Key requirements

1. More than 50,000 employees. 5 levels of hierarchy
2. Tree-view of employees (from top to bottom levels)
3. Every employee’s info should be stored in a database and should consist of: Full name; Position; Hiring date; Salary;

Additional tasks

1. Create a database using database migrations
2. Seeder-algorithm which should fill out the database
3. Use bootstrap for page styling
4. Personal page for each employee
5. Search with sorting
6. Search without updating a page
7. Access restrictions
8. CRUD client-side operations
9. Employee photo upload
10. Subordinates should be rearranged to different bosses in case their boss is deleted
11. Tree-view should be procedurally generated
12. Boss rearrangement with drag-n-drop in tree view
13. Rest-framework interface implementation

Table of contents

ORM implementation

Model creation

We'll need the following models for our tasks:
- Position
- Department
- Employee

More detailed view of fields:

```
class Position(models.Model):
   name = models.CharField(max_length=100)
   salary = models.IntegerField(default=0)
   boss_position = models.ForeignKey(...)
   expected_workers = models.IntegerField(default=1)

class Department(models.Model):
   name = models.CharField(max_length=100)

class Employee(models.Model):
   first_name = models.CharField(max_length=20)
   last_name = models.CharField(max_length=20)
   photo = models.ImageField(default="nophoto.png", blank=True)
   position = models.ForeignKey(Position, on_delete=models.CASCADE)
   department = models.ForeignKey(...)
   hiring_date = models.DateField(default=datetime.date.today)
   salary = models.IntegerField(default=0)
   boss = models.ForeignKey(...)
```

Actually, `salary` and `expected_workers` fields in `Position` are only helping fields and will be used only during the seeding process.

Also, I should mention that, despite that `department` and `boss` fields can be nullified, `position` field is necessary and employee will be deleted incase it gets deleted.

Database creation

Django is really helpful during this step. We only need to make sure our models work and use the following commands:

```
python manage.py makemigrations
python manage.py migrate
```

Database seeding

There are different methods to seed the database, including `lazy` ones. However, since I intended to implement employee hierarchy via object relations (every employee has their boss as one of their fields), I have chosen migration seeding method.

For a start, let's create an empty migration with:

```
python manage.py makemigrations employees --empty
```

After that, we tell our migration which function we want to call:

```
def seeder(apps, schema_editor): ...
operations = [migrations.RunPython(seeder)]
```

`seeder` will create fictional positions, departments and employees, (see employees/migrations/0002_employee_seeder.py). All with proper hierarchy in mind. Let's also not forget about 5 levels task, starting from a managing director and finishing with a junior programmer.

Working with objects

To better understand how objects work, I recommend you look into employees/views.py and employees/models.py. I left a bunch of comments explaining which action does what.

However, I still should shortly mention that the main tool in working with objects representing a database are QuerySet objects. They provide a really handy and effective interface in working with database queries.

Example (we search for an employee, whose position is a boss position of another employees position, but also check that they aren't in different departments):

```
result = Employee.objects.filter(
    models.Q(position=position.boss_position),
    models.Q(department__isnull=True) |
    models.Q(department=dept)
)
```

Explanation: the logic is that an employee can be subordinate to another employee only if they:

1. Have a position which is a `boss` position to first employee's position
2. Are in the same department as a first employee or atleast one of them is not in any department.

That logic is not forced on the object scale so you can still end up with an improper boss. However, I try to avoid such situations.

Front-end implementation

Basic styling using bootstrap

Including bootstrap styling is actually incredibly easy. Go to the official bootstrap site and look for a similar code:

```
<link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
```

You can add this code to your page in any appropriate way. Now you can use bootstrap functionality on your page.

In this way I also included in this project jQuery, jQuery-UI, Popper and Bootstrap JS.

For Django forms i also used django-crispy-forms:

```
pip install django-crispy-forms
```

Navbar

I used bootstrap navbar, however with a small addition:

```
title_id = $("head title").attr('id')
$("li.nav-item > a").each(function (index) {
    nav_id = $(this).attr("id")
    if (title_id == nav_id) {
        $(this).addClass("active")
    }});
```

I look for a title element id tag (if any) and using it I choose currently active navbar element.

Tree-view of employees

I used something akin to recursion for tree-view implementation. Firstly, we load every employee without a boss. Every employee card also has a list of their subordinates (if they have any) in a collapsible list.

Every employee card in that list also has the same functionality, so it works in a similar way to recursion.

However, in reality, none of those lists are actually loaded before the user has opened them. More on that later.

Personal pages of employees

Almost any operation with an employee object is tied to its database id. Personal pages are no exception. Every personal page link contains employee's id within itself so we'll use that to our advantage:

```
path('employees/<int:employee_id>/', views.employee)
path('employees/<int:employee_id>/delete/', views.delete_employee)
path('employees/<int:employee_id>/edit/', views.edit_employee)
```

We should include those three routes in `urlpatterns`.

Mentioned views will handle `employee_id` as its named argument. The only thing left is to find employee's object by id and perform required actions.

