STYLING AND DEPLOYING AN APP:

Styling Learning Log:

So far, we've kind of ignored the looks of our web project.
We've focused mainly on functionality and making it work fully, which is a good way of approaching these kind of projects.
An app is only useful if it works...
Looks are important for users, which is why it's a critical part of development.
To style our project, we'll use and integrate the django-bootstrap4 app.

The django-bootstrap4 App:

We use pip install django-bootstrap4 to install Bootstrap to our environment.
But to include Bootstrap, we need to include it in the INSTALLED_APPS list inside settings.py.

Using Bootstrap to Style Learning Log:

Bootstrap has a large collection of styling tools and templates.
Templates apply to the overall style which makes them easier to use than individual tools.
At getbootstrap.com, you can find all tools and templates.
We'll use the Navbar Static template.

Modifying base.html:

To use the bootstrap4 template, we need to modify the base template.

Defining the HTML Headers:

We delete all content inside base.html.
The first thing we'll add and configure, is the HTML headers.

```
...
{% load bootstrap4 %}

<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=devicewidth, initial-scale=1,
    shrink-to-fit=no">
  <title>Learning Log</title>

  {% bootstrap_css %}
```

```
  {% bootstrap_javascript jquery='full' %}

</head>
...
```

On the first line, we load the collection of templates tags from bootstrap4.
The doctype defines what kind of file it should be considered as.
We also define the HTML language to english.

An HTML file contains two parts, the head and the body.
We begin defining the head on the fifth line and indent its content.
The head doesn't display any content, it just tells the browser what it needs
to know in order to actually display the page correctly.

The title tag, tells the browser what to name the page in the title bar.

The last two tags in the head are from bootstrap customs.
Bootstrap_CSS tells django to include all style files.
The last one enables all interactive behavior you might want to use on a page,
such as collapsible navigation bars.

Defining the Navigation Bar:

The code defining the navigation bar will be pretty long, so we'll work in sections.

```
...
<body>

  <nav class="navbar navbar-expand-md navbar-light bg-light mb-4 border">

    <a class="navbar-brand" href="{% url 'learning_logs:index'%}">
       Learning Log</a>

    <button class="navbar-toggler" type="button" data-toggle="collapse"
       data-target="#navbarCollapse" aria-controls="navbarCollapse"
       aria-expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span></button>
...
```

We start off by opening the body.
Then we open the navigation element (nav) which controls the navigation sextion on the page.

Everything inside this element is styled according to the Bootstrap style rules
defined by the selectors navbar, navbar-expand-md, adn the rest you see in "class"
at the nav-tag opener.

A selector determines which page elements apply to a certain style rule.
Selectors are defined in the class.
The navbar-ligth and bg-light selectors styles the navbar with a light background.
The mb-4 selector (mb stands for margin bottom) makes a little space
appear between the navbar and the rest of the page.
The border selector provides a thin little border around the light background
to set it off a little from the rest of the page.

After opening the nav section, we define an anchor tag using the navbar-brand class.
Navbar-brand selector styles this link so it stands out a little from the rest.
A way of branding the site.
This project-name link will be on the left side and appear on every page.

Next, we define a button tag.
This button will only appear if the browser window is too narrow to display
the whole navbar horizontally.
When a user clicks this button, a drop down list of the navbar elements will appear.
The collapse reference causes the navbar to collapse when the user shrinks the browser window
or when the site is displayed on mobile devices with small screens.

```
...
<div class="collapse navbar-collapse" id="navbarCollapse">
 <ul class="navbar-nav mr-auto">
  <li class="nav-item">
   <a class="nav-link" href="{% url 'learning_logs:topics'%}">
      Topics</a></li>
 </ul>
...
```

We add the code above just below the span-tag.
By opening the div-tag, we start a new section of the navbar.
Div stands for division.
You build web pages by dividing it into divisons (sections)
and design behaviour and styling for each section.

The styling or behaviour rules you assign in the div-opening affect all the content in the div section.
This code is the beginning of the collapsed navbar that will appear on smaller screens or windows.

At the UL-tag, we define a new set of links.
Bootstrap defines the navigation elements as items in an unordered list with style
rules that make it look nothing like a list.
Any link or element you need or want on the navbar can be included in one of these UL lists.

