**django-admin startproject DjangoProject – создание нового проекта**

**Django-admin startapp Приложение – создание нового приложения**

**Для создания супер-админа python manage.py createsuperuser.**

BASE\_DIR – корневая папка проекта.

SECRET\_KEY – обеспечивает целостность передачи данных.

DEBUG – дебаг режим разработки сайта.

ALLOWED\_HOSTS ­– домены для проекта

INSTALLED\_APPS – установленные приложения

ROOT\_URLCONF – где располагаются Url адреса

TEMPLATES – определяет настройки для шаблонов

DATABASES – определяет используемую базу данных

AUTH\_PASSWORD\_VALIDATORS – автоматическая валидация пароля

LANGUAGE\_CODE – язык

TIME\_ZONE – временной пояс

STATIC\_URL – хранит информацию о корневом месте хранения всех статических файлов – не загружаемые из вне файлы(css,js,images)

STATICFILES\_DIRS – список файлов и папок со статикой

Для хранения медиа-файлов: MEDIA\_URL = '/media/'  
MEDIA\_ROOT = BASE\_DIR/'media'

В urls добавляется:

if settings.DEBUG:

urlpatterns+=static(settings.MEDIA\_URL,document\_root=settings.MEDIA\_ROOT)

Django.contrib.admin – админ панель

Django.contrib.auth – приложение для работы с пользователем

Основа джанго – приложения. Любой чих – новое приложение. Любое приложение нужно регистрировать в settings.py.

Состав приложения:

1. Admin.py – регистрация таблиц модели в админке для манипуляций с ними.
2. Apps.py – хранит данные о конфигурации приложения.
3. Models.py – файл для создания таблиц базы данных.
4. Tests.py – тесты для приложения джанго.
5. Views.py – создание шаблонов страниц для их отображения
6. Migrations – миграция базы данных.

Как работает джанго:

1. Происходит http запрос
2. Запрос отправляется в urls.py
3. Запрос перенаправляется на соответствующее представления / контроллер
4. Происходит чтение запись данных из модели и подгружается шаблон страницы
5. Выдается http ответ(в виде html)

Функции = контроллеры = представления в views.py

Обычно для оживления шаблона сохраняются необходимые данные в объект -словарь, который потом будет использоваться в шаблоне. Данный словарь называется контекстом. Словарь содержит переменный(placeholders) и теги(шаблонные теги). Обращение к переменной {{ключ в контексте}}. Теги обычно используются для условной отрисовки {% if ключ\_тега %} {% endif %}, циклов {% for I in smth %} {% endfor %}

{% load static %} – загружает относительный путь до папки со статикой.

{% static ‘путь в папке ’%} – склеивает путь до статики с путем до файла внутри папки со статикой.

Для составления динамических ссылок используется следующий тег:

{% url 'name из файла url' %}

Для составления шаблонов для шаблона используется конструкция вида {% block block\_name %} {% endblock %}. В начале html использующего такой шаблон используется конструкция {% extends ‘путь\_до\_шаблона от папки templates’ %}. Затем в {% block block\_name %} {% endblock %} вставляется тот контент, который должен содержаться в нашей странице на основе шаблона. Можно и наоборот, включать шаблоны {% include ‘путь до шаблона’ %}

Models используются для создания таблиц базы данных приложения. В Django автоматически создается поля id в качестве первичного ключа. Иначе при создания поля добавляем параметр primary\_key=True. Параметр default – значение по умолчанию, null и blank – определяют возможность пустых значений в поле. Upload\_to поле с папкой для загрузки изображений.

Для переноса структуры классов в sql используются миграции – по сути перенос структуры модели на структуру БД. Два этапа: makemigrations – создание миграции и migrate – применение миграции.

Для создания записи в таблице – модели создается экземпляр соответствующего класса модели с параметрами конструктора равными значениям полей соответствующей таблицы: с = ProductCategory(name=”JJ”,description=”Desc”). Для сохранения данной вставки вызывается метод save от созданного экземпляра класса: c.save()

Для извлечения 1 объекта из таблиц используется конструкция имя\_модели.objects.get(значение параметров для определения извлекаемой записи).

Для создания объекта можно также использовать команду имя\_модели.objects.create(значения полей у новой записи): ProductCategory.objects.create(name=”FFF”)

Для получения всех объектов из таблицы используется метод all. Он возвращает особый список querySet, к которому применимы все методы обычных списков.

Для получения нескольких объектов по условию используется метод filter:

ProductCategory.objects.filter(description=None). Также возвращает querySet.

Для регистрации модели: в admins.py admin.site.register(имя\_модели)

DJangoFixtures используются для хранения структуры и содержимого бд в виде какого-либо файла (например json). Пример создания:

python manage.py dumpdata products.Product > goods.json.

