Основные требования

1. Количество сотрудников более 50,000. 5 уровней иерархий
2. Страница сотрудников в древовидной форме (от начальника к подчиненным)
3. Информация о каждом сотруднике должна храниться в базе данных и содержать следующие данные: ФИО; Должность; Дата приема на работу; Размер заработной платы;

Дополнительные задания

1. Создание базы данных с применением миграции данных
2. Реализация seeder-алгоритма, который заполнит базу данных
3. Использование bootstrap для стилизации страниц
4. Личная страница сотрудника
5. Поиск с сортировкой
6. Поиск без обновления страницы
7. Ограничение доступности страниц сайта
8. CRUD операции над сотрудниками на клиентской стороне
9. Возможность добавлять фотографию
10. Автоматическое перераспределение сотрудников в случае увольнения начальника
11. Древовидная форма должна быть процедурно генерируемой (не загружать вложенных подчиненных)
12. Изменение начальника с помощью drag-n-drop операции
13. Реализация rest-framework интерфейса (не было в техзадании, но было интересно разобраться)

Оглавление

Реализация ORM

Создание моделей

Нам понадобятся следующие модели для наших задач:

- Position
- Department
- Employee

Более подробный вид полей:

```
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(...)
```

На самом деле, поля salary и expected_workers в Position являются лишь вспомогательными и будут использоваться только при “засеве” базы данных.

Также стоит заметить, что хотя сотрудник может и не иметь отдела или начальника, он обязательно должен иметь должность, иначе будет каскадно удален.

Создание базы данных

Django очень упрощает данный шаг. Нам достаточно лишь убедиться, что наши модели работают и применить следующие команды:

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

“Засев” базы данных

Есть несколько вариантов “засева” базы данных, в том числе упрощенные (lazy). Однако, поскольку я хотел реализовать иерархию сотрудников через отношение объектов (каждый сотрудник ссылается на своего начальника), я выбрал “засев” с помощью миграции базы данных.

Для начала, создаем пустую миграцию командой:

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

Затем, внутри миграции указываем функцию, которую хотим выполнять:

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

Внутри seeder создаются фиктивные должности, отделы и сотрудники (см. employees/migrations/0002_employee_seeder.py). Все с соблюдением необходимой иерархии. Не забываем также про условие с 5 уровнями иерархии, начиная от управляющего директора и заканчивая junior программистом.

Работа с объектами

Для лучшего понимания работы с объектами я рекомендую ознакомиться с employees/views.py и employees/models.py. Я оставил там довольно много комментариев, объясняющих, какие именно действия я совершаю.

Однако вкратце стоит упомянуть, что основным инструментом работы с объектами, представляющими базу данных, являются так называемые QuerySet. Они предоставляют очень удобный и эффективный интерфейс для работы с запросами к базам данных.

Пример (получение сотрудника, который занимает должность потенциального начальника искомого, а также не состоит ни в одном отделе или состоит в том же отделе, что и искомый):

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

Пояснение: логика такова, что сотрудник может быть подчиненным другого сотрудника только в том случае, если тот:

1. Занимает должность, которая является главенствующей должностью для должности подчиненного
2. Состоит в том же отделе, что подчиненный или хотя бы один из них не состоит ни в одном отделе

Эта логика не форсируется на уровне объекта и выставление некорректного начальника возможно. Однако я стараюсь избегать подобных ситуаций.

Реализация front-end части

Базовая стилизация с использованием bootstrap

Включение bootstrap-стилизации в страницу неимоверно простое. Заходим на официальную страницу bootstrap и находим подобную строку:

```
<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">
```

Затем просто добавляем её в свою страницу любым способом. Теперь мы можем использовать html классы bootstrap.

Подобным образом я включил в проект jQuery, jQuery-UI, Popper и Bootstrap JS.

Для форм Django я также использовал django-crispy-forms:

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

Навигация по сайту

Я использовал navbar от bootstrap, однако с небольшой модификацией.

```
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")
    }});
```

Данный код получает id оглавления и в зависимости от него (id) выбирает активный элемент навигационной панели.

Древовидное отображение сотрудников

Для реализации дерева сотрудников я использовал подобие рекурсии. Сначала мы загружаем на страницу всех сотрудников, у которых нет начальника. У каждого из этих сотрудников вложен список их подчиненных (если они есть) в collapsible списке.

Каждый из вложенных сотрудников имеет точно такую же структуру, чем достигается подобие рекурсии.

Однако на самом деле, списки вложенных сотрудников ничего не содержат до момента, когда пользователь их откроет. Подробнее об этом далее.

Личные страницы сотрудников

Почти все операции, связанные с сотрудниками, привязаны к их id в базе данных. Не являются исключением и личные страницы сотрудников. Каждая ссылка на сотрудника содержит в себе его id, чем мы и пользуемся при вызове view метода.

```
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)
```

Эти три адресации нам необходимо включить в urlpatterns.

