

# Создание Web-сервисов на Python

Московский физико-технический институт и Mail.Ru Group

Неделя 5

# Содержание

<b>1</b>	<b>Отправка данных из браузера</b>	<b>2</b>
1.1	Работа с пользовательскими данными . . . . .	2
1.2	HTML-формы, элементы ввода . . . . .	3
1.3	Клиентская валидация данных . . . . .	12
<b>2</b>	<b>Обработка данных на сервере</b>	<b>14</b>
2.1	Прием данных в Django . . . . .	14
2.2	Валидация данных . . . . .	16
2.3	Использование форм в Django . . . . .	19
2.4	Использование сторонних валидаторов (jsonschema) . . . . .	21
2.5	Использование сторонних валидаторов (marshmallow) . . . . .	25
<b>3</b>	<b>Аутентификация, авторизация, сессии</b>	<b>28</b>
3.1	Аутентификация и авторизация . . . . .	28
3.2	Аутентификация пользователей в Django . . . . .	29
3.3	Улучшаем проект . . . . .	32
3.4	Авторизация в Django . . . . .	34

# 1. Отправка данных из браузера

## 1.1. Работа с пользовательскими данными

Пользовательские данные - это наиболее ценный ресурс вашего веб-приложения.

Является ли ключевое слово поиска пользовательскими данными? Да, является. Вы получаете их от пользователя, соответственно, они являются пользовательскими данными. Является фамилия, имя пользователя пользовательскими данными? Да, тоже являются.

Итак, каким образом мы хотим получить, допустим, ключевое слово поиска? Обычно для таких целей используется Query string. Query string - это неиерархическая часть URL. URL, как вы должны помнить из предыдущих частей курса, - это uniform resource locator.

Query string - это набор параметров в виде ключ-значение. Вы можете передавать в query string ASCII-совместимые символы, либо кодированный юникод. Query string очень часто применяется именно для выборки неких объектов.

Второй популярный способ - это передача пользовательских данных через request body. Обычно через request body передаются более структурированные данные, это могут быть различные большие документы или же это могут быть файлы, или же это может быть бинарная информация. Также вы можете передавать multipart forma, то есть вы можете передавать файлы одновременно с текстом, одновременно с JSON-файлами и так далее. Вы можете выбрать любой другой источник данных, которые есть в main type стандарта HTTP.

Очень важно помнить о методах запросов HTTP. Мы должны помнить о свойстве их демпотентности. Этот термин пришел из математики. Он означает примерно следующее: если метод неидемпотентен, то последующие его вызовы влияют на состояние систем.

Например, метод get для получения информации. Допустим, у нас есть точка, где мы выдаем данные пользователя, и мы запрашиваем эту точку. Нам вернутся данные пользователей. Если мы запросим эту точку еще раз, то вернутся те же самые данные пользователя, при условии что они не были изменены на сервере. То есть каждый последующий вызов этого метода приводит ровно к тем же самым результатам, система не изменяется.

С другой стороны, есть метод post, которым обычно создаются новые ресурсы, объекта и так далее. Если мы вызовем метод post один раз, то произойдет создание объекта. Если мы вызовем его ещё один раз, произойдет ещё создание объекта. Таким образом, каждый последующий вызов будет создавать новый объект. Таким образом мы можем говорить, что метод post неидемпотентен, а метод get идемпотентен.

Методы delete. Метод delete является так же идемпотентным, метод put также является идемпотентным. При каждом последующем запросе на метод put объект будет заменяться другим объектом. Если мы повторим это несколько раз, то эффект будет тот же самым.

Веб-серверы, которые работают с запросами от пользователей, очень часто по умолчанию настроены на то, чтобы кешировать некоторые виды запросов, наиболее часто это запросы вида get, потому что этот запрос гарантировано идемпотентен, и его очень удобно кешировать, то есть не выдавать каждый раз одну и ту же информацию о пользователе, таким образом нагружая базу данных веб-приложения, а сохранить где-то уже сформированный документ и выдавать каждый раз его, это быстрее и эффективнее.

