Django and Interacting with Protocols

Helpful References

- https://docs.djangoproject.com
- https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django
- https://simpleisbetterthancomplex.com/series/2017/09/04/a-complete-beginners-guide-to-django-part-1.html

Django

- Popular framework for writing server-side code in Python
- Structure is complex
- Recommendation:
  - Download the slides
  - Write down questions as you listen to this presentation
  - Review your questions; some should have been answered by the presentation
  - Start working on Lab 7, to resolve the remaining questions
  - Of course, feel free to contact me at any point

Django Files

Django Project Setup

Please try this now!

- From within PyCharm:
  Create a new project (do **not** skip this step!)
  pipenv install django

- Outside of Pycharm:
  cd into the project directory (do **not** skip this step!)
  pipenv shell
  pipenv install django

- Create a Django project (say, *ics226*):
  django-admin startproject ics226

Project File Structure

- ics226/
    - manage.py
    - ics226/
        - __init__.py
        - asgi.py
        - settings.py
        - urls.py
        - wsgi.py

Django Project Files

- Default files that are part of a Django project:
  - *manage.py* - tool for managing the project
  - *__init__.py* - marks a directory as a package directory; allows statements like *from <package> import <module>* where the module is the name of a Python file in the directory
  - *settings.py* - contains settings related to the project (e.g., *DEBUG* flag)
  - *urls.py* - maps URLs to web apps that are part of the project (e.g., /*admin* to the *admin* app)
  - *asgi.py*/*wsgi.py* - ignore (used for integration with the web server)

Verifying Project Setup

Please try this now!

- Launch the server from the *ics226* directory, using:
  ./manage.py runserver

- Problem: For security reasons, this is only running on localhost port 8000 (i.e., your VM)

- Must set up an ssh tunnel for port 8000 traffic from the A machine to your VM:
  ssh -L 8000:127.0.0.1:8000 _____@_____
  where the first blank is your user ID, and the second blank your VM's IP address

- Can now browse to http://127.0.0.1:8000

- Once the server is running, you can keep it running, even when modifying files; changes will be applied on the fly

Django App Setup

Please try this now!

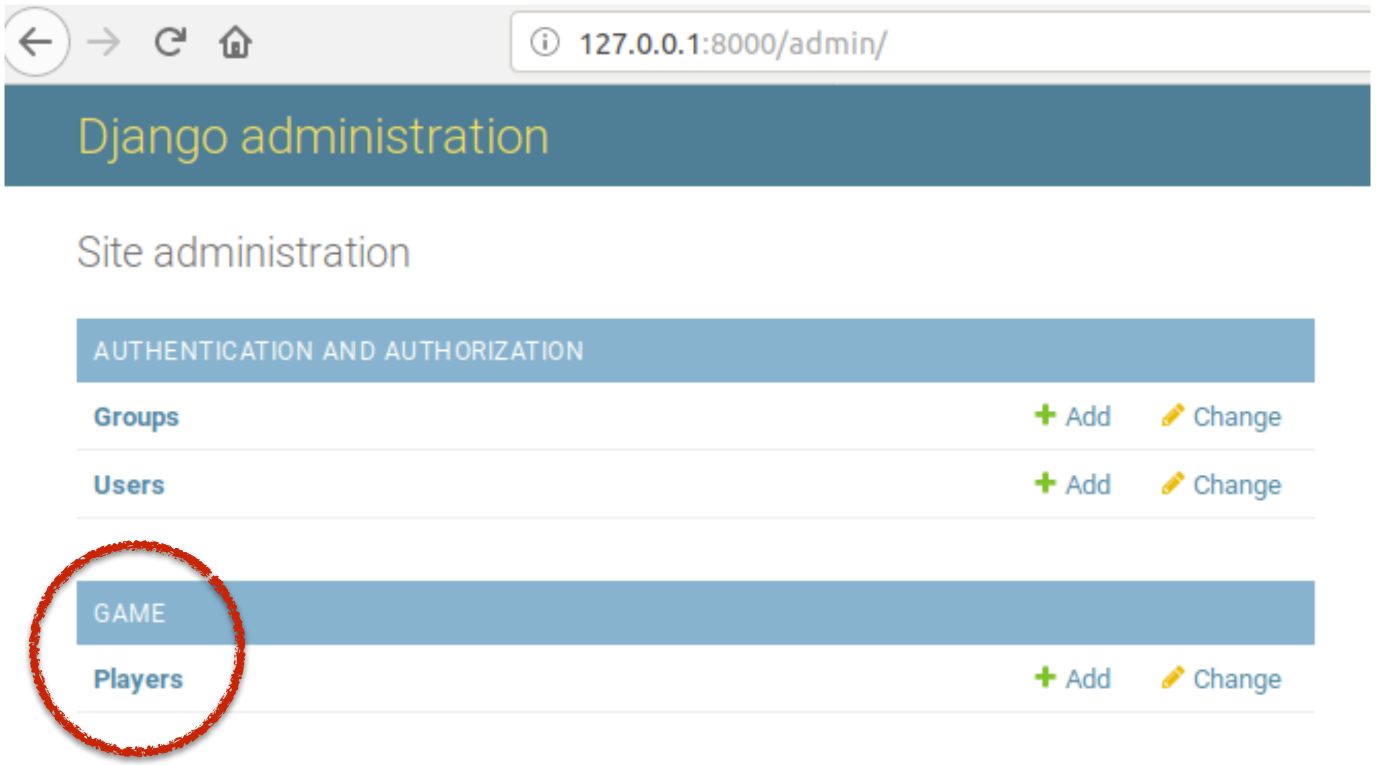
- A Django project is made up of several apps (e.g., a game app and a level editor app)
- Say we want to create a *game* app:
  ./manage.py startapp game

