Lab 4 - Forms

Introduction

In this lab, we will utilise **[Django forms](https://docs.djangoproject.com/en/4.1/topics/forms/)** **(https://docs.djangoproject.com/en/4.1/topics/forms/)** to integrate user management into our Comment Store web application. After this lab, you will be able to:

1) build a simple Django project using forms for User Registration, Login, and Logout
2) use Bootstrap IntelliJ Plugin to enhance productivity by utilising live templates and template filters

Note:

- *Remember to clean and build your project before running it after you make any changes to your code.*
- *This lab will build upon the previous one.*

Adding User Registration

In this example, you will use forms to write a simple registration interface.

1. Open the CommentStore project on IntelliJ of **Lab 3** **(https://canvas.sussex.ac.uk/courses/28153/pages/lab-3-django-templates)** .

2. Create a new application named *register* using the *manage.py* file.
Type *makemigrations* command*, press Enter,* and then type *migrate.*

3. Create a super user account with username *webapps* and password *wa123456!* using the *manage.py* file.

4. Open the file *register/forms.py* (create the file if it doesn't exist).

5. Django comes with a ***UserCreationForm*** ***(https://docs.djangoproject.com/en/4.1/topics/auth/default/#django.contrib.auth.forms.UserCreationForm)*** form for creating a new user connected to the model ***User*** ***(https://docs.djangoproject.com/en/4.1/ref/contrib/auth/#django.contrib.auth.models.User)*** . However, the *UserCreationForm* only requires a *username*, *password1*, the initial password, and *password2*, the password one. On the *forms.py*, create a class called

RegisterForm that inherits from *UserCreationForm, and* add another field called *email*. It is also necessary to create a *Meta* class inside the *RegistrationForm* class in order to change the User model whenever we save something in this form. It is also necessary to define the fields property that enables layout where the fields should be, i.e., first is the *username*, then the *email*, followed by the *password1 and password2* fields.

```
class RegisterForm(UserCreationForm):
    email = forms.EmailField()

    class Meta:
        model = User
        fields = ["username", "email", "password1", "password2"]
```

Note: Don't forget to import *forms* from *django*, *UserCreationForm* from *django.contrib.auth.forms*, and *User* from *django.contrib.auth.models* in the *forms.py* file.

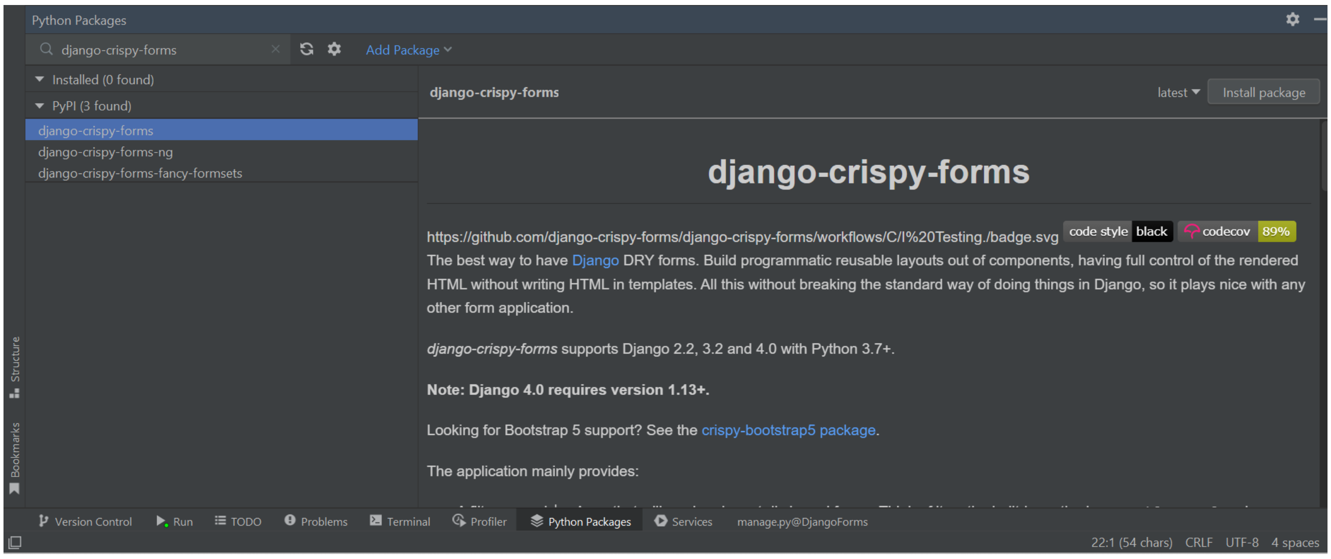
6. Now, we need to add a function inside the *views.py* file of our *register* app to render this page. Import *RegisterForm* from *forms.py* and *login* from *django.contrib.auth*. Then, write a new view function called *register_user*. Then, we need two if/else statements within the view function. The first checks if the form is being posted, while the second checks if the form is valid. If both are True, then the form information is saved under a user, the user is logged in, and "redirected" to the homepage showing a success message. Otherwise, if the form is not valid, an error message is shown. But if the request is not a POST, which means that the first if statement returned False, we need to render the empty form in the register template. Import *messages* from *django.contrib*.

