Create Band Website with Django

Estimated time: 90 minutes

Welcome to the **Create Band Website wth Django** hands-on lab. In this lab, you will create the main website for the band. The lab provides a GitHub template repository to get you started.

At the end of the lab, you will:

1. Create the main home page for the band
2. Create the songs page that lists all the titles of the songs
3. Craete the photos page that displays pictures from past events
4. Allow users to log in and:
    1. See concerts they are attending
    2. Change attendance at concerts
5. Allow admin user to create, delete, and edit concerts

Objectives

In this lab, you will:

- Start a Django Server
- Create and run database migrations
- Create a Django admin user
- Create backend views for the Django application
- Send JSON data to templates
- Create Django models
- Check for authenticated users in Django backend

Note: Important Security Information

Welcome to the Cloud IDE. This is where all your development will take place. It has all the tools you will need to use, including **Python** and **Flask**.

It is important to understand that the lab environment is ephemeral. It only lives for a short while before it is destroyed. It is imperative that you push all changes made to your own GitHub repository so that it can be recreated in a new lab environment any time it is required.

Also, note that this environment is shared and, therefore, not secure. You should not store any personal information, usernames, passwords, or access tokens in this environment for any purpose.

Your Task

If you haven't generated a GitHub Personal Access Token you should do so now. You will need it to push code back to your repository. It should have `repo` and `write` permissions, and be set to expire in `60` days. When Git prompts you for a password in the Cloud IDE environment, use your Personal Access Token instead. Follow the steps in the Generating Git Token Lab for detailed instructions.

The environment may be recreated at any time, so you may find that you have to perform the Initialize Development Environment each time the environment is created.

Note on Screenshots

Throughout this lab, you will be prompted to take screenshots and save them on your device. You will need these screenshots to either answer graded quiz questions or upload the screenshots as your submission for peer review at the end of this course. Your screenshot must have either the .jpg or .png extension.

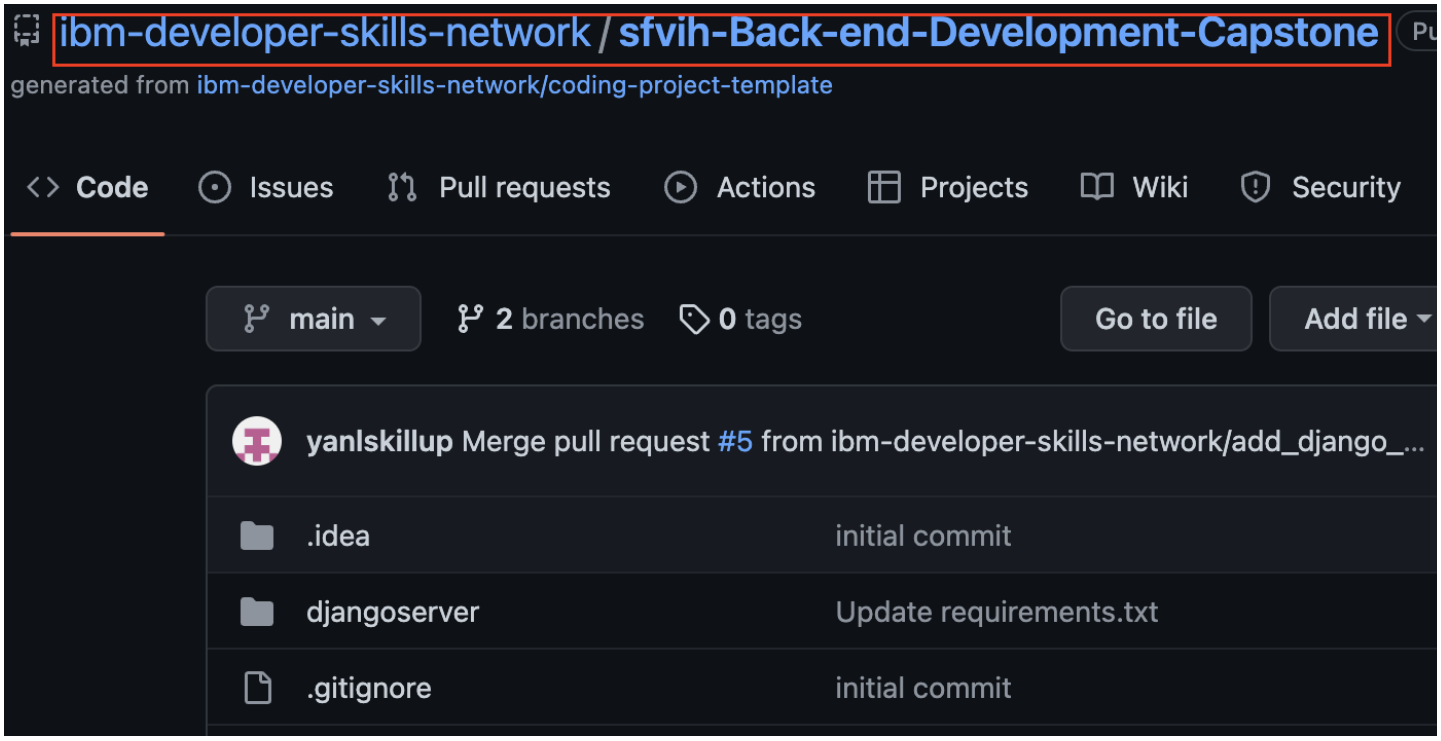
To take screenshots, you can use various free screen-capture tools or your operating system's shortcut keys. For example:

- Mac: you can use `Shift + Command + 3 (⇧ + ⌘ + 3)` on your keyboard to capture your entire screen, or `Shift + Command + 4 (⇧ + ⌘ + 4)` to capture a window or area. It will be saved as a .jpg or .png file on your Desktop.
- Windows: you can capture your active window by pressing `Alt + Print Screen` on your keyboard. This command copies an image of your active window to the clipboard. Next, open an image editor, paste the image from your clipboard to the image editor, and save the image as .jpg or .png.

Create New Repository from Template

1. Click this URL to open the starter code project: https://github.com/ibm-developer-skills-network/sfvih-Back-end-Development-Capstone
2. Use the green **Use this template** button to clone this repository to your own private GitHub account.

   Do not use Fork; use the Template button.



3. Give your repository the name `Back-end-Development-Capstone`. This is the name that graders will be looking for to grade your work.
4. Ensure you select the `Public` option for your repository and then create it.

Evidence

1. Note down the URL of your GitHub repository (not the template) to submit for peer review. Recall the graders are looking for a repository named `Back-end-Development-Capstone` in your account.

Initialize Development Environment

Because the Cloud IDE environment is ephemeral, it may be deleted at any time. The next time you come into the lab, a new environment may be created. Unfortunately, this means that you will need to initialize your development environment every time it is recreated. This shouldn't happen too often as the environment can last for several days at a time, but when it is removed, this is the procedure to recreate it.

Overview

Each time you need to set up your lab development environment, you will need to run three commands.

Each command will be explained in further detail, one at a time.

