Unit_04 – Forms, User Input, Authentication and User Management

Building forms using Django's

- Django provides a Form class which is used to create HTML forms. It describes a form and how it works and appears.
- It is similar to the ModelForm class that creates a form by using the Model, but it does not require the Model.
- Each field of the form class map to the HTML form <input> element and each one is a class itself, it manages form data and performs validation while submitting the form.
- Django's form functionality cam simplify and automate vast portions of work like prepared for display in a form, rendered as HTML, edit using a convenient interface, returned to the server, validated and cleaned up etc. and can also do it more securely than most programmers would be able to do in code they wrote themselves.

Django handles three distinct parts of the work involved in forms:

- Preparing and restricting data to make it ready for rendering
- Creating HTML forms for the data
- Receiving and processing submitted forms and data from the client.

- To create Django form we have to create a new file inside the application folder let's say file name is forms.py. Now we can write below code inside the forms.py to create a form.

Syntax:

```
from django import forms
class FormClassName(forms.Form);
    label=forms.FieldType()
    label=forms.FiledType(label='display_label')
```

Example: (forms.py)

```
from django import forms
class StudentRegistration(forms.Form):
    name = forms.CharField()
    email = forms.EmailField()
```

A "StudentRegistration" form is created that contains two fields

- Charfield is a class and used to create an HTML text input component in the form.
- Emailfield is a class and used to create an Email Input component in the form.

Now, we need to instantiate the form in views.py file. See, the below code.

views.py

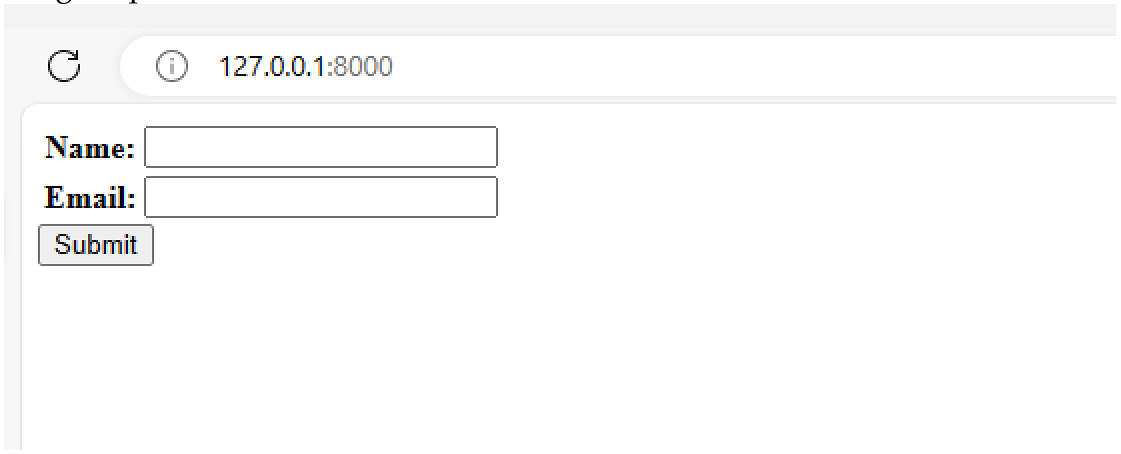
```
from django.shortcuts import render
from .forms import StudentRegistration
# Create your views here.
def showformdata(request):
    fm = StudentRegistration()
    return render(request, 'userregister.html',
{'form': fm})
```

Passing the context of form into ***"userregistrer"*** template that looks like this:

userregistrer.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewpoint" content="width=device-
width,initial-scale=1.0">
    <title>User Registration</title>
</head>
<body>
<table>
  {{form}}
</table>
<input type="submit" value="Submit">
</body>
</html>
```

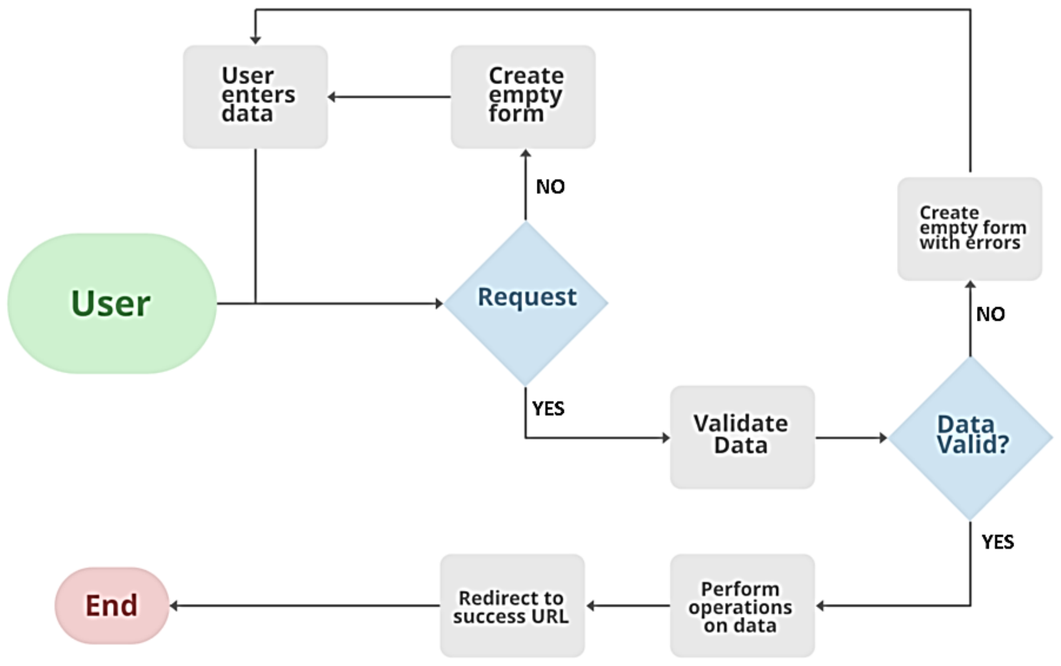
Run Server and access the form at browser by localhost:8000, and it will produce the following output.



Handling user input and validation

The primary functions of Django's form handling include:

1. Display the default form the first time it is requested by the user.
    - The form may contain blank fields if you're creating a new record, or it may be pre-populated with initial values (for example, if you are changing a record, or have useful default initial values).
    - The form is referred to as *unbound* at this point, because it isn't associated with any user-entered data (though it may have initial values).
2. Receive data from a submit request and bind it to the form.
    - Binding data to the form means that the user-entered data and any errors are available when we need to redisplay the form.
3. Clean and validate the data.
    - Cleaning the data performs sanitization of the input fields, such as removing invalid characters that might be used to send malicious content to the server, and converts them into consistent Python types.
    - Validation checks that the values are appropriate for the field (for example, that they are in the right date range, aren't too short or too long, etc.)
4. If any data is invalid, re-display the form, this time with any user populated values and error messages for the problem fields.
5. If all data is valid, perform required actions (such as save the data, send an email, return the result of a search, upload a file, and so on).
6. Once all actions are complete, redirect the user to another page.



Example:
forms.py

```
from django import forms

class StudentRegistration(forms.Form):
    name = forms.CharField()
    email = forms.EmailField()
```

views.py

```
from django.shortcuts import render
from .forms import StudentRegistration
from django.http import HttpResponse

def showformdata(request):
    if request.method=='POST':
        fm = StudentRegistration(request.POST)
        if fm.is_valid():
            print('Form Validated')
            print('Name:',fm.cleaned_data['name'])
            print('email:', fm.cleaned_data['email'])
    else:
        fm = StudentRegistration()
    return render(request,'userregistration.html',{'form':fm})
```

userregistration.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <form method="POST"> {% csrf_token%}
        {{form.as_p}}
        <input type="submit" value="Submit">
    </form>

</body>
</html>
```

Form widgets

- A widget is Django's representation of an HTML input element.
- The widget handles the rendering of the HTML, and the extraction of data from a GET/POST dictionary that corresponds to the widget.
- The HTML generated by the built-in widgets uses HTML5 syntax, targeting <!DOCTYPE html>.
- Whenever you specify a field on a form, Django will use a default widget that is appropriate to the type of data that is to be displayed.
- Each form field has an appropriate default widget class, but these can be overridden as required
- Form fields deal with the logic of input validation and are used directly in templates.
- Widgets deal with rendering of HTML form input elements on the web page and extraction of raw submitted data.

Default Widget in Form Fields

- Every field has a predefined widget, for example IntegerField has a default widget of NumberInput.

Custom Django form field widgets

- The default widget can be override for various purposes. We must specifically declare the widget we wish to attach to a field in order to override the default one.

