Learning Logs project

1. Base project from `Python Crash course 2nd ed` by `Eric Matthes`
2. Prior to deployment, the following steps are required to launch project
    - Launch virtual environment
        - `source ll_env/bin/activate`
    - Run the Django server
        - `python manage.py runserver`
3. Each `Model` is essentially a special class Django uses to present data
4. Each `View` is a special function that processes the data of user request and presents a UI to the browser. A base `template` is often used to prepare the UI for different views.
5. The interactive shell is useful for debugging within the virtual environment `python manage.py shell`

Initial Setup

Virtual environment

A virtual environment ties the application with all its required packages into a

1. Create virtual environment called `ll_env` by entering into terminal:
    - `python3 -m venv ll_env`
    - There should be a new folder called `ll_env` that contains a `pyenv.cfg` file.
2. Activate the virtual environment (for Mac Os)
    - `source ll_env/bin/activate`
    - the start of your path should now show `(ll_env)`
3. Exit or deactivate virtual environment
    - enter `deactivate` in terminal OR close terminal to exit the virtual environment

Setting up Django

Django Installation

1. Activate virtual environment
2. Install Django in terminal using
    - `pip install Django`

Create new `project` called `learning_log`

1. !! You MUST include the period at the end !! Create project using
    - `django-admin.py startproject learning_log .`
2. This will create the following :
    - `manage.py` a project specific file that directs commands to various parts of Django. Specifically, it
        - puts project packages on the `sys.path`

        - sets up the `DJANGO_SETTINGS_MODULE` environment variable so that it points to the project settings file
    - `learning_log` directory that contains:
        - settings.py >> manages Django behavior
        - urls.py >> manages pages Django creates
        - wsgi.py >> manages a `web server gateway interface`

Although `django-admin` is the CLI for administrative tasks, it is simpler to manage a project through `manage.py`

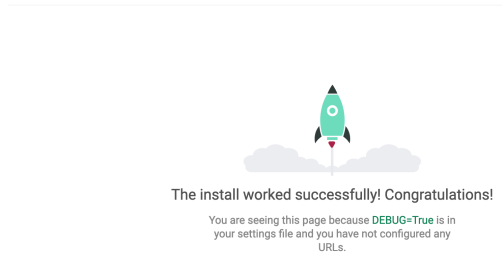
Database Installation

1. Create the database (SQLite by default):
    - `python manage.py migrate`
2. this will create a `db.sqlite3` file

Run Server to view the boiler plate project

1. Run the server to verify the project setup

    - `python manage.py runserver`
    - The following default page can be seen on a browser directed to `http://127.0.0.1:8000/`



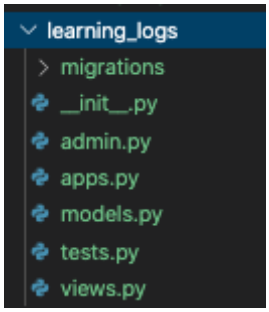
2. If you accidentally close the terminal in VS Code and need to quit the server, enter the following in a new terminal

    - `sudo kill $(lsof -t -i:8000)`

3. Quit the server using `CONTROL-C`

Create Django setup files for new `app` called `learning_logs`

1. In a new terminal window, activate the virtual environment -`source ll_env/bin/activate`
2. Run startapp to build initial infrastructure files
    - `python manage.py startapp learning_logs`
3. This will create a new directory with the following files:



- `models.py` >> manages app data
- `admin.py` >> displays models in the django admin panel
- `views.py` >> views are a Python function or class that takes a web request and returns a web response. In Django, views have to be created in the `views.py` file within each `app` inside the `project`

Define models

Models are simply classes that define how Django should interact with the app's data.

