy as contrasted to Ruby on Rails—Rails is an unapologetically opinionated framework: "You're using my framework, you do it my way." Django is intended to offer certain facilities, and then stay out of your way if you choose to do things differently. Here we have used Django persistence facilities, but not played to their usual strengths. We realized a problem existed while thinking somewhat outside the box, and chose a basic implementation that Django's documentation doesn't really suggest. This does not showcase the natural strengths of Django persistence facilities, but it does showcase how Django avoids demanding, "It's my way or the highway." With that stated, let's look a little more at Django persistence facilities and how they would be used in a solution that plays better to Django's natural strengths.

A tour of Django persistence facilities

It has been said that **Object-relational Management** (**ORM**) is the Vietnam War of computer science. People keep throwing money, resources, and work at the problem, and it remains unsolved. If the problem is conceived in a general way, namely to take whatever objects Python programmers may create, and straightforwardly have an automatic system that will retroactively figure out how to serialize/deserialize (pickle/unpickle) them in a relational database, that problem, in general, is a hard problem and one that may never have a good solution. However, Django handles ORM in such a way that it is not only solvable, but solved neatly.

Django models and related classes are designed from the ground up to be mappable to databases. And that is a different matter from designing out of object-oriented concerns without consideration for relational databases, and then trying to retrofit storage in relational databases. A model class corresponds to a table, an instance of a model corresponds to an individual row, and fields of the model correspond to columns in the table/row. While Django is meant to let you stay in "Python mode" and avoid frequent Python/SQL context switching, it asks you to correctly solve problems like foreign keys and many-to-one relationships (as we have used) or many-to-many relationships. This means that the programmer is asked to solve the problem so that the ORM is straightforward, and keeps Django's ORM code from the tarpit of correctly implementing "Do what I mean!" functionality of mapping any object(s) to a relational database correctly, and do what the programmer would have meant to do. Django's approach to ORM is to provide carefully designed, Pythonic facilities and asks us as developers, "Say what you mean!"

For search and lookups, you can, on the model class, call something like:

```
Entity.objects.get(name__icontains = u'ed')
```

For `name__icontains`, on the left of the double underscore separator is a field name, and to the right is an operator, such as:

- `contains`: the string you supply is contained in the field value, case-sensitive. One compatibility note: `description__contains = u'book'` is translated into SQL of `...description LIKE '%book%'...`. On some database backends, `LIKE` preserves case sensitivity; on other backends, the interpretation of this SQL command forces case insensitivity. Django seems not, in general, to smooth over differences between database backends to provide uniform behavior; this means that switching the database backend could cause a change in how identical Django searches behave. In one case, `description__contains = u'book'` will match a description of `u'Books for sale!'`, and in another case it won't.
- `day`: for date/datetime fields, exact one-based day match.
- `endswith`: ends with, case-sensitive.
- `exact`: a case-sensitive, exact match.
- `gt`: greater than.
- `gte`: greater than or equal to.
- `icontains`: the string you supply is contained in the field value, case-insensitive.
- `iexact`: a case-insensitive exact string match.
- `in`: the field value is in a list you provide.
- `iendswith`: ends with, case-insensitive.
- `iregex`: case-insensitive regular expression search.
- `isnull`: if this is equal to true, specifies that a field is null; if it is false, specifies that the field is not.
- `istartswith`: starts with, case-insensitive.
- `lt`: less than.
- `lte`: less than or equal to.
- `month`: for date/datetime fields, exact one-based month match.
- `regex`: case-sensitive regular expression search.
- `startswith`: starts with, case-sensitive.
- `week_day`: Day of week, from 1 (Sunday) to 7 (Saturday).
- `year`: for date/datetime fields, exact four-digit year match.

There is some similarity between the combinations of field names and the list above, and jQuery selectors like `$(".emphasized")` that served as a "virtual higher-level language" from JavaScript. There are several parallels, and we could say in similar fashion that just as jQuery offers a "virtual higher-level language" on top of bare JavaScript, Django offers a "virtual higher-level language" over a mixture of bare Python and bare SQL. And the similarity goes further. Just as jQuery is intended to allow chaining, Django allows chained operations. Let's suppose that we want to search for entities with descriptions containing the string "book", take out all entries containing the string "bookworm", be case insensitive in both instances, and sort them in alphabetical order by name and then description. In Django this looks remarkably similar to jQuery chaining:

```
queryset = Entity.objects.filter(description__icontains = u'book').
exclude(description__icontains = u'bookworm').order_by(u'name',
u'description')
```

We've made a few different adjustments as we've refined our requirements in the code above. Now how many database hits has this cost us? Trick question: the answer is, "None." Django's database handling is lazy, and doesn't fetch results until it knows what results are really needed, when they are needed. If we make an initial request and then make adjustments, all it does is keep its notes up to date. If and when we want results, we can get them, with database hits as needed, by something like the following:

```
for entity in queryset.all():
    # Do something with each entity
```

And in classic relational database tradition, this means that you can have a much bigger dataset without increasing the memory taken. If you want to, you can create a computer with a more than a terabyte hard drive and only 64 megabytes of RAM, and do database searches through genomic data without running out of memory.

This example was for illustration purposes. If you are a researcher working with genomic data, don't reinvent the internal combustion engine. Use BioPerl. It's a far more Pythonic solution than building in Python from scratch.

The objects we have been dealing with are queries and QuerySet and further Django documentation is online.

Summary

We have taken an overview of Django's persistence facilities, its way of building models so that ORM is not a tarpit but is solved cleanly, and we have looked at how a solution might work using these building blocks. Specific points covered include why the server is a natural home to address persistence issues though a database or otherwise, as well as what a Django model that demonstrates several key features could look like, to make a view that performs a basic search on this model and returns JSON, which could be useful to client-side Ajax parts of an application, finally, what some of the other Django search functionality is, and how it resembles one of jQuery's selling points.

Let us continue. Our next chapter, *Signing Up and Logging into a Website Using Ajax* covers both front end and back end tools to have a Django Ajax login. This will get our hands dirty, and we will be working to build a graceful Web 2.0-style login process.