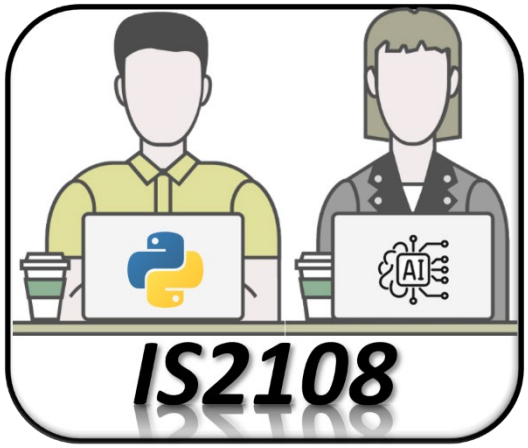


Lecture 7

Web Application Development with Django (III)

IS2108 – Full-stack Software Engineering for AI Solutions I
AY 2025/26 Semester 1

Lecturer: A/P TAN Wee Kek

Email: distwk@nus.edu.sg :: **Tel**: 6516 6731 :: **Office**: COM3-02-35

Consultation: Tuesday, 12 pm to 2 pm. Additional consultations by appointment are welcome.

Learning Objectives

- At the end of this lecture, you should understand:
  - ORM in Django (cont.):
    - Model Inheritance.
  - Advanced concepts on web application development with Django:
    - Advanced form processing with Form class and ModelForm class.
    - Handling HTTP requests with class-based views.
    - Conversational state handling with Django session framework.
    - Working with Django middlewares.

Readings

- Required readings:
  - None.
- Suggested readings:
  - None

ORM in Django (cont.)

lecture07

Model Inheritance

- Recall that model inheritance in Django works almost identically as Python class inheritance:
  - However, the base class should still inherit from `django.db.models.Model`.
- However, there is an important design decision to make about the parent models:
  - Whether the parent models are to be models in their own right with their own database tables); or
  - Just holders of common information that will only be visible through the child models.
- Consequently, there are three possible styles of inheritance in Django.

Model Inheritance Styles

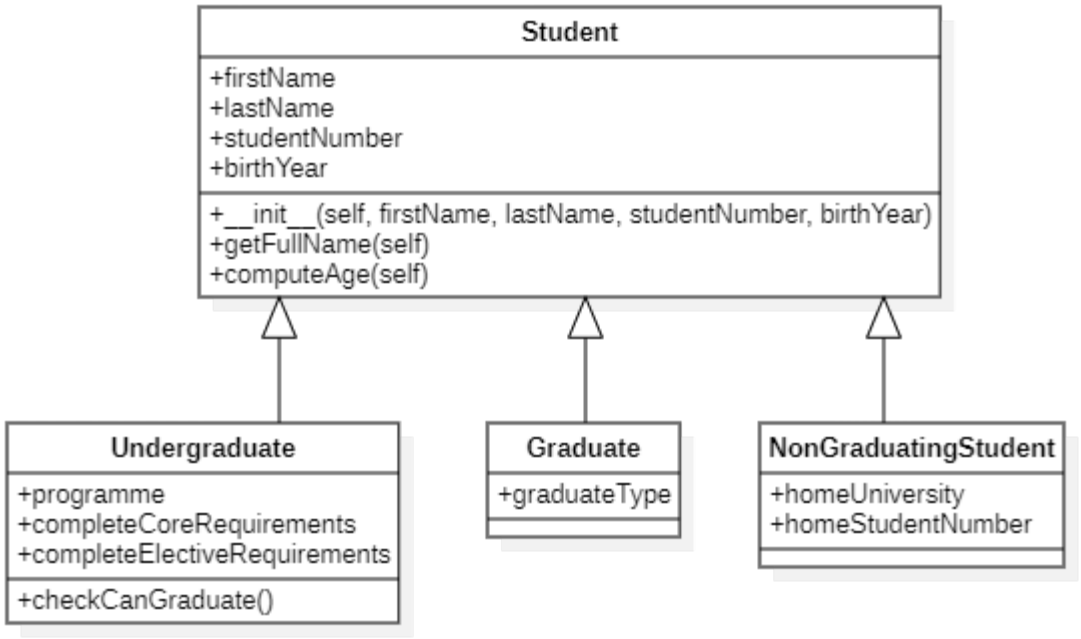
- **Abstract base classes**:
  - Use the parent class to hold common information.
  - This class will not be used in isolation.
  - Implementation:
    - Create base class and set `abstract=True` in the `Meta` class.
    - Base model will then not be used to create any database table.
    - Used only as a base class for other models.
    - Its fields will be added to those of the child class.
- **Multi-table inheritance**:
  - Subclassing an existing model and each model will have its own database table.
  - Each model in the hierarchy can be queried and created individually.