Add the `Topic` class to `models.py`

```
from django.db import models

# Create your models here.

class Topic(models.Model):
    """ Topic that the use in learning about """
    text = models.CharField(max_length=200) # allocates 200 chars in database
    data_added = models.DateTimeField(auto_now_add=True) # sets to current
data/time when new topic created
    def __str__(self):
        """ Return a string representation of the model """
        return self.text
```

Activate model

Django looks inside `settings.py` at the `INSTALLED_APPS` list to determine which apps are part of the project beyond the initial built in apps. Add the follow section to bottom of the `INSTALLED_APPS` section to let Django know that `learning_logs` is part of the project.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
```

```
    'django.contrib.staticfiles',
    # my apps
    'learning_logs',
]
```

Update database for `Topic model` data

The `makemigrations` command builds a migrations file to create a table for the Topic model. In this case, it creates `0001_initial.py`

1. From the virtual environment in terminal run:

   `python manage.py makemigrations learning_logs`

2. The following output indicates that the file was successfully created

   ```
   Migrations for 'learning_logs':
   learning_logs/migrations/0001_initial.py
       - Create model Topic
   ```

Apply migration

This will modify database with the latest migrations file.

1. Run migrations

   `python manage.py migrate`

2. The following output indicates a successful migration

   ```
   Operations to perform:
   Apply all migrations: admin, auth, contenttypes, learning_logs, sessions
   Running migrations:
   Applying learning_logs.0001_initial... OK
   ```

The typical flow to update the database is:

1. Add specific `class` model to `models.py`
2. Run `makemigrations` to build a `migrations` file `python manage.py makemigrations` *appname*
3. Run `migrations` `python manage.py migrate`

Setting up `Admin Site`

Create `superuser`

Administrators of site are `superusers` and have access rights to all privileges (actions) and also maintain the privileges of users of the site. A user has restrictive access, typically read/write of their data.

1. Create a superuser

   `python manage.py createsuperuser`

2. A prompt guides setting up `superuser` credentials:

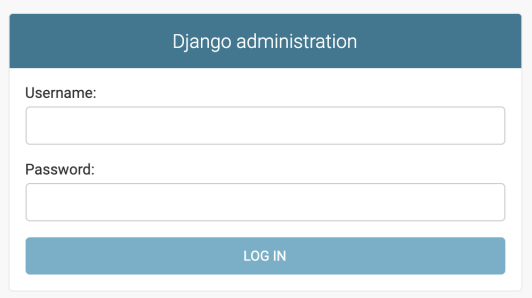
- Username *you can leave this blank to user current user logged into computer*
- email *this can be left blank*
- password *you will need to enter this twice*

Register Model

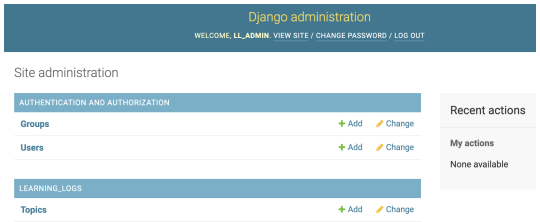
Any models added to `models.py` must be registered manually in `admin.py`

```
    from django.contrib import admin

    # Register your models here.
    from learning_logs.models import Topic  # import Topic model to be
registered
    admin.site.register(Topic)  # registers Topic to be managed via admin
site
```

- The admin can now be seen on a browser directed to `http://127.0.0.1:8000/admin`



1. Login using the `superuser` credentials
2. The Site Adminstration interface should now display



Adding data to the database

Topics can be added to the database by clicking `Add`

Create `Entry` model

A `many-to-one` relationship allows `many entries` to be associated with a single `topic`. This relationship may also be defined as a `one-to-many` where a single `topic` can have many `entries`.

In the `Entry class` model, a `foreign key` will relate an `entry` with a specific `topic` value.

1. `Add` the `Entry` class to `models.py`

```
class Entry(models.Model): # inherit from Django's base Model class
    """ Something specific learned about a topic """
    topic = models.ForeignKey(Topic,  on_delete=models.CASCADE)  #
connects entry to a topic
    text = models.TextField()
    date_added = models.DateTimeField(auto_now_add=True)
    # tells Django to use  the word 'entries' deal with multiple entries
    # otherwise Django creates the word 'entrys'
    class Meta:
        verbose_name_plural = 'entries'

