ALLOWING USERS TO ENTER DATA:

Before building an authentication system for users to create accounts,
we'll add some pages allowing users to enter their own data.
They should have the ability to create topics, new entries and also edit entries.

Currently, only the superuser can do these things using the admin site
and we don't want users to interact with the admin site, so we'll use
Django's form-building tools to build pages allowing users to enter data.

Adding New Topics:

Let's begin with allowing users to create new topics.
Building a form-based page isn't much different to building the ones we already have,
the major difference is the addition of a new module called forms.py containing the forms.

The Topic ModelForm:

Any web page that let's users enter and submit information is called a form.
Whenever users enter information, we need to validate that information
to ensure it's the right kind of data and that it's not malicious (code interrupting our server).

When we recieve information from a user, we need to process it and save the valid data
to the right place in our database. Django is good with this and automates much of this for us.

Using a ModelForm is the simplest way to build a form in Django.
A ModelForm using the data from our models in models.py to automatically create forms.

We'll write our first form in forms.py, which we create in the app directory.

The new_topic URL:

Now we want to create a page where users can actually add new topics.
URLs should preferably be short and descriptive, so we'll use 'new_topic'.
We add this URL-pattern to learning_logs/urls.py and send the requests to
new_topic() in views.py, which we'll create next.

The new_topic() View Function:

The view function for the new_topic page needs to handle two major situations:

Initial requests for the new_topic page (in which case it should show a blank form)
and the processing of any data submitted in the form.
It should also redirect the user back to the topics page, when data from a
submitted form is processed.

We create new_topic() in views.py.

GET and POST Requests:

There are different kinds of requests, the two main types are GET and POST when building web apps.
GET requests are used for pages that only read data from the server.
POST request are usually used when a user needs to submit information through a form.

There are a few more types of requests but we won't use them for this project.

When the user initially requests the page, the browser will send a GET request.
But, once the user has filled out and submitted the form,
the browser will send a POST request.

By knowing what kind of request it is, we know whether the user requests
a blank form (GET) or asking us to process a completed form (POST).

See views.py and the new_topic function.

The new_topic Template:

We'll create the new_topic.html template to display the form.

```
...
{% extends 'learning_logs/new_topic.html' %}

{% block content %}
 <p>Add a new topic:</p>

 <form action="{% url 'learning_logs:new_topic' %}" method='post'>
  {% csrf_token %}
  {{ form.as_p }}
  <button name="submit">Add topic</button>
 </form>

{% endblock content %}
...
```

On the sixth row, we define an HTML form.
The first argument 'action', tells the browser where to send the information from the form.
We'll send it to the new_topic() view function.
The second argument 'method', tells the browser what request to submit.
We tell it to submit a POST-request.

On row 7, we write a template tag which prevents attackers from using the form
to gain unauthorized access to the server.
This kind of attack is called a cross-site request forgery (csrf).

On row 8, we display the form.
See how simple Django does a task like this.
The template variable {{ form.as_p }} tells Django to create the fields and display the form.
as_p is a modifier which tells Django to render all the form elements in
a paragraph format, as a simple way to display the form neatly.

At row 9, we create a button because Django doesn't do it by default.
The name argument tells Django what kind of a button it is,
and we tell Django it will submit the form.

Linking to the new_topic Page:

Next step is to add a link to the new_topic page on the topics page.
We modify topics.html and add a link after the list of existing topics.

Add a new topic

Adding New Entries:

Let's also allow users to create and edit entries.

The Entry ModelForm:

Like we did with topics, we'll also need to create a form associated with the Entry model.
We create EntryForm in forms.py.

The new_entry URL:

We define its URL pattern in urls.py.

path('new_entry/<int:topic_id>/', views.new_entry, name='new_entry')

We include the topic_id argument because this page needs to be associated with a specific topic.

The new_entry() View Function:

We create the new_entry view function in views.py.
This function will look much like the new_topic() function.
See views.py.

