Web Application Frameworks (COMP6006/3011)

Lecture 3: Advanced Django Topics

Dr. Sheik Mohammad Mostakim Fattah
Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

COMP6006 - Unit Learning Outcomes

- Evaluate and argue for architectural design approaches for developing web applications in terms of security, usability, performance, and other properties;
- Create sophisticated client-side web applications based on modern frameworks and tool sets.
- Create server-side back-ends to web applications to support complex functionality and secure data management;
- Assess the workability, interoperability and quality of client-side and server-side aspects of web applications.

Outline

Teaching Staffs: Monday 8 am – 11 am

- Dr. Muhammad Hasan, Session: 114 + online, hasan.a.hasan@curtin.edu.au
- Parav Pathak, Session: 116, parav.pathak@curtin.edu.au
- Hilal Ibrahim Suleman Alwaneh, Session: 116, h.alawneh@postgrad.curtin.edu.au
- Jinguo XU (David), Session: 117, david.xu1@curtin.edu.au

Teaching Staffs: Monday 4 pm – 7 pm

- Hilal Ibrahim Suleman Alwaneh, Session: 220, h.alawneh@postgrad.curtin.edu.au
- Jordan Richards, Session: 232, jordan.richards@curtin.edu.au
- Dr. Nadith Pathirage, Session: 219, nadith.pathirage@curtin.edu.au

Revision: Django so far!

- Only the headers are listed here, as we'll discuss our knowledge of each of the below topics in class:
  - Projects and Apps;
  - Models and Migrations;
  - Views and Routes;
  - Templates and Rendering.
- Have we forgotten anything (that we've covered so far)?

'The Rest' of Django (but not really)

- So far, we've covered the architecture of a 'traditional' web application and how we can build it using Django.
  - Our main concerns are regarding routing of user requests, storage of data, as well as building our application using an MVC architecture.
- However, contributing to these (and extending from them) are many general concerns that may solve particular issues we have when building a web application.
  - We won't (and don't have time) to cover everything – the scope is enormous – but more detail is available in the 'Resources' on Blackboard.

YOUR TURN (5min): Practical pre-requisites and introduction

- In the previous tutorial, you created the models for your blog and activated the admin interface.
- In this tutorial, you are going to create a separate view for each blog post that can be accessed through a unique URL.
- Ensure that you have completed Workshop 2 before continuing.

What are URL Parameters?

- The `urls.py` file handles mapping between *functions* and URL requests (generally via Apps).
  - For example, if an app is called `books`, your project `urls.py` file passes all requests that begin with `books/` to the App `urls.py`.
- URL parameters are a way to pass (more) information about an interaction through a URL.
  - For example, with the URL `https://www.djangoproject.com/start/`, the portion `/start/` is a parameter.

URL Parameters - URL's File

The green text is enclosed with ''

- Update your App `urls.py` file:

```
urlpatterns = [
    path( , views.index),
    path('<book_id>' , views.detail, name = 'books_detail' )
]
```

- Recall our three arguments from last week – how have they changed?
  - `<book_id>` is the parameter that maps to the `detail` function in `views.py`.
  - The empty path means that `books/` will still go to the `index` function (but not `books` – an important distinction).

