Learning Objectives

- **Render a Django form**
- **Setup a Django project to use Crispy Forms**
- **Use Crispy on an existing form**
- **Use `FormHelper` to customize a form**
- **Simplify a form with the `crispy` template tag**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

Forms Part 1

Crispy Forms

Bootstrap Forms

There was one part of Boostrap that we didn't cover: the form library. Bootstrap provides its own styling for HTML form elements to give them a consistent cross-browser look and a nicer aesthetic. They are applied to forms elements/inputs similar to how other Bootstrap styling is applied: by adding classes to existing HTML elements. For example, a "normal" form input might be composed like this (this is an example, do not enter the code into the IDE):

```
<label for="id_first_name">First Name:</label>
<input type="text" name="first_name" id="id_first_name">
```

To convert this to a Bootstrap form, we just need a couple of classes added:

```
<label for="id_first_name" class="form-label">First Name:
        </label>
<input type="text" name="first_name" id="id_first_name"
        class="form-control">
```

Bootstrap forms are extensive so rather than try to explain them all here, we recommend checking out the forms overview documentation. Once you've had a browse of the documentation, and have an idea of how the things like labels, layout and help text work, then return here and we'll continue on with using Bootstrap forms with Django.

Rendering Django Forms

There are a few ways that Django forms can be rendered. As you may know, `Form` subclasses have three built-in methods to render the entire form: `as_table`, to render each form element as a table row; `as_p`, to render each element in `<p>` tags; and `as_ul`, to render inside a `<ul>` with each element in an `<li>`. Whichever of these methods you pick, since Django doesn't know about Bootstrap, it renders the inputs and labels without any `class` attributes.

You also have the option of rendering a single field. For example, a field called `first_name` could be rendered anywhere on the page by adding `{{ form.first_name }}` to your template. Unfortunately you still have the same problem: Django renders the field without Bootstrap classes.

Finally, you also have the option of accessing the individual attributes of a field and rendering them in HTML manually. So, you could render a Bootstrap field by doing something like this:

```
<label for="{{ form.first_name.id_for_label }}" class="form-
        label">{{ form.first_name.label }}</label>
<input type="text" name="{{ form.first_name.html_name }}" id="{{
        form.first_name.id_for_label }}" class="form-control">
```

This is definitely the most flexible way, but it can be hard to get right, and is a lot of work if you add/remove fields to your form or change their name. You could write a few templates and other things to help, but luckily, someone else has already done that for you.

Project Forms Setup

Django Crispy Forms is a third party library that provides filters, template tags, and a few Django form helpers, to make it easier to use forms with Bootstrap.

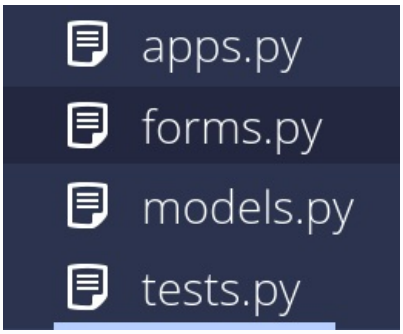
Before we implement it, let's build a form in the usual Django way. We'll then make it "crispy", to see the difference. We will add a comment form to the Post detail page. Submitting it will create a new Comment object associated with that post.

Try It Out

The first step is to set up the form class. Create a new file called `forms.py` in the `blog` app. Add a `ModelForm` subclass called `CommentForm`. Its model should be the `Comment`, and the only field that it should show is `content`.

▼ **Solution**

- This is what the `blog` directory looks like after adding `forms.py`.



forms.py file in the file tree

- Your `forms.py` should look like this:

```
from django import forms

from blog.models import Comment


class CommentForm(forms.ModelForm):
    class Meta:
        model = Comment
        fields = ["content"]
```

Next we'll update the `post_detail` view, to create a `Comment` using the posted form data. After fetching the `Post` in the view, and before returning the render, here's the code you should add (we'll explain it afterward):

Open views.py

```
    if request.user.is_active:
        if request.method == "POST":
            comment_form = CommentForm(request.POST)

            if comment_form.is_valid():
                comment = comment_form.save(commit=False)
                comment.content_object = post
                comment.creator = request.user
                comment.save()
                return redirect(request.path_info)
        else:
            comment_form = CommentForm()
    else:
        comment_form = None
```

First, we check if the user is active. Users who are inactive or aren't logged in (anonymous users) will fail this test and default to having the `comment_form` variable set to `None`.

