Curtin University

Web Application Frameworks (COMP3011/COMP 6006)

Lecture 2: Django Models, Admin & Templates

Outline

- Revision
- Django Admin
- Django Models
- Django Templates

Assignment 1

- It will be released over the weekend. You will be able to ask questions in the next week's lab.
- Complete all the 'YOUR TURN' Part. It will be helpful for the assignment

Revision: Django and MVC

- Django web applications start with a Project that contains one or more Apps.
  - Each App consists of (data) Model definitions in `models.py`, View (Controller) definitions in `views.py` and Template (Views) in template files.
  - We must also define our routes using the `urls.py` files in the Project and App.
- Before we can do any of that, we must start a Project and then an App.
  - Of course, if we are building on our existing work (as we are about to do), we don’t need to start again – we can augment what we have already written.

YOUR TURN (5min): Defining the model (Part 1)

- Models encapsulate the organisation of data as well as the structure of the database.
  - Before coding the actual models, typically the first step to building a website in Django is to define how the data is organised and then use this work out how to build the models.
- Start by establishing what data you expect to be in the database and then use Django models to specify and organise the data.
  - Base this upon the description on the next slide.
  - Write this down either on paper or in a word processor document – identifying what the classes may be named, what fields it will have and of what types.

YOUR TURN (10min): Defining the model (Part 2)

- A typical blog post will contain the following information:
  - Post title – self-explanatory;
  - Description – the content of the blog;
  - Publication date – self-explanatory;
  - Tag – a keyword describing the post;
  - Slug – a URL component for the blog post.
- Separately, describe why you need the slug attribute above.
  - What is the main benefit of using slugs?
- Feel free to include other attributes within your model that you would like to store about a blog post.
  - With the rest of the model and the above question, write this down in a text document or on paper for reference.

YOUR TURN (10min): Creating Django models (Part 1)

- The goal of this exercise is to write Django code for the blog post model with its attributes defined above.
  - Django is designed such that it automatically handles communication with the database and even generates the database.
- To create a model, simply create a Python class named after the model through editing `models.py` in the **App**’s directory, building upon your work from last week’s workshop.

YOUR TURN (10min): Creating Django models (Part 2)

- In other words, within `blogapp/models.py`, change the file to show:

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=255)
    # Continue on with the other attributes...
```

- Complete the code above by adding the remaining attributes of a blog post.
  - Ensure that you choose appropriate `max_length` values for text fields and appropriate field types for each attribute.
  - Refer to the link in last week's workshop as to what field types can be chosen.
  - Replace the line starting with `#` (a comment that is not run) with your remaining attributes, one on each line and noting the indenting as usual.

YOUR TURN (5min): Custom model methods

- Custom methods on a model can be used to define custom functionality to the instances of that model.
  - For example, you can have a model method to check if a post is more than 2 years old, to concatenate (join) attributes together or to determine the length of an attribute, such as whether a slug is less than 64 characters.
- Create a method attached to your model class to determine if the post is pre-COVID (published on 10 Mar 2020 or earlier) or post-COVID.
  - For this, you will need to do some personal research. You may wish to check the Django documentation online at https://docs.djangoproject.com/en/4.2/topics/db/models/#model-methods for hints on how to do this.

YOUR TURN (5min): Migrations

- What is the purpose of migrations in Django? Answer this question in your text document or on paper.
- Next, create new migrations based on the model you have defined in the previous exercise, then apply the new migrations.
  - Afterwards, run the local development web server to make sure everything is working properly:

```
[user@pc]$ python3 manage.py runserver
```

- While you should see an error message within the Terminal if there is one, visit your website to confirm there are no errors.
  - Other than the Page Not Found error at `/`, which is still OK.

Django’s admin interface

- The purpose of the admin app is to provide you with an interface to *create*, *read*, *update* and *delete* (commonly known as CRUD) content.
    - Django automatically enables the admin app when we create our project.
- Recall from the last lecture that there was already a path in the project `urls.py` to allow us to access the admin interface.
    - To access it, run the development server and navigate to http://127.0.0.1:8000/admin/.