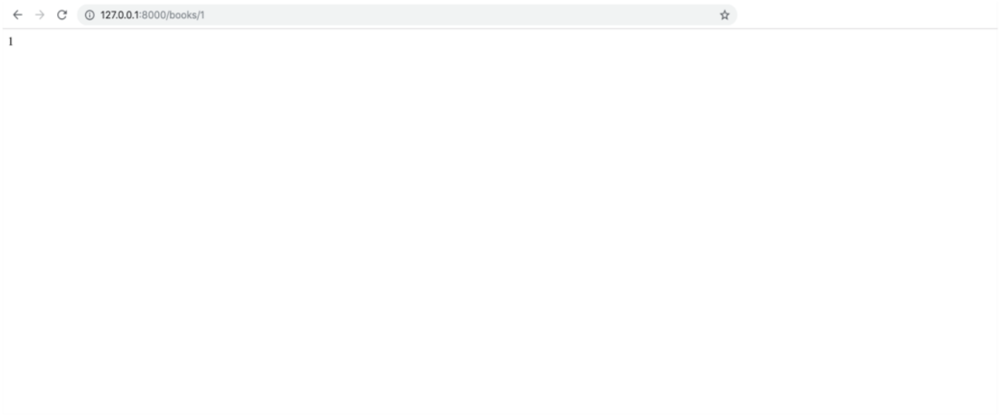
Mapping a URL to a View

- Update your App's `views.py` as such:

```
def detail(request, book_id):
    return HttpResponse(book_id) # For now, this just prints the parameter.
```

- When a user enters `localhost:8000/books/2` into the browser, Django automatically extracts the '2' and passes it to the `detail` function as the parameter (and hence variable) `book_id`.

Mapping a URL to a View – Result



```
1
```

URL Parameters – Type Converter

- To ensure that the parameter is an `int`, type conversion can be used to check for a parameter of type `int`:

```
urlpatterns = [
    path( , views.index),
    path('<int:book_id>' , views.details, name = 'books_detail' )
]
```

- What happens if the supplied parameter is **not** an `int`?

YOUR TURN (5min): Define a new URL path (Part 1)

- Imagine that a user types https://localhost:8000/blog/1 in the browser. You would want to navigate the user to the first blog post on your site.
  - In other words, the user should be navigated to the blog post with the `id=1`.
- The first step in implementing this functionality is to define the path in the `urls.py` file to tell Django how to handle this request.
  - Open your App's `urls.py` file of your app and add the following line starting with `path` to your `urlpatterns` list, noting the comma on the line before:

```
urlpatterns = [
    ...,
    path('<int:post_id>' , views.post_detail)
]
```

YOUR TURN (5min): Define a new URL path (Part 2)

- Note that the first parameter of the path function, `<int: post_id>`, captures the post id from the URL. In general, at least in Django, angle brackets are used to capture a value from the URL.
    - The '`int:`' in '`<int:post_id>`' is called a converter type and is used to capture an integer parameter.
    - The second parameter, `views.post_detail`, is the view function that you will define in the next step.
    - For example, a request to `/blog/10` would match the path above in the `urlpatterns` list and Django would call the function `views.post_detail(request, post_id)` where `post_id` would have a value of `10`.

Getting the object of a URL parameter – `views.py` file

- Update your App's `views.py` file as such:

```
def details(request, book_id):
    book = Book.objects.get(id = book_id)
    return render(request, 'detail.html' , {'book' : book})
```

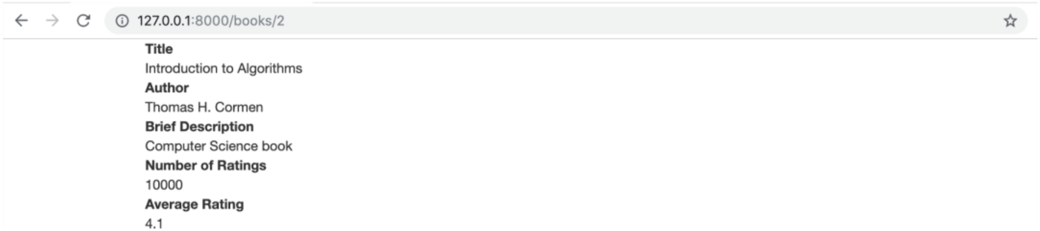
- In this case, `render` renders the '`details.html`' page and passes the specific `book` as the `context_instance` (in the usual form of a dictionary).

