SWPP Practice Session #6

Django

2022 Oct 12

Announcement

- Sprint #1 Report
  - due: 10/15(Sat) 6pm
  - send Requirements and Specifications in PDF format to swpp.22.staff@spl.snu.ac.kr
  - upload Requirements and Specifications to team github repo wiki
- Homework 4
  - out: 10/13(Thu), due: 11/2(Wed) 6pm
  - HW will cover backend implementation and testing with Django.

Clone Repo

- **Please fork and clone this repository**
- https://github.com/swpp22fall-practice-sessions/swpp-p6-django-tutorial
- We have checkpoint branches ready. If you're in trouble and can't keep up, you can jump to the following branches with `$ git checkout {branch_name}`

Introduction to Django

Django

- High-level Python Web Framework

- Project homepage: https://www.djangoproject.com/
- Doc: https://docs.djangoproject.com/
  - Keep this under your pillow

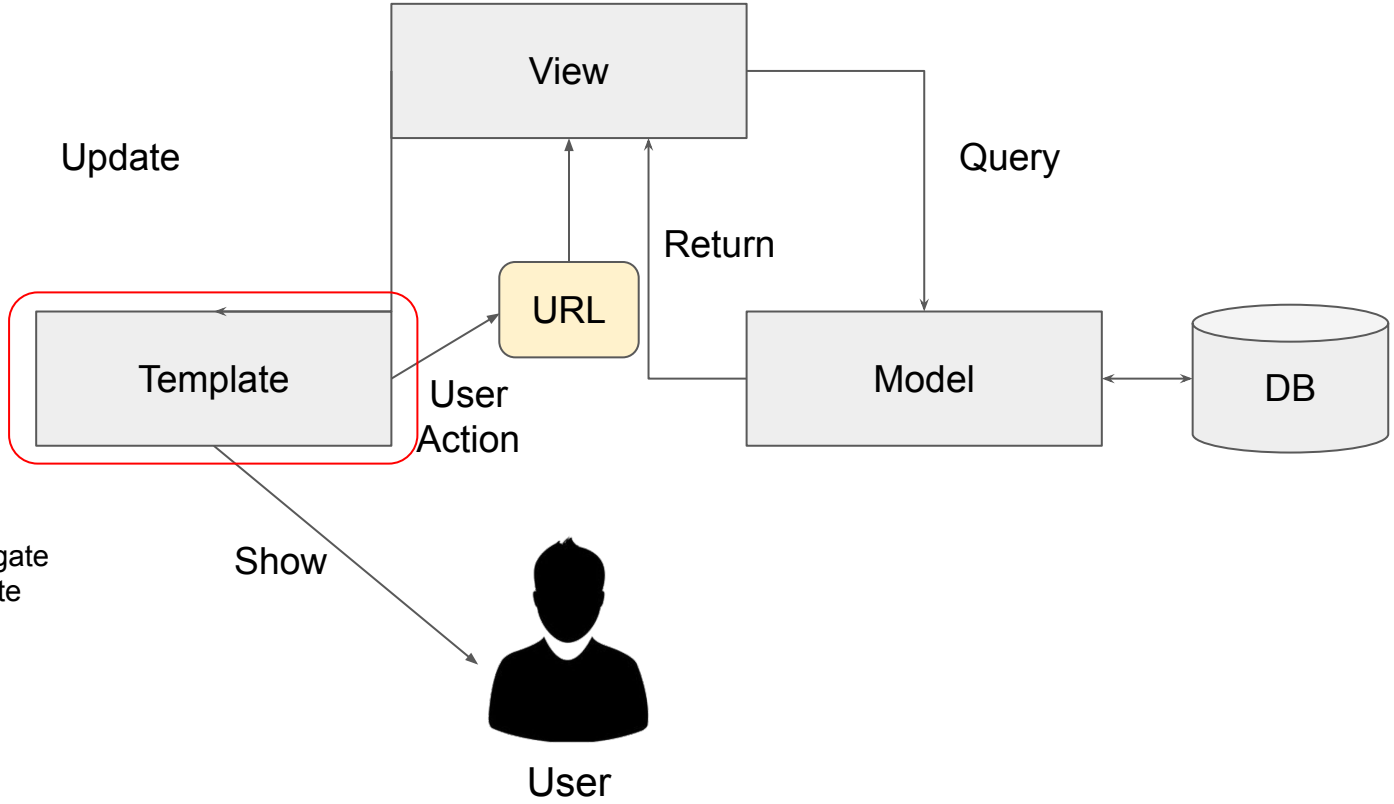
Why Django?

- Fast & Secure & Scalable
- Python based → Many useful libraries
- Admin page support
- Useful debugging tools
  - Django Debug Toolbar, Django Extension Debugger
- Easy to interact with DB via Django ORM (Object-Relational Mapping)
- Used by well-known sites (eg. Instagram, Mozilla, NASA, etc)

Django updates (FYI)

