ТЕКСТ БЕЗ ИЗМЕНЕНИЙ КАК ЕСТЬ

Сайт позволит зарегистрированным пользователям публиковать объявления о продаже чего-либо. Объявления будут разноситься по рубрикам, причём структура рубрик будет иметь два уровня иерархии: на первом уровне расположатся рубрики общего плана («Недвижимость», «Транспорт» и пр.), а на втором – более конкретные («Жильё», «Гаражи», «Дачи», «Легковой», «Грузовой»).

Для вывода списка объявлений мы применим пагинацию, т.к. объявлений может оказаться очень много, и страница, содержащая все объявления, будет слишком большой. Также мы предусмотрим возможность поиска объявлений по введённому посетителем слову.

Под любым объявлением (на странице сведений об объявлении) может быть составлено произвольное количество комментариев. Оставлять комментарии будет позволено любому пользователю, в том числе и гостю.

В составе объявления пользователь может поместить основную графическую иллюстрацию, которая будет выводится и в списке объявлений, и в составе сведений об объявлении, а также произвольное количество дополнительных иллюстраций, которые можно будет увидеть лишь на странице сведений об объявлении. И основная и дополнительные иллюстрации не являются обязательными к размещению.

Процедура регистрации нового пользователя на сайте будет разбита на два этапа. На первом этапе посетитель вводит свои данные на странице регистрации, после чего на указанный им адрес электронной почты приходит письмо с гиперссылкой, ведущей на страницу активации. На втором этапе посетитель переходит по гиперссылке, полученной в письме, попадает на страницу активации и становится полноценным пользователем.

Сайт доски объявлений включит в себя следующие страницы:

* Главная – показывающая десять последних опубликованных объявлений без разбиения их на рубрики;
* Страница списка объявлений – показывающая (с использованием пагинации) объявления из определённой рубрики. Также она будет содержать форму для поиска объявлений по введённому слову;
* Страница сведений о выбранном объявлении – выведет ещё все оставленные для него комментарии и форму для добавления нового комментария;
* Страницы регистрации и активации нового пользователя.
* Страницы входа и выхода.
* Страница профиля зарегистрированного пользователя – выведет список объявлений, оставленных текущим пользователем;
* Страницы добавления, правки, удаления объявлений;
* Страницы изменения пароля, правки и удаления пользовательского профиля;
* Страницы сведений о сайте, о правах его разработчика, пользовательского соглашения и пр.

Таков в общих чертах план сайта. Остальное будет уточняться далее……

Создание проекта

**django-admin startproject имяпроекта**

Переходим в папку проекта и подаём команду на создание приложения

**manage.py startapp ИмяПриложения**

Установка Bootstrap:

**pip install django-bootstrap4**

Запуск проекта

**manage.py runserver**

Создание миграций

**manage.py makemigration**

Выполнение миграции

**manage.py migrate**

После выполнения миграций, создадим Суперпользователя, подав команду:

**manage.py createsuperuser**

Введём выбранные нами имя, адрес электронной почты и пароль создаваемого пользователя.

Напоследок откроем модуль admin.py пакета приложения, в котором объявляются классы-редакторы и регистрируются модели в административном сайте. Зарегистрируем нашу модель пользователя, добавив в этот модуль код:

**from .models import AdvUser**

**admin.site.register(AdvUser)**

AdvUser – Имя модели.

**--------------------------------------------------------------------------------**

**main.views.py. index** - выводит Главную страницу (Контроллер)

**main/templates/main/index.html** - Шаблон главной страницы

**Вывод вспомогательных страниц**

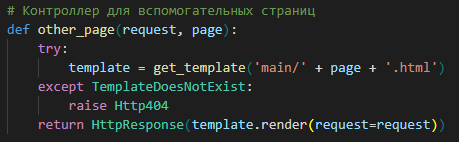
Первый способ, самый очевидный, заключается в том, что для каждой страницы пишется отдельный контроллер и отдельный маршрут. Этот способ весьма трудоёмок, поскольку придётся писать несколько контроллеров с практически одинаковым кодом, и подходит лишь для тех случаев, когда страницы должны выводить какие-либо данные, извлекаемые из базы или формируемые программно.

Второй способ – вывод всех страниц с применением одного контроллера и, соответственно, одного маршрута. Какой либо идентификатор страницы, предназначенной к выводу на экран, передаётся контроллеру с URL-параметром. Трудоёмкость работы в таком случае существенно снижается, поскольку нужно написать один контроллер.

**path(‘<str:page>/’, other\_page, name=’other’),**

Имя шаблона выводимой страницы передаём через URL-параметр **page**. А контроллер, выводящий вспомогательные страницы, назовём **other\_page** и реализуем в виде функции. Вообще, контроллеры-функции – идеальный инструмент для написания чего-либо нестандартного, специфического.

**main.views.other\_page -** Вывод вспомогательных страниц



Имя выводим страницы получаем из параметра **page**, добавляем к нему путь и расширение, получив тем самым полный путь к нужному шаблону, и пытаемся загрузить его вызовом функции **get\_template().** Если загрузка прошла успешно, то формируем на основе этого шаблона страницу.

Если же шаблон загрузить не удалось, то функция **get\_template()** возбудит исключение **TemplateDoesNotExist**. Мы перехватываем это исключение и возбуждаем другое исключение – **Http404**, которое приведёт к отправке страницы с сообщением об ошибке 404 (запрошенная страница не существует).

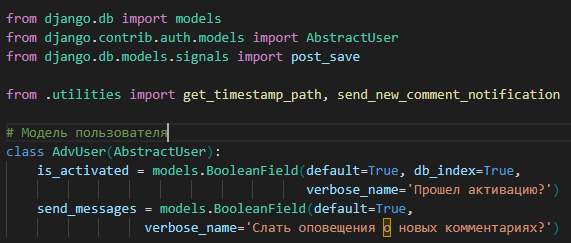
**main/templates/main/about.html** - Шаблон страницы со сведениями о сайте и правах его разработчиков. У этого шаблона нет контроллера. Ссылка на эту страницу в базовом шаблоне.

-------------------------------------------------------------------------

РАБОТА С ПОЛЬЗОВАТЕЛЯМИ РАЗГРАНИЧЕНИЕ ДОСТУПА

Теперь займёмся инструментами для работы с пользователями и разграничения доступа. Мы создадим страницы для входа и выхода, регистрации, активации, страницы пользовательского профиля, для смены данных о пользователе, смены его пароля и удаления профиля.

Стандартная модель пользователя **User**, предлагаемая стандартным же приложением **django.contrib.auth**, не подходит, поскольку нам нужно хранить дополнительные данные о пользователе. Поэтому создадим свою собственную модель, сделав её производной от абстрактной модели **AbstractUser**, объявленной в модуле **django.contrib.auth**.



