KAZAKH BRITISH TECHNICAL UNIVERSITY

Midterm

Building a task management application using Django and Docker

Prepared by Azimkhanov Bauyrzhan

Almaty, 13.10.2024

Table of contents

Executive summary

The goal of this assignment is to gain hands-on experience with Django and Docker, focusing on Django, models and views. I used Python library version 3 called Django, Docker, Docker compose. From Django tools I used views, models, apps and routing via urls. As a result, I got an application with a task model, where you can perform CRUD operations both using the view and using the admin panel of Django itself.

Introduction

Nowadays, it is very important to use or switch to a microservice architecture. Since it allows you to better scale projects, reduces support costs, etc.

Another very common problem is that you often have to either reboot the application or move/run it in another environment. This can cause many problems, ranging from the need to store the order of application startup somewhere to the difference in the architecture of the hardware on which the application is launched. Most of them can be solved by a tool such as Docker and its derivative Docker compose. Docker allows you to prepare an environment and run it on any platform that supports it. That is, the developer and engineer do not need to constantly look back at these aspects.

My main motivation for making a task manager using Docker and Django was the desire to master a new stack of relevant technologies and the desire to get a good grade at the end of the semester.

Project objectives

Among the goals of this project, I can highlight the creation of a prototype of the application with the necessary requirements. All this is necessary in order to better understand this subject and master the material.

List of goals:

1. Understand the concepts of containerization and its benefits. Install Docker and set up a simple Docker container to run a basic application.
2. Create a Dockerfile to define the environment for the Django application. Specify the necessary dependencies, including Python and Django packages.
3. Use Docker Compose to define and run multi-container applications. Set up a docker-compose.yml file to manage both the Django app and the database service.
4. Configure Docker networking to allow communication between containers. Use Docker volumes to persist data and manage the database state.
5. Set up a Django project and create a basic application structure. Configure settings to work with the Docker environment.
6. Define Django models for the task management application. Implement migrations to create the necessary database tables.

Intro to containerization: Docker

Containerization is a software deployment process that bundles an application's code with all the files and libraries it needs to run on any infrastructure. Traditionally, to run any application on your computer, you had to install the version that matched your machine's operating system. For example, you needed to install the Windows version of a software package on a Windows machine. However, with containerization, you can create a single software package, or container, that runs on all types of devices and operating systems.

Developers use containerization to build and deploy modern applications because of the following advantages:

1. **Portability** - Software developers use containerization to deploy applications in multiple environments without rewriting the program code. They build an application once and deploy it on multiple operating systems. For example, they run the same containers on Linux and Windows operating systems. Developers also upgrade legacy application code to modern versions using containers for deployment.
2. **Scalability** - Containers are lightweight software components that run efficiently. For example, a virtual machine can launch a containerized application faster because it doesn't need to boot an operating system. Therefore, software developers can easily add multiple containers for different applications on a single machine. The container cluster uses computing resources from the same shared operating system, but one container doesn't interfere with the operation of other containers.
3. **Fault tolerance** - Software development teams use containers to build fault-tolerant applications. They use multiple containers to run microservices on the cloud. Because containerized microservices operate in isolated user spaces, a single faulty container doesn't affect the other containers. This increases the resilience and availability of the application.
4. **Agility** - Containerized applications run in isolated computing environments. Software developers can troubleshoot and change the application code without interfering with the operating system, hardware, or other application services. They can shorten software release cycles and work on updates quickly with the container model.

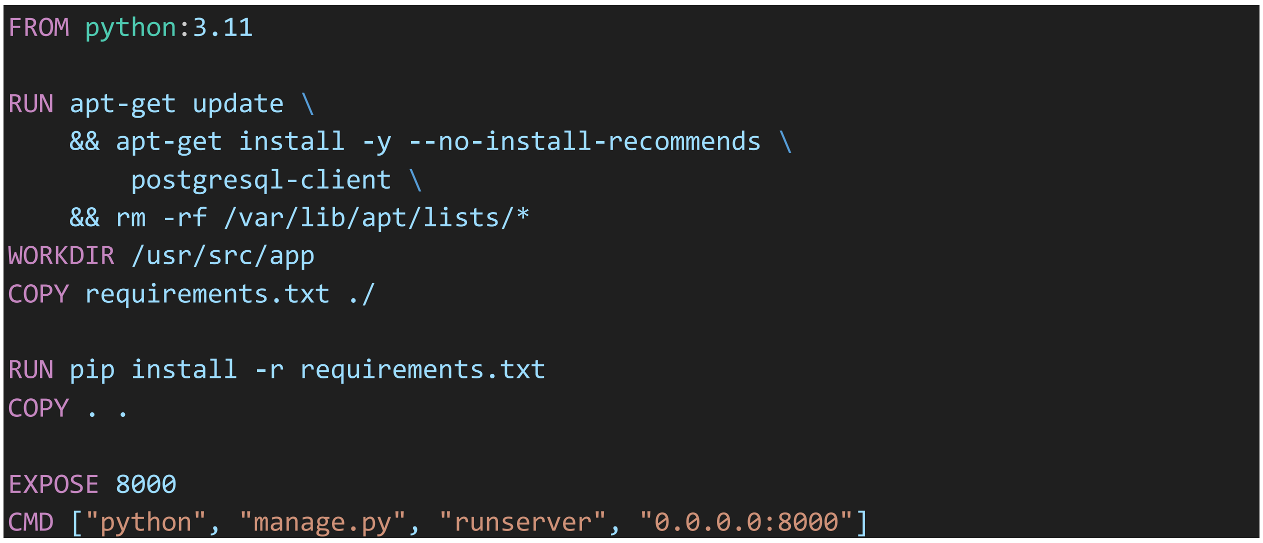
To install Docker, I used the official documentation. I attached a link to it in the refferences section. In short, to install, I connected the repository with the Docker code to my package manager (apt). Then I updated the list of packages and Docker and Docker compose appeared for installation. All that was left was to install the required packages, which I successfully did. All that was left was to perform the optional steps of adding the user to the docker group, so that in the future all commands could be executed without using superuser rights (sudo).