Project File Structure

- ics226/
  - db.sqlite3
  - game/
    - __init__.py
    - admin.py
    - apps.py
    - migrations/
    - models.py
    - tests.py
    - views.py
  - manage.py
  - ics226/
    - __init__.py
    - asgi.py
    - settings.py
    - urls.py
    - wsgi.py

Django App Files

- *db.sqlite3* - contains the app data
- *__init__.py* - see previous explanation
- *admin.py* - makes models visible from the admin panel (create an admin account using ***./manage.py createsuperuser***)


127.0.0.1:8000/admin/

Django administration

Site administration

AUTHENTICATION AND AUTHORIZATION		
Groups	+ Add	Change
Users	+ Add	Change

GAME		
Players	+ Add	Change

Django App Files

- *apps.py* - ignore (used for integration)
- *models.py* - contains the data models (must be registered in *admin.py*)
- *migrations* - stores files necessary for migrating data models from *models.py* to the database
- *tests.py* - used for testing
- *urls.py* - not created automatically; maps app-specific URLs to views
- *views.py* - contains app-specific views (web pages)

Creating a Simple View
(A Static Web Page)

Creating Web Pages

- To create a web page in a Django web app, there are two main files in the **app's** directory that need to be updated
  - *urls.py* - maps a URL (provided by a web browser to the web server) to a function in *views.py*
  - *views.py* - contains the function that responds to a request and generates the appropriate HTML code

Creating a View, Step 1

Please try this now!

- In the *game* directory, modify *views.py* so that it contains a function corresponding to an individual view

```
from django.http import HttpResponse

def index(request):
    return HttpResponse('Hello world!')
```

- The *request* variable contains information regarding the request, e.g., <*WSGIRequest: GET '/game/'*>

- The *HttpResponse* is what is sent back to the web browser; here we only set the body, but the header can be modified as well

Creating a View, Step 2

Please try this now!