Otherwise, we check the request method. If it's not `POST`, a blank `CommentForm` is created.

If it is a `POST`, then we create the `CommentForm` using the posted data. Then, if it's valid, we'll save the form, using the `commit=False` argument. This won't write the `Comment` object to the database, instead it will return it. We need to do this to set the other attributes on the `Comment` before saving.

The attributes we set are the `content_object` (the current `Post` being viewed); and the `creator` (the current logged in user). The `Comment` is then saved, and finally, we perform a `redirect` back to the current `Post` (this essentially just refreshes the page for the user so they see their new comment).

We have to also adjust the `render` call at the end of the view to include the `comment_form` variable:

```
    return render(
        request, "blog/post-detail.html", {"post": post,
        "comment_form": comment_form}
    )
```

And finally, make sure to import `redirect` from `django.shortcuts` and `CommentForm` from `blog.forms` at the start of the file.

```
from django.shortcuts import redirect
from blog.forms import CommentForm
```

Forms Part 2

Django Forms Part 2

Next we'll create a template to render the existing comments on a `Post`, as well as show the `CommentForm` to add a new one. Create a new file inside the `blango/templates/blog` directory named `post-comments.html`. You can copy and paste this content into it:

```
{% load blog_extras %}
<h4>Comments</h4>
{% for comment in post.comments.all %}
{% row "border-top pt-2" %}
    {% col %}
        <h5>Posted by {{ comment.creator }} at {{
        comment.created_at|date:"M, d Y h:i" }}</h5>
    {% endcol %}
{% endrow %}
{% row "border-bottom" %}
    {% col %}
        <p>{{ comment.content }}</p>
    {% endcol %}
{% endrow %}
{% empty %}
    {% row "border-top border-bottom" %}
        {% col %}
            <p>No comments.</p>
        {% endcol %}
    {% endrow %}
{% endfor %}
{% if request.user.is_active %}
{% row "mt-4" %}
    {% col %}
        <h4>Add Comment</h4>
        <form method="post">
            {% csrf_token %}
            {{ comment_form.as_p }}
            <p>
                <button type="submit" class="btn btn-
        primary">Submit</button>
            </p>
        </form>
    {% endcol %}
{% endrow %}
{% endif %}
```

In the top half of the file, we iterate over the existing comments, or if there are no comments, just output the message *No comments.*

The bottom half of the file has a `<form>` element containing the `{% csrf_token %}` template tag, the `comment_form` rendered with the `as_p` method, and a `submit` button. This is all wrapped in a check that the current user is active, just like the view.

The last file to edit is `post-detail.html`. All we need to do is include the `blog-comments.html` file, like this:

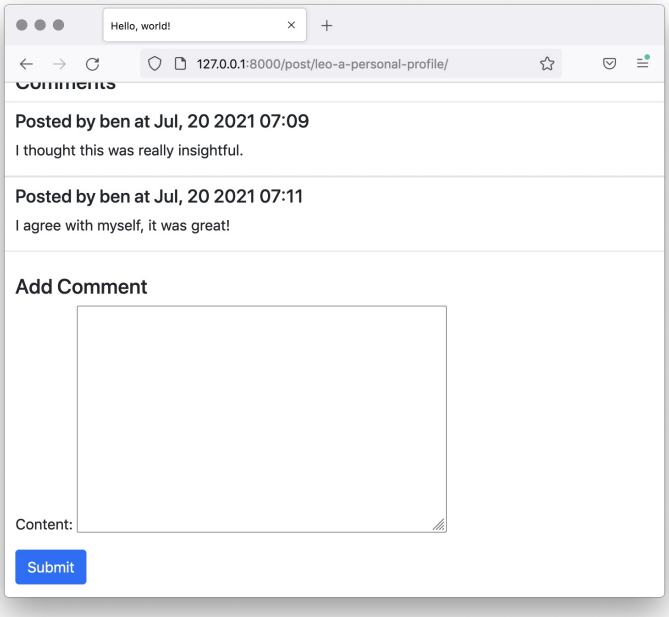
Open post-detail.html

```
{% include "blog/post-comments.html" %}
```

Place this above the row that contains the recent post template tag.