    # show only first 50 chars of text
    def __str__(self):
        """ Return a string representation of the model """
        return self.text[:50] + "..."
```

A `many-to-many` relationship allows `many entries` to be associated with a `topic` as well as `many topics` to be associated with an `entry`. Creating a `many-to-many` relationship often involves connecting two `many-to-one` relationships using a `join table`.

2. `Build` this migrations file with `makemigrations` `python manage.py makemigrations learning_logs`

```
Migrations for 'learning_logs':
learning_logs/migrations/0002_entry.py
- Create model Entry
```

3. `Run` the migration `python manage.py migrate`

```
Migrations for 'learning_logs':
learning_logs/migrations/0002_entry.py
    - Create model Entry
(ll_env) 🤓 [skutz@ ~/Documents/GitHub/learning_log_django $ python
manage.py migrate
```

```
Operations to perform:
Apply all migrations: admin, auth, contenttypes, learning_logs, sessions
Running migrations:
Applying learning_logs.0002_entry... OK
```

The typical flow to update the database is:

1. Add specific `class` model to `models.py`
2. Run `makemigrations` to build a `migrations` file `python manage.py makemigrations` *appname*
3. Run `migrations` `python manage.py migrate`

Register `Entry` model

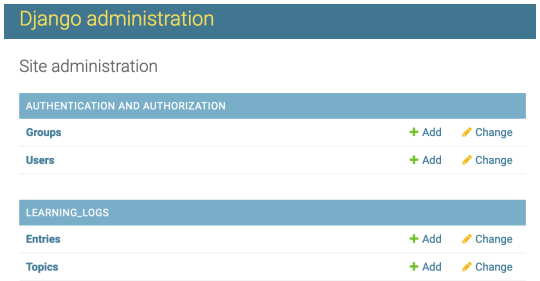
1. Import the `Entry` model

```
from .models import Topic, Entry    # Entry added
```

2. Register the `Entry` model

```
admin.site.register(Entry)
```

3. Run the server
   - `python manage.py runserver`
4. Verify that the admin panel shows entries



5. Add simple entry info to verify that info is saved to database

Mapping URLs and creating Views

- the imported `path` function maps URLs to views
- the `. import views` imports `views.py` view from within same folder
- the `app_name` variable defines the `urls.py` file from others with the same name
- the `urlpaterns` variable defines the pages available within the app
- the `path` function receives:
  - the `route` of the base URL
  - a call to the `index` function within `views.py`

- defines a the name of the link to the index page

1. Inside the `learning_log` folder, add the module `learning_logs.urls` to `urls.py`

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('learning_logs.urls')),
]
```

2. Inside the `learning_logs` folder(note difference) from above, create another `urls.py` and add the following:

```
from django.urls import path

from . import views

app_name = 'learning_logs'
urlpatterns = [
    # Home page
    path('', views.index, name = 'index'),
]
```

Creating Home page `view`

1. Inside the `learning_logs` folder, add the following to `views.py`

```
# Create your views here.
def index(request):
    """ The home page for Learning Log """
    return render(request, 'learning_logs/index.html')
```

The render function receives:

- the `request object`
- a `template` used to build define how Django renders a page

Creating a base template

1. Inside the `template/learning_logs` folder, create a new file `base.html`. This will have a core set of elements that other pages will inherit.

2. Using a `template tag`, create a `namespace` and `index`. We can pass in various `namespace` as a `URL pattern` in `url.py`

```
<p>
    <a href = "{% url 'learning logs:index' %}"> Learning Log </a>
</p>
{% block content %}{% endblock content %}
```

3. See Django documentation on templates for more info

Creating a child template

1. The original `index.html` is modified to inherit content from the base template `base.html`

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

{% block content %}

<p> Learning Log helps you keep track of you learning, for any topic you
are learning about </p>
{% endblock content %}
```

Creating Topics page

1. Add to `urls.py` beneath `path('', views.index, name = 'index'),`