Model Inheritance Styles (cont.)

- Implementation:
  - The inheritance relationship is implemented as links between the child model and each of its parents.
  - The inheritance relationship is automatically-created as an `OneToOneField`.
- **Proxy models**:
  - Modify the Python-level behavior of a model, without changing the models fields:
    - In multi-table inheritance, a new database table is created for each subclass of a model.
    - Sometime, we might just want to change the Python behavior of a model, e.g., to change the default manager or add a new method.
  - This style of inheritance essentially creates a proxy for the original model.

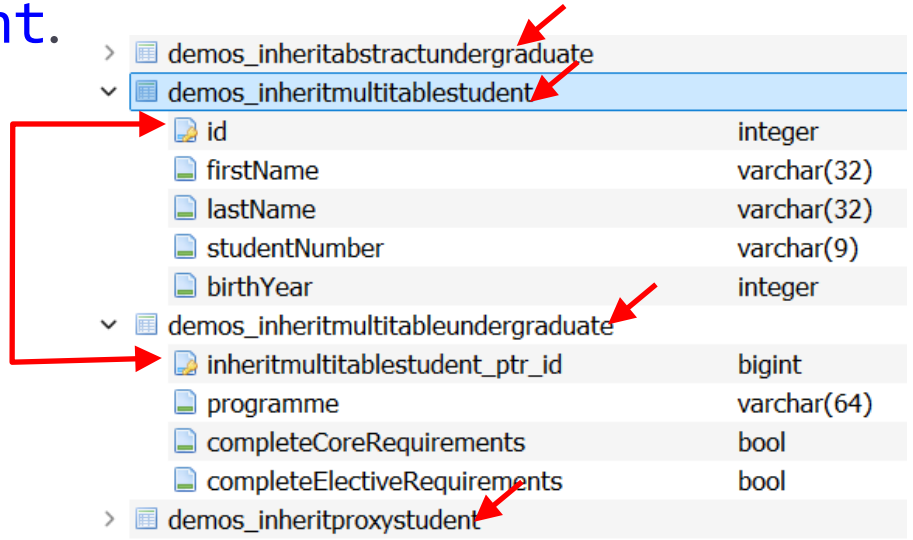
Model Inheritance Styles (cont.)

- Implementation:
  - Proxy models are declared similar to normal models.
  - But in child class, set `proxy=True` in the `Meta` class.
- See `demo` for examples of `Student` and `Undergraduate`:



Model Inheritance Styles (cont.)

- Important points to observe:
  - There are only four tables created.
  - There is no table for `InheritAbstractStudent` and `InheritProxyUndergraduate`.
  - There is a foreign key in `InheritMultiTableUndergraduate` referencing `InheritMultiTableStudent`.



Model Inheritance Styles (cont.)

- In summary:

Factor	Abstract Base Class	Multi-table Inheritance	Proxy Models
Child class has unique attributes	Yes	Yes	No
Child class has unique methods	No	No	Yes
Child class has separate table	N/A	Yes	No
Parent class can be instantiated	No	Yes	Yes

- See `demo01` for examples on creating new instances of the various models.

Querying Model Inheritance

- Model inheritance can be queried using the same techniques that have been discussed in the previous lecture.
- However, there are some important points to note:
  - Abstract base case cannot be queried.
  - Querying multi-table base class will retrieve all instances of the various child classes.
  - Querying multi-table child class will only retrieve instances of that child class.
  - Querying proxy parent class and proxy child class will retrieve all instances regardless.
- See `demo02` for examples on querying model instances.