- Create a *urls.py* file inside the same directory as *views.py*

- *urls.py* tells Django how to map a URL pattern to a function in *views.py*

```
from django.urls import path
from . import views

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

- The first argument is the route (blank in this case), the second the view (the index function in the *views.py* file), and the third a name that is useful for automatic URL generation (useful for forms)

Creating a View, Step 3

Please try this now!

- If not already done so, make the Django project aware of the new app URLs file by updating the **project-wide** *urls.py* file

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('admin/', admin.site.urls),
    path('game/', include('game.urls')),
]
```

- http://127.0.0.1:8000/game/ will now display *Hello World*

Adding Variables

Please try this now!

- Add the following to *views.py*:

```
def greet(request, name):
    return HttpResponse(f'Hello {name}')
```

- Modify *urlpatterns* in *game/urls.py* as follows:

```
urlpatterns = [
    path('', views.index, name='index'),
    path('greet/<str:name>', views.greet, name='greet'),
]
```

- http://127.0.0.1:8000/game/greet/Michael will now display *Hello Michael*

Creating a Model
(A Database Table)

Creating a Model

- Models in Django are essentially classes that are mapped to database tables
- They can be managed via the Django website, the Django app, or using SQLite3 tools

Game Description

- Turn-based
- 2 Players on a 2D board
- The board contains treasures and obstacles
- Can move 1 tile at a time
- Player to obtain the most points, once all treasures have been found, wins
- This is **NOT** Lab 7, although the gameplay is similar

Creating a Model, Step 1

- Update the *models.py* file to contain a Python-based description of your database tables

```
from django.db import models


class Player(models.Model):
    tag = models.CharField(max_length=1)
    row = models.IntegerField()
    col = models.IntegerField()

    def __str__(self):
        return f'{self.tag} @({self.row}, {self.col})'
```

- __*str*__ is part of the Player class; make sure you indent it correctly

Creating a Model, Step 2

- Update *admin.py* so that it is possible to access the Player table from the admin panel

```
from django.contrib import admin
from .models import Player

admin.site.register(Player)
```

Creating a Model, Step 3

- Update the *settings.py* file; the *migrate* command on the next slide will only generate database tables for apps included in this list

```
...
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'game.apps.GameConfig',
]
...
```

Creating a Model, Step 4

- Create a database migration script and then execute it using the commands

```
./manage.py makemigrations
./manage.py migrate
```

- Any time you add or remove attributes to or from Player, or add additional models, be sure to rerun these commands!

Accessing a Model
(A Database Table Lookup)

Looking up an Object, Step 1

- Add the following to *views.py*

```
from .models import Player


def get_player(request, player_id):
    players = Player.objects.filter(pk=player_id)
    if len(players) == 1:
        player = players[0]
        return HttpResponse(f'Player {player.tag} is at row {player.row} and col 
{player.col}')
    else:
        return HttpResponse('No such player')
```

- *player* is a collection of objects (a *QuerySet*)
- *Player.objects.all()* would return all Players

Looking up an Object, Step 2

- Update *urls.py* to include

  ```
  path('player/<int:player_id>/', views.get_player, name='player'),
  ```

- This enables http://127.0.0.1:8000/game/player/1/ to call *views.get_player* with the *player_id* argument set to '1'

Modifying a Model (A Database Table Insertion/ Update/Deletion)

Modifying an Object

- Django provides default views for creating, updating, and deleting views

Modifying an Object, Step 1

- First, we need a form in *game/templates/game/player_form.html*

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Player</title>
    </head>
    <body>
        <form action="" method="post">
            {% csrf_token %}
            <table>
                {{ form.as_table }}
            </table>
            <input type="submit" value="Submit">
        </form>
    </body>
</html>
```

- Don't forget to restart the server, to allow it to find this form

Modifying an Object, Step 2

- Next, add the following to *views.py*

```
from django.urls import reverse_lazy
from django.views.generic.edit import CreateView, UpdateView


class PlayerCreate(CreateView):
    model = Player
    fields = '__all__'
    success_url = reverse_lazy('players')

class PlayerUpdate(UpdateView):
    model = Player
    fields = ['row', 'col']
    success_url = reverse_lazy('players')
```

- Note the *players* name in *reverse_lazy*; this will look for *players* in *urlpatterns* (*get_all_players* method, next slide)

Modifying an Object, Step 3

- Add the following to *urlpatterns* in *urls.py*