Getting the object of a URL parameter – template

- Create a new template file called '`details.html`' in the `templates` directory (note the template tags):

```
{% extends base.html %}
{% block content %}
<div class="container">
<dl>
    <dt>Title</dt>
    <dd>{{ book.title }}</dd>
    <dt>Author</dt>
    <dd>{{ book.author }}</dd>
    <dt>Brief Description</dt>
    <dd>{{ book.description }}</dd>
    <dt>Number of Ratings</dt>
    <dd>{{ book.rating.numReviews }}</dd>
    <dt>Average Rating</dt>
    <dd>{{ book.rating.avgRating }}</dd>
</dl>
</div>
{% endblock %}
```

Getting the object of a URL parameter – Result


127.0.0.1:8000/books/2

Title
Introduction to Algorithms
Author
Thomas H. Cormen
Brief Description
Computer Science book
Number of Ratings
10000
Average Rating
4.1

The other way – POST

- So far what we have explored is termed GET – using URL parameters.
- Forms can be used to generate interfaces to supply data to a view function.
  - A simple example – user registration where a username and password is supplied.
- The `action` of a HTML `<form>` is set to a URL defined in `urls.py`.
  - Each form element (text box, drop down) has a `name` which is used by Django to identify it.
- Within the view function, `request.POST` is used to extract the POSTed parameters by name.
  - POST has some additional security features baked in (to Django) which we will cover later today.

YOUR TURN (5min): Implementing the `post_detail` view (Part 1)

- Previously, you defined a path that calls the `view.post_detail` function when an integer parameter is found the URL (e.g. `.../blog/7`).
- To implement the `view.post_detail` function, open the `views.py` file and define a new function:

```
def post_detail(request, post_id):
```

YOUR TURN (5min): Implementing the `post_detail` view (Part 2)

- To get the post with the correct `post_id`, the `objects.get()` QuerySet can be used to begin to complete the function.
  - The `.get()` function returns the object matching the given lookup parameter.
- To get the correct post:

```
post = Post.objects.get(id = post_id)
```

- Finally, you need to call the `render` function to render the template `details.html` with the `post` object added to the template context.
  - Write the code to do this. We will make the file itself later on.

Putting it all Together

- We now have the ability to return the data of a single post.
- We also know how to write our templates (MVC Views), views (MVC Controllers) and models.
- Let's put it all together to create the (basis of) a blog!

YOUR TURN (10min): Creating a new template

- Create a new file called `details.html` in the `templates` directory.
- Open `details.html` and write the HTML to display the post object.
  - You can use the template you created in Week 2, but only for a single post.
  - Look at the example above for a `Book` in this lecture/workshop.

YOUR TURN (5min): Test your blog

- If you do not have any posts in the database, use the admin interface to manually create five posts.
- Run the local development web server and navigate to `/blog/1` through `/blog/5` to make sure each post are being rendered correctly.

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

YOUR TURN (5min): Template Extending (Part 1)

- Template extending allows you to use the same parts of your HTML for different pages of your website.
  - For example, you can code the navigation bar of your website in a base template (`base.html`) and extend this on every page of your website.
- In this exercise, you will create a `base.html` template that will contain the basis of a webpage and a heading.
  - This is so that you do not have to implement the heading on every page of your site, each time you want the same thing.
  - Other templates, such as `index.html` (posts page) and `details.html`, will simply inherit from this template.

YOUR TURN (5min): Template Extending (Part 2)

- Create a new file called `base.html` in the `templates` directory.
- Open the `base.html` file and add the following code:

```
{% load static %}
<html>
<head>
<title>Your title</title>
</head>
<body>
<h1>My Blog</h1>
{% block content %}
{% endblock %}
</body>
</html>
```

YOUR TURN (5min): Template Extending (Part 3)

- Note the `{% block %}` template tag can be used to make an area that will have HTML inserted in it and that HTML will come from a child template that extends this template.
  - Go to your `index.html` and `details.html` and extend the base template using the `extends` tag.
  - Further, add `{% block content %}` and `{% endblock %}` to your `index.html` and `details.html` files to define which code will be used to fill in the blocks defined in the base template.
  - Your block tag in the child template (i.e. the two we are extending) must match the `block` tag in your `base.html` file.
- Save the file and check that your website is still working properly on the local development web server.

What is a 404 Error?

- A 404 error is an HTTP status code that means that the page you are trying to reach couldn't be found.
- Django has a built-in function, `get_object_or_404` that handles 404 errors for us when retrieving data.

