Competency Based Learning Material (CBLM)

Web Application Development with Python

Level-4

Module: Processing Forms

Code: CBLM- OU-ICT-WADP-04-L4-V1

National Skills Development Authority
Chief Advisor's Office
Government of the People's Republic of Bangladesh

Copyright

National Skills Development Authority
Chief Advisor's Office
Level10-11, Biniyog Bhaban,
E-6 / B, Agargaon, Sher-E-Bangla Nagar Dhaka-1207, Bangladesh.
Emailec@nsda.gov.bd
Websitewww.nsda.gov.bd.
National Skills Portalhttp:\\skillsportal.gov.bd

This Competency Based Learning Materials (CBLM) on "**Process Forms**" under the **Web Application Development with Python , Level-4** qualification is developed based on the national competency standard approved by National Skills Development Authority (NSDA)

This document is to be used as a key reference point by the competency-based learning materials developers, teachers/trainers/assessors as a base on which to build instructional activities.

It serves as the document for providing training consistent with the requirements of industry in order to meet the qualification of individuals who graduated through the established standard via competency-based assessment for a relevant job.

This document has been developed by NSDA in association with industry representatives, academia, related specialist, trainer, and related employee. Public and private institutions may use the information contained in this CBLM for activities benefitting Bangladesh.

Approved by the Authority ……. meeting held on .....

How to use this Competency Based Learning Material (CBLM)

The module contains training materials and activities for you to complete. These activities may be completed as part of structured classroom activities or you may be required you to work at your own pace. These activities will ask you to complete associated learning and practice activities in order to gain knowledge and skills you need to achieve the learning outcomes.

1. Review the **Learning Activity** page to understand the sequence of learning activities you will undergo. This page will serve as your road map towards the achievement of competence.

2. Read the **Information sheet s.** This will give you an understanding of the jobs or tasks you are going to learn how to do. Once you have finished reading the **Information sheet s** complete the questions in the **Self-Check.**

3. **Self-**Checks are found after each **Information sheet** . **Self-Checks** are designed to help you know how you are progressing. If you are unable to answer the questions in the **Self-Check** you will need to re-read the relevant **Information sheet** . Once you have completed all the questions check your answers by reading the relevant **Answer Keys** found at the end of this module.

4. Next move on to the **Job Sheets. Job Sheets** provide detailed information about *how to do the job* you are being trained in. Some **Job Sheets** will also have a series of **Activity Sheets**. These sheets have been designed to introduce you to the job step by step. This is where you will apply the new knowledge you gained by reading the Information sheet s. This is your opportunity to practise the job. You may need to practise the job or activity several times before you become competent.

5. Specification **sheets**, specifying the details of the job to be performed will be provided where appropriate.

6. A review of competency is provided on the last page to help remind if all the required assessment criteria have been met. This record is for your own information and guidance and is not an official record of competency

When working though this Module always be aware of your safety and the safety of others in the training room. Should you require assistance or clarification please consult your trainer or facilitator.

When you have satisfactorily completed all the Jobs and/or Activities outlined in this module, an assessment event will be scheduled to assess if you have achieved competency in the specified learning outcomes. You will then be ready to move onto the next Unit of Competency or Module

Table of Contents

Module Content

Unit of Competency	**Process Forms**
Unit Code	OU-ICT-WADP-04-L4-V1
Module Title	**Processing Forms**
Module Descriptor	This module covers the knowledge, skills and attitudes required to process Forms It includes the task of creating forms, submitting forms from scratch, using Model Form and Django Custom Form Fields and Widgets
Nominal Hours	**3**0 Hours
Lerning Outcome	After completing the practice of the module, the trainees will be able to perform the following jobs 1. Create forms 2. Submit forms from scratch 3. Use Model Form 4. Use Django Custom Form Fields and Widgets

Assessment Criteria

1. Form is initialized with fields and forms using init method
2. Django form field types are set.
3. Field Layout Values are initiated.
4. Form is Rendered
5. From data is sent with post request
6. Form data is validated
7. Data is saved to database
8. Partial form is used
9. Model form is created
10. Model form is rendered
11. Model form is validated and saved
12. Custom Form Fields are created
13. Built-In Widgets are customized
14. Custom Form Widgets are created
15. Custom Form Widget Configuration Options are created

Learning Outcome 1: Create Forms

Assessment Criteria	1. Form is initialized with fields and forms using init method 2. Django form field types are set. 3. Field Layout Values are initiated. 4. Form is Rendered
Conditions and Resources	1. Real or simulated workplace 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil, Eraser 7. Internet facilities 8. White board and marker 9. Audio Video Device
Contents	1. Form with fields and forms using init method 2. Django form field types a. Widgets b. Options c. Validations 3. Field Layout Values a. label b. label_suffix35 c. help_text
Activities/job/Task	1. Render a Django Form
Training Methods	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Assessment Methods	Assessment methods may include but not limited to 1. Written Test 2. Demonstration 3. Oral Questioning 4. Portfolio

Learning Experience 1: Create Forms

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials 'Create Forms'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Define Tasks of the Project"	2. Read Information sheet 1: Create Forms 3. Answer Self-check 1: Create Forms 4. Check your answer with Answer key 1: Create Forms
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job Sheet 1: Create a Django Form

Information sheet 1: Create Forms

Learning Objective:

After completion of this Information sheet , the learners will be able to explain, define and interpret the following contents:

1.1. Form with fields and forms using init method
1.2. Django form field types
1.3. Field Layout Values
1.4. Form Render

1.1. Initialize Form with fields and forms using init method

After creating a Django Form, if one requires some or all fields of the form be filled with some initial data, one can use functionality of Django forms to do so. It is not the same as a placeholder, but this data will be passed into the view when submitted. There are multiple methods to do this, most common being to pass the data dictionary when we initialize the form in Django view. Other methods include passing initial values through form fields or overriding the __init__ method.

How to pass initial data to a Django form ?

Illustration of passing initial data using an Example. Consider a project named geeksforgeeks having an app named geeks.

Now let's create a demo form in "geeks/forms.py",

```
from django import forms

// creating a django form
class GeeksForm(forms.Form):
    title = forms.CharField()
    description = forms.CharField()
    available = forms.BooleanField()
    email = forms.EmailField()
```

Now to render this form we need to create the view and template which will be used to display the form to user. In geeks/views.py, create a view

```
from django.shortcuts import render
from .forms import GeeksForm

# creating a home view
def home_view(request):
    context = {}
    form = GeeksForm(request.POST or None)
    context['form'] = form
    return render(request, "home.html", context)
```

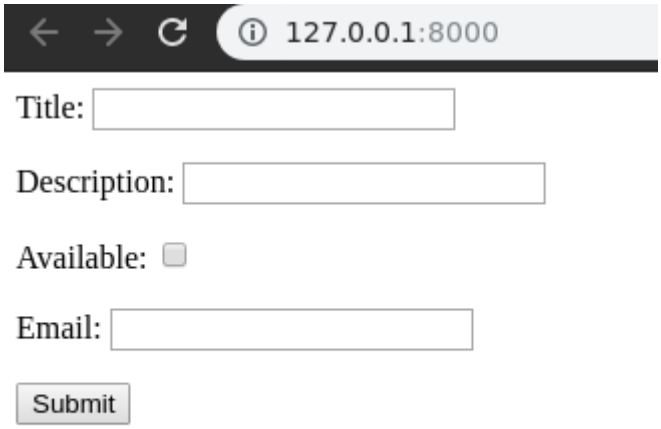
and in templates/home.html,

```
<form method="POST">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Submit">
</form>
```

Now let's display the form by running

Python manage.py runserver

visit http://127.0.0.1:8000/

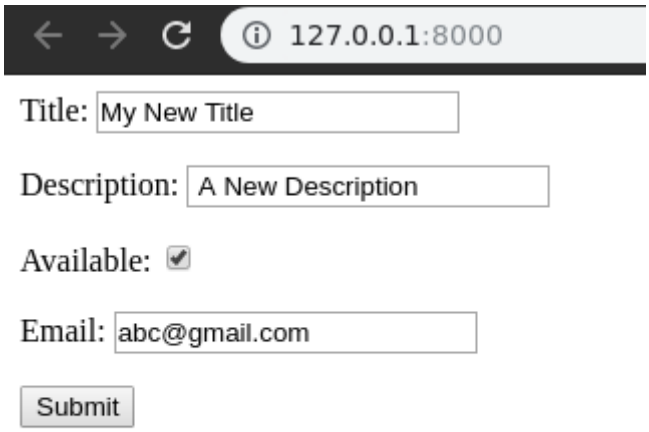


Method 1 – Adding initial form data in views.py

This first and most commonly used method to add initial data through a dictionary is in view.py during the initialization of a form. Here is the code of views.py with some added data.

```
from django.shortcuts import render
from .forms import GeeksForm

def home_view(request):
    context ={}

    # dictionary for initial data with
    # field names as keys
    initial_dict = {
        "title" : "My New Title",
        "description" : " A New Description",
        "available":True,
        "email":"abc@gmail.com"
    }

    # add the dictionary during initialization
    form = GeeksForm(request.POST or None, initial = initial_dict)

    context['form']= form
    return render(request, "home.html", context)
```

Now open http://127.0.0.1:8000/. This method is senior of all and will override any data provided during other methods.

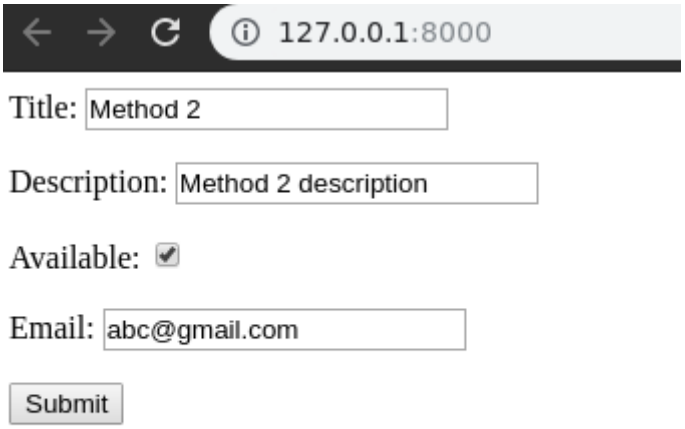


Method 2 – Adding initial form data using fields in forms.py

One can add initial data using fields in forms.py. An attribute **initial** is there for this purpose.

```
In forms.py,
from django import forms

class GeeksForm(forms.Form):
   # adding initial data using initial attribute
   title = forms.CharField(initial = "Method 2 ")
   description = forms.CharField(initial = "Method 2 description")
   available = forms.BooleanField(initial = True)
   email = forms.EmailField(initial = "abc@gmail.com")
```

Now visit, http://127.0.0.1:8000/. One can see the data being updated to method 2.



This way one can add initial data to a form in order to ease the work by a user or any related purpose. This data will be passed to models or views as defined by user and would act as normal data entered by user in the form.