После создания модели пользователя **main.models.AdvUser**

Создаём миграци (отладочный веб-сервер при этом должен быть остановлен)

**manage.py makemigration**

Выполняем миграции

**manage.py migrate**

После выполнения миграций, создадим Суперпользователя, подав команду:

**manage.py createsuperuser**

Введём выбранные нами имя, адрес электронной почты и пароль создаваемого пользователя.

Напоследок откроем модуль admin.py пакета приложения, в котором объявляются классы-редакторы и регистрируются модели в административном сайте. Зарегистрируем нашу модель пользователя, добавив в этот модуль код:

**from .models import AdvUser**

**admin.site.register(AdvUser)**

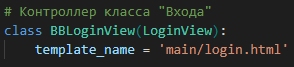
**AdvUser** – Имя модели.

**--------------------------------------------------------------------------**

Для реализации входа создадим подкласс контроллера-класса LoginView, назовём его **BBLoginView**

**main.views.BBLoginView**





В классе мы указали лишь путь к файлу шаблона, занеся его в атрибут **template\_name.** Остальные параметры сохранят значения по умолчанию.

Запишем новый маршрут, указывающий на контроллер BBLoginView, в списке уровня приложения

****

****

В маршруте мы указали шаблонный путь accounts/login/. По умолчанию именно по нему Django выполняет перенаправление при попытке гостя получить доступ к закрытой от него странице.

**НА ЗАМЕТКУ**

Мы могли бы записать маршрут и таким образом:

**from django.contrib.auth.views import LoginView**

**urlpatterns = [**

**path(‘accounts/login/’,**

***LoginView.as\_view(template\_name=’main/login.html’), name=’login’),***

]

Указав в маршруте непосредственно класс **LoginView** и записав все необходимые параметры контроллера в вызове метода **as\_view().** Но в таком случае код контроллеров окажется записан в двух модулях: **view.py** и **urls.py**, и в дальнейшем при сопровождении сайта, в поисках нужного фрагмента кода нам придётся просматривать оба этих модуля, что неудобно.

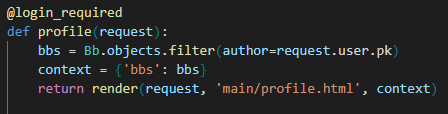
Поэтому давайте держать ***код контроллеров*** в модуле **views.py**, а ***код списка маршрутов*** – в модуле **urls.py**. Так нам будет проще.

**-----------------------------------------------------------------------------------------------**

**Пользовательский профиль.**

main.views.profile





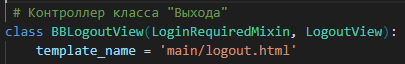
Поскольку страница пользовательского профиля должна быть доступна только зарегистрированным пользователям, выполнившим вход на сайт, мы пометили контроллер-функцию **profile()** декоратором **login\_required().**

**----------------------------------------------------------------------------------------------------**

**Страница выхода**







Страница выхода должна быть доступна только зарегистрированным пользователям, выполнившим вход. Поэтому мы добавили в число суперклассов контроллера-класса **BBLogoutView** примесь **LoginRequiredMixin**.

**Веб-страницы правки личных данных пользователя**

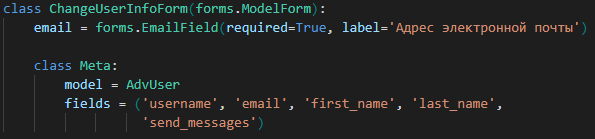
На этой странице пользователь сможет исправить свои имя (логин), адрес электронной почты, реальные имя, фамилию и признак, хочет ли он получать по электронной почте оповещения о появлении новых комментариев к его объявлениям. Адрес электронной почты будет обязательным к заполнению.

Сначала объявим форму **ChangeUserInfoForm**, связанную с моделью **AdvUser** и предназначенную для ввода основных данных.

**main.forms.ChangeUserInfoForm**

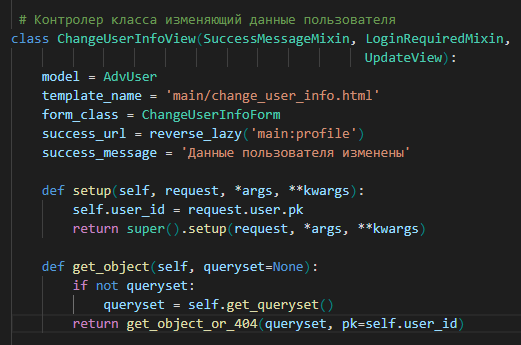






Так как мы хотим сделать поле email модели **AdvUser** обязательным для заполнения, то выполним полное объявление поля email формы. А поскольку параметры остальных полей формы: **username, first\_name, last\_name** и **send\_messages** – у нас не меняются, в их отношении применим быстрое объявление.

Контроллер страницы основных данных должен выполнять правку записи модели, так что мы можем написать его на базе высокоуровнего класса UpdateView

****

В процессе работы этот контроллер должен извлечь из модели **AdvUser** запись, представляющую текущего пользователя, для чего ему нужно предварительно получить ключ текущего пользователя. Получить его можно из объекта текущего пользователя, хранящегося в атрибуте **user** объекта запроса.

Вероятно, наилучшее место для получения ключа текущего пользователя – метод **setup(),** наследуемый всеми контроллерами-классами от их общего суперкласса **View**. Этот метод выполняется в самом начале исполнения контроллера-класса и получает объект запроса в качестве одного из параметров. В переопределённом методе **setup()** мы извлечём ключ пользователя и сохраним его в атрибуте **user\_id**.

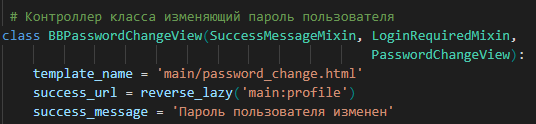
Извлечение исправляемой записи выполним в методе **get\_object(),** который контроллер-класс унаследовал от примеси **SingleObjectMixin**. В переопределённом методе сначала учитываем тот момент, что набор записей, из которого следует извлечь искомую запись, может быть передан методу с параметром **queryset**, а может быть и не передан – в этом случае набор записей следует получить вызовом метода **get\_queryset().** После этого непосредственно ищем запись, представляющую текущего пользователя.

В качестве одного из суперклассов этого контроллера-класса мы указали примесь **LoginRequiredMixin**, запрещающую доступ к контроллеру гостям, и примесь **SuccessMessageMixin**, которая применяется для вывода всплывающих сообщений об успешном выполнении операции.

----------------------------------------------------------------------------------------------------

Страница правки пароля

**main.views.BBPasswordChangeView**



После усрешной смены пароля выполняем перенаправление на страницу профиля пользователя с выводом соответствующего всплывающего сообщения.