Raising a 404 Error – Code

- In your App's `views.py` file, you will need to import `get_object_or_404`:

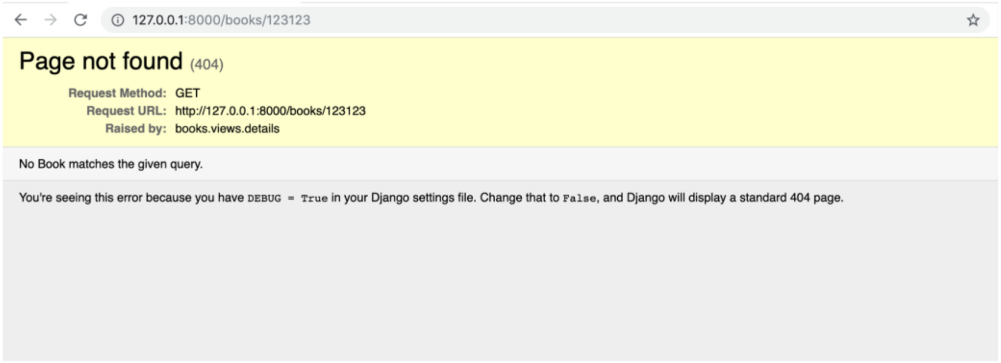
```
from django.shortcuts import render, get_object_or_404
```

- Then, update the `detail` function in `views.py`:

```
def detail(request, book_id):
    book = get_object_or_404(Book, id = book_id)
    return render(request, 'books.detail.html' , {'book' : book})
```

- The function takes two arguments – the `model_class` and the `id`.

Raising a 404 Error – Result


127.0.0.1:8000/books/123123

Page not found (404)

Request Method: GET
Request URL: http://127.0.0.1:8000/books/123123
Raised by: books.views.details

No Book matches the given query.

You're seeing this error because you have `DEBUG = True` in your Django settings file. Change that to `False`, and Django will display a standard 404 page.

Other Errors

- Note that the error page only comes up because we are on `DEBUG` mode.
  - We could make a 'pretty' one for a real project or just return the error code.
- Occasionally, you will run in to other errors as well:
  - If your code breaks, you will get a `500 Internal Server Error` (or errors with numbers similarly within the `5xx` range).
  - Other errors such as `400 Bad Request` or `403 Forbidden` can also be raised and handled by Django.

Global Error Handling in Django

- Within our `urls.py` file of our Project, we can specify values for `handler400`, `handler403`, `handler404` and `handler500` to return a particular view function as a response if these errors occur anywhere within our web application.

```
handler404 =  myapp.views.error_404
handler500 =  myapp.views.error_500
handler403 =  myapp.views.error_403
handler400 = myapp.views.error_400
```

- Much like `urlpatterns`, these `handler` variables have a special meaning, and the path on the right must refer to a view function.
  - We must also set `DEBUG = False` in `settings.py`!

The controller (`views.py`) functions for Error Handling

- We must create the functions referred to on the previous slide (of course, with our App's name, rather than `myapp`). These are relatively similar to what we have seen before:

```
def error_404(request, exception):
    return HttpResponse('Cannot find the content!' , status = 404)
```

- We can use the `exception` parameter in some cases to get more information about the particular error (but must always specify it).
  - Except for `500` errors – they should just have the `request`!
- The `status = 404` parameter sets the status of the `HttpResponse`.
  - A `render`er would be better here, for usability.

Considerations of `DEBUG = False`

- Setting the debugging flag to `False` will stop the informative error messages appearing when you `runserver`.
    - Of course, when actually deploying, this is a very good idea!
    - For testing however, it is generally more helpful to have this set to `True`, unless (like now) you need to test it with it being `False`.
- Once set to false, you will have to set the `ALLOWED_HOSTS` value to `['*']`.
    - This allows us to create a basic 'firewall' to restrict requests to particular hosts (computers). For testing, we can limit it to anything (represented by the 'star').

YOUR TURN (5min): Raising a 404 Error

- Modify your `post_detail` function, such that it returns a 404 error if the requested post cannot be found.
  - This should just be a one-line change!