Now, start the Django dev server if it's not already running, then view any post detail page. You should be able to add Comments to a post and then see them show.

View Blog



Normal Django Form

▼ **Cannot add a comment?**

If you are not able to add a comment, follow these steps:

- Go to the admin panel by adding `/admin` to the URL
- Log in to the admin panel
- Refresh the Django project

- Click on `Read More`
- You should now be able to add a comment

Now, lets get back to Django Crispy Forms.

Crispy Forms

Django Crispy Forms Installation

As you might expect, Django Crispy Forms is installed using `pip`. The base package includes support (template packs) for Bootstrap versions 2, 3 and 4. Bootstrap 5 support is added by installing the package `crispy-bootstrap5`.

Since `crispy-bootstrap5` depends on `django-crispy-forms`, you only need to install `crispy-bootstrap5` with `pip` and `django-crispy-forms` will be installed automatically.

Try It Out

Get `crispy-bootstrap5` installed with `pip`

▼ **Solution**

Use `pip3` when installing Crispy Forms since we are starting the Django dev server with `python3`.

```
$ pip3 install crispy-bootstrap5
...
Successfully installed crispy-bootstrap5-0.4 django-crispy-
        forms-1.12.0
```

Now we need to make a few tweaks to settings get Django Crispy Forms set up. Open your `settings.py` and make the following changes:

Open settings.py

1. Add `"crispy_forms"` and `"crispy_bootstrap5"` to your `INSTALLED_APPS`.
2. Add the setting `CRISPY_ALLOWED_TEMPLATE_PACKS = "bootstrap5"`.
3. Finally, add the setting `CRISPY_TEMPLATE_PACK = "bootstrap5"`.

Django doesn't automatically render a form using Cripsy, you need to either use one of the Crispy filters or template tags.

The `crispy` filter is the easier method, so we'll try that out first. Open your `post-comments.html` file, and load the `crispy_forms_tags` tag library. That is, upload the `load` template tag to look like this:

Open post-comments.html

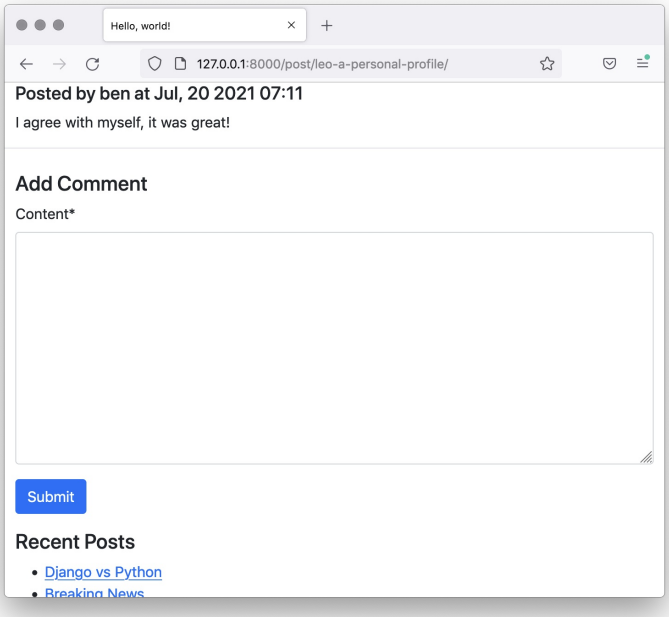
```
{% load blog_extras crispy_forms_tags %}
```

Then, instead of rendering the `comment_form` using `as_p`, we'll pass it through the `crispy` filter.

```
{{ comment_form|crispy }}
```

Refresh the page detail page, and see how the form has changed its appearance:

View Blog



Rendered Crispy Form

▼ **Cannot add a comment?**

If you are not able to add a comment, follow these steps:

- Go to the admin panel by adding `/admin` to the URL
- Log in to the admin panel
- Refresh the Django project

- Click on `Read More`
- You should now be able to add a comment

The field now has the label on the top, and if you view the page source you'll see the Bootstrap classes.