Example

```
from django import forms
class StudentRegistration(forms.Form):
   title = forms.CharField(widget = forms.Textarea)
   description = forms.CharField(widget = forms.CheckboxInput)
    views = forms.IntegerField(widget = forms.TextInput)
    date = forms.DateField(widget=forms.SelectDateWidget)
```

Now, we need to instantiate the form in views.py file. See, the below code.
views.py

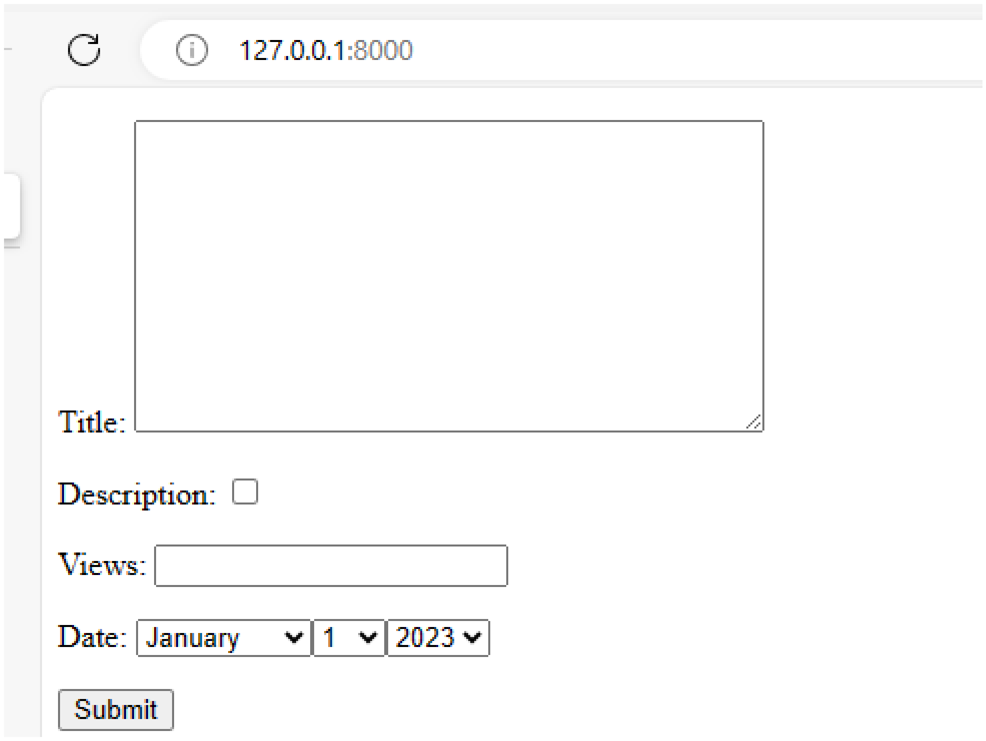
```
from django.shortcuts import render
from .forms import StudentRegistration
# Create your views here.
def showformdata(request):
    fm = StudentRegistration()
    return render(request, 'userregister.html', {'form':
fm})
```

Passing the context of form into ***"userregistrer"*** template that looks like this:

userregistrer.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewpoint" content="width=device-
width,initial-scale=1.0">
    <title>User Registration</title>
</head>
<body>
<table>
  {{form}}
</table>
<input type="submit" value="Submit">
</body>
</html>
```

Run Server and access the form at browser by localhost:8000, and it will produce the following output.



Handling form errors

- Built-in Form Field Validations in Django Forms are the default validations that come predefined to all fields.
- Every field comes in with some built-in validations from Django validators. Each Field class constructor takes some fixed arguments.
- The error_messages argument allow us to specify manual error messages for attributes of the field.
- The error_messages argument overrides the default messages that the field will raise.
- Pass in a dictionary with keys matching the error messages that the user want to override.

➢ Enter the following code into forms.py file of our app. We will be using ***CharField*** and ***Emailfield*** in our application.

forms.py

```
from django import forms

class StudentRegistration(forms.Form):
    name = forms.CharField(error_messages={
            'required': "Enter your Name"})
    email = forms.EmailField(error_messages={
            'required': "Enter your Email"})
```

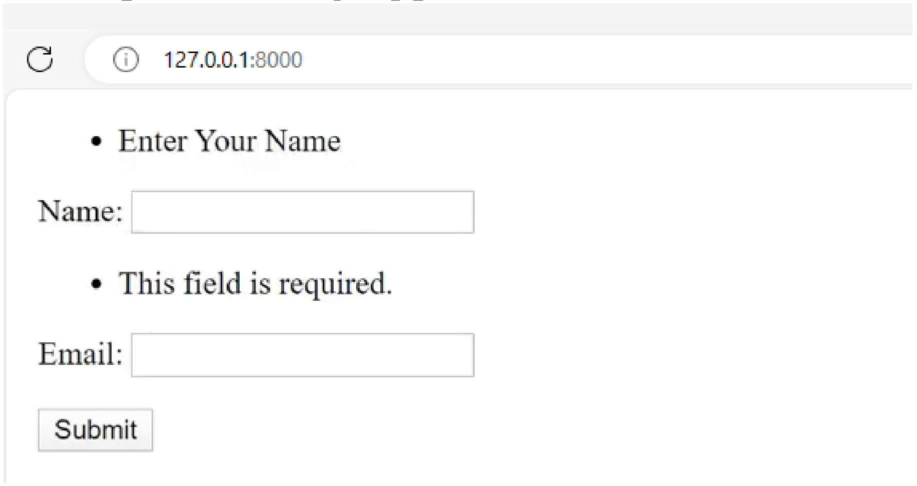
Now to render this form into a view we need a view and a URL mapped to that view. Let's create a view first in views.py of our app,

