Web Services and Web Data

XJCO3011

Session 7. More Django: Accessing and Filtering Data

SOUTHWEST JIAOTONG UNIVERSITY • UNIVERSITY OF LEEDS
西南交通大学一利兹学院
SWJTU-LEEDS JOINT SCHOOL

In this lecture

- All we need now is to implement the business logic for each of the view functions.
  - As we explained, a view is responsible for doing some arbitrary logic, and then returning a response.
  - In modern web applications, the arbitrary logic often involves interacting with a database. Behind the scenes, a database-driven website connects to a database server, retrieves some data out of it, and displays that data on a web page. The site might also provide ways for site visitors to populate the database on their own.
  - Many complex websites provide some combination of the two. www.amazon.com, for instance, is a great example of a database-driven site. Each product page is essentially a query into Amazon's product database formatted as HTML, and when you post a customer review, it gets inserted into the database of reviews.
  - Django is well suited for making database-driven websites because it comes with easy yet powerful tools for performing database queries using Python.

- For this, we need to know how to retrieve and manipulate data from that database within the Django models, which we will discuss in this lecture.

Django Girls Tutorial

https://tutorial.djangogirls.org/en/

It works for boys as well!

MVC

- Best web frameworks are built around the MVC concept. The MVC design pattern is really simple to understand:

- The **Model(M)** is a model or representation of your data. It's not the actual data, but an interface to the data. The model allows you to pull data from your database without knowing the intricacies of the underlying database. The model usually also provides an *abstraction* layer with your database, so that you can use the same model with multiple databases.

- The **View(V)** is what you see. It's the presentation layer for your model. On your computer, the view is what you see in the browser for a web app, or the UI for a desktop app. The view also provides an interface to collect user input.

- The **Controller(C)** controls the flow of information between the model and the view. It uses programmed logic to decide what information is pulled from the database via the model and what information is passed to the view. It also gets information from the user via the view and implements business logic: either by changing the view, or modifying data through the model, or both.

MTV

- Django follows the MVC pattern closely, however, it does use its own logic in the implementation. Because the C is handled by the framework itself and most of the work in Django happens in models, templates and views, Django is often referred to as an *MTV framework*. In the MTV development pattern:

- **M stands for "Model,"** the data access layer. This layer contains anything and everything about the data: how to access it, how to validate it, which behaviors it has, and the relationships between the data.

- **T stands for "Template,"** the presentation layer. This layer contains presentation related decisions: how something should be displayed on web page or other type of document.

- **V stands for "View,"** the business logic layer. This layer contains the logic that accesses the model and defers to the appropriate template(s). You can think of it as the bridge between models and templates.

Difference between MVC and MTV

Django's View is more like the Controller in MVC,

and

MVC's View is actually a Template in Django.

Django Models
A basic book/author/publisher data layout

- The conceptual relationships between books, authors, and publishers are well known; a book that was written by authors and produced by a publisher!

- Some assumptions:
  - An author has a first name, a last name, and an email address.
  - A publisher has a name, a street address, a city, a state/province, a country, and a website.
  - A book has a title and a publication date. It also has one or more authors (a many-to-many relationship with authors) and a single publisher (a one-to-many relationship–aka foreign key–to publishers).