1.2. Set Django form field types

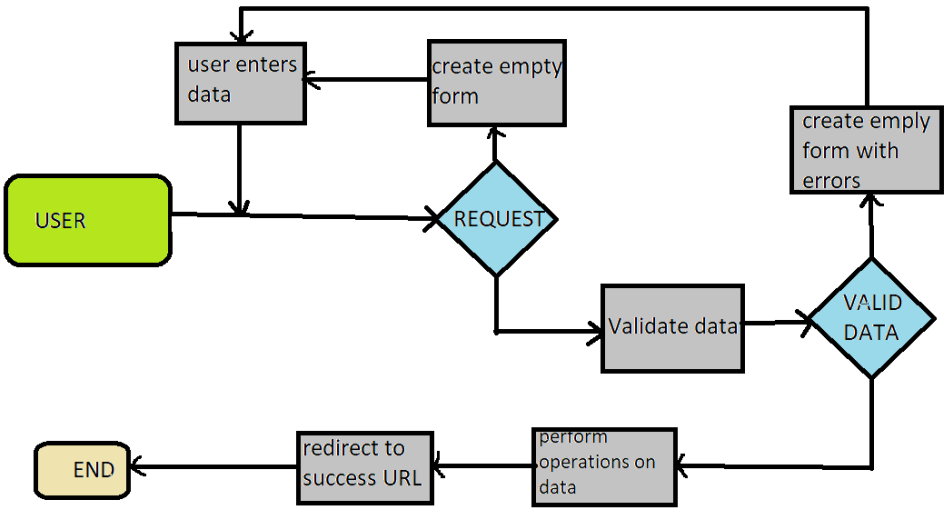
The most crucial component of creating a Form class is defining the form's fields. Each field has its own set of validation rules, as well as a few other hooks.

This article discusses the many fields that can be used in a form, as well as the various features and approaches associated with Django Forms. Forms are used to collect user input and use it to perform logical operations on databases. For example, the process of registering a user by taking input as his name, email, password, etc.

Django Form Fields

Django transforms Django form fields into HTML input fields. Django takes care of three main aspects of form work:

- preparing and reorganizing data in preparation for rendering
- creating HTML forms for the data
- collection and processing client-submitted forms and data



It's worth noting that all of the work done by Django forms can be done with advanced HTML, but Django makes it easier and more efficient, particularly the validation aspect. Once you've mastered Django forms, you'll never go back to HTML forms.

Syntax

Django Fields are similar to Django Model Fields in that they have the following syntax:

```
field_name = forms.FieldType(**options)
```

Example

Following is a simple example of forms in Django

```
from django import forms

 # creating a form

class NinjaForm(forms.Form):

title = forms.CharField()

description = forms.CharField()
```

How to use Django Forms

To utilize Django Forms, you must first create a project and an app in it. You can construct a form in **app/forms.py** after you've started an app.

Creating a Django Form

In Django, building a form is quite similar to creating a model. The type of fields that would exist in the state must be specified. To fill out a registration form, for example, you might need to know your first name (CharField), your roll number (IntegerField), and so on.

Enter the code in ninjas/forms.py to build a form.

```
# import the standard Django Forms

# from built-in library

from django import forms


# creating a form

class InputForm(forms.Form):

 first_name = forms.CharField(max_length = 200)

last_name = forms.CharField(max_length = 200)

roll_number = forms.IntegerField(

help_text = "Enter 6 digit roll number"

)

password = forms.CharField(widget = forms.PasswordInput())
```

Render Django Forms

Although Django form fields include various built-in methods to help developers, customizing the User Interface occasionally necessitates manual implementation (UI). A form has three built-in ways for rendering Django form fields.

- {{ form.as_table }} will render them as table cells wrapped in <tr> tags
- {{ form.as_p }} will render them wrapped in <p> tags

- {{ form.as_ul }} will render them wrapped in <li> tags

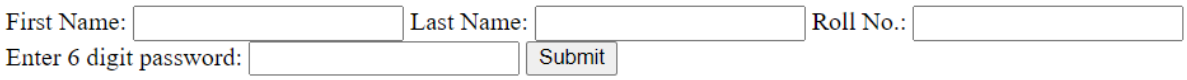
Move to views.py and construct a home view below to render this form into a view.

```
from django.shortcuts import render
from .forms import InputForm


# Create your views here.
def home_view(request):
context ={}
context['form']= InputForm()
return render(request, "home.html", context)
```

In view, one needs to just create an instance of the form class created above in forms.py. Now let's edit templates > home.html

```
<form action = "" method = "post">
{% csrf_token %}
{{form }}
<input type="submit" value=Submit">
</form>
```

The output window at localhost would look like this:



Create Django Form from Models

Django ModelForm is a class that is used to directly convert a model into a Django form. If you're building a database-driven app, chances are you'll have forms that map closely to Django models.

Syntax:

```
from django import forms


class FormName(models.Model):
        # each field would be mapped as an input field in HTML
       field_name = models.Field(**options)
```

Now when we have our project ready, create a model in ninjas/models.py,

```
# import the standard Django Model
# from built-in library
from django.db import models



# declare a new model with a name "GeeksModel"
class NinjasModel(models.Model):
# fields of the model
title = models.CharField(max_length = 200)
description = models.TextField()
last_modified = models.DateTimeField(auto_now_add = True)
img = models.ImageField(upload_to = "images/")



# renames the instances of the model
# with their title name
def __str__(self):
return self.title
```

Enter the following code in ninjas/forms.py to construct a form for this model directly:

```
# import form class from django
from django import forms



# import GeeksModel from models.py
from .models import NinjasModel



# create a ModelForm
class NinjasForm(forms.ModelForm):
# specify the name of model to use
class Meta:
model = NinjasModel
fields = "__all__"
```

Basic Form Data Types and Fields

The list of fields that a form defines is the most significant and only required portion of the form. Fields are specified via class attributes.

The following is a list of all Django Form Field types.

Name	Class	HTML Input
BooleanField	class BooleanField(**kwargs)	CheckboxInput
CharField	class CharField(**kwargs)	TextInput
ChoiceField	class ChoiceField(**kwargs)	Select
TypedChoiceField	class TypedChoiceField(**kwargs)	Select
DateField	class DateField(**kwargs)	DateInput
DateTimeField	class DateTimeField(**kwargs)	DateTimeInput
DecimalField	class DecimalField(**kwargs)	When Field.localize is False, use NumberInput; otherwise, use TextInput
DurationField	class DurationField(**kwargs)	TextInput
EmailField	class EmailField(**kwargs	EmailInput
FileField	class FileField(**kwargs)	ClearableFileInput
FilePathField	class FilePathField(**kwargs)	Select
FloatField	class FloatField(**kwargs)	When Field.localize is False, use NumberInput; otherwise, use TextInput

ImageField	class ImageField(**kwargs)	ClearableFileInput
IntegerField	class IntegerField(**kwargs)	If Field.localize is False, use NumberInput; otherwise, use TextInput
GenericIPAddressField	class GenericIPAddressField(**kwargs)	TextInput
URLField	class URLField(**kwargs)	URLInput
TimeField	class TimeField(**kwargs)	TimeInput
RegexField	class RegexField(**kwargs)	TextInput
MultipleChoiceField	class MultipleChoiceField(**kwargs)	SelectMultiple
TypedMultipleChoiceField	class TypedMultipleChoiceField(**kwargs)	SelectMultiple
SlugField	class SlugField(**kwargs)	TextInput
NullBooleanField	class NullBooleanField(**kwargs)	NullBooleanSelect
UUIDField	class UUIDField(**kwargs)	TextInput

Core Field Arguments

The arguments given to each field for applying a constraint or imparting a specific characteristic to that field are known as core Field arguments.

If you give CharField core field argument **required = False** option, the user will be able to leave it blank.

Each constructor of the Field class takes at least these arguments. Although some Field classes allow field-specific arguments. The following is a list of core field arguments.

Field Options	Description
required	Because each Field class expects the value is needed by default, you must set required=False to make it optional.
label	The label argument allows you to give this field a "human-friendly" label. When the field is presented in a Form, this is used.
label_suffix	On a per-field basis, the label suffix parameter allows you to override the form's label suffix.
widget	When rendering this field, the widget parameter allows you to choose a Widget class to use. For further information, see Widgets.
help_text	You can specify descriptive text for this field using the help text option. When produced by one of the convenient Form methods, help text will be displayed next to the field if you give it
error_messages	You can override the field's default error messages with the error messages option. To override error messages, pass in a dictionary with keys that match the ones you wish to override
validators	You can give a list of validation functions for this field using the validators option.
localize	The localize option allows form data input and displayed output to be localized.
disabled	When set to True, the disabled boolean argument disables a form field via the disabled HTML property, making it non editable by users. localize

1.3. Field Layout Values

When you pass a Django form – unbound or bound – to a template there are many options to generate its layout. You can use one of Django's pre-built HTML helpers to quickly generate a form's output or granularly output each field to create an advanced form layout (e.g. responsive design[3]).

Example shows the Django form I'll use throughout the remaining layout sections – which is the same form used throughout this sheet.

Example. Django form class definition

```
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField(required=False)
    email = forms.EmailField(label='Your email')
    comment = forms.CharField(widget=forms.Textarea)
```

Output form fields: form.as_table, form.as_p, form.as_ul & granularly by field

Django forms offer three helper methods to simplify the output of all form fields. The syntax form.as_table outputs a form's fields to accommodate an HTML <table> as illustrated in Example. The syntax form.as_p outputs a form's fields with HTML <p> tags as illustrated in Example1. Where as the syntax form.as_ul outputs a form's fields to accommodate an HTML <ul> list tag, as illustrated in Example.

Caution: If you use form.as_table, form.as_p, form.as_ul you must declare opening/closing HTML tags, a wrapping <form> tag, a Django {% csrf_token %} tag and an <input type="submit"> button.

Example. Django form output with form.as_table

```
<tr>
  <th><label for="id_name">Name:</label></th>
  <td><input id="id_name" name="name" type="text" /></td>
</tr>\n
<tr>
  <th><label for="id_email">Your email:</label></th>
  <td><input id="id_email" name="email" type="email" required/></td>
</tr>\n
<tr>
  <th><label for="id_comment">Comment:</label></th>
  <td><textarea cols="40" id="id_comment" name="comment" rows="10" required>\r\n</textarea></td>
</tr>
```

Example. Django form output with form.as_p

```
<p>
  <label for="id_name">Name:</label>
  <input id="id_name" name="name" type="text" />
</p>\n
<p>
  <label for="id_email">Your email:</label>
  <input id="id_email" name="email" type="email" required/>
</p>\n
```

```
<p>
   <label for="id_comment">Comment:</label>
   <textarea cols="40" id="id_comment" name="comment" rows="10"
required>\r\n</textarea>
</p>'
```

Example. Django form output with form.as_ul

```
<li>
   <label for="id_name">Name:</label>
   <input id="id_name" name="name" type="text" />
</li>\n
<li>
   <label for="id_email">Your email:</label>
   <input id="id_email" name="email" type="email" required/>
</li>\n
   <li><label for="id_comment">Comment:</label>
   <textarea cols="40" id="id_comment" name="comment" rows="10"
required>\r\n</textarea>
</li>
```

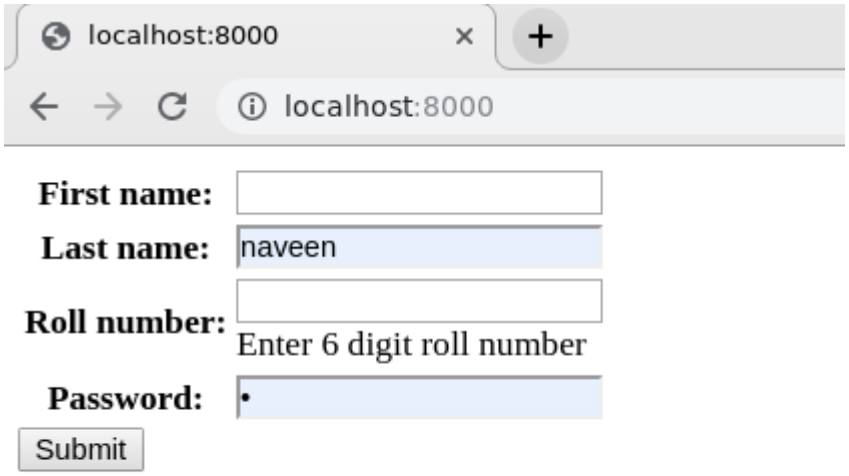
Under certain circumstances, none of the previous helper methods may be sufficient to achieve certain form layouts. For example, to create a responsive design you'll need to output each of field manually to accommodate specific layout requirement (e.g. Bootstrap CSS grid columns). To achieve the custom output of fields, every form instance permits access to its fields through the form.<field_name> syntax using the attributes in table.