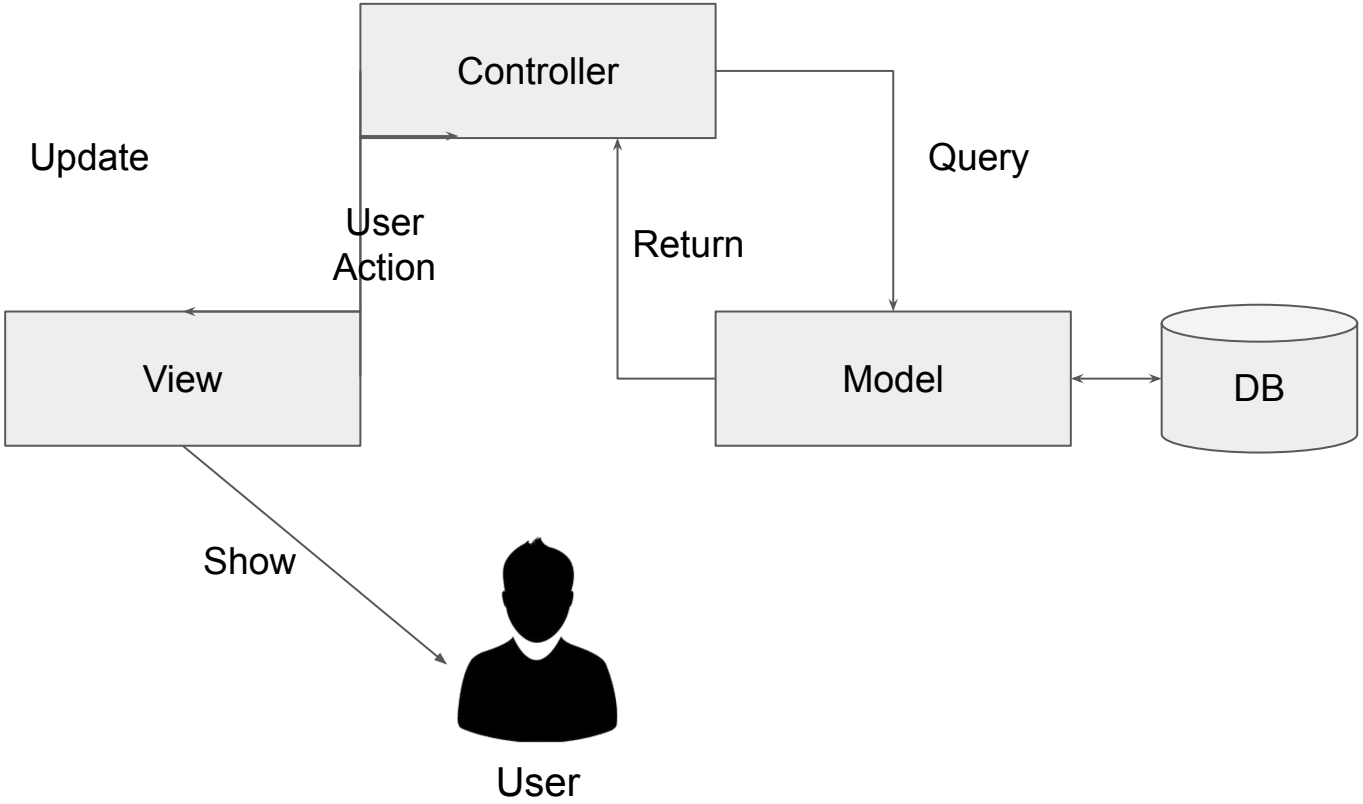
- mainstream support version for Django is now 4.1 (3.2 last year)
- main differences
  - supports python 3.8, 3.9, 3.10 (no more 3.6, 3.7)
  - zoneinfo default timezone implementation
  - Functional unique constraints
  - Redis cache backend
  - template-based form rendering
  - …
  - more info: https://docs.djangoproject.com/en/4.1/releases/4.0/

Django follows MTV pattern



In our case, React will delegate the role of Django template

MVC pattern


Controller
Update
Query
User
Action
Return
View
Model
DB
Show
User

Our Architecture

1. Client sends the HTTP request

2. Frontend sends request to Django if the request needs backend service

3. Django handles the request

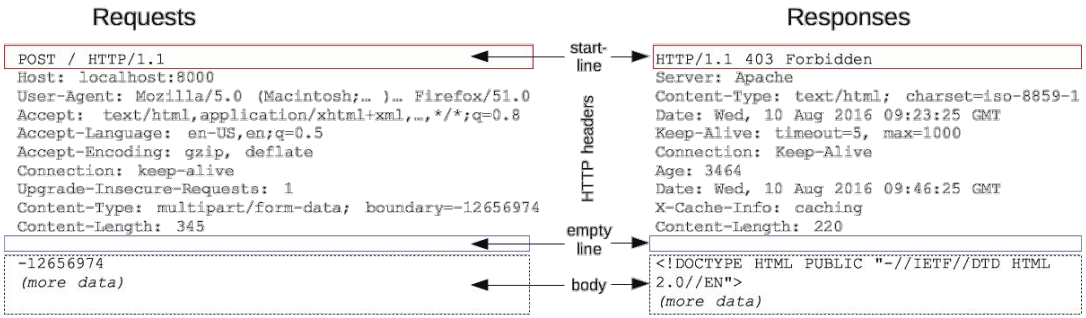
5. Data is passed to the frontend

4. Access database and build the response data

6. Frontend renders the view

HTTP

- stands for “Hypertext Transfer Protocol”
- the protocol used to transfer data over the web
- uses a client-server model
  - A client (e.g. web browser) sends request to a server.
  - The server returns response to the client.
- Most commonly-used HTTP methods are POST, GET, PUT, PATCH and DELETE.
  - POST: create operation
  - GET: read operation
  - PUT, PATCH: update operation
  - DELETE: delete operation



https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages

Let’s dive into Django

- Assuming:
  - You are in Linux environment
  - You can create and activate virtualenv with Python 3
  - You can install Django via pip (or already installed one)
    - We will use Django version 4.1.2
- Before you start:
  - Activate virtualenv
  - Install Django (`$ pip install django`)

Setup - python virtual environment

- create python virtual environment with virtualenv

```
# in the directory where you want
$ virtualenv --python=python3.9 django-env

# check if django-env dir has been created
$ ls

# activate virtual environment
$ source django-env/bin/activate

# deactivate virtual environment
$ deactivate
```

Setup - docker container

- run docker container

```
$ docker run --rm -it \
     --ipc=host \
     --name "django" \
     -p 0.0.0.0:8000:8000 \
     -v ${PWD}:/home \
     snuspl/swpp:python3.9
```

- with docker, you have to modify *settings.py*, open server with 0.0.0.0:8000 (later)

```
# settings.py
...
ALLOWED_HOSTS = ['0.0.0.0']
```

Start a Django project

- `$ django-admin startproject toh`
  - This will generate a project directory named by ‘toh’
- `$ cd toh`

- Take a look inside of the project
  - You can see `manage.py` and `toh` directory
  - `manage.py` is a script for managing your project
  - `toh` directory contains overall settings of your project

Test your project

- `$ python manage.py migrate`

- `$ python manage.py runserver`
  (`$ python manage.py runserver 0.0.0.0:8000` for docker users)
  - Open your web browser and navigate to http://127.0.0.1:8000 … or
  - `$ curl http://127.0.0.1:8000`
    - `curl` is a very versatile tool for testing HTTP. Use it!
    - `$ man curl`

Start a Django app

- `$ python manage.py startapp hero`
  - This will make app named ‘`hero`’ under your ‘`toh`’ project

- Projects vs Apps
  - Project: A collection of configuration and apps for a particular website
    - multiple apps can be contained in one project
  - App: Web application that do something
    - pluggable!
    - An app can be in multiple projects

Directory structure

- Your directory structure may look like this

```
.
├── db.sqlite3
├── hero
│   ├── admin.py
│   ├── apps.py
│   ├── __init__.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── manage.py
└── toh
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py

3 directories, 13 files
```

Let’s write our first view

- Django views are python functions or classes that receive a web request and return a web response.