------------------------------------------------------------------------------------------------------------------

**РЕГИСТРАЦИЯ И АКТИВАЦИЯ ПОЛЬЗОВАТЕЛЕЙ**

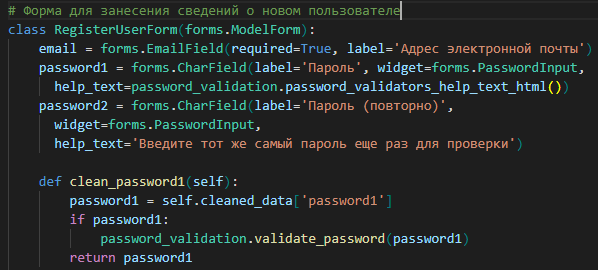
Напишем форму для ввода сведений о новом пользователе, контроллеры и шаблоны для страниц непосредственно регистрации и уведомлений об успешной регистрации.

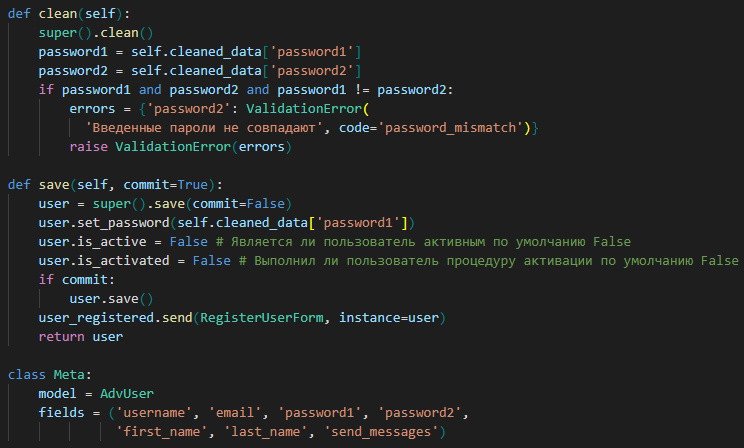
Для отправки письма о необходимости активации мы объявим свой сигнал. Называться он будет **user\_registered** и получит в качестве единственного параметра **instance** объект вновь созданного пользователя.

Сигнал мы объявим в модуле **apps.py** пакета приложения. Этот модуль выполняется непосредственно при инициализации приложения и, таким образом, является идеальным местом для записи кода, объявляющего сигналы.

**Форма для заненсения сведений о новом пользователе**

main.forms.RegisterUserForm





Здесь мы также комбинируем быстрое и полное объявление полей. Полное объявление используем для создания полей электронной почты (поскольку хотим сделать его обязательным для заполнения) и обоих полей для занесения пароля.

В качестве дополнительного поясняющего текста у первого поля пароля указываем объединённый текст с требованиями к вводимому паролю, предоставленный всеми доступными в системе валидаторами.

В методе **clean\_password1()** выполняем валидацию пароля, введённого в первое поле, с применением доступных в системе валидаторов пароля. Проверять таким же образом пароль из второго поля нет необходимости. Если пароль из первого поля некорректен, не имеет значения, является ли корректным пароль из второго поля.

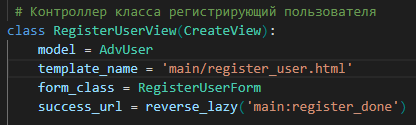
В переопределённом методе **clean()** проверяем, совпадают ли оба введённых пароля. Эта проверка будет проведена после валидации пароля из первого поля.

В переопределённом методе **save()** при сохранении нового пользователя заносим значение **False** в поля **is\_active** (признак, является ли пользователь активным) и **is\_activated** (признак, выполнил ли пользователь процедуру активации), тем самым сообщая фреймворку, что этот пользователь ещё не может выполнять вход на сайт. Далее сохраняем закодированный пароль и отправляем сигнал **user\_registered**, чтобы отослать пользователю письмо с требованием активации.

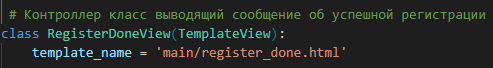
**Средства для регистрации пользователя**

Эти средства будут включать две страницы: одна выведет веб-форму для ввода данных о регистрирующемся пользователе, вторая сообщит об успешной регистрации и отправке письма с требованиями активации.

Контроллер-класс, регистрирующий пользователя, мы назовём **RegisterUserView** и сделаем производным от класса **CreateView**.



Контроллер, который выведет сообщение об успешной регистрации, будет называться **RegisterDoneView** и, в силу его исключительной простоты, станет производным от класса **TemplateView**.



Шаблоны

**main/templates/main/register\_user.html** - Страница регистрации

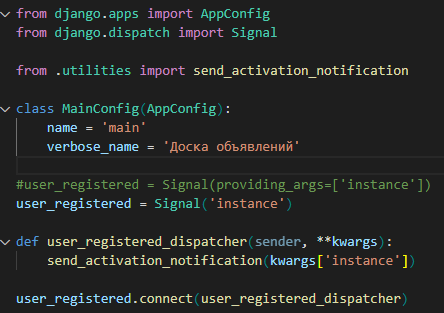
**main/templates/main/register\_done.html** - Уведомление об успешной регистрации

---------------------------------------------------------------------------------------------------------

Средства для отправки писем с требованиями активации

Непосредственно рассылку электронных писем будет выполнять функция **send\_activation\_notification()**, которую объявим чуть позже, во вновь созданном модуле **utilities.py.**

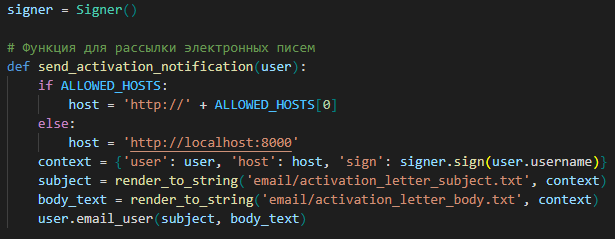
Объявим сигнал **user\_registered** в модуле **apps.py** и привяжем к нему обработчик



Создадим в пакете приложения модуль utilities.py и объявим в нём функцию send\_activation\_notification()







Чтобы сформировать интернет-адрес, ведущий на страницу подтверждения активации, во-первых, домен, на котором находится наш сайт, а во-вторых, некоторое значение, уникально идентифицирующее только что зарегистрированного пользователя и при этом устойчивое к попыткам его подделать.

Домен мы можем извлечь из списка разрешённых доменов, который записан в параметре **ALLOWED\_HOST** настроек проекта. Выберем самый первый домен, присутствующий в списке. Если же список доменов пуст, мы задействуем интернет-адрес, используемый отладочным веб-сервером **Django**.

В качестве уникального и стойкого к подделке идентификатора пользователя применяем его имя, защищённое цифровой подписью. Создание цифровой подписи выполняется посредством класса **Signer**.