```
app_name = 'learning_logs'
urlpatterns = [
    # Home page
    path('', views.index, name = 'index'),
    # path that shows all topics
    path('topics/', views.topics, name = 'topics'),

]
```

2. Add to the top of `views.py`

- This is needed to import the data needed

```
from .models import Topic
```

1. Add the `topics()` function to `views.py`

```
def topics(request):
""" Show all topics """
topics = Topic.objects.order_by('data_added')
context = {'topics': topics}
return render(request, 'learning_logs/topics.html', context)
```

- The request object is returned from the server and a query is set up. A `context` (e.g. dictionary with keys which represent names for the types of data) is returned to the template

Creating Topics template

1. The `{% extends %}` tag tells `Django` to inherit from the defined base.html file.

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

2. Within the `{% block content %}` tags, a special tag that emulates a `for loop` is implemented. This will render topics if present. Otherwise default content is provided

```
{% for topic in topics %}
        <li> {{topic}} </li>
    {% empty %}
        <li> No topics have been added yet </li>
{% endfor%}
```

3. Within `base.html`, add a link to Topics. The dash after the first link is used to help separate the links for the user.

```
<p>
    <a href = "{% url 'learning_logs:index' %}"> Learning Log </a> -
    <a href = "{% url 'learning_logs:topics' %}"> Topics </a>
</p>
```

Creating individual topics page

1. Append the following into `urls.py`. The `topic_id` is used to call `topic()` with a unique database entry

```
# detail page for a single topic based on id
path('topics/<int:topic_id>/', views.topic, name = 'topic')
```

2. Create a `view function` within `views.py` that accepts a parameter for the topic_id in addition to the `request object`. Queries are created for `topic` and `entries`.

```
def topic(request, topic_id):
    """ Show a single topic and all of its entries"""
    topic = Topic.objects.get(id=topic_id)
    entries = topic.entry_set.order_by('-date_added')
    context = {'topic': topic, 'entries': entries}
    return render(request, 'learning_logs/topic.html', context)
```

3. Add a `topic.html` page that inherits from `base.html`.

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

{% block content %}

<p> Topic:: {{topic} </p>}

<p> Entries: </p>

<ul>
{% for entry in entries %}
    <li>
    <p>{{ entry.date_added|date:'M d, Y h:i' }}</p>
    <p>{{ entry.text|linebreaks }}</p>
    </li>
{% empty %}
    <li> There are no entries for this topic yet </li>
{% endfor %}
</ul>

{% endblock content %}
```

Creating forms for user input

New Topics Form

1. Using `Djangos` class `ModelForm`, create `forms.py` inside the application folder `learning_logs`
    - `TopicForm` inherits from `forms.ModelForm`
    - The `model` is based from the `Topic` model
    - The form will have a `text field` without a `label`

```
    from django import forms

    from .models import Topic

    class TopicForm(forms.ModelForm):
        class Meta:
            model = Topic
            fields = [ 'text' ]
            labels = { 'text': '' }
```

2. Add the `URL` for the form to `urls.py`

   - The URL pattern will direct requests to the `new_topic()` view function

```
urlpatterns = [
    # Home page
    path('', views.index, name = 'index'),
    # path that shows all topics
    path('topics/', views.topics, name = 'topics'),
    # detail page for a single topic based on id
    path('topics/<int:topic_id>/', views.topic, name = 'topic'),
    # form page for adding new topic
    path('new_topic/', views.new_topic, name = 'new_topic')
]
```

3. Create the `new_topic` view function in `views.py`

   - In addition to render, the `redirect` function is imported in order to send the user(e.g. redirect) to the topics back after a new topic is written to the database via `form.save()`
   - `form = TopicForm()` assigns an instance of TopicForm to the variable `form`
   - the `context dictionary` contains the template for the form

```
def new_topic(request):
""" Add new topic """
if (request.method != 'POST' and request.method == 'GET'):
    # There was no POST data submitted, return blank form
    form = TopicForm()
else:
    # POST data exists, process data within request.POST
    form = TopicForm(data = request.POST)
    if form.is_valid():
        form.save()
        return redirect('learning_logs:topics')

# Display a blank or invalid form
```

```
    context = {'form': form}
    return render(request, 'learning_logs/new_topic.html', context)
```

