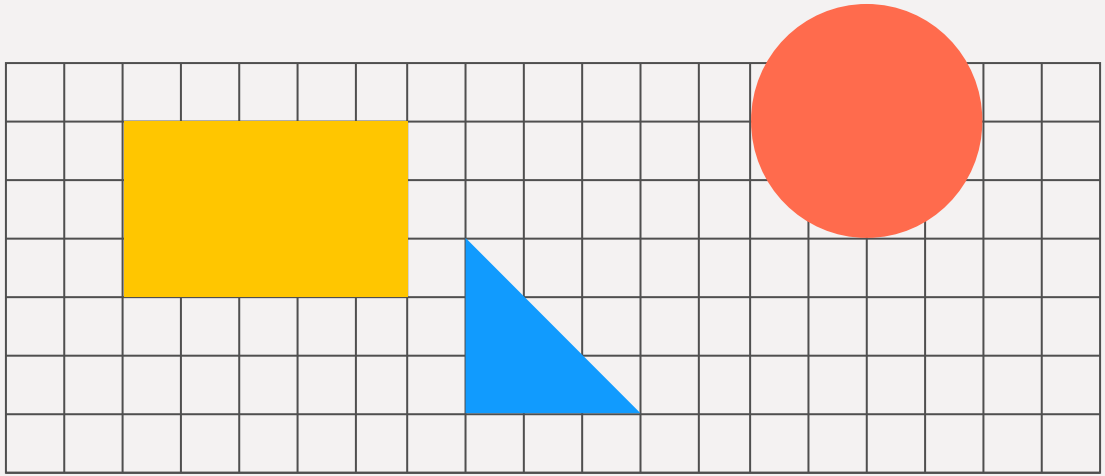
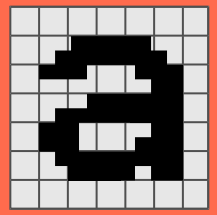
# Django Forms

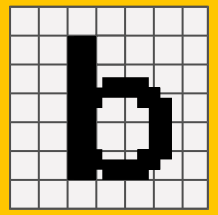
Ali Abrishami



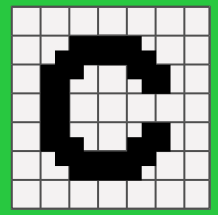
# In this Lecture You will...