Then we create a list item, which is a link to our topics page.

```
...
   <ul class="navbar-nav ml-auto">
    {% if user.is_authenticated %}
     <li class="nav-item">
      <span class="navbar-text">Hello, {{user.username}}.</span>
     </li>
     <li class="nav-item">
      <a class="nav-link" href="{% url 'users:logout' %}">Log out</a>
     </li>
    {% else %}
     <li class="nav-item">
      <a class="nav-link" href="{% url 'users:register' %}">Register</a>
     </li>
     <li class="nav-item">
      <a class="nav-link" href="{% url 'users:login' %}">Log in</a>
     </li>
    {% endif %}
   </ul>
  </div>

</nav>
...
```

Here, we do once again begin a new set of links by opening the UL tag.
You are allowed to have as many groups of links on a page as you want.
This gruop will be the login, logout and registration links, which will
be located on the right side of the navbar due to ml-auto selector in the UL opening.
ml-auto stands for margin left automatic, which means it examines the other elements in navbar
and pushes these to the right side with the right amount. It creates a margin to the left.

Then, we start an if block, the same we used before to examine whether a user
is logged in or not.
If the user is logged in, we start by displaying a welcoming message.
For this, we use the span tag.
The span element is used to create and style pieces of text or other elements on a page
that are part of a longer line. In this case we don't want this text to be a link
like logout, login and register will be.

Then we create our other navbar elements, which are all going to be links using
anchor tags.

We also close this div, which contains the elements that will be in the collapsed navbar.
The navbar is done for now and we also close the nav tag.

Defining the Main Part of the Page:

The rest of base.html will contain the main page.

```
...
 <main role="main" class="container">
  <div class="pb-2 mb-2 border-bottom">
   {% block page_header %}{% endblock page_header %}
  </div>
  <div>
   {% block content %}{% endblock content %}
  </div>
 </main>

</body>

</html>
...
```

We begin with opening the main tag, which is the main and most significant part
of the body.
We use the container selector, which is a simple and effective way of grouping elements.
In this container, we use two div tags.
The first one, the page_header will mostly be used for titles and headlines.
And compared to the other div, we use some selectors to make this div stand out.

The pb-2 selector stands for padding bottom, and padding refers to space between
an elements content and it's border. This selector provides a moderate amount
of padding at the bottom of the element.

The mb-2 selector stands for margin bottom, and a margin refers to the space between
an elements border and other elements on the page.

We also the border-bottom selector, which provides a thin border at the bottom
of the page_header block.

The next div tag includes the content block.
For this part, we don't use any selectors because the styling may differ from
page to page.

Start the server and see the new page with the professional navbar.

Styling the Home Page Using a Jumbotron:

To update and style the home page, we'll use something called a jumbotron.
A jumbotron is a large box that stands out from the rest of the page,

it can contain whatever you want.
Most likely, it's used on a home page to hold a brief description of the overall project.

Let's go into index.html:

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

{% block page_header %}
  <div class="jumbotron">
    <h1 class="display-3">Track your learning.</h1>

    <p class="lead">
      Make your own Learning Log, and keep a list of the topics
      you're learning about. Whenever you learn something new about a topic
      make an entry summarizing what you've learned.
    </p>

    <a class="btn btn-lg btn-primary" href="{% url 'users:register' %}"
      role="button">Register &raquo;</a>
  </div>
{% endblock page_header %}
...
```

We begin by opening the page_header block and start the division (section) using the jumbotron selector.
A jumbotron is not really a thing of its own, it's just a div element with a certain set of
styling directives from Bootstrap applied to it.

The jumbotron consists of three elements.
The first one is the h1, which is a first level header (rubrik), which displays a short message.
For the h1, we use the display-3 selector which gives it a thinner and taller look.

The second element is just a simple paragraph with some more detailed information about the project.

The third element is a welcoming register button.
As you can see, we can create a button of an anchor tag link using selectors.
btn (button) is making the element a button.
btn-lg makes the button a bit larger.
btn-primary give the button a blue color which makes it pop out more.

In between the anchor tags, we define what text the button should display.
After register we write » which is called an HTML entity that looks like >>.

Styling the Login Page:

Let's modify login.html.
Here's the new code:

```
...
{% extends "learning_logs/base.html" %}
{% load bootstrap4 %}

{% block page_header %}
  <h2>Log in to your account.</h2>
{% endblock page_header %}

{% block content %}
  <form method="post" action="{% url 'users:login' %}" class="form">
    {% csrf_token %}
    {% bootstrap_form form %}
    {% buttons %}
      <button name="submit" class="btn btn-primary">Log in</button>
    {% endbuttons %}

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

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

We begin with loading the bootstrap template tags onto our file.