4. Create the form for creating a new topic
    - The `new_topic` function receives the submitted form data as a `POST` request
    - `{% crsf token %}` prevents `cross-site request forgery` hacker attacks to the server
    - The `{{ form.as_p }}` template variable uses the `as_p` modifier to render the form as `p elements`

```
{% extends "learning_logs/base.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 a new topic </button>
    </form>

{% endblock content %}
```

5. Link the `new_topic` page within `topics.html`
    - Add the last line so that `topics.html` now looks like

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

{% block content %}

    <p> Topics </p>

    <ul>
        {% for topic in topics %}
            <li>
                <a href="{% url 'learning_logs:topic' topic.id %}">
{{ topic }}</a>
            </li>
        {% empty %}
            <li> No topics have been added yet </li>
        {% endfor%}
```

```
    </ul>

    <a href = "{% url 'learning_logs:new_topic' %}"> Add a new topic
</a>

{% endblock content %}
```

6. Append to the import in `forms.py` and add a class to manage user entries

    - the default Django widget is overridden with the custom widget attribute to make the test area 80 columns wide(default would be 40 columns)
    - The labels field is assigned a blank label

```
from .models import Topic, Entry

...

class EntryForm(forms.ModelForm):
    class Meta:
        model = Entry
        fields = ['text']
        labels = {'text': ''}
        widget = {'text': forms.Textarea(attrs = {'cols': 80})}
```

New Entries Form

1. Add URL for new_entry to `urls.py`

```
# Page for new entries
path('new_entry/<int:topic_id>/', views.new_entry, name = 'new_entry'),
```

2. Create `new_entry` View function in `views.py`

    - topic variable is assigned to specific id of topic
    - if request method is `POST`, create an instance of `EntryForm` using the returned `POST` data
    - the `commit=False` argument prevents the database from being updated. The `new_entry` object also requires the `topic` attribute to be set with the `topic_id` before writing to the database
    - the `redirect` function requires 2 arguments this time since the `topic` function requires the specified `view` and the specific `topic_id`

3. Create template for `new_entry`

```
{% 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 %}
```

4. Link `new_entry` page in `topic.html`

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

Edit Entry Form

1. Add URL for edit_entry to `urls.py`

```
# form page for editing entries based on entry_id
    path('edit_entry/<int:entry_id>/' , views.edit_entry, name =
'edit_entry'),
```

2. Create `edit_entry` View function in `views.py`
    - Make sure that the `Entry` model is imported at the top of `views.py`
    - the `entry_id` and `topic_id` are stored from the `Entry` object
    - the `GET` request will populate the form with current data in the entry object
    - the `POST` request will populate the form with updated data from the entry object
    - the `redirect` returns the user to the topic page with the recent updates
    - the `edit_entry` template is returned if either
        - the initial form for editing is returned
        - the submitted form is invalid

```
def edit_entry(request, entry_id):
    """ Edit an exiting entry """
    entry = Entry.objects.get(id = entry_id)
    topic = entry.topic

    if (request.method != 'POST' and request.method == 'GET'):
        # GET request => this is an initial request, pre-fill form
with current entry data
        form = EntryForm(instance=entry)
    else:
        # POST request => updated data submitted, fill form with
updated data
        form = EntryForm(instance = entry, data = request.POST)
        if form.is_valid():
            form.save()
            return redirect('learning_logs:topic', topic_id =
topic.id)