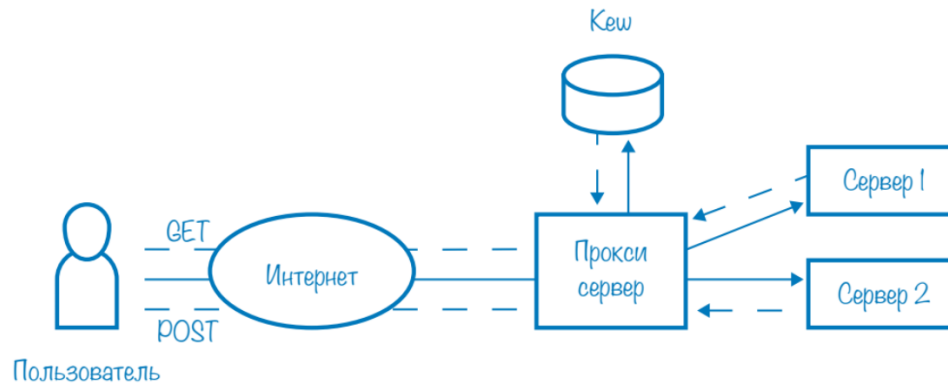


Рис. 1

Также стоит вспомнить про методологию REST - representational State Transfer. Это набор правил, благодаря которому вы можете построить очень удобный и легко обслуживаемый API. Мы остановимся только на основных ключевых понятиях. Каждый ресурс вашего веб-приложения является либо конечным ресурсом, либо коллекцией.

Например, `.api/v1/users/` - это коллекция объектов, там храним пользователей и можем создавать новых. Вот пример запроса на создание пользователя:

`/api/v1/users/` (POST)

Как видите, он делается в коллекцию, нам возвращается репрезентация созданного пользователя, и мы уже можем с ней работать. Либо мы можем запросить конкретный объект, который идентифицируется каким-то конкретным идентификатором.

Почему стоит строить свое API именно так? Потому что разработчик, который будет работать с вашими API, а с той стороны обычно бывают не только пользователи, а еще и другие разработчики, им удобнее будет понимать именно такую схему, потому что она достаточно распространена.

## 1.2. HTML-формы, элементы ввода

Создадим с вами `venv`, делается это достаточно стандартным способом. Итак, создаем `venv`; `venv`, как видим, создан. Теперь мы этот `venv` активируем.

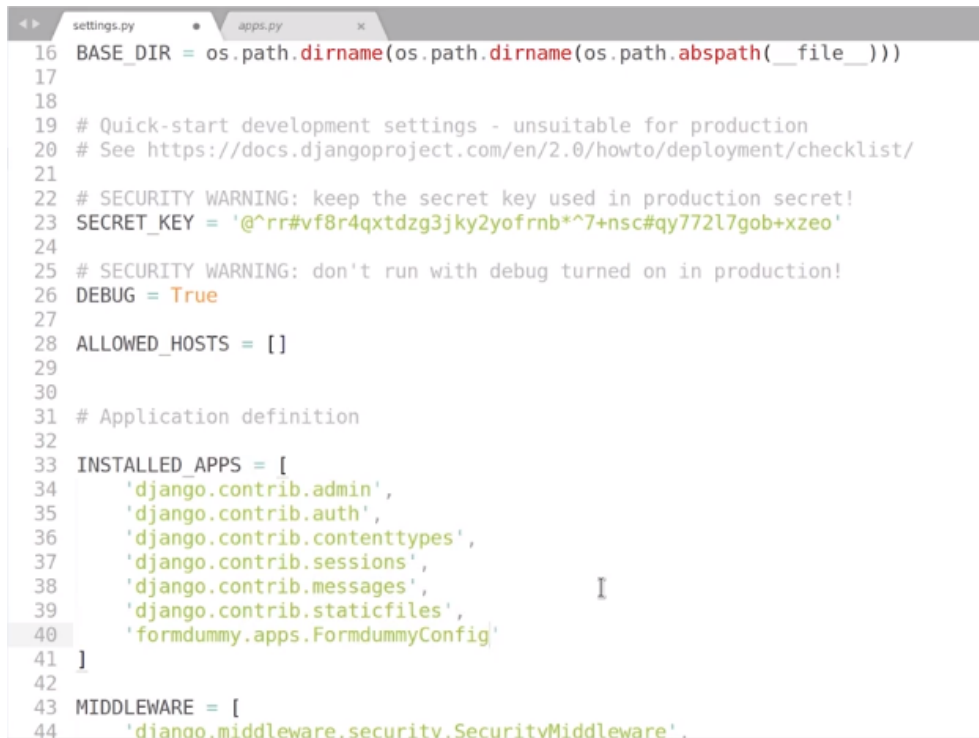
Устанавливаем Django без указания версии, поэтому будет установлена на данный момент последняя версия Django 2.0. Так, всё установилось. Теперь мы создаем проект Django. Делается это через команду `django admin, как вы помните. startproject coursera. Создаем coursera_forms. Теперь переходим внутрь проекта и создаем приложение внутри этого проекта. В отличие от создания проекта, создание приложения нужно выполнять через команду manage.py.`

```
droppoint@Partlabs-Mail:~/coursera$ python3 -m venv ~/venv/coursera
droppoint@Partlabs-Mail:~/coursera$ source ~/venv/coursera/bin/activate
(coursera) droppoint@Partlabs-Mail:~/coursera$ pip install django
Collecting django
  Using cached Django-2.0-py3-none-any.whl
Collecting pytz (from django)
  Using cached pytz-2017.3-py2.py3-none-any.whl
Installing collected packages: pytz, django
Successfully installed django-2.0 pytz-2017.3
(coursera) droppoint@Partlabs-Mail:~/coursera$ django-admin startproject coursera_forms
(coursera) droppoint@Partlabs-Mail:~/coursera$ ls
coursera_forms
(coursera) droppoint@Partlabs-Mail:~/coursera$ cd coursera_forms/
(coursera) droppoint@Partlabs-Mail:~/coursera/coursera_forms$ python ./manage.py startapp formdummy
my
(coursera) droppoint@Partlabs-Mail:~/coursera/coursera_forms$ ls
coursera_forms  formdummy  manage.py
(coursera) droppoint@Partlabs-Mail:~/coursera/coursera_forms$ █
```