In the page_header, we display a lvl 2 header showing a informative message.

Then we start the content block.
Notice that we removed the {% if form.errors %}-block, this is because bootstrap handles form errors automatically.

When we open the form tag, we insert the form class from bootstrap so bootstrap knows it's a form and uses its styling.
We replace {{ form.as_p }} with {% bootstrap_form form %} to apply the form styling from bootstrap to the form's individual elements.

After that, we open the bootstrap template tag {% buttons %} which adds bootstrap styling to buttons.

Styling the Topics Page:

The topics page doesn't need a lot of work.

We move the topics header into the page_header block and remove it from the content block.

We add an h3 tag before the links to the topics and to the other elements.

Styling the Entries on the Topic Page:

The individual topic page needs some work, it's the page with the most content.

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

{% block page_header %}
<h3>{{ topic }}</h3>
{% endblock page_header %}

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

{% for entry in entries %}
  <div class="card mb-3">
  <h4 class="card-header">
    {{ entry.date_added|date:'M d, Y H:i' }}
    <small><a href="{% url 'learning_logs:edit_entry' entry.id %}">
      edit entry</a></small>
  </h4>
  <div class="card-body">
    {{ entry.text|linebreaks }}
  </div>
  </div>
{% empty %}
  <p>There are no entries for this topic yet.</p>
{% endfor %}

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

The first thing we do is to move the topic-name into the header as a headlining lvl 3 header.

Before, we displayed each entry as a list item in a UL tag but
now we create a whole new div section for each entry.
These div elements use the card selector which makes each entry pop out and have
its own section. Looks like a card.

Our cards has two nested elements, one header and one body.
The header is a lvl 4 header tag with the card-header selector.
This element contains the datetime the entry was created and also
a link to edit the entry using the <small> tag.
The small tag makes it appear a bit smaller than the timestamp.

The second element is a nested div element with the card-body selector.
Here, we place the actual text from the entry, in a simple box on the card.

Deploying Learning Log:

Our project is now fully functional, usable and professional looking.
Let's now deploy it to a live server so anyone on the internet can use it.

For this we'll use Heroku, which is a web-based platform that let's you
manage and deploy web applications.

Making a Heroku Account:

I register at heroku.com.

Installing the Heroku CLI:

To deploy and manage projects on Heroku's servers, we need the tools available
in tge Heroku Command Line Interface (CLI).
We go to the Heroku Docs and find the installation for CLI.

Installing Required Packages:

pip install psycopg2
The psycopg2 package is required to manage the database that Heroku uses.

pip install django-heroku
The django-heroku package handles pretty much all the configuration that is needed
for our app to run properly on Heroku's servers.

pip install gunicorn
The gunicorn package provides a server capable of serving apps in a live environment.

Creating a requirements.txt File:

Heroku needs to know which packages our project depends on, so we'll create a list of these
using pip and the command below.
pip freeze > requirements.txt

The freeze command tells pip to write the names of all the packages currently installed
in the project into the file requirements.txt which will be created.
Check this file, and we'll see our packages and what versions they are.

Heroku needs these because it will create an environment when we deploy our project,
and it needs to install all of the packages for our project to work properly.

Specifying the Python Runtime:

Unless we specify a Python version, Heroku will use its default version but let's
make sure we're on the same page.
We check our Python version using python --version in CMD.
We copy this version (Python 3.8.3) into a new file called runtime.txt which
we place in the same directory as our manage.py file.

python-3.8.3
  Make sure to only have one line in this file containing the text above.
  Make sure to format the python version like above.
  Python should be lowercase and bind python and the version number together using a hyphen.

Modifying settings.py for Heroku:

```
...
# Heroku settings:
import django_heroku
django_heroku.settings(locals())
...
```

We add the code above to the very end of settings.py to define some specific
settings for the Heroku environment.