Table: Django form field attributes accessible in templates

Attribute name	**Description**
{{form.<field_name>}} (i.e. No attribute, just the field name by itself)	Outputs the HTML form tag -- technically known as the Django widget -- associated with the field (e.g. <input type="text">)
{{form.<field_name>.name}}	Outputs the name of a field, as defined in the form class.
{{form.<field_name>.value}}	Outputs the value of the field assigned with initial or user provided data. Useful if you need to separately output the HTML form tag's value attribute (e.g. for <input type="text" name="name" value="John Doe">, {{form.name.value}} outputs John Doe)
{{form.<field_name>.label}}	Outputs the label of a field, which by default uses the syntax "Your <field_name>" (e.g. for the email field, {{form.email.label}} outputs Your email).
{{form.<field_name>.id_for_label}}	Outputs the label id of a field, which by default uses the syntax id_<field_name> (e.g. for the email field, {{form.email.id_for_label}} outputs id_email).

{{form.<field_name>.auto_id}}	Outputs the auto id of a field, which by default uses the syntax id_<field_name> (e.g. for the email field, {{form.email.auto_id}} outputs id_email).
{{form.<field_name>.label_tag}}	Helper method to output the HTML <label> tag along with id_for_label and label(e.g. for the email field, {{form.email.label_tag}} outputs <label for="id_email">Your email:</label>).
{{form.<field_name>.help_text}}	Outputs the help text associated with a field.
{{form.<field_name>.errors}}	Outputs the errors associated with a field.
{{form.<field_name>.css_classes}}	Outputs the CSS classes associated with a field.
{{form.<field_name>.as_hidden}}	Outputs the HTML of a field as a hidden HTML field (e.g. <input type="hidden" >)
{{form.<field_name>.is_hidden}}	Boolean result of a field's hidden status.
{{form.<field_name>.as_text}}	Outputs the HTML of a field as a text HTML field (e.g. <input type="text">)
{{form.<field_name>.as_textarea}}	Outputs the HTML of a field as a textarea HTML field (e.g. <textarea></textarea>)
{{form.<field_name>.as_widget}}	Outputs the Django widget associated with a field; technically produces the same output as calling the standalone field with the syntax {{form.<field_name>}} -- shown at the top of this table.

As you can see in table, there are many field attributes available to customize the layout of a form. Just be careful that if you output form fields granularly you don't miss a field, because if you do miss a field, the most likely outcome is Django won't be able to process the form as it won't receive values from missing fields.



Example illustrates a standard {% for %} loop which ensures you don't miss any field and provides more flexibility than the previous form.as_table, form.as_p & form.as_ul methods.

Example. Django form {% for %} loop over all fields

```
{% for field in form %}
  <div class="row">
    <div class="col-md-2">
     {{ field.label_tag }}
     {% if field.help_text %}
       <sup>{{ field.help_text }}</sup>
     {% endif %}
     {{ field.errors }}
    </div><div class="col-md-10 pull-left">
      {{ field }}
    </div>
  </div>
{% endfor %}
```

In Example, a loop is created over the form reference to ensure no fields are missed. If you want to avoid presenting a field in certain form layouts, then I recommend you use the {{field.as_hidden}} vs. {{field}}, as this ensures the field still forms part of the form for validation purposes and is simply hidden from a user.

Output field order: field_order and order_fields.

If you use any of the techniques presented in Examples 4-20, 4-21, 4-22 or 4-23, the form fields are output in the same order as they're declared in the form class in Example (i.e. name,email,comment). However, you can use several techniques to alter the order in which form fields are output.

The first and obvious approach is to change the form field order directly in the form class definition. Because this last technique requires altering a form's source code, Django also offers the field_order option. The field_order option accepts a list of form field names in the order you want them output (e.g. field_order=['email','name','comment'] outputs the email field first, followed by name and comment). The field_order option is flexible enough that you can provide a partial list of form fields (e.g. field_order=['email'] outputs the email field first and the remaining form fields in their declared order) as well as declare non-existent field names which are ignored and is helpful when using form inheritance.

The field_order option can be declared in two locations. First, it can be declared as part of a form class definition, as illustrated in Example.

Example, Django form field_order option to enforce field order

```
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField(required=False)
    email = forms.EmailField(label='Your email')
    comment = forms.CharField(widget=forms.Textarea)
    field_order = ['email','comment','name']
```

As you can see in Example, field_order is declared as any other form field and assigned a list of field names to ensure the fields are output in the order: email, comment and name. It's also possible to use the field_order option as part of a form's initialization processes -- described in detail in the form processing section. It's worth mentioning that if you use the field_order option on both the class definition -- as shown in Example -- and form instance initialization, the latter value takes precedence over the former.

In addition to the field_order option, Django also offers order_fields which also expects a list of field names to alter a form's output field order. But unlike the field_order option which must be declared in a form class or as part of the initialization of a form instance, order_fields can be called directly on a form instance which makes it a good option to use in a view method or template (e.g. form.order_fields(['email'])).

Output CSS classes, styles & field attributes: error_css_class, required_css_class, widget customization and various form field options.

By default, when you output form fields and labels there are no CSS classes or styles associated with them. Django offers several mechanisms to associate CSS classes with form fields. The first two approaches are the error_css_class and required_css_class fields which are declared directly in a Django form, as illustrated in Example.

Example. Django form error_css_class and required_css_class fields to apply CSS formatting

```
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField(required=False)
    email = forms.EmailField(label='Your email')
    comment = forms.CharField(widget=forms.Textarea)
    error_css_class = 'error'
    required_css_class = 'bold'
```

As you can see in Example, the error_css_class and required_css_class fields are added just like regular form fields. When a field associated with a form instance of this kind is rendered on a template, Django adds the error CSS class to all fields marked with an error and adds the bold CSS class to all fields marked as required.

For example, all form fields are treated as required except when they explicitly use required=False. This means if you output an unbound form instance from Example using form.as_p, the comment field is output as Comment: <textarea cols="40" name="comment" rows="10" required>\r\n</textarea> -- Note: the class in the tag. Similarly, if a field associated with a bound form instance from Example raises an error, Django adds the error CSS class to the field (e.g. if the email field value is not valid, the email field is output as Your email: <input name="email" type="email" value="aninvalidemail" required /> , Note: the bold CSS class remains because the form field is also required).

As helpful as the error_css_class and required_css_class fields are, they still offer limited CSS formatting functionality. To gain full control over CSS class output, you'll need to either use some of the more granular output options for fields in table or customize a form field's widget as illustrated in Example.

Example. Django form with inline widget definition to add custom CSS class

```
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField(required=False)
    email = forms.EmailField(label='Your email',
widget=forms.TextInput(attrs={'class' : 'myemailclass'}))
    comment = forms.CharField(widget=forms.Textarea)
```

Notice in Example how the email field is declared with the widget=forms.TextInput(attrs={'class' : 'myemailclass'}) argument. This last statement tells Django that when it outputs the email field, it use the custom forms.TextInput widget which declares the CSS class attribute with the myemailclass value.

By using the form definition in Example, the email field is output as <input class="myemailclass" type="text"...>.

The approach presented in Example is a powerful technique, because just as you can declare the CSS class attribute, you can also declare any other form field HTML attribute. For example, if you wanted to declare custom HTML attributes -- such as those used by frameworks like jQuery or Bootstrap -- you can easily use this same technique (e.g.widget=forms.TextInput(attrs={'role' : 'dialog'}) would output <input role="dialog" type="text"...>).

However, a word of Caution: now that you know how easy it's to output any HTML attribute alongside a Django form field. Be aware that nearly all Django form field data types come with built-in options that get translated into HTML attributes. For example, the forms.CharField(max_length=25) statement gets output to <input type="text" maxlength="25"...>, which means the form field max_length option automatically generates the HTML maxlength="25" attribute. So be careful to start adding HTML attributes indiscriminately using the approach in Example, as they may already be supported through built-in data type options.

Output form field errors: form.<field_name>.errors, form.errors, form.non_field_errors

Just as form fields can be output in different ways, form field errors can also be output in different ways. Toward the end of the first section in Example, you can see how we use the {{field.errors}} syntax to output errors associated with a particular field. However, an important thing to keep in mind when outputting a field's errors value in this manner is the output is generated as an HTML formatted list:

```
<ul class="errorlist">
   <li>Name is required.</li>
</ul>
```

As you can see in Example, the {{fields.errors}} value is list with the errorlist CSS class -- which allows you to provide CSS behaviors like a background color or borders -- and the values are pre-wrapped as list elements.

If you want to strip these wrapping HTML list tags to gain more control over the error layout (e.g. creating a responsive design or CSV list) you can do so creating a loop on each field.errors as illustrated in Example.

Example. Django loop over form.<field_name>.errors

```
{% for field in form %}
  <div class="row">
    <div class="col-md-2">
      {{ field.label_tag }}
      {% if field.help_text %}
        <sup>{{ field.help_text }}</sup>
        {% endif %}
      {% for error in field.errors %}
       <div class="row">
         <div class="alert alert-danger">{{error}}</div>
       </div>
      {% endfor %}
     </div><div class="col-md-10 pull-left">
      {{ field }}
     </div>
  </div>
 {% endfor %}
```

You can see in Example that inside the loop for each field, another loop is made on the field.errors reference to granularly output and assign custom markup to each field error.

As granular as the error output in 4-27 is, this type of layout assumes you want to display a form's error messages besides each field, in addition to requiring a loop over a form's fields. But what if you want to display a form's errors at the top or besides the main form ? Or if you want to keep using Django's short-cut methods (i.e. form.as_table, form.as_p & form.as_ul) and still display errors ?

Besides the form.<field_name>.errors syntax in Example to access field errors, Django also can output form's errors with the errors and non_field_errors dictionaries as illustrated in Example.

Example. Django form.errors and form.non_field_errors with custom HTML output

```
<!-- Field errors -->
 {% if form.errors %}
  <div class="row">
    {% for field_with_error,error_messages in form.errors.items %}
       <div class="alert alert-danger">{{field_with_error}}  {{error_messages}}</div>
    {% endfor %}
  </div>
 {% endif %}

<!--                Non-field                errors                -->
 {% if form.non_field_errors %}
  <div class="row">
    {% for error in form.non_field_errors %}
    <div class="alert alert-danger">{{error}}</div>
    {% endfor %}
  </div>
 {% endif %}
```

As you can see in Example, the form.errors dictionary provides an aggregated version of all the form.<field_name>.errors, where each dictionary key represents the form field name and the value is a list of error messages with error code (e.g. required) to further filter the error list. If you want to output every form error at the top of a form/page, require error filtering by code type or want to keep using Django's shortcut output form methods (e.g. form.as_table) and obtain form errors, then using form.errors on a template is the way to go.

In addition, notice toward the top of Example the loop over the form.non_field_errors dictionary. The form.non_field_errors contains errors that don't belong to a specific form field -- as discussed earlier in the 'Error form values: errors' section and the special error placeholder field named __all__. Because non-field errors don't apply to a specific form field, it's common to output these type of errors at the top of a form accessing the non_field_errors dictionary.

Be aware that if you use form.errors or form.non_field_errors to output errors, by default the error reference -- {{error_messages}} and {{error}} in Example -- are wrapped as an HTML formatted list (e.g. <ul class="errorlist">
...) but you can add an additional for loop to the error list -- as in Example -- to create a custom HTML error layout.

Finally, it's worth mentioning there are a series of auxiliary methods designed to facilitate error output (e.g. in JSON format), Table in the Django form processing section describes these methods.