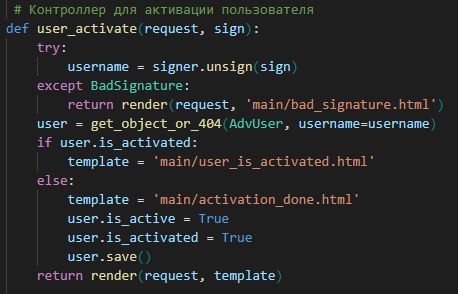
Текст темы и тела письма формируем с применением текстовых шаблонов

----------------------------------------------------------------------------------------------------------------

**Страницы активации пользователя.**

Чтобы реализовать активацию нового пользователя, напишем Контроллер и три Шаблона. Они создадут страницы с сщщбщением об успешной активации, о том что активация была выполнена ранее, и о том, что цифровая подпись у идентификатора пользователя, полученного в составе интернет-адреса, скомпроментирована.

Контроллер реализуем в виде функции **user\_activate()**



Подписанный идентификатор пользователя, передаваемый в составе интернет-фдреса, получаем с параметром **sign**. Далее извлекаем из него имя пользователя, ищем пользователя с таким именем, делаем его активным, присвоив значения **True** полям **is\_active** и **is\_activated** модели, и выводим страницу с сообщением об успешной активации. Если цифровая подпись оказалась скомпроментированной, выводим страницу с сообщением о неуспехе активации, а если пользователь был активирован ранее (поле **is\_activated** уже хранит значение **True**) – страницу с сообщением, что активация уже произошла.

Для обработки подписанного значения используем экземпляр класса **Signer**, созданный в модуле **utilities.py** и хранящийся в переменной **signer**. Так мы сэкономим оперативную память.

Шаблоны

**main/templates/activation\_done.html** - Успешная активация

**main/templates/bad\_signature.html** - Ошибка активации

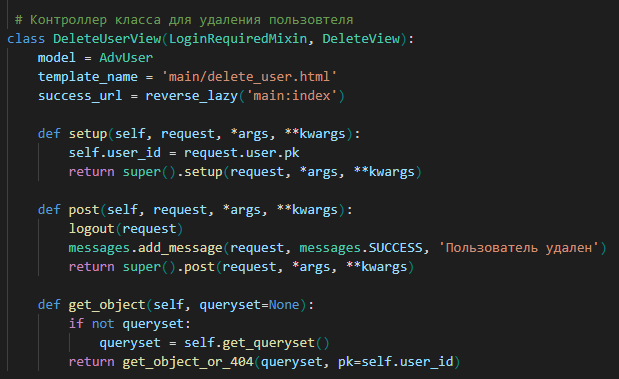
**main/templates/user\_is\_activated.html** - Пользователь уже активирован

-------------------------------------------------------------------------------------------------------

Страница удаления пользователя

Контроллер-класс **DeleteUserView**, удаляет текущего пользователя. Производный от класса **DeleteView**.

**Main.views.DeleteUserViews**



Здесь используются те же программные приёмы, что и в контроллере ChangeUserInfoView.

Перед удалением текущего пользователя необходимо выполнить выход, что мы и сделали в переопределённом методе **post().** В этом же методе **post()** мы создали всплывающее сообщение об успешном удалении пользователя.

Шаблон

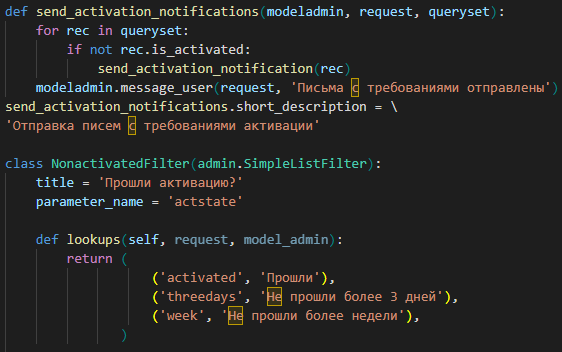
main/templates/main/delete\_user.html

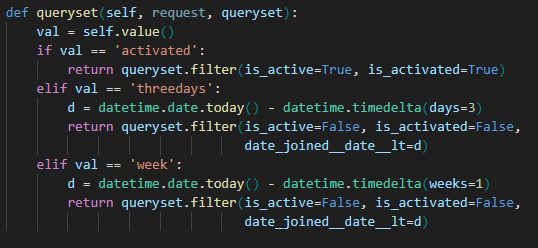
**-------------------------------------------------------------------------------------------------------------**

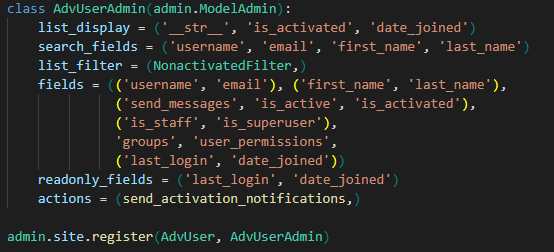
**Инструменты для администрирования пользователей**

Редактор, посредством которого администрация сайта будет работать с зарегистрированными пользователями. В него добавим возможность фильтрации пользователей по именам, адресам электронной почты, уже выполнивших активацию, не выполнивших её в течении трёх дней и недели, и действие по отправке выбранным пользователям писем с требованиями пройти активацию.

**main.admin.AdvUserAdmin**

****

****

****

В списке записей указываем выводить строковое представление записи (имя пользователя – как реализовано в модели **AbstractUser**, от которой наследует наша модель), поле признака, выполнил ли пользователь активацию, временнУю отметку его регистрации. Также разрешаем выполнять фильтрацию по полям имени, адреса электронной почты, настоящих имени и фамилии.

При выполнении фильтрации пользователей, выполнивших активацию, не выполнивших её в течении трёх дней и недели, используем класс **NonactivatedFilter**. Обратим внимание на код, непосредственно фильтрующий пользователей по значению даты их регистрации.

Мы явно указываем список полей, которые должны выводится в формах для правки пользователей, чтобы выстроить их в удобном для работы порядке. Поля даты регистрации пользователя и последнего его входа на сайт делаем доступными только для чтения.

Наконец, регистрируем действие, которое разошлёт пользователям письма с предписаниями выполнить активацию. Это действие реализовано функцией **send\_activation\_notifications().** В ней мы перебираем выбранных пользователей и для каждого, кто не выполнил активацию, вызываем функцию **send\_activation\_notification(),** объявленную ранее в модуле **utilities.py** и непосредственно производящую отправку писем.