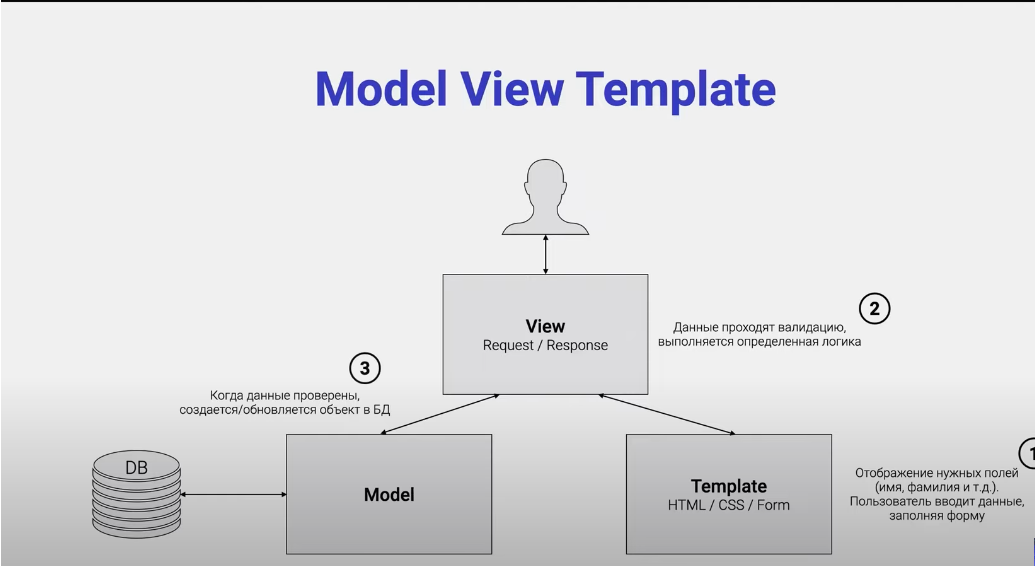
Для загрузки данных: python manage.py loaddata products/fixtures/goods.json

Для добавления пространств ссылок используется конструкция вида:

path('имя\_пространства/',include('приложение.urls',namespace='имя\_пространсва'))

Для передачи параметров помимо request в urls объявляется путь следующего вида: path('baskets/add/<int:product\_id>/', basket\_add, name='basket\_add'),

Пример передачи доп аргумента: {% url 'products:basket\_add' product.id %}



Внутри каждой формы создается класс мета, принимающий в качестве полей модель для работы с формой и поля формы. Формы передаются в шаблоны через контекст. Обращение к форме идет также как и к placeholder. При обращение обычно выбирают метод отображения формы (.as\_p)

Проверка формы на валидность form.is\_valid()

Создание полей формы: либо через класс мета,либо в классе формы в виде:

username = forms.CharField(widget=forms.TextInput(  
 attrs={  
 'class':'form-control py-4','placeholder':'Введите имя пользователя'  
 }  
))

Авторизация состоит из 3 этапов: 1) Audit – контроль 2) Authentication – подтверждение подлинности 3) Authorisation – разрешение

Аутентификация пользователя: auth.authenticate(username,password)

Авторизация: auth.login(request,user)

Можно создавать свои менеджеры моделей: В данном случае в качестве self будут выступать все существующие объекты модели.

objects = BasketQuerySet.as\_manager()

class BasketQuerySet(models.QuerySet):  
 def total\_sum(self):  
 return sum(basket.sum() for basket in self)  
 def total\_quantity(self):  
 return sum(basket.quantity for basket in self)

Для создания пагинации в Django существует класс Paginator(object\_list, per\_page), с его помощью можно из списка объектов выделять страницы методом: paginator.page(number=page\_number). После этого можно обращаться к этому же пагинатору через созданный предыдущим методом объект. У пагинатора есть свои методы.

ModelAdmin:

Use the **fields** option to make simple layout changes in the forms on the “add” and “change” pages such as showing only a subset of available fields, modifying their order, or grouping them into rows. For example, you could define a simpler version of the admin form for the **[django.contrib.flatpages.models.FlatPage](https://docs.djangoproject.com/en/5.0/ref/contrib/flatpages/" \l "django.contrib.flatpages.models.FlatPage" \o "django.contrib.flatpages.models.FlatPage)** model as follows:

**class** **FlatPageAdmin**(admin.ModelAdmin):

fields = ["url", "title", "content"]

To display multiple fields on the same line, wrap those fields in their own tuple. In this example, the **url** and **title** fields will display on the same line and the **content** field will be displayed below them on its own line:

**lass** **FlatPageAdmin**(admin.ModelAdmin):

fields = [("url", "title"), "content"]

Set **list\_display** to control which fields are displayed on the change list table page of the admin.

Example:

list\_display = ["first\_name", "last\_name"]

If you don’t set **list\_display**, the admin site will display a single column that displays the **\_\_str\_\_()** representation of each object.

Set **list\_editable** to a list of field names on the model which will allow editing on the change list page. That is, fields listed in **list\_editable** will be displayed as form widgets on the change list page, allowing users to edit and save multiple rows at once.

Once you've got a lot of items in a list, it can be useful to be able to filter which items are displayed. This is done by listing fields in the list\_filter attribute.

readonly\_fields = ('name','description','image')  
search\_fields = ('name',)  
ordering = ('name','-category')