

# Sharing Ideas: Bulletin Board

## goals

- Incrementally develop a web app using Django frameworks and *Python*
- Learn how web pages are connected and accessed



## description of web page

Bulletin Boards: Create an app as part of a website framework.

## Essential Questions

1. What are the similarities and differences in app creation for mobile devices and app creation for browsers?
2. Why is computer science considered a form of art and creative expression by many?
3. What are some essential operations you do over and over with lists or collections?

## essential Concepts

- Django Framework
- Cascading Style Sheets
- HTML
- Databases
- Conditionals and Modulo

# Bulletin Board Overview

The blog you created in the last activity represented a portal by which one user could create many blog posts. What if you wanted more than one user to be able to post?

In 1973, the first online bulletin board system was created in Berkeley, California. Rather than a one-to-many form of communication (*blog*), bulletin board apps (*bb*) allow many-to-many communication. Online communities can form around topics, and in many cases, can help each other more quickly as users communicate for all to see. A key feature of any bulletin board site is a user credentialing system, which allows access to only individuals with an account.

In this activity, you will create a user credentialing system as part of your bulletin board site. Users will be able to create accounts with unique user names and passwords. Posts to the bulletin board will be marked with the user name that created them. You will also learn how to use Django templates to simplify writing code for your views.

## What's New in this Activity?

- **User Credentialing System** User credentialing systems are so common, Django provides a *User* model for developers. Most sites need fields, like username and password, and the architecture to register people as new users.
- **Templates** HTML templates are a powerful tool for managing formatting and connecting web pages. Previously you created a page view and managed formatting within your code by applying HTML tags. You also added CSS templates to make your pages easier to read. Rather than manage links to other pages in your code with HTML anchor tags `<a href=''>`, you will now manage these in HTML templates.

## Getting Started

1. Navigate to Cloud9 and log in.
2. Create a new workspace using the naming convention set forth by your teacher. Example: *bb-lastnamefirstinitial*
3. Choose the **Django** template and create the workspace.
4. Click the **Run Project** button at the top right of the screen. Your framework is now running and can be shared or viewed through any web browser that has the URL.
5. Visit the URL for your new site.
6. Verify that your site presents the “It worked!” message.
7. Create the new app directory for 'bb' by typing the appropriate command in bash.
8. Create an index view with a temporary message.
9. Import *HttpResponse*.
10. Define a function to handle an *HttpResponse* request argument.
11. Create a *urls.py* file in your bb app.
12. Register the URL for your bb app.
13. Navigate to the correct URL in the web browser to verify the test message.

## Create and Import Models

In previous apps, you created models to capture information to add to a database. These models included Question, Choice, and Post.

- Each model had fields defined, like *post\_text*, *pub\_date*, and *post\_title*, which held specific data of Posts.
- Each field had functions that described the limits of what the field may contain.

In the *bb* app, you will import a commonly used model called *User*. It makes sense that Django would provide a model to handle user credentialing. Knowing how to set up this feature of a website is important because of how commonly used it is. Think of all the sites that need to register users and handle logins and logouts. Each user model will have fields that hold information about the user, such as username and password.

**Important:** As you create models, you may be able to reuse and modify some of your model code from previous apps.

14. In the *bb/models.py*, import *User* from *django.contrib.auth.models*. This will import the *User* model from the Django framework.
15. In the *bb/models.py*, create a *Post* class with the following fields. Set the attributes of each field to the same ones you used in the blog post app.
  - *post\_title* (max character field length = 1000)
  - *post\_text* (max character field length = 70)
  - *pub\_date* (set the *DateTimeField* to 'date published')

In a credentialing system, you can define what the program should do when a user is deleted from the database. When a user is deleted from the *bb* app, you will want all posts the user made to be deleted as well. However, if a post is deleted from the database, the user is still in the database. The user and the posts are an example of a many-to-one relationship. *ForeignKey* is an object that accepts other arguments to define the details of how the relation works. The *ForeignKey* function added to your *user* field will reference the *User* model that was imported.

16. Explore more information about the [User model](#).
17. Add a *user* field to the *Post* class.
  - Include a *models.ForeignKey* function with *User* as an argument.
  - Include an *on\_delete* argument with the value equal to *models.CASCADE*.
18. Check with your elbow partner to make sure your models look reasonable based on the other fields in your model and the fill-in-the-blank outline below.

```
user = ____ . ____ (____ , ____ = ____ . ____)
```

19. Add *bb* to your list of *INSTALLED\_APPS* in the *settings.py* file.

Make sure that the capitalization pattern remains the same as in previous apps.