The new_entry Template

Similar to new_topic.html, we define new_entry.html.

```
...
{% extends 'learning_logs/base.html' %}

{% block content %}

  <p><a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a></p>

  <p>Add a new entry:</p>
  <form action="{% url 'learning_logs:new_entry' topic.id %}" method='post'>
    {% csrf_token %}
    {{ form.as_p }}
    <button name='submit'>Add entry</button>
  </form>

{% endblock content %}
...
```

We extend the base.html template on the first row.
Then we display the current topic the user is writing an entry for,
we also link it to the corresponding topic page using an anchor tag and a URL-tag.

When we define the HTML form, we include the topic.id in the URL-tag.
This sends the entered form data to the new_entry() view function with the appropriate topic object.

Other than that, this template is pretty much identical to the new_topic template.

Linking to the new_entry Page:

Next step is to include a link to the new_entry page from every individual topic page.
We modify the topic.html template.

```
...
<p>
  <a href="{% url 'learning_logs:new_entry' topic.id %}">Add new entryr</a>
</p>
...
```

We add this anchor tag below 'Entries:' before the UL-tag.
We link to the new_entry view function and give it the current topic id, which will return the URL.

Editing Entries:

Now we'll make a page where users can edit their entries.

The edit_entry URL:

We add and prepare our path in the app's urls.py:
path('edit_entry/<int:entry_id>/', views.edit_entry, name='edit_entry')

The edit_entry() View Function:

When the edit_entry() view function recieves a GET request, it should return
a form which allows the user to edit the entry.
When it's a POST request, with an edited entry, we'll save it to the database.
See views.py

The edit_entry Template:

```
...
{% extends "learning_logs/base.html" %}

{% block content %}

 <p><a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a></p>

 <p>Edit entry:</p>

 <form action="{% url 'learning_logs:edit_entry' entry.id %}" method='post'>
  {% csrf_token %}
  {{ form.as_p }}
  <button name="submit">Save changes</button>
 </form>

{% endblock content %}
...
```

At the action argument, we enter edit_entry with the entry id which...
sends the form back to the edit_entry() function for processing.

Linking to the edit_entry Page:

```
...
<a href="{% url 'learning_logs:edit_entry' entry.id %}">Edit entry</a>
...
```

We include the anchor tag above to link to the edit_entry page from every topic page.
It links to the edit_entry page for the current entry.

Now Learning Log has most of the functionality it needs.
Users can add topics, add entries and change entries.
Now we'll implement a registration system so anyone can create an account
and manage their own topics and entries.

SETTING UP USER ACCOUNTS:

Let's begin setting up the user registration and authorization system.

The users App:

We'll create an app called users to contain all the functionality we need
for the users system in the terminal using:
python manage.py startapp users

Adding users to settings.py:

We add 'users' to settings.py in the INSTALLED_APPS list under # My apps.
This will make Django include the app in the project.

Including the URLs from users:

We add the users URLs to the urlpatterns list in the project's urls.py.

The Login Page:

The first thing we'll do is to implement a login page.
Django actually provides a default login view, which makes our life a bit easier.
Because it's a Django default, the URL pattern will look a little different.

To begin with, we create the urls.py file for the users app in its directory.
We import the things we need, set the app_name attribute and create the urlpatterns list.

Inside the urlpatterns list, we include the default authentication urls...'
which includes named URL patterns like 'login' and 'logout'.
By default, when Django sees the login URL, it sends the request to the default login view.

The login Template:

When someone requests the login page, Django uses the default login view but
it doesn't use a default template for it. We need to make that ourselves.

The default authentication views from Django look for templates inside a special folder
called registration, which we need to create since it's not initially made by Django.

This registration folder should be inside the templates folder, which we also need to make.