Example

Step 6 of Lab 2 in models.py file

```
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

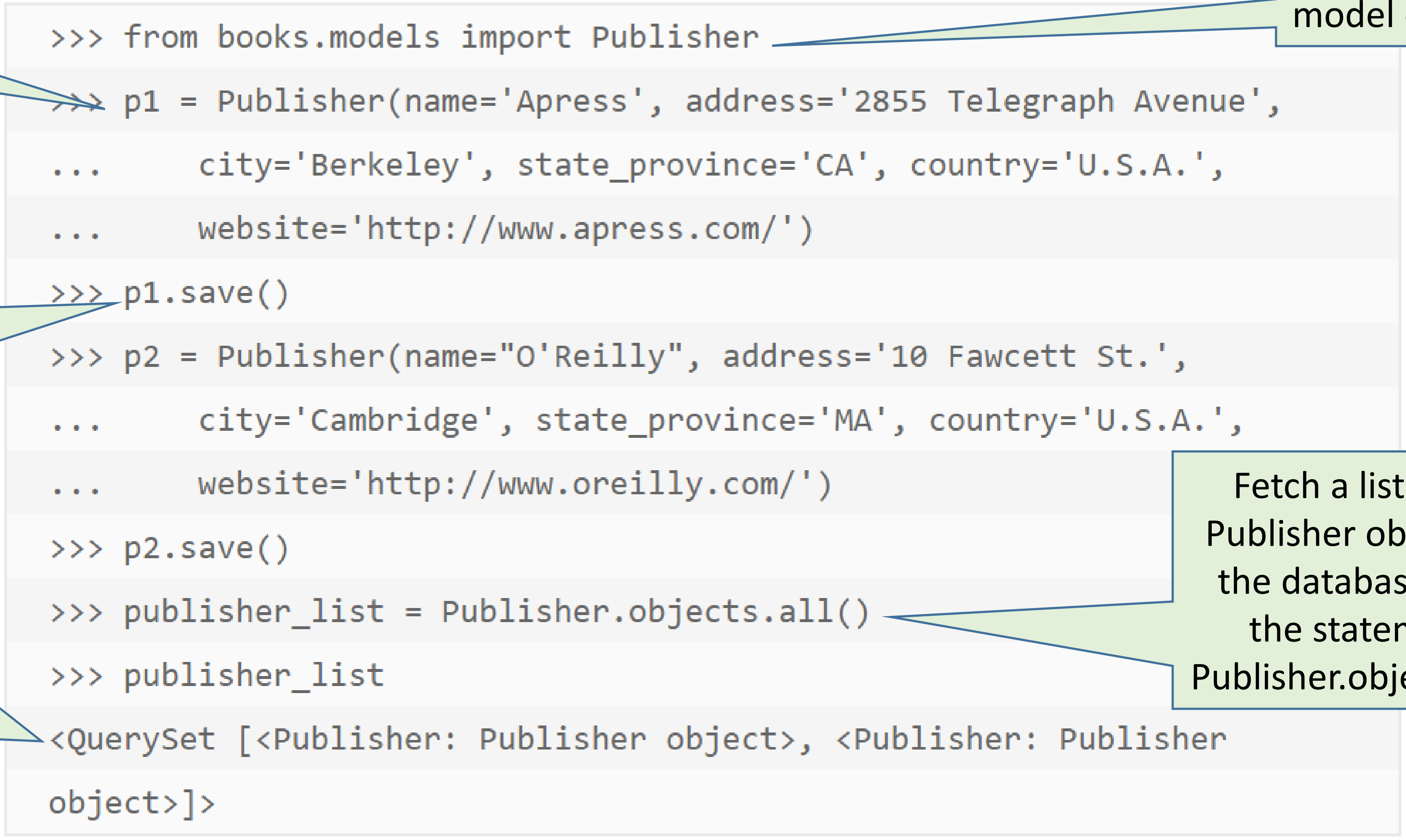
Creating and retrieving records

Once you've created a model, Django automatically provides a high-level Python API for working with those models.

import our Publisher model class

Creates a new Publisher object

Saves the object to the underlying database table database



```
>>> from books.models import Publisher
>>> p1 = Publisher(name='Apress', address='2855 Telegraph Avenue',
...     city='Berkeley', state_province='CA', country='U.S.A.',
...     website='http://www.apress.com/')
>>> p1.save()
>>> p2 = Publisher(name="O'Reilly", address='10 Fawcett St.',
...     city='Cambridge', state_province='MA', country='U.S.A.',
...     website='http://www.oreilly.com/')
>>> p2.save()
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
<QuerySet [<Publisher: Publisher object>, <Publisher: Publisher
object>]>
```

Fetch a list of all Publisher objects in the database with the statement Publisher.objects.all()

The QuerySet is a list-like object that contains model objects

The objects.create() method

If you want to create an object and save it to the database in a single step, use the objects.create() method