The commands include:

```
1
2
3
4
```

```
git clone https://github.com/$GITHUB_ACCOUNT/Back-end-Development-Capstone.git
cd /home/project/Back-end-Development-Capstone
bash ./bin/setup.sh
exit
```

Copied!

Now, let's discuss each of these commands and explain what needs to be done.

Task 1 - Clone the repository

Initialize your environment using the following steps:

1. Open a terminal with `Terminal` > `New Terminal` if one is not open already.
2. Next, use the export GITHUB_ACCOUNT command to export an environment variable that contains the name of your GitHub account.

    Note: Substitute your real GitHub account for the {your_github_account} placeholder below:

```
1
```

```
export GITHUB_ACCOUNT={your_github_account}
```

Copied! Executed!

3. Then use the following commands to clone your repository.

```
1
```

```
git clone https://github.com/$GITHUB_ACCOUNT/Back-end-Development-Capstone.git
```

Copied! Executed!

Task 2 - Initialize the development environment

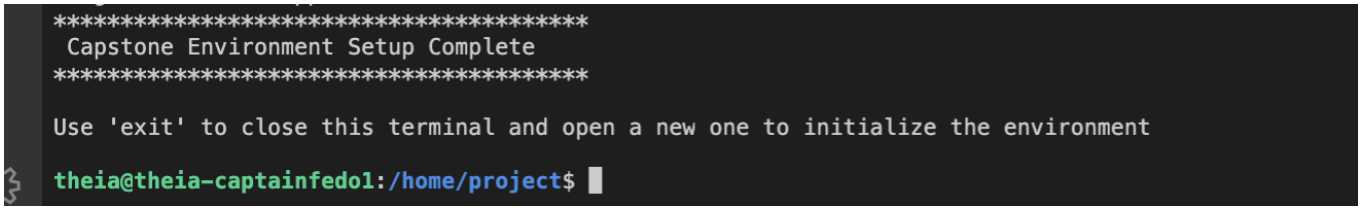
4. Change to the Back-end-Development-Capstone directory, and execute the `./bin/setup.sh` command.

```
1
2
```

```
1. cd /home/project/Back-end-Development-Capstone
2. bash ./bin/setup.sh
```

Copied! Executed!

5. You should see the follow at the end of the setup execution:



6. Finally, use the `exit` command to close the current terminal. The environment will not be fully active until you open a new terminal in the next step.

```
1. 1
```

```
1. exit
```

Copied! Executed!

Validate

To validate that your environment is working correctly, you must open a new terminal because the Python virtual environment will only activate when a new terminal is created. You should have ended the previous task by using the `exit` command to exit the terminal.

1. Open a terminal with `Terminal` > `New Terminal` and check that everything worked correctly by using the `which python` command:

   Check which Python you are using:

   ```
   1. 1
   1. which python
   ```
   Copied! Executed!

   You should get back:

   ```
   1. 1
   2. 2
   1. (backend-django-venv) theia:project$ which python
   2. /home/theia/backend-django-venv/bin/python
   ```
   Copied!

   Check the Python version:

   ```
   1. 1
   1. python --version
   ```
   Copied! Executed!

   You should get back some patch level of Python 3.8:

   ```
   1. 1
   1. Python 3.8.0
   ```
   Copied!

This completes the setup of the development environment. Anytime your environment is recreated, you will need to follow this procedure.

Project Overview

You created the `songs` and the `photos` microservices in the last two modules. You are now asked to create the main `Django` website for the band. An incomplete template has been provided by the previous developer. You have already created a repository in your GitHub account from this template. You will now complete the following exercises to run this Django application locally in the lab environment.

Exercise 1: Complete Django Data Models

Migrations are Django's way of propagating changes you make to your models (adding a field, deleting a model, and others) into your database schema. They are designed to be mostly automatic, but you will need to know when to make migrations, when to run them, and the common problems you might encounter.

There are several commands which you will use to interact with migrations and Django's handling of database schema:

- **migrate**, responsible for applying and dissociating migrations.
- **makemigrations**, responsible for creating new migrations based on the changes you have made to your models.
- **sqlmigrate**, displays the SQL statements for a migration.
- **showmigrations**, lists a project's migrations and their status.

Before you make and run migrations, you need to finish the data models in the code. The classes are provided and you will fill out the data types in this exercise.

Your Tasks

Task 1: Create a branch to work on

1. Change into the project directory.

   ```
   1. 1
   1. cd /home/project/Back-end-Development-Capstone
   ```
   Copied! Executed!

2. Since you are working in GitHub, you must pull the latest changes from the main branch to stay up to date. You can then create a new branch.

   The steps are:

   ```
   1. 1
   2. 2
   3. 3
   1. git checkout main
   2. git pull
   3. git checkout -b backend-rest
   ```
   Copied!

   This will switch to the main branch, pull the latest changes, and create a new branch. You will be asked to push all your changes to your GitHub repo and merge all code back into your main branch with a pull request.

You can use the `git branch` command to see your current branch:

```
1
```

```
git branch
```

Copied! Executed!

Your output should look something like this:

```
1
2
3
```

```
$ git branch
* backend-rest
  main
```

Copied!

Task 2: Complete the data model classes

Open the `Back-end-Development-Capstone/concert/models.py` file in the editor.

Open **models.py** in IDE

1. You will notice that the previous developer left the `Concert` model properties commented out.

```
1
2
3
4
5
class Concert(models.Model):
    # concert_name
    # duration
    # city
    # date
```

Copied!

2. Let's add these properites with the following attributes:

    - `concert_name`: CharField with a max_length of 255
    - `duration`: an IntegerField
    - `city`: CharField with a max_length of 255
    - `date`: DateField with the default time of now

3. Similarly, you will notice the `Photo` model commented out:

```
1
2
3
4
5
6
7
class Photo(models.Model):
    # id
    # pic_url
    # event_country
    # event_state
    # event_city
    # event_date
```

Copied!

4. You will fill it out with the following attributes:

    - `id`: IntegerField as a primary key
    - `pic_url`: CharField with max_length of 1000
    - `event_country`: CharField with max_length of 255
    - `event_state`: CharField with max_length of 255
    - `event_city`: CharField with max_length of 255
    - `event_date`: DateField with the default time of now

5. Finally, the `Song`model also needs all the attributes as follows:

    - `id`: IntegerField as a primary key
    - `title`: CharField with max_length of 255
    - `lyrics`: TextField

Solutions

1. `Concert` model should look as follows:

▶ Click here for the solution.

2. `Photo` model should look as follows:

▶ Click here for the solution.

3. `Song` model should look as follows:

▼ Click here for the solution.

```
1
2
3
4
class Song(models.Model):
    id = models.IntegerField(primary_key=True)
    title = models.CharField(max_length=255)
    lyrics = models.TextField()
```

Copied!

Task 3: Make and run migrations

1. Create the initial migrations using the `makemigrations` command.
▼ Click here for a hint.

```
1
python manage.py makemigrations
```

Copied! Executed!