We create login.html in 'templates/registration':

```
...
{% extends "learning_logs/base.html" %}

{% block content %}

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

 <form method="post" action="{% url 'users:login' %}">
  {% csrf_token %}
  {{ form.as_p }}

  <button name="submit">Log in</button>
  <input type="hidden" name="next"
   value="{% url 'learning_logs:index' %}" />
 </form>

{% block content %}
...
```

We extend this page from the base-template.
Note that a template can inherit from another template in another app.

In the if-block, we check if the default login-form's error attribute is set.
If it is (if login credentials were incorrect) we display an error message.

We get the login form from the login view function.
We also add the submit button.

At <input... we add a hidden form element.
When we give "next" to the name argument, we tell Django where we want to send
the user if the login was successful. The value argument after name argument
needs a URL where it will send the user.
We send the user back to the home page.

Linking to the Login Page:

Let's add a link to the Log In page in the base template.
We only want this link to show if a user isn't logged in,
so we nest this link inside an if-statement.

If a user is logged in, we display a greetings message to the user.

```
...
{% if user.is_authenticated %}
  Hello, {{ user.username }}.
% else %}
  <a href="{% url 'users:login' %}">Log in</a>
{% endif %}
...
```

If the users is_authenticated attribute is set to True, which it is when a user is logged in,
we display a message saying hello.
We access their usernames using the users username attribute which is set
if a user has been authenticated.

Using the Login Page:

We have already set up a user account (superuser).
We try the login page to see if it works as it should.

It works, it displays the correct greeting.

Logging out:

We need to provide a way for users to logout.
We'll add a link to base.html which logs users out and sends them to a page confirming that.

Adding a Logout Link to base.html:

```
<a href="{% url 'users:logout' %}">Log out</a>
```

We add the anchor tag above to base.html under the if-block.
We only want to show it when a user is logged in.
We use the default named URL pattern to logout which is simply 'logout'.

The Logout Confirmation Page:

Users will want to know if their logout was successful.
If so, we create a page called logged_out.html which will tell them that.
Why logged_out.html?
Because the default logout view function searches for a template called
logged_out.html in its render function.
We need to create that ourselves, and Django will automatically render it
when a user log's out.

We store this page in the registration folder inside templates.

```
...
{% extends "learning_logs/base.html" %}

{% block content %}
  <p>You have been logged out. Thank you for visiting!</p>
{% endblock content %}
...
```

The Registration Page:

Next, we want to build a page so new users can actually register. We'll use Django's default UserCreationForm but write our own view function and template.

The register URL:

We add the URL pattern to the register page in urls.py in the users app.
Don't forget to import the views module from the users directory to the urls.py.

The register() View Function:

Check views.py in users app.

The register template:

```
...
{% extends "learning_logs/base.html" %}

{% block content %}

  <form method="post" action="{% url 'users:register' %}">
    {% csrf_token %}
    {{ form.as_p }}

    <button name="submit">Register</button>
    <input type="hidden" name="next" value="{% url 'learning_logs:index' %}" />
  </form>

{% endblock content %}
...
```

This template looks similar to most other templates we've built.
We display the form from the users:register view function.

At <input... we give Django a URL where to redirect a user if the
form submission and register was successful.

Linking to the Registration Page:

Next up, we'll add a link to the registration template into the base template...
if a user isn't currently logged in.
We'll add this anchor tag under the if-block.

```
...
<p>
--snip--
 {% if user.is_authenticated %}
   Hello, {{ user.username }}.
   <a href="{% url 'users:logout' %}">Log out</a>
 {% else %}
   <a href="{% url 'users:register' %}">Register</a> -
   <a href="{% url 'users:login' %}">Log in</a>
 {% endif %}
</p>
...
```

ALLOWING USERS TO OWN THEIR DATA:

Users should be able to enter data that only belongs to them exclusively.
Next, we'll create a system to figure out which data belongs to which user.
We'll also restrict certain pages for users so they only can work with their own data.
We'll also modify the Topic model, so every Topic object belongs to a specific user.

Let's begin with restricting users access to certain pages.