20. Create a migration for the models in your *bb* app.
  - Check how you previously did migrations.
  - If it does not migrate, read and follow up on any errors.
  - Make sure you have saved all your files.
21. Run the migration that you just created (hint: *sqlmigrate*).
22. Register your model with your admin site.

## Set Up the Django Admin Portal

23. Create a superuser account for your admin site.
  24. Log in using the credentials that you just created.
  25. Create a new User object.
    - To make it easy to remember for now, you may want to use a similar *user\_name* and *password* that you just used to create your superuser account.
- Important:** You cannot have two superusers with the same name.
- You do not need to change any settings, just click through to save.
  - You should see two users, one with staff status (superuser) and one that you just created.
26. Create a new *Post* object under the superuser account in the admin portal.
  27. Create a new *Post* object under the base user account in the admin portal.
  28. Open the *views* file and update the index view to show the *Post* objects:
    - Check your *pollsite* code to see how Choice and Question were displayed to the user in the index.
    - Do not forget the necessary imports to call specific functions from the models file.
    - Add HTML tags in strings to help with readability to differentiate the user name and the Post texts.

## Index Template

In previous activities, you provided URLs to images and other files as typed strings in *Python* code. If the location of your files changed, you would have to manually edit the code. This is a disadvantage of **hard coding**, or fixing the values in a program in such a way that they must be modified in the code in the future. As an alternative, in this part of the activity, you will use **Django templates**. Templates are *Python* strings or text files that allow anyone with an understanding of HTML to maintain the appearance of a website that is otherwise programmed in *Python*.

29. In your *bb* directory, create a directory called “templates”.
30. Inside the *templates* directory, create a directory called “bb”.
31. Within *bb/templates/bb*, create a file called “index.html”.
32. Copy the following code into *index.html* and save the file:

```
{% if post_list %}
    {% for post in post_list %}
        <p>{{ post.post_title }} {{post.pub_date}}</p>
        <p>{{post.post_text}}</p>
    {% endfor %}
{% else %}
    <p>No posts are available.</p>
{% endif %}
```

The code above is an example of using Django templates to structure your HTML. The advantage of using a template is that templates allow for use of control flow structures like loops and conditionals.

33. Now you will connect your *views.py* code to this template.

- Within *bb/views.py*, from *django.template* import *loader*.
- In your *index* view, store a list of your Post objects in reverse order by *pub\_date* in a variable identified by *post\_list*.  
Hint: This is done in the blog app, if you need to review.
- Copy the following code into your index view function in *bb/views.py*:

```
template = loader.get_template('bb/index.html')
context = {
    'post_list': post_list,
}
return HttpResponse(template.render(context, request))
```

34. View your running index page.



**PLTW DEVELOPER'S JOURNAL** Make connections between the provided code and what you see on your index page.

## Connecting the Template to the Code

The template you copied into the *views.py* file can access values from the *Python* code in your app, as follows:

- Importing the loader from the Django template enables the *render* function.
- The *loader* function takes two arguments, the context that you just provided in the *index* function and the request made to the *view* function. In this instance, the request is of the index view function.
- The context must be defined prior to the *render* function, and is defined with the `context = {}`. Using an empty context will not pass anything to the template that is being loaded.
  - In this instance, the code defines the context as storing a set of key value pairs.
  - On the left is the key, *'post\_list'*, which is how the list of posts is referenced in the template.
  - The *post\_list* is the value that will replace the *post\_list* reference when the template is rendered.

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

35. Test your code. Your posts should display with their title, publication date, and text.  
36. Modify your index template so that the post's username appears beneath the title.

## Login Template

Now you will add a *login* view to your app to manage who can post and who cannot post. The *login*

view template will rely on some of the built-in Django functions for logins.

37. Within your templates folder, add a new folder called "registration".
38. Within the registration folder, create a file called "login.html".
39. Inside the *login.html* file, add a link to the template that will take a user back to the index view if they so choose. Use the fill-in-the-blank outline below to help create the link.