Making a Procfile to Start Processes:

A PROCFILE tells Heroku which processes to start to properly serve the project.
We'll create a new file called "Procfile" with no extension in the same directory
as the manage.py file.
In this one-line file, we write:
  web: gunicorn learning_log.wsgi --log-file -

This line tells Heroku to use gunicorn as a server and to use the settings
in learning_logs/wsgi.py to launch the actual app.
The log-file flag tells Heroku the kind of events to log.

Using Git to Track the Project's Files:

We make a new git repo and commit our working state. Se boken.

Pushing to Heroku:

Finally, we're ready to push our project onto Heroku.
While active in our venv, we issue the following commands:

heroku login
This takes us to a page where we login to our Heroku account on the CLI.

heroku create
This line tells heroku to create a new empty project.
Heroku makes up a random name (word-word-number), which can later be changed.

git push heroku master
This line tells Git to push the master branch of our project to the
new repository heroku just created.
Then, Heroku builds the project on its live servers using these files.

When this process is complete, you can find the URL we'll use
to access the live project, which we also can change along with the project name.
"https://stark-scrubland-20886.herokuapp.com/"

Now our project is actually deployed! ...but not fully configured.
To check that the server process started correctly, use heroku ps in console.

At the first line, we can see how many free hours are left for our project to
be live this month. Heroku allows for up to 550 free hours a month.
If this limit is reached, Heroku will show an error page, which we'll customize later.

heroku open
We're live and we can request our project in the browser using a URL but this command
opens it for us automatically in the browser.

***
Notera att det inte går att använda appen fullt ut än eftersom databasen inte är
konfigurerad, går exempelvis inte att registrera eller logga in.
***

Setting Up the Database on Heroku:

Just like we did in our local Django project, we need to run the migrate command
to create and migrate the database.
You can run Django and Python commands on a Heroku project using the command heroku run.

heroku run python manage.py migrate
After running this, we see the same migration process as when we did this locally.

Now we can use our project just like we could locally!!!

Notice the data (users, topics...) isn't there, this is because we're not on the same database.
This is very normal practice, you usually don't copy the local data onto the live project
because it's most likely just testing data.

Refining the Heroku Deployment:

It's time to refine and "perfect" our deployment.
We will create a superuser so we can administrate.
We'll also set DEBUG to False, so the Django error specifications don't come up
when people get errors. This could be dangerous if hackers got to see the detailed info.

Creating a Superuser on Heroku:

Instead of running commands using heroku run...
We'll try opening a bash terminal session using 'heroku run bash'.
The Bash language runs in many Linux terminals and we'll use it to
set up our superuser account.

python manage.py createsuperuser

Now we can access the admin page.

Creating a User-Friendly URL on Heroku:

https://stark-scrubland-20886.herokuapp.com/

The above URL is weird and not really memorable.
We can thankfully rename the app using a single command in the venv:

heroku apps:rename learning-log-020315
This name needs to be all unique.

With the free version of heroku, the server will go to sleep when it hasn't
recieved any request for a while. When you request it again the first time,
it will be a bit slower.

Securing the Live Project:

The fact that the DEBUG setting is set to True is a big security issue for our deployment.
The error pages Django provides when DEBUG is True contain vital information for attackers.

We'll control whether this debugging info is shown on the live server by
setting something called an environment variable.
Like the sound of it, environment variables are values set in a specific environment.

Using environment variables is a way to store sensitive information on a server,
by keeping it separate from the rest of the project's code.

Let's modify settings.py.
We add 4 lines under Heroku settings so Django looks for an
environment variable when the project is running on Heroku.

```
...
if os.environ.get('DEBUG') == 'TRUE':
    DEBUG = True
elif os.environ.get('DEBUG') == 'FALSE':
    DEBUG = False
...
```

The os.environ.get() method reads the value associated with a specific environment variable
in any environment where the project is running.
Using the if-elif block, we'll set DEBUG to True if its value is TRUE and False if value is FALSE.

Committing and Pushing Changes:

The changes to settings.py will now need to be committed to the Git-repo and then pushed to heroku.
In the terminal (venv):

```
git commit -am "Set DEBUG based on environment variables."
git status
```

We issue the git commit command including a descriptive message of what changes have been made.
Don't forget the -am flag which makes the commit include all the files that have been changed.

