

XJCO3011: Web Services and Web Data



Session #8–

Let's Django

Instructor: Dr. Guilin Zhao
Spring 2023

- In the first coursework, we will be using the **Django** framework.
- Django is a free open-source web framework, written in **Python**.
- Created by Adrian Holovaty and Simon Willison in 2003, and named after the **French jazz guitarist Django** Reinhardt.



Adrian Holovaty



Django Reinhardt



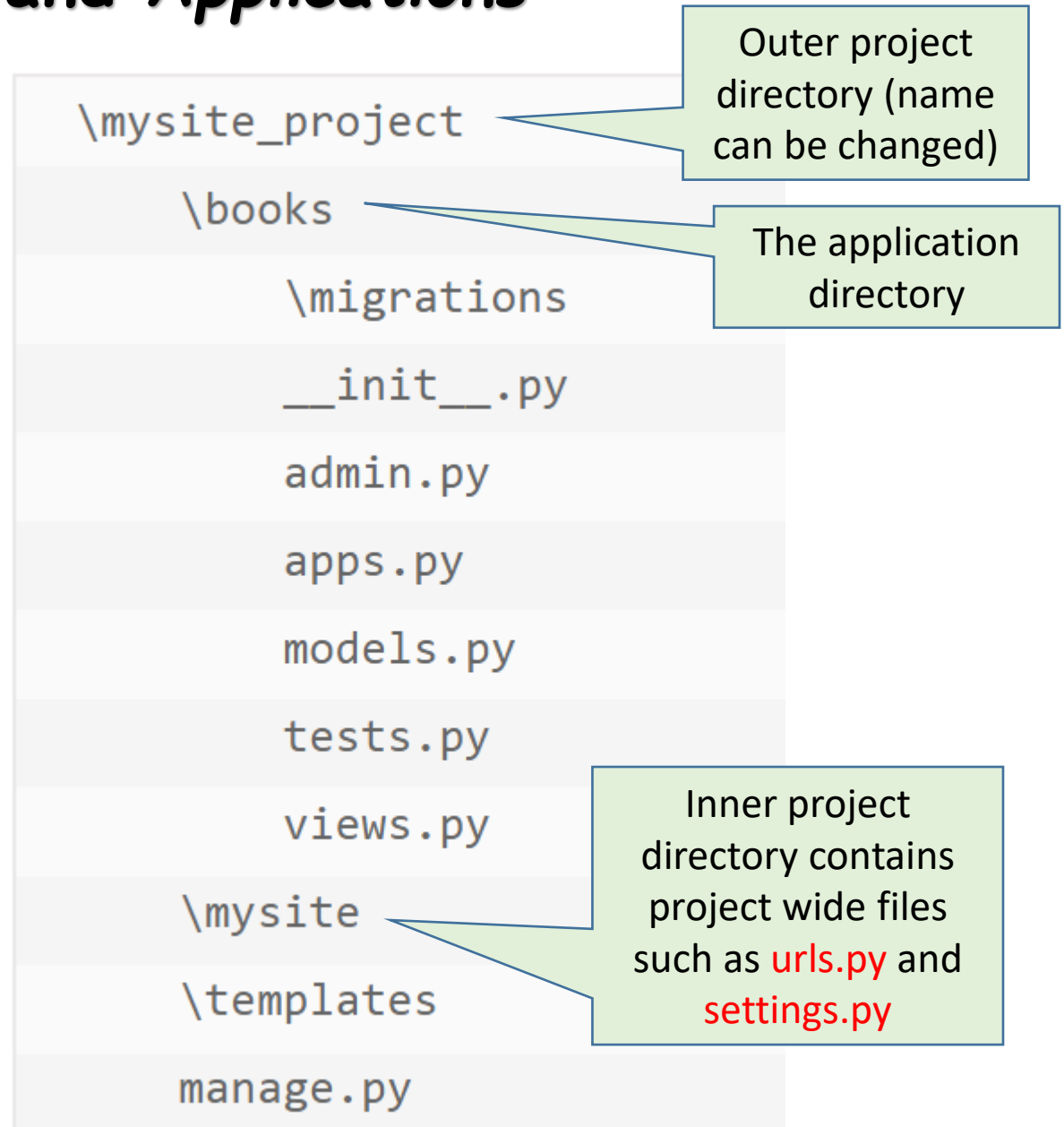
Installing Django

```
$ python -m venv vooo // to create a virtual environment
$ cd vooo
$ source bin/activate // to activate the virtual env.
$ pip install django // to install the latest version of the
Django framework
// to check that django has been successfully installed:
$ python // start the python interpreter
>>> import django
>>> django.VERSION
(2, 1, 5, 'final', 0)
>>> quit ()
```

// the version number of the software or library as 2.1.5,
with a version status of 'final' (i.e., stable release), and a
numerical representation of the version status as 0.