Restricting Access with @login_required:

Django makes this easy for us using the @login_required director,
which can restrict access to certain pages for logged in users.

A director is a directive written before the definition of a function,
which tells the function how to behave.

Restricting Access to the Topics Page:

In the finished project each topic will be owned by a user,
so we only want registered and logged in users to be able to access the topics page.

We import the login_required to views.py (in learning_logs) and assign it to the topics function.

Check views.py.

As a result, it only let's logged in users request the topics page.
If the user isn't logged in, it redirects the user to the login page.
This redirection doesn't work by default, we need to tell Django where to find the login page.

We tell Django by adding the following to the bottom of settings.py:

```
# My settings
LOGIN_URL = 'users:login'
```

The LOGIN_URL variable tells Django which URL the login page has.

Restricting Access Throughout Learning Log:

As you can see, Django makes it easy to restrict access to pages.
But we need to decide ourselves, which pages to restrict.

The best way of doing this is to first think about the pages which don't
need restriction, and restrict all the others.

This method does most likely not leave out any pages that need restriction unrestricted.

In our Learning Log project, we'll only leave the home page and the registration page
unrestricted.
We add the login_required decorator before the view functions we want to restrict.

Connecting Data to Certain Users:

The next thing we need to do is to connect the data to the user who submitted it.

In Learning Log, each entry is connected to a specific topic,
which makes topics the higher level data. The lower level entries will follow the topics.

Due to that, we only need to assign each topic to a certain user.

We'll modify the Topic model to make each topic belong to a user,
using a ForeignKey. Don't forget to migrate the database after modifying a model.

The next step will be to change some view functions to only show data
owned by the current logged in user.

Modifying the Topic Model:

We add a new attribute to the topic model called owner.

This attribute contains a ForeignKey-relationship binding the topic to a user.
Check models.py.

Identifying Existing Users:

Our already existing topics don't have a user assigned to them,
so we'll start by giving them all to the superuser.

If there are more users registered, which is more realistic...
you can identify them using the django shell.

Like we did in models.py, we can import the User model and check existing
users using a for loop looping through User.objects.all().

Migrating the Database:

Now we want to migrate our model changes to the database.
We enter makemigrations into the cmd prompt, which will let us either set
a default value in the prompt or a default value in models.py.

This value should be the ID of the user, which you can access in the shell using user.id.

Restricting Topics Access to Appropriate Users:

Currently, when you're logged in and visit the topics page,
you can see all topics no matter the owner.
Now we'll modify the topics page so the user only can see their own topics.

We change the following at the topics attribute:

```
topics = Topic.objects.filter(owner=request.user).order_by('date_added')
```

...
Notice the request.user attribute.
If a user is logged in, the request will always contain an attribute called user.
The user attribute contains the logged in user object with its information.
Topic.objects.filter(owner=request.user) is a query featuring a filter,
and we filter the topics off of the owner attribute, which is a user-object.
This query will contain all the topics owned by the current logged in user.

Protecting a User's Topics:

Even if a user don't own any topics, he can atm reach other user's topics
by requesting the URLs of the topic pages.

To protect user's from this, we raise a 404 when this happens.
We check in views.py - topic function if the topic.owner isn't equal to the
user coming from the request, if so we raise 404.
Check views.py.

Protecting the edit_entry Page:

Like the topics, people can also access the edit_entry URLs to edit other users entries.
Let's protect the edit_entry pages like we did with the topics pages.
We check if the topic owner is the same user as request.user.
Check views.py and edit_entry().

We raise a 404 exception if a user shouldn't be on the current page.

Associating New Topics with the Current User:

Currently, our page for creating new topics is broken.
This is because we haven't defined a way for Django to know which user the topic should belong to.
Because of this, Django raises the IntegrityError.
It says u can't create a topic with no owner assigned to it.

What we need to do is to create a way to assign the current logged in user to the
topic that is being created.
We do this in views.py in the new_topic() definition.