```
>>> p1 = Publisher.objects.create(name='Apress',
...     address='2855 Telegraph Avenue',
...     city='Berkeley', state_province='CA', country='U.S.A.',
...     website='http://www.apress.com/')
>>> p2 = Publisher.objects.create(name="O'Reilly",
...     address='10 Fawcett St.', city='Cambridge',
...     state_province='MA', country='U.S.A.',
...     website='http://www.oreilly.com/')
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
<QuerySet [<Publisher: Publisher object>, <Publisher: Publisher
object>]>
```

Adding Model String Representations

- A __str__() method tells Python how to display a human-readable representation of an object.
- Always define this method for each of your models

```
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()

    def __str__(self):
        return u'%s %s' % (self.first_name, self.last_name)
```

So now, we can get a more meaningful QuerySet (or a desired representation of an object)

```
>>> from books.models import Publisher
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
<QuerySet [<Publisher: Apress>, <Publisher: O'Reilly>]>
```

Filtering Data

Naturally, it's rare to want to select everything from a database at once; in most cases, you'll want to deal with a subset of your data. You can filter your data using the filter() method:

```
>>> Publisher.objects.filter(name='Apress')
<QuerySet [<Publisher: Apress>]>
```

- Retrieve all the Publisher objects from the database where the name attribute is equal to 'Apress'
- The filter() method has successfully retrieved a subset of Publisher objects from the database, and returned them as a list-like object containing the matching Publisher objects

You can pass multiple arguments into filter() to narrow down things further:

```
>>> Publisher.objects.filter(country="U.S.A.",
state_province="CA")
<QuerySet [<Publisher: Apress>]>
```

Approximate matching of field content can be done by appending __contains to the field's name, i.e. a double underscore then the word contains.

```
>>> Publisher.objects.filter(name__contains="press")
<QuerySet [<Publisher: Apress>]>
```

Retrieving Single Objects

- The filter() examples above all returned a QuerySet, which you can treat like a list.
- To fetch only a single object, instead of a QuerySet. Use the get() method :

```
>>> Publisher.objects.get(name="Apress")
<Publisher: Apress>
```

- only a single object is returned

- Because of that, a query resulting in multiple objects will cause an exception:

```
>>> Publisher.objects.get(country="U.S.A.")
Traceback (most recent call last):
    ...
books.models.MultipleObjectsReturned: get() returned more than one
Publisher - it returned 2!
```

- A query that returns no objects also causes a models.DoesNotExist exception, try to catch it:

```
>>> Publisher.objects.get(name="Penguin")
Traceback (most recent call last):
    ...
books.models.DoesNotExist: Publisher matching query does not exist.
```

Deleting Objects

- To delete an object from your database, simply call the object's delete() method:

```
>>> p = Publisher.objects.get(name="O'Reilly")
>>> p.delete()
(1, {'books.Publisher': 1})
>>> Publisher.objects.all()
<QuerySet [<Publisher: Apress>, <Publisher: GNW Independent
Publishing>]>
```

- You can also delete objects in bulk by calling delete() on the result of any QuerySet.

