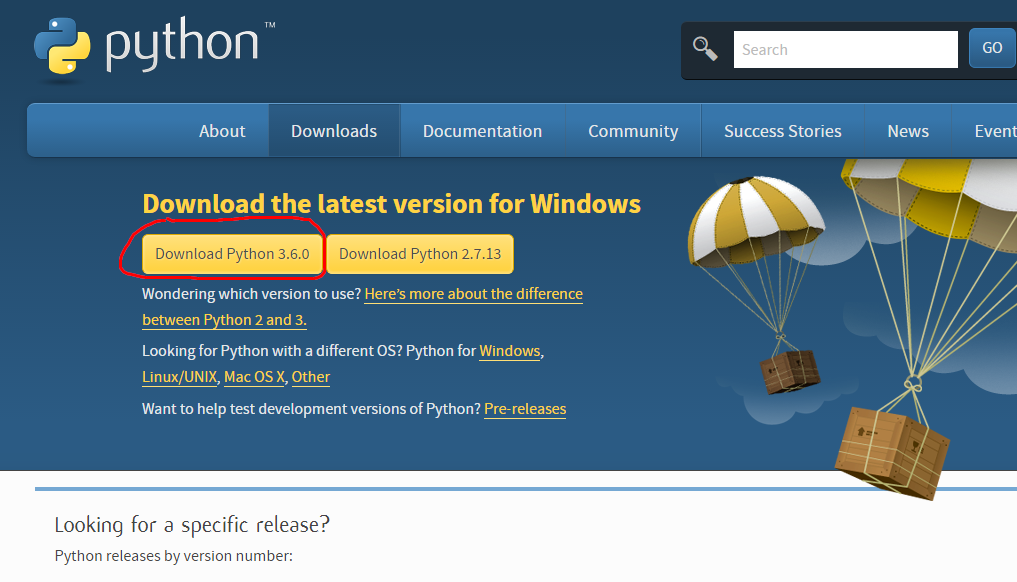
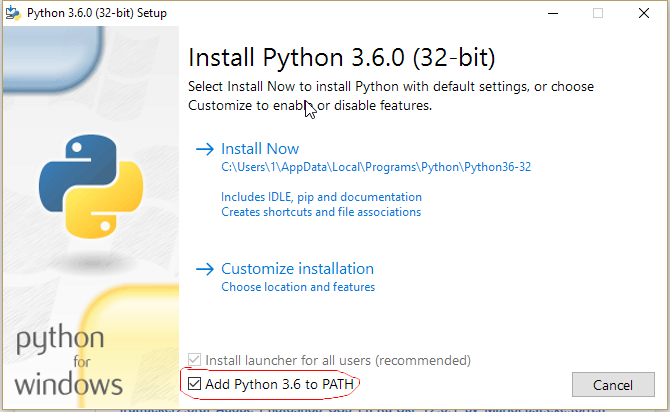
**Пошаговая инструкция по установке Python + Django для Windows**

1. Скачиваем Python с сайта <https://python.org/downloads/>

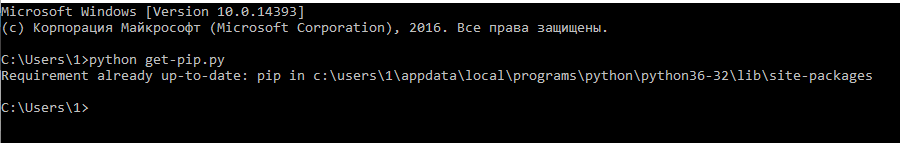


1. Производим установку, предварительно установив галочку «Add Python to PATH»



1. По завершении установки Python, установим *pip* – систему управления пакетами (в случае, если он еще не установлен). Для этого скачиваем файл по ссылке [*https://bootstrap.pypa.io/get-pip.py*](https://bootstrap.pypa.io/get-pip.py). В командной строке выполняем команду

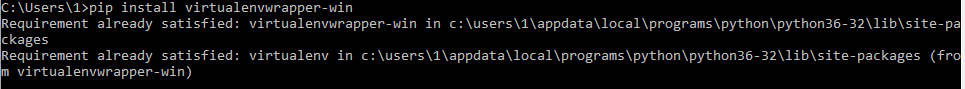
python get-pip.py



*(в моём случае pip уже установлен)*

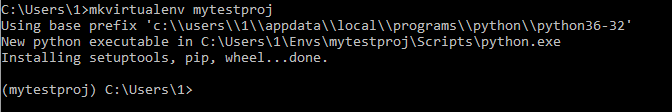
1. Далее установим *virtualenv*, который будет обеспечивать специализированную среду для каждого созданного проекта *Django*.