Make sure to issue git status to make sure that all changes have been committed to the master branch.

Let's push this to heroku:

```
git push heroku master
```

Heroku recognizes the update of the repository and rebuilds our project with the new changes taken into account.

Setting Environment Variables on Heroku:

Now that our heroku environment is ready, we can set DEBUG to what we want.
In terminal sesh:

heroku config:set DEBUG='FALSE'

...

Setting DEBUG and restarting □ learning-log-020315... done, v7

DEBUG: 'FALSE'

...

The command heroku config:set sets an environment variable for us.
We set DEBUG to FALSE (everytime an environment variable is set on heroku, it restarts the project).

ERROR!!!
WRITING ...:set DEBUG='FALSE' does not work. IT SHOULD BE ...:set DEBUG=FALSE

Creating Custom Error Pages:

We've configured Learning Log to return 404 error pages when a user requests
pages they can't have access to. There are also 500 server errors (internal errors).
A 404 usually means you code is correct, but the object requested doesn't exist.
A 505 ususally means there are code errors, such as errors inside functions in views.py.
Currently, our project returns the same general page for these errors but
now we'll make our own error pages to match the design of the Learning Log project.

Making Custom Templates:

For these HTML files, we need to make a new template directory located in the outermost
learning_log folder where manage.py is located.

404.html:

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

{% block page_header %}
  <h2>The item you requested is not available. (404)</h2>
{% endblock page_header %}
...
```

500.html:

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

{% block page_header %}
  <h2>There has been an internal error. (500)</h2>
{% endblock page_header %}
...
```

Two basic error messages.
These two new files require a slight change to settings.py.
Inside the dict in the TEMPLATES list, we go to the 'DIRS' key and create a list item in its list.
...
'DIRS': [os.path.join(BASE_DIR, 'templates')]
...
This change tells Django to look in the root template directory for the error page templates.

Viewing the Error Pages Locally:

Before pushing the error templates to heroku, we might want to see them locally.
We do this by setting DEBUG inside settings.py to False.
To see 404, request a URL that doesn't exist or belong to current user.
To see 500, request like http://localhost:8000/topics/999/ unless you've created 999 topics already!

For further development, set DEBUG to True.

Pushing the Changes to Heroku:

In venv terminal session:

```
git add .
git commit -am "Added custom 404 and 500 pages."
git push heroku master
```

Notice 'git add .'
We include that because we made new files which git needs to include into the repository.

Using the get_object_or_404() Method:

At this point, when a user requests a topic or entry that doesn't exist,
they get a 500 server error.
Shouldn't that be a 404 instead. Yes, that's more like a 404.
The thing that happens is that Django tries to render the nonexistent page,
but it doesn't have enough information to do so, which returns a 500 error thinking it's some code issues.

This situation would be more accurately handled as a 404 error and we can implement
this behavior using the Django shortcut function get_object_or_404().

This function tries to get the request from the database, but if the request object
doesn't exist, it raises a 404 error instead.

To make this work, we import the function to views.py and replace Topic.objects.get()
inside the topic-function with get_object_or_404().

We push the changes to Heroku.

Ongoing Development:

There's a fairly consistent for updating projects like the Learning Log.

Step 1:
Make your local project file changes, and if these changes include new files:
Issue the git add . command to add them to the Git repo.
If your changes require a database migration, git add . is necessary because
the migration generates a new migration file.

Step 2:
Commit your changes to the Git repo using the git commit -am "informative message".
After the commit, you need to push the changes to Heroku using
git push heroku master
If you migrated the database locally, you'll need that on the live Heroku database aswell.
Enter this: heroku run python manage.py migrate

Step 3:
Make sure the changes have been made and are active in the live project!

The SECRET_KEY Setting:

Django uses the value of the SECRET_KEY variable in settings.py
to implement a number of security protocols.

For this project, we've included our SECRET_KEY in settings.py in our final commit.
For a practice project like this, that's fine but you should be more careful
with this when you're building a serious project.

Deleting a Project on Heroku:

To make things clear: Deleting a project on Heroku does nothing to your local project.
Delete your project either on their website or in the command prompt using the following:
heroku apps:destroy --app appname