< \_\_\_\_ \_\_\_\_ = "/" \_\_\_\_/" > \_\_\_\_ < / \_\_\_\_ >

40. Create the form in the *login.html*. Below is an example form that you can use as the contents of your *login.html* file. You may copy and paste to use this code as is or modify it to get the visual effect you like.

```
{% if form.errors %}
<p>Your username and password didn't match. Please try again.</p>
{% endif %}

<form method="post" action="{% url 'login' %}">
{% csrf_token %}

<div>
    <td>{{ form.username.label_tag }}</td>
    <td>{{ form.username }}</td>
</div>

<div>
    <td>{{ form.password.label_tag }}</td>
    <td>{{ form.password }}</td>
</div>

<div>
    <input type="submit" value="login" />
    <input type="hidden" name="next" value="{{ next }}" />
</div>

</form>
```

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.


**Important:** To create the bulletin board app, you will take advantage of some of Django's predefined URL configurations. Since so many websites need login and authentication pages, Django provides these in a special library. To see all the URLs this library file includes that you may use in the future, you can find more at information [Djangos website](#).

41. So that your entire site can access the libraries about logging in, logging out, and authenticating users, add the following URL pattern to your site's *urls.py* file:

```
url(r'^$', include('django.contrib.auth.urls')),
```

Django predefines the login view function for you when you import it in your *urls.py* file. You cannot see the contents of that function, but the import is working behind the scenes. One thing the login function does is pass a form variable to the template. This is how you can use variables like *form.password.label\_tag* in your login template, and the information is passed on without you needing to code additional segments as with the blog app.

42. To verify that you see the login form, navigate to your site's URL with `/login` at the end.



The screenshot shows a web browser window. The address bar contains a green lock icon, the word 'Secure', and the URL 'https://bbsite-pltw-projectleadtheway.c9users.io/login/'. Below the address bar, there is a login form. It consists of two text input fields. The first is labeled 'Username:' and the second is labeled 'Password:'. Below these fields is a button labeled 'login'.

43. Try entering the wrong login information. You should receive feedback that the username and password did not match.
44. Try entering the correct information. A correct login takes you to a 404 error page.
45. The login page does not have information for where to direct a user after they have logged in. You need to specify in the *settings.py* file that a validated login redirects that user to the index page.
- Add the following assignment statement to the end of your *settings.py* file.

```
LOGIN_REDIRECT_URL = '/bb/'
```

- Add a link to the index template to allow a user to go to the login view.
- Add a link from the login view back to the index view.

**Important:** If you are not sure how to add links, review your HTML notes about using anchor tags.

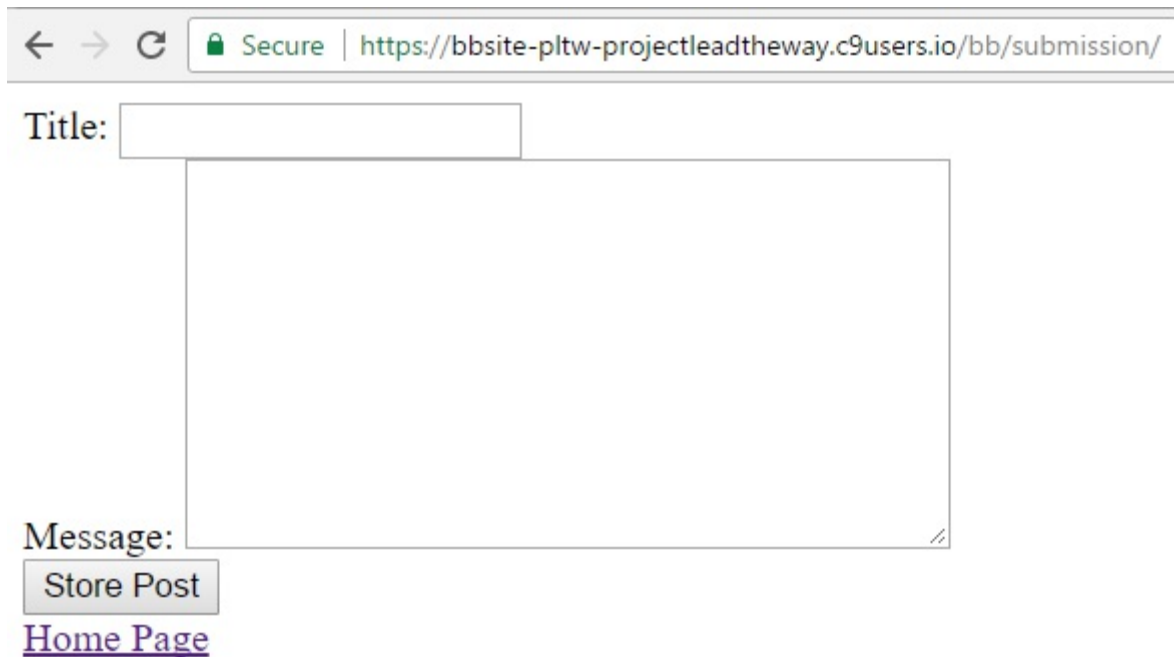
## Forms Submission

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

Now, when the user logs in, they will be taken to your index page. From here, the user needs to be able to post to the bulletin board. You will create a form class to help populate your submission template. When you are done, you will compare this template to the submission function from your

blog app.