Django Projects and Applications

- A project is created with the command:
django-admin startproject <project name>
- One Django project can have a number of Django applications. An application can be created with:
python manage.py startapp <application name>
- Database models can only live within an application.
- For simple websites, one app is sufficient for all required functionality.



Registering the Application

The application name

- Even though our new app exists within the Django project, Django doesn't "know" about it until we **explicitly add it**.
- Your new application will not work until you register it in the project's **settings.py** file.
- Add your application configuration class to the list of **INSTALLED_APPS**

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

The name of a class that is automatically created in the apps.py file.

The name of this class is derived from the name of the application with first letter capitalized (all other letters converted to small letters) and the word Config appended

In the apps.py file (in the application directory)

```
class BooksConfig(AppConfig):  
    name = 'books'
```

Django Models

- Provide an **ORM** (object relational mapping) **interface** to the underlying relational DBMS (**database management system**).
- Created within the **models.py** file.
- Each database table is represented as one class **derived from the Model** class in the **Django.db.models module**.
- Each column (field) is defined as an attribute using one of the predefined model Fields.
- No need to define the primary key. It is automatically created. It is called id in the tables, and pk in the class model
- A complete list of all available fields and their attributes can be found in **Appendix A of the Django Book**.

```
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()
```

- ❖ CharField is a data type in the Django framework that represents character fields.
- ❖ The max_length parameter specifies the maximum number of characters allowed for this field.

[from Chapter 4 of the Django book](#)

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).
- The first step in using this database layout with Django is to express it as Python code. In the **models.py** file that was created by the startapp command, enter the codes as shown in the picture.
- Each model is represented by a Python class that is a subclass of **django.db.models.Model**. The parent class, Model, contains all the machinery necessary to make these objects capable of interacting with a database – and that leaves our models responsible solely for defining their fields, in a nice and compact syntax.

Field Creation Arguments

- **default**: assigns a default value to the field when it is created, e.g.
`BusinessName = models.CharField (max_length = 64 , default = 'Not Assigned')`
- **choices**: allows the value of a field to be chosen from a list, e.g:
`BusinessTypes = [('airline', 'Airline Company'), ('payment', 'Payment Service Provider')]`
`BusinessType = models.CharField (max_length = 32 , choices = BusinessTypes , default = 'unknown')`
The list should contain tuples of items, the first one is the field value and the second one is a human readable form
- **unique**: enforces the rule that the field value should be unique, e.g.
`BusinessCode = models.CharField (max_length = 8 , unique = True)`
- **null**: allows the field to store null values
- **on_delete**: used with `models.ForeignKey` fields to define what happens to the record when the record in the primary table (the one containing the primary key) is deleted, e.g.:
`destination_airport = models.ForeignKey (Airport, on_delete=models.CASCADE)`
The following options can be used as values for the `on_delete` argument:
 - `models.CASCADE`: when the referenced object is deleted, also delete the objects that have references to it.
 - `models.PROTECT`: forbid the deletion of the **referenced** object.
 - `models.SET_NULL`: set the reference to NULL (requires the field to be nullable).

Creating the underlying relational data base check, makemigrations and migrate!

- The process of transforming model classes into actual database tables.
- In Django, each model class corresponds to a database table, where the class attributes are mapped to table columns and class instances are mapped to table rows. Once the model classes are defined, a Django database migration command needs to be run to generate the corresponding database tables based on the model classes.
- To create the underlying relational database for a Django model, we need to do the following steps:
 1. Check the validity of the model using the check command:
`python manage.py check`
 2. If the model is correct, you can proceed to creating the migration files (files from which the database will be created and synchronized), using the makemigrations command:
`python manage.py makemigrations <application name>`
 3. Finally, create the database using the migrate command:
`python manage.py migrate`

The admin Site

- Django can automatically create an admin site for you.
- This site provides an easy way for site administration, and can be used to populate your data base with actual data.
- To create the admin site, follow the following steps:
 1. Create a super user using the `createsuperuser` command. You will be prompted for the user name, email, and password.