**---------------------------------------------------------------------------------------------------------------------**

**РУБРИКИ**

**Базовая модель рубрик**

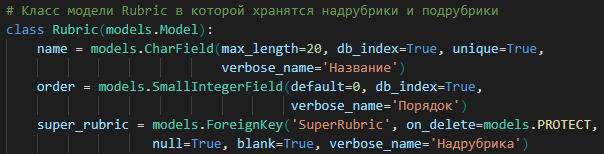
|  |  |  |  |
| --- | --- | --- | --- |
| Имя | Тип | Дополнительные параметры | Описание |
| name | CharField | Длинна – 20 символов, индексированое | Название |
| order | IntegerField | Значение по умолчанию – 0, индексированое | Порядок |
| super\_rubric | ForeignKey | Необязательное запрет каскадного удаления | Надрубрика |

Поле **order** будет хранить целое число, обозначающее порядок следования рубрик друг за другом: при выводе рубрики сначала будут сортироваться по возрастанию значения порядка, а уже потом - по их названиям.

Поле **super\_rubric** будет хранить надрубрику, к которой относится текущая подрубрика. Оно будет иметь следующие важные особенности:

* Связь, создаваемая этим полем, должна устанавливаться с моделью надрубрик, которую мы объявим чуть позже. Условимся называть эту модель **SuperRubric**;
* Это поле будет заполнятся только в том случае, если запись хранит подрубрику. Если запись хранит надрубрику, то поле заполнять не нужно (собственно, отсутствие значения в этом поле является признаком надрубрики – ведь надрубрика в принципе не может ссылаться на надрубрику). Поэтому поле сделано необязательным для заполнения.
* Нужно обязательно запретить каскадное удаление записей, чтобы пользователь по ошибке не удалил надрубрику вместе со всеми подрубриками.

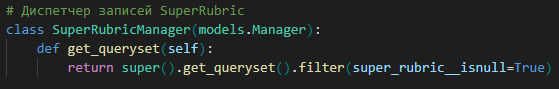
Класс модели **Rubric**

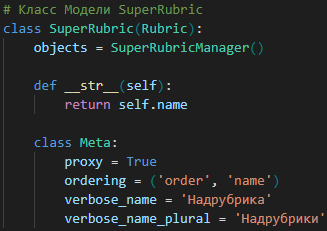


**Модель надрубрик**

Для работы с надрубриками объявим прокси-модель **SuperRubric**, производную от **Rubric** (прокси-модель позволяет менять лишь функциональность модели, но не набор объявленных в ней полей)

Чтобы изменить состав обрабатываемых моделью записей, нужно задать для неё свой диспетчер записей, который и укажет необходимые условия фильтрации.





Условия фильтрации записей указываем в переопределённом методе **get\_queryset()** класса диспетчера записей **SuperRubricManager**. Он станет выбирать только записи с пустым полем **super\_rubric**, т.е. надрубрики.

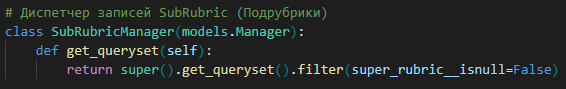
В самом классе **SuperRubric** задаём диспетчер записей **SuperRubricManager** в качестве основного. И не забываем объявить метод **\_\_str\_\_** , который станет генерировать строковое представление надрубрики – её название.

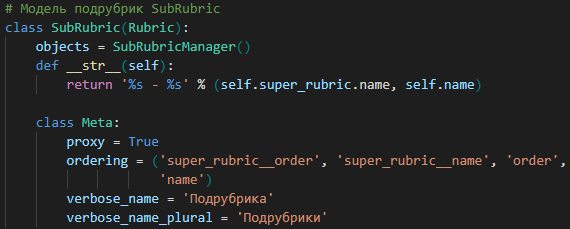
Как условились ранее, указываем сортировку записей сначала по возрастанию значений порядка, а потом – по названию.

--------------------------------------------------------------------------------------------------------------

**Модель подрубрик**

Модель подрубрик SubRubric мы создадим таким же образом, как и модель надрубрик. Только теперь диспетчер записей, который мы создадим для неё, будет выбирать лишь подрубрики.





Диспетчер записей **SubRubricManager** будет отбирать лишь записи с непустым полем **super\_rubric** (т.е. подрубрики). Строковое представление, создаваемое моделью, будет выполнено в формате **<*Название надрубрики*> - <*Название подрубрики*>** . А сортировку записей укажем по порядку надрубрики, названию надрубрики, порядку подрубрики, названию подрубрики.

Объявив все необходимые классы, остановим отладочный сервер (если он работает), создадим и выполним миграции:

**manage.py makemigrations**

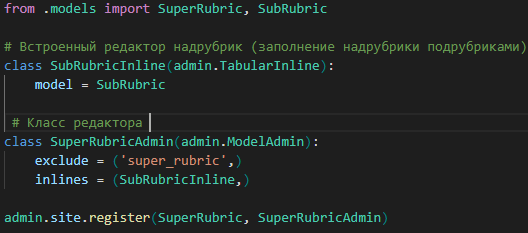
**manage.py migrate**

**--------------------------------------------------------------------------------------------------**

**Инструменты для администрирования рубрик**

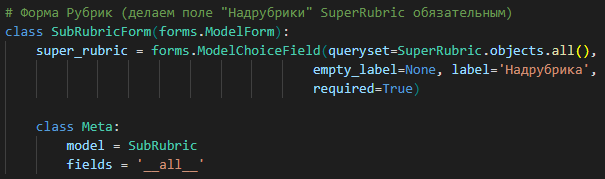
Вся работа с надрубриками и подрубриками будет проводится средствами административного сайта.

Для надрубрик мы создадим встроенный редактор, чтобы пользователь, добавив новую надрубрику, смог сразу же заполнить её подрубриками. Из формы для ввода и правки надрубрик мы исключим поле надрубрики (**super\_rubric**), поскольку оно там совершенно не нужно и, более того, собьёт пользователя с толку.



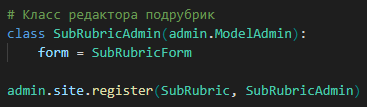
Код редакторов, ровно как и код, регистрирующей модели и редакторы в подсистеме административного сайта, должен записываться в модуль **admin.py** пакета приложения.

У подрубрик сделаем поле надрубрики (**super\_rubric**) обязательным для заполнения. Для этого мы объявим форму **SubRubricForm**, в модуле **forms.py** пакета приложения.

****

Мы убрали у раскрывающегося списка, с помощью которого пользователь будет выбирать подрубрику, «пустой» пункт, присвоив параметру **empty\_label** конструктора класса поля **ModelChoiceField** значение **None**. Так мы дополнительно дадим понять, что в это поле обязательно должно быть занесено значение.

Объявление класса редактора.



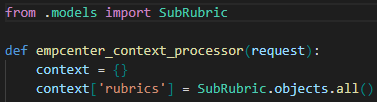
-------------------------------------------------------------------------------------------------

**Вывод списка рубрик в вертикальной панели навигации**

Сначала необходимо поместить в состав контекста каждого шаблона переменную, в которой хранится список подрубрик (именно на его основе мы будем формировать пункты панели навигации). Можно создать такую переменную в каждом контроллере, но это очень трудоёмко. Поэтому объявим и зарегистрируем в проекте обработчик контекста, в котором и будет формироваться список подрубрик.