Creating a Dockerfile

Dockerfile components:

1. **FROM -** The FROM instruction initiates a new build phase and sets the base image for subsequent instructions. As such, a valid Dockerfile must begin with the FROM instruction.
   FROM <imge_name>:<version>
2. **RUN -** The RUN instruction allows you to install your application and the required packages for it. It executes any command on top of the current image and creates a new layer giving the result.
   RUN <command> (the command is run in a shell)
3. **CMD -** The CMD command defines a default command to run when your container starts.
   CMD <command>
4. **ENTRYPOINT -** An ENTRYPOINT is the same work as CMD. An ENTRYPOINT allows you to the extra argument of the command that will run as an executable.
   ENTRYPOINT <command> <param1> <param2>….
5. **COPY -** The COPY instruction copies files or directories from the Base os <src> and adds them to the filesystem of the container at the path <dest>
   COPY <src> <dest>

6. **WORKDIR -** The WORKDIR instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Dockerfile.
   WORKDIR </path/to/workdir>
7. And more components to be used.



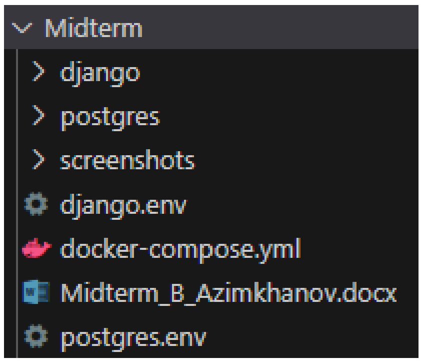
```
FROM python:3.11

RUN apt-get update \
    && apt-get install -y --no-install-recommends \
        postgresql-client \
    && rm -rf /var/lib/apt/lists/*
WORKDIR /usr/src/app
COPY requirements.txt ./

RUN pip install -r requirements.txt
COPY . .

EXPOSE 8000
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

Figure 1. Dockerfile of container with Django

Dockerfile explaination:

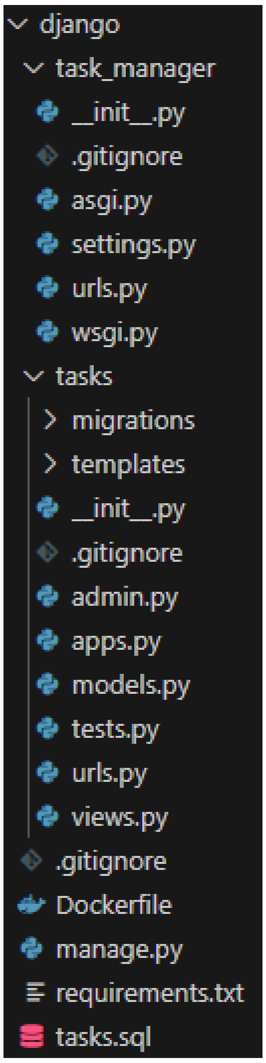
1. **FROM python:3.11** - This line imports the Python 3.11 image from the Docker registry.
2. **RUN apt-get update \**
   && apt-get install -y --no-install-recommends \
   postgresql-client \
   && rm -rf /var/lib/apt/lists/* - This command updates the package lists for the apt package manager, installs the PostgreSQL client without any additional recommended packages and removes the cached package lists to reduce the image size.
3. **WORKDIR /usr/src/app** - This sets the working directory inside the container to /usr/src/app.
4. **COPY requirements.txt ./** - This copies the requirements.txt file from the local machine to the current directory in the container.
5. **RUN pip install -r requirements.txt** - This installs the Python packages listed in requirements.txt.
6. **COPY . .** - This copies all files from the local directory to the current directory in the container.
7. **EXPOSE 8000 -** This informs Docker that the container will listen on port 8000 at runtime.
8. **CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]** - This specifies the command to run when the container starts, which is to start the Django development server on all network interfaces at port 8000.

Using Docker Compose



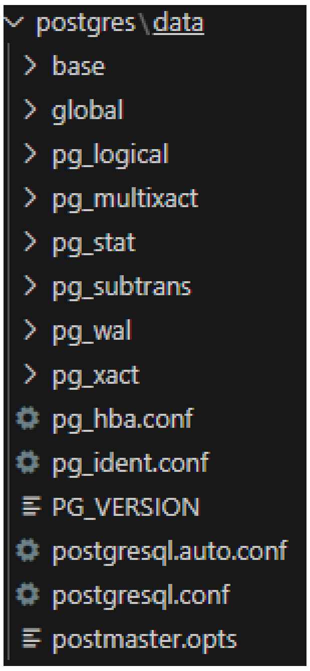
```
Midterm
  django
  postgres
  screenshots
  django.env
  docker-compose.yml
  Midterm_B_Azimkhanov.docx
  postgres.env
```

Figure 2. Project structure



```
django
  task_manager
    __init__.py
    .gitignore
    asgi.py
    settings.py
    urls.py
    wsgi.py
  tasks
    migrations
    templates
    __init__.py
    .gitignore
    admin.py
    apps.py
    models.py
    tests.py
    urls.py
    views.py
  .gitignore
  Dockerfile
  manage.py
  requirements.txt
  tasks.sql
```

Figure 3. Django structure



```
postgres\data
  base
  global
  pg_logical
  pg_multixact
  pg_stat
  pg_subtrans
  pg_wal
  pg_xact
  pg_hba.conf
  pg_ident.conf
  PG_VERSION
  postgresql.auto.conf
  postgresql.conf
  postmaster.opts
```

Figure 4. Postgres structure

This docker-compose.yml file defines a multi-container Docker application with two services: django and postgres. It also sets up a custom network named net.

```yaml
services:
  django:
    build: django/.
    ports:
      - 8000:8000
    env_file:
      - ./django.env
    command:
    depends_on:
      postgres:
        condition: service_healthy
        restart: true
    networks:
      - net
    volumes:
      - ./django:/usr/src/app

  postgres:
    image: postgres:16.4-bullseye
    ports:
      - 5432:5432
    env_file:
      - ./postgres.env
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U bauyrzhan -d midterm"]
      interval: 10s
      retries: 3
      start_period: 30s
      timeout: 10s
    networks:
      - net
    volumes:
      - ./postgres/data:/var/lib/postgresql/data