Get to know Django Templates



Explore Django's Admin Panel

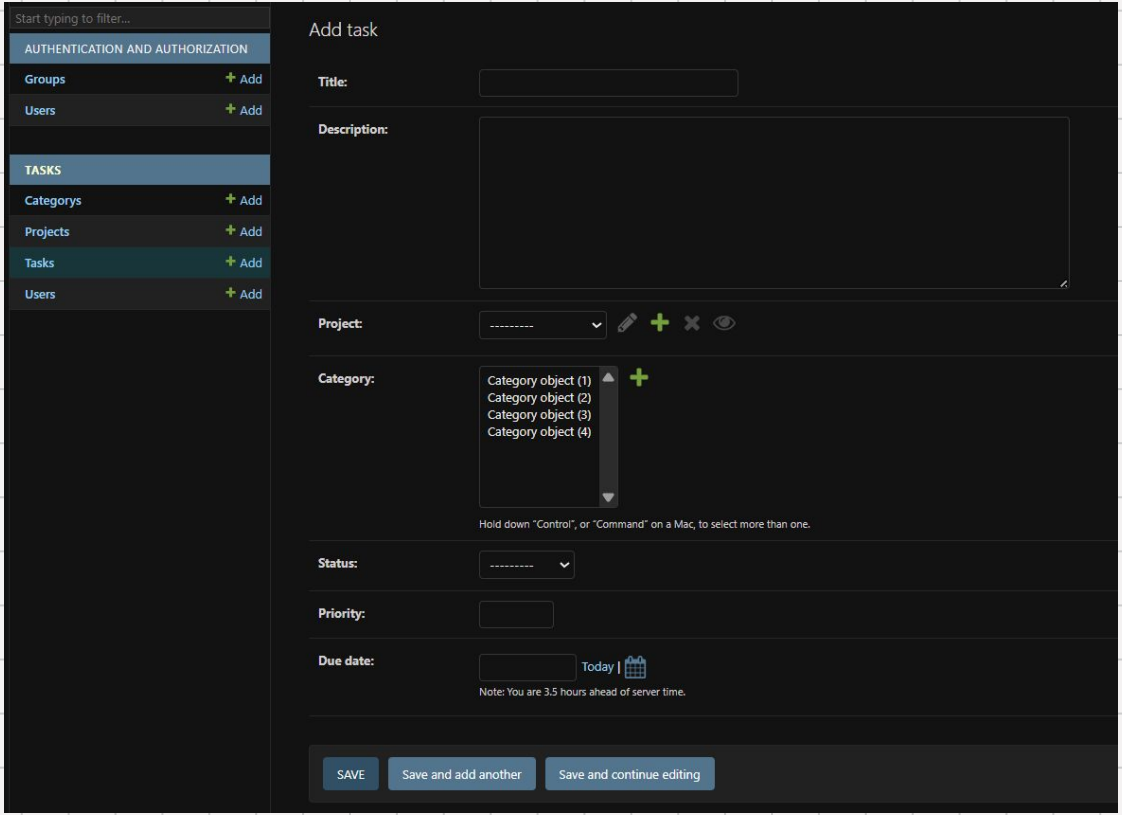


Learn Django development best practices

# Review HTML Form

- An HTML Form is a group of one or more fields/widgets on a web page, which can be used to collect information from users for submission to a server.
- Provide a structured way for users to submit data to the server using `<form>` and input fields. Containing at least one input element of `type="submit"`
- Use `action` (target URL) and `method` (`GET`/`POST`) to define how and where data is sent.
- Each field becomes a key–value pair in the request; names matter more than IDs.
- Submission triggers a full HTTP request, which Django turns into `request.GET`, `request.POST`, and `request.FILES`.

# A Sample Form for Task



# Parameters!

- Query parameters (`request.GET`): key–value pairs from the URL, typically used for filtering and simple inputs.
- Form parameters (`request.POST`): data sent via HTML forms using POST, including non-file fields.
- File parameters (`request.FILES`): uploaded files delivered as UploadedFile objects.
- Combined access: Django never merges GET/POST for you (you must decide which source is valid.)
- Encoding rules: parameters arrive URL-encoded or multipart-encoded. Django parses them automatically.

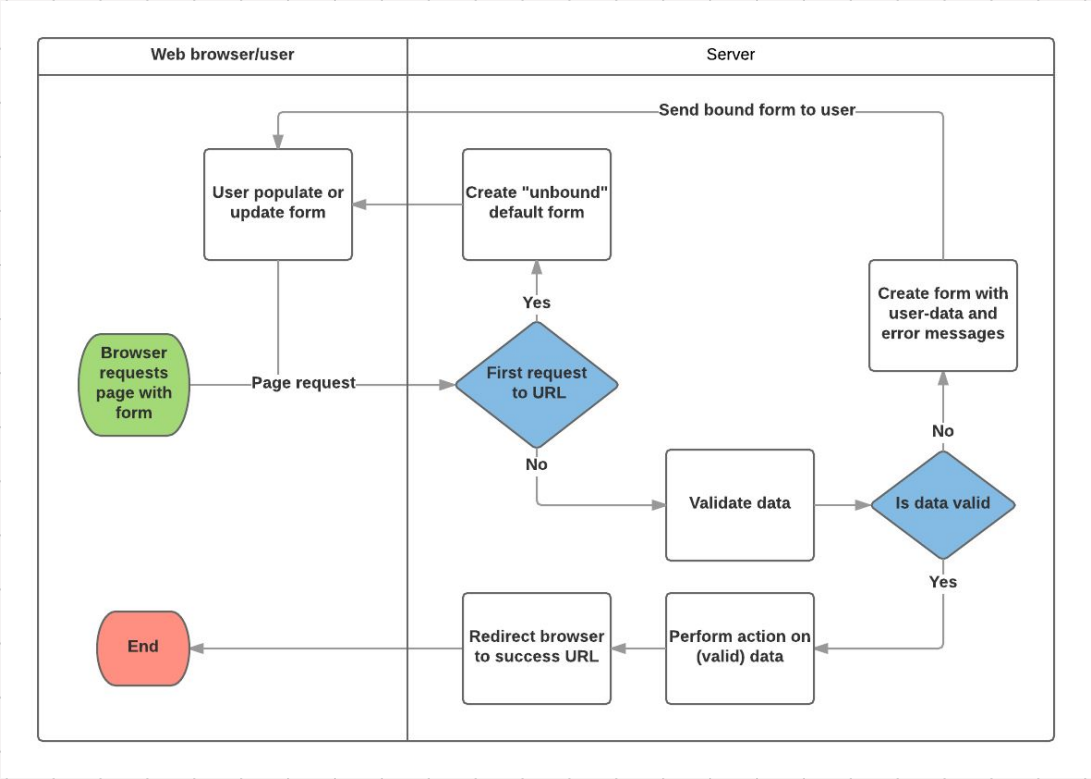
# Why Forms?