```
[user@pc]$ python3 manage.py runserver
```

Creating an admin user

- You will notice that to log in to and use the admin app, you will need a username and password.
  - This is good – as it means not just anyone can access it!
- To get a username and password, you can create a *superuser* from the command line, by issuing the command:

```
[user@pc]$ python3 manage.py createsuperuser
```

  - Follow the prompts to add in the username, email and password.
  - The email doesn’t really matter at this point!
- Once you have created a superuser, you can create further (regular or super) users from within the admin interface.

Admin considerations and concerns

- The admin interface is not something you want to have in a fully-functional, deployed app, used by many people – even if they are in different users groups.
    - It is quite general purpose and not specifically designed for your App – we will see this in the next section;
    - Your users may need to CRUD, but they would do it through your App proper – this would ensure constraints and permissions are applied;
    - Otherwise, we are really just using Django as a data management system, rather than building a web application;
    - Following this, the Django team recommends using it only for internal management.

Admin Models and Actions

- There are two main things required for use within admin to make it useful:
  - **Models** – as we know, this defines the data structures that we will use;
  - **Actions** – these allow us to manipulate data in instances of our Models.
- By adding additional functions to `admin.py` within our App, we can create functions that will CRUDs one (or more) of our Model instances.
- The admin site can also assist with logging – an important security concern.

What admin.py could look like (‘proper’ way – example with Article)

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

@admin.action(description = 'Mark selected stories as published')
def make_published(modeladmin, request, queryset):
    queryset.update(status = 'p' )

class ArticleAdmin(admin.ModelAdmin):
    list_display = ['title' , 'status']
    ordering = ['title' ]
    actions = [make_published]

admin.site.register(Article, ArticleAdmin)
```

YOUR TURN (5min): Practical pre-requisites

- In the previous tutorial, you created your first Django application.
- In this tutorial, you are going to define and implement the models for a blog and activate the Admin interface and then use the model content within a template you write.
- Before continuing, ensure you have completed last weeks’ work.

Adding models to the admin interface

- Django is not aware of the models within any of your apps until you `register` them with the admin app.
- Once you have done so, you can CRUD instances of your models in the database from the admin interface.
  - By default, the admin interface will only let you CRUD users.
- To do so, within the app's `admin.py` file, call the `admin.site.register` function as seen in the example on the next slide.

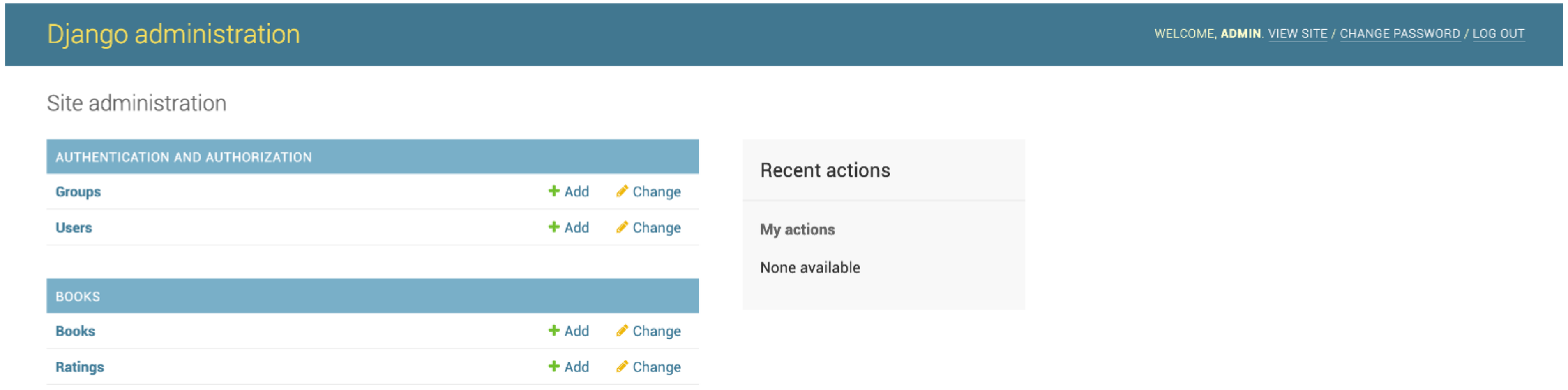
Adding Models to the admin interface – example (within `admin.py`)

```
from django.contrib import admin
from .models import Book, Rating


admin.site.register(Book)
admin.site.register(Rating)
```

- This is an alternate (and lazier) method to the example seen on previous slides – you will not get an interface that is as user-friendly as before.
- This example registers both the `Book` and `Rating` models, as seen in our previous examples (rather than the `Post` completed in the practical sections).