Рис. 2

Назовем наш проект formDummy, просто так. И переходим в текстовый редактор. Некоторые из файлов, которые создались при команде startapp, вам не понадобятся, но удалять мы их не будем.

Итак, давайте для начала созданное наше приложение подключим к проекту. Для этого в настройках проекта, в файле settings.py, нужно добавить указания на config приложения, которое мы создали, то есть на вот этот класс. Таким образом мы подключили приложение.

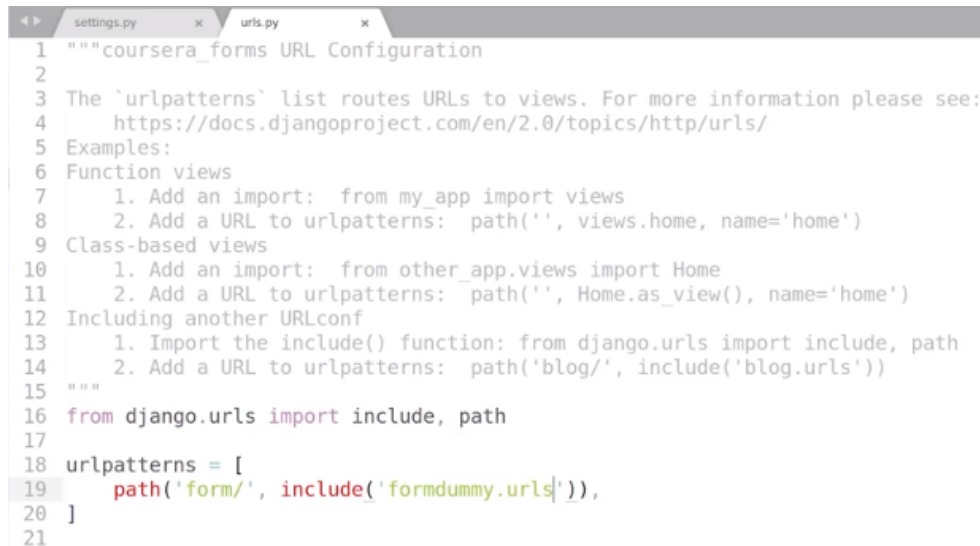


```
16 BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
17
18
19 # Quick-start development settings - unsuitable for production
20 # See https://docs.djangoproject.com/en/2.0/howto/deployment/checklist/
21
22 # SECURITY WARNING: keep the secret key used in production secret!
23 SECRET_KEY = '@^rr#vf8r4qxtdzg3jky2yofrnbnb*^7+nsc#qy772l7gob+xzeo'
24
25 # SECURITY WARNING: don't run with debug turned on in production!
26 DEBUG = True
27
28 ALLOWED_HOSTS = []
29
30
31 # Application definition
32
33 INSTALLED_APPS = [
34     'django.contrib.admin',
35     'django.contrib.auth',
36     'django.contrib.contenttypes',
37     'django.contrib.sessions',
38     'django.contrib.messages',
39     'django.contrib.staticfiles',
40     'formdummy.apps.FormdummyConfig',
41 ]
42
43 MIDDLEWARE = [
44     'django.middleware.security.SecurityMiddleware',
```

Рис. 3

Теперь давайте пробросим туда наши URL паттерны для того, чтобы мы могли зайти во View этого приложения и что-то сделать. Зайдем в urls.py нашего проекта, уберем оттуда подключения админской панели, потому что она в данный момент нам не понадобится. Нужно выбрать