Каждый из соответствующих view будет принимать employee_id в качестве именованного аргумента. Далее находим необходимого сотрудника в базе данных и выполняем необходимые действия.

Поиск

Для поиска сотрудника, осуществляем фильтрацию по каждому слову из поискового запроса. Если каждое из них находится хотя бы в одном из полей (должность, отдел, имя, фамилия) сотрудника, этот сотрудник подходит под запрос.

Затем, отфильтрованных сотрудников сортируем по выбранному полю.

Результаты поискового запроса также необходимо ограничить в пределах запрашиваемой страницы.

Создание, редактирование и удаление сотрудников

Каждая страница сотрудника содержит в себе ссылки на страницу удаления и страницу редактирования этого сотрудника. О том, как работает маршрутизация этих страниц, упоминалось выше.

Также добавляем в навигационную панель ссылку на страницу создания нового сотрудника.

Принцип работы основан на использовании Django forms. Формы могут хранить в себе данные из объектов и преобразовывать их для работы с ними с клиентской стороны. Пользователю передается форма, которую, в свою очередь, можно инициализировать из данных объекта или наоборот, записывать из неё объект в базу данных.

Пример использования формы для редактирования данных сотрудника:

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

Формы очень упрощают работу с объектами и нередко являются основным связующим звеном между front-end и back-end частями реализации объектов.

Реализация ajax-функционала

Для реализации любого функционала с запросами к серверу, но без перезагрузки страницы нам понадобятся ajax-запросы. С их помощью мы можем отправлять запросы к серверу, не прерывая работы javascript на данной странице и не вызывая переход на новую страницу.

Я использовал ajax-функции из jQuery. Они удобнее в использовании и занимают меньше места в коде, чем “ванильный” код javascript.

Древовидное отображение сотрудников работает следующим образом: если запрос не содержит в себе аргумента ‘id’, отобразить исходное состояние страницы, иначе вернуть рендер со списком подчиненных. При нажатии на кнопку “показать подчиненных”, помимо прочей логики, будет также выполнен ajax-запрос с аргументом, представляющим id данного сотрудника. То есть, поддерживается подобие рекурсивной логики, при котором каждый сотрудник может содержать в себе других сотрудников. Однако, вместе с тем, мы можем использовать тот же view (и тот же адрес) и для рендера исходной страницы.

При поисковом запросе ajax работает с почти тем же принципом - если пользователь загружает страницу, отдаём её целиком. Если же выполняется ajax запрос - отдаём только рендер результатов поиска.

Регистрация и логин

Логику регистрации и логина реализует моё приложение users. Для логина и разлогина я использовал стандартный функционал Django:

```
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"),]
```

Форму с регистрацией отдает и принимает views.register.

Drag-n-drop

В jQuery-UI есть очень полезные и удобные методы draggable и droppable, которые берут большую часть графической работы на себя.

Используя аргумент `hoverClass: "on-draggable-hover"` сообщаем, что пока перетаскиваемый объект находится над данным, данному объекту будет присвоен класс `"on-draggable-hover"`. Не забываем изменить стилизацию данного класса для визуального отображения пользователю:

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

В аргумент drop записываем функцию с ajax-запросом, которая и попросит у сервера выполнить необходимое действие. В данном случае сервер попытается назначить начальником “перетаскиваемого” сотрудника того сотрудника, на которого “перетащили”. Затем, сообщаем пользователю о результате попытки.

Дополнительная back-end логика

Ограничение доступа

Для начала, импортируем очень удобный декоратор:

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

Однако, данный декоратор, при попытке зайти на страницу, переадресовывает пользователя на форму логина для администрации. Нам же необходим другой адрес переадресации. Передаем его нашему декоратору в качестве аргумента:

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

Также я дописал свой собственный декоратор user_access_error для сообщения пользователю об ошибке доступа.

Работа с файлами

Я использовал Pillow и models.ImageField для реализации поля фото в объекте сотрудника:

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

Также необходимо указать, куда именно файлы будут загружаться:

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

Thumbnail картинки - это она же, просто с меньшим параметром ширины.

Back-end генерация пагинатора

Интерактивная генерация пагинатора для страницы поиска сотрудников происходит в back-end. В зависимости от текущей страницы, размера страниц и количества элементов в результате поискового запроса, выбираем, какая страница должна быть активной, какие не стоит показывать вовсе.

Подробнее в employees/utilities.py.

Перераспределение подчиненных в случае удаления начальника

В случае удаления сотрудника выполнится следующий код:

```
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()
```

В случае удаления сотрудника, для каждого его подчиненного попытаться найти другого подходящего начальника, иначе обнулить поле начальника.

Использование Rest Framework

Устанавливаем djangorestframework:

```
pip install djangorestframework
```

Пишем сериалайзеры для каждой из моделей (employees/api/serializers.py). Пишем view для каждой из моделей (employees/api/views.py). Затем создаем маршрутизацию:

```
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)
```

Добавляем rest-пагинацию для сотрудников, (50 тысяч объектов - это очень много):

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

Делаем view сотрудников приватным:

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