Using the `crispy` filter doesn't give an opportunity to customize the form rendering. We can use the `crispy` template tag along with some Crispy helper classes to move more of the output configuration into Python.

Template Tag & Helper Class

Crispy Template Tag and FormHelper Class

When rendering a form using just Django, it only renders the form's fields. We have to add the `<form>` element wrapping it, the `{% csrf_token %}` template tag and the submit button. When submitting files through the form, we also need to set the attribute `enctype="multipart/form-data"` on the form.

The `crispy` template tag, can take care of all of these things for us (including setting the `enctype` if the form contains `file` fields), but we might need to set some options using a `FormHelper` instance attached to the form.

`FormHelper` is imported from `crispy_forms.helper`. To use it, we should implement a form's`__init__` method and have it assign `self.helper` to a `FormHelper` instance. For our `CommentForm`, after doing that, it would look like this:

```
class CommentForm(forms.ModelForm):
    class Meta:
        model = Comment
        fields = ["content"]

    def __init__(self, *args, **kwargs):
        super(CommentForm, self).__init__(*args, **kwargs)
        self.helper = FormHelper()
```

Now that we have a `FormHelper` instance, how can we use it to customize the form? There are lots of attributes to set, and you can find them all at the `FormHelper` documentation. Some of the more common ones that you'd use are:

- `form_method`: Set the form method, `GET` or `POST`. It defaults to `POST`. If set to `POST`, then the `crispy` template tag will automatically render the CSRF token in the form.
- `form_action`: If you want the form to submit to a different page than the one on which it was loaded, you can set the URL, path, or URL name to this attribute.
- `form_id`: The value to set as the `id` attribute of the `<form>` tag.
- `form_class`: The value to set as the `class` attribute of the `<form>` tag.
- `attrs`: A dictionary of attributes to set on the `<form>` tag.

As well as setting attributes, we can also add extra inputs. Most of the time, you'd just use this to add a submit button to your form. Let's do that now.

Try It Out

Since we're just posting the form to the same page on which it was loaded, we don't actually need to set any of the `FormHelper` attributes. But we will use the `FormHelper.add_input()` method to add the submit button.

First, import the `Submit` and `FormHelper` classes.

```
from crispy_forms.layout import Submit
from crispy_forms.helper import FormHelper
```

Then, we'll instantiate them and add it to the form in one go. Add this inside the `__init__` method.

```
        self.helper = FormHelper()
        self.helper.add_input(Submit('submit', 'Submit'))
```

▼ **FormHelper**
The FormHelper can also be used to change the layout your form, for example, by wrapping your fields in fields sets or other holder tags, or to add accessory fields (like icons) to inputs. We won't be looking at these extra layout options, but if you think they'd be useful at customizing the layout of your form, you can check out the layout documentation.

That's all that we need to change in `forms.py`, so it can be saved. We can now return to `post-comments.html` and simplify it.

Open post-comments.html

You can remove the opening and closing `<form>` tags, and everything in between, including the `csrf_token` template tag, `comment_form` rendering and submit button.

Replace it all with a `crispy` template tag:

```
{%  crispy comment_form %}
```

Then return to the post detail page in your browser and refresh it. You should see no change, and you can try submitting another comment to confirm it still works.

View Blog

Conclusion

We're now done with Django Crispy Forms. You can see the advantage of the `crispy` form tag is that you can move all your form setup into the form class, and just have a single line in your template to render it with all the right options it requires. Since you do all your other customization of the form in Python (such as what fields you want to display) it makes sense to also choose how to display your buttons, form tag and other helper elements in Python too.

We're also done with module one of the course. Module two will look at setting up your app for deployment and discuss some of Django's built in security features.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish crispy forms"
```

- Push to GitHub:

```
git push
```