networks:
  net:
    driver: bridge
```
Figure 5. docker-compose.yml

Services

1. **django**
    1. **Build context**: The django service is built from the django/ directory.
    2. **Ports**: Maps port 8000 on the host to port 8000 in the container.
    3. **Environment variables**: Loaded from the ./django.env file.
    4. **Dependencies**: Depends on the postgres service being healthy before starting.
    5. **Restart Policy**: Always restart the container if it stops.
    6. **Networks**: Connected to the net network.
    7. **Volumes**: Mounts the ./django directory on the host to /usr/src/app in the container.

2. **postgres**
    1. **Image**: Uses the postgres:16.4-bullseye image.
    2. **Ports**: Maps port 5432 on the host to port 5432 in the container.
    3. **Environment variables**: Loaded from the ./postgres.env file.
    4. **Healthcheck**:
        1. **Test**: Runs the command “pg_isready -U bauyrzhan -d midterm” to check if the database is ready.
        2. **Interval**: Checks every 10 seconds.
        3. **Retries**: Retries up to 3 times before considering the service unhealthy.
        4. **Start Period**: Waits 30 seconds before starting health checks.
        5. **Timeout**: Each health check command must complete within 10 seconds.
    5. **Networks**: Connected to the net network.
    6. **Volumes**: Mounts the ./postgres/data directory on the host to /var/lib/postgresql/data in the container.

Networks

- **net**: A custom bridge network used to facilitate communication between the django and postgres services.

This setup ensures that the Django application can communicate with the PostgreSQL database, and the health of the PostgreSQL service is monitored before the Django service starts.
The entire database state is preserved by mounting volumes in the Postgres service. Similarly, mounting helps dynamically apply and reflect changes in the Django project code.

Docker networking and volumes

The docker-compose.yml file defines a custom network named net using the bridge driver. This network facilitates communication between the django and postgres services. Here are the benefits of this setup:

1. **Isolation**: The custom network isolates the services from other containers running on the host, enhancing security.
2. **Service discovery**: Docker automatically assigns each service a hostname based on the service name, making it easy for services to discover and communicate with each other.
3. **Simplified configuration**: By using a custom network, you avoid the need to manually configure IP addresses and ports for inter-service communication.

Volumes are used in the docker-compose.yml file to ensure data persistence for both the django and postgres services:

1. **Django service**:

    The volume ./django:/usr/src/app mounts the ./django directory on the host to /usr/src/app in the container. This allows the Django application code to be accessible and editable from the host machine.
2. **Postgres service**:

    The volume ./postgres/data:/var/lib/postgresql/data mounts the ./postgres/data directory on the host to /var/lib/postgresql/data in the container. This ensures that the PostgreSQL database data is stored persistently on the host, even if the container is stopped or removed.

Defining Django models

To create models in Django, I created a model in the models.py file. This model is a python class with fields representing columns in the table. In this case, there are 4 columns: title, description, created_at, completed. CharField, TextField, DateTimeField and BooleanField are the data types of the corresponding columns.

To create migrations, follow these steps:

1. **Generate migrations** - run the command *python manage.py makemigrations*. This command inspects the models and creates migration files that describe the changes to be made to the database schema.
2. **Apply migrations** - use the command *python manage.py migrate*. This command applies all pending migrations to the database, ensuring the schema is up-to-date with the current state of the models.
3. **Check migration status** - the command *python manage.py showmigrations* lists all migrations and their applied status.
4. **Rollback migrations** - If needed, migrations can be rolled back using *python manage.py migrate <app_name> <migration_name>* to revert to a specific migration.

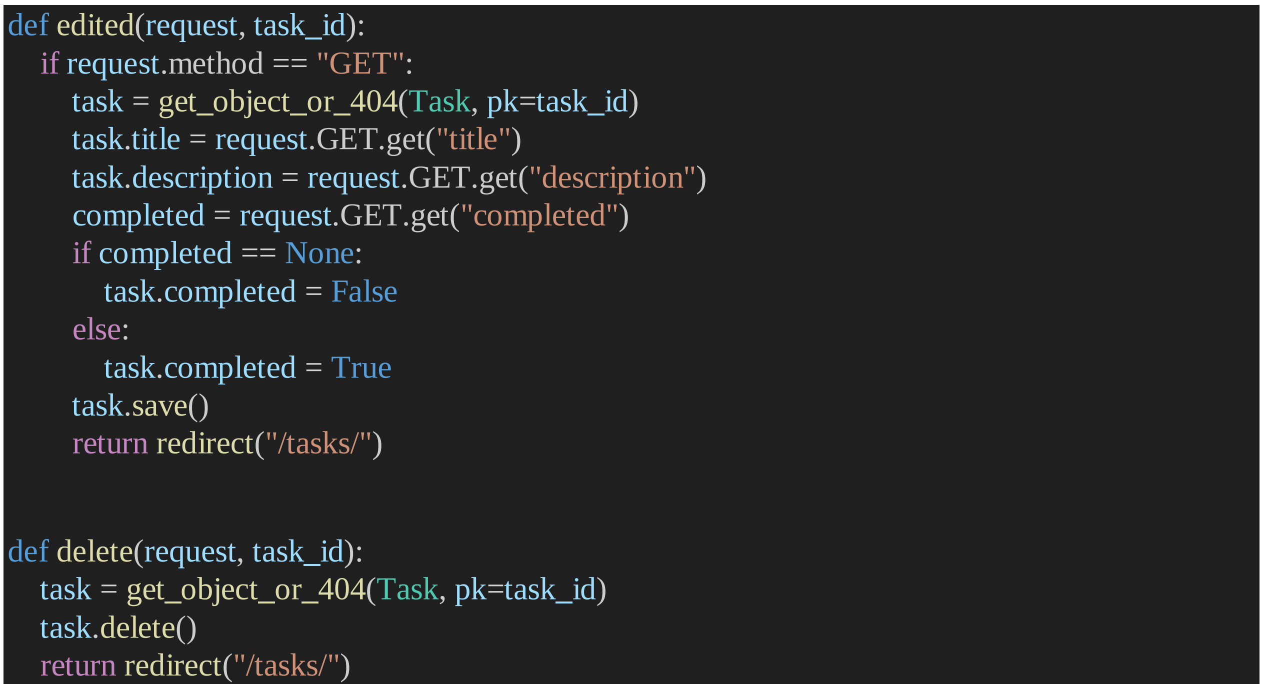
Creating views

To create view we should make corresponding changes in views.py file.

There are different views. Let us briefly describe them:

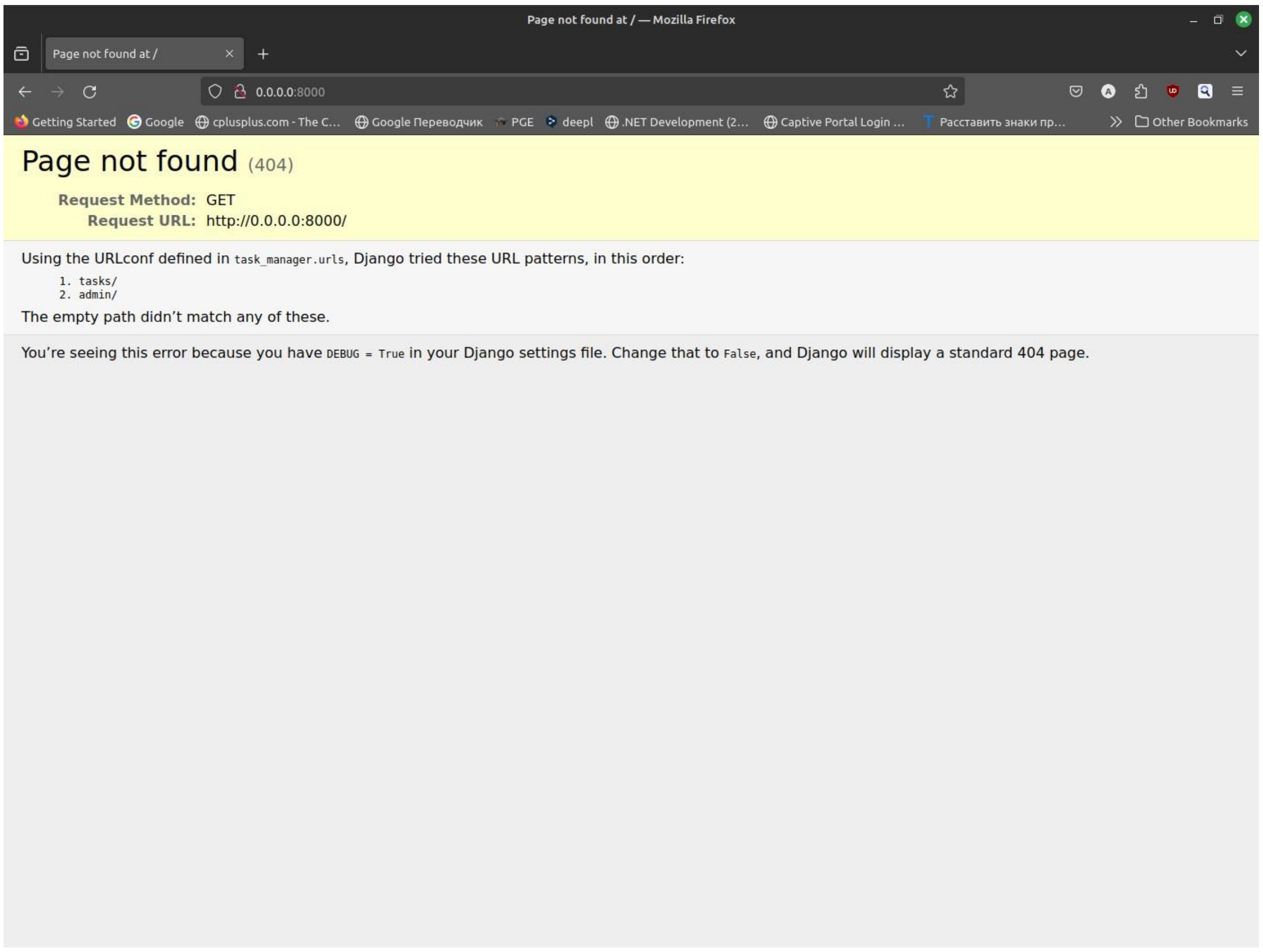
1. Index - Fetches the latest tasks and renders the index.html template with the task list.
2. Detail - Retrieves a specific task by task_id and renders the detail.html template with the task details.
3. Finish - Marks a specific task as completed and renders the finished.html template with the updated task.
4. Incomplete - Marks a specific task as incomplete and renders the incompleted.html template with the updated task.
5. Create - Renders the task_form.html template for creating a new task.
6. Created - Handles the creation of a new task from GET request data and redirects to the task list.
7. Edit - Retrieves a specific task by task_id and renders the task_form.html template for editing the task.
8. Edited - Handles the update of a specific task from GET request data and redirects to the task list.
9. Delete - Retrieves a specific task by task_id, deletes it from the database, and then redirects to the task list.