Admin interface – screenshot of homepage


Django administration

WELCOME, **ADMIN**. VIEW SITE / CHANGE PASSWORD / LOG OUT

Site administration

AUTHENTICATION AND AUTHORIZATION		
Groups	+ Add	Change
Users	+ Add	Change

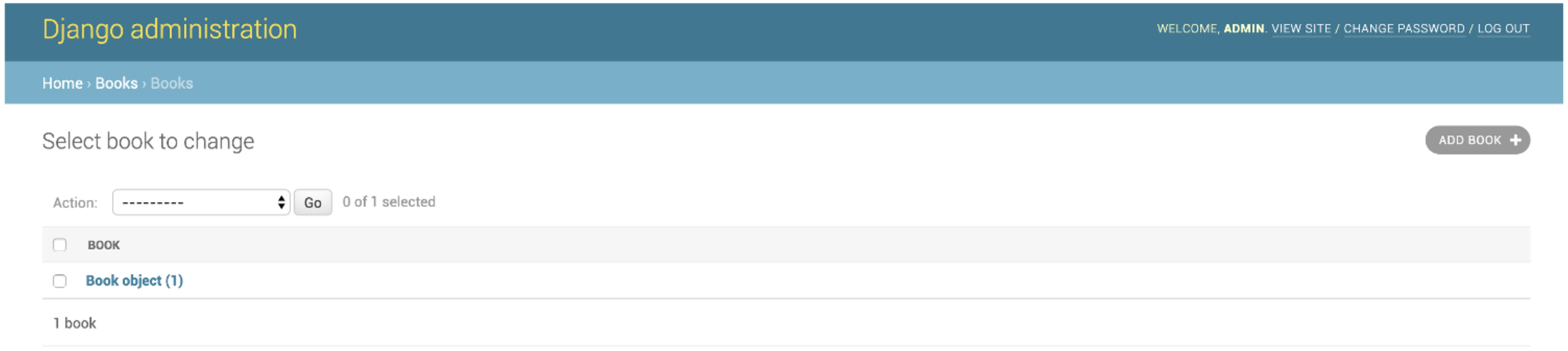
BOOKS		
Books	+ Add	Change
Ratings	+ Add	Change

Recent actions

My actions

None available

Admin interface – screenshot of `Book`s, the ‘lazy’ way



Select book to change

0 of 1 selected

BOOK
Book object (1)

1 book

YOUR TURN (5min): Django admin interface (Part 1)

- The purpose of Django’s admin interface is to provide a quick and user-friendly interface where trusted users can manage the contents of your site.
- To be able to create, read, update or delete the content of your website using the admin interface, you will need to register your App’s models with the admin App, so that it knows your models exist.
- Open the App’s `admin.py` file and add the following code:

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


admin.site.register(Post)
```

YOUR TURN (5min): Django admin interface (Part 2)

- Run the development server and navigate to http://127.0.0.1:8000/admin/.
  - You will be greeted by a login page.
- To create a superuser to login to the admin interface, run the following command in the terminal and follow the instructions on the screen:

```
[user@pc]$ python3 manage.py createsuperuser
```

  - You will have to stop the server first (`CTRL+C`) and then run the command, before restarting the server.
- Log in to the admin interface, to confirm your user has been created.

Templates

- One of the most powerful features of Django is that it allows us to return dynamic HTML content to the user.
    - That is, content from your database, rendered within HTML, or anything else that can be generated within Python.
- A common approach to achieve this functionality is through using Django’s template engine.
- A template contains the static parts of the desired HTML output, as well as some special syntax describing how dynamic content will be inserted and processed.
    - In the next few slides, you will learn how to return HTML content from your view (controller) functions and integrate it with the rest of your Project’s apps.

Templates – the Python side in `views.py`

- In the `views.py` file of your App, using our `Book`s example:

```
from .models import Book


def index(request):
    booksQuerySet = Book.objects.all()
    return render(request, 'index.html', {'books' : booksQuerySet})
```

- `render` takes three arguments:
  - The `request` object;
  - The path to the template – i.e. which template;
  - A dictionary used to pass data to our template.

Templates – what’s happening in Python?

- The first line `imports` the `Book` model from the `models` module (`.models`) in the current folder (i.e. the current App).
- Next, creates a variable named `booksQuerySet`, which will store an array of all `Book` objects within our database.
- Then, pass this to our template, which when we render it becomes what we will return/display to the user.
  - This itself is nested within a dictionary as required by the `render` function.
- If you get an error that `pylint` is required for `Book.objects.all()` or the equivalent for any other model class, install it.
  - To do this: `python3 -m pip install pylint-django`.

Templates – folder structure

- exampleproject: project container;
  - exampleapp: app container directory;
    - migrations: where database migrations are kept;
    - templates: create a directory where templates are to be kept;
      - index.html: create a template file, we will be using this as our base template;
    - __init__.py: unchanged as before.
    - admin.py
    - apps.py
    - models.py
    - tests.py
    - urls.py
    - views.py
  - exampleproject

Template – going in blind

Within the `index.html` file…

```
<table>
   <thead>
       <tr>
           <th>Book</th>
           <th>Author</th>
           <th>Genre</th>
       </tr>
   </thead>
   <tbody>
       {% for book in books %}
       <tr>
           <td>{{ book.title }}</td>
           <td>{{ book.author }}</td>
           <td>{{ book.rating.avgRating }}</td>
       </tr>
       {% endfor %}
    </tbody>
</table>
```

Django special template notation

- `{%` and `%}` are used for logical operations, e.g. `for`/`while` loops, `if` statements.
- `{{` and `}}` are used to render data values, i.e. ‘fill in the blanks’ from data within your system that is passed into the template by the view function (controller).

Template language notation – what does it mean?

- The section between the `{% for` to `endfor %}` functions the same as `for` loops generally do within programming languages.
  - This is despite strictly being in the presentation layer.
  - `book in books` will iterate over a list of books (i.e. all of the books within the database) and allow you to define the display of an individual book using the variable `book`.
  - As seen in the example, each attribute of the Book is output, using ‘dot notation’ to access each attribute based upon how it was defined in the model.
- `if` statements and `while` loops are effected in a similar manner.

‘Double dot’ notation

- When there is a (strictly) `ForeignKey` relationship between two models, the attribute of the foreign model can be accessed through the foreign key.
  - `book.rating` – access the `Rating` associated with the `Book`.
  - `book.rating.avgRating` – access the `Rating`’s `avgRating`.

Extending a template

- Templates are useful for when you want to reuse components of your layout in more than one place – i.e. on more than one page or function.
- Template extending allows you to use the same parts of your HTML for different pages of your website.
- Usually, a base template (`base.html`) is the most basic template that you extend on every page of your website and provides the 'core' of the layout.
  - For example, if you are using a front-end framework like Bootstrap, the template goes in your `base.html` file, in the same folder as `index.html`.

Writing a base template – with a `block`

Within the `base.html` file…

```
<html>
    <head>
        <title>COMP6006</title>
        <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
        <link  rel="stylesheet"  href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">
    </head>
    <body>
{% block content %}
{% endblock %}
    </body>
</html>
```

Writing a base template – with a `block`

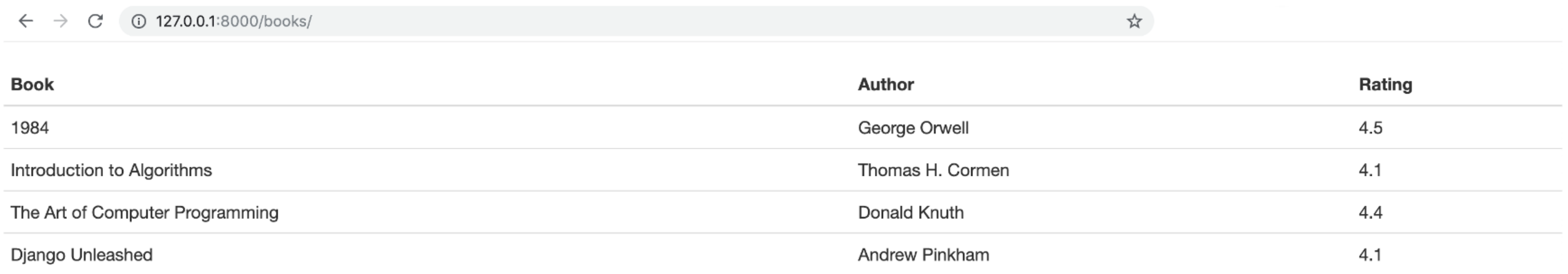
- `block` is used for 'overriding' specific parts of a template.
- In the example of the previous page, a block named `content` is supposed to be overridden by children that inherit from this template.
  - In practice, this means that the `{% block` to `endblock %}` section will be replaced by the content of another file.
  - That file contains the content that should appear here, defined using the same template tags and a reference to the `base.html` file.

Extending a template – making a sub-template with logic

Within the `index.html` file...