pip install virtualenvwrapper-win



1. Создадим виртуальное окружение для нашего проекта

mkvirtualenv myproject

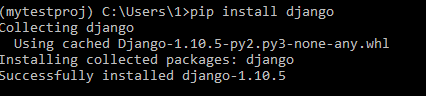


Теперь мы видим, что работа ведется в созданном виртуальном окружении по названию этого окружения в скобках. В случае если нужно снова в него попасть, набираем команду

workon mytestproj

1. Для установки Django набираем

pip install django

**

1. Для проверки работоспособности наберем следующие команды:

django-admin startproject mytestproj

Это создаст каталог mytestproj. который будет содержать папку-контейнер для нашего проекта, файл manage.py (скрипт, позволяющий взаимодействовать с проектом Django), файл инициализации, файл настроек и др.

cd mytestproj

Переход в папку mytestproj

python manage.py startapp djangoapp

Эта команда создаст каталог djangoapp. По умолчанию созданный каталог содержит файл [models.py](https://vk.com/away.php?to=http%3A%2F%2Fmodels.py) и другие файлы шаблона приложения.

/djangoapp

\_\_init\_\_.py

admin.py

migrations/

models.py

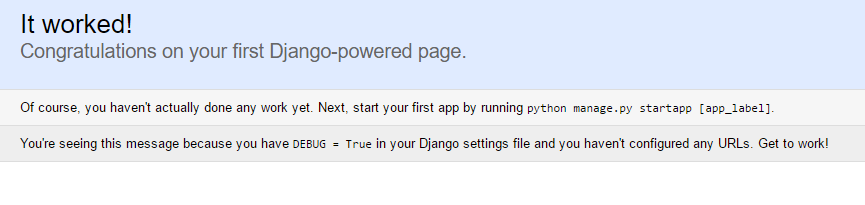
tests.py

views.py

python manage.py runserver

Произведен запуск веб-сервера

Переходим по адресу <http://localhost:8000/> и видим, что всё работает.



**PostGreSQL**

Для работы с базой данных PostgreSQL:

1. Скачиваем и устанавливаем pgAdmin - свободное кроссплатформенное программное обеспечение, предоставляющее графический интерфейс для работы с базой данных.

<http://www.postgresql.org/download/>

1. Устанавливаем psycopg2 - наиболее популярный PostgreSQL адаптер для языка программирования Python.

pip install psycopg2

В случае, если установить psycopg2 с помощью pip не удалось, это можно сделать вручную.

1. Скачиваем файл для установки по ссылке <http://www.lfd.uci.edu/~gohlke/pythonlibs/#psycopg>
2. Помещаем скачанный psycopg2 в папку site-packages, которая находится в папке с проектом, либо в папку с Python.
3. Выполняем команду *pip install psycopg2-2.6.1-cp35-none-win32.whl*
4. Далее найдем следующие строки в файле settings.py и заменим их соответствующими данными нашей БД  
     
   Строка

*'ENGINE': 'django.db.backends.postgresql\_psycopg2',*

должна быть одинакова у всех для использования постгреса.  
 *DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.postgresql\_psycopg2',   
        'NAME': 'postgres',  
        'USER': 'postgres',  
        'PASSWORD': '123',  
        'HOST': '',  
        'PORT': '',  
    }  
}*

1. Обратите внимание на настройку [*INSTALLED\_APPS*](http://djbook.ru/rel1.9/ref/settings.html#std:setting-INSTALLED_APPS) в начале файла. Она содержит названия всех приложений Django, которые активированы в вашем проекте. Приложения могут использоваться на разных проектах, вы можете создать пакет, распространить его и позволить другим использовать его на своих проектах.

По умолчанию, [INSTALLED\_APPS](http://djbook.ru/rel1.9/ref/settings.html#std:setting-INSTALLED_APPS) содержит следующие приложения, которые предоставляются Django:

* [django.contrib.admin](http://djbook.ru/rel1.9/ref/contrib/admin/index.html#module-django.contrib.admin) – интерфейс администратора.
* [django.contrib.auth](http://djbook.ru/rel1.9/topics/auth/index.html#module-django.contrib.auth) – система авторизации.
* [django.contrib.contenttypes](http://djbook.ru/rel1.9/ref/contrib/contenttypes.html#module-django.contrib.contenttypes) – фреймверк типов даных.
* [django.contrib.sessions](http://djbook.ru/rel1.9/topics/http/sessions.html#module-django.contrib.sessions) – фреймверк сессии.
* [django.contrib.messages](http://djbook.ru/rel1.9/ref/contrib/messages.html#module-django.contrib.messages) – фреймверк сообщений.
* [django.contrib.staticfiles](http://djbook.ru/rel1.9/ref/contrib/staticfiles.html#module-django.contrib.staticfiles) – фреймверк для работы со статическими файлами.

Эти приложения включены по умолчанию для покрытия основных задач.