Advanced Concepts of Django Development

src

Recap on Form Processing

- Recall that in an earlier lecture, we have discussed HTML form processing:
  - The most basic approach is to parse the form data manually using `request.POST`.
- This basic approach requires the developer to perform many tasks manually:
  - Create the form elements in the template.
  - Copy the form data into Python variables.
  - Validate the data.
  - Restore the data in the template for editing.
- Django supports the use of Form class to automate these tasks.

Django Form Class

- Recall that a Django **Model** class describes the logical structure of an object, its behavior and the way its parts are represented.
- Similarly, a Dango **Form** class describes a form and determines how it works and appears:
  - A model class's fields map to database fields.
  - A form class's fields map to HTML form `<input>` elements.
  - A ModelForm maps a model class's fields to HTML form `<input>` elements via a Form.

More About Form Fields

- Form fields:
  - Form fields are themselves classes.
  - They manage form data and perform validation when a form is submitted.
  - E.g., `CharField` and `EmailField`.
  - A `DateField` and a `FileField` handle very different kinds of data and perform different things with it.
- A form field is represented to a user in the browser as an HTML widget:
  - Widget is a piece of user interface machinery.
  - Each field type has an appropriate default `Widget` class, but these can be overridden as required.

Instantiating, Processing, and Rendering Forms

- General process of rendering an object in Django:
  - Get hold of the object in the view (e.g., retrieve from database).
  - Pass it to template context.
  - Expand it to HTML markup using template variables.
- Rendering a form in a template involves a similar process but with some key differences:
  - The form can be left empty or prepopulate with some data.
  - Form data can come from:
    - Data from a saved model instance.
    - Data collected from other sources.
    - Data received from a previous HTML form submission.

Building a Form in Django

- Define a `Form` class:
  - Define the required form fields.
  - For each form field, use the most appropriate form field class:
    - Basic classes include `CharField`, `EmailField`, `IntegerField`, `FloatField`, `BooleanField`, `ChoiceField`, etc.
    - Advanced classes include `FileField` and `ImageField`.
  - Define additional field options such as `label` and `max_length`.
  - Observe that the process is similar to defining a `Model` class.

Building a Form in Django (cont.)

- A `Form` instance has an `is_valid()` method:
  - This method runs validation routines for fields in the form.
  - When this method is called, if all fields contain valid data, it will:
    - return `True`.
    - place the form's data in its `cleaned_data` attribute.
- Form rendering:
  - The form is rendered into a collection of `<label>` and `<input>` elements.
  - The `<form>` tag and submit button are not rendered.
  - These have to be added in the template.

Processing a Form in Django

- Form data sent back are processed by a view:
  - This is usually the same view which rendered the form initially.
  - This approach allows the reuse of some logic.
- To handle the form, it is instantiated in the view for the URL where it is to be rendered:
  - **Initial GET request**:
    - Create an empty form instance and place it in the template context to be rendered.
  - **Subsequent POST request**:
    - The view will once again create a form instance and populate it with data from the request.
    - This process is known as binding data to the form and the form is now a bound form.

Processing a Form in Django (cont.)

- The form data can be validated by calling the `is_valid()` method:
  - If it does not return `True`, the template is rendered again:
    - But the form is no longer empty and so the HTML form will be populated with the data previously submitted.
    - Makes it easier for editing.
  - If it returns `True`, the validated form data will be stored in its `cleaned_data` attribute.