- Edit `hero/views.py`

hero/views.py

```
1 from django.http import HttpResponse
2
3 def index(request):
4     return HttpResponse('Hello, world!')
```

Define routes (1/2)

- Django will receive a request in the form of URL.

some part of react code

```
axios.get(“/hero/”)
```

- To get from a URL to a view, we need to map urlpatterns to view.

```
axios.get(“/hero/”)
```

hero/views.py

```
1 from django.http import HttpResponse
2
3 def index(request):
4     return HttpResponse('Hello, world!')
```

More info at:
https://docs.djangoproject.com/en/4.1/ref/urls/#path

Define routes (2/2)

- Create `hero/urls.py`

hero/urls.py

```
1 from django.urls import path
2
3 from . import views
4
5 urlpatterns = [
6     path('', views.index, name='index'),
7 ]
```

- Edit `toh/urls.py`

toh/urls.py

```
1 from django.contrib import admin
2 from django.urls import include, path
3
4 urlpatterns = [
5     path('hero/', include('hero.urls')),
6     path('admin/', admin.site.urls),
7 ]
```

request

```
axios.get("/hero/")
```

The route argument can have…

- a url pattern

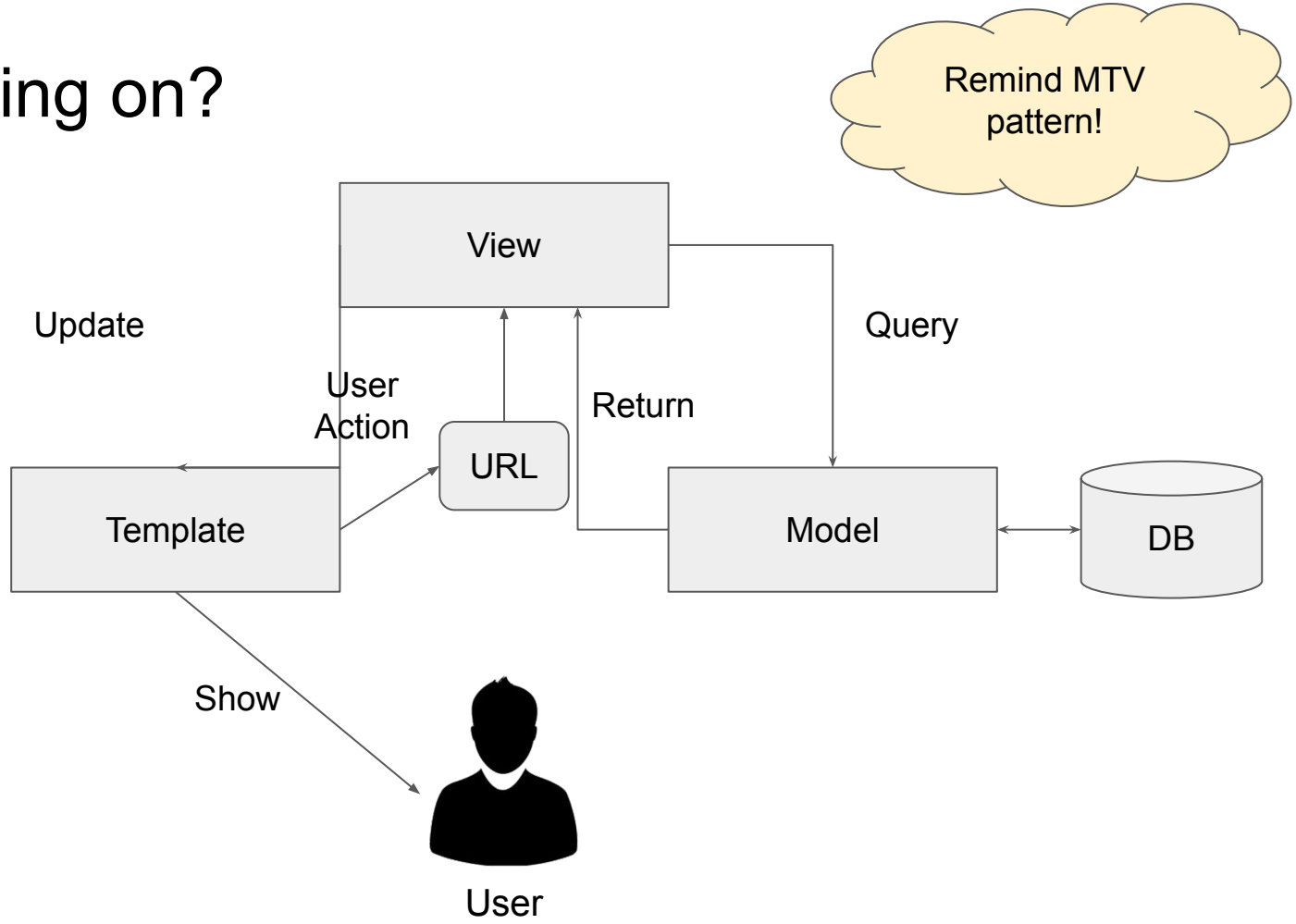
The view argument can have...

- a view function
- django.urls.include()