```
{% extends 'base.html' %}
{% block content %}
<table>
    <tr>
        <th>Book</th>
        <th>Author</th>
        <th>Genre</th>
    </tr>
    {% for book in books %}
    <tr>
        <td>{{ book.title }}</td>
        <td>{{ book.author }}</td>
        <td>{{ book.rating.avgRating }}</td>
    </tr>
    {% endfor %}
</table>
{% endblock %}
```

Templates – example of showing multiple `Book` instances



127.0.0.1:8000/books/

Book	Author	Rating
1984	George Orwell	4.5
Introduction to Algorithms	Thomas H. Cormen	4.1
The Art of Computer Programming	Donald Knuth	4.4
Django Unleashed	Andrew Pinkham	4.1

YOUR TURN (10min): Django Templates

- One of the most powerful features of the Django web framework is that it allows a way to generate HTML dynamically using templates.
- In your text file, describe how templates in Django work.
- Next, design a mock-up of how you want your blog posts to look.
  - You are welcome to use any style you like with whatever software you like; ensure that the attributes of your `Post` model are shown as they would be displayed in a real web page.

YOUR TURN (5min): Creating a template (Part 1)

- In your App directory, create a new folder called `templates`.
  - You will be storing your templates for your posts in this directory.
- Create an `index.html` file and save it in the `templates` directory.
  - It can remain blank for now.
- Open the `views.py` file of your app and modify it as such:

```
from django.shortcuts import render
from .models import Post


def index(request):
    postsQuerySet = Post.objects.all()
    return render(request, 'index.html' , {'posts'  : postsQuerySet})
```

YOUR TURN (5min): Creating a template (Part 2)

- Then, describe the `.objects.all()` function within your text file.
- Note that the `render` method defined above takes three arguments:
  - `request` – the request object used to generate this response,
  - `'index.html'` – the path to the template to use, and
  - `{'posts': posts}` – a dictionary of values to use within the template context.
- Open the `index.html` file from your template folder and write the HTML for how you want your blog posts to look.
  - Design this per your mock-ups in previous exercises.
  - Use the code sample at https://gist.github.com/cmalven/1885287 to ensure that you have a valid HTML page and/or the links on Blackboard.
  - Do not worry about the HTML so far; your content will go between the `<body>` and `</body>` tags and a descriptive title should be provided within the `<title>` and `</title>` tags.

YOUR TURN (5min): Creating a template (Part 3)

- Use the following Django special notations to render dynamic content in your HTML file:
  - `{{ variable }}` – when the template engine encounters a variable, it evaluates the variable and replaces it with the result. For example, `{{ posts.title }}` will be replaced with the value of the `title` attribute of the `posts` object.
  - `{% tag %}` – tags control the logic of the template. For example, `{% for post in posts %}` and `{% endfor %}` provide for a block between them that will function as the content of a `for` loop.

YOUR TURN (5min): Test your Django site

- Run the local development web server using the usual command:

```
[user@pc]$ python3 manage.py runserver
```

- Using the admin interface, manually add a few posts in the database.
- Navigate to `/blog/` to make sure your posts are being rendered correctly.
    - If not, correct your errors and try again.

Summary

- To create a superuser, issue the command:

```
[user@pc]$ python3 manage.py createsuperuser
```

- To access the admin interface, navigate to http://127.0.0.1:8000/admin/.
- Django template notations are based on the following:
  - `{%` and `%}` are used for logical operations, e.g. `for`/`while` loops, `if` statements.
  - `{{` and `}}` are used to render data values, i.e. ‘fill in the blanks’ from data within your system that is passed into the template by the view function (controller).