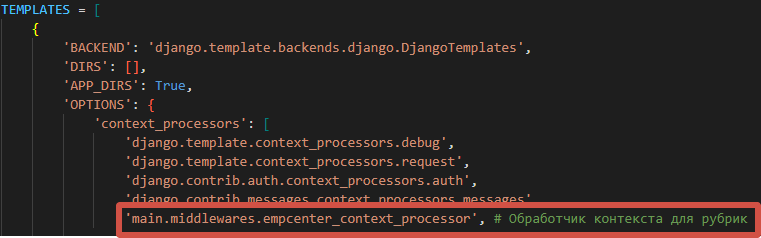
Условимся, что список подрубрик будет помещаться в переменную **rubrics** контекста шаблона.

Создадим в пакете приложения модуль **middlewares.py** и запишем в него код обработчика контекста **bboard\_context\_processor()**

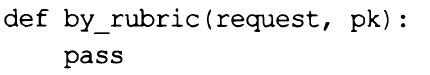
****

****

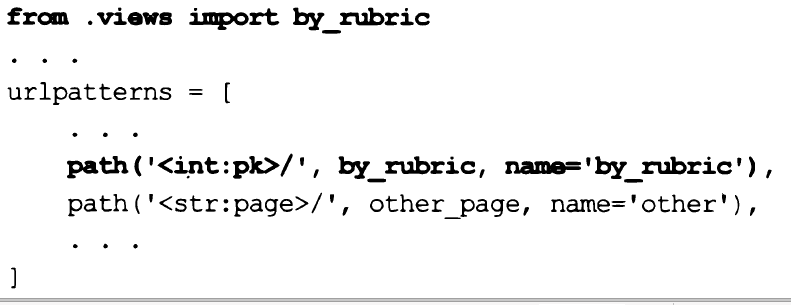
В модуле **settings.py** пакета конфигурации зарегистрируем только что написанный обработчик контекста.



Выполним ещё несколько подготовительных действий. Во-первых, в модуле **views.py** пакета приложения объявим «пустой» контроллер-функцию **by\_rubric():**



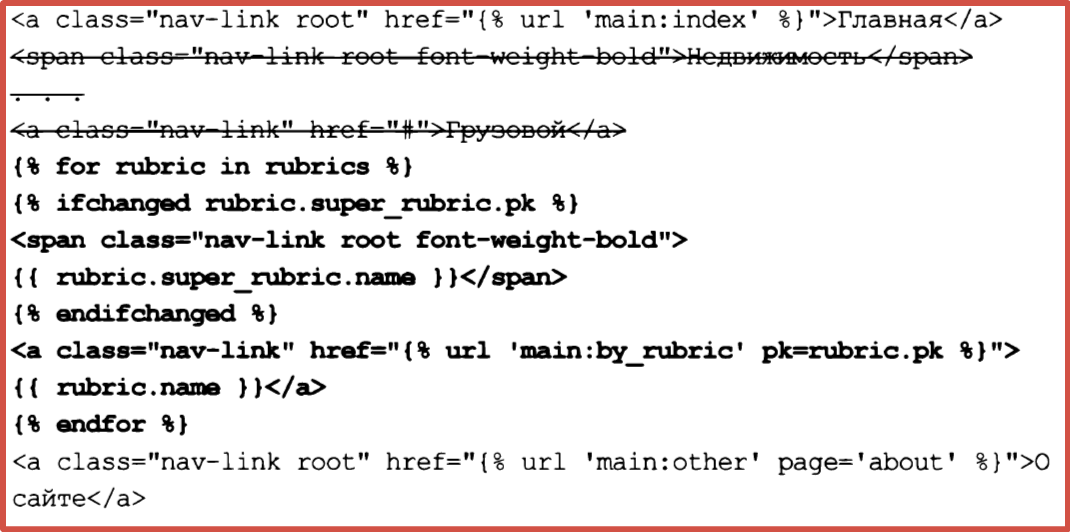
Во-вторых, добавим в список маршрутов уровня приложения маршрут, ведущий на этот контроллер:



Этот маршрут мы поместим перед маршрутом, ведущим на контроллер **other\_page(),** который выводит вспомагательные страницы. Если же мы поместим его после упомянутого ранее маршрута, то при просмотре списка маршрутов Django примет присутствующий в интернет-адресе ключ рубрики за имя шаблона страницы и запустит контроллер **other\_page(),** что приведёт к ошибке 404.

Теперь мы можем сформировать в панели навигации гиперссылки с правильными интернет-адресами.

Откроем базовый шаблон basic.html



Мы перебираем список подрубрик, хранящийся в переменной rubrics контекста шаблона (эту переменную создал наш обработчик контекста bboard\_context\_processor()), и для каждой подрубрики выводим:

* Если ключ связанной надрубрики изменился (т.е. если начали выводится рубрики из другой надрубрики) – пункт с именем надрубрики
* Пункт-гиперссылку с именем подрубрики.

-------------------------------------------------------------------------------------------------------------------

**ОБЪЯВЛЕНИЯ**

Создадим страницу для просмотра объявлений, относящихся к выбранной рубрике, с поддержкой пагинации и поиска, страницу сведений о выбранном объявлении, страницы для добавления, правки и удаления объявлений. На странице профиля выведем объявления оставленные текущим пользователем.

**Подготовка к обработке выгруженных файлов.**

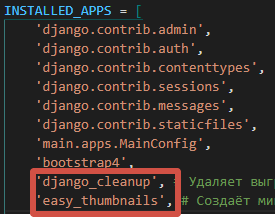
В составе каждого объявления будет присутствовать графическое изображение с основной иллюстрацией к продаваемому товару. Помимо этого, пользователь может создать в объявлении произвольное количество дополнительных иллюстраций.

Чтобы **Django** смог обработать выгруженные посетителями файлы, необходимо установить три дополнительные библиотеки: **Easy Thumbnails** (создаёт миниатюры), **django-cleanup** (удаляет выгруженные файлы после удаления хранящих их записей моделей) и **Pillow** (обеспечивает поддержку графики, будет автоматически установлена при установке **Easy Thumbnails**).

**pip install easy-thumbnails**

**pip install django-cleanup**

Добавим программные ядра двух последних библиотек: приложения **easy\_thumbnails** и **django\_cleanup** - в список зарегистрированных в проекте (**settings.py**)

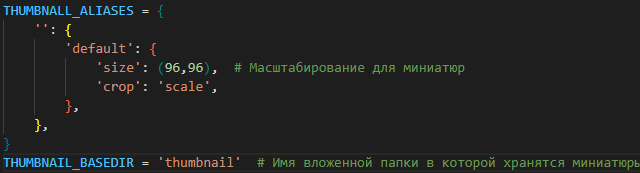


Для хранения выгруженных файлов отведём папку **media**, которую создадим в папке проекта. Для хранения миниатюр создадим в ней папку **thumbnails**.