префикс для нашего приложения formDummy, давайте префикс у нас будет form. Префикс — это часть URL перед теми URL, которые поддерживают приложение.



```
1 """coursera_forms URL Configuration
2
3 The 'urlpatterns' list routes URLs to views. For more information please see:
4     https://docs.djangoproject.com/en/2.0/topics/http/urls/
5 Examples:
6 Function views
7     1. Add an import: from my_app import views
8     2. Add a URL to urlpatterns: path('', views.home, name='home')
9 Class-based views
10    1. Add an import: from other_app.views import Home
11    2. Add a URL to urlpatterns: path('', Home.as_view(), name='home')
12 Including another URLconf
13    1. Import the include() function: from django.urls import include, path
14    2. Add a URL to urlpatterns: path('blog/', include('blog.urls'))
15 """
16 from django.urls import include, path
17
18 urlpatterns = [
19     path('form/', include('formdummy.urls')),
20 ]
21
```

Рис. 4

URL приложение подключается через метод include. Подключаем formdummy.url, которого у нас не существует пока, но сейчас мы его создадим. Создаем новый файл, сохраняем его в приложении formdummy, называем его urls.py. Для того чтобы не печатать всё заново, копируем из основного urls.py. Include нам здесь не понадобится, поэтому мы его убираем. И подключаем наше View, которого тоже пока еще не существует. from import views. Называем наше View FormDummyView, чтобы быть оригинальным. Так как впоследствии мы будем делать View на базе class based View, нужно в конце указать вызов функции as\_view.

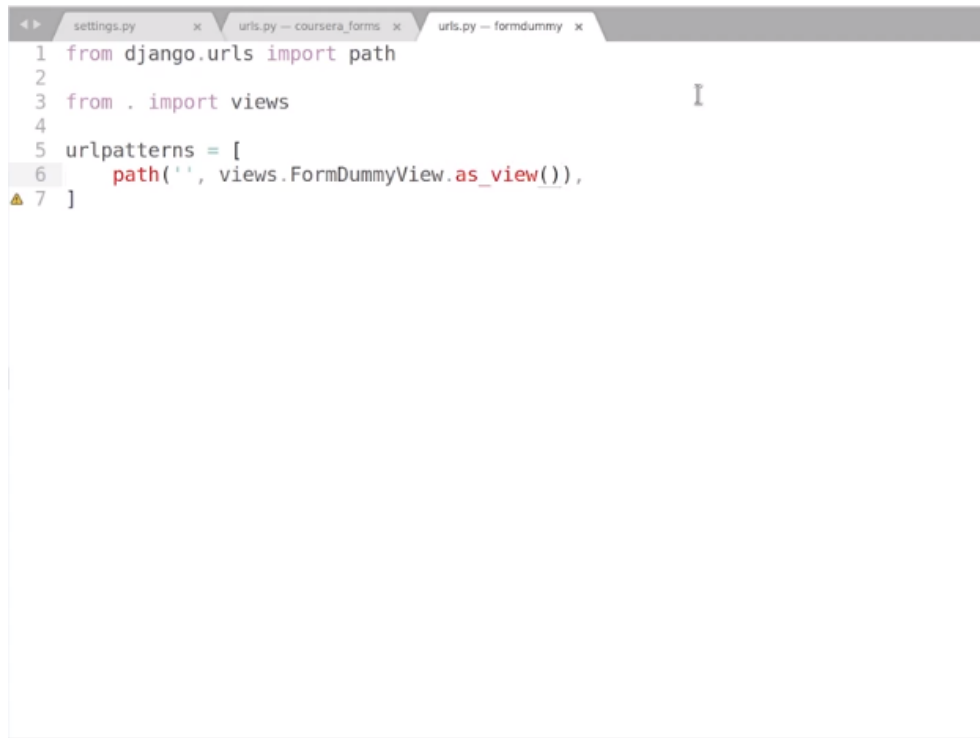
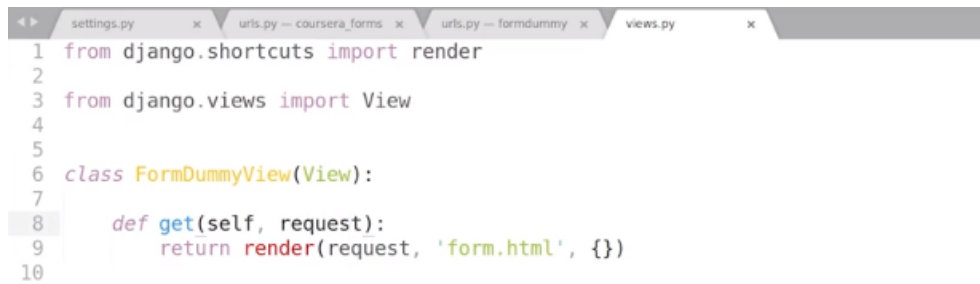


Рис. 5

Теперь нам нужно создать наш View. Для этого мы импортируем класс View. Как помним, мы должны назвать наш View FormDummyView. Определим метод get, потому что мы будем делать запросы из браузера методом get.