YOUR TURN (5min): Catering for other Errors

- Modify your Project's `urls.py` file to introduce handlers for errors with codes `400`, `403`, `404` and `500`.
    - Create the required functions referred to in the `urls.py` file within your `views.py` file.
    - Don't forget the required changes in `settings.py`!

What are API's?

- Application Programming Interfaces (API's) allows other computer systems – such as websites or mobile applications – to work with your application's data.
  - Think of it as how a computer would want to see your webpage/webapp.
- A RESTful API is one that uses HTTP requests to GET, PUT, POST and DELETE data.
  - Technically via these HTTP request types but often not in practice.
- Tastypie is a web service API framework for Django.
  - Tastypie makes exposing your models easy, but gives you full control over what you expose, letting you abstract away the model as much as needed.

Installing TastyPie

- Install the `django-tastypie` package using `pip`:

```
[user@pc]$ python3 -m pip install django-tastypie
```

- Create a new App in your project named `api`.

```
[user@pc]$ python3 manage.py startapp api
```

- Add this new App to the list of `INSTALLED_APPS` in `settings.py`:

```
'api.apps.ApiConfig'
```

Building API's – Models

- Models in RESTful API's are called *resources*. All API's must have a resource (i.e. a model).
- Go to `models.py` in your `api` App and create a class named `BookResource`:

```
from django.db import models # Don t really need it!
from tastypie.resources import ModelResource
from books.models import Book

class BookResource(ModelResource):
    class Meta:
        queryset = Book.objects.all()  # This returns a query, not an object.
        resource_name =  'books'
```

Building API's – Models

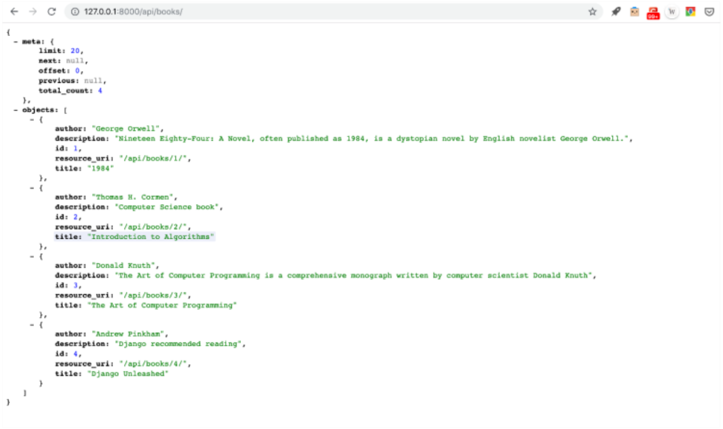
- `books` is the API URL parameter (i.e. `localhost:8000/api/books`).
- We couple together interfaces for all resources in one API, hence the use of a new App that imports from `books`.

Building API's – URL's

- Update your project `urls.py` accordingly to generate the URL endpoints for the API, using our `Book`s example:

```
from api.models import BookResource
book_resource = BookResource()
urlpatterns = [
    path('admin/', admin.site.urls),
    path('books/', include('books.urls')),
    path('api/', include(book_resource.urls))
]
```

Building API's – Result



127.0.0.1:8000/api/books/

```
{
 - meta: {
       limit: 20,
       next: null,
       offset: 0,
       previous: null,
       total_count: 4
   },
 - objects: [
   - {
         author: "George Orwell",
         description: "Nineteen Eighty-Four: A Novel, often published as 1984, is a dystopian novel by English novelist George Orwell.",
         id: 1,
         resource_uri: "/api/books/1/",
         title: "1984"
     },
   - {
         author: "Thomas H. Cormen",
         description: "Computer Science book",
         id: 2,
         resource_uri: "/api/books/2/",
         title: "Introduction to Algorithms"
     },
   - {
         author: "Donald Knuth",
         description: "The Art of Computer Programming is a comprehensive monograph written by computer scientist Donald Knuth",
         id: 3,
         resource_uri: "/api/books/3/",
         title: "The Art of Computer Programming"
     },
   - {
         author: "Andrew Pinkham",
         description: "Django recommended reading",
         id: 4,
         resource_uri: "/api/books/4/",
         title: "Django Unleashed"
     }
   ]
}
```