    context = {'entry': entry, 'topic': topic, 'form': form}
    return render(request, 'learning_logs/edit_entry.html', context)
```

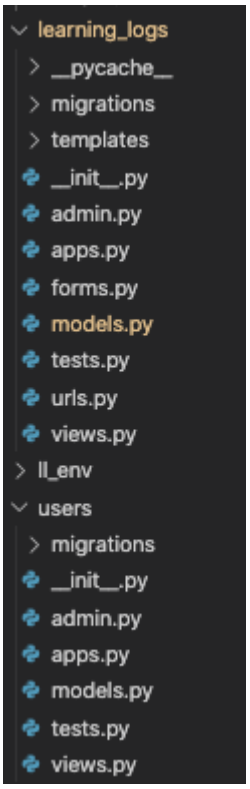
3. Link to edit_entry page to each topic in `topic.html`

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

...

<ul>
{% for entry in entries %}
    <li>
        <p>{{ entry.date_added|date:'M d, Y H:i' }}</p>
        <p>{{ entry.text|linebreaks }}</p>
        <p>
            <a href = "{% url 'learning_logs:edit_entry' entry.id %}">
Edit entry </a>
        </p>
    </li>
...
{% endblock content %}
```

Setting up User Accounts

- a new app will used to manage user accounts
- The `Topic` model will adjusted to reflect each user's topics

1. From the virtual environment launched in terminal, create the users app `python manage.py startapp users`

- A similar directory structure to the `learning_logs` app is created 

1. Add the users app to `INSTALLED_APPS` in `settings.py` ~~~ py INSTALLED_APPS = ['django.contrib.admin', 'django.contrib.auth', 'django.contrib.contenttypes', 'django.contrib.sessions', 'django.contrib.messages', 'django.contrib.staticfiles', # my apps 'learning_logs', 'users',] ~~~
    1. Add URL from users

```
...
urlpatterns = [
    path('admin/', admin.site.urls),
    path('users/', include('users.urls')),
    path('', include('learning_logs.urls')),     # added
]
```

Building the Login Page

1. Create a new `urls.py` file
    - Django provides a default `login view`
    - a new `urls.py` is created inside the `learning_log/users` folder
    - the `path` & `include` function must be imported in order to implemement Django's default authentication
    - the login URL will be at `localhost:8000/users/login`
    - within this path the word `users` directs Django to use `users/urls.py`

- within this path, the word `login` directs Django to send requests to the default login `view`

```
""" Define URL patterns for user """

from django.urls import path, include

app_name = 'users'

urlpatterns = [
    # Include the default authentication urls
    path('', include('django.contrib.auth.urls'))

]
```

2. Create a `login.html`
   - this file is based upon the main `base.html`
   - Django allows templates to be inherited between different apps. This allows `base.html` from the path `learning_logs/templates/learning_logs/base.html` to be used in `users/tempaltes/registration/login.html`
   - the hidden form element `next` provides Django with redirect info for a successful login which is at `index`, the home page

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

{% block content %}
    {% if form.errors %}
        <p> Username and password did NOT match, Please re-enter </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>
{% endblock content %}
```

3. Modify the original `base.html` to handle logged in/out users

- since every template has its own `user` variable, the `is_authenticated` attribute will either True or False if logged in
- a greeting for logged in users is setup if True
- a login prompt is setup if False

```
<p>
    <a href = "{% url 'learning_logs:index' %}"> Learning Log </a> -
    <a href = "{% url 'learning_logs:topics' %}"> Topics </a> -
    {% if user.is_authenticated %}
        Hello, {{ user.username }}
    {% else %}
        <a href = "{% url 'users:login' %}"> Log in </a>
    {% endif %}
</p>

{% block content %}{% endblock content %}
```

Building the Logout page

1. Append a link to logout to `base.html`

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

2. Create a simple page to indicate logged out status

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

{% block content %}
    <p> You are now logged out </p>

{% endblock content %}
```

3. `IMPORTANT` The logout page will not be shown and the admin panel logged out message will appear unless the apps you created come before `django.contrib.admin` within `INSTALLED_APP` inside of `settings.py`

```
INSTALLED_APPS = [
    # apps I created must come before django.contrib.admin in order for
logout page to be shown, otherwise admin logout is shown
    'learning_logs',
    'users',


    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

]
```