Django custom form fields and widgets

In table you saw the wide variety of built-in Django form fields, from basic text and number types to more specialized text types (e.g. CSV, pre-defined options), including file and directory types. But as extensive as these built-in form fields are, in certain circumstances it can be necessary to build custom form fields.

Similarly, in table you learned how all Django forms fields are linked to Django widgets which define the HTML produced by a form field. While in previous sections (e.g. Example and Example) you learned how it's possible to use a form field's widget property to override its default widget with custom properties (e.g. a CSS class attribute) or assign it a different widget altogether (e.g. a forms.Textarea widget to a forms.CharField field), in certain circumstances, playing around with Django's built-in widgets (i.e. adding attributes or switching one built-in widget for another) can be insufficient for complex HTML form inputs.

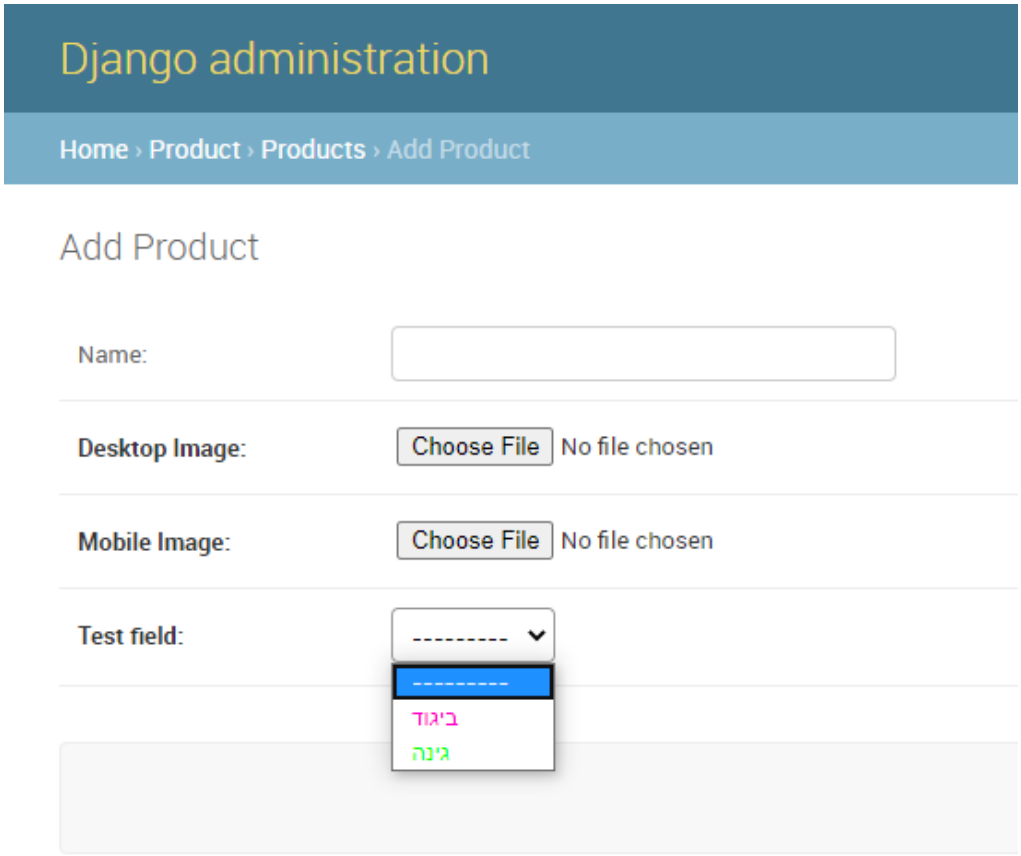
Customize Django form fields, widgets or both ?

As you've learned throughout this sheet, there's a fuzzy relationship between a form field and a form widget, which can make it hard to determine which one to customize when built-in options become insufficient.

As a rule of thumb, if you need to constantly change a form field's data or validation logic, you should use a custom form field. If you need to constantly change a form field's HTML output (e.g. form tags, CSS classes, JavaScript events), you should use a custom widget.

Create custom form fields

The good news about creating custom form fields is you don't have to write everything from scratch, compared to other custom Django constructs (e.g. a custom template filter or custom context processor). Since Django's built-in form fields are sub-classes that inherit their behavior from the forms.Form class, you can further sub-class a built-in form field into a more specialized form field. Therefore, a custom form field can start with a basic set of functionalities present in a built-in form field, which you can then customize as required.



For example, if you find yourself creating forms with the built-in forms.FileField and constantly customizing it in a certain way (e.g. for certain file sizes or types), you can create a custom form field (e.g. PdfFileField, MySpecialFileField) that inherits the behavior from forms.FileField, customize it and use the custom form field directly in your forms.

Example illustrates a custom form field that inherits its behavior from the built-in forms.ChoiceField form field.

Example. Django custom form field inherits behavior from forms.ChoiceField

```
class GenderField(forms.ChoiceField):

    def __init__(self, *args, **kwargs):

        super(GenderField, self).__init__(*args, **kwargs)

        self.error_messages = {"required":"Please select a gender, it's required"}

        self.choices = ((None,'Select gender'),('M','Male'),('F','Female'))
```

As you can see in Example, the GenderField class inherits its behavior from the built-in forms.ChoiceField field class, giving it automatic access to the same behaviors and features as this latter class. Next, the __init__ method -- used to initialize instances of the class -- a call to super() is made to ensure the initialization process of the parent class (i.e. forms.ChoiceField) is made and immediately after values are assigned to the error_messages and choices fields.

The error_messages and choices fields in Example might look familiar because they're arguments typically used in the built-in forms.ChoiceField and are part of Django's standard form field arguments described earlier in this sheet. You can similarly add any other argument supported by form fields (e.g. self.required, self.widget) so the custom form field is created with behaviors set by form field arguments.

Once you have a custom form field like the one in Example, you can use it to declare a form field in a Django form class (e.g. gender = <pkg_location>.GenderField() vs. gender = forms.ChoiceField(error_messages={...},choices=...)). As you can see, custom form fields are a great choice if you're doing repetitive customizations on built-in form fields.

Customize built-in widgets

In Example and Example you already explored some customizations associated with Django's built-in widgets. For example, in Example you saw how it's possible to customize the default built-in widget assigned to form fields, while in Example you saw how it's possible to add custom attributes to built-in widgets. In this section you'll learn how to globally customize built-in widgets.

Looking back at table, you can see for example the forms.widget.TextInput() widget produces an HTML output like <input type="text" ...> and similarly all the other widgets in table produce their own specific HTML output.

Given the high-expectations set forth by many front-end designs, producing this type of basic boilerplate HTML output can be a non-starter for a lot of Django projects. For example, if you want to tightly integrate JavaScript jQuery or ReactJS logic into HTML forms, customizing the default HTML produced by Django's built-in widgets can be a necessity. In these circumstances, the ideal approach is to produce custom HTML for Django's built-in widgets, forgoing the use of the default markup defined by Django in its out-of-the-box state.

The first thing you need to know about customizing Django's built-in widgets is where Django keeps its default built-in widgets. Django builds the HTML output for its built-in widgets from the Django templates located inside the django/forms/templates/django/forms/widgets/ directory in the main Django distribution (e.g. if your Python installation is located

at /python/coffeehouse/lib/python3.5/site-packages/, append this directory path to locate the widget templates)

If you look inside this last directory, you'll find templates (e.g. input.html, radio.html) for each built-in Django widget. Be aware all these widgets templates don't use plain HTML, but instead use Django template syntax (e.g. {% include %} tags, {% if %} conditionals) to favor code re-use.

Now, while you could directly modify the templates in this location to alter the HTML output produced by each widget, don't do this. The recommended approach to customize the output for each built-in widget is to include custom built-in widgets on a project basis. Therefore the first step to customize Django's built-in widgets is to build custom built-in widgets and make them part of a Django project.

Since built-in widgets are Django templates, they need to be placed in a project directory where they can be discovered. This means custom built-in widgets must be placed in a project directory that's part of the DIRS list declared in the TEMPLATES variable in settings.py. In most cases, a project declares a directory named templates -- as part of DIRS -- that also contains a project's templates -- but you can use any directory so long as it's declared as part of DIRS.

Besides using a directory that's part of the DIRS list to locate widget templates, Django also expects to locate custom built-in widgets in the same path it uses for its default built-in widgets (i.e. those included in the Django distribution).

Therefore, if you have a project directory named templates as part of the DIRS list, inside this templates directory you'll need to create the same directory path django/forms/widgets/ and inside this last widgets sub-directory place the custom built-in widgets (e.g. to customize the built-in input.html widget located in the Django distribution at django/forms/templates/django/forms/widgets/input.html you would create a project version at <project_dir>/templates/django/forms/widgets/input.html, this way the latter input.html template takes precedence over the default built-in distribution template).

Once you have the custom built-in widgets set up in your project, you need to make two configuration changes to your project's settings.py The first configuration requires you add the FORM_RENDERER variable, as follows:

```
FORM_RENDERER = 'django.forms.renderers.TemplatesSetting'
```

By default, Django built-in widgets are loaded through a standalone Django template renderer -- django.forms.renderers.DjangoTemplates -- which is unrelated to a project's main TEMPLATES configuration containing the DIRS list with a project's templates. Because you now placed custom built-

in widgets inside a directory declared as part of the DIRS list, you need to configure Django to use a project's main TEMPLATES configuration as part of the form rendering process. By setting the FORM_RENDERER variable to django.forms.renderers.TemplatesSetting, Django also inspects paths in the DIRS list of the main TEMPLATES configuration for built-in widgets.

Finally, because you're overriding the django.forms package in a project to get custom built-in widgets, you must also declare the django.forms package as part of the INSTALLED_APPS list in settings.py.

Create custom form widgets

Custom form widgets are used for cases where you need to keep Django's built-in widget functionality as-is, but still need to customize the HTML output produced by widgets.

Example. Django custom form widget inherits behavior from forms.widgets.Input

```
class PlaceholderInput(forms.widgets.Input):
    template_name = 'about/placeholder.html'
    input_type = 'text'
    def get_context(self, name, value, attrs):
        context = super(PlaceholderInput, self).get_context(name, value, attrs)
        context['widget']['attrs']['maxlength'] = 50
        context['widget']['attrs']['placeholder'] = name.title()
        return context
```

Note: The forms.widgets.Input widget is a more general purpose widget than forms.widgets.TextInput and does not include text input behaviors, hence it's often the preferred choice to build custom widgets due to its basic feature set. It's worth mentioning forms.widgets.Input is the parent widget of forms.widgets.TextInput, forms.widgets.NumberInput, forms.widgets.Email Input and other input widgets described in table.

As you can see in Example, the PlaceholderInput class inherits its behavior from the built-in forms.widgets.Input widget class, giving it access to the same behaviors and features as this latter class.

Next, are two class fields. The template_name field defines the backing template for the custom widget which points to 'about/placeholder.html'-- Note: that if you omit the template_name field, the parent class template is used (i.e. for the forms.widgets.Input widget the template is django/forms/widgets/input.html). The input_type field is a requirement for forms.widgets.Input sub-classes and is used to assign the HTML

input type attribute, values can include: text, number, email, url, password or any other valid HTML input type value.

Inside the custom widget class is the get_context() method, which is used to set the context for the backing widget template -- just like standard Django templates. In this case, a call is made to super() to ensure the context for the parent widget class template (i.e. forms.widgets.Input) is set and immediately after a pair of attributes -- maxlength and placeholder -- are set on the widget context in the ['widget']['attrs'] dictionary for use inside the widget template. Note: the value for maxlength is fixed and the value for placeholder is taken from the field name attribute and converted to a title with Python's standard title method. Finally, the get_context() method returns the updated context reference to pass it to the widget template.

