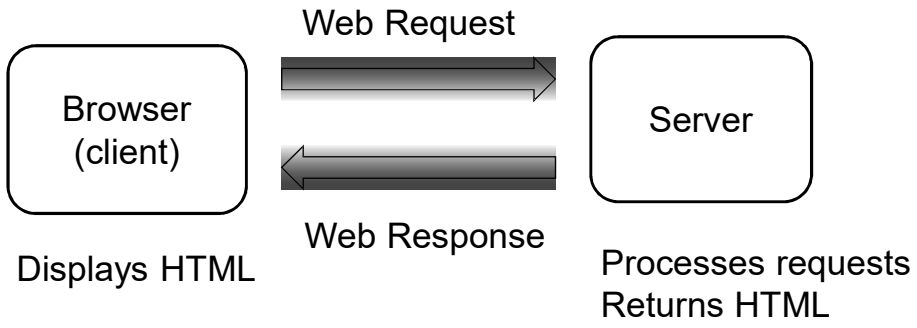
Writing View Functions

Maxim Mozgovoy

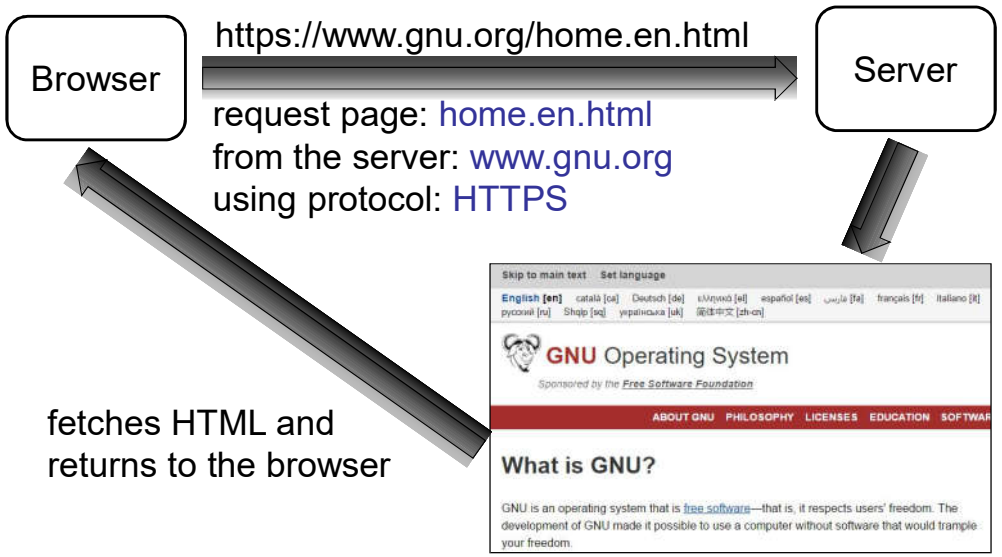
Web App Architecture Reviewed



Browsers and servers can communicate since they understand the same protocol (HTTP and HTTPS)

Web App Architecture Reviewed

In the simplest case, a server just returns a static HTML page requested by a browser:



Views (View Functions) in Django

- In Django, a view is a function that processes a web request and returns a web response (HTML) to the browser.
- One view normally corresponds to one type of browser request:
  - Display the index page.
  - Register a new user.
  - Return the result of the database query…
- Technically, views are Python functions declared in the *App-name*`/views.py` file.
- Different views are triggered by different URLs, so we also must specify the correspondence between URLs and views.

Creating Simple Views

Each view should return a `HttpResponse` object. In the simplest case, it can be a plain text/HTML string.

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Welcome to the site!")

def hello(request):
    return HttpResponse("Hello there!")
```

Note: usually Django apps use plain HTTP protocol to communicate. HTTPS/HTTP redirection is performed with a Web server or a reverse proxy system, such as Nginx.

Mapping URLs to Views

URL mapper rules are defined in *`Project-Name`*`/urls.py`. Each rule maps a URL to a certain view function. It analyzes `urlpatterns` elements one by one until a match is found.

View *names* are useful for *redirects*.

```
from FirstApp import views # import our views.py
urlpatterns = [
  path('admin/', admin.site.urls), # by default
  path('hello', views.hello),
  path('', views.index, name='view-index')]
```

Now try:
http://localhost:8000/hello
http://localhost:8000/admin
http://localhost:8000

Redirects

Sometimes a view function should do some actions and open another URL. This is done with redirects:

```
from django.shortcuts import redirect

def my_view(request):
    # do something
    # and pass control to the view
    # with the specified name
    return redirect('view-index')
```


Working with the Database

Now let's consider how to program some basic database operations. We will again deal with a simple two-table database:

```
class Manufacturer(models.Model):
   name = models.CharField(max_length=20)

class Car(models.Model):
   number_plate = models.CharField(max_length=10)
   owner = models.CharField(max_length=20)
   manufacturer = models.ForeignKey(Manufacturer,
                       on_delete=models.CASCADE)
```

Creating Entities

Creating a new manufacturer:

```
from FirstApp.models import Manufacturer, Car
m = Manufacturer(name='Fiat')
m.save() # implicitly calls SQL INSERT statement
m.name = 'Honda'
m.save() # implicitly calls SQL UPDATE statement
```

Creating a new car:

```
c = Car(number_plate='12345',
        owner='John Doe',
        manufacturer=m)    # foreign key
c.save()
```


Updating Many to Many Fields