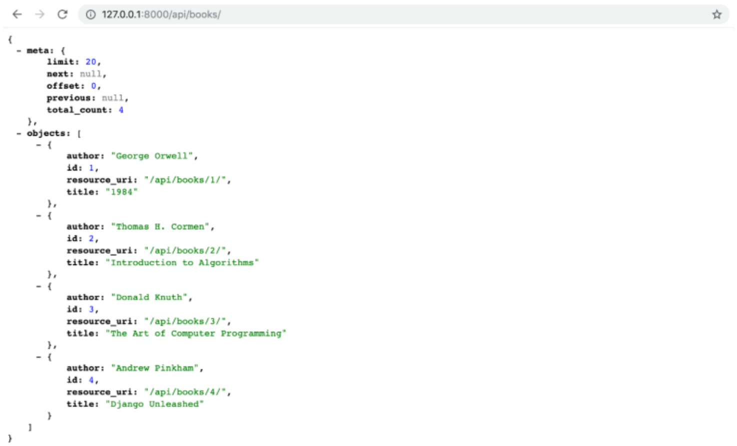
Modifying API's

- To control what data gets exposed through the API, in the `api` App's `models.py`, add:

```
excludes = ['date_created' ]
```

- `excludes` is a list the specifies the attributes that will be excluded from the API JSON output.

Modifying API's – Result



127.0.0.1:8000/api/books/

```
{
  - meta: {
        limit: 20,
        next: null,
        offset: 0,
        previous: null,
        total_count: 4
    },
  - objects: [
      - {
            author: "George Orwell",
            id: 1,
            resource_uri: "/api/books/1/",
            title: "1984"
        },
      - {
            author: "Thomas H. Cormen",
            id: 2,
            resource_uri: "/api/books/2/",
            title: "Introduction to Algorithms"
        },
      - {
            author: "Donald Knuth",
            id: 3,
            resource_uri: "/api/books/3/",
            title: "The Art of Computer Programming"
        },
      - {
            author: "Andrew Pinkham",
            id: 4,
            resource_uri: "/api/books/4/",
            title: "Django Unleashed"
        }
    ]
}
```

Modifying API's – Further Work

- So far, we only return a list (GET) of all `Book`s.
  - Generally, our API would also have endpoints to retrieve a single resource, create a new resource, update a resource and delete a resource.
- We won't actually practice this on our own project – but we will be doing this when we look at server-side considerations with Node (i.e. Express.js).

Design Considerations

- What if we provided the ability to edit data via our web app?
  - How could we ensure only the right people can edit the right things?
  - What about user privacy in general – you could see everyone's data!
- We must consider who can do what when we design our web app.
  - This is why we have our authentication system – to ensure that only certain users have certain permissions.

CSRF Tokenisation

- What if someone tried to fake your authentication credentials?
- A common attack involves injecting code to run on the users' system to submit a different request.
- This can be fought with a cross-site request forgery (CSRF) token using a `{% csrf_token %}` tag within a form within a Django template.
- The Django middleware can look for this per-request token when a form is submitted and if it does not match, the request will not succeed.
- Tokens can also be passed via GET – but in both cases they are only needed if we are modifying data or viewing restricted data.

Summary

- (Dynamic) URL parameters can be passed via `<parameter>` in the `urls.py` file. This allows the value within the URL to be changed and read by Django.
- We can use the `get_object_or_404` function to return an error page if we try and retrieve a non-existent model instance.
- Tastypie allows us to create a machine-readable version of our model (an API) within Django.
- We must consider whether everyone should have access to everything and protect against attack when this is not the case.

The First Assessment

- Building a simple Django app.
  - Take home format, two weeks to do it.
  - Just to get an idea of how you are tracking.
- Submission Deadline: 11.59 PM, 24th March 2024

Q&A and What's Next

- That's the end of our Django section of the course.
  - Next week, we look at user interface development with Bootstrap.
- Now is a good time to ask any Django-related questions and/or finish off practical sections!
  - Of course, I am still happy to answer questions later!