Now lets take a look at the widget template 'about/placeholder.html' in Example.

Example. Django custom form widget inherits behavior from forms.widgets.Input

```
# about/placeholder.html

<input type="{{ widget.type }}"

       name="{{ widget.name }}"

         {% if widget.value != None %}

           value="{{ widget.value }}"

         {% endif %}

         {% include "django/forms/widgets/attrs.html" %} />

# django/forms/widgets/attrs.html

{% for name, value in widget.attrs.items %}

  {% if value is not False %} {{ name }}

      {% if value is not True %}="{{ value }}"{% endif %}

  {% endif %}

{% endfor %}
```

You can see in Example the HTML input tag is generated with values from the widget dictionary set through the get_context method. The values

for widget.type, widget.name and widget.value are all set behind the scenes in the parent class (i.e. forms.widgets.Input). In addition, notice the HTML input tag uses the built-in widget template django/forms/widgets/attrs.html shown at the bottom of Example. This last widget template loops over all the elements in the widgets.attrs context dictionary and generates the input attributes. Since the widgets.attrs context dictionary is modified in Example to include the maxlength and placeholder attributes, the input tag is generated with these additional attributes (e.g. <input type="text" name="email" maxlength="50" placeholder="Email">).

Now let's take a step back to discuss the location of the 'about/placeholder.html' widget template. By default, Django custom widgets are only searched for inside the templates folders in all Django apps defined in INSTALLED_APPS. This means that in order for Django to find the custom 'about/placeholder.html' widget template, it must be placed inside any project app's templates folder (e.g. given a project app named about, the custom widget should be located under templates/about/placeholder.html , where the templates folder is at the same level of an app's models.py and views.py files).

Finally, once you define the custom widget in the right location, you can use it as part of a Django form field (e.g. email=forms.EmailField(widget=<pkg_location>.PlaceholderInput)) to generate an HTML input tag with a default placeholder attribute.

Custom form widget configuration options

In the previous two sections on customizing form widgets, I didn't mention a variety of configuration options in order to avoid getting sidetracked from the main task at hand. But now that you know how to customize Django's built-in widgets and how to create custom form widgets, I can provide you with these additional details.

The first topic is related to finding and loading custom widget templates. Django defines a form renderer through the FORM_RENDERER variable in settings.py. Table describes the three different values supported by the FORM_RENDERER variable.

Table: FORM_RENDERER values that influence finding and loading custom widget templates

Form renderer class	Description
django.forms.renderers.DjangoTemplates **(Default)**	Searches and loads widget templates from the built-in django/forms/templates/ directory (i.e. the distribution). Searches and loads widget templates from all the templates directories inside apps declared in INSTALLED_APPS.
django.forms.renderers.JinjaTemplates	Searches and loads widget templates from the built-in django/forms/jinja2/ directory (i.e. the distribution). Searches and loads widget templates from all the jinja2 directories inside apps declared in INSTALLED_APPS.
django.forms.renderers.TemplatesSetting	Searches and loads widget templates based on a project's TEMPLATES configuration (e.g. its DIRS values). NOTE: This renderer requires you declare the django.forms package as part of INSTALLED_APPS

As you can see in table, there's even a Jinja template renderer to allow you to customize widgets backed by Jinja templates, in addition to other renderers used in the previous sections.

In Example you learned how custom widget classes use fields (e.g. template_name, input_type) to specify certain behaviors. Fields in widgets classes are highly dependent on the parent widget class. For example, although template_name is valid for all built-in widgets, more specialized built-in widgets can accept additional fields. If in doubt, consult the fields supported for the built-in widget[4] used as a widget's parent class.

In Example you also learned how to access and modify a widget's template context through the get_context() method. Although in Example you only added a couple of widget attributes to the context['widget']['attrs'] dictionary, the parent context['widget'] dictionary is a large data structure that stores all data associated with a widget, where you can inclusively update a widget's field values. The following snippet illustrates the contents of the context['widget'] for a form field using the PlaceholderInput widget class from Example:

```
{'widget':
   {'attrs': {'placeholder': 'Email', 'maxlength': 50},
    'name': 'email',
    'is_hidden': False,
    'type': 'text',
    'value': None,
    'template_name': 'about/placeholder.html',
    'required': True
    }
}
```

As you can see, in addition to the HTML input attributes stored under the attrs key, there are other widget field keys (e.g. name, is_hidden) that are made available in a template to render the final output. This also means there's nothing limiting you from adding custom data keys to the context['widget'] dictionary inside the get_context() method to integrate them as part of the final template layout (e.g. 'react':{<react_data>}, 'jquery':{<jquery_data>}).

1.4. Form Render

When rendering an object in Django, we generally:

1. get hold of it in the view (fetch it from the database, for example)
2. pass it to the template context
3. expand it to HTML markup using template variables

Rendering a form in a template involves nearly the same work as rendering any other kind of object, but there are some key differences.

In the case of a model instance that contained no data, it would rarely if ever be useful to do anything with it in a template. On the other hand, it makes perfect sense to render an unpopulated form - that’s what we do when we want the user to populate it.

So when we handle a model instance in a view, we typically retrieve it from the database. When we're dealing with a form we typically instantiate it in the view.

When we instantiate a form, we can opt to leave it empty or prepopulate it, for example with:

- data from a saved model instance (as in the case of admin forms for editing)
- data that we have collated from other sources
- data received from a previous HTML form submission

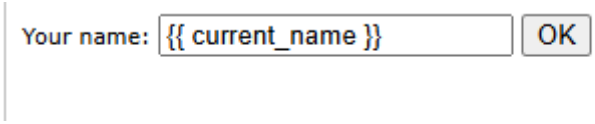
The last of these cases is the most interesting, because it's what makes it possible for users not just to read a website, but to send information back to it too.

Building a form

The work that needs to be done

Suppose you want to create a simple form on your website, in order to obtain the user's name. You'd need something like this in your template:

```
<form action="/your-name/" method="post">

  <label for="your_name">Your name: </label>

  <input id="your_name" type="text" name="your_name" value="{{ current_name }}">

  <input type="submit" value="OK">

</form>
```



This tells the browser to return the form data to the URL **/your-name/**, using the **POST** method. It will display a text field, labeled "Your name:", and a button marked "OK". If the template context contains a **current_name** variable, that will be used to pre-fill the **your_name** field.

You'll need a view that renders the template containing the HTML form, and that can supply the **current_name** field as appropriate.

When the form is submitted, the **POST** request which is sent to the server will contain the form data.

Now you'll also need a view corresponding to that **/your-name/** URL which will find the appropriate key/value pairs in the request, and then process them.

This is a very simple form. In practice, a form might contain dozens or hundreds of fields, many of which might need to be prepopulated, and we might expect the user to work through the edit-submit cycle several times before concluding the operation.

We might require some validation to occur in the browser, even before the form is submitted; we might want to use much more complex fields, that allow the user to do things like pick dates from a calendar and so on.

At this point it's much easier to get Django to do most of this work for us.

Building a form in Django

The Form class

We already know what we want our HTML form to look like. Our starting point for it in Django is this:

```
forms.py
from django import forms

class NameForm(forms.Form):
    your_name = forms.CharField(label="Your name", max_length=100)
```

This defines a **Form** class with a single field (**your_name**). We've applied a human-friendly label to the field, which will appear in the **<label>** when it's rendered (although in this case, the **label** we specified is actually the same one that would be generated automatically if we had omitted it).

The field's maximum allowable length is defined by **max_length**. This does two things. It puts a **maxlength="100"** on the HTML **<input>** (so the browser should prevent the user from entering more than that number of characters in the first place). It also means that when Django receives the form back from the browser, it will validate the length of the data.

A **Form** instance has an **is_valid()** method, which runs validation routines for all its fields. 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.

The whole form, when rendered for the first time, will look like:

```
<label for="your_name">Your name: </label>
<input id="your_name" type="text" name="your_name" maxlength="100" required>
```

Note: that it **does not** include the **<form>** tags, or a submit button. We'll have to provide those ourselves in the template.

The view

Form data sent back to a Django website is processed by a view, generally the same view which published the form. This allows us to reuse some of the same logic.

To handle the form we need to instantiate it in the view for the URL where we want it to be published:

```
views.py
from django.http import HttpResponseRedirect
from django.shortcuts import render

from .forms import NameForm
def get_name(request):
    # if this is a POST request we need to process the form data
    if request.method == "POST":
        # create a form instance and populate it with data from the request:
        form = NameForm(request.POST)
        # check whether it's valid:
        if form.is_valid():
            # process the data in form.cleaned_data as required
            # ...
            # redirect to a new URL:
            return HttpResponseRedirect("/thanks/")

    # if a GET (or any other method) we'll create a blank form
    else:
        form = NameForm()
    return render(request, "name.html", {"form": form})
```

If we arrive at this view with a **GET** request, it will create an empty form instance and place it in the template context to be rendered. This is what we can expect to happen the first time we visit the URL.

If the form is submitted using a **POST** request, the view will once again create a form instance and populate it with data from the request: **form = NameForm(request.POST)** This is called "binding data to the form" (it is now a *bound* form).

We call the form's **is_valid()** method; if it's not **True**, we go back to the template with the form. This time the form is no longer empty (*unbound*) so the HTML form will be populated with the data previously submitted, where it can be edited and corrected as required.

If **is_valid()** is **True**, we'll now be able to find all the validated form data in its **cleaned_data** attribute. We can use this data to update the database or do other processing before sending an HTTP redirect to the browser telling it where to go next.

The template

We don't need to do much in our **name.html** template:

```
<form action="/your-name/" method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Submit">
</form>
```

All the form's fields and their attributes will be unpacked into HTML markup from that **{{ form }}** by Django's template language.

Forms and Cross Site Request Forgery protection

Django ships with an easy-to-use protection against Cross Site Request Forgeries. When submitting a form via **POST** with CSRF protection enabled you must use the **csrf_token** template tag as in the preceding example. However, since CSRF protection is not directly tied to forms in templates, this tag is omitted from the following examples in this document.

HTML5 input types and browser validation

If your form includes a **URLField**, an **EmailField** or any integer field type, Django will use the **url**, **email** and **number** HTML5 input types. By default, browsers may apply their own validation on these fields, which may be stricter than Django's validation. If you would like to disable this behavior, set the **novalidate** attribute on the **form** tag, or specify a different widget on the field, like **TextInput**.

We now have a working web form, described by a Django **Form**, processed by a view, and rendered as an HTML **<form>**.

That's all you need to get started, but the forms framework puts a lot more at your fingertips. Once you understand the basics of the process described above, you should be prepared to understand other features of the forms system and ready to learn a bit more about the underlying machinery.

Self-Check Sheet 1: Use Model Form

1. **What is the primary purpose of a Django form class?**

Answer:

- A. To define the visual appearance of a form in a template.
- B. To define form functionality and handle data validation.
- C. To process submitted form data and interact with databases.
- D. To enforce Cross-Site Request Forgery (CSRF) protection.

2. **What happens when a Django form instance is created with** request.POST **data?**

Answer:

- A. The form instance remains empty.
- B. The form instance is automatically validated.
- C. The form instance is populated with user-submitted data.
- D. The form instance is redirected to a new URL.

3. **What does the** is_valid() **method on a Django form instance do?**

Answer:

- A. It generates the HTML output for the form.
- B. It sets the initial values of the form fields.
- C. It checks if the form data is valid based on field definitions.
- D. It adds a CSRF token to the form.

4. **What is the role of the CSRF token in a Django form?**

Answer:

- A. It defines the layout of the form fields.
- B. It specifies the method (GET or POST) for submitting the form.
- C. It helps prevent Cross-Site Request Forgery attacks.
- D. It validates the format of email addresses entered in the form.