- Provide a structured channel for collecting user input and sending it to the server.
- Allow applications to **validate**, **sanitize**, and **normalize** input before using it.
- Reduce security risks by preventing malformed data, injection attempts, and unsafe types.
- HTML forms only describe how data is submitted. Django Forms define what valid data looks like.
- Django Forms integrate **validation**, **error messages**, **widgets**, **cleaned data**, and **safe binding** into one system.

# Django Form Handling

1. Display the **default form** the first time it is requested by the user.
2. **Receive data** from a submit request and **bind it** to the form.
3. **Clean** and **validate** the data.
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.

# Process Flowchart



# Django Form Classes

- Django offers `Form` for standalone input handling and `ModelForm` for generating forms directly from models.
- The choice depends on whether the data is independent of the database or meant to be saved as a model instance.
- A form class declares its **fields**, **widgets**, and **validation methods** in one coherent structure.
- **Field types** define the shape of expected data, while **validators** enforce rules and constraints.
- The validation pipeline normalizes input, raises meaningful errors, and produces clean, trustworthy data for the app.

# A Sample Django Form

```python
import datetime

from django.core.exceptions import ValidationError
from django import forms


class UpdateTaskDueDateForm(forms.Form):  3 usages
    new_due_date = forms.DateField(help_text="Enter new due date.")

    def clean_new_due_date(self):
        data = self.cleaned_data['new_due_date']
        # Check if a date is not in the past.
        if data < datetime.date.today():
            raise ValidationError('Invalid date - due date in past')
        # Check if a date is in the allowed range (+4 weeks from today).
        if data > datetime.date.today() + datetime.timedelta(weeks=4):
            raise ValidationError('Invalid date - due date more than 4 weeks ahead')
            # Remember to always return the cleaned data.
        return data
```

# The View

```
/task/{pk}/change-due-date/
def change_task_due_date(request, pk):  2 usages
    task_instance = get_object_or_404(Task, pk=pk)

    # If this is a POST request then process the Form data
    if request.method == 'POST':
        # Create a form instance and populate it with data from the request (binding):
        form = UpdateTaskDueDateForm(request.POST)

        # Check if the form is valid:
        if form.is_valid():
            task_instance.due_date = form.cleaned_data['new_due_date']
            task_instance.save()

            return HttpResponseRedirect(reverse('task-list'))

    # If this is a GET (or any other method) create the default form.
    else:
        proposed_date = datetime.date.today() + datetime.timedelta(weeks=3)
        form = UpdateTaskDueDateForm(initial={'new_due_date': proposed_date})

    context = {'form': form, 'task_instance': task_instance}
    return render(request, template_name: 'tasks/task_update.html', context)
```