← → ↻ Secure | https://bbsite-pltw-projectleadtheway.c9users.io/bb/submission/

Title:

Message:

[Home Page](#)

46. Within your *bb* directory, create a file *forms.py*.
47. In the *forms.py* file, import forms from django.
48. Copy the following code into *forms.py*:

```
class PostForm(forms.Form):  
    title = forms.CharField(label='Title', max_length=100)  
    message = forms.CharField(widget=forms.Textarea)
```

49. Take a moment to discuss the class that was provided in the code above for your forms file.



**PLTW DEVELOPER'S JOURNAL** What does each part of the *PostForm* class do?

50. In the next few steps, you will construct lines of code and script based on directions using computer science concepts. Review some of those now before you continue.

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.





**PLTW DEVELOPER'S JOURNAL** Take notes about any patterns that you were not 100% sure of.

51. Now that you have this new information being collected through the built-in forms, update the view so a user can see the submission form page.
  - Open the *views.py* file, if it is not already open.
  - To use your newly defined form in your submission view function and your submission template, import *PostForm* from *.forms*.
  - Add a submission view to your *views.py* file. Users will see the submission view after they are signed in and click the link **Make a Post**.
  - In your submission function, create a new *PostForm* object that uses the *import PostForm from forms.py*.
  - In the function, add and complete `VariableForStorage = ClassFromForm(method.MethodType)`.
  - Create a context that includes *'form'* as a key, and your *PostForm* as its value pair.
52. Try to navigate to your submission site. You get an error because you have created the *view* function, but it does not return a value and there is no template. You also need to register the URL pattern with the app.
53. Add the *submission* view as a URL pattern in the appropriate *urls.py* file.
54. Add a template variable that will navigate back to the login view.
55. Add a return statement to the submission view function that will load the template and the context. For help, look at your index view function and the fill-in-the-blank outline below.

\_\_\_\_\_ `HttpResponse ( _____ . _____ ( _____ , _____ ) )`

56. Based on the template for the login, create a submission template file that includes the following:
  - Form action that takes the user to the submission view instead of the login view
  - An area for the user to type a title (similar to the area for the user to type their username and password in the login template, except that now you are referencing the forms)
  - An area for the user to type the message for the post they want to make
  - A Submit button
57. Test your submission template. You should be able to:
  - ☐ Type a title
  - ☐ Type a message
  - ☐ Click a Submit button (However, it should not yet store the data in the database, as you have not added the code specifying to do so.)

## User Credentials and Conditionals

Much like you experienced in previous activities, the data entered into the form does nothing until programmed to be stored in a specific way. You are going to store the information in a way similar to

the blog app, but you want to add a few nested conditionals so that the information is only stored by an authenticated user, when they complete the form in a verified, valid way to prevent errors.

**Important:** You will be given directions about code to use to make these conditionals possible. However, having a basic idea of structure first will help you know where to use the code lines properly.



**PLTW DEVELOPER'S JOURNAL** Take a minute to plan out the basic structure of the nested *if* *e/se* conditionals that could accomplish all the tasks it needs to in the submission view to properly display the correct page and messages.

1. If the user did not reach the view through the POST method:  
Direct the user to the submission template IF they are logged in
2. If the user is not authenticated:  
Direct the user to the login template
3. If logged in, and POSTing, and the form is valid:  
Store the information in the form to the database
4. If logged in, and POSTing, and the form is NOT valid:  
Provide feedback about what is wrong with the form

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

58. Modify your submission function in *views.py* to properly store a new Post object in your database if the view is reached via the POST method.
- Update the template loader to navigate to the submission view, so that if the user has not already posted, they are at the correct view to do so.
  - Create a conditional for whether the submission view is reached through the POST method or not. While you add the other parts from this list, be sure to consider which part of the conditional the code segments should be in.
  - Take the existing code in the submission view and place it into the correct part of the conditional.
  - *PostForm(request.POST)* will create a new *PostForm* object using the data from the submission template. Here is a helpful code example:

```
post_to_submit.post_title = form.cleaned_data['title']
```

- Add an *is\_valid()* function of your *PostForm* class to check whether the information is OK before it is stored.
- If the *is\_valid()* function evaluates to True, then the code should access your *PostForm* data.