5. **What does the Django template tag** {% csrf_token %} **do?**

Answer:

- A. It defines the action attribute of the form tag.
- B. It validates the submitted form data.
- C. It includes a CSRF token in the form for security purposes.
- D. It redirects the user to a confirmation page after form submission.

Answer Key 1: Use Model Form

1. **What is the primary purpose of a Django form class?**

Answer: B. To define form functionality and handle data validation.

2. **What happens when a Django form instance is created with** request.POST **data?**

Answer: C. The form instance is populated with user-submitted data.

3. **What does the** is_valid() **method on a Django form instance do?**

Answer: C. It checks if the form data is valid based on field definitions.

4. **What is the role of the CSRF token in a Django form?**

Answer: C. It helps prevent Cross-Site Request Forgery attacks.

5. **What does the Django template tag** {% csrf_token %} **do?**

Answer: C. It includes a CSRF token in the form for security purposes.

Job Sheet 1: Create a Django Form

UoC Cover

OU-ICT-WADP-04- L4-V1: Process Forms

Working Procedure / Steps

1. **Define a Form Class:**
    - Create a class that inherits from forms.Form.
    - Define form fields within the class using appropriate field types.
    - Optionally, set properties like required=False on form fields.
2. **Instantiate the Form Class in the View:**
    - In the view method handling the request, create an instance of the form class.
3. **Pass the Form Instance to the Template:**
    - In the view, return the rendered template with the form instance as a context variable.
4. **Render the Form in the Template:**
    - Use Django template tags to render the form instance as HTML.
    - Common methods include:
        - {{ form.as_table }}: Renders the form as an HTML table.
        - Individual field access (e.g., {{ form.name }}): Renders individual form fields.
5. **Wrap the Form in an HTML Form Tag:**
    - The rendered form elements need to be wrapped within a standard HTML <form> tag.
    - Set attributes like method (usually POST) and action (URL to submit the form).
    - Include a submit button (<input type="submit">).

Specification Sheet-1 : Create a Django Form

Technical Requirements

- **Programming Language:** Python 3.7 or later
- **Framework:** Django 3.2 or later
- **Text Editor/IDE:** Visual Studio Code, PyCharm, Sublime Text (or any preferred code editor)

Tools and Equipment

- **Computer:** A computer with sufficient processing power, RAM, and storage.
- **Internet Connection:** A reliable internet connection for downloading and installing software, accessing online resources, and deploying the application (if applicable).
- **Development Environment:** A virtual environment is recommended to isolate project dependencies.
- **Web Server (Optional):** If deploying your application, you'll need a web server like Apache or Nginx.

Learning Outcome 2: Submit forms from scratch

Assessment Criteria	1. From data is sent with post request 2. Form data is validated 3. Data is saved to database 4. Partial form is used
Conditions and Resources	1. Real or simulated workplace 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil, Eraser 7. Internet facilities 8. White board and marker 9. Audio Video Device
Contents	1. From data send with post request 2. Validate Form data 3. Save data to database 4. Use of partial form
Activities/job/Task	1. Exploring Django Forms
Training Methods	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Assessment Methods	Assessment methods may include but not limited to 1. Written Test 2. Demonstration 3. Oral Questioning 4. Portfolio

Learning Experience 2: Submit forms from scratch

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials 'Submit forms from scratch'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Define Tasks of the Project"	2. Read Information sheet 1: Submit forms from scratch 3. Answer Self-check 1: Submit forms from scratch 4. Check your answer with Answer key 1: Submit forms from scratch
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job Sheet-2: Exploring Django Forms

Information sheet 2: Submit forms from scratch

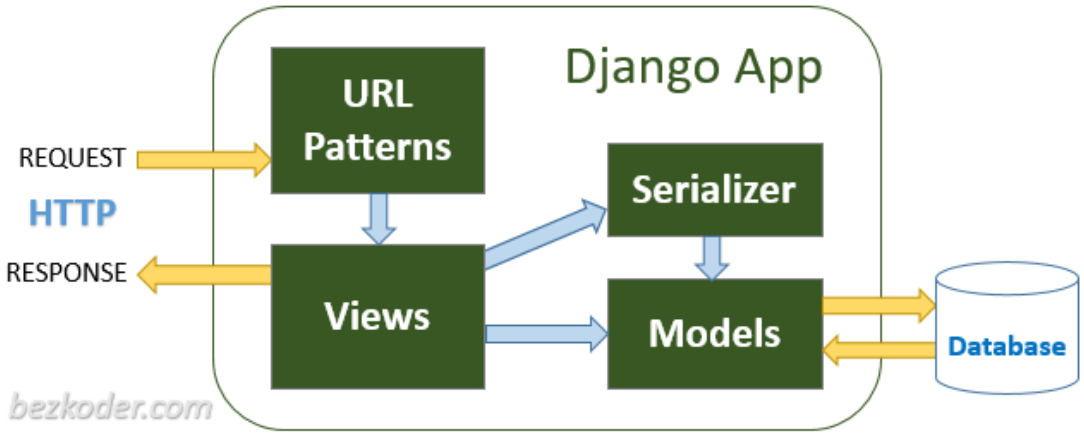
Learning Objective:

After completion of this Information sheet , the learners will be able to explain, define and interpret the following contents:

2.1. From data is sent with post request
2.2. Form data is validated
2.3. Data is saved to database
2.4. Partial form is used

2.1. Form Data is Sent with POST Request

When submitting data through a form in Django, the POST request method is used to send data to the server. This method is typically used for actions like creating or updating records in the database.



Creating a Simple Form and Sending Data with POST

Consider a simple Contact form where users submit their name, email, and message. The data is sent to the server via a POST request.

Example: Form in HTML

```
<!-- contact_form.html -->

<!DOCTYPE html>

<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <title>Contact Us</title>
</head>
<body>
    <h1>Contact Form</h1>
    <form method="POST">
        {% csrf_token %}
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" required><br><br>

        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required><br><br>

        <label for="message">Message:</label><br>
        <textarea id="message" name="message" required></textarea><br><br>

        <button type="submit">Send</button>
    </form>
</body>
</html>
```

Here:

- The form uses the POST method to send data when the submit button is clicked.
- {% csrf_token %} is used for security purposes to prevent Cross-Site Request Forgery (CSRF) attacks.

Handling the Form in Views:

```
# views.py

from django.shortcuts import render, redirect

from .models import Contact


def contact_view(request):

    if request.method == 'POST':

        name = request.POST.get('name')

        email = request.POST.get('email')

        message = request.POST.get('message')


        # You could validate the data here (see the next section)


        # Saving the data to the database

        Contact.objects.create(name=name, email=email, message=message)

        return redirect('thank_you')  # Redirect to a 'thank you' page


    return render(request, 'contact_form.html')
```

In this view:

- The form data is retrieved via request.POST.get('field_name').
- If the form is submitted, the data is used to create a new Contact object and saved to the database.

2.2. Form Data is Validated

After receiving the form data from the user, it is important to validate the data to ensure it meets the necessary criteria (e.g., checking that all required fields are filled out and the email is in the correct format).

Validating Form Data Before Saving

You can validate the form data either manually or by using Django's built-in form validation mechanisms.

Manual Validation Example:

```
# views.py

def contact_view(request):

   if request.method == 'POST':

      name = request.POST.get('name')

      email = request.POST.get('email')

      message = request.POST.get('message')

            # Manual validation

      if not name or not email or not message:

         error_message = "All fields are required."

         return render(request, 'contact_form.html', {'error': error_message})

            # If valid, save the data

      Contact.objects.create(name=name, email=email, message=message)

      return redirect('thank_you')  # Redirect to a 'thank you' page

      return render(request, 'contact_form.html')
```

In this example:

- If any field is empty, the form is considered invalid, and an error message is displayed.

Django forms provide more advanced validation features (e.g., checking that the email field contains a valid email) through the forms.Form class.

2.3. Data is Saved to Database

Once the form data is validated, it can be saved to the database. Django models are used to represent database tables, and the form data can be saved as a model instance.

Saving the Data

In the previous example, we saved the data to the database using Django's ORM. The Contact model was used to store the form data.

Example: Contact Model

```
# models.py

from django.db import models



class Contact(models.Model):

    name = models.CharField(max_length=100)

    email = models.EmailField()

    message = models.TextField()



    def __str__(self):

        return self.name
```

Here:

- The Contact model has fields for name, email, and message.
- Contact.objects.create() is used to insert a new record into the Contact table in the database.

2.4. Partial Form is Used

In some cases, you may want to submit only part of the form data instead of all fields at once. For example, a multi-step form where the user fills out some fields initially, submits them, and then comes back to complete the rest later.

Handling Partial Forms

Let’s say the form consists of two parts: user details (name and email) and the message. We can create two views where the first part is submitted and saved, and the second part is handled separately.

Step 1: Collect User Details

```
<!-- step1_form.html -->

<form method="POST">

    {% csrf_token %}

    <label for="name">Name:</label>

    <input type="text" id="name" name="name" required><br><br>



    <label for="email">Email:</label>

    <input type="email" id="email" name="email" required><br><br>



    <button type="submit">Next</button>

</form>
```

Step 1: View to Handle Partial Data

```
# views.py

from django.shortcuts import render, redirect

from .models import Contact



def step1_view(request):

    if request.method == 'POST':

        name = request.POST.get('name')

        email = request.POST.get('email')
```

```
        # Save user details temporarily
        contact = Contact.objects.create(name=name, email=email, message=None)

        # Redirect to the next step to complete the message
        return redirect('step2', contact_id=contact.id)

    return render(request, 'step1_form.html')
```

In this view:

- We save only the user details (name and email) and create a Contact object with a None message.
- The user is redirected to the second step where they can fill out the message.

Step 2: Collect the Message

```
<!-- step2_form.html -->
<form method="POST">
    {% csrf_token %}
    <label for="message">Message:</label><br>
    <textarea id="message" name="message" required></textarea><br><br>

    <button type="submit">Submit</button>
</form>
```

Step 2: View to Handle Final Data

```
# views.py

def step2_view(request, contact_id):

    contact = Contact.objects.get(id=contact_id)


    if request.method == 'POST':

        message = request.POST.get('message')


        # Save the message to the existing contact object

        contact.message = message

        contact.save()


        # Redirect to a confirmation or thank you page

        return redirect('thank_you')


    return render(request, 'step2_form.html', {'contact': contact})
```

In this view:

- The contact object is retrieved by its ID (which was passed from step 1).
- The message is saved to the existing contact record in the database.

Self-Check Sheet 2: Use Model Form

1. **What is the primary purpose of a Django form class?**

Answer:

- A. To define form functionality and handle data validation.
- B. To define the visual appearance of a form in a template.
- C. To process submitted form data and interact with databases.
- D. To enforce Cross-Site Request Forgery (CSRF) protection.

2. **How does a Django form instance get populated with user-submitted data?**

Answer:

- A. The form instance remains empty.
- B. The form instance is populated with data from request.POST.
- C. The form instance is automatically validated.
- D. The form instance is redirected to a new URL.

3. **What does the** is_valid() **method on a Django form instance do?**

Answer:

- A. It generates the HTML output for the form.
- B. It sets the initial values of the form fields.
- C. It checks if the form data is valid based on field definitions.
- D. It adds a CSRF token to the form.

4. **What is the role of the CSRF token in a Django form?**

Answer:

- A. It defines the layout of the form fields.
- B. It specifies the method (GET or POST) for submitting the form.
- C. It helps prevent Cross-Site Request Forgery attacks.
- D. It validates the format of email addresses entered in the form.

5. **What does the Django template tag** {% csrf_token %} **do?**

Answer:

- A. It defines the action attribute of the form tag.
- B. It validates the submitted form data.
- C. It includes a CSRF token in the form for security purposes.
- D. It redirects the user to a confirmation page after form submission.