```
from django.shortcuts import render, get_object_or_404, redirect
from django.template import loader
from django.utils import timezone

from .models import Task

def index(request):
    latest_tasks_list = Task.objects.order_by("-created_at")
    template = loader.get_template("tasks/index.html")
    context = {
        "latest_tasks_list": latest_tasks_list,
    }
    return render(request, "tasks/index.html", context)

def detail(request, task_id):
    task = get_object_or_404(Task, pk=task_id)
    return render(request, "tasks/detail.html", {"task": task})

def finish(request, task_id):
    task = get_object_or_404(Task, pk=task_id)
    task.completed = True
    task.save()
    return render(request, "tasks/finished.html", {"task": task})

def incomplete(request, task_id):
    task = get_object_or_404(Task, pk=task_id)
    task.completed = False
    task.save()
    return render(request, "tasks/incompleted.html", {"task": task})

def create(request):
    return render(request, "tasks/task_form.html")

def created(request):
    if request.method == "GET":
        title = request.GET.get("title")
        description = request.GET.get("description")
        completed = request.GET.get("completed")
        if completed == None:
            completed = False
        else:
            completed = True
        task = Task(title = title, description = description, created_at = timezone.now(), completed
= completed)
        task.save()
        return redirect("/tasks/")

def edit(request, task_id):
    task = get_object_or_404(Task, pk=task_id)
    return render(request, "tasks/task_form.html", {"task": task})
```