```
from django.shortcuts import render
from .forms import StudentRegistration
from django.http import HttpResponse

def showformdata(request):
    if request.method=='POST':
        fm = StudentRegistration(request.POST)
        if fm.is_valid():
            print('Form Validated')
            print('Name:',fm.cleaned_data['name'])
            print('email:', fm.cleaned_data['email'])
    else:
        fm = StudentRegistration()
    return
render(request,'userregistration.html',{'form':fm})
```

Let's run the server and check what has actually happened, Run

```
    Python manage.py runserver
```



- Enter Your Name

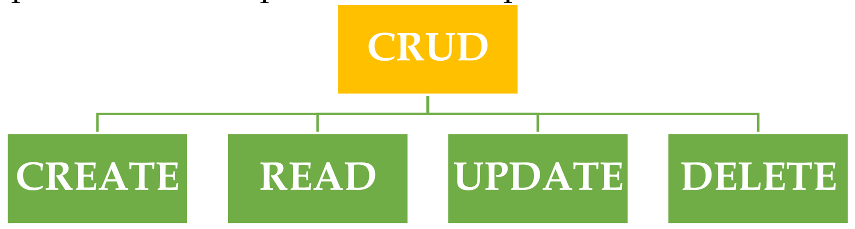
Name:

- This field is required.

Email:

CRUD operations

- CRUD is the acronym for ***CREATE, READ, UPDATE and DELETE***.
- These terms describe the four essential operations for creating and managing persistent data elements, mainly in relational and NoSQL databases.
- Applications that use persistent storage—those that retain data even after the machine is powered down—perform CRUD operations.



-
- These are different from operations on data stored in volatile storage, like Random Access Memory or cache files.
- CRUD is extensively used in database applications. This includes Relational Database Management Systems (RDBMS) like Oracle, MySQL, and PostgreSQL.
- It also includes NoSQL databases like MongoDB, Apache Cassandra, and AWS DynamoDB.

➢ **CREATE**
  - The CREATE operation adds a new record to a database.
  - In RDBMS, a database table row is referred to as a record, while columns are called attributes or fields.
  - The CREATE operation adds one or more new records with distinct field values in a table.

➢ READ
- READ returns records (or documents or items) from a database table (or collection or bucket) based on some search criteria.
- The READ operation can return all records and some or all fields.

➢ UPDATE
- UPDATE is used to modify existing records in the database.
- For example, this can be the change of address in a customer database or price change in a product database.
- Similar to READ, UPDATEs can be applied across all records or only a few, based on criteria.
- An UPDATE operation can modify and persist changes to a single field or to multiple fields of the record. If multiple fields are to be updated, the database system ensures they are all updated or not at all.

➢ DELETE
- DELETE operations allow the user to remove records from the database.

- A hard delete removes the record altogether, while a soft delete flags the record but leaves it in place.
- For example, this is important in payroll where employment records need to be maintained even after an employee has left the company.

How is CRUD performed in a database?

➢ In RDBMS, CRUD operations are performed through Structure Query Language (SQL) commands.

- The INSERT statement is used for CREATE:
```
    INSERT INTO <table name> VALUES (field value 1, field
value, 2…)
```
- The SELECT statement is used for READ:
```
    SELECT field 1, field 2, …FROM <table name> [WHERE
<condition>]
```
- The UPDATE statement is used for UPDATE:
```
    UPDATE <table name> SET field1=value1, field2=value2,…
[WHERE <condition>]
```
- The DELETE statement is used for DELETE:
```
    DELETE FROM <table name> [WHERE <condition>]
```

➢ CRUD operations in NoSQL databases will depend on the language of the specific database platform.

- CREATE
```
    db.collection.insertOne()or db.collection.insertMany().
```
- READ
```
    db.collection.find() or db.collection.findOne().
```
- UPDATE
```
    db.collection.updateOne(), db.collection.updateMany() , or
db.collection.replaceOne().
```
- DELETE
```
    db.collection.deleteOne() or db.collection.deleteMany().
```