Answer Key 2: Use Model Form

1. **What is the primary purpose of a Django form class?**

Answer: A. To define form functionality and handle data validation.

2. **How does a Django form instance get populated with user-submitted data?**

Answer: B. The form instance is populated with data from request.POST.

3. **What does the** is_valid() **method on a Django form instance do?**

Answer: C. It checks if the form data is valid based on field definitions.

4. **What is the role of the CSRF token in a Django form?**

Answer: C. It helps prevent Cross-Site Request Forgery attacks.

5. **What does the Django template tag** {% csrf_token %} **do?**

Answer: C. It includes a CSRF token in the form for security purposes.

Job Sheet-2: **Exploring Django Forms**

UoC Cover

OU-ICT-WADP-04- L4-V1: Process Forms

Working Procedure / Steps

- Create a form class named ContactForm that inherits from django.forms.Form.
- Define form fields like name (optional), email (EmailField), and subject (CharField).
- In the view, handle the form submission and display validation errors using {{ form.non_field_errors }} in your template.

- Create a model named Message in models.py that inherits from django.db.models.Model.
- Define fields in Message like name (CharField, optional), email (CharField), and subject (CharField).
- In the view that handles form submission (assuming valid data):
    - Create a new instance of the Message model.
    - Assign cleaned data from the form to the corresponding fields in the model instance.
    - Save the model instance using instance.save().

- Consider a scenario where you want to reuse the ContactForm but exclude the name and email fields.
- There are two approaches:

Approach A: Hiding Fields in Template

* Keep the `ContactForm` class as-is.
* In the template, use `{{ form.as_p }}` to render the form as paragraphs.
* Utilize CSS to hide the unwanted fields (e.g., `display: none;`) for a visual effect.

Approach B: Creating a Subclass (PartialContactForm)

* Create a new form class `PartialContactForm` that inherits from `ContactForm`.
* Override the `__init__()` method in `PartialContactForm` to exclude unwanted fields from the form's `fields` attribute.
* Use `PartialContactForm` in your view and template for the partial form.

Specification Sheet 2: Exploring Django Forms

Technical Requirements

- Programming Language: Python 3.7 or later
- Framework: Django 3.2 or later
- Text Editor/IDE: Visual Studio Code, PyCharm, Sublime Text (or any preferred code editor)

Tools and Equipment

- Computer: A computer with sufficient processing power, RAM, and storage.
- Internet Connection: A reliable internet connection for downloading and installing software, accessing online resources, and deploying the application (if applicable).
- Development Environment: A virtual environment is recommended to isolate project dependencies.
- Web Server (Optional): If deploying your application, you'll need a web server like Apache or Nginx.

Learning Outcome 3: Use Model Form

Assessment Criteria	1. Model form is created 2. Model form is rendered 3. Model form is validated and saved
Conditions and Resources	1. Real or simulated workplace 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil, Eraser 7. Internet facilities 8. White board and marker 9. Audio Video Device
Contents	1. Create model form 2. Render model form, model form validation and saving technique
Activities/job/Task	1. Create Django Model Form
Training Methods	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Assessment Methods	Assessment methods may include but not limited to 1. Written Test 2. Demonstration 3. Oral Questioning 4. Portfolio

Learning Experience 3: Use Model Form

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
4. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials 'Use Model Form'
5. Read the Information sheet and complete the Self Checks & Check answer sheets on "Define Tasks of the Project"	2. Read Information 1: Use Model Form 3. Answer Self-check : Use Model Form 4. Check your answer with Answer key : Use Model Form
6. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job Sheet 3 : Create Django Model Form

Information sheet 3: Use Model Form

Learning Objective:

After completion of this Information sheet , the learners will be able to explain, define and interpret the following contents:

3.1. Model form is created
3.2. Model form is rendered
3.3. Model form is validated and saved

3.1. Model Form is Created

A **Model Form** in Django is a form that is directly tied to a model. It automatically generates form fields based on the fields of the associated model, making it easier to create forms for models.

Django provides the ModelForm class, which is used to create forms that will save and validate data directly into the corresponding database table.

Creating a Model Form

Consider a model Book that represents a book in the database. We can create a BookForm that will be used for adding or editing book records.

Example: Creating a Model Form for a Book Model

```
# models.py

from django.db import models



class Book(models.Model):

    title = models.CharField(max_length=100)

    author = models.CharField(max_length=100)

    published_date = models.DateField()

    genre = models.CharField(max_length=50)
```

```
    def __str__(self):
        return self.title
```

Now, create a model form for this Book model.

```
# forms.py

from django import forms

from .models import Book


class BookForm(forms.ModelForm):
    class Meta:
        model = Book
        fields = ['title', 'author', 'published_date', 'genre']
```

Here:

- The BookForm is a ModelForm that automatically generates fields for the title, author, published_date, and genre based on the Book model.
- The Meta class tells Django which model the form is tied to (Book) and which fields to include.

3.2. Model Form is Rendered

Once a ModelForm is created, it can be rendered in an HTML template. Django automatically generates the form fields for the model based on the ModelForm definition.

Rendering the Model Form in a Template

To render the form in a template, you first need to create a view that passes the form to the template.

```
# views.py

from django.shortcuts import render

from .forms import BookForm
```

```
def create_book(request):
    form = BookForm()
    return render(request, 'create_book.html', {'form': form})
```

In this view:

- We instantiate the BookForm.
- The form is passed to the create_book.html template for rendering.

Template to Render the Form

```
<!-- create_book.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Create a New Book</title>
</head>
<body>
    <h1>Create a New Book</h1>
    <form method="POST">
        {% csrf_token %}
        {{ form.as_p }}  <!-- Render the form fields -->
        <button type="submit">Save</button>
    </form>
</body>
</html>
```

Here:

- {{ form.as_p }} automatically renders the form fields as <p> elements.
- The POST method is used for submitting the form.
- {% csrf_token %} is required for security to prevent CSRF (Cross-Site Request Forgery) attacks.

Rendered HTML:

```
<form method="POST">

  <p><label for="id_title">Title:</label> <input type="text" name="title" maxlength="100" required id="id_title"></p>

  <p><label for="id_author">Author:</label> <input type="text" name="author" maxlength="100" required id="id_author"></p>

  <p><label for="id_published_date">Published Date:</label> <input type="date" name="published_date" required id="id_published_date"></p>

  <p><label for="id_genre">Genre:</label> <input type="text" name="genre" maxlength="50" required id="id_genre"></p>

  <button type="submit">Save</button>

</form>
```

3.3. Model Form is Validated and Saved

When the form is submitted, Django will validate the form data based on the model's field types and constraints (e.g., max_length, required). If the data is valid, it can be saved directly to the database.

Validating and Saving the Model Form

In the view, you can check if the form is valid using the is_valid() method. If the form is valid, the save() method is used to save the form data to the database.

```
# views.py

from django.shortcuts import render, redirect

from .forms import BookForm



def create_book(request):
```

```
    if request.method == 'POST':
        form = BookForm(request.POST)
        if form.is_valid():  # Validate the form
            form.save()  # Save the form data to the database
            return redirect('book_list')  # Redirect to a new page (e.g., book list)
    else:
        form = BookForm()

    return render(request, 'create_book.html', {'form': form})
```

In this view:

- When the form is submitted (POST method), the request.POST data is passed to the BookForm.
- form.is_valid() checks if the form data meets the model field requirements.
- If valid, form.save() saves the form data as a new Book record in the database.
- After saving, the user is redirected to a list of books (you can define the book_list URL to show all books).

Example: Book List View and Template

```
# views.py
from django.shortcuts import render
from .models import Book

def book_list(request):
    books = Book.objects.all()
    return render(request, 'book_list.html', {'books': books})
<!-- book_list.html -->
<h1>All Books</h1>
<ul>
```

```
   {% for book in books %}

      <li>{{ book.title }} by {{ book.author }} (Published on {{ book.published_date }})</li>

   {% endfor %}

</ul>
```

When the form is successfully submitted and saved, the user is redirected to this book_list.html page that displays all books.

Self-Check Sheet 3: Use Model Form

1. Which of the following statements is TRUE about model forms in Django?
Answer:
A. They are a standalone form and do not require a model.
B. They are a way to create forms based on existing Django models.
C. They cannot be used to process and save data to the database.
D. They require manually writing all the form fields.

2. What are the two required options in the Meta class of a model form?
Answer:
A. fields and exclude
B. model and fields
C. model and exclude (you can only choose one)
D. model and widgets

3. How does a model form field determine its data type?
Answer:
A. It is always the same data type as the corresponding model field.
B. It is based on a pre-defined mapping between model and form field data types.
C. It is manually specified in the Meta class of the model form.
D. It is inherited from the form class used to create the model form.

4. What is the purpose of the to_field_name **option in a ModelChoiceField?**
Answer:
A. To specify the label for the entire field.
B. To customize the text displayed for each option in the select element.
C. To filter the queryset used to populate the field.
D. To change the underlying value stored for each option. (Caution advised)

5. How can you add a new form field to a model form that is not present in the underlying model?
Answer:
A. You cannot add new form fields to model forms.
B. By defining the new field in the Meta class with the fields option.
C. By overriding a default model field with a new form field definition.
D. You can simply add the new field directly to the form class definition.

Answer Key 3: Use Model Form

1. Which of the following statements is TRUE about model forms in Django?
Answer: B. They are a way to create forms based on existing Django models.

2. What are the two required options in the Meta class of a model form?
Answer: B. model and fields

3. How does a model form field determine its data type?
Answer: B. It is based on a pre-defined mapping between model and form field data types.

4. What is the purpose of the to_field_name **option in a ModelChoiceField?**
Answer: D. To change the underlying value stored for each option. (Caution advised)

5. How can you add a new form field to a model form that is not present in the underlying model?
Answer: D. You can simply add the new field directly to the form class definition.

Job Sheet 3 : Create Django Model Form

UoC Cover

OU-ICT-WADP-04- L4-V1: Process Forms

Working Procedure / Steps

1. **Create a Django Model:**
    - Define the data structure for your application using Django models. This involves creating a Python class that inherits from models.Model and specifies fields using various model field types like CharField, EmailField, etc.
2. **Create a Model Form:**
    - Inherit from django.forms.ModelForm to create a form class that automatically maps your model fields to form fields.
    - Use the Meta class to configure the model form:
        - model: Specify the Django model this form is based on.
        - fields: Indicate which model fields to include in the form (optional, defaults to all).
        - exclude: Specify which model fields to exclude from the form (optional).
3. **Process the Model Form:**
    - Handle form submissions in views.py using methods like GET and POST.
    - In the POST method:
        - Create a ModelForm instance with the submitted data (request.POST).
        - Validate the form data using is_valid().
        - If valid, save the form data to the database using the form's save() method.
        - If invalid, display error messages and re-render the form for user corrections.

Specification Sheet 3 : Create Django Model Form

Technical Requirements

- **Programming Language:** Python 3.7 or later
- **Framework:** Django 3.2 or later
- **Text Editor/IDE:** Visual Studio Code, PyCharm, Sublime Text (or any preferred code editor)

Tools and Equipment

- **Computer:** A computer with sufficient processing power, RAM, and storage.
- **Internet Connection:** A reliable internet connection for downloading and installing software, accessing online resources, and deploying the application (if applicable).
- **Development Environment:** A virtual environment is recommended to isolate project dependencies.