Test your first view

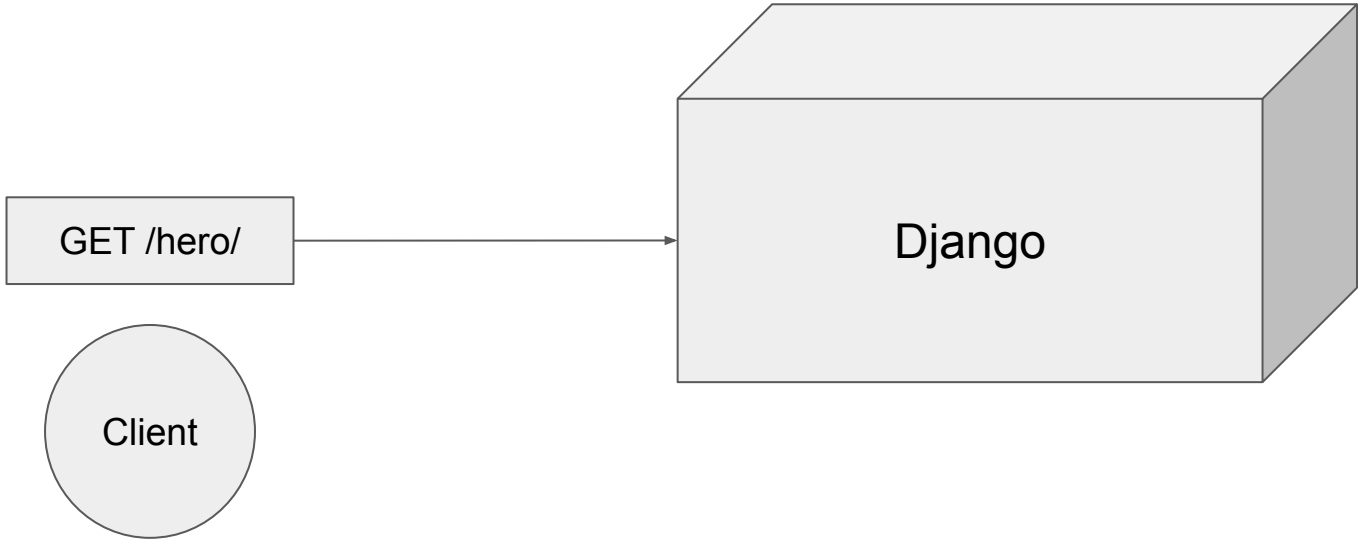
- `$ python manage.py runserver`
  - Open your web browser and navigate to http://127.0.0.1:8000/hero/ (http://0.0.0.0:8000/hero/)
  - … or `$ curl` http://127.0.0.1:8000/hero/ (`$ curl http://0.0.0.0:8000/hero/`)
    - Trailing slash is important: Do not omit it!

What’s going on?



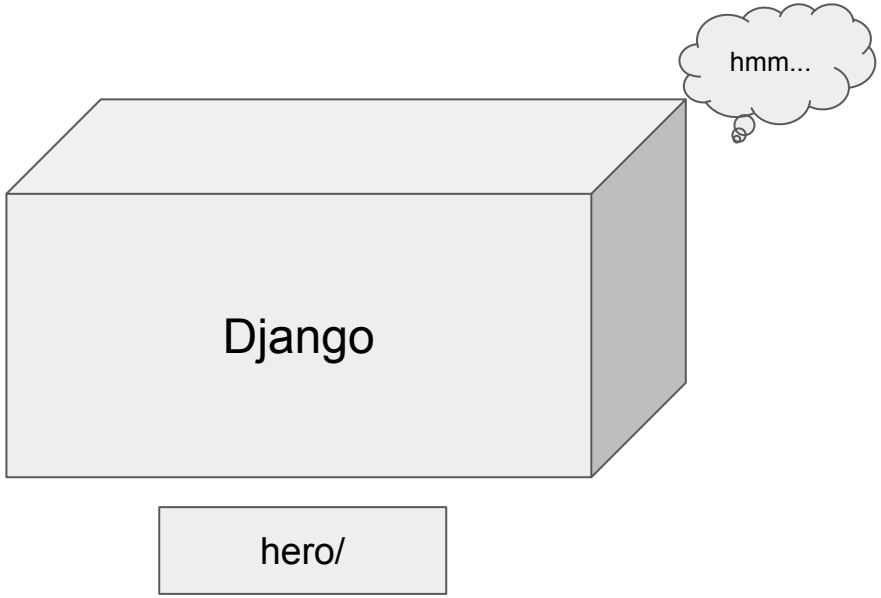
What’s going on?

1. HTTP request arrives to Django.



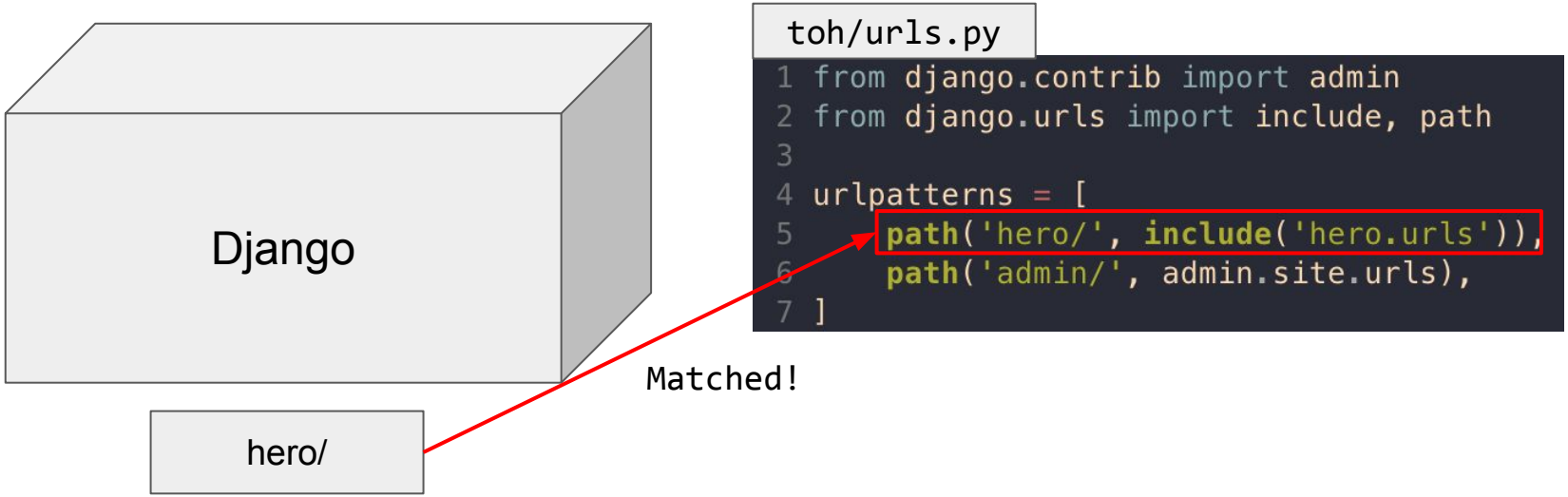
What’s going on?