2. Run the migrate command for the migrations you just created.
▼ Click here for a hint.

```
1. 1
1. python manage.py migrate
```
Copied! Executed!

Evidence

1. Take a screenshot of the terminal after executing the `migrate` command.
2. Save the screenshot as `django-migrate.jpg` (or `.png`).

Exercise 2: Run the Django App in the Lab Environment

Before you proceed with the rest of the backend exercises, you can run the application locally in the lab enivonrment and see the home page.

Your Tasks

1. Run the `runserver` command to run the application locally.

   ▼ Click here for a hint.

   ```
   1. 1
   1. python manage.py runserver
   ```
   Copied! Executed!

   This should start the Django server at port 8000.

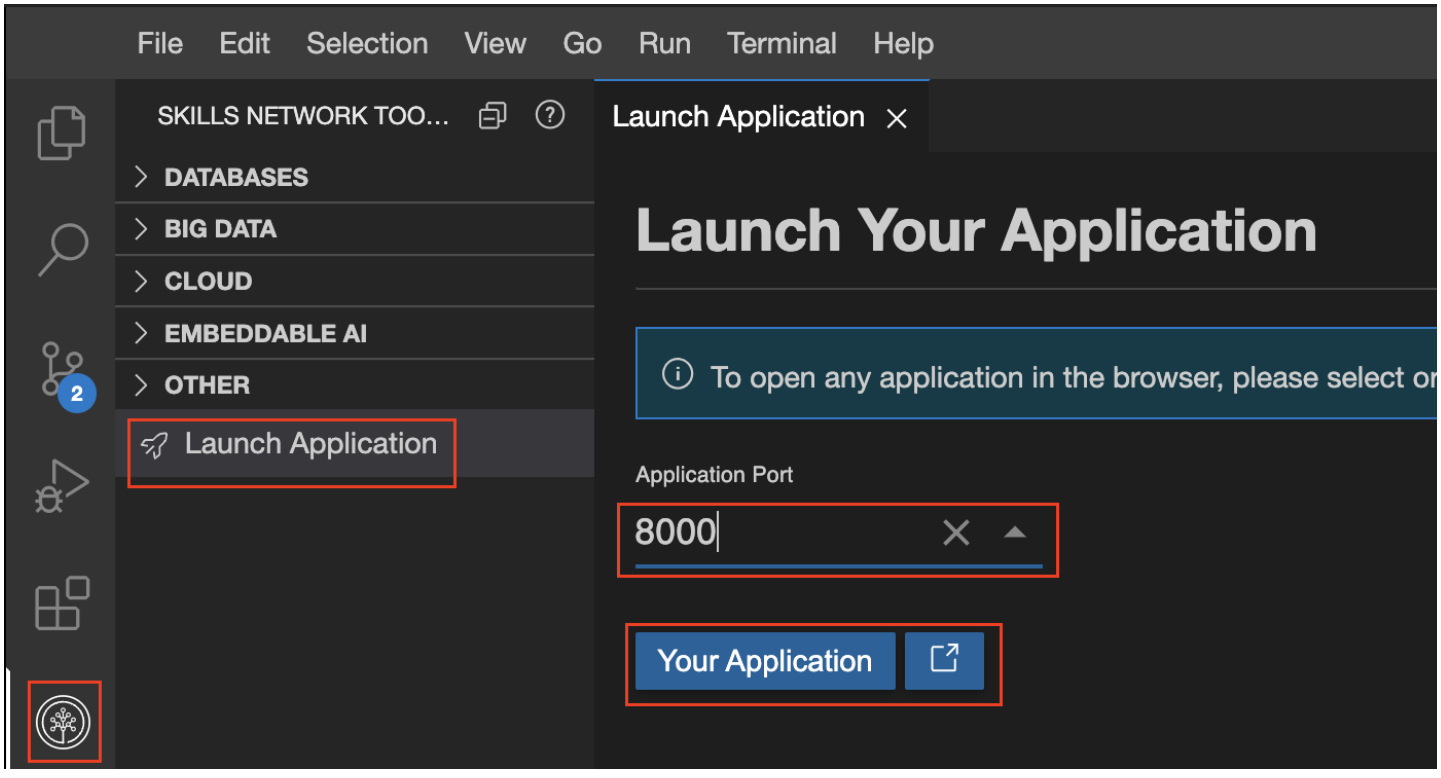
2. Curl `localhost:8000` to see if you get a 200 HTTP code from the running server. Note that you are already running the server in the terminal. Open a new terminal or split the existing terminal to execute the curl command:

   ```
   1. 1
   1. curl -s -o /dev/null -w "%{http_code}" http://localhost:8000
   ```
   Copied! Executed!

   You should see an output of `200`:

   ```
   1. 1
   2. 2
   3. 3
   1. (venv) theia:project$ curl -s -o /dev/null -w "%{http_code}" http://localhost:8000
   2.
   3. 200(venv)
   ```
   Copied!

3. If you got 200 OK, you can proceed to launch the application by clicking `Launch Applciation` icon on the left bar. Once the tab opens, you can enter port as `8000` and click the `Your Application` button.



You should see the application open in a new tab. Note that none of the links on the top of the page are working. The remaining exercises in the lab will guide you on how to get these links and the admin site working.

Evidence

1. Take a screenshot of the browser running your application.
2. Save the screenshot as `django-app-browser.jpg` (or `.png`).

Exercise 3: Fix Songs page

The `Songs` page displays all songs and lyrics for each when the user clicks on a song title. Currently, there is no URL mapped to the songs method in the view file. All the URLs are stored in `concert/urls.py` file. You need to add the URL for the songs.

Your Tasks

Task 1 : Fix songs URL

Open the `Back-end-Development-Capstone/concert/urls.py` file in the editor.

Open **urls.py** in IDE

1. You should see all the URLs defined in this file:

```
1
2
3
4
5
urlpatterns = [
    re_path(r"^$", views.index, name="index"),
    path("", views.songs, name="songs"),
    ...
]
```

Copied!

2. Replace the first empty string for the songs urlpattern with `song/`

▼ Click here for a hint.

```
1
path("songs/", views.songs, name="songs"),
```

Copied! Executed!

3. If you click on the songs link now, you will get an error. However, you have made progress. Instead of the link not working at all, you are one step closer to getting it to work.

Task 2 : Fix songs view

The songs link is working, but the view is still broken. Open the `Back-end-Development-Capstone/concert/views.py` file in the editor.

Open **views.py** in IDE

1. Go to the method named `songs`.

```
1
2
3
4
def songs(request):
    # songs = {"songs":[]}
    # return render(request, "songs.html", {"songs": [insert list here]})
    pass
```

Copied!

2. You will eventually hook this method with the song microservice you created in the previous lab. For now, let's return the following dummy data back to the `songs.html` template.

```
1
[{"id":1,"title":"duis faucibus accumsan odio curabitur convallis","lyrics":"Morbi non lectus. Aliquam sit amet diam in magna bibendum imperdiet. Nullam orci pede, venenatis non, sodales sec
```

Copied!