```
def edited(request, task_id):
    if request.method == "GET":
        task = get_object_or_404(Task, pk=task_id)
        task.title = request.GET.get("title")
        task.description = request.GET.get("description")
        completed = request.GET.get("completed")
        if completed == None:
            task.completed = False
        else:
            task.completed = True
        task.save()
        return redirect("/tasks/")


def delete(request, task_id):
    task = get_object_or_404(Task, pk=task_id)
    task.delete()
    return redirect("/tasks/")
```

Figure 6. tasks/views.py



Page not found (404)

Request Method: GET
Request URL: http://0.0.0.0:8000/

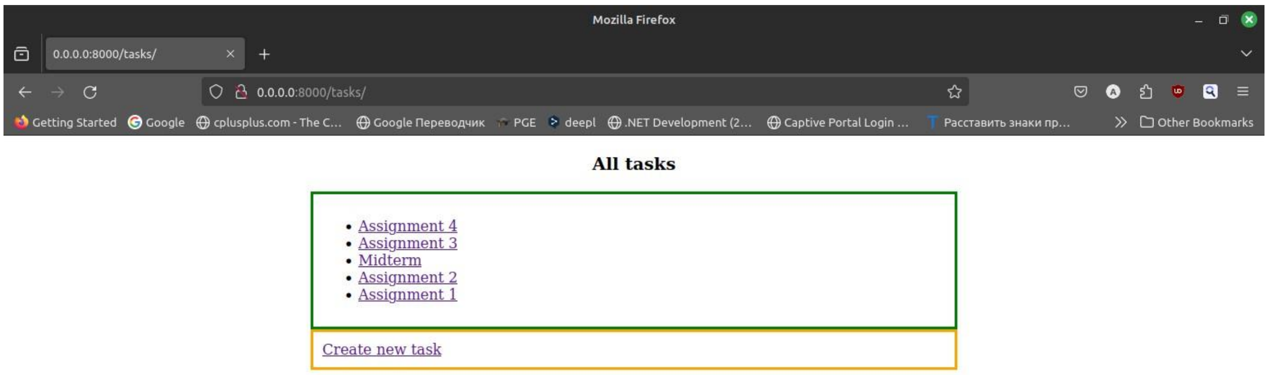
Using the URLconf defined in `task_manager.urls`, Django tried these URL patterns, in this order:

```
1. tasks/
2. admin/
```

The empty path didn’t match any of these.

You’re seeing this error because you have `DEBUG = True` in your Django settings file. Change that to `False`, and Django will display a standard 404 page.

Figure 7. Main page



Mozilla Firefox

0.0.0.0:8000/tasks/

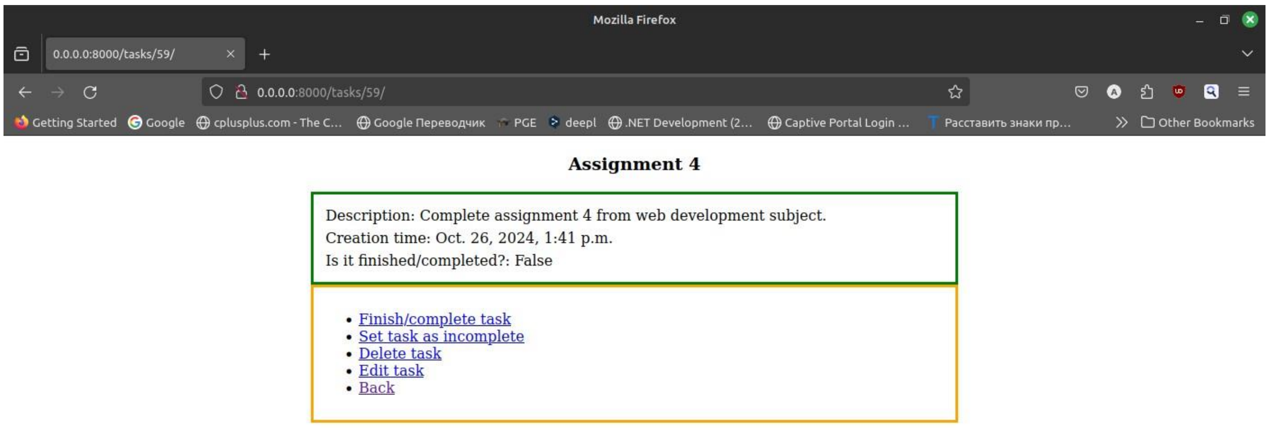
Getting Started Google cplusplus.com - The C... Google Переводчик PGE deepl .NET Development (2... Captive Portal Login ... Расставить знаки пр... Other Bookmarks

All tasks

- Assignment 4
- Assignment 3
- Midterm
- Assignment 2
- Assignment 1

Create new task

Figure 8. Tasks page

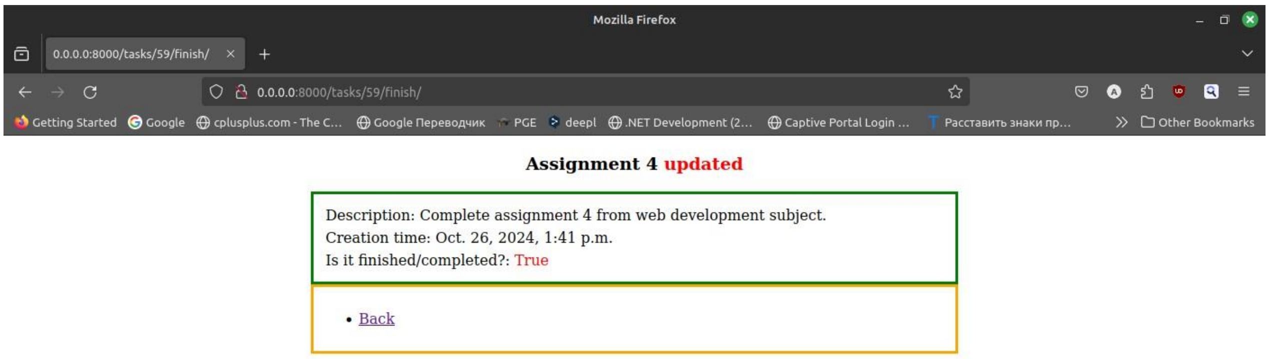


Assignment 4

Description: Complete assignment 4 from web development subject.
Creation time: Oct. 26, 2024, 1:41 p.m.
Is it finished/completed?: False

- Finish/complete task
- Set task as incomplete
- Delete task
- Edit task
- Back

Figure 9. Task detail page


Mozilla Firefox

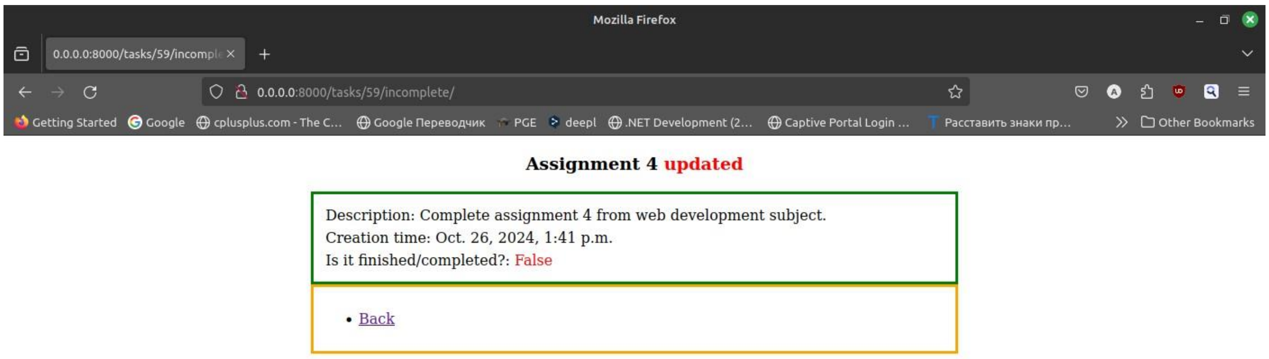
0.0.0.0:8000/tasks/59/finish/

Assignment 4 updated

Description: Complete assignment 4 from web development subject.
Creation time: Oct. 26, 2024, 1:41 p.m.
Is it finished/completed?: True

- Back

Figure 10. Pressed “Finish/complete task” button

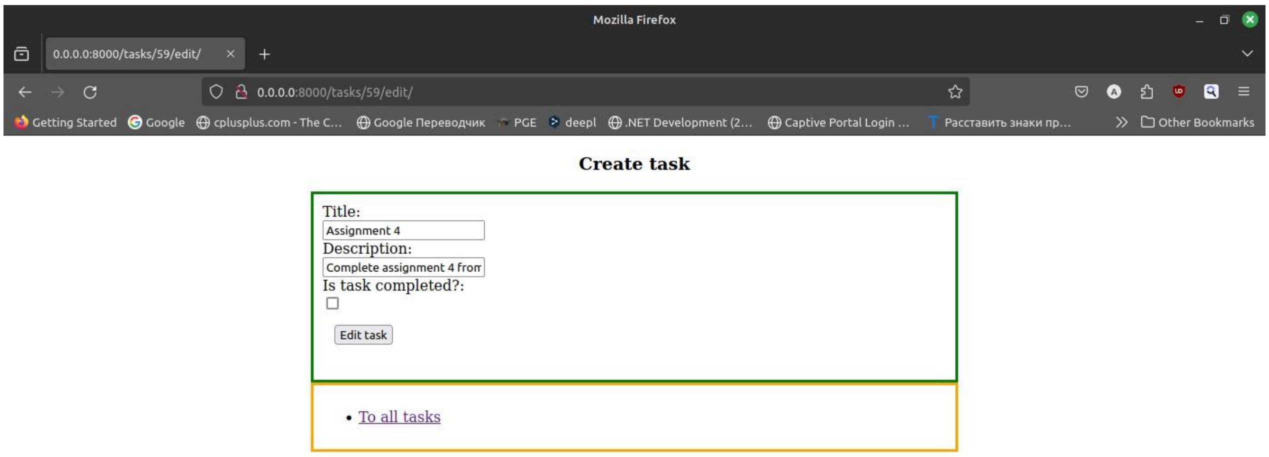


Assignment 4 updated

Description: Complete assignment 4 from web development subject.
Creation time: Oct. 26, 2024, 1:41 p.m.
Is it finished/completed?: False

- Back

Figure 11. Pressed "Set task as incomplete"



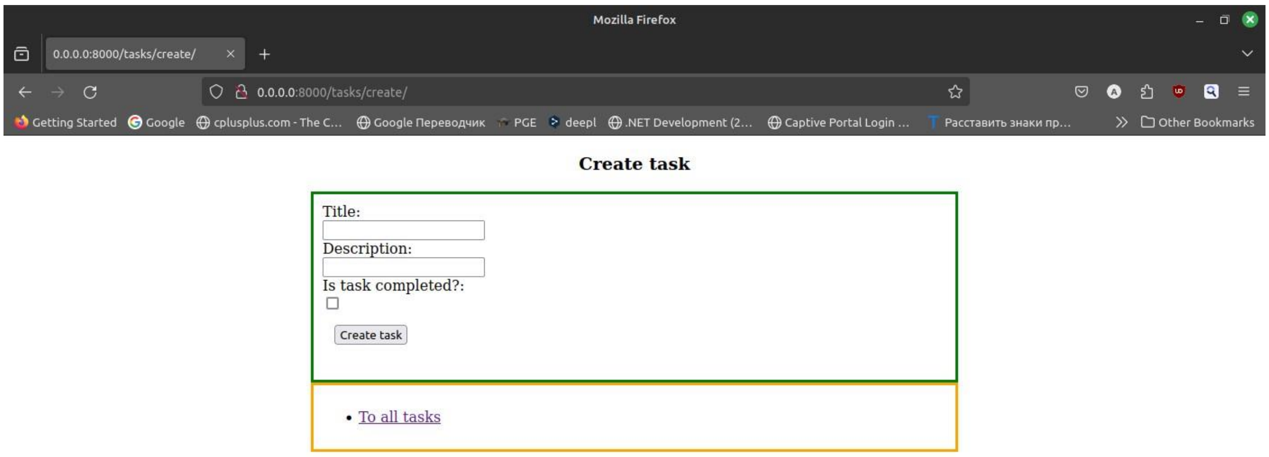
Create task

Title:
Assignment 4
Description:
Complete assignment 4 from
Is task completed?:
☐

Edit task

- To all tasks

Figure 12. Edit task



Create task

Title:

Description:

Is task completed?:

Create task

- To all tasks

Figure 13. Create task

Conclusion