```
>>> Publisher.objects.filter(country='USA').delete()
(1, {'books.Publisher': 1})
>>> Publisher.objects.all().delete()
(1, {'books.Publisher': 1})
>>> Publisher.objects.all()
<QuerySet []>
```

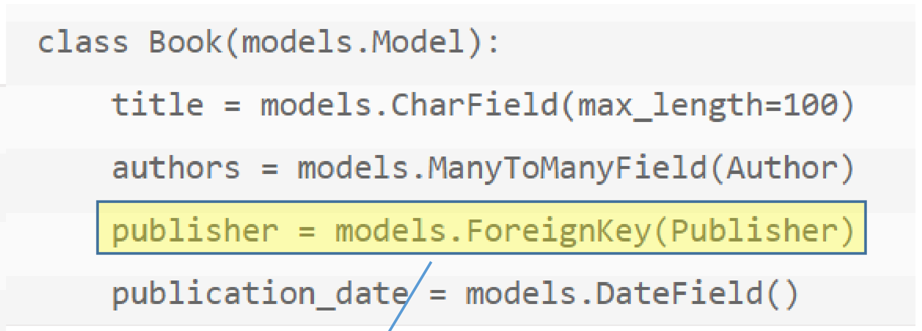
- deletes all Publisher objects where the country is 'USA'

- deletes all Publisher objects

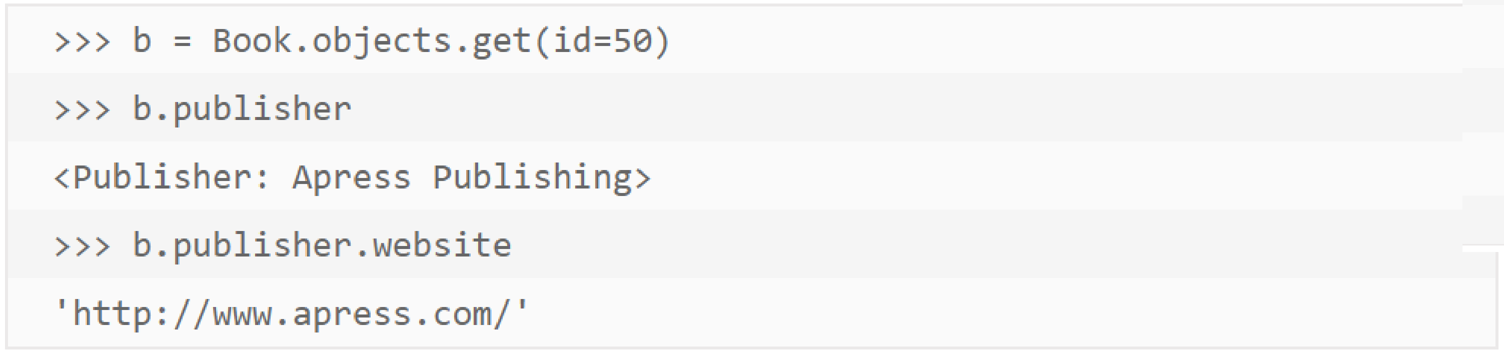
- queries all Publisher objects

Accessing Foreign Key Values (1)

- When you access a field that’s a ForeignKey, you’ll get the related model object.



```
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```



```
>>> b = Book.objects.get(id=50)
>>> b.publisher
<Publisher: Apress Publishing>
>>> b.publisher.website
'http://www.apress.com/'
```

- queries the Book object with an id of 50 from the database and assigns it to the variable b.
- accesses the Publisher object associated with the variable b. This is achieved through the foreign key field in the Book model, which establishes a one-to-many relationship between the Book and Publisher models, ,where each Book instance is associated with one Publisher instance.）
- accesses the website attribute of the Publisher object associated with the variable b. This is a string representing the website address of the publishing house.

Chapter 9 from Django Book

Accessing Foreign Key Values (2)

- With ForeignKey fields, it works the other way, too, but it’s slightly different due to the non-symmetrical nature of the relationship. To get a list of books for a given publisher, use publisher.book_set.all(), like this:

```
>>> p = Publisher.objects.get(name='Apress Publishing')
>>> p.book_set.all()
[<Book: The Django Book>, <Book: Dive Into Python>, ...]
```

```
class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()
```

- queries the Publisher object with a name of "Apress Publishing" from the database and assigns it to the variable p.
- accesses the set of Book objects associated with the variable p. This is achieved through the foreign key relationship defined in the Book model, where each Book instance is associated with one Publisher instance. Django automatically adds a property named book_set to each Publisher
- The attribute name book_set is generated by appending the lower case model name to _set.
- book_set is just a QuerySet, and it can be filtered and sliced like any other QuerySet. For example:

```
>>> p = Publisher.objects.get(name='Apress Publishing')
>>> p.book_set.filter(title__icontains='django')
[<Book: The Django Book>, <Book: Pro Django>]
```

Chapter 9 from Django Book

- query the Publisher object with a name of "Apress Publishing" from the database, and retrieve a filtered list of associated Book objects based on the substring "django" appearing in their title attribute in a case-insensitive manner

