Deliverable 1

Notes and report

Bustos Gracia, Marc - 49753478Q
Badaoui Mahdad, Faysal - 53923695B
Bernárdez Matalobos, Andrés - 53197271L
Batlle Gispert, Miquel - 48250329D

Universitat de Lleida
Web Project

Curs 2022 - 2023

Contents

1 GitHub repository

`https://github.com/FaysalBM/Web-Project`

2 Model implementation

The main idea of our model, was presented on the first delivery made.

The model, consists of a UML representing five databases with the corresponding server. These databases are User, Project, Tasks, Department and Companies. All of them are necessary to implement the web app consisting on a platform where users would be able to join companies departments or invited to join, In case that's their goal. If not, they can use the web privately. Inside a company's department or profile, they can create a project. This one is formed by users and tasks, which all of them can work If wanted inside the same project. So, to implement the availability to do this, we created all the tables "models" inside the models.py file. The result Is the following:

```
    class Task(models.Model):
    name = models.CharField(max_length=256)
    def __unicode__(self):
        return self.name

class Project(models.Model):
    name = models.CharField(max_length=256)
    tasks = models.ManyToManyField(Task)
    def __unicode__(self):
        return self.name


class Department(models.Model):
    name = models.CharField(max_length=256)
    projects = models.ManyToManyField(Project)
    num_projects = models.IntegerField(default=0)
    users = models.ManyToManyField(User)
    num_users = models.IntegerField(default=0)
    def __unicode__(self):
        return self.name

class Company(models.Model):
    name = models.CharField(max_length=256)
    email_com = models.EmailField(max_length=255, unique=True, default='defaultEmail')
    num_workers = models.IntegerField(default=0)
    departments = models.ManyToManyField(Department)
    workers = models.ManyToManyField(User)
    def __unicode__(self):
        return self.name

class User(AbstractBaseUser):
    id_user = models.CharField(max_length=255, primary_key=True)
    email_user = models.EmailField(max_length=255, unique=True, default='defaultUserMa
    name = models.CharField(max_length=255)
    profile_photo = models.BinaryField(null=True, blank=True)
    projects = models.ManyToManyField(Project, related_name='projects', blank=True)
    groups_number = models.IntegerField()
    groups = models.ManyToManyField(Department, related_name='departments', blank=True
    company = models.ManyToOneRel(Company, field_name='company', to='Company')
    is_active = models.BooleanField(default=True)
    is_admin = models.BooleanField(default=False)
    def __unicode__(self):
        return self.name
```

As Is visible on the code, each model has the fields on the UML design and, with the use of ManyTomanyField, OneToOneField, ... We are able to represent all the relations in the diagram UML between models.

3 Django admin interface to add, modify and delete instances

To turn on the admin interface, it was needed to work on the urls.py file.
In this file, we had to import admin from django.contrib and insert a new path inside the urlpatterns.

The new path was *path('admin/', admin.site.urls)*, this was, once the web app is launched on the web browser, introducing the field '/admin' on the link, we'll have the admin login page and page loaded. Finally, to login as an admin, we needed to create a superuser. Which was created with the command on terminal *python manage.py createsuperuser*.

In our case, the login parameters:
username : admin
password: admin

4 Django built-in authentication and user management

To install and add the django built-in authentication and user management, we introduced on the settings.py file, inside installed apps: 'django.contrib.auth' and django.contrib.contenttypes.

Following that, we added the necessary databases with the comand python manage.py migrate, and after this, on the file urls.py, we imported the django.contrib.auth to create the necessary urls.
In this case, we only added the login site, but there are a lot more possibilities. The path added consisted of *path('login/', LoginView.as view(), name='login')*, which gets the login.html file from the folder 'registration' and uses the html to create the site for the login feature. To get the login page, is necessary put the tag '/login' on the web link, or click the login button on the nav bar.

5 Environmental variables

To add the environmental variables, we had to create the database, to get the url, because the SECRET KEY was already on settings.py file. So, once the postgres database was created we added the link on settings.py file with the name of DATABASE URL.

6 Implementation of the deployment schema

6.1 Fly.io

The fly.toml file is a configuration file used by the Fly.io platform to deploy web applications.

The "app" variable indicates the name of the application being deployed.
"kill signal" is the signal that will be sent to the application to stop it if necessary, and "kill timeout" indicates how long Fly.io should wait before forcing the application to stop.
services refers to the services that run in the application. In this case, there is a single service running on internal port 8000 and the protocol is TCP.
services.concurrency sets concurrency limits for the service. In this case, the hard limit is 10 connections and the soft limit is 5 connections.
services.ports indicates how ports are handled in the service. In this case, ports are being handled by the HTTP protocol.
In summary, the fly.toml file is important because it tells Fly.io how to configure and deploy the application on their platform. The configuration in this file ensures that the application runs correctly and that the necessary resources are managed appropriately to keep it running smoothly.

6.2 Docker compose

The docker compose.yml file is a configuration file used by Docker Compose, which is a tool for defining and running multi-container Docker Applications.

The "version" specifies the version of the Docker Compose file syntax being used (in this case, version 3).
Under the "services" section, there are three services defined: "web", "db", and "nginx".
The "web" service builds an image from the current directory (".") and runs the command "python manage.py runserver 0.0.0.0:8000" to start a development server. It also maps the

container's port 8000 to the host's port 8000, depends
on the "db" service, and is connected to the "webnet" network.
The "db" service uses the "postgres" image, mounts a volume to persist the database data, and is also connected to the "webnet" network.
The "nginx" service uses the "nginx" image, maps the container's port 80 to the host's port 80, mounts a configuration file, depends on the "web" service, and is connected to the "webnet" network. The "networks" section defines the "webnet" network that the services are connected to.
The "volumes" section defines the "dbdata" volume that the "db" service uses to persist database data.

In summary, the docker-compose.yml file defines a multi-container application consisting of a Django development server ("web" service), a PostgreSQL database ("db" service), and an Nginx web server ("nginx" service) that serves as a reverse proxy for the development server. The file also specifies how the containers are connected and how data is persisted.

7 How to run the web app

To run the web app:
1- Go into the folder where manage.py is located.
2- Check that django is downloaded(can be using poetry).
3- Run on terminal python manage.py runserver