2. Django looks at the url, and starts to match up with url configurations.



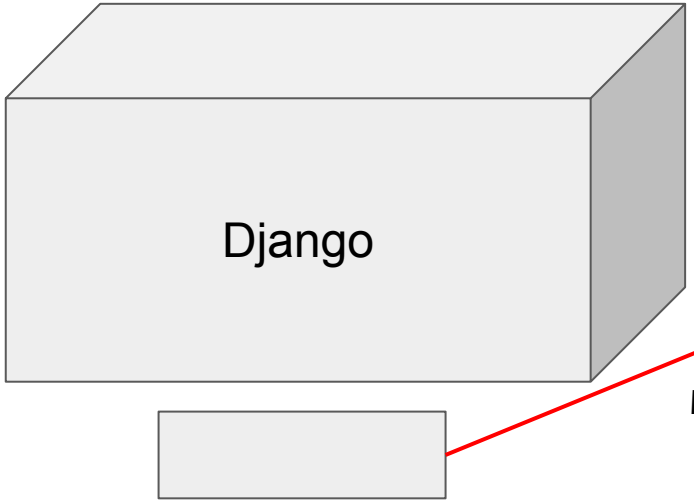
What's going on?

3. First, Django always looks up the 'root' urlconf (`toh/urls.py`).



What’s going on?

4. Consumes matched URL part, and keeps matching with remaining.



hero/urls.py

```
1 from django.urls import path
2
3 from . import views
4
5 urlpatterns = [
6     path('', views.index, name='index'),
7 ]
```

Matched!

Note: `hero/` was consumed at the root urlconf

What’s going on?

5. Finally, Django invoke the view function with HTTP request context.

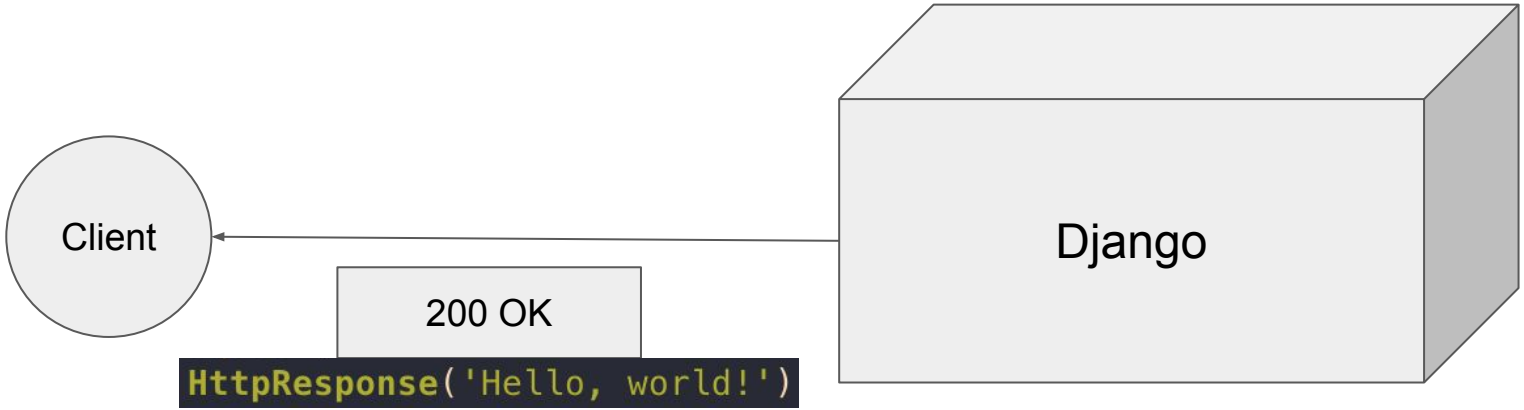
GET /hero/

hero/views.py

```
1 from django.http import HttpResponse
2
3 def index(request):
4     return HttpResponse('Hello, world!')
```

What’s going on?

6. Send the return value of view function back as response.



More URLconf

- You can also “capture” the variable from URL.

```
urlpatterns = [
    path('articles/2003/', views.special_case_2003),
    path('articles/<int:year>/', views.year_archive),
    path('articles/<int:year>/<int:month>/', views.month_archive),
    path('articles/<int:year>/<int:month>/<slug:slug>/', views.article_detail),
]
```

- For more info, read the following document.
  - https://docs.djangoproject.com/en/4.1/topics/http/urls/

Today's Task 1

- Write two more views.
  - one with capturing int from URL and returning it.
    - Hint: use <int:id> to capture it.
  - one with capturing string from URL and returning it.
    - Hint: use <str:name> to capture it.

Use captured value by keyword args

```
def hero_name(request, name=""):
    return HttpResponse('Your name is
```

```
~/toh/hero
curl http://localhost:8000/hero/10/
Your id is 10!
~/toh/hero
curl http://localhost:8000/hero/ironman/
Your name is ironman!
```

You need database

- There are many DB backends
  - MySQL
  - PostgreSQL
  - SQLite
  - etc
- What if you want to change DB backend in development?
  - or even in production?

ORM

- Object-relational mapping
  - Use OOP concept to make data converting compatible between incompatible system

- Django model
  - Use Python code to interact with database.
  - No breaking change in your model code even if DB backend is changed.

Before you use Model

- Let’s learn the basics about database

Database Basics

Terminology

- Record (Row)
  - Data for a single item (e.g. person, course, etc.)
- Field (Column)
  - Part of a record that contains a single piece of data
  - e.g. name of the person, classroom location of the course
- Schema
  - Definition of database including tables, fields, and types.
- Migration
  - Change of schema and database from one to another

Terminology

Record

Student_ID	Name	Age	Language
2017-11111	Shin	24	Python
2017-22222	Song	25	Typescript

Field

Schema

Student_ID [INT: 9]	Name [CHAR: 10]	Age [INT: 4]	Language [CHAR: 10]

Terminology

- Primary Key
  - Set of one or more fields which can uniquely identify a record in the table
  - e.g. student id,
- Relationship
  - Logical connection between different tables
- Foreign Key
  - Key for identifying the related record in other table
  - Necessarily a primary key in the related table

Terminology

Primary Key

Foreign Key

Student ID	Name	Team Number
11111	Alice	1
22222	Bob	2
33333	Charlie	2
44444	Dave	3

Primary Key

Team Number	Team Name
1	Wonderland
2	B.C.
3	Doomed

Let’s think about relationship

- Let’s assume that you want to preserve relationship between two different fields.
  - e.g. Book - Author, Team - Person, Lecture - Student
- Followings are types of relationships that can exist between two entities.
  - One-to-One
  - Many-to-One
  - Many-to-Many

One to One

- One record in a table is associated with one and only one record in another table.
- No need to make seperate table
  - Just add a field into the table

ID	Name	Student number
1	Quick brown fox	2020-12345

Many to One

- one instance of an entity - one or more instance of another entity
- You need to separate table into two
- You need additional field to indicate the row of another table

ID	Name	Team id
1	Alice	1
2	Bob	1
3	Carol	2
4	Eve	3

ID	Team
1	swpp-team1
2	swpp-team2
3	swpp-team3

What if a person can join more than one Team?