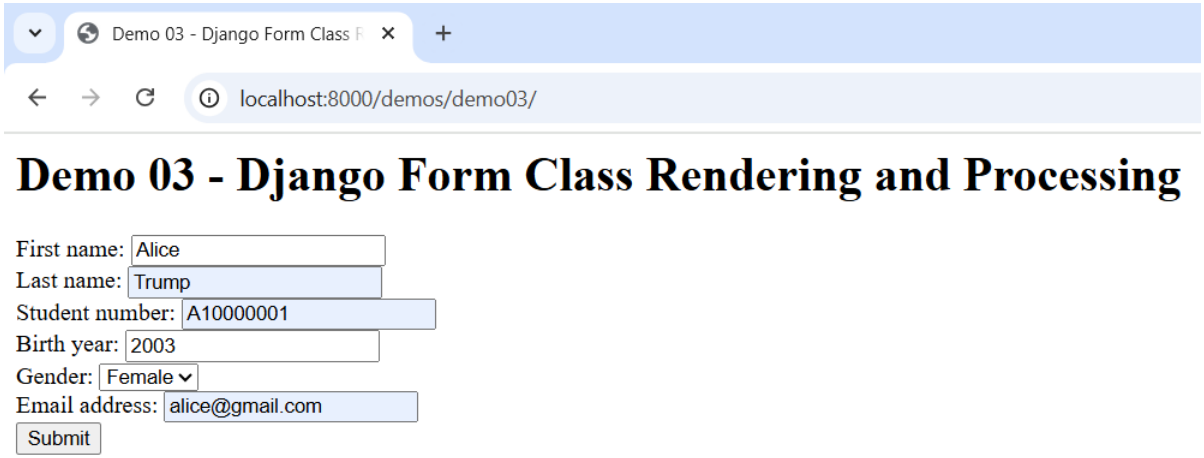
Example of Django Form Class

- In this example, a simple HTML form for inputting a Student record is created and processed using Django Form class.
  - Refer to the app-level `forms.py` for the actual Form class `StudentForm`.
  - See `demo03` for the example on processing `StudentForm`.

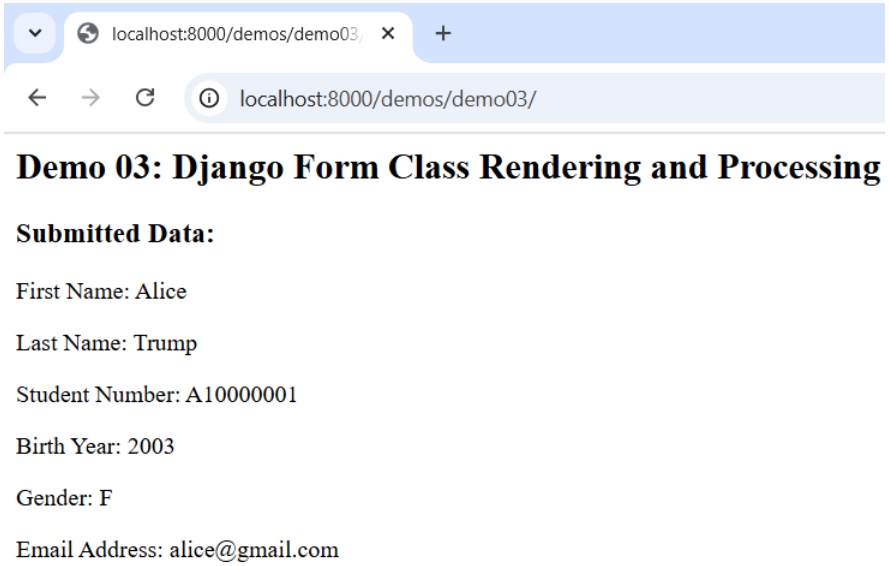
```python
from django import forms


class StudentForm(forms.Form):

    GENDER = [
        ('M', 'Male'),
        ('F', 'Female'),
        ('O', 'Other')
    ]

    firstName = forms.CharField(label='First name', max_length=32)
    lastName = forms.CharField(label='Last name', max_length=32)
    studentNumber = forms.CharField(label='Student number', max_length=9)
    birthYear = forms.IntegerField(label='Birth year')
    gender = forms.ChoiceField(choices=GENDER, label='Gender')
    email = forms.EmailField(label='Email address', required=False)
```

```html
<html>
    <head>
        <title>Demo 03 - Django Form Class Rendering and Processing</title>
    </head>
    <body>
        <h1>Demo 03 - Django Form Class Rendering and Processing</h1>

        <form action="/demo03" method="post">

            {% csrf_token %}
            {{ form }}

            <input type="submit" value="Submit">
        </form>

    </body>
</html>
```

Example of Django Form Class (cont.)