# And the Template!

```
<h1>Renew: {{ task_instance.title }}</h1>
<p>Description: {{ task_instance.description }}</p>

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

# Built-in Form Fields

- Django provides typed fields (`CharField`, `IntegerField`, `BooleanField`, `DateField`, `FileField`, etc.) to match common input patterns.
- Each field knows how to **validate** raw input and **convert** it into a proper Python type.
- Choice-based fields (`ChoiceField`, `TypedChoiceField`, `MultipleChoiceField`) enforce controlled input.
- File-related fields manage uploads safely and integrate with `request.FILES`.

# Important Field Arguments

- Field.required
- Field.label
- Field.initial
- Field.widget
- Field.help_text
- Field.error_messages
- Field.validators

# Form Validation

- Django **validates** forms by checking each field's **type**, **constraints**, and **built-in** or **custom** validators.
- Field-level `clean_<name>()` methods refine or reject **individual** inputs.
- The form-wide `clean()` method handles **cross-field** logic and consistency checks.
- Errors are collected and attached to fields or the form, producing clear messages for users.
- Successful validation yields cleaned_data, a normalized and trustworthy representation of the input.

# Binding and Processing

- A form becomes **“bound”** when instantiated with `data` (and optionally `files`), enabling validation and error reporting.
- Unbound forms render **blank fields** and cannot be validated.
- The typical flow is: instantiate - bind data - call `is_valid()` - inspect errors or use `cleaned_data`.
- `request.POST` supplies form fields; `request.FILES` supplies uploads for file-capable forms.
- After successful validation, apps either persist the data (e.g., via a `ModelForm`) or process it in custom logic.

# Rendering Forms

- The `{% csrf_token %}` added just inside the form tags is part of Django's cross-site forgery protection.
- Django can render forms automatically using `{{ form }}`, `{{ form.as_p }}`, `as_table`, or `as_ul` for quick layouts.
- Each field renders through its widget, producing **HTML inputs** already tied to names and error states.
- **Errors** and **help text** appear alongside fields when rendering a bound form.
- Developers can also manually render each field for full layout control.
- Custom widgets, templates, and form themes allow consistent styling across an application.

# Widgets

- Widgets determine how a form field is rendered in HTML (text boxes, checkboxes, selects, file inputs, etc.).
- They handle **presentation** only; validation and data logic remain the field's responsibility.
- Most widgets map directly to HTML input types but allow customization via attributes.
- They support styling, placeholders, CSS classes, and custom templates.
- Widgets can be replaced per-field or globally to control form appearance across an app.

# `ModelForm` Deep Dive

- A `ModelForm` links directly to a Django model, generating fields and validation rules from the model's schema.
- The inner `Meta` class declares the target model and which fields to include or exclude.
- Validation integrates seamlessly with model constraints, while custom form methods (`clean()`, `clean_<field>()`) allow finer control.
- Calling `form.save()` constructs or updates a model instance, with `commit=False` giving room for extra manipulation.
- Ideal for CRUD workflows since it reduces duplicate definitions, enforces consistency, and keeps logic close to the data model.

# Form Helper Features

- Automatic error handling that attaches field-specific and form-wide messages for clean UI feedback.
- Built-in help texts, labels, and initial values to guide users and shape form behavior.
- Field and widget attributes (`attrs`) for quick customization of placeholders, CSS classes, and HTML properties.
- Built-in CSRF protection when rendering forms in templates, safeguarding POST submissions.
- Built-in management of prefixes, file handling, and multi-form scenarios, simplifying complex form flows.

# Handling File Uploads

- Forms must use `enctype="multipart/form-data"` for sending files.
- Uploaded files appear in `request.FILES` as `UploadedFile` objects.
- `FileField` and `ImageField` handle type checks and basic validation.
- Django stores uploads temporarily, then saves them to your `MEDIA_ROOT`.
- Proper `MEDIA_URL`/`MEDIA_ROOT` config controls storage and serving.

# Security & Best Practices

- **Always** validate and sanitize user input; never trust raw request data.
- Use CSRF protection for all POST forms to prevent cross-site request forgery.
- **Limit** accepted file types and sizes to avoid malicious uploads and resource abuse.
- Avoid exposing sensitive fields; whitelist form fields instead of trusting client-side data.
- Handle errors gracefully and avoid leaking internal details in validation messages.