Building the Registration page

- the registration page has its own view and requires the `views module` to be imported in `users/urls.py`
- the URL pattern `http://localhost:8000/users/register/` will send requests to the `register()` function

1. Update the `users/urls.py` file

```
""" Define URL patterns for user """

from django.urls import path, include  # for login page
from . import views   # for registration page

app_name = 'users'

urlpatterns = [
    # Include the default authentication urls
    path('', include('django.contrib.auth.urls')),
    # Define URL pattern for registration
    path('register/', views.register, name = 'register'),

]
```

2. Using `Django's` class `UserCreationForm`, create a view within `users/views.py` for the `register()` view function
    - both `render` and `redirect need to be imported`
    - the login() function must be imported to process credentials
    - `UserCreationForm` must be imported

- a `GET` request creates a new instance of `UserCreationForm` that contains no data
- `POST` request data creates a new instance of `UserCreationForm`
- a valid form:
  - writes the user and password hash to the database
  - creates a session for the user with the `request` and `new_user` objects
  - redirects user to the own homepage

```
from django.shortcuts import render, redirect
from django.contrib.auth import login
from django.contrib.auth.forms import UserCreationForm

# Create your views here.

def register(request):
    """ Regsiter a new user if GET request made """
    if request.method != 'POST' and request.method == 'GET':
        # present user a registration form with no data
        form = UserCreationForm()
    else:
        # process a completed form with POST data
        form = UserCreationForm(data = request.POST)

        if form.is_valid():
            new_user = form.save()
            # Log in user and the redirect to home page

            login(request, new_user)
            return redirect('learning_logs:index')

    # Display blank or invalid form
    context = { 'form': form }
    return render(request, 'registration/register.html', context)
```

3. Build template for user registration
   - Django automatically notifies user if login form is not filled in correctly

```
{% 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 %}
 ~~~
```

4. Modify `base.html` to show registration link if user not logged in

```
    ...
    {% else %}
        <a href = "{% url 'users:register' %}"> Register </a> -
        <a href = "{% url 'users:login' %}"> Log in </a>
    {% endif %}
    ...
```

Restricting access to pages

- Each topic is owned by a user
- the @login_required decorator(a directive placed before a function) verified if a user is logged in
- users not logged in are redirected to the login page

1. Import the `login_required()` decorator to `learning_logs/views.py`

```
from django.contrib.auth.decorators import login_required
```

2. Direct `Django` to the location of the login page in `settings.py`

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

3. Restrict access to pages by adding the `@login_required` decorator to all of the views in `learning_logs/views.py` except `index()`

Connnect data to specific users

1. Add the User model to `learning/logs/models.py`

```
from django.contrib.auth.models import User
```

2. Add a new field called `owner` to `Topic`
    - This will create a foreign key relationship with the `User` model
    - the `on_delete=models.CASCADE` option will remove all topic associated with a deleted user

```
class Topic(models.Model): # inherit from Django's base Model class
    """ Topic that the use in learning about """
    text = models.CharField(max_length=200) # allocates 200 chars in database
    data_added = models.DateTimeField(auto_now_add=True) # sets to current data/time when new topic created
    owner = models.ForeignKey(User, on_delete=models.CASCADE)
```

Migrate the database

1. `python manage.py makemigrations learning_logs`
2. Select option 1 at the prompt choice to `provide a one-off default now`
3. Enter a default value of 1
4. Complete the migration by entering `python manage.py migrate`

Restrict Topics to particular users

1. Modify `views.py` to only show topics that belong to a specific user
    - only topics objects that match the currently logged in user request.user are provided to the `topics` query object

```
    @login_required
    def topics(request):
        """ Show all topics """
        # topics = Topic.objects.order_by('data_added')
        topics = Topic.objects.filter(owner=request.user).order_by('data_added')
        context = {'topics': topics}
```

2. Protect users topics from direct URL links

- add `from django.http import Http404` to top of `views.py`

```
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import login_required
from django.http import Http404
```

- redirect to 404 page if logged in user is not topic .owner by adding this to `def_topic` and `edit_entry`