Initial GET request



Subsequent POST request

Working with Form Templates

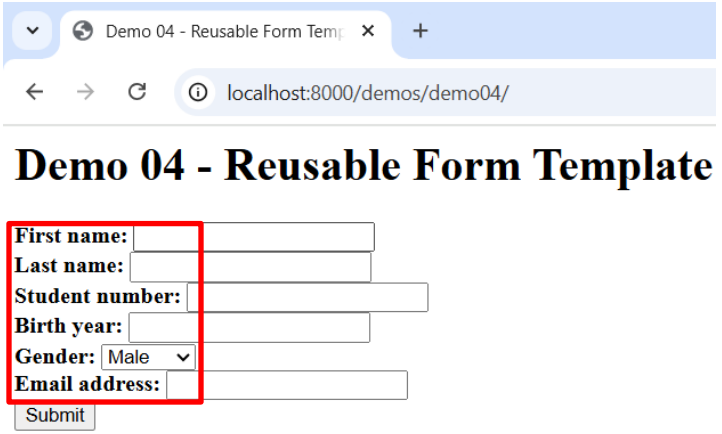
- In general, the form instance is placed in a template's context:
  - The `{{ form }}` DTL syntax is then used to render a form's `<label>` and `<input>` elements.
  - Note that the name of the form instance variable can be anything.
- Reusable form templates:
  - Since the HTML output of a form is generated via a template, the process can be controlled and customised.
  - A template setting file can be created and reused.
  - To reuse site-wide, set a custom `FORM_RENDERER` in `settings.py` to use the required `CustomFormRenderer`.

Working with Form Templates (cont.)

- To reuse in a specific view function, pass the template name directly to `Form.render()`.
- More fine grained control over field rendering is possible:
  - Can be combined with form template.
  - See https://docs.djangoproject.com/en/5.2/topics/forms/ for more information.
- See `demo04` for an example using `Form.render()`.

```
1  {% for field in form %}
2      <div class="fieldWrapper">
3          {{ field.errors }}
4          <b>{{ field.label_tag }}</b> {{ field }}
5      </div>
6  {% endfor %}
```

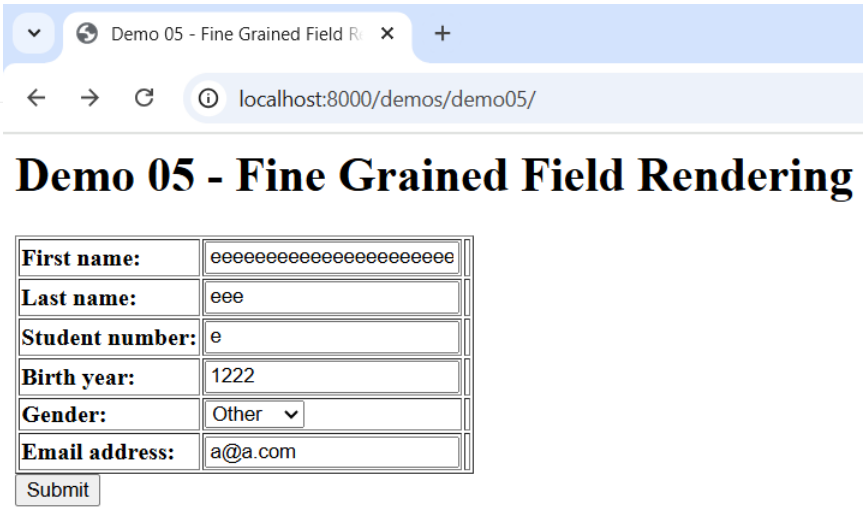
`form_template.html` – Observe the use of the `<b>` tag to bold the field label.



Working with Form Templates (cont.)