Search

To find an employee we should filter employee objects by every word from a search query. If all of those are contained within `position`, `department`, `last_name` or `first_name` fields, we have our match.

Next, we sort a list of matched employees by a chosen field.

We also should limit the results of a search by the limits of a page we are currently on.

Employee creation, updating and deletion

Every employee page contains update and deletion links of that employee within itself. It was mentioned early, how exactly that routing works.

We'll put a new employee creation page in a navbar.

Those pages work using Django forms. Forms can store data from objects and convert them for use on the client-side. User gets a form which can contain field already filled in or vice versa, user can send data to your object using a form.

An example of using a form to edit employee info:

```
form = EmployeeForm(
    request.POST or None, request.FILES or None, instance=employee)
if form.is_valid():
    form.save()
```

Forms enhance the workflow while you are working with objects and often are the most important connecting link between front-end and back-end parts of object implementation.

Implementing ajax functionality

To implement any server request functionality without the need to reload the page we'll need ajax methods. Using them, we could send server requests without updating a page and also without breaking the javascript loop on a said page.

I used jQuery ajax methods. They are handier to use than vanilla javascript and take up less space in the code.

Tree-view works in the following way: if the query does not contain an \`id\` argument, render the initial page. Otherwise render the list of supposed subordinates of an employee whose id was sent. While pressing the \`show subordinates\` along with other logic an ajax query will be sent to a server with a said \`id\` argument.

I try to imitate the recursive logic in this way, when each employee card contains a functionality to get other employee cards. All while we can use the same view (and the same address) to render an initial page.

During the search query ajax works almost in the same way. If a user loads a page, render the page. If, on the other hand, an ajax query is sent, we render only the search results.

Registration and login

Registration and login logic are implemented in my \`users\` app. I used standard Django views to implement login and logout pages:

```
from django.contrib.auth import views as auth_views
urlpatterns = [
    ...,
    path('login/', auth_views.LoginView.as_view(
        template_name="users/login.html"), name="login"),
    path('logout/', auth_views.LogoutView.as_view(
        template_name="users/logout.html"), name="logout"),]
```

Registration form is handled by the `views.register`.

Drag-n-drop

jQuery-UI has really useful and handy methods `draggable` and `droppable`, which take the most of the graphical work upon themselves.

Using a `hoverClass: "on-draggable-hover"` argument we tell that while a draggable objects is hovering over a droppable one, the droppable one will have a `"on-draggable-hover"` class. Don't forget to change the styling of that class for visual user representation:

```
<style>.on-draggable-hover {background-color: #cdcdcd;}</style>
```

An argument `drop` should contain a function with an ajax query which will request a necessary operation from server. That being an attempt to make a `dropped on` employee a new boss for a `dragged` employee. After that we tell the user the result of a said operation.

Additional back-end logic

Access restriction

For starts, let's import a useful Django decorator:

```
from django.contrib.admin.views.decorators import staff_member_required
```

However said decorator is redirecting a user to an admin login page upon denied access event. We do not want that and need it to send user to our login address. This is solved by passing our login address as a decorator argument:

```
@staff_member_required(login_url='login')
```

I also used my own decorator called `user_access_error` to tell a user about access violation.

Working with files

I used Pillow and models.ImageField to implement an image handling field in an employee model:

```
photo = models.ImageField(default="nophoto.png", blank=True)
```

We also should set a directory to which said photos will be uploaded:

```
MEDIA_URL = '/media/'
```

Image thumbnail is actually the same image, just with a lower `width` parameter.

Back-end pagination generator

Pagination is actually interactively generated on a server upon any search request. Depending on a current search query, current page and page size it will determine which pages should be shown, which should be hidden and what page is currently active.

Look for more details at ‘employees/utilities.py’.

Employee reassignment upon their boss deletion

Whenever an employee is deleted the following code will fire:

```
from django.dispatch import receiver
@receiver(pre_delete, sender=Employee)
def handlePreDelete(sender, **kwargs):
    boss = kwargs['instance']
    subordinates = boss.employee_set.all()

    for employee in subordinates:
        try:
            appropriate = get_appropriate_bosses(
                    employee.position, employee.department, boss)
            employee.boss = choice(appropriate)
        except IndexError:
            employee.boss = None
        employee.save()
```

In case of employee deletion for every subordinate they had we will try to find all the other appropriate bosses and then chose a random one of those if any. Otherwise just set the `boss` field to None.

Using Rest Framework

Let's install djangorestframework:

```
pip install djangorestframework
```

We'll need serializers for each model (employees/api/serializers.py), views for each model (employees/api/views.py) and then the routers:

```
from rest_framework import routers
rest_router = routers.DefaultRouter()
rest_router.register('_employees', views.EmployeeRestView)
rest_router.register('_positions', views.PositionRestView)
rest_router.register('_departments', views.DepartmentRestView)
```

Rest pagination is also going to be needed (50 thousand employees is a lot):

```
from rest_framework.pagination import PageNumberPagination
class EmployeePagination(PageNumberPagination):
   page_size = 50
```

Also, let's make the employee view private:

```
permission_classes = (IsAdminUser,)
```