As a result, we received an application that fully complies with the requirements of this work, namely "ask management application using Django and deploying it in a Docker container. The application will allow users to create, view, update, and delete tasks, and will utilize Docker to ensure a consistent development and production environment. The project will cover the fundamentals of containerization, Docker configuration, and Django model creation."

By using Docker as a containerization tool, this project has gained all the benefits of applications using this approach:

1. Portability
2. Efficiency
3. Agility
4. Faster delivery
5. Improved security
6. Faster app startup
7. Easier management
8. Flexibility

References

Links to Django tutorials from official documentation:

1. https://docs.djangoproject.com/en/5.1/intro/tutorial01/
2. https://docs.djangoproject.com/en/5.1/intro/tutorial02/

Link to official Docker documentation:

- https://docs.docker.com/

Link to official Docker image registry where from I took base images for compose:

- https://hub.docker.com/

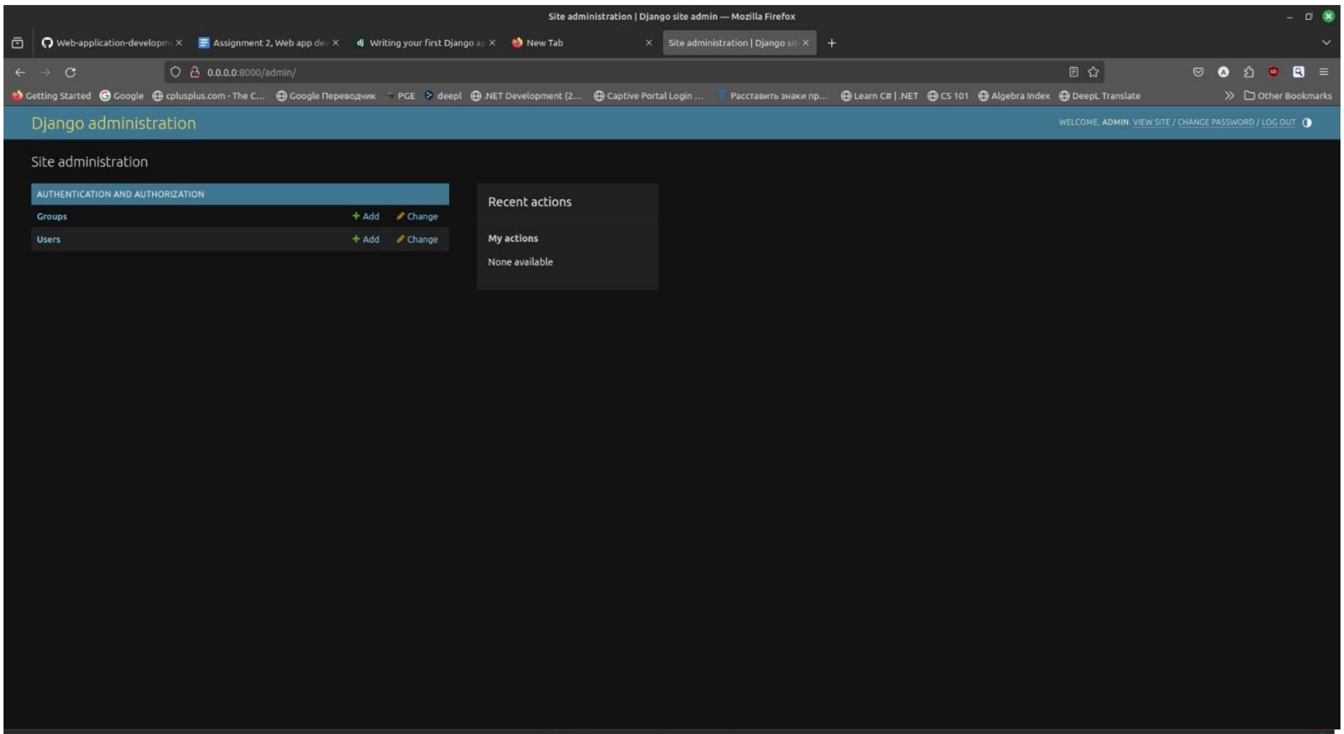
Links to some topics in StackOverflow:

1. https://stackoverflow.com/questions/13164048/text-box-input-height
2. https://stackoverflow.com/questions/3681627/how-to-update-fields-in-a-model-without-creating-a-new-record-in-django
3. https://stackoverflow.com/questions/42614172/how-to-redirect-from-a-view-to-another-view-in-django
4. https://stackoverflow.com/questions/3805958/how-to-delete-a-record-in-django-models
5. https://stackoverflow.com/questions/14719883/django-can-not-get-a-time-function-timezone-datetime-to-work-properly-gett
6. https://stackoverflow.com/questions/3289601/null-object-in-python
7. https://stackoverflow.com/questions/11336548/how-to-get-post-request-values-in-django