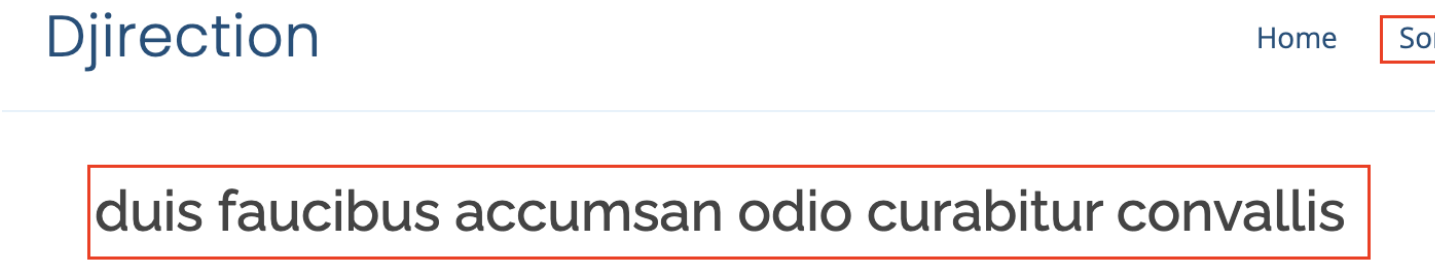
▼ Click here for a hint.

```
1
2
3
def songs(request):
    songs = {"songs":[{"id":1,"title":"duis faucibus accumsan odio curabitur convallis","lyrics":"Morbi non lectus. Aliquam sit amet diam in magna bibendum imperdiet. Nullam orci pede, vener
    return render(request, "songs.html", {"songs":songs["songs"]})
```

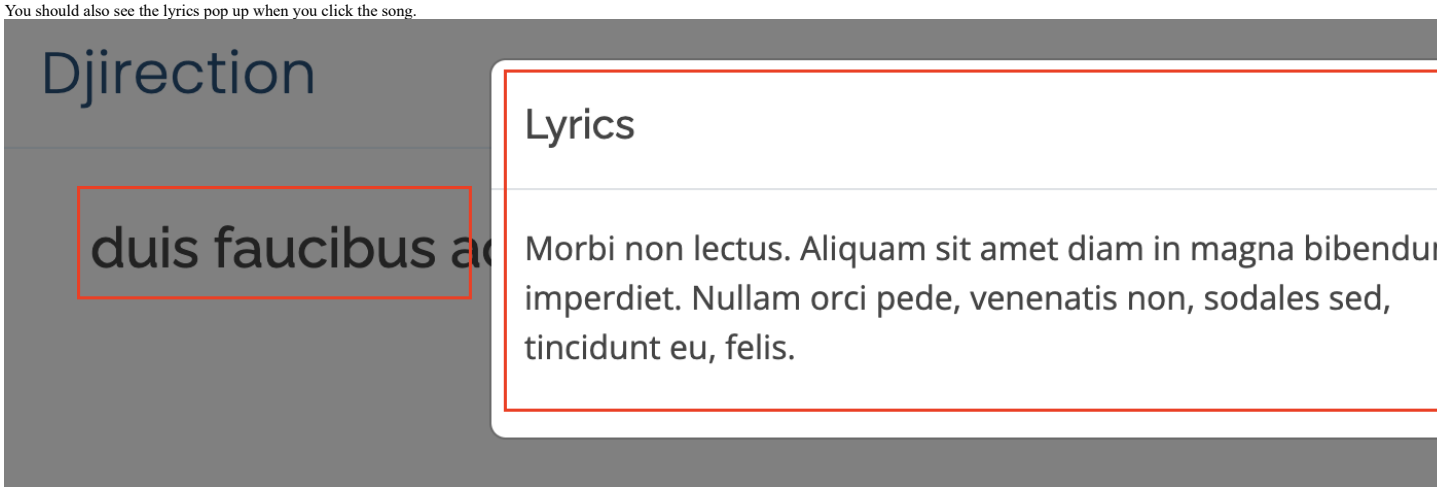
Copied!

> Note that you also need to remove the `pass` statement from the last line of the method.

3. You should now see a song dislayed when you click the `Songs` page.



4. You should also see the lyrics pop up when you click the song.



Evidence

1. Take a screenshot of your application showing the modal dialog along with the single song in the backgroud.
2. Save the screenshot as `django-song-modal.jpg` (or `.png`).

Exercise 4: Fix Photos page

The `Photos` page displays pictures from past events by the band in different cities across the world. Currently, there is no URL mapped to the photos method in the view file. All the URLs are stored in `concert/urls.py` file. You need to add the URL for the photos.

Your Tasks

Task 1 : Fix photos URL

Open the `Back-end-Development-Capstone/concert/urls.py` file in the editor.

Open **urls.py** in IDE

1. You should see all the URLs defined in this file:

```
1
2
3
4
5
6
urlpatterns = [
    re_path(r"^$", views.index, name="index"),
    path("songs", views.songs, name="songs"),
    path("", views.photos, name="photos"),
    ...
]
```

Copied!

2. Replace the first empty string for the photos urlpattern with `photos`.

▼ Click here for a hint.

```
1
path("photos/", views.photos, name="photos"),
```

Copied! Executed!

3. If you click on the photos link now, you will get an error. However, again, you have made progress.

Task 1 : Fix photos view

The photos link is working, but the view is still broken. Open the `Back-end-Development-Capstone/concert/views.py` file in the editor.

Open **views.py** in IDE

1. Go to the method named `photos`.

```
1
2
3
4
def photos(request):
    # photos = None
    # return render(request, "photos.html", {"photos": photos})
    pass
```

Copied!

2. You will eventually hook this method with the pictures microservice you created in the previous lab. For now, let's return the following dummy data back to the `photo.html` template.

```
1
2
3
4
5
6
7
8
[{
    "id": 1,
    "pic_url": "http://dummyimage.com/136x100.png/5fa2dd/ffffff",
    "event_country": "United States",
    "event_state": "District of Columbia",
    "event_city": "Washington",
    "event_date": "11/16/2022"
}]
```

Copied!