Learning Outcome 4: Use Django Custom Form Fields and Widgets

Assessment Criteria	1. Custom Form Fields are created 2. Built-In Widgets are customized 3. Custom Form Widgets are created 4. Custom Form Widget Configuration Options are created
Conditions and Resources	1. Real or simulated workplace 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil, Eraser 7. Internet facilities 8. White board and marker 9. Audio Video Device
Contents	1. Custom Form Fields 2. Built-In Widgets 3. Custom Form Widgets 4. Custom Form Widget Configuration Options
Activities/job/Task	1. Create a Django Form with Custom Fields and Widgets
Training Methods	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Assessment Methods	Assessment methods may include but not limited to 1. Written Test 2. Demonstration 3. Oral Questioning 4. Portfolio

Learning Experience 4: Use Django Custom Form Fields and Widgets

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials 'Use Django Custom Form Fields and Widgets'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Define Tasks of the Project"	2. Read Information sheet 4: Use Django Custom Form Fields and Widgets 3. Answer Self-check 4: Use Django Custom Form Fields and Widgets 4. Check your answer with Answer key 4: Use Django Custom Form Fields and Widgets
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job Sheet 4: Create a Django Form with Custom Fields and Widgets

Information sheet 4: Use Django Custom Form Fields and Widgets

Learning Objective:

After completion of this Information sheet , the learners will be able to explain, define and interpret the following contents:

4.1. Custom Form Fields
4.2. Customize Built-In Widgets are customized
4.3. Custom Form Widgets
4.4. Custom Form Widget Configuration Options

4.1. Create Custom Form Fields

Django forms allow developers to define fields, which are typically linked to models. However, there are scenarios where you may need to create custom fields, particularly when the built-in fields do not meet your requirements. Django allows you to create **custom form fields** that can perform specific validation or functionality tailored to your needs.

Creating a Custom Form Field

To create a custom form field, you need to subclass forms.Field and implement your custom behavior, such as validation or data processing.

Example: Custom Form Field for a Date Range

Let's say we need to create a custom field for a date range where the user provides a start and end date in a specific format. We can create a custom form field to handle this.

```
# forms.py
from django import forms
from datetime import datetime

class DateRangeField(forms.Field):
    def to_python(self, value):
        # Custom logic to handle date range input
        try:
            start_date, end_date = value.split(' - ')
            start_date = datetime.strptime(start_date, '%Y-%m-%d')
            end_date = datetime.strptime(end_date, '%Y-%m-%d')
```

```
            return (start_date, end_date)
        except ValueError:
            raise forms.ValidationError('Enter a valid date range in the format YYYY-MM-DD - YYYY-MM-DD')

    def validate(self, value):
        super().validate(value)
        start_date, end_date = value
        if start_date > end_date:
            raise forms.ValidationError('Start date cannot be later than end date.')
```

Usage in a Form:

```
class MyForm(forms.Form):
    date_range = DateRangeField(label='Select Date Range')
```

- **to_python()** converts the input value into a Python object (a tuple of two dates in this case).
- **validate()** checks whether the input is valid, ensuring that the start date is before the end date.

Form Rendering and Output:

When the form is displayed in a template, users will be able to input a date range, which will be validated by the custom form field.

4.2. Customize Built-In Widgets

Django provides built-in widgets for handling form input (e.g., text fields, checkboxes, date pickers, etc.). Sometimes, you may need to customize these widgets to modify their appearance, behavior, or to integrate with other tools like JavaScript libraries.

A widget is responsible for rendering the HTML for a form field.

Example: Customizing the TextInput Widget

You can customize built-in widgets by setting attributes or using a custom template.

```
class CustomForm(forms.Form):
    name = forms.CharField(widget=forms.TextInput(attrs={'placeholder': 'Enter your name', 'class': 'form-control'}))
```

In this example:

- The TextInput widget is customized to include a placeholder and a CSS class (form-control) for styling.
- When rendered, the HTML for the name field will include the placeholder and CSS class.

Rendered HTML:

```
<input type="text" name="name" placeholder="Enter your name" class="form-control">
```

4.3. Custom Form Widgets

Custom form widgets allow you to control how a form field is rendered and how its value is processed. You can create a custom widget by subclassing forms.Widget and defining how to render the widget (render()) and how to convert the input data into Python values (value_from_datadict()).

Example: Custom Widget for a Color Picker

Let's create a custom widget for a color picker that renders an HTML input of type color.

```
# forms.py
from django import forms

class ColorPickerWidget(forms.Widget):
    def render(self, name, value, attrs=None, renderer=None):
        # Custom rendering of the color picker input
        return f'<input type="color" name="{name}" value="{value}" />'

    def value_from_datadict(self, data, files, name):
        # Get the color value from the submitted data
        return data.get(name, None)
```

Using the Custom Widget in a Form:

```
class MyForm(forms.Form):
    favorite_color = forms.CharField(widget=ColorPickerWidget())
```

This custom widget renders a color picker input field that allows the user to select a color. When the form is submitted, the selected color is retrieved and returned as a string (e.g., #ff5733).

Rendered HTML:

```
<input type="color" name="favorite_color" value="#ff5733" />
```

4.4. Custom Form Widget Configuration Options

When customizing widgets, you can define several configuration options such as attributes, validation logic, and the form of the rendered HTML. You can pass these options to the widget to control its behavior.

Example: Custom Widget with Additional Attributes

Suppose you want to create a custom widget for a file input field that includes a custom accept attribute and adds some styling.

```
class CustomFileInput(forms.FileInput):
    def __init__(self, attrs=None):
        # Add custom attributes to the file input widget
        default_attrs = {'accept': 'image/*', 'class': 'custom-file-input'}
        if attrs:
            default_attrs.update(attrs)
        super().__init__(attrs=default_attrs)
```

Using Custom Widget in a Form:

```
class FileUploadForm(forms.Form):
    file = forms.FileField(widget=CustomFileInput())
```

In this example, we customize the file input widget to accept only image files (accept="image/*") and apply a custom CSS class (custom-file-input).

Rendered HTML:

```
<input type="file" name="file" accept="image/*" class="custom-file-input">
```

Self-Check Sheet 4: Use Model Form

1. Which of the following statements is true about customizing Django form fields?

Answer:

- A. You can completely rewrite the form field's validation logic.
- B. You can only make minor changes to the field's behavior.
- C. It's best suited for repetitive customizations to built-in fields.
- D. You cannot change the data type of the form field.

2. What is the first step to customizing Django's built-in widgets?

Answer:

- A. Modify the widget templates in the Django distribution directory. (This is not recommended)
- B. Create a custom widget class that inherits from a built-in widget class.
- C. Override the widget attributes in your form field definition.
- D. Include a third-party widget library in your project.

3. Where should you place your custom built-in widget templates?

Answer:

- A. In a directory listed in the DIRS setting of your project's TEMPLATES variable.
- B. In the Django distribution directory (django/forms/templates/django/forms/widgets/).
- C. In a separate app specifically for custom widgets.
- D. You cannot create custom built-in widget templates.

4. What is the benefit of using a custom form widget over overriding widget attributes?

Answer:

- A. Custom widgets cannot be reused across multiple forms.
- B. Custom widgets offer more control over the widget's behavior and appearance.
- C. Widget attributes are the only way to customize the HTML output of a widget.
- D. There is no benefit, they achieve the same result.

5. Which Django variable controls the form renderer used for finding and loading widget templates?

Answer:

- A. FORM_CLASS
- B. WIDGET_RENDERER
- C. TEMPLATE_DIRS
- D. FORM_TEMPLATES

6. What are some factors to consider when choosing the right approach for customizing built-in widgets?

Answer:

- A. The complexity of the customization and desired level of control.
- B. Whether you are using a third-party widget library.
- C. The specific built-in widget you are customizing.
- D. All of the above.

Answer Key 4: Use Model Form

1. Which of the following statements is true about customizing Django form fields?
Answer: D. You cannot change the data type of the form field.

2. What is the first step to customizing Django's built-in widgets?
Answer: D. Include a third-party widget library in your project.

3. Where should you place your custom built-in widget templates?
Answer: A. In a directory listed in the DIRS setting of your project's TEMPLATES variable.

4. What is the benefit of using a custom form widget over overriding widget attributes?
Answer: B. Custom widgets offer more control over the widget's behavior and appearance.

5. Which Django variable controls the form renderer used for finding and loading widget templates?
Answer: B. WIDGET_RENDERER **(Answer)**

6. What are some factors to consider when choosing the right approach for customizing built-in widgets?
Answer: A. The complexity of the customization and desired level of control.

Job Sheet 4: Create a Django Form with Custom Fields and Widgets

UoC Cover

OU-ICT-WADP-04- L4-V1: Process Forms

Working Procedure / Steps

1. **Identify Needs:**
    - Determine which form fields require custom validation or behavior (custom form fields).
    - Decide which built-in widgets need adjustments to appearance or behavior (customized built-in widgets).
    - Consider if any form fields require complete control over HTML and functionality (custom form widgets).

2. **Create Custom Form Fields (Optional):**
    - Inherit from a built-in form field (e.g., CharField, EmailField).
    - Override methods like clean() to implement custom validation logic.

3. **Customize Built-in Widgets (Optional):**
    - **Override Widget Attributes:** Pass keyword arguments to the form field constructor (e.g., attrs={'class': 'custom-class'}).
        - Modify attributes like class, style, readonly, etc.
    - **Override Templates:** Create custom templates for built-in widgets
        - Access widget data and attributes in the template.
        - Render the desired HTML structure.

4. **Create Custom Form Widgets (Optional):**
    - Inherit from a built-in widget class (e.g., widgets.TextInput).
    - Override methods like render() to control HTML output and behavior.
    - Example:
5. **Use Custom Fields and Widgets in Forms:**
    - Include custom fields and widgets in your form class.

Specification Sheet 4: Create a Django Form with Custom Fields and Widgets

Technical Requirements

- **Programming Language:** Python 3.7 or later
- **Framework:** Django 3.2 or later
- **Text Editor/IDE:** Visual Studio Code, PyCharm, Sublime Text (or any preferred code editor)

Tools and Equipment

- **Computer:** A computer with sufficient processing power, RAM, and storage.
- **Internet Connection:** A reliable internet connection for downloading and installing software, accessing online resources, and deploying the application (if applicable).
- **Development Environment:** A virtual environment is recommended to isolate project dependencies.
- **Web Server (Optional):** If deploying your application, you'll need a web server like Apache or Nginx.

Reference

1. https://forum.djangoproject.com/
2. https://www.webforefront.com/django/
3. geeksforgeeks

Review of Competency

Below is yourself assessment rating for module “Process Forms”

Assessment of performance Criteria	Yes	No
Form is initialized with fields and forms using init method	☐	☐
Django form field types are set.	☐	☐
Field Layout Values are initiated.	☐	☐
Form is Rendered	☐	☐
From data is sent with post request	☐	☐
Form data is validated	☐	☐
Data is saved to database	☐	☐
Partial form is used	☐	☐
Model form is created	☐	☐
Model form is rendered	☐	☐
Model form is validated and saved	☐	☐
Custom Form Fields are created	☐	☐
Built-In Widgets are customized	☐	☐
Custom Form Widgets are created	☐	☐

I now feel ready to undertake my formal competency assessment.

Signed:

Date:

Development of CBLM

The Competency based Learning Material (CBLM) of '**Process Forms**' (Occupation: Web Application Development with Python , Level-4) for National Skills Certificate is developed by NSDA with the assistance of SAMAHAR Consultants Ltd.in the month of June, 2024 under the contract number of package SD-9C dated 15th January 2024.

SL No.	**Name and Address**	**Designation**	**Contact Number**
1	Khan Mohammad Mahmud Hasan	Writer	Cell: 01714087897 Email: kmmhasan@gmail.com
2	A K M Mashuqur Rahman Mazumder	Editor	Cell: 01676323576 Email : mashuq.odelltech@odell.com.bd
3	Khan Mohammad Mahmud Hasan	Co-Ordinator	Cell: 01714087897 Email: kmmhasan@gmail.com
4	Md. Saif Uddin	Reviewer	Cell:01723004419 Email: engrbd.saif@gmail.com