Некоторые приложения используют минимум одну таблицу в базе данных, поэтому нам необходимо их создать перед тем, как мы будем их использовать. Для этого переходим в директорию нашего проекта, где лежит файл manage.py и выполняем команду

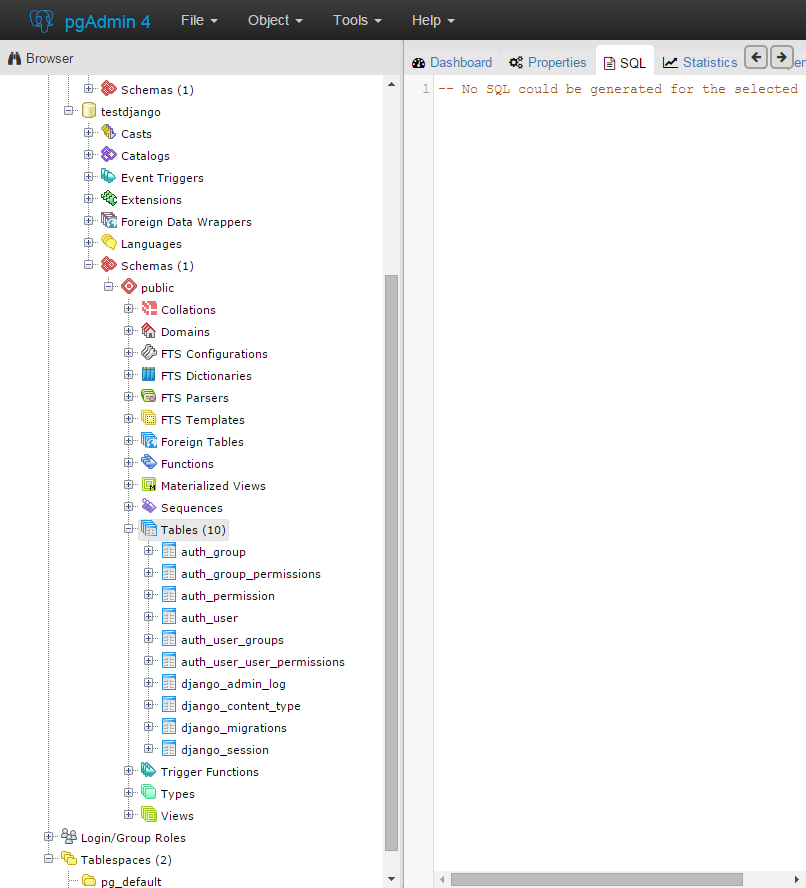
*python manage.py migrate*

Команда [*migrate*](http://djbook.ru/rel1.9/ref/django-admin.html#django-admin-migrate) проверяет настройку [*INSTALLED\_APPS*](http://djbook.ru/rel1.9/ref/settings.html#std:setting-INSTALLED_APPS) и создает все необходимые таблицы в базе данных, указанной в *mysite/settings.py*, применяя миграции, которые находятся в приложении

1. Создадим пользователя с правами администратора. Для этого используем команду

*python manage.py createsuperuser*

Вы должны будете ввести имя пользователя, e-mail, пароль и получите сообщение об успешном создании нового пользователя.  
Теперь если вы зайдете pgAdmin, вы увидите созданные новые таблицы



1. Запускаем сервер в консоли в директории с файлом manage.py .  
   *python manage.py runserver*  
   Переходим по ссылке http://127.0.0.1:8000/admin/, вводим данные и попадаем в админку.

**MySQL**

Для работы с базой данных MySQL нужно провести предварительную настройку:

1. Произведем установку драйвера базы данных. Существуют следующие драйвера для работы с MySQL:

* MySQLdb (отсутствует поддержка Python 3)
* mysqlclient
* MySQL Connector
* Pymysql

В нашем примере будем использовать mysqlclient

**В случае использования Pymysql для корректной работы необходимо добавить следующий код в файл *init.py*:**

***import pymysql***

***pymysql.install\_as\_MySQLdb()***

Произведем его установку с помощью

*pip install mysqlclient.*

1. Откроем файл *settings.py* и в секции *DATABASES* сделаем следующую запись:

DATABASES = {

'default': {

'default': {

'ENGINE': 'django.db.backends.mysql',

'NAME': 'django\_test',

'USER': 'root',

'PASSWORD': '',

'HOST': '',

'PORT': '3306',

'HOST': 'localhost', }

}

}

Пароль по умолчанию для базы MySQL пустой.



python manage.py migrate

Команда [*migrate*](http://djbook.ru/rel1.9/ref/django-admin.html#django-admin-migrate) проверяет настройку [*INSTALLED\_APPS*](http://djbook.ru/rel1.9/ref/settings.html#std:setting-INSTALLED_APPS) и создает все необходимые таблицы в базе данных, указанной в *mysite/settings.py*, применяя миграции, которые находятся в приложении (мы расскажем об это ниже). Вы увидите сообщение для каждой примененной миграции.