7. Now, add a register path to the project's URLs (i.e., *CommentStore/urls.py*), so we can refer to it in the views. Since you already imported *views* from *commentstoreapp*, you need to import ***views as register_views*** from *register* (to differentiate from the previous import). This allows to differentiate views imported from different applications, namely *commentstoreapp* and *register*.

```
path("register/", register_views.register_user, name="register")
```

Follow IntelliJ's advice and fix all errors marked with red, and run your application.

8. We need to install crispy forms, which does some nice styling of our forms for us. Look for the **Python Packages (https://www.jetbrains.com/help/idea/installing-uninstalling-and-upgrading-packages.html)** tool window. Look for the *django-crispy-forms* and *crispy-bootstrap5* packages and install them.



django-crispy-forms

https://github.com/django-crispy-forms/django-crispy-forms/workflows/C/I%20Testing./badge.svg code style black codecov 89%
The best way to have Django DRY forms. Build programmatic reusable layouts out of components, having full control of the rendered HTML without writing HTML in templates. All this without breaking the standard way of doing things in Django, so it plays nice with any other form application.

django-crispy-forms supports Django 2.2, 3.2 and 4.0 with Python 3.7+.

Note: Django 4.0 requires version 1.13+.

Looking for Bootstrap 5 support? See the crispy-bootstrap5 package.

The application mainly provides:

Now that we have installed crispy forms, we need to add the following lines into **settings.py** to configure the CSS framework to use, i.e., **bootstrap (https://getbootstrap.com/docs/5.3/getting-started/introduction/)** . Add *'crispy_forms'* and *'crispy_bootstrap5'* to the *INSTALLED_APPS.* Add *CRISPY_TEMPLATE_PACK = "bootstrap5"* and CRISPY_ALLOWED_TEMPLATE_PACKS = "bootstrap5" to the bottom of the file.

9. We also need to update the *base.html* template as follows:

- Include required meta tags for proper responsive behaviour in mobile devices.

```
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=n
o">
```

- Include Bootstrap's CSS and JS. Specifically, Place the <*link*> tag in the <head> for our CSS and the <*script*> tag for our JavaScript bundle (including Popper for positioning dropdowns, poppers, and tooltips) before the closing </body>

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.
min.css" rel="stylesheet" integrity="sha384-GLhlTQ8iRABdZLl6O3oVMWSktQOp6b7In1Zl3/J
r59b6EGGoI1aFkw7cmDA6j6gD" crossorigin="anonymous">
```

```
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/js/bootstrap.
bundle.min.js" integrity="sha384-w76AqPfDkMBDXo30jS1Sgez6pr3x5MlQ1ZAGC+nuZB+EYdgRZg
iwxhTBTkF7CXvN" crossorigin="anonymous"></script>
```

- Include the *{% load crispy_forms_tags %}* tag before the *<head>.*

10. Now, create a *register* directory in the *templates* directory of the project to house our register templates, i.e., *register.html*, *login.html* and *logout.html*. First, create the

register.html template in this directory. You can copy-paste the *register.html* file at the end of this page. We need to apply the *crispy* **filter (https://docs.djangoproject.com/en/4.1/ref/templates/language/#filters)** to the *register_user* variable so that it becomes *{{ register_user|crispy }}* in the body.

11. Finally, re-run your application. Now, the appearance is much better!

Adding User Login

In our previous task, the views function automatically logged a user during its account creation. However, we want the user to have the ability to log in freely. Therefore, we need a login template, URL, and views function.

1. Create a *login.html* template in the *register* directory. You can re-use the *register.html* template structure, given that the only differences are the form and the link at the bottom. Specifically:

- Update the title tag text to "Login to an Account", the <h1> and <button> to "Login"
- In the form variable, replace the *register* to *login*, i.e., *{{ login_user|crispy }}*
- Add a new paragraph to redirect users that don't have an account:

```
<p class="text-center">Don't have an account? <a href="/register">Create an account
</a>.</p>
```

2. Add a login path to the URLs.