```
    # Make sure the topic belongs to the current user
    if topic.owner != request.user:
        raise Http404
```

- `def_topic` will now look like

```
@login_required
def topic(request, topic_id):
    """ Show a single topic and all of its entries"""
    topic = Topic.objects.get(id=topic_id)

    # Make sure the topic belongs to the current user
    if topic.owner != request.user:
        raise Http404

    entries = topic.entry_set.order_by('-date_added')
    context = {'topic': topic, 'entries': entries}
    return render(request, 'learning_logs/topic.html', context)
```

- `edit_entry` will now look like

```
@login_required
def edit_entry(request, entry_id):
    """ Edit an exiting entry """
    entry = Entry.objects.get(id = entry_id)
    topic = entry.topic

    # Make sure the topic belongs to the current user
    if topic.owner != request.user:
        raise Http404

    if (request.method != 'POST' and request.method == 'GET'):
        # GET request => this is an initial request, pre-fill form with
current entry data
        form = EntryForm(instance=entry)
    else:
        # POST request => updated data submitted, fill form with updated
data
        form = EntryForm(instance = entry, data = request.POST)
        if form.is_valid():
            form.save()
            return redirect('learning_logs:topic', topic_id = topic.id)
```

```
    context = {'entry': entry, 'topic': topic, 'form': form}
    return render(request, 'learning_logs/edit_entry.html', context)
```

- the error message `Django IntegrityError - NOT NULL constraint failed: learning_logs_topic.owner_id` will appear if a user tries to create a new topic
- the new topic must have a value in the user that owns)e.g. is associated with) that topic
- the `new_topic = form.save(commit=False)` prevents writing to the database
- the `new_topic.owner = request.user` creates the association between the user and topic
- `new_topic.save()` writes the new topic to the database
- after modifying `new_topic` , new topics & entries only specific to a particular users can now be created

```
@login_required
def new_topic(request):
    """ Add new topic """
    if (request.method != 'POST' and request.method == 'GET'):
        # There was no POST data sumitted, return blank form
        form = TopicForm()
    else:
        # POST data exists, process data within request.POST
        form = TopicForm(data = request.POST)
        if form.is_valid():
            new_topic = form.save(commit=False)
            new_topic.owner = request.user
            new_topic.save()

            # form.save()
            return redirect('learning_logs:topics')

    # Display a blank or invalid form
    context = {'form': form}
    return render(request, 'learning_logs/new_topic.html', context)
```

Refactor for restricting users topics

```
- a function called `check_new_entry` is added into
    - new_entry
    - edit_entry
    - topic
- this prevents a user from attempting to access the topic URL of another
user and creating new entries
```

```
def check_topic_owner(topic, request):
    if topic.owner != request.user:
        raise Http404
```

Styling !

Setup styling

1. From the virtual environment, install `django-bootstrap4` using pip

```
pip install django-bootstrap4
```

2. Update the `INSTALLED_APPS` section of `settings.py`

```
...
INSTALLED_APPS = [
    # my apps
    'learning_logs',
    'users',

    # 3rd party apps
    'bootstrap4',

    'django.contrib.admin',
...
```

3. Replace `learning_logs/base.html` with the following

- `{% load bootstrap4 %}` loads the available template tags in `django-bootstrap4`
- within the `head` section, the following loads all available Bootstrap style files

```
{% load bootstrap4 %}

<!doctype html>
<html lang="en" >

<head>
    <meta charset = "utf-8">
    <meta name = "viewport" content = "width=device-width",
        initial-scale=1,
        shrink-to-fit="no"
    >
```

```
    <title>Learning Log</title>

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

</head>

<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>

            <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>
                <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>
    <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>
```

4. Update index.html with the following

```
{% 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've learned about. Whenever you learn something new
about a topc,
        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 %}
```

5. Update `login.html`

```
{% 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 %}
```

5. Update `topic.html`

```
{% 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 %}
    <li> There are no entries for this topic yet </li>
```

```
{% endfor %}


{% endblock content %}
```