В модуле **settings.py** укажем путь к папке **media** и префикс для интернет-адресов выгруженных файлов:



И сразу же добавим туда настройки приложения **easy\_thumbnails**



Мы задали для миниатюр один-единственный пресет, указывающий выполнять простое масштабирование до размеров 96х96 пикселов. Также мы задали имя вложенной папки.

Добавим в список маршрутов уровня проекта маршрут для обработки выгруженных файлов:



-------------------------------------------------------------------------------------------------------------------------

**Модели объявлений и дополнительных иллюстраций**

Создадим две модели: одну - для объявлений и вторую – для дополнительных иллюстраций

**Модель самих объявлений**

Модель, хранящая объявления называется

|  |  |  |  |
| --- | --- | --- | --- |
| Имя | Тип | Дополнительные параметры | Описание |
| rubric | ForeignKey | Запрет каскадного удаления | Подрубрика |
| title | CharField | Длина - 40 символов | Название товара |
| content | TextField |  | Описание товара |
| price | IntegerField | Значение по умолчанию - 0 | Цена товара |
| contacts | TextField |  | Контакты |
| image | ImageField | Необязательное | Основная иллюстрация к объявлению |
| author | ForeignKey |  | Пользователь, оставивший объявление |
| is\_active | BooleanField | Значение по умолчанию – True, индексированное | Признак, показывать ли объявление в списке |
| created\_at | DateTimeField | Подставляется текущее значение даты и времени, индексированное | Дата и время публикации объявления |

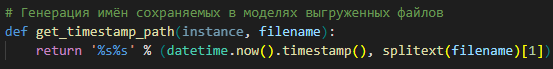
В поле **rubric**, устанавливается связь с моделью подрубрик **SubRubric**, мы указали запрет каскадного удаления, чтобы предотвратить случайное удаление подрубрики вместе со всеми относящимися к ней объявлениями.

Графические файлы, сохраняемые в поле **image** модели, будут иметь в качестве текущих имён текущие временнЫе отметки. Так мы приведём имена к единому типу и заодно устраним ситуацию, когда имя выгруженного файла настолько длинное, что оно не помещается в поле модели.

При разработке модели объявления нужно учесть ещё один момент. Чуть позже мы напишем модель дополнительных иллюстраций, которую свяжем с моделью объявлений связью «один-со-многими». Если при объявлении этой связи мы разрешим каскадное удаление, то при удалении объявления будут уничтожены все относящиеся к нему дополнительные иллюстрации. Но это действие выполнит не Django, а СУБД, отчего приложение **django\_cleanup** не получит сигнала об удалении записей и не сможет в ответ удалить хранящиеся в них графические файлы. В результате эти файлы останутся на диске бесполезным мусором.

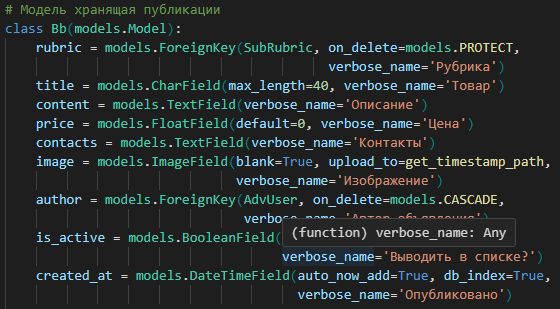
Решим эту проблему чуть позже. А сейчас условимся об имени модели дополнительных иллюстраций – **AdditionalImage**.

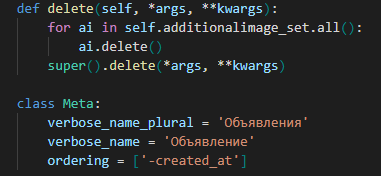
Ранее мы создали в пакете приложения модуль **utilities.py**, в который записали объявление функции, выполняющей отправку писем. Этот модуль – отличное место для хранения кода, не относящегося напрямую ни к моделям, ни к редакторам, ни к контроллерам. Поместим в него объявление функции **get\_timestamp\_path(),** генерирующей имена сохраняемых в модели выгруженных файлов.



Код модели **Bb**

****

****



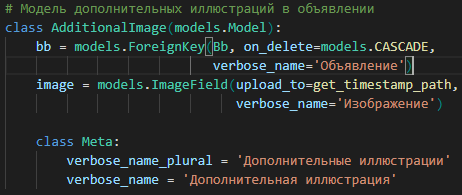
В переопределённом методе **delete()** перед удалением текущей записи мы перебираем и вызовом метода **delete()** удаляем все связанные дополнительные иллюстрации. При вызове метода **delete()** возникает сигнал **post\_delete**, обрабатываемый приложением **django\_cleanup**, которое в ответ удалит все файлы, хранящиеся в удалённой записи.

**Модель дополнительных иллюстраций.**

Модель дополнительных иллюстраций мы назвали AdditionalImage

|  |  |  |
| --- | --- | --- |
| Имя | Тип | Описание |
| bb | ForeignKey | Объявление, к которому относится иллюстрация |
| image | ImageField | Иллюстрация |

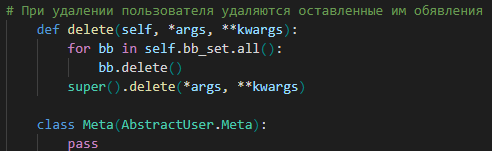
Графические файлы, сохранённые в поле image, также получат в качестве имён текущие временнЫе отметки. Для формирования имён файлов применим объявленную ранее функцию **get\_timestamp\_path()** из модуля **utilities.py**



Сохраняем код, создаём и выполняем миграции.

**Реализация удаления объявлений в модели пользователя**

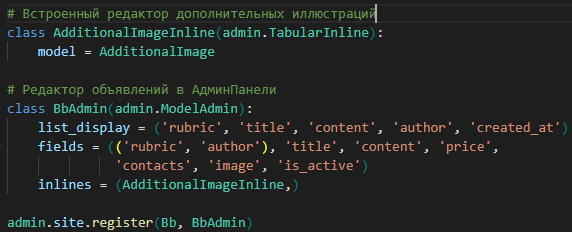
Добавляем в код модели AdvUser следующий фрагмент



**Инструменты для администрирования объявлений**

Чтобы с объявлениями можно было работать посредством административного сайта, объявим редактор объявлений **BbAdmin** и встроенный редактор дополнительных иллюстраций **AdditionalImageInline**.





На страницах добавления и правки объявлений выведем раскрывающиеся списки подрубрики и пользователя в одну строку – ради компактности.