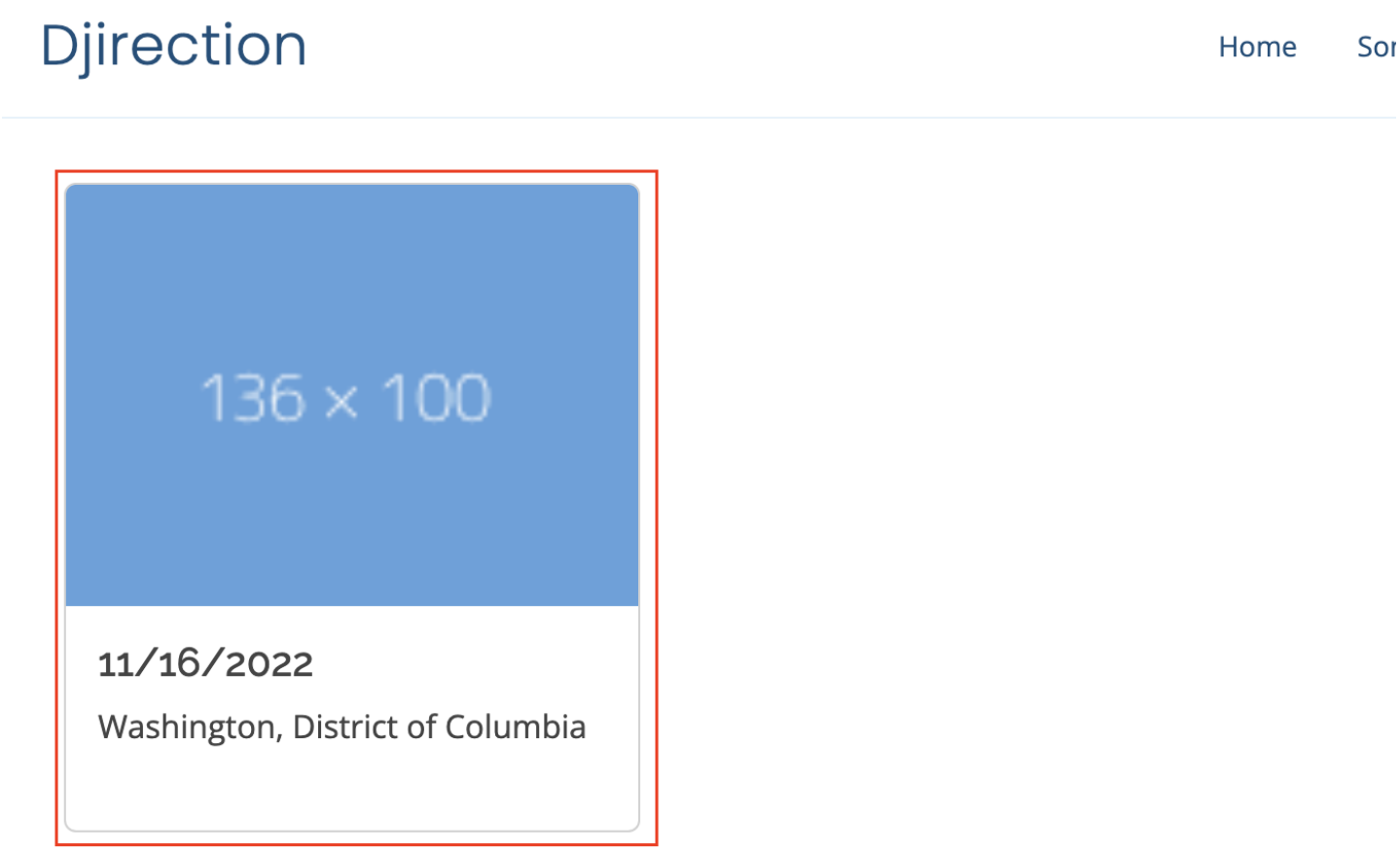
▼ Click here for a hint.

```
1
2
3
4
5
6
7
8
9
10
def photos(request):
    photos = [{
    "id": 1,
    "pic_url": "http://dummyimage.com/136x100.png/5fa2dd/ffffff",
    "event_country": "United States",
    "event_state": "District of Columbia",
    "event_city": "Washington",
    "event_date": "11/16/2022"
    }]
    return render(request, "photos.html", {"photos": photos})
```

Copied!

Note that you also need to remove the `pass` statement from the last line of the method.

3. You should now see a photo dislayed when you click the `photos` page



Evidence

1. Take a screenshot of your application showing the single photo in the photos page.
2. Save the screenshot as `django-photos.jpg` (or `.png`).

Exercise 5: Fix the Sign Up flow

The `Signup` flow allows the user to create an account with the site so they can book concerts.

Your Tasks

Task 1 : Fix `signup` URL

Open the `Back-end-Development-Capstone/concert/urls.py` file in the editor.

Open **urls.py** in IDE

1. You should see all the URLs defined in this file:

```
1
2
3
4
5
6
7
8
9
urlpatterns = [
```

```
    re_path(r"^$", views.index, name="index"),
    path("songs", views.songs, name="songs"),
    path("photos", views.photos, name="photos"),
    path("", views.login_view, name="login"),
    path("", views.logout_view, name="logout"),
    path("", views.signup, name="signup"),
    ...
]
```

Copied!

2. Replace the first empty string for the signup urlpattern with `signup/`.

▼ Click here for a hint.

```
1
path("signup/", views.signup, name="signup"),
```

Copied! Executed!

3. If you were to click the `Signup` link now, you will get an error. However, again, you have made progress and we will fix this error in the next Task.

Task 2 : Fix Signup View

The Signup link is working, but the view is still broken. Open the `Back-end-Development-Capstone/concert/views.py` file in the editor.

Open **views.py** in IDE

1. Go to the method named `signup`.

```
1
2
def signup(request):
    pass
```

Copied!

2. This view will handle both use cases when the user is asking for the signup form and also when the user is submitting the signup form after adding their details. Let's consider the first. You simply return `signup.html` template with the `SignUpForm` form.

```
1
return render(request, "signup.html", {"form": SignUpForm})
```

Copied!

3. The second case is a little more involved. You will have to check if the request is a `POST` request. A `POST` request will indicate the user submitted a form. There are a number of steps after this:

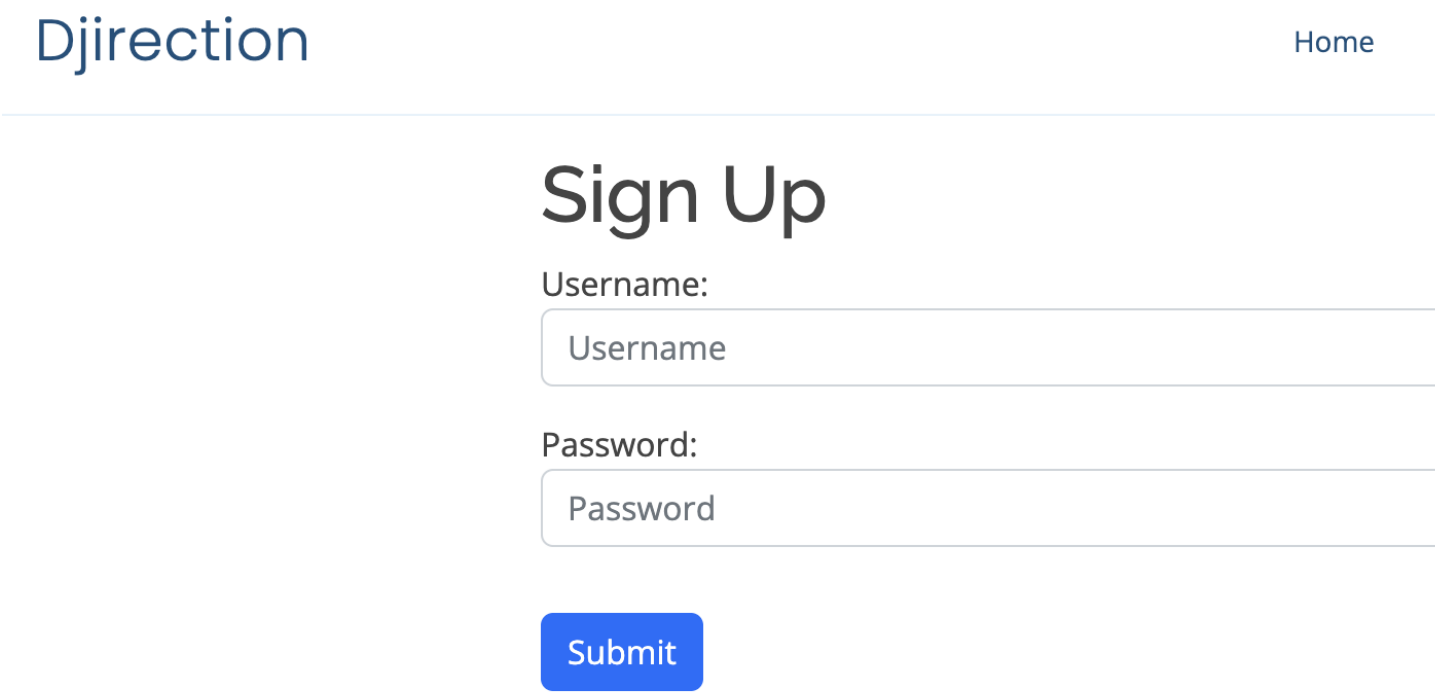
   - get the `username` from the request.
   - get the `password` from the request.
   - look for the user with this username.
   - if the user already exists, return to `Signup`page with the message `user already exists`.
   - if the user does not exist, log in the user wih their new credentials, and return to the index page.

   We have provided the following skeleton code to get you started:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
if request.method == "POST":
    username = {insert code to get username from the request}
    password = {insert code to get password from the request}
    try:
        user = {insert code to find user using User.objects.filter method}
        if user:
            return {insert code to render the signup.html page with the SignUpForm form and a message of "user already exist"}
        else:
            user = {insert code to create a new user using the User.objects.create method. Remmeber to use the make_password method to create the password securely}
            {insert code to log in the user with the django.contrib.aut. module}
            {insert code to return the user back to the index page}
    except User.DoesNotExist:
        return {insert code to render the 'signup.html' page with the 'SignUpForm' form}
return {insert code to render the 'signup.html' page with the 'SignUpForm' form}
```

Copied!

4. If your code works without errors, you should now see a sign up form when you click the `signup` link.



5. You should be able to create a new user and sign in. However, once you have signed in, the `logout` link is still broken. We will fix that in the next exercise.

Final Solution

1. Ensure your solution for the login_view method matches the following:

▼ Click here for the solution.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
```

```
def signup(request):
    if request.method == "POST":
        username = request.POST.get("username")
        password = request.POST.get("password")
        try:
            user = User.objects.filter(username=username).first()
            if user:
                return render(request, "signup.html", {"form": SignUpForm, "message": "user already exist"})
            else:
                user = User.objects.create(
                    username=username, password=make_password(password))
                login(request, user)
                return HttpResponseRedirect(reverse("index"))
        except User.DoesNotExist:
            return render(request, "signup.html", {"form": SignUpForm})
    return render(request, "signup.html", {"form": SignUpForm})
```

Copied!

Exercise 6: Fix Login and Logout

The `Login` flow allows the user to log in with their account. They are then able to see the concert page with a list of all concerts. The `Logout` flow will log the user out of the application. If you open the application in its current state, you will see the `Login` button, but nothing happens when you click it. Your task in this exercise is to fix both these views.

Your Tasks

Task 1 : Fix Login and Logout URL

Open the `Back-end-Development-Capstone/concert/urls.py` file in the editor.

Open **urls.py** in IDE

1. You should see all URLs defined in this file:

```
1
2
3
4
5
6
```

```
7. 7
8. 8
1. urlpatterns = [
2.     re_path(r"^$", views.index, name="index"),
3.     path("songs", views.songs, name="songs"),
4.     path("photos", views.photos, name="photos"),
5.     path("", views.login_view, name="login"),
6.     path("", views.logout_view, name="logout"),
7.     ...
8. ]
```
Copied!

2. Replace the first empty string for the login urlpattern with `login/` and the logout urlpattern with `logout/`.

▼ Click here for a hint.

```
1. 1
2. 2
1. path("login/", views.login_view, name="login"),
2. path("logout/", views.logout_view, name="logout"),
```
Copied!

3. If you click the `Login` link now, you will get an error. However, again, you have made progress and we will fix this error in the next Task.

Task 2 : Fix Login View

The Login link is working, but the view is still broken. Open the `Back-end-Development-Capstone/concert/views.py` file in the editor.

Open **views.py** in IDE

1. Go to the method named `login_view`.

```
1. 1
2. 2
1. def login_view(request):
2.     pass
```
Copied!

2. This view will handle both use cases when the user is asking for the login form and also when the user is submitting a login form. Let's consider the first. You simply return `login.html` template with the `LoginForm` form.

```
1. 1
1. return render(request, "login.html", {"form": LoginForm})
```
Copied!