1. Если вам интересно, запустите консоль базы данных и введите

*SHOW TABLES;*

чтобы посмотреть таблицы, которые создал Django.

**Создадим простую модель.**

Первый шаг в создании Web-приложения в *Django* – создание его модели, которая является, по сути, схемой базы данных с дополнительными метаданными.

Модель - это основной источник данных. Он содержит набор полей и поведение данных, которые вы храните. *Django* следует принципу *DRY* (*Don’t* *repeat* *yourself*). Смысл в том, что бы определять модели в одном месте.

В нашем простом приложении голосования, мы создадим две модели: *Question* и *Choice*. *Question* содержит вопрос и дату публикации. *Choice* содержит: текст ответа и количество голосов. Каждый объект *Choice* связан с объектом *Question*.

Эти понятия отображаются простыми классами *Python*. Отредактируйте файл *djangoapp* */models.py*, чтобы он выглядел таким образом:

from django.db import models

class Question(models.Model):

question\_text = models.CharField(max\_length=200)

pub\_date = models.DateTimeField('date published')

class Choice(models.Model):

question = models.ForeignKey(Question, on\_delete=models.CASCADE)

choice\_text = models.CharField(max\_length=200)

votes = models.IntegerField(default=0)

Код очень простой. Каждая модель представлена подклассом [*django.db.models.Model*](http://djbook.ru/rel1.4/ref/models/instances.html#django.db.models.Model)*.* Каждая модель содержит несколько атрибутов, каждый из которых отображает поле в таблице базы данных.

Каждое поле представлено экземпляром класса [*Field*](http://djbook.ru/rel1.4/howto/custom-model-fields.html#django.db.models.Field) – например, [*CharField*](http://djbook.ru/rel1.4/ref/models/fields.html#django.db.models.CharField) для текстовых полей и [*DateTimeField*](http://djbook.ru/rel1.4/ref/models/fields.html#django.db.models.DateTimeField) для полей даты и времени. Это указывает *Django* какие типы данных хранят эти поля.

Названия каждого экземпляра [*Field*](http://djbook.ru/rel1.4/howto/custom-model-fields.html#django.db.models.Field) (например, *question* или *pub\_date* ) это название поля, в “машинном”(machine-friendly) формате. Вы будете использовать эти названия в коде, а база данных будет использовать их как названия колонок.

Вы можете использовать первый не обязательный аргумент конструктора класса [*Field*](http://djbook.ru/rel1.4/howto/custom-model-fields.html#django.db.models.Field), чтобы определить отображаемое, удобное для восприятия, название поля. Оно используется в некоторых компонентах *Django*, и полезно для документирования. Если это название не указанно, *Django* будет использовать “машинное” название. В этом примеры, мы указали отображаемое название только для поля *Poll*.*pub*\_*date*. Для всех других полей будет использоваться “машинное” название.

Некоторые подклассы [*Field*](http://djbook.ru/rel1.4/howto/custom-model-fields.html#django.db.models.Field) содержат обязательные параметры. [*CharField*](http://djbook.ru/rel1.4/ref/models/fields.html#django.db.models.CharField), например, требует аргумент max\_*length*. Этот аргумент используется не только при создании таблицы, но и при проверке данных, что мы скоро увидим.

Заметим, что связь между моделями определяется с помощью [*ForeignKey*](http://djbook.ru/rel1.4/ref/models/fields.html#django.db.models.ForeignKey). Это указывает *Django* что каждый *Choice* связан с одним объектом *Poll*. *Django* поддерживает все основные типы связей: многое-к-одному, многое-ко-многим и один-к-одному.

Первым делом мы должны указать нашему проекту, что приложение установлено. Отредактируйте файл *settings.py* и измените настройку [*INSTALLED\_APPS*](http://djbook.ru/rel1.4/ref/settings.html#std:setting-INSTALLED_APPS) добавив строку ' *djangoapp* '. В результате получим:

INSTALLED\_APPS = [

'djangoapp.apps.FirstConfig',

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

]

Теперь *Django* знает что необходимо использовать приложение *djangoapp*. Давайте выполним следующую команду:

python manage.py makemigrations djangoapp

Выполняя *makemigrations*, вы говорите *Django*, что внесли некоторые изменения в ваши модели (в нашем случае мы создали несколько новых) и хотели бы сохранить их в миграции.

Миграции используются *Django* для сохранения изменений ваших моделей (и структуры базы данных) - это просто файлы на диске. Вы можете изучить миграцию для создания ваших моделей, она находится в файле *djangoapp/migrations/0001\_initial.py*. Не волнуйтесь, вам не нужно каждый раз их проверять. Но их формат удобен для чтения на случай, если вы захотите внести изменения.

В *Django* есть команда, которая выполняет миграции и автоматически обновляет базу данных - она называется [*migrate*](http://djbook.ru/rel1.9/ref/django-admin.html#django-admin-migrate).