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

- To make sure that when `is_valid()` evaluates to True, it returns the user name, use:

```
post_to_submit.user = request.user
```

59. Verify that your submission view now allows data storage.
60. Add a link to navigate from the submission view to the index view.
61. Add a link to navigate from the index view to the submission view.
62. Modify the index template so that the index page shows the information in the following way, if it does not already:

pltw made a post:

TITLE May 17, 2017, 1:09 p.m.

MESSAGE

63. Modify your submission function in `views.py` to display the appropriate page based on whether the user is posting or not posting and logged in.
  - You can determine whether a user is **authenticated** with this expression:

```
if request.user.is_authenticated():
```

- Use a context method similar to `context = {'form' : form}` based on the variables you used.

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

## User Signup

It is not always practical to have someone log in to an admin account to add each new user. Therefore, many sites use a self-registration process where users can sign up and create their own account outside of the admin interface. You are going to create a *registration* view and a *success* view, so users know when they have signed up successfully. To avoid security issues, you will use some of the built-in forms that Django offers to set up this part.

Refer to your downloadable resources for this material. Interactive content may not be available in the PDF edition of this course.

64. To access the user creation form, add the following *import* statement to your *forms.py* file:

```
from django.contrib.auth.forms import UserCreationForm, User
```

The form has fields already set up and defined. It also has the added benefit of checking for a valid password, such as a password that is at least eight characters long and is different from the username. You could program all these features and fields individually or just let Django do it for you.

65. Create a success template that notifies a user after they have successfully registered their account. In your template, include the following:

- Message to the user that they successfully made a post
- Message that they need to log in with their new account
- Link to the home view
- Link to the registration view
- Link to the login view
- Link to make a post

66. Create a *success* view that selects the success template with the following return statement:

```
return HttpResponseRedirect(template.render(request))
```

67. Create a registration template similar to the login template. This time, you will use a built-in template, so you do not need your own labels in the form.

- Set the action of the form to the registration URL.
- To replace the inputs (except the button) with the default form as a paragraph, enter:

```
{{ form.as_p }}
```

- Add links to the other views.

68. Create a *registration* view that is modeled after the *submission* view.

- Add the following imports:

```
from django.contrib.auth.models import User
from django.contrib.auth.forms import UserCreationForm, User
```

- Include a conditional that checks if the request method was post.
  - If the request method was not *post*, it should load the default *UserCreationForm()*, identify the registration template, and return the form in the context.
  - If the request method is *post*, there should be a conditional that checks if the form is valid.
    - If the form is not valid, then it should display a response that the form was not valid, should include at least eight characters, should not be all numbers, and should not have the same password as the username. This feedback should

- be formatted in an easily readable way.
- If the form is valid, use *form.save()*, identify the success template, and have an empty context.
- The end of registration function should have a single return statement that responds with the template and renders the context and the request.

## User Logout

Django has built-in options to handle a user logging out, much like the options for logging in. You are going to import and add navigation to each view so users can log out.

69. In *views*, import *logout* from *django.contrib.auth*.

70. In each of the templates, add a link that connects to the default `/logout/` view.

## Extra Touches and Review

### Logout Error

After logging out, the user will get an error if they try to submit a post. If the user is not logged in, there is no username, which causes an error in the form. To make a post, a user must be registered and logged in.

It is possible to use the registration function in the *view.py* file to set up a conditional that prevents the error message by not allowing them to access the submission view without being logged in. Consider what you did in the rest of the conditional and brainstorm what you could do to update your conditional to prevent this error when you build your own app.

### Additional Touches

You may notice that the times are off on the posts. If so, you can go into the *settings.py* file and scroll to the bottom to change `TIME_ZONE = ''` to your local time zone.

Add the CSS design to the index page to create a more professional appearance. Follow the process you used in [Activity 3.2.2](#).

### Review



**PLTW DEVELOPER'S JOURNAL** Compare and contrast the process of creating a submission function using templates in the bulletin board app to the process of creating a submission function without templates in the blog app.

### Conclusion

1. Describe three ways in which basic programming constructs were used in this activity.
2. How did you interpret and respond to the essential questions? Capture your thoughts for future conversations.