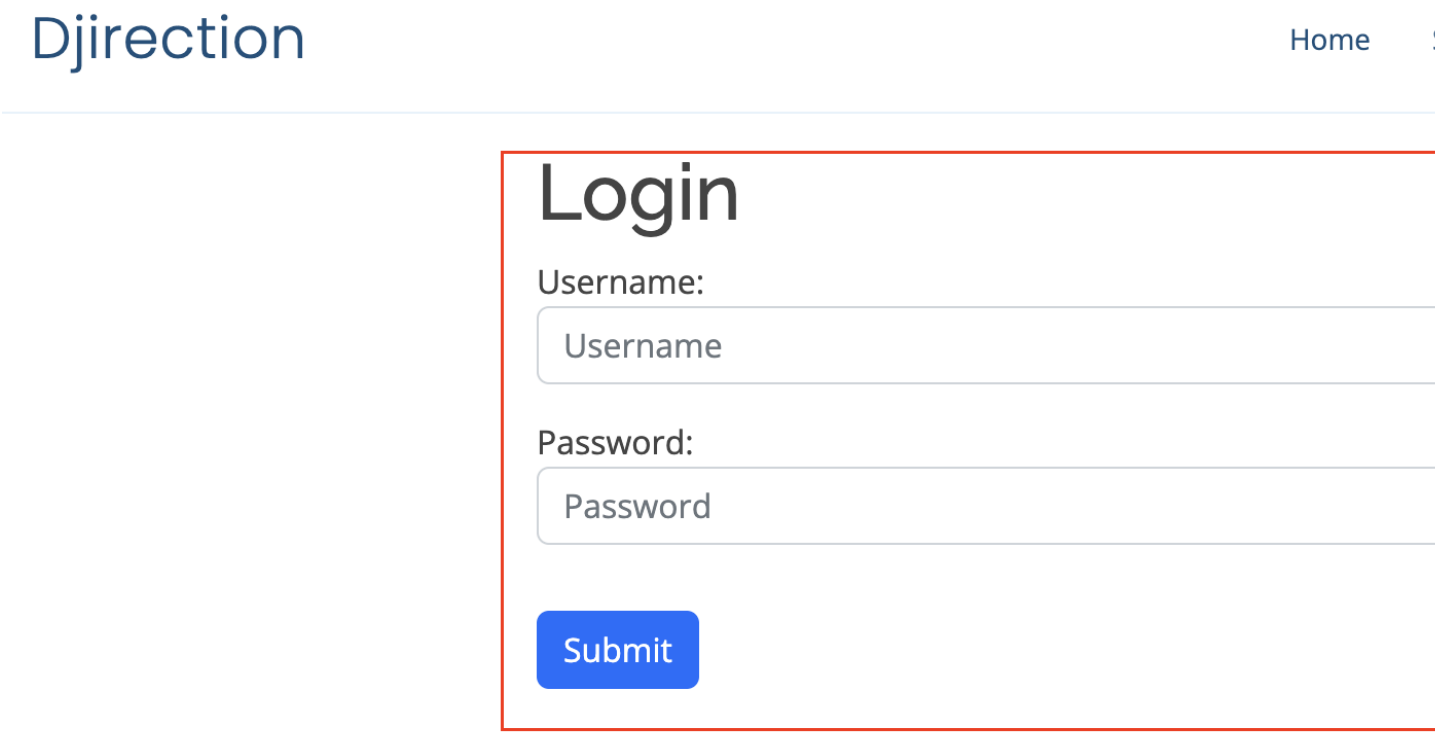
3. The second case is a little more involved. You will have to check if the request is a `POST` request. A `POST` request would indicate the user submitted a form. There are a number of steps after this:

   - get the `username` from the request
   - get the `password` from the request
   - if the username and password are correct, send them to the index page
   - if the username or password are incorrect, send them back to the login page

   We have provided the following skeleton code to get you started:

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
1. if request.method == "POST":
2.     if request.method == "POST":
3.     username = {insert code to get username from the request}
4.     password = {insert code to get password from the request}
5.     try:
6.         user = {insert code to find the user with the username}
7.
8.         if {insert code to check the username and password}:
9.             {insert code to log in the using the django.contrib.auth module}
10.             return HttpResponseRedirect(reverse("index"))
11.     except User.DoesNotExist:
12.         return {insert code to render the `login.html` method using the `LoginForm` form}
```
Copied!

4. If your code works without errors, you should now see a login form when you click the `Login` button.



You can try logging in with the user you created in the previous task. Alternatively, you can sign up as a new user and log in to ensure your code works.

Task 3 : Fix Logout View

As noted in the previous exercise, the logout link is still broken. Open the `Back-end-Development-Capstone/concert/views.py` file in the editor.

Open **views.py** in IDE

1. Go to the method named `logout_view`.

```
1
2
def logout_view(request):
    pass
```

Copied!

2. The logout view will simply log out the user using the `django.contrib.auth` module. Additionally, it will redirect the user to the login screen.
▼ Click here for hint.

```
1
2
3
def logout_view(request):
    {insert code to logout the user using the django.contrib.auth module}
    {insert code to return the user to the login page using the HttpResponseRedirect module}
```

Copied!

You can test the logout flow by logging in and then clicking on the `Logout` button.

Final Solution

1. Ensure your solution for the login_view method matches the following:
▼ Click here for the solution.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
def login_view(request):
    if request.method == "POST":
        username = request.POST.get("username")
        password = request.POST.get("password")
        try:
            user = User.objects.get(username=username)

            if user.check_password(password):
                login(request, user)
                return HttpResponseRedirect(reverse("index"))
        except User.DoesNotExist:
            return render(request, "login.html", {"form": LoginForm})
    return render(request, "login.html", {"form": LoginForm})

def logout_view(request):
    logout(request)
    return HttpResponseRedirect(reverse("login"))
```

Copied!

Exercise 7: Fix the Concert Page

The `Concert` page shows a logged in authenticated user and their concerts. The user can either attend a concert or choose not to attend. As before, you will first fix the URL and then fix the concert view. You need to sign in as a valid user to test the tasks below. You can log in with a user you created in the previous exercises or create a brand new user.

Your Tasks

Task 1 : Fix concert URL

Open the `Back-end-Development-Capstone/concert/urls.py` file in the editor.

Open **urls.py** in IDE

1. You should see all the URLs defined in this file:

```
1
2
3
4
5
6
7
8
9
10
```

```
urlpatterns = [
    re_path(r"^$", views.index, name="index"),
    path("songs", views.songs, name="songs"),
    path("photos", views.photos, name="photos"),
    path("login/", views.login_view, name="login"),
    path("logout/", views.logout_view, name="logout"),
    path("signup/", views.signup, name="signup"),
    path("", views.concerts, name="concerts"),
    ...
]
```

Copied!

2. Replace the first empty string for the concert urlpattern with `concert/`.

▼ Click here for a hint.

```
1
```

```
path("concert/", views.concerts, name="concerts"),
```

Copied!

3. If you click the `Concert` link now after logging in as a valid user, you will get an error. However, again, you have made progress and we will fix this error in the next Task.

Task 2 : Fix Concert View

The Concert link is working, but the view is still broken. Open the `Back-end-Development-Capstone/concert/views.py` file in the editor.

Open **views.py** in IDE

1. Go to the method named `concerts`.

```
1
2
```

```
def concerts(request):
    pass
```

Copied!

2. This view will first check if the user is authenticated and if so, show the list of concerts to the user by rendering the `concerts.html` template with the appropriate data. We have provided the following skeleton code to get you started:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
```

```
if {insert code to check if the user is authenticated}:
    lst_of_concert = {insert code to create an empty list}
    concert_objects = {insert code to get all Concerts using the Concert.objects object}
    {insert code to loop through all items in the concert_objects}:
        try:
            status = item.attendee.filter(
                user=request.user).first().attending
        except:
            status = "-"
        lst_of_concert.append({
            "concert": item,
            "status": status
        })
    return {insert code to render the `concerts.html` page with the data of {"concerts": lst_of_concert}}
else:
    return {insert code to redirect the user to the login page as the user is not authenticated}
```

Copied!

3. If your code works without errors, you should now see the list of concerts when you click the `concert` link after logging in as a valid user.

Djirection

Date	Concert Name	Duration
		hr

Notice that the concert details are not showing up. You will fix this in the next exercise.

Final Solution

1. Ensure your solution for the login_view method matches the following:
   ▼ Click here for the solution.