`python manage.py createsuperuser`

2. Edit the `admin.py` file, to import and register your models (tables), for example:

```
...  
from .models import Publisher, Author, Book  
  
admin.site.register(Publisher)  
admin.site.register(Author)  
admin.site.register(Book)
```

In the `admin.py` file (in the application directory)

Testing your data base

To test your database using the admin site, follow the following steps:

1- run the development server using the runserver command:

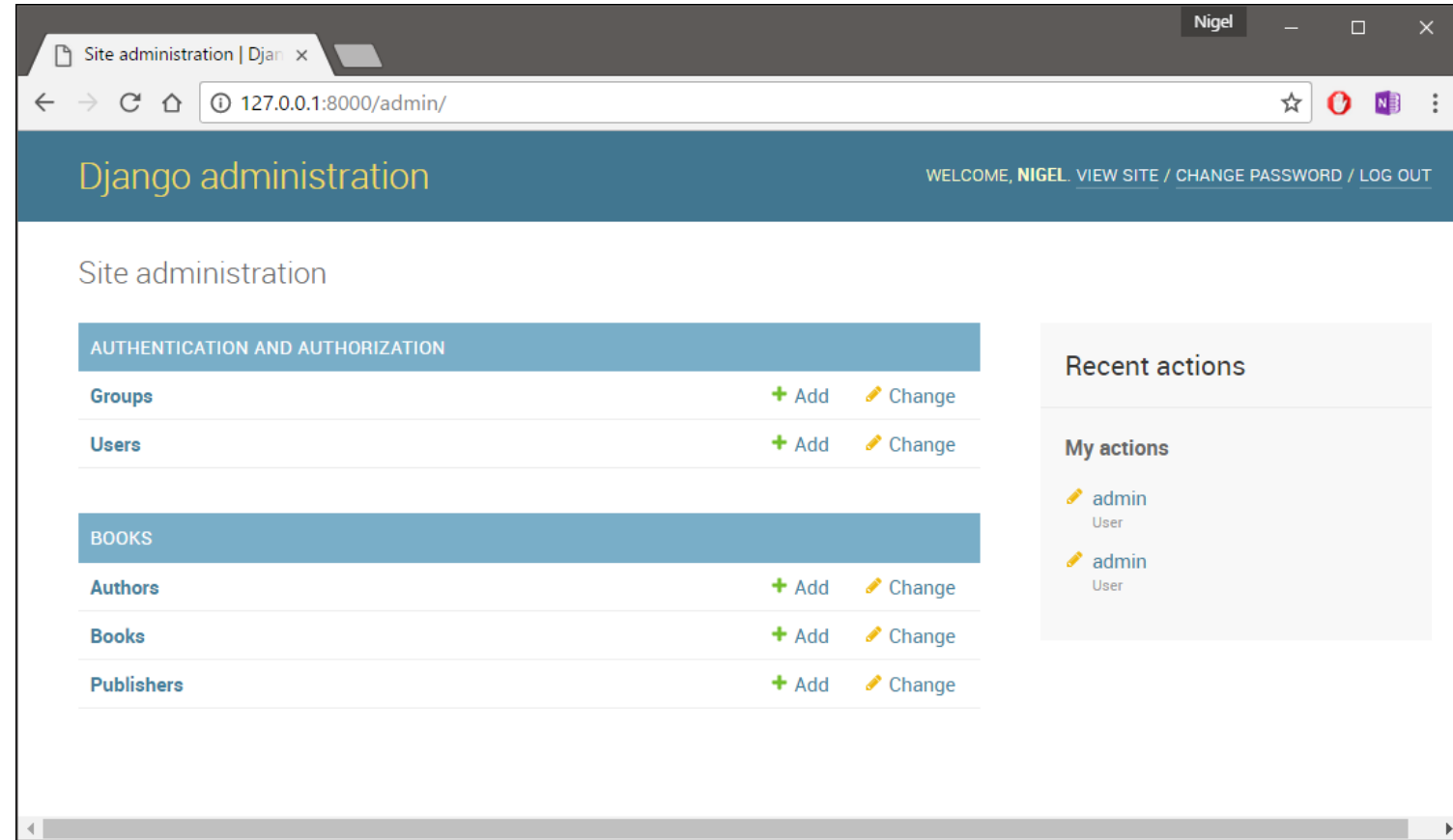
`python manage.py runserver`

2. Open a web browser and type the url of the admin site, in this case it will be

<http://127.0.0.1:8000/admin/>

3. You will be asked for the admin user and password.

4. You can then see an interface for the database tables where you can populate and edit the database content.



Creating the API view functions

- Now that your database model is ready, you can create an API to interact with the database.
- The first step is to create one function to handle each request. These functions are called **view functions** in Django.
- The functions should be defined in the views.py file (in the application directory).
- The first positional argument that will always be passed to this function is an object of type HttpRequest that encapsulates all the elements of the incoming HTTP request.
- Initially you can define a skeleton function for each of the requests, for example:

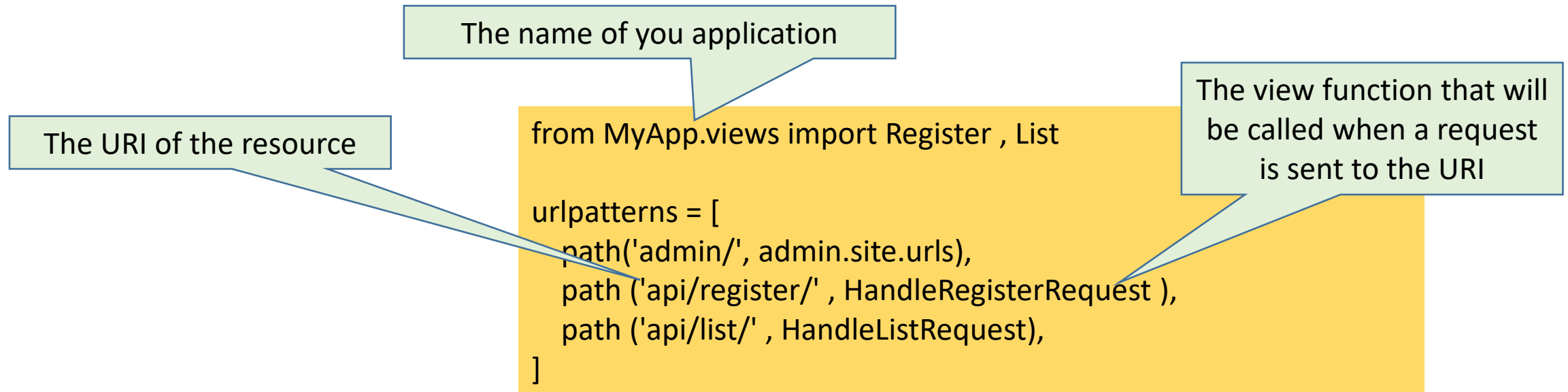
```
def HandleRegisterRequest (request):  
    return HttpResponse ('not yet implemented')
```

In the views.py file (in the application directory)

❖ HandleRegisterRequest is a Python function that takes a request object as a parameter and returns an HttpResponse object.

Linking view functions to resource URIs

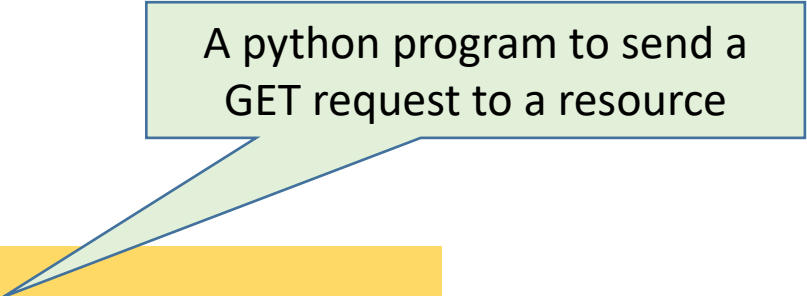
- Each view function should handle a request sent to one of the resources of the API
- Django can automatically call the appropriate view function when a request is received if a resource URI is linked to the functions in the urls.py file (in the project directory)
- To do this, edit the url.py file, and:
 1. Import the view functions from your views.py file
 2. For each function (resource) add a call to path (for fixed URIs), or re_path (for variable URI defined with a regular expressions) to the **urlpatterns** list, for example



Testing the view functions

- Before proceeding to detailed implementation of each view function, you may like to test that you can fire the appropriate function by sending a request to the resource associated with that function.
- To do this compose a request (using a command line tool such as curl or HTTPie), or better still by using the Requests python library in a client application.

```
import requests  
r = requests.get('https://api.github.com/events')
```



A python program to send a
GET request to a resource

HttpRequest Objects

- The request argument that is passed to a view function is an instance of an HttpRequest class. This object has a number of useful attributes that can be used to probe the different parts of a request:
- `HttpRequest.body`: The raw HTTP request body as a byte string.
- `HttpRequest.method`: A string representing the HTTP method used in the request. This is guaranteed to be uppercase. e.g.

```
if request.method == 'GET':  
    do_something()  
elif request.method == 'POST':  
    do_something_else()
```
- `HttpRequest.META`: A standard Python dictionary containing all available HTTP headers, e.g:

```
headers = request.META  
content_type = headers ['CONTENT_TYPE']
```
- `HttpRequest.COOKIES` : A standard Python dictionary containing all cookies in the. Keys and values are strings.
- Likewise the `HttpResponse` object has a number of attributes that can be set to return different response types.
- Full description of the `HttpRequest` and `HttpResponse` classes are available in the Django book.

Overriding the default CSRF protection in Django

- Sending a POST request to a view function will create an exception.
- This is because by default Django does not allow POST requests to be sent to view functions without a CSRF (cross site request forgery) token being attached to the request.
- CSRF is a malicious exploit of a website where unauthorized commands are transmitted from a user that the web application trusts.
- For the time being, we will override this feature by attaching the following python decorator before each view function that will receive a POST request:

```
@csrf_exempt  
def HandleRegisterRequest (request):  
    return HttpResponse ('not yet implemented')
```

That's it Folks



Further Reading

The Django book