**Вывод объявлений**

Мы создадим две страницы:

* Страницу списка объявлений, относящихся к выбранной посетителем рубрике, с поддержкой пагинации и поиска объявлений по введённому слову;
* Страницу сведений о выбранном объявлении, на которой будут выводится также и дополнительные иллюстрации.

Кроме того, реализуем вывод десяти наиболее «свежих» объявлений на главной странице.

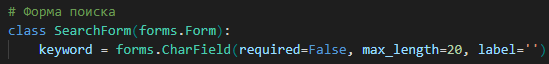
**Вывод списка объявлений**

Вывод списка объявлений с поддержкой поиска – достаточно сложная задача. Нам понадобится обычная, не связанная с моделью, форма для ввода искомого слова, контроллер и шаблон. А ещё придётся решить весьма серьёзную проблему корректного возврата, о котором поговорим чуть позже.

**Форма поиска и контроллер списка объявлений**

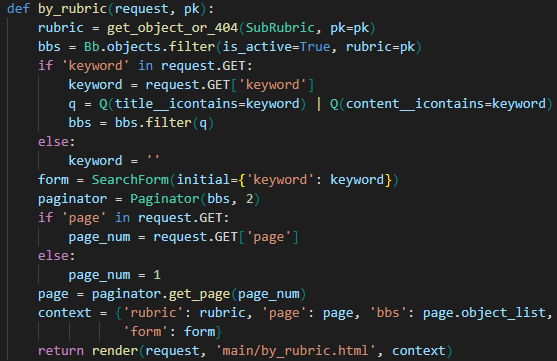
Сразу условимся, что искомое слово, введённое посетителем, будет пересылать контроллеру методом **GET** в **GET**-параметре **keyword**. Поле в форме для ввода искомого слова назовём так же.

Код формы поиска

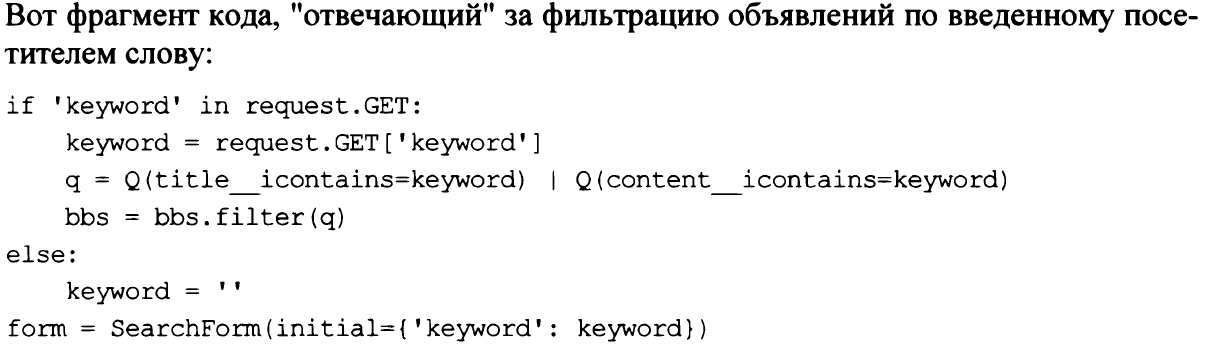


Поскольку посетитель может ввести в поле **keyword** искомое слово, а может и не ввести (чтобы отменить выполненный ранее поиск и вновь вывести все объявления из списка), мы пометили это поле как необязательное для заполнения. А ещё убрали из этого поля надпись, присвоив параметру **label** пустую строку.

Наполняем содержимым функцию **by\_rubric()** из модуля **views.py**



Извлекаем выбранную посетителем рубрику – нам понадобиться вывести на странице её название. Затем выбираем объявления, относящиеся к этой рубрике и помеченные для вывода (те, у которых поле **is\_active** хранит значение **True**). После этого выполняем фильтрацию уже отобранных объявлений по введённому посетителем искомому слову, взятому из **GET**-параметра **keyword**.



Ради простоты получаем искомое слово непосредственно из **GET**-параметра **keyword**. Затем формируем на основе полученного слова условие фильтрации, применив объект **Q**, и выполняем фильтрацию объявлений.

Далее создаём экземпляр формы **SearchForm**, чтобы вывести её на экран. Конструктору её класса в параметре **initial** передаём полученое ранее искомое слово, что-бы оно присутствовало в выведенной на экран форме.

**Реализация корректного возврата**

Предположим ситуацию. Посетитель заходит на сайт, выбирает какую-либо рубрику, пролистывает несколько частей, сформированных пагинатором, находит нужное ему объявление и щёлкает на гиперссылке, чтобы посмотреть это объявление полностью. Открывается страница со сведениями об объявлении, после чего щёлкает на гиперссылке возврата на список объявлений…. И попадает на самую первую часть этого списка.

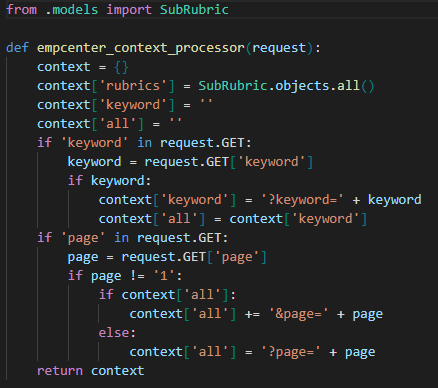
То же самое произойдёт, если посетитель выполнит поиск, а уже потом отправится смотреть сведения о каком либо объявлении. Когда он щелкнет на гиперссылке возврата, то вернётся в изначальный список объявлений, в котором не был выполнен поиск.

Как избежать этой проблемы, в общем понятно. Номер выводимой части и искомое слово у нас передаются посредством **GET**-параметров **page** и **keyword** соответственно. Тогда, чтобы вернуться на нужную часть списка уже отфильтрованных по заданному слову объявлений, следует передать эти параметры странице сведений об этом объявлении.

Конечно, готовый набор **GET**-параметров можно получить из элемента с ключом **QUERY\_STRING** словаря, который хранится в атрибуте **META** объекта запроса. Но нет смысла передавать параметр **page**, если его значение равно **1**, и параметр **keyword** с пустой строкой – это их значения по умолчанию.

Также можно формировать набор GET-параметров в контроллере. Но по принятым в Django соглашениям, весь код, «ответственный» за формирование страниц, слудует помещать в шаблон, посредник – обработчик контекста.

В модуле middlewares.py пакета приложения, код обработчика контекста выглядит следующим образом.



Код создаёт в контексте шаблона две переменные:

**keyword** - с **GET**-параметром **keyword**, который понадобится для генерирования интернет-адресов в гиперссылках пагинатора;

**all** - с **GET**-параметрами **keyword** и **page**, которые мы добавим к интернет-адресам гиперссылок, указывающих на страницы сведений об объявлениях.

**Шаблон веб-страницы списка объявлений**