```
path('player/', views.get_all_players, name='players'),
path('player/create/', views.PlayerCreate.as_view(), name='player_create'),
path('player/update/<int:pk>/', views.PlayerUpdate.as_view(),
name='player_update'),
```

- In *views.py*, add a *get_all_players* method

```
def get_all_players(request):
    players = Player.objects.all()
    result = ''
    for player in players:
        result += str(player) + '<br>'
    return HttpResponse(result)
```

- You can now go to http://127.0.0.1:8000/game/player/create/ to create a new player, and http://127.0.0.1:8000/game/player/update/1/ to update the first player

Working Without an HTML Form

- Can use *@classmethod* to instantiate a model object, e.g.
  - *models.py*:

```
@classmethod
def create_player(cls):
    model = cls(tag='A', row=0, col=0)
    return model
```

  - *views.py*:

```
Player.create_player().save()
```

- Can also get an object (e.g., use *filter* and assign the result to *current_player*) and make changes:

```
current_player.row += 1
current_player.save()
```

Further Refinements

Rendering HTML

- There is another way to integrate HTML code
- Create a file *game/templates/game/player_list.html*

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Players</title>
    </head>
    <body>
        {% if player_list %}
            <ul>
                {% for player in player_list %}
                    <li><a href="{% url 'player' player.id %}">{{ player.tag }}</
a></li>
                {% endfor %}
            </ul>
        {% else %}
            <p>No players.</p>
        {% endif %}
    </body>
</html>
```

Rendering HTML

- In *views.py*, replace *get_all_players* with

```
def get_all_players(request):
    player_list = Player.objects.all()
    context = {'player_list': player_list}
    return render(request, 'game/player_list.html', context)
```

- This will now display all the player tags in the form of HTML links; clicking on a link will display details about a player by placing another HTTP request

Refining Searches

- We can also add pattern matching to our database searches
- In *urls.py*, add:

```
path('player/search/<str:name>/', views.get_player_by_name, name='player_by_name'),
```

- In *views.py*, add:

```
def get_player_by_name(request, name):
    players = Player.objects.filter(tag__startswith=name)
    context = {'player_list': players}
    return render(request, 'game/player_list.html', context)
```

- This will now display all the player tags that start with a given name

Generating 404 Errors

- In *views.py*, update the following

```
from django.shortcuts import get_object_or_404

def get_player(request, player_id):
    player = get_object_or_404(Player, pk=player_id)
    return HttpResponse(player)
```

- This will now return a 404 if an invalid player ID is provided

Working With Buttons

- Create a file *game/templates/game/button.html*

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Greeting</title>
    </head>
    <body>
        Hello {{ name }}, how are you?
        <br>
        <form action="{% url 'followup' %}" method="post">
            {% csrf_token %}
            <button name="button_id" value="fine">Fine</button>
            <button name="button_id" value="not_fine">Not Fine</button>
        </form>
    </body>
</html>
```

Working With Buttons

- In *urls.py*, add

```
path('followup/', views.followup, name='followup'),
```

- In *views.py*, modify

```
def greet(request, name):
    return render(request, 'game/button.html', {'name': name})
```

- and add

```
def followup(request):
    reply = 'Sorry to hear that'
    try:
        answer = request.POST['button_id']
        if answer == 'fine':
            reply = 'Great'
    except KeyError as details:  # in case button_id is not found
        reply = 'Sorry...'
    return HttpResponse(reply)
```

Working With Buttons

- When going to the greetings link, say http://127.0.0.1:8000/game/greet/Michael, the page will now present two buttons
- Clicking either button will go to the *followup* function, which will reply differently, depending on which button was pressed

Encoding in JSON

- To encode simple types, we can just use *json.dumps(o)* where *o* is a dictionary, integer, list, or string

- To encode classes, we need to do two things:

```
class PlayerEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, Player):
            return {'id': obj.id, 'tag': obj.tag, 'row': obj.row, 
'col': obj.col}
        return json.JSONEncoder.default(self, obj)
```

- Then, when we encode, we need to use *json.dumps(player, cls=PlayerEncoder)*

- You cannot directly return a JSON, but have to wrap it inside an HttpResponse; if you ever get an error relating to a response that is missing context or a *get*, you most likely forgot to do that

Validating a Model

Validating a Model

- We are still missing validation. Update *models.py* to check user entries

```
from django.core.exceptions import ValidationError

def validate_col_range(value):
    if value < 1 or value > 10:
        raise ValidationError('Column out of range', code='col_value')

def validate_row_range(value):
    if value < 1 or value > 10:
        raise ValidationError('Row out of range', code='row_value')

def validate_unique_tag(value):
    players = Player.objects.filter(tag=value)
    if len(players) != 0:
        raise ValidationError('Tag already taken', code='duplicate')

class Player(models.Model):
    tag = models.CharField(max_length=1, validators=[validate_unique_tag])
    row = models.IntegerField(validators=[validate_row_range])
    col = models.IntegerField(validators=[validate_col_range])

    def __str__(self):
        return f'{self.tag} @({self.row}, {self.col})'
```

Validating a Model, Step 2

- We also have to check that the Player does not skip a row or column. This requires a bit more work in *models.py*, **specifically the *Players* class**

```
def clean(self):
    prev = Player.objects.filter(pk=self.pk)
    if len(prev) > 0:
        if abs(self.row - prev[0].row) > 1:
            raise ValidationError('Row too far', code='row_distance')
        if abs(self.col - prev[0].col) > 1:
            raise ValidationError('Column too far', code='col_distance')
```

Testing

Testing in Django

- Tests are launched via ***./manage.py test***
- Tests are defined as methods in *tests.py*; they **must** start with *test_* or Django won't find (and run) them
- *assertEqual* is called to test for equality; failure will be reported as part of the test run

```
from django.test import TestCase
from .models import Player


class PlayerTestCase(TestCase):
    def test_create(self):
        self.client.post('/game/player/create/', {'tag': 'T',
'row': 3, 'col': 7})
        p = Player.objects.get(tag='T')
        self.assertEqual(p.tag, 'T')
        self.assertEqual(p.row, 3)
        self.assertEqual(p.col, 7)
```

Testing in Django

- Responses involving a form can be tested for validation errors using *assertFormError*
- Can also use *Player.objects.get()* to retrieve objects from the database to do further testing

```
def test_out_of_bounds_row(self):
    response = self.client.post('/game/player/create/', {'tag': 'U', 'row': -3, 
'col': 4})
    self.assertFormError(response, 'form', 'row', 'Row out of range')
    try:
        Player.objects.get(tag='U')
        self.fail()
    except Player.DoesNotExist:
        pass  # the player was not created, so all is good
```

- Recall: ***pass*** is a NOP; it does not "pass" the test!

Testing in Django

- *response* on the previous slide can contain form validation errors; sometimes, it is hard to know which field to look for; use *vars(response)* to get all fields of the response variable

- For example, the 'Row too far' message is tied to the form, not a particular field, since validation happens in *clean*, so we can use *vars(response)* to find the attribute that contains the error message (*_container* in this case)

```
        response = self.client.post('/game/player/1/update/', 
{ 'row':3, 'col':0 })
        self.assertIn(b'Row too far', response._container[0])
```

- *assertIn(s,t)* verifies that *s* is contained in *t*

- Use *assertFormError* rather than *assertIn*, if possible; *assertFormError* will allow you to make sure the error is tied to the correct field

Exercises

- On your own:
  - Work on the questions in the *Django* section of *Practice Questions and Solutions*

Labs 7 and 8

- Implement the game using Django
- Write tests

Key Skills

- Create Django web apps
- Test Django web apps