Many to Many

- You need an additional table to indicate the relationship “arrow”

ID	Name
1	Alice
2	Bob
3	Carol
4	Eve

from	to
1	1
1	2
2	2
2	3
3	1
4	3

intermediary join table

ID	Club
1	Band
2	Soccer
3	Dance

Django model

Let your project know about your app

- Add your `hero` app to `INSTALLED_APPS`
- From now, your project will know the model of your app and be able to generate migrations and write schema in database

toh/settings.py

```
31 # Application definition
32
33 INSTALLED_APPS = [
34     'hero.apps.HeroConfig',
35     'django.contrib.admin',
36     'django.contrib.auth',
37     'django.contrib.contenttypes',
38     'django.contrib.sessions',
39     'django.contrib.messages',
40     'django.contrib.staticfiles',
41 ]
```

Write a first model

- Let’s make our hero model

hero/models.py

```
1 from django.db import models
2
3
4 class Hero(models.Model):
5     name = models.CharField(max_length=120)
```

Make migrations

```
~/toh
python manage.py makemigrations hero
Migrations for 'hero':
  hero/migrations/0001_initial.py
    - Create model Hero
```

- **`$ python manage.py makemigrations hero`**
  - This command will create **migration** of your model.
    - The changes in `models.py` are first saved under `migration` via **`python manage.py makemigrations`**, and then gets reflected to DB schema via **`python manage.py migrate`**
  - Keep track of changes in DB schema.
    - (version control system for DB schema)
    - makemigrations ⇔ git commit
  - Should be committed into your version control system.
  - Each app contains own migrations.

Migrate

```
~/toh
python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, hero, sessions
Running migrations:
  Applying hero.0001_initial... OK
```

- `$ python manage.py migrate`
  - Commit all changes of DB schema into your database.
  - Hopefully, our Hero model is now mapped into the database.

`makemigrations` vs `migrate`

- `makemigrations`
  - Generate SQL queries that alter DB schema.
  - Give you warning when your modification is conflicting with existing DB schema.
    - e.g. setting nullable field to not null without providing default value
  - No affect on real database.
- `migrate`
  - Actually apply the queries that generated from `makemigrations`.

When should I run migration?

- When you add, delete, or modified the field
- When you add new model, or delete existing model

Test the model

- Open Django interactive shell
  - `$ python manage.py shell`
  - All of the environment will be set up to interact with Django
  - Same as Python shell

```
~/toh
python manage.py shell
Python 3.7.4 (default, Aug 13 2019, 15:17:50)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.8.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

Playing with model

- Initially, there is no record in Hero table

```
In [1]: from hero.models import Hero

In [2]: Hero.objects.all()
Out[2]: <QuerySet []>

In [3]:
```

Create a hero

- Creating hero by making Hero class object

```
In [3]: hero = Hero(name='Superman')

In [4]: hero
Out[4]: <Hero: Hero object (None)>

In [5]: hero.name
Out[5]: 'Superman'
```

Save the hero model

- You should invoke `save()` method to actually save the model into the database
- Now this model have its own `id`, which is the primary key
  - Auto-generated by Django
    - Of course, you can override this behaviour by modifying your settings

```
In [6]: hero.save()

In [7]: hero.id
Out[7]: 1
```

Change field data

- You can modify the attribute of model object
- You should invoke `save()` method, as the same reason

```
In [8]: hero.name = 'Batman'

In [9]: hero.save()

In [10]: hero.name
Out[10]: 'Batman'
```

Change REPL representation

- Exit interactive shell, and edit `models.py`
- Add `__str__` method
- No need to migrate!
  - This is not altering any DB schema
- Ensure it works

hero/models.py

```
1 from django.db import models
2
3
4 class Hero(models.Model):
5     name = models.CharField(max_length=120)
6
7     def __str__(self):
8         return self.name
```

```
In [1]: from hero.models import Hero

In [2]: Hero.objects.all()
Out[2]: <QuerySet [<Hero: Batman>]>
```

QuerySet

- QuerySet is a list of model instance from DB.
- ORM is easy to use, but might be slower than hand-written SQL query.
- We need to understand how ORM create queryset.
- Frequent query makes your service slow.
- <u>QuerySet is lazily evaluated</u>, when the results are needed.
  - **`hero_set = Hero.objects.filter(name='Batman')`**
    - No query is delivered to DB.
  - You can evaluate a QuerySet in the following ways
    - condition
    - iterating
    - slicing
    - `len(), repr(), list()`
    - `bool()`

QuerySet

Query is evaluated, and cached.

- The evaluated queryset is cached.
- Excessive queryset cache degrades your application.
  - Extremely large records
- Use `.exists()` and `.iterator()`

```
In [7]: for hero in hero_set:
   ...:     print(hero.name)
   ...:
Batman

In [8]: for hero in hero_set:
   ...:     print(hero.name)
   ...:
Batman
```

Use cached queryset

QuerySet

- `.exists()` query checks the existence of record.
- `.iterator()` reads results without any caching at queryset level.
  - The queryset will be re-evaluated at the next re-use.

```
In [1]: from hero.models import Hero

In [2]: hero_set = Hero.objects.all()

In [3]: if hero_set:
   ...:     for hero in hero_set:
   ...:         print(hero.name)
```

```
In [1]: from hero.models import Hero

In [2]: hero_set = Hero.objects.all()

In [3]: if hero_set.exists():
   ...:     for hero in hero_set.iterator():
   ...:         print(hero.name)
```

Making queries

- Create some other heroes.

```
In [3]: Hero(name='Ironman').save()

In [4]: Hero(name='Hulk').save()

In [5]: Hero(name='Spiderman').save()
```

Making queries

- Various kind of search queries can be used.