Если необходимо, можете выполнить [*python manage.py check*](http://djbook.ru/rel1.9/ref/django-admin.html#django-admin-check). Эта команда ищет проблемы в вашем проекте не применяя миграции и не изменяя базу данных.

Теперь, выполните команду [*migrate*](http://djbook.ru/rel1.9/ref/django-admin.html#django-admin-migrate) снова, чтобы создать таблицы для этих моделей в базе данных:

python manage.py migrate

Команда [migrate](http://djbook.ru/rel1.9/ref/django-admin.html#django-admin-migrate) выполняет все миграции, которые ещё не выполнялись, (*Django* следит за всеми миграциями, используя таблицу в базе данных *django*\_*migrations*) и применяет изменения к базе данных, синхронизируя структуру базы данных со структурой ваших моделей.

Миграции позволяют изменять ваши модели в процессе развития проекта без необходимости пересоздавать таблицы в базе данных. Их задача изменять базу данных без потери данных. Мы ещё вернемся к ним, а пока запомните эти инструкции по изменению моделей:

Внесите изменения в модели (в *models.py*).

Выполните [*python manage.py makemigrations*](http://djbook.ru/rel1.9/ref/django-admin.html#django-admin-makemigrations) чтобы создать миграцию для ваших изменений

Выполните [*python manage.py migrate*](http://djbook.ru/rel1.9/ref/django-admin.html#django-admin-migrate) чтобы применить изменения к базе данных.

Далее будем использовать консоль. Запустим ее с помощью следующей команды:

python manage.py shell

И выполним следующие команды:

Импортируем только что созданную модель классов

**from** djangoapp.models **import** Question, Choice

Удостоверимся в том, что вопросов в базе еще нет

Question**.**objects**.**all()

Создадим новый вопрос

**from** django.utils **import** timezone

q **=** Question(question\_text**=**"What's new?", pub\_date**=**timezone**.**now())

Сохраним изменения в базу данных

q**.**save()

Можем просмотреть ID и атрибуты только что созданного вопроса

q**.**id

q**.**question\_text

q**.**pub\_date

Также можно изменить значение атрибуты, но после этого необходимо вызвать метод save()

q**.**question\_text **=** "What's up?"

q**.**save()

Вызов метода objects.all() отобразит все вопросы в базе

Question**.**objects**.**all()

Отредактируем модель Question (в файле djangoapp/models.py) и добавим метод [\_\_str\_\_()](http://djbook.ru/rel1.9/ref/models/instances.html#django.db.models.Model.__str__)для моделей Question и Choice:

**from** django.db **import** models

**from** django.utils.encoding **import** python\_2\_unicode\_compatible

**class** **Question**(models**.**Model):

*# ...*

**def** **\_\_str\_\_**(self):

**return** self**.**question\_text

**class** **Choice**(models**.**Model):

*# ...*

**def** **\_\_str\_\_**(self):

**return** self**.**choice\_text

Важно добавить метод [\_\_str\_\_()](http://djbook.ru/rel1.9/ref/models/instances.html#django.db.models.Model.__str__) не только для красивого отображения в консоли, но так же и потому, что Django использует строковое представление объекта в интерфейсе администратора.

Заметим, это стандартные методы Python. Давайте добавим свой метод для демонстрации:

djangoapp/models.py

**import datetime**

**from django.db import models**

**from django.utils import timezone**

classQuestion**(models**.**Model):**

**# ...**

defwas\_published\_recently**(self):**

return **self**.**pub\_date** >= **timezone**.**now()** - **datetime**.**timedelta(days**=**1)**

Мы добавили import datetime и from django.utils import timezone для использования стандартной библиотеки Python [datetime](https://docs.python.org/3/library/datetime.html#module-datetime) и модуля Django для работы с временными зонами [django.utils.timezone](http://djbook.ru/rel1.9/ref/utils.html#module-django.utils.timezone) соответственно.

Сохраните эти изменения и запустите консоль Python снова, выполнив python manage.py shell:

**from** djangoapp.models **import** Question, Choice

Убедимся, что метод \_\_str()\_\_ работает корректно

Question**.**objects**.**all()

Джанго предоставляет множество методов для работы с базой

**Question**.**objects**.**filter(id**=**1)**

**Question**.**objects**.**filter(question\_text\_\_startswith**=**'What')**

**from django.utils import timezone**

**current\_year** = **timezone**.**now()**.**year**

**Question**.**objects**.**get(pub\_date\_\_year**=**current\_year)**

Убедимся, что наш метод работает корректно

q **=** Question**.**objects**.**get(pk**=**1)

q**.**was\_published\_recently()

#True

Дадим вопросу несколько вариантов ответа. Сначала

q **=** Question**.**objects**.**get(pk**=**1)

Создадим три варианта ответов

q**.**choice\_set**.**create(choice\_text**=**'Not much', votes**=**0)