8. https://stackoverflow.com/questions/11714721/how-to-set-the-space-between-lines-in-a-div-without-setting-line-height
9. https://stackoverflow.com/questions/3430432/django-update-table
10. https://stackoverflow.com/questions/15128705/how-to-insert-a-row-of-data-to-a-table-using-djangos-orm
11. https://stackoverflow.com/questions/67155473/how-can-i-get-values-from-a-view-in-django-every-time-i-access-the-home-page
12. https://stackoverflow.com/questions/3500859/django-request-get
13. https://stackoverflow.com/questions/50346326/programmingerror-relation-django-session-does-not-exist
14. https://stackoverflow.com/questions/18713086/virtualenv-wont-activate-on-windows

Link to my GitHub repository (screenshots, logs and SQL structure included):

- https://github.com/BauyrzhanAzimkhanov/Web-application-development-MSc-

Screenshots



Site administration | Django site admin — Mozilla Firefox

0.0.0.0:8000/admin/

Django administration

WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT

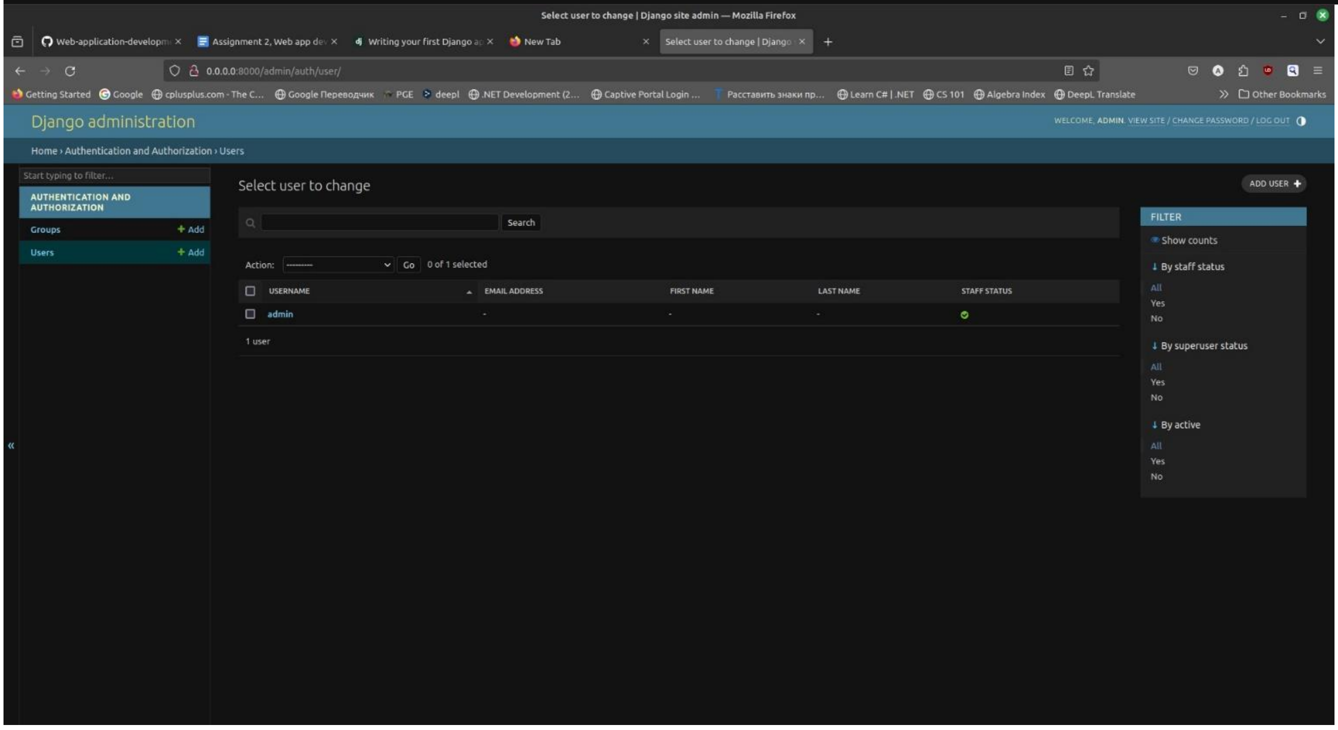
Site administration

AUTHENTICATION AND AUTHORIZATION		
Groups	+ Add	Change
Users	+ Add	Change

Recent actions

My actions

None available



Select user to change | Django site admin — Mozilla Firefox

0.0.0.0:8000/admin/auth/user/

Django administration

WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Authentication and Authorization › Users

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

Select user to change

ADD USER +

Search

Action: --------- Go 0 of 1 selected

	USERNAME	EMAIL ADDRESS	FIRST NAME	LAST NAME	STAFF STATUS
☐	admin	-	-	-	✓

1 user

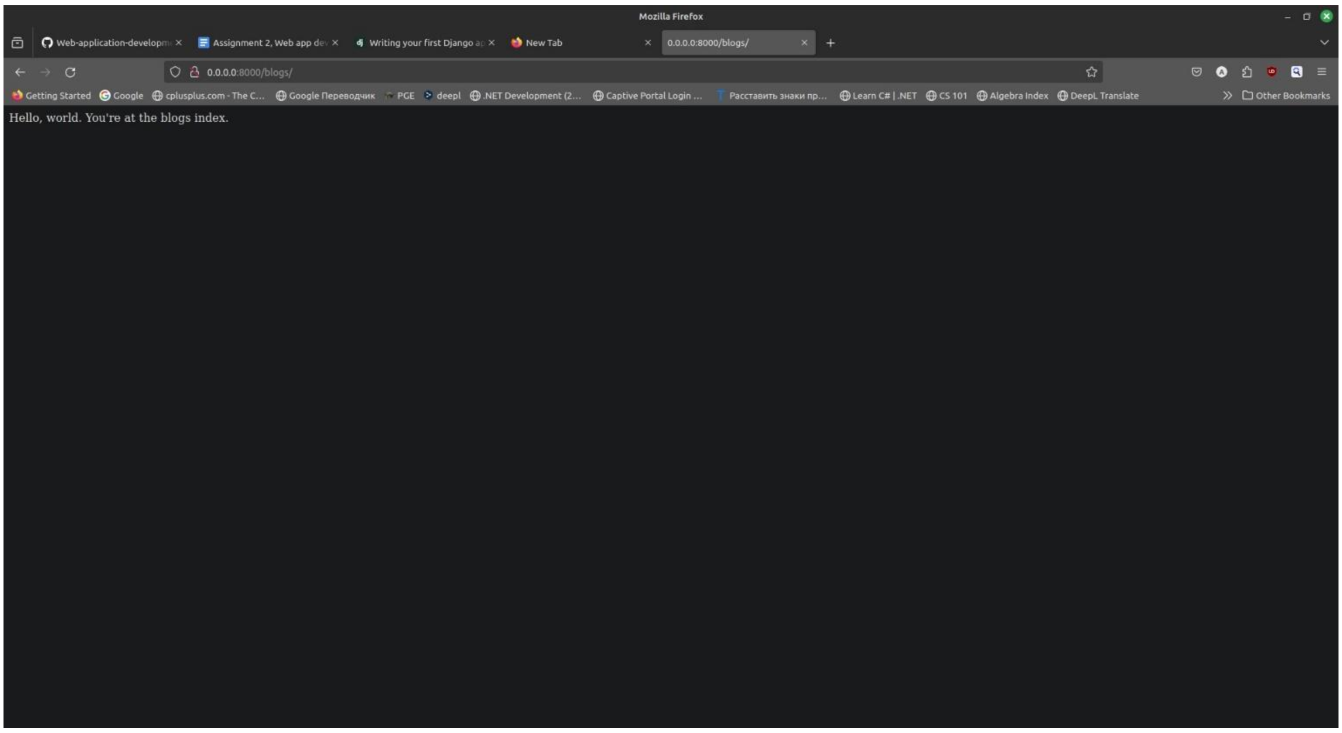
FILTER

Show counts

↓ By staff status
All
Yes
No

↓ By superuser status
All
Yes
No

↓ By active
All
Yes
No



```
Hello, world. You're at the blogs index.
```