```
In [1]: from hero.models import Hero

In [2]: Hero.objects.filter(name='Spiderman')
Out[2]: <QuerySet [<Hero: Spiderman>]>

In [3]: Hero.objects.filter(name__endswith='man')
Out[3]: <QuerySet [<Hero: Batman>, <Hero: Ironman>, <Hero: Spiderman>]>

In [4]: Hero.objects.get(name='Hulk')
Out[4]: <Hero: Hulk>

In [5]: Hero.objects.get(name='Antman')
---------------------------------------------------------------------------
DoesNotExist                              Traceback (most recent call last)
<ipython-input-5-c6ba49598726> in <module>
----> 1 Hero.objects.get(name='Antman')
```

Making queries

- `filter()` method can search table with advanced options.
  - e.g. case insensitive, contains, startswith, endswith…
  - Use double-underscore(__) in keyword argument name.
  - See this doc:
    - https://docs.djangoproject.com/en/4.1/topics/db/queries/#field-lookups-intro

```
In [3]: Hero.objects.filter(name__endswith='man')
Out[3]: <QuerySet [<Hero: Batman>, <Hero: Ironman>, <Hero: Spiderman>]>
```

Making queries

- Search model objects by `filter()`.
  - Always return `QuerySet` object.
    - Similar behaviour as Python List.
    - You can traverse, get by index, and etc.
    - Empty QuerySet when there is no matching objects with query.

```
In [2]: Hero.objects.filter(name='Spiderman')
Out[2]: <QuerySet [<Hero: Spiderman>]>
```

Making queries

- `get()` method retrieves a single model object
  - Same usage as `filter()`
  - `DoesNotExist, MultipleObjectsReturned` exception can be raised
  - See this doc:
    - https://docs.djangoproject.com/en/4.1/topics/db/queries/#retrieving-a-single-object-with-get

```
In [4]: Hero.objects.get(name='Hulk')
Out[4]: <Hero: Hulk>

In [5]: Hero.objects.get(name='Antman')
---------------------------------------------------------------------------
DoesNotExist                              Traceback (most recent call last)
<ipython-input-5-c6ba49598726> in <module>
----> 1 Hero.objects.get(name='Antman')
```

Use your model in view

- Just import your model class and write code

```
# hero/urls.py
from django.urls import path

from . import views

urlpatterns = [
    path('', views.hero_list)
]
```

```python
# hero/views.py
from django.http import HttpResponseBadRequest, HttpResponseNotAllowed, JsonResponse
from django.views.decorators.csrf import csrf_exempt
import json
from json.decoder import JSONDecodeError
from .models import Hero

@csrf_exempt
def hero_list(request):
    if request.method == 'GET':
        hero_all_list = [hero for hero in Hero.objects.all().values()]
        return JsonResponse(hero_all_list, safe=False)
    elif request.method == 'POST':
        try:
            body = request.body.decode()
            hero_name = json.loads(body)['name']
        except (KeyError, JSONDecodeError) as e:
            return HttpResponseBadRequest()
        hero = Hero(name=hero_name)
        hero.save()
        response_dict = {'id': hero.id, 'name': hero.name}
        return JsonResponse(response_dict, status=201)
    else:
        return HttpResponseNotAllowed(['GET', 'POST'])
```

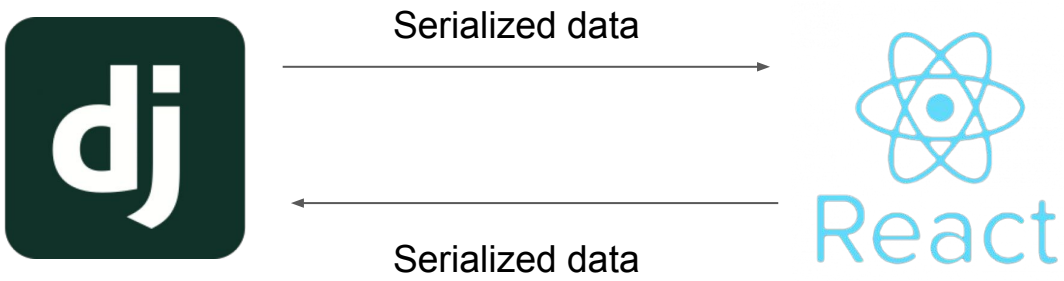
Example code explained

```
if request.method == 'GET':
    hero_all_list = [hero for hero in Hero.objects.all().values()]
    return JsonResponse(hero_all_list, safe=False)
```

- Return whole hero list.
- `values()` method in `QuerySet` returns List of Python dict representation of each model, not the model object itself.
- Why?
  - Because dict is serializable into JSON, but model object itself is not.

Serialization

- Frontend uses Javascript, Django uses Python
  - Javascript cannot understand the model object of Django, vice versa
- We have to convert model object into common data type, which should be storable and easy to transmitted to network.



Use JSON to serialize your data

- JSON(JavaScript Object Notation)
  - https://www.json.org/
- Readable
- Text format
  - Yes! We can transfer it using HTTP
- Can represent common data structure

```
{
   "username":"swpp",
   "favorites":[
    "Python",
    "Django"
   ],
   "profile":{
    "email":"swpp@snu.ac.kr",
    "dog":"cat",
    "number":1,
    "float":-1.02
   }
}
```

Example code explained

```
if request.method == 'GET':
    hero_all_list = [hero for hero in Hero.objects.all().values()]
    return JsonResponse(hero_all_list, safe=False)
```

- `JsonResponse` object is similar to `HttpResponse`, but it returns JSON-serialized data.
  - Read: https://docs.djangoproject.com/en/4.1/ref/request-response/
- The first argument of `JsonResponse` is dict instance.
- With `safe=False` option, any object can be passed for serialization.
  - `safe=False` is needed when you have to send List.
    - vulnerable to json-hijacking attack, but mitigated in almost all modern browsers,

Example code explained

- Create new Hero when request method is POST.

```
elif request.method == 'POST':
    try:
        body = request.body.decode()
        hero_name = json.loads(body)['name']
    except (KeyError, JSONDecodeError) as e:
        return HttpResponseBadRequest()
    hero = Hero(name=hero_name)
    hero.save()
    response_dict = {'id': hero.id, 'name': hero.name}
    return JsonResponse(response_dict, status=201)
```

Example code explained

- Deserialize the request body with `json.loads`
  - `decode()` is built-in method of `bytes`.