<Choice: Not much>

q**.**choice\_set**.**create(choice\_text**=**'The sky', votes**=**0)

<Choice: The sky>

c **=** q**.**choice\_set**.**create(choice\_text**=**'Just hacking again', votes**=**0)

Через объекты вариантов ответов мы можем получить доступ к связанным с ними объектам вопросов

c**.**question

Объекты вопросов так же имеют доступ к объектам вариантов ответов

q**.**choice\_set**.**all()

q**.**choice\_set**.**count()

Choice**.**objects**.**filter(question\_\_pub\_date\_\_year**=**current\_year)

[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]

Давайте удалим один из вариантов. Для этого используем функцию delete()

c **=** q**.**choice\_set**.**filter(choice\_text\_\_startswith**=**'Just hacking')

c**.**delete()

**Добавим представления**

Представление – это “тип” страниц вашего приложения, которое является функцией для обработки запроса и использует шаблон для генерации страницы. В нашем приложении для голосования мы реализуем следующие представления:

* Главная страница вопросов – показывает несколько последних вопросов.
* Страница вопроса – показывает вопрос без результатов, но с формой для ответа.
* Страница результата опроса – показывает результаты опроса.
* Обрабатывает процесс опроса – обрабатывает ответ на вопрос.

В Django страницы и остальной контент отдается представлениями. Представление - это просто функция Python(или метод представления-класса). Django выбирает представление, анализируя запрошенный URL(точнее часть URL-а после домена).

Теперь создадим пару представлений в djangoapp/views.py:

**from django.http import HttpResponse**

**from django.shortcuts import render**

**def index(request):**

**return HttpResponse("Hello, world. You're at the projects index.")**

**def** **detail**(request, question\_id):

**return** HttpResponse("You're looking at question %s." **%** question\_id)

**def** **results**(request, question\_id):

response **=** "You're looking at the results of question %s."

**return** HttpResponse(response **%** question\_id)

**def** **vote**(request, question\_id):

**return** HttpResponse("You're voting on question %s." **%** question\_id)

Создадим файл urls.py и добавим ссылку на polls.urls в главной конфигурации URL-ов. В``mytestproj/urls.py`` добавим импорт django.conf.urls.include, затем [include()](http://djbook.ru/rel1.9/ref/urls.html#django.conf.urls.include) добавим в список urlpatterns. Вы должны получить следующий код:

**from** django.conf.urls **import** include, url

**from** django.contrib **import** admin

urlpatterns **=** [

url(r'^polls/', include('polls.urls')),

url(r'^admin/', admin**.**site**.**urls),

]

Привяжем наше представление новостей в модуле djangoapp.urls добавив вызов [url()](http://djbook.ru/rel1.9/ref/urls.html#django.conf.urls.url) в файле djangoapp/urls.py:

**from** django.conf.urls **import** url

**from** . **import** views

urlpatterns **=** [

*# ex: /djangoapp/*

url(r'^$', views**.**index, name**=**'index'),

*# ex: /djangoapp/5/*

url(r'^(?P<question\_id>[0-9]+)/$', views**.**detail, name**=**'detail'),

*# ex: /djangoapp/5/results/*

url(r'^(?P<question\_id>[0-9]+)/results/$', views**.**results, name**=**'results'),

*# ex: /djangoapp/5/vote/*

url(r'^(?P<question\_id>[0-9]+)/vote/$', views**.**vote, name**=**'vote'),

]

Откройте страницу “/djangoapp/34/”. Будет выполнена функция detail() и показан ID, который вы указали в URL. Откройте “/djangoapp/34/results/” и “/djangoapp/34/vote/” – вы увидите наши будущие страницы результатов и голосования.

X

Каждое представление должно выполнить одно из двух действий: вернуть экземпляр [HttpResponse](http://djbook.ru/rel1.9/ref/request-response.html#django.http.HttpResponse) с содержимым страницы, или вызвать исключения, такое как [Http404](http://djbook.ru/rel1.9/topics/http/views.html#django.http.Http404). Все остальное на ваше усмотрение.

Мы будем использовать API Django для работы с базой данных. Изменим index() так, чтобы оно отображало последние 5 вопросов разделенные запятой от самого нового к самому старому, все остальное оставим без изменений:

**from** django.http **import** HttpResponse

**from** .models **import** Question

**def** **index**(request):

latest\_question\_list **=** Question**.**objects**.**order\_by('-pub\_date')[:5]

output **=** ', '**.**join([q**.**question\_text **for** q **in** latest\_question\_list])

**return** HttpResponse(output)

…

**Используем шаблоны**

Внешний вид страницы определяется в представлении. Если вы захотите изменить дизайн страницы, вам придется менять код. Давайте воспользуемся системой шаблонов Django, чтобы отделить представление от кода.

Для начала создайте каталог templates в каталоге приложения djangoapp. Django будет искать шаблоны в этом каталоге.