```
path("login/", views.login_user, name="login"),
```

3. Add a login view function to the *views.py* file. First, import ***AuthenticationForm (https://docs.djangoproject.com/en/4.1/topics/auth/default/#django.contrib.auth.forms.AuthenticationForm)*** from *django.contrib.auth.forms*. The class *AuthenticationForm* is the Django form for logging a user in. We need three if/else statements within the view function to write the view function. The first checks if the form is being posted, while the second checks if the form is valid. The last uses the Django authenticate() function. This function is used to verify user credentials (i.e., username and password) and return the correct User object stored in the backend. You can check all user accounts stored in the backend by login into the admin site. If the credentials are authenticated, the function will run Django login() to log in to the authenticated user. Otherwise, if the user is not authenticated, it returns a message to the user stating they entered an invalid username or password. The second else statement is used when the form is

invalid and returns a similar error message. If the request is not a POST, Django returns a blank login form.

4. Run your application.

Adding User Logout

Now, we need to handle user logout. The logout link will be placed in a **navigation header** **(https://getbootstrap.com/docs/5.0/components/navbar/)** (i.e., navbar), hence only visible to authenticated users. The navbar code can be found at the bottom of this page. You should add it to all HTML documents in the *templates/register* directory. For the logout, we need a URL and views function.

1. Similarly to the login task, you need to add a logout path to the URLs.

2. Add a logout view function to the *views.py* file. The view function uses the Django *logout()* function to log users out of their accounts and redirect them to the desired URL, e.g., the homepage or login URL. You should send a message stating that the logout was successful.

3. Run your application.

Adding Insert Comment Form

1. Open the file *commentstoreapp/forms.py* (create the file if it doesn't exist). Import forms from django.

2. Create a new class called *InsertNewComment* that receives the argument *forms.Form*. In it, define similarly to **Lab 2** **(https://canvas.sussex.ac.uk/courses/28153/pages/lab-2-django-views)** three new variables, namely *name*, *visit_date* and *comment_str*.

- *name* is a *CharField*, has a label "Insert a name:" and a maximum length of 100.
- *visit_date* is a *DateField* and has a label "Insert the date of visit:", and the *date input format* is ['%d/%m/%Y'].
- *comment_str* is a *CharField*, has a label "Insert a comment:" and a maximum length of 500.

3. Open the file *commentstoreapp/views.py* and remove the GET method, as it is no longer necessary. On the POST method, we need to create a new *InsertNewComment*

that receives as an argument a *request.POST*.

```
form = InsertNewComment(request.POST)
```

Then, we need to check if the form is valid. If True, for each field in the form, we need to validate and “clean” data before using it. Now, we can store it in our local database.

In the else statement, we do the same but without the argument. Lastly, to render as return a *commentstoreapp/comment.html* and pass as context the *form*.

```
return render(request, "commentstoreapp/comment.html", {"form": form})
```

4. Now that we have a form, we update the *commentstoreapp/comment.html* template and replace all the <label> and < input> tags with a variable containing our form and the filter *crispy*, i.e., *{{ form|crispy }}*. Also, include the *{% load crispy_forms_tags %}* tag.

5. Now, we need to update our template as follow: first, we need to check if the user is authenticated. If False, display the message "Comment Store Application Homepage. Please login to start ...". Otherwise, we need to check if the cmnt_list is empty. If True, display the message "There are no comments for user: <em>{{user.username}}". Otherwise, we need to display the table headers Name, Date and Comment, and then iterate over all comments in the list.

6. Run your application.

Optional Homework

Adding messages to the base.html template: Up until now, we can register, login and log out. However, if something goes wrong, we cannot know what, as no visual information is displayed. To add messages, we need to follow the following steps:

1. Open the *base.html* template and add the code below in the body before the block tags.

```
{% if messages %}
    {% for message in messages %}
            {{ message }}
        </div>
    {% endfor %}
{% endif %}
```

We are now able to see the messages but as simple text. To make the message more visually appealing, we can use **bootstrap alerts (https://getbootstrap.com/docs/5.3/components/alerts/)** . One example is the **dismissing (https://getbootstrap.com/docs/5.3/components/alerts/#dismissing)** alert.

2. Copy the HTML copy from the site and replace the message with our variable {{message}}.

```
<div class="alert alert-warning alert-dismissible fade show" role="alert">
  {{ message }}
  <button type="button" class="btn-close" data-bs-dismiss="alert" aria-label="Clos
e"></button>
</div>
```

3. Run your application.

Files

register.html (https://canvas.sussex.ac.uk/courses/28153/files/4226338?wrap=1) **(https://canvas.sussex.ac.uk/courses/28153/files/4226338/download?download_frd=1)**

navbar.html (https://canvas.sussex.ac.uk/courses/28153/files/4226715?wrap=1) **(https://canvas.sussex.ac.uk/courses/28153/files/4226715/download?download_frd=1)**

Lab3DjangoTemplatesSolution.zip (https://canvas.sussex.ac.uk/courses/28153/files/4667519?wrap=1) **(https://canvas.sussex.ac.uk/courses/28153/files/4667519/download?download_frd=1)**