```
try:
    body = request.body.decode()
    hero_name = json.loads(body)['name']
```

- Create new `Hero` model object and send Created response

```
hero = Hero(name=hero_name)
hero.save()
response_dict = {'id': hero.id, 'name': hero.name}
return JsonResponse(response_dict, status=201)
```

Example code explained

- Let’s see if it works
  - Send a post request to your server:
    - ```
      $ curl -X POST -H "Content-Type: application/json" \
        -d '{"name":"Spiderman"}' http://127.0.0.1:8000/hero/
      ```
  - Then check if your POST request is handled correctly:
    - `curl -X GET http://127.0.0.1:8000/hero/`
      ```

Today's Task 2

- Add `age` field to your Hero model
  - Hint1: use `models.IntegerField(default=...)`, or

    `models.IntegerField(blank=True, null=True)`
  - Hint2: Don't forget to

```
$ python manage.py makemigrations hero && \
  python manage.py migrate
```

- Then, `$ curl -X GET http://127.0.0.1:8000/hero/`

```
expected response: [{"id": 1, "name": "Spiderman", "age": "25"}]
```

  - Hint3: You'll have to change the view function to get the correct response.

Today's Task 2

- Add a new view function(`hero_info`) and a url pattern, so that it returns a hero with specified id.
  - We will send **GET** request to http://127.0.0.1:8000/hero/info/<int:id>/
  - `$ curl -X GET http://127.0.0.1:8000/hero/info/1/`

    `expected response: {"id": 1, "name": "Spiderman", "age": "25"}`
- Make the view function(`hero_info`) available to modify hero info.
  - We will send **PUT** request to http://127.0.0.1:8000/hero/info/<int:id>/
  - `$ curl -X PUT -H "Content-Type: application/json" \`

    `  -d '{"name":"Batman", "age": "50"}' http://127.0.0.1:8000/hero/info/1/`

    `⇒ expected response: {"id": 1, "name": "Batman", "age": "50"}`
  - `$ curl -X GET http://127.0.0.1:8000/hero/info/1/`

    `⇒ expected response: {"id": 1, "name": "Batman", "age": "50"}`

Relationship

Add relationship in model

- Add `Team` model
  - Contains `ForeignKey`, `ManyToManyField`
- Each team has a leader, and multiple members

hero/models.py

```
4  class Hero(models.Model):
5      name = models.CharField(max_length=120)
6
7      def __str__(self):
8          return self.name
9
10 class Team(models.Model):
11     name = models.CharField(max_length=120)
12     leader = models.ForeignKey(
13         Hero,
14         on_delete=models.CASCADE,
15         related_name='leader_set',
16     )
17     members = models.ManyToManyField(
18         Hero,
19         related_name='teams',
20     )
21
22     def __str__(self):
23         return self.name
```

[Recap] Foreign Key

Terminology

Primary Key

Foreign Key

Student ID	Name	Team Number
11111	Alice	1
22222	Bob	2
33333	Charlie	2
44444	Dave	3

Primary Key

Team Number	Team Name
1	Wonderland
2	B.C.
3	Doomed

on_delete

- When an object referenced by a `ForeignKey` is deleted, the specified behavior will be emulated.
- Example
  - `Team` instance will be deleted when its `leader` (`Hero` instance) is deleted.

hero/models.py

```
4  class Hero(models.Model):
5      name = models.CharField(max_length=120)
6
7      def __str__(self):
8          return self.name
9
10 class Team(models.Model):
11     name = models.CharField(max_length=120)
12     leader = models.ForeignKey(
13         Hero,
14         on_delete=models.CASCADE,
15         related_name='leader_set',
16     )
17     members = models.ManyToManyField(
18         Hero,
19         related_name='teams',
20     )
21
22     def __str__(self):
23         return self.name
```

related_name

- Provides *reverse-lookup*
- Example
  - `hero.leader_set` returns the `QuerySet<Team>` that contains team whose leader is `hero`

hero/models.py

```
4  class Hero(models.Model):
5      name = models.CharField(max_length=120)
6
7      def __str__(self):
8          return self.name
9
10 class Team(models.Model):
11     name = models.CharField(max_length=120)
12     leader = models.ForeignKey(
13         Hero,
14         on_delete=models.CASCADE,
15         related_name='leader_set',
16     )
17     members = models.ManyToManyField(
18         Hero,
19         related_name='teams',
20     )
21
22     def __str__(self):
23         return self.name
```

Migrate

- Since we added new model, we have to make migrations to DB
- `$ python manage.py makemigrations hero`
- `$ python manage.py migrate`

```
~/toh
python manage.py makemigrations hero
Migrations for 'hero':
  hero/migrations/0003_auto_20201009_0748.py
    - Remove field age from hero
    - Create model Team
~/toh
python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, hero, sessions
Running migrations:
  Applying hero.0003_auto_20201009_0748... OK
```

Test the model relationship

- Create a team
- To fill the `ForeignKey` field, simply assign a model object
- Don’t forget to save the model

```
In [1]: from hero.models import Hero, Team

In [2]: team_marvel = Team(name='Marvel')

In [3]: ironman = Hero.objects.get(name='Ironman')

In [4]: team_marvel.leader = ironman

In [5]: team_marvel.save()
```

Test the model relationship

- Add heroes to team member
- No need to save
  - `ManyToManyField` is automatically committed for each query
- You can then make queries to `members` similarly

```
In [6]: hulk = Hero.objects.get(name='Hulk')

In [7]: spiderman = Hero.objects.get(name='Spiderman')

In [8]: blackpanther = Hero.objects.get(name='BlackPanther')

In [9]: team_marvel.members.add(hulk, spiderman, blackpanther)
```

```
In [10]: team_marvel.members.all()
Out[10]: <QuerySet [<Hero: Hulk>, <Hero: Spiderman>, <Hero: BlackPanther>]>
```

More about model

- Model Field reference
  - https://docs.djangoproject.com/en/4.1/ref/models/fields/
- Making queries
  - https://docs.djangoproject.com/en/4.1/topics/db/queries/

Today's task 3

- Add `IntegerField` named `score` with default value 0
- Add `introduce()` method which prints like below

```
In [4]: hero = Hero.objects.get(name='Batman')

In [5]: hero.introduce()
Out[5]: 'Hello, my name is Batman and my score is 100!'
```

- Hints
  - https://docs.djangoproject.com/en/4.1/ref/models/fields/#default
  - https://docs.djangoproject.com/en/4.1/topics/db/models/#model-methods

Thank you!

- Any questions?