Accessing Many-to-Many Values

Many-to-many values work like foreign-key values, except we deal with QuerySet values instead of model instances. For example, here’s how to view the authors for a book:

```
>>> b = Book.objects.get(id=50)
>>> b.authors.all()
[<Author: Adrian Holovaty>, <Author: Jacob Kaplan-Moss>]
>>> b.authors.filter(first_name='Adrian')
[<Author: Adrian Holovaty>]
>>> b.authors.filter(first_name='Adam')
[]
```

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

It works in reverse, too. To view all of the books for an author, use author.book_set, like this:

```
>>> a = Author.objects.get(first_name='Adrian',
last_name='Holovaty')
>>> a.book_set.all()
[<Book: The Django Book>, <Book: Adrian's Other Book>]
```

```
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()
```

Model methods

- Model methods define custom methods on a model to add custom row-level functionality to your objects.
- Model methods act on a particular model instance.

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    birth_date = models.DateField()

    def baby_boomer_status(self):
        # Returns the person's baby-boomer status.
        import datetime
        if self.birth_date < datetime.date(1945, 8, 1):
            return "Pre-boomer"
        elif self.birth_date < datetime.date(1965, 1, 1):
            return "Baby boomer"
        else:
            return "Post-boomer"

    def _get_full_name(self):
        # Returns the person's full name."
        return '%s %s' % (self.first_name, self.last_name)
    full_name = property(_get_full_name)
```

- a class called "Person" that inherits from Django's "models.Model" class.
- three attributes: "first_name", "last_name", and "birth_date", which are defined as character fields and a date field

A baby boomer is a person born in the years following the Second World War, when there was a temporary marked increase in the birth rate.

- an attribute named full_name, which uses the Python built-in property function to convert the _get_full_name method into a property.
- This allows accessing the person's full name by accessing the full_name attribute of a Person object.

Overriding Predefined Model Methods

- There's another set of model methods that encapsulate a bunch of database behaviour that you'll want to customize. In particular, you'll often want to change the way save() and delete() work.
- A classic use-case for overriding the built-in methods is if you want something to happen whenever you save an object. For example, to update a calculated field (such as the arrival data for a flight)

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, *args, **kwargs):
        do_something()
        super(Blog, self).save(*args, **kwargs) # Call the "real"
save() method.
        do_something_else()
```

- It calls a custom "do_something()" function before saving the instance.
- It calls the "super()" method to invoke the original "save" method from the parent class, passing any arguments and keyword arguments received by this method.
- It calls another custom "do_something_else()" function after saving the instance.

The QuerySet

- QuerySets are list-like objects
- A QuerySet is iterable, e.g.:

for e in Entry.objects.all(): # retrieve all objects of the Entry model from the database, and then iterates over each object and prints its headline attribute.
print(e.headline)

- It can be sliced like any Python list.
- It has the values() method that returns dictionaries, rather than model instances, e.g.:

```
# This list contains a Blog object.
>>> Blog.objects.filter(name__startswith='Beatles')
<QuerySet [<Blog: Beatles Blog>]>

# This list contains a dictionary.
>>> Blog.objects.filter(name__startswith='Beatles').values()
<QuerySet [{'id': 1, 'name': 'Beatles Blog', 'tagline': 'All the latest Beatles
news.'}]>
```