Настройка [TEMPLATES](http://djbook.ru/rel1.9/ref/settings.html#std:setting-TEMPLATES) указывает Django как загружать и выполнять шаблоны. По умолчанию используется бэкенд DjangoTemplates, с опцией [APP\_DIRS](http://djbook.ru/rel1.9/ref/settings.html#std:setting-TEMPLATES-APP_DIRS) равной True. В этом случае DjangoTemplates проверяет подкаталог “templates” в приложениях, указанных в [INSTALLED\_APPS](http://djbook.ru/rel1.9/ref/settings.html#std:setting-INSTALLED_APPS).

В только что созданном каталоге templates, создайте каталог djangoapp, и в нем создайте файлindex.html. То есть создайте файл djangoapp/templates/djangoapp/index.html.

Добавьте следующий код в шаблон:

{%[if](http://djbook.ru/rel1.9/ref/templates/builtins.html#if) **latest\_question\_list** %}

**<ul>**

{%[for](http://djbook.ru/rel1.9/ref/templates/builtins.html#for) **question in latest\_question\_list** %}

**<li><a href="/djangoapp/**{{ **question.id** }}**/">**{{ **question.question\_text** }}**</a></li>**

{% **endfor** %}

**</ul>**

{% **else** %}

**<p>No djangoapp are available.</p>**

{% **endif** %}

Теперь изменим наше представление index в djangoapp/views.py, чтобы использовать шаблон:

1

**from** django.http **import** HttpResponse

**from** django.template **import** RequestContext, loader

**from** .models **import** Question

**def** **index**(request):

latest\_question\_list **=** Question**.**objects**.**order\_by('-pub\_date')[:5]

template **=** loader**.**get\_template('djangoapp/index.html')

context **=** RequestContext(request, {

'latest\_question\_list': latest\_question\_list,

})

**return** HttpResponse(template**.**render(context))

Этот код загружает шаблон djangoapp/index.html и передает ему контекст. Контекст - это словарь, содержащий название переменных шаблона и соответствующие им значения.

Загрузите страницу в браузере по адресу “/djangoapp/”, вы должны увидеть список с опросом “What’s up”. Ссылка ведет на страницу опроса.

Процесс загрузки шаблона, добавления контекста и возврат объекта [HttpResponse](http://djbook.ru/rel1.9/ref/request-response.html#django.http.HttpResponse), вполне тривиальный. Django предоставляет функцию render() для всех этих операций. Вот как будет выглядеть наш index() в djangoapp/views.py:

**from** django.shortcuts **import** render

**from** .models **import** Question

**def** **index**(request):

latest\_question\_list **=** Question**.**objects**.**order\_by('-pub\_date')[:5]

context **=** {'latest\_question\_list': latest\_question\_list}

**return** render(request, 'djangoapp/index.html', context)

Так как мы используем такой подход во всех наших представлениях, нет необходимости импортировать [loader](http://djbook.ru/rel1.9/topics/templates.html#module-django.template.loader), [RequestContext](http://djbook.ru/rel1.9/ref/templates/api.html#django.template.RequestContext) и [HttpResponse](http://djbook.ru/rel1.9/ref/request-response.html#django.http.HttpResponse) ([HttpResponse](http://djbook.ru/rel1.9/ref/request-response.html#django.http.HttpResponse) еще нужен, если остались старые detail, results и vote).

Функция [render()](http://djbook.ru/rel1.9/topics/http/shortcuts.html#django.shortcuts.render) первым аргументом принимает объект запроса, также название шаблона и необязательный словарь значений контекста. Возвращает объект [HttpResponse](http://djbook.ru/rel1.9/ref/request-response.html#django.http.HttpResponse) содержащий выполненный шаблон с указанным контекстом.

Теперь создадим страницу опроса, которая отображает вопрос и варианты ответа. Вот так будет выглядеть наше представление (djangoapp/views.py):

**from** django.shortcuts **import** get\_object\_or\_404,render

**from** .models **import** Question

*# ...*

**def** **detail**(request, question\_id):

question **=** get\_object\_or\_404(Question, pk**=**question\_id)

**return** render(request, 'djangoapp/detail.html', {'question': question})

Вызов [get()](http://djbook.ru/rel1.9/ref/models/querysets.html#django.db.models.query.QuerySet.get) и [Http404](http://djbook.ru/rel1.9/topics/http/views.html#django.http.Http404) при отсутствии объекта – обыденные операции. Django предоставляет функцию, которая выполняет эти действия. Функция [get\_object\_or\_404()](http://djbook.ru/rel1.9/topics/http/shortcuts.html#django.shortcuts.get_object_or_404) первым аргументом принимает Django модель и произвольное количество именованных аргументов, которые передаются в метод [get()](http://djbook.ru/rel1.9/ref/models/querysets.html#django.db.models.query.QuerySet.get) менеджера модели. Если объект не найден, вызывается исключение [Http404](http://djbook.ru/rel1.9/topics/http/views.html#django.http.Http404).