```
def concerts(request):
    if request.user.is_authenticated:
        lst_of_concert = []
        concert_objects = Concert.objects.all()
        for item in concert_objects:
            try:
                status = item.attendee.filter(
                    user=request.user).first().attending
            except:
                status = "-"
            lst_of_concert.append({
                "concert": item,
                "status": status
            })
        return render(request, "concerts.html", {"concerts": lst_of_concert})
    else:
        return HttpResponseRedirect(reverse("login"))
```

Copied!

Exercise 8 - Admin and Models

You need to create a super user before you can create concerts.

Task 1: Create a superuser

1. Open the terminal and change into the project directory

```
cd /home/project/Back-end-Development-Capstone
```

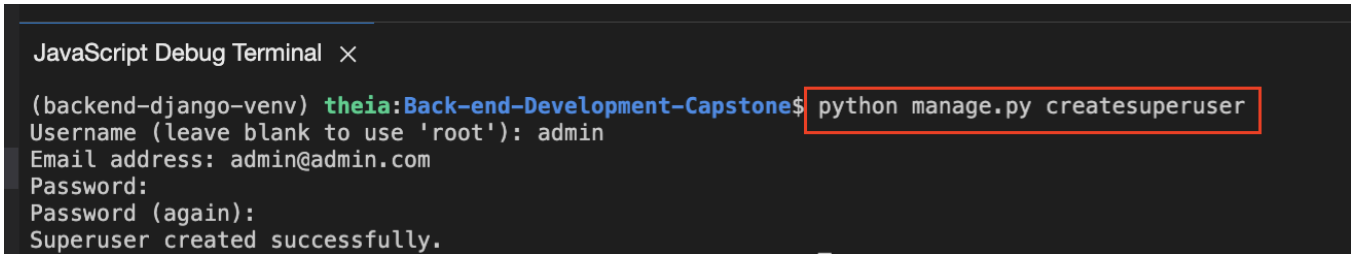
Copied! Executed!

2. Create a superuser. Remember the username and password. The lab uses a username of `admin`, an email of `admin@admin.com`, and a password of `admin`.

```
python manage.py createsuperuser
```

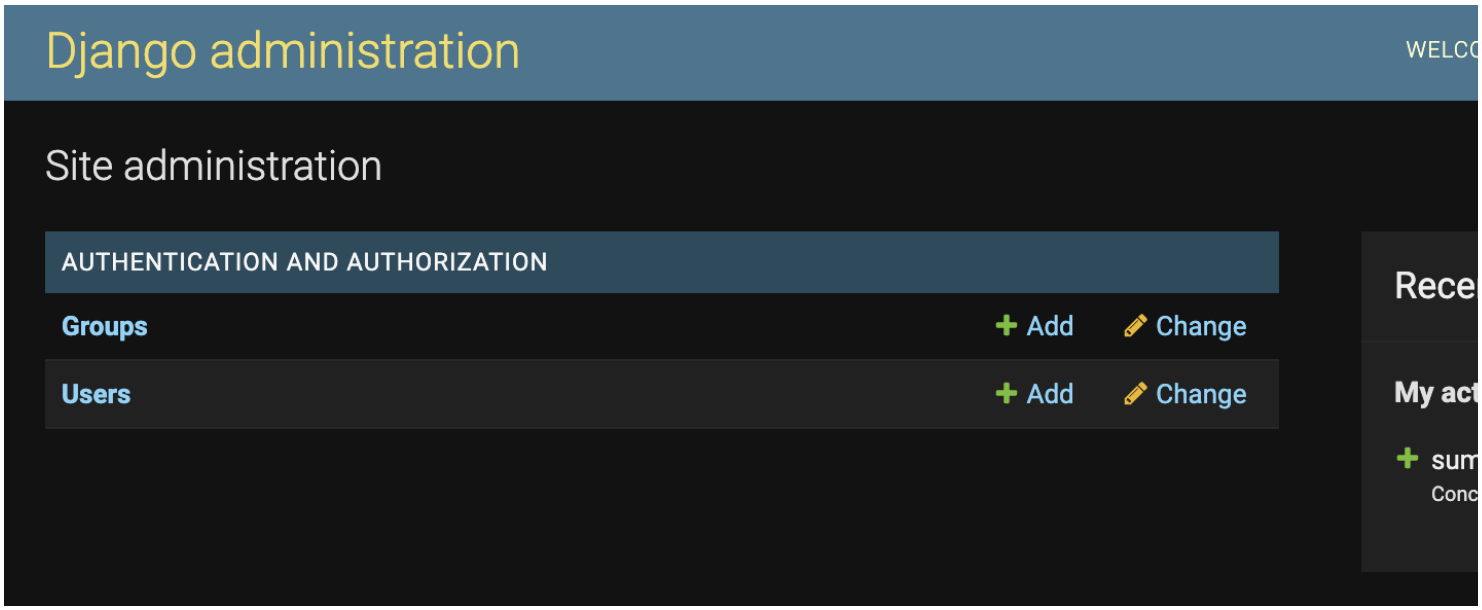
Copied! Executed!

The output should look as follows:



Task 2 : Register Concert model with the admin interface

If you log into the admin site `/admin` with the superuser username and password, you will see that the concert object does not appear there. In this exercise, register the `concert` model with the admin interface.



Open the `Back-end-Development-Capstone/concert/admin.py` file in the editor.

Open **admin.py** in IDE

1. Register the `Concert model` with the admin interface.

▼ Click here for a hint.

```
1. 1
1. admin.site.register(Concert)
```

Copied!

Exercise 9: Push code back to GitHub

Now that you have finished the code for the microservice, you can push the `backend-rest` branch back to your GitHub fork. Since you are the only one working on this project, go ahead and merge the PR and delete the branch. Make sure all your code changes are pushed back to the main branch before proceeding to the next lab.

1. Use the `git commit -am` command to commit your changes with the message "implemented django application", and the `git push` command to push those changes to your repository.

    Note: You will be prompted to set up your git user and email the first time you push:

```
1. 1
2. 2
1. git config --local user.name "{your GitHub name here}"
2. git config --local user.email {your GitHub email here}
```

Copied!

▼ Click here for a hint.

```
1. 1
2. 2
1. git commit -am "{message here}"
2. git push --set-upstream origin {branch name here}
```

Copied! Executed!

▼ Click here for a hint.

```
1. 1
2. 2
1. git commit -am "implemented django application"
2. git push --set-upstream origin backend-rest
```

Copied! Executed!

2. Create a pull request on GitHub to merge your changes into the main branch, and, since there is no one else on your team, accept the pull request, merge it, and delete the branch.

The main branch, at this point, should have your completed code.

Conclusion

Congratulations! You have finished the Django project. You fixed the URLs and the various views. You also implemented the models that are used in the views and the admin interface. Note that we are hardcoding the songs and photos service. We will connect the real microservices in the final lab of Capstone.

Next Steps

You have created all the microservices required for this course. You are well on your way to deploying all the services on the cloud.

Author(s)

CF

Changelog

Date	Version	Changed by	Change Description
2023-02-04	0.1	CF	Initial version created
2023-02-09	0.2	SH	QA pass with edits
2023-02-21	0.3	SH	QA pass after updates
2023-02-21	0.3	CF	Code fixes