Example

Implementing the view function for the directory List service

List

Service Aim: to get a list of all agencies in the directory

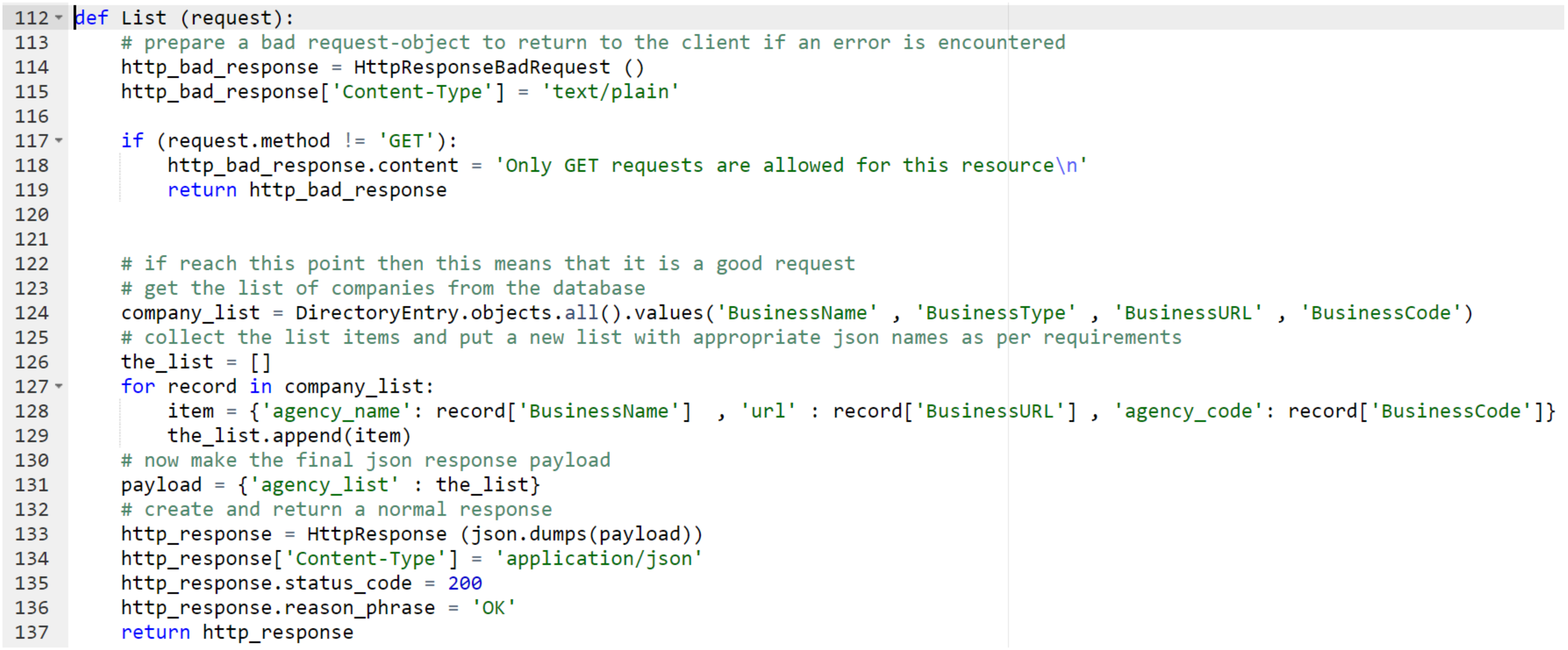
Service Details:

The client sends a **GET** request to **/api/list/** with an empty payload.

If the request is processed successfully, the server responds with **200 OK** and the list of agencies ("agency_list", array) in a JSON payload. For each agency in the list, the following data must be given:

- The name of the agency (“agency_name”, string)
- The URL of the agency’s website (“url” , string)
- The agency’s unique code (“agency_code”, string)

If the server is unable to process the request for any reason, the server should respond with the appropriate error code.



```python
def List (request):
    # prepare a bad request-object to return to the client if an error is encountered
    http_bad_response = HttpResponseBadRequest ()
    http_bad_response['Content-Type'] = 'text/plain'

    if (request.method != 'GET'):
        http_bad_response.content = 'Only GET requests are allowed for this resource\n'
        return http_bad_response


    # if reach this point then this means that it is a good request
    # get the list of companies from the database
    company_list = DirectoryEntry.objects.all().values('BusinessName' , 'BusinessType' , 'BusinessURL' , 'BusinessCode')
    # collect the list items and put a new list with appropriate json names as per requirements
    the_list = []
    for record in company_list:
        item = {'agency_name': record['BusinessName']  , 'url' : record['BusinessURL'] , 'agency_code': record['BusinessCode']}
        the_list.append(item)
    # now make the final json response payload
    payload = {'agency_list' : the_list}
    # create and return a normal response
    http_response = HttpResponse (json.dumps(payload))
    http_response['Content-Type'] = 'application/json'
    http_response.status_code = 200
    http_response.reason_phrase = 'OK'
    return http_response
```

- **views.py** is where we handle the request/response logic for our web app

That's it Folks

Further Reading

The Django Book