- See `demo05` for an example with more fine grained control of field rendering.

```
1  <table border="1">
2
3  {% for field in form %}
4
5      <tr>
6          <td><b><label for="{{ field.id_for_label }}">{{ field.label_tag }}</label></b></td>
7          <td>{{ field }}</td>
8          <td style="color: red;">{{ field.errors }}</td>
9      </tr>
10
11 {% endfor %}
12
13 </table>
```



Django ModelForm Class

- When building a database driven application, it is likely that there will be forms that map closely to Django models:
  - It will be redundant to define the fields in another form since they would have already been defined in the model.
  - Django provides a helper class that can be used to create a Form class from a Django model.
  - This helper class is the `django.forms.ModelForm`.
- Using `ModelForm` makes it easier to perform create and update operations on data stored in a database.

Django ModelForm Class (cont.)

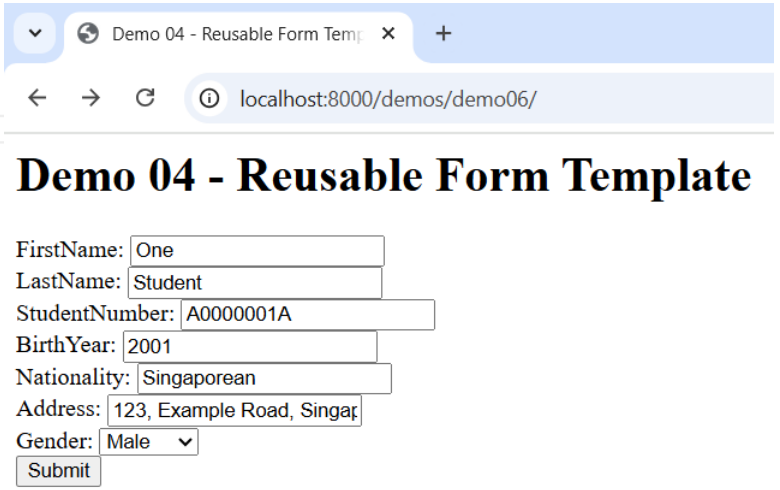
- See `demo06` for the demonstration using the `BetterStudent` model.

```python
class BetterStudent(models.Model):

    GENDER = [
        ('M', 'Male'),
        ('F', 'Female'),
        ('O', 'Other')
    ]

    firstName = models.CharField(max_length=32)
    lastName = models.CharField(max_length=32)
    studentNumber = models.CharField(max_length=9)
    birthYear = models.IntegerField()
    nationality = models.CharField(max_length=32, default='Singaporean')
    address = models.CharField(max_length=128, blank=True, null=True)
    gender = models.CharField(max_length=1, choices=GENDER)
```

```python
class BetterStudentForm(forms.ModelForm):

    class Meta:

        model = BetterStudent
        fields = ['firstName', 'lastName', 'studentNumber', 'birthYear', 'nationality', 'address', 'gender']
```

Django ModelForm Class (cont.)

```python
def demo06(request):

    if request.method == 'GET':

        s1 = BetterStudent.objects.get(pk=1)
        form = BetterStudentForm(instance=s1)

        return render(request, 'demos/demo06.html', {'form': form})

    elif request.method == 'POST':

        form = BetterStudentForm(request.POST)

        if form.is_valid():

            s1 = BetterStudent.objects.get(pk=1)
            s1.firstName = form.cleaned_data['firstName']
            s1.lastName = form.cleaned_data['lastName']
            s1.studentNumber = form.cleaned_data['studentNumber']
            s1.birthYear = form.cleaned_data['birthYear']
            s1.nationality = form.cleaned_data['nationality']
            s1.address = form.cleaned_data['address']
            s1.gender = form.cleaned_data['gender']
            s1.save()

            return HttpResponse('Demo 06: Student information updated successfully.')
```



Django Class-based Views

- **Class-based views** provide an alternative way to implement views as Python objects instead of functions:
  - Class-based views do not replace function-based views.
- But class-based views have certain differences and advantages when compared to function-based views:
  - Organization of code related to specific HTTP methods (GET, POST, etc.) can be addressed by separate methods instead of conditional branching.
  - Object oriented techniques such as mixins (multiple inheritance) can be used to factor code into reusable components.

Django Class-based Views (cont.)