Существует также функция [get\_list\_or\_404()](http://djbook.ru/rel1.9/topics/http/shortcuts.html#django.shortcuts.get_list_or_404), которая работает аналогично [get\_object\_or\_404()](http://djbook.ru/rel1.9/topics/http/shortcuts.html#django.shortcuts.get_object_or_404), но использует [filter()](http://djbook.ru/rel1.9/ref/models/querysets.html#django.db.models.query.QuerySet.filter) вместо [get()](http://djbook.ru/rel1.9/ref/models/querysets.html#django.db.models.query.QuerySet.get). Вызывает [Http404](http://djbook.ru/rel1.9/topics/http/views.html#django.http.Http404), если получен пустой список.

Вернемся к представлению detail(). Вот как может выглядеть наш шаблон djangoapp/detail.html, использующий контекстную переменную question:

<h1>**{{** question.question\_text **}}**</h1>

<ul>

**{%** [**for**](http://djbook.ru/rel1.9/ref/templates/builtins.html#for) choice **in** question.choice\_set.all **%}**

<li>**{{** choice.choice\_text **}}**</li>

**{%** **endfor** **%}**

</ul>

Система шаблонов использует точку для доступа к атрибутам переменной. Например, для {{ question.question\_text }} Django сначала пытается обратиться к question как к словарю. При неудаче ищется атрибут переменной, в данном случае он и используется. Если атрибут не найден, будет искаться индекс в списке.

В теге [{% for %}](http://djbook.ru/rel1.9/ref/templates/builtins.html#std:templatetag-for) выполняется вызов метода: question.choice\_set.all, интерпретируется как код Python question.choice\_set.all(), который возвращает итератор по Choice для использования в теге [{% for %}](http://djbook.ru/rel1.9/ref/templates/builtins.html#std:templatetag-for).

## Избавляемся от “хардкода” URL-ов в шаблонах

Когда мы указывали ссылку в шаблоне djangoapp/index.html, она была прописана прямо в коде:

<li><a href="/djangoapp/**{{** question.id **}}**/">**{{** question.question\_text **}}**</a></li>

Проблема в том, что нам будет очень сложно поменять URL-ы в проекте с большим количеством шаблонов. Однако, так как мы указали названия при вызове [url()](http://djbook.ru/rel1.9/ref/urls.html#django.conf.urls.url) в модуле djangoapp.urls, мы можем ссылаться на шаблоны URL-ов используя шаблонный тег {% url %}:

<li><a href="**{%** [**url**](http://djbook.ru/rel1.9/ref/templates/builtins.html#url) 'detail' question.id **%}**">**{{** question.question\_text **}}**</a></li>

Определение URL-а будет найдено в модуле djangoapp.urls. Вот где определен наш URL с названием ‘detail’:

**...**

*# the 'name' value as called by the {% url %} template tag*

url(r'^(?P<question\_id>[0-9]+)/$', views**.**detail, name**=**'detail'),

**...**

Теперь, если вы захотите поменять URL, например на djangoapp/specifics/12/, вам не придется менять все шаблоны, вы можете сделать это в djangoapp/urls.py:

**...**

url(r'^specifics/(?P<question\_id>[0-9]+)/$', views**.**detail, name**=**'detail'),

**...**

## Пространства имен в названиях URL-ов

Наш проект содержит только одно приложение djangoapp. В реальных проектах может быть 5, 10, 20 и больше приложений. Как же Django понимает где чей URL по его названию? Например, приложение djangoapp содержит представление detail, аналогичное представление может быть и в приложении для блогов. Как же Django понимает для какого представления создается URL при использовании тега {% url %}?

Для этого используются пространства имен в URLconf. Изменим djangoapp/urls.py и добавим пространство имен в app\_name:

**from** django.conf.urls **import** url

app\_name **=** 'djangoapp'

urlpatterns **=** [

url(r'^$', views**.**index, name**=**'index'),

url(r'^(?P<question\_id>[0-9]+)/$', views**.**detail, name**=**'detail'),

url(r'^(?P<question\_id>[0-9]+)/results/$', views**.**results, name**=**'results'),

url(r'^(?P<question\_id>[0-9]+)/vote/$', views**.**vote, name**=**'vote'),

]

Теперь поменяем в шаблоне djangoapp/index.html:

<li><a href="**{%** [**url**](http://djbook.ru/rel1.9/ref/templates/builtins.html#url) 'detail' question.id **%}**">**{{** question.question\_text **}}**</a></li>

чтобы использовать пространство имен URL-ов:

<li><a href="**{%** [**url**](http://djbook.ru/rel1.9/ref/templates/builtins.html#url) 'djangoapp:detail' question.id **%}**">**{{** question.question\_text **}}**</a></li>