```
class Group(models.Model): # Students, Golfers
   name = models.CharField(max_length=20)

class Person(models.Model): # Mike, John, Mary
   name = models.CharField(max_length=20)
   groups = models.ManyToManyField(Group)

# create() is the same as "construct & save"
g = Group.objects.create(name="Students")
p = Person(name="John")
p.groups.add(g)
p.save()
```


Retrieving Objects

Use *ClassName*`.objects` to retrieve table elements.

use `all()` to return all objects:

```
# (SQL SELECT * FROM car-table)
cars = Car.objects.all()
```

Use `filter()` to set one or more conditions:

```
# return John's cars;
# return John's cars with the given plate number
# (SQL SELECT * FROM car-table
#               WHERE cond1 AND cond2 …)
c = Car.objects.filter(owner='John')
c = Car.objects.filter(owner='John',
            number_plate='12345')
```


Retrieving Objects

Use `except()` to return the objects NOT matching the condition:

```
# return everyone's cars except John's;
# (SQL SELECT * FROM car-table
#               WHERE NOT(cond1 AND cond2…))
c = Car.objects.exclude(owner='John')
```

Chain conditions to perform complex queries:

```
c = Car.objects.filter(…).filter(…).exclude(…)
```

Each result of database query is a `QuerySet` object (a list of objects). Use the index operator to get individual objects:

```
car = c[0] # get the first car from the list
n_cars = len(c) # total number of objects
```

Retrieving Objects

If only one object is expected, use `get()`:

```
c = Car.objects.filter(…).get()
```

Reuse existing `QuerySet` objects for convenience:

```
q1 = Car.objects.all()
q2 = q1.filter(owner='John')
```

Use `print()` on `QuerySet` objects for debugging:

```
print(q2)
```

Use `filter()` with class objects for complex values such as `DateField`:

```
q1 = Person.objects.filter(date_of_birth=
            datetime(1995, 1, 30))
```


Field Lookups

In the previous examples, queries returned *exact* matches:

```
q = Car.objects.filter(owner='John')
```

It is possible to have more flexibility with *field lookups* (use *field-name*`__operator` syntax):

```
# field "owner" contains a substring 'Jo':
Car.objects.filter(owner__contains='Jo')

owner__startswith='Jo' # owner starts with 'Jo'
owner__endswith='ohn'  # owner ends with 'ohn'
```


Field Lookups

```
age__gt=5  # age > 5
age__gte=5 # age >= 5
age__lt=5  # age < 5
age__lte=5 # age <= 5
age__range=(5,10) # 5 <= age <= 10
                  # works with dates, too!
age__in=(5,10,15) # age = 5, 10, or 15
```

There are lookups for dates/times, regex search, isNull checks…

More QuerySet Operations

```
# get objects from 5 to 10 (SQL LIMIT clause)
q = Car.objects.all()[5:10]

# sorting results (SQL ORDER BY clause)
# simple order by a certain field
q = Car.objects.order_by('owner')

order_by('-owner') # sorting in the reverse order
order_by('?')      # random sorting

# soring cars of the same owner by number plate
order_by('owner', 'number_plate')
```

Referencing Tables in Queries

To search a table referenced as a foreign key, use *table-name__field-name* syntax:

```
# search cars by manufacturer
q = Car.objects.filter(
        manufacturer__name='Toyota')

# all normal operators apply to such fields:
q = Car.objects.filter(
        manufacturer__name__contains='oyo')
```


Other Operations

To delete a record or a `QuerySet`, use `delete()`:

```
car = Car.objects.all()[0]
car.delete()
Car.objects.all().delete() # delete everything
```

To update records, use `update()`:

```
# sell all John's cars to Paul
Car.objects.filter(owner='John').update(
                                owner='Paul')
```

Note: for individual objects we call `save()` to save changes, for the sets of objects (`QuerySet` items) we call `update()`.

Transactions in Django

Sometimes you need to do several operations and rollback the whole sequence if one of the them fails. To do it, use `atomic()` clause, and throw an exception if something goes wrong:

```
from django.db import transaction
...
try:
    with transaction.atomic():
        # do something
        # do something
        # in case of error, throw an exception
except my_exception:
    # handle error
```


Object Locking in Django

Sometimes you need to lock some objects while doing operations (to prevent simultaneous access). This can be done by retrieving a QuerySet with `select_for_update()` call inside an atomic transaction block:

```
from django.db import transaction
...
with transaction.atomic():
    objs = Cars.objects.select_for_update().all()
    # do something
    # Cars will be locked till the end of
    # the atomic block
```

Note: there are many ways to achieve this behavior; this solution works not on all database engines.

Shell Experiments

The easiest way to experiment with data is to run Python shell:

```
python manage.py shell  # inside FirstPrj
```

Then you can try simple commands:

```
from FirstApp.models import Car, Manufacturer
m = Manufacturer.objects.all()
m # <QuerySet [<Manufacturer: Toyota>,
    <Manufacturer: Opel>, <Manufacturer: Ford>]>
len(m)     # 3
m[1]       # <Manufacturer: Opel>
```


Coming Exercise

During the next exercise session you have to start implementing the basic logic for your application.

Make sure to create a separate view for each typical operation (register a new user, display some list of items, display results matching a user query…)

Each time you need a user-supplied value (user name, query), just hard-code it in some variable inside a view.