- Using class-based views:
  - A class-based view can be used to respond to different HTTP request methods with different class instance methods.
  - This negates the use of conditional branching code inside a single view function.
  - The code to handle HTTP GET in a view function (left) versus in a class-based view (right) will resemble the followings:

```python
from django.http import HttpResponse


def my_view(request):
    if request.method == "GET":
        # <view logic>
        return HttpResponse("result")
```

```python
from django.http import HttpResponse
from django.views import View


class MyView(View):
    def get(self, request):
        # <view logic>
        return HttpResponse("result")
```

Django Class-based Views (cont.)

- Handling forms with class-based views:
  - A basic function-based view that handles forms (left) versus a similar class-based view (right) will resemble the followings:

```python
from django.http import HttpResponseRedirect
from django.shortcuts import render

from .forms import MyForm


def myview(request):
    if request.method == "POST":
        form = MyForm(request.POST)
        if form.is_valid():
            # <process form cleaned data>
            return HttpResponseRedirect("/success/")
    else:
        form = MyForm(initial={"key": "value"})

    return render(request, "form_template.html", {"form": form})
```

```python
from django.http import HttpResponseRedirect
from django.shortcuts import render
from django.views import View

from .forms import MyForm


class MyFormView(View):
    form_class = MyForm
    initial = {"key": "value"}
    template_name = "form_template.html"

    def get(self, request, *args, **kwargs):
        form = self.form_class(initial=self.initial)
        return render(request, self.template_name, {"form": form})

    def post(self, request, *args, **kwargs):
        form = self.form_class(request.POST)
        if form.is_valid():
            # <process form cleaned data>
            return HttpResponseRedirect("/success/")

        return render(request, self.template_name, {"form": form})
```

Django Class-based Views (cont.)

- When defining the URL in `views.py`, the `as_view()` method of the class-based view should be specified instead of a view function.
- See `demo07` for the demonstration using the `BetterStudent` model:
  - This demonstration combines the use of class-based view with ModelForm class and form template.
  - The use of OOP in Django development significantly enhances code modularity and reusability.

Conversational State

- Many web applications require that a series of requests from the same client/user be associated with one another:
  - E.g., a web application can save the state of a user's shopping cart across requests.
  - Web applications are responsible for maintaining such state because HTTP is stateless.
  - The state is typically known as the **conversational state** or a **session**.
  - A session is not shared and belongs to only one client/user.
  - Session state is considered transient in nature, i.e., the state is not persistent.

Conversational State (cont.)

- Example of a session interaction in an e-commerce application:
  - A client invokes the `addItem()` method of a shopping cart repeatedly to add multiple items only to its own cart.
  - Each client has its own shopping cart, and the items are not mixed together.
  - Throughout a shopping session, a client's shopping cart items are not lost.
  - The same logic applies to the `removeItem()` method.
  - How about the `checkout()` method?

Handling Conversational State with Django Session

- Django provides support for handling conversational state with its session framework:
  - The session framework allows the storing and retrieval of arbitrary data on a per-site-visitor basis.
  - It stores data on the server side and abstracts the sending and receiving of cookies.
  - Cookies contain a session ID but not the data itself.
- To ensure session functionality, ensure that the `MIDDLEWARE` section in `settings.py` contains the `SessionMiddleware` declaration:
  - We will discuss more about middleware in the next sub-topic.

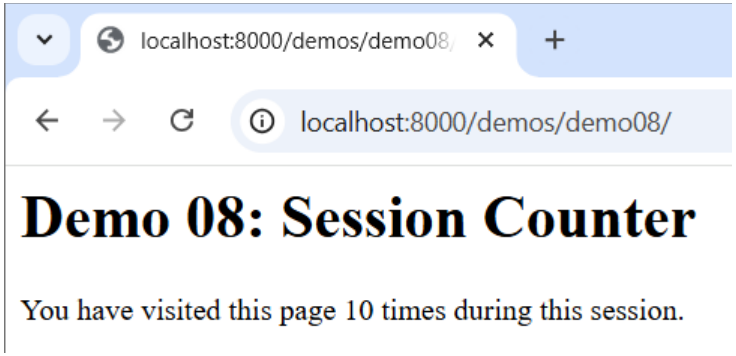
Handling Conversational State with Django Session (cont.)

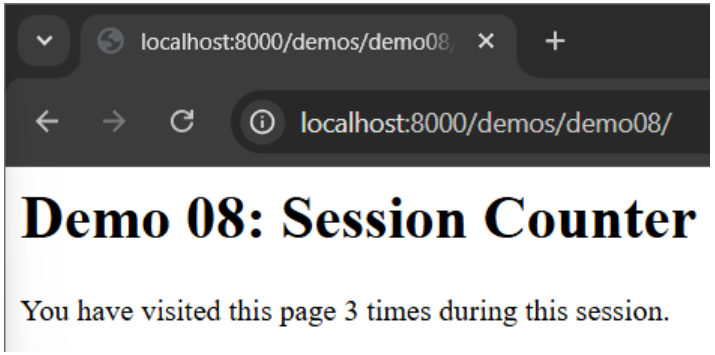
- Configuring the session engine:
  - By default, Django stores sessions in a database.
  - To use database-backed sessions, ensure that "django.contrib.sessions" is added to the `INSTALLED_APPS` section in `settings.py`.
  - Django can also be configured to store session data in the filesystem or cache.
- Using sessions in views:
  - When `SessionMiddleware` is activated, each `HttpRequest` object (the first argument to any Django view function) will have a `session` attribute.
  - The `request.session` is a dictionary-like object.

Handling Conversational State with Django Session (cont.)

- You can read and write to `request.session` at any point in a view.

- See `demo08` for a demonstration with the increment of a session counter:
  - The counter is initialised to 1 if it is not currently in `request.session`.
  - The screenshot below shows two web browser instances each with its own session counter:





Handling Conversational State with Django Session (cont.)

- The default session timeout duration is 2 weeks:
  - The duration can be changed in `settings.py` by setting `SESSION_COOKIE_AGE` to a suitable value in seconds.
  - The session data can be cleared by call `request.session.flush()`, e.g., after logout.
  - Even though Django stores session data in a database by default, it is still considered transient.

Django Middleware

- In Django, **middleware** is a framework of hooks into its request/response processing lifecycle:
  - Think of middleware as a filter.
  - A HTTP request is intercepted by the filter for preprocessing before the request is handled by a view.
  - Middleware can be used for altering input and output.
- Each middleware component is responsible for doing some specific function:
  - E.g., Django includes a middleware component, `AuthenticationMiddleware`, that enables user authentication.
  - Custom middleware can also be created.

Creating a Custom Middleware

- A middleware factory is a callable that takes a `get_response` callable and returns a middleware.
- A middleware is a callable that takes a `request` and returns a `response`, just like a view.
- A middleware can be written as a function that resembles the followings:

```python
def simple_middleware(get_response):
    # One-time configuration and initialization.

    def middleware(request):
        # Code to be executed for each request before
        # the view (and later middleware) are called.

        response = get_response(request)

        # Code to be executed for each request/response after
        # the view is called.

        return response

    return middleware
```

Creating a Custom Middleware (cont.)

- A middleware can also be written as a class whose instances are callable.
- The `get_response` callable:
  - Can be the actual view if this is the last listed middleware.
  - Can also be the next middleware in the chain.
- Activating middleware:
  - To activate a middleware component, add it to the `MIDDLEWARE` section of `settings.py`.
- Middleware order and layering:
  - During the request phase (i.e., before calling the view), Django applies middleware in the order it is defined in the `MIDDLEWARE` section of `settings.py` top-down.

Creating a Custom Middleware (cont.)

- See `demo09` for a demonstration with the checking of whether user has already login:
  - If user has not login, an error message is rendered.
  - Otherwise, the request eventually reaches the view.

Summary

- Django Model supports class inheritance through three different inheritance styles.
- Django provides advanced support for HTML form processing via Form class and ModelForm class.
- Django provide class-based view for handling HTTP requests in view objects instead of view functions.
- Django supports conversational state via the session framework.
- Django middleware can be used to filter the request/response processing lifecycle.

Q&A

Next Lecture…

- Learn about:
  - Creating API endpoint with Django Rest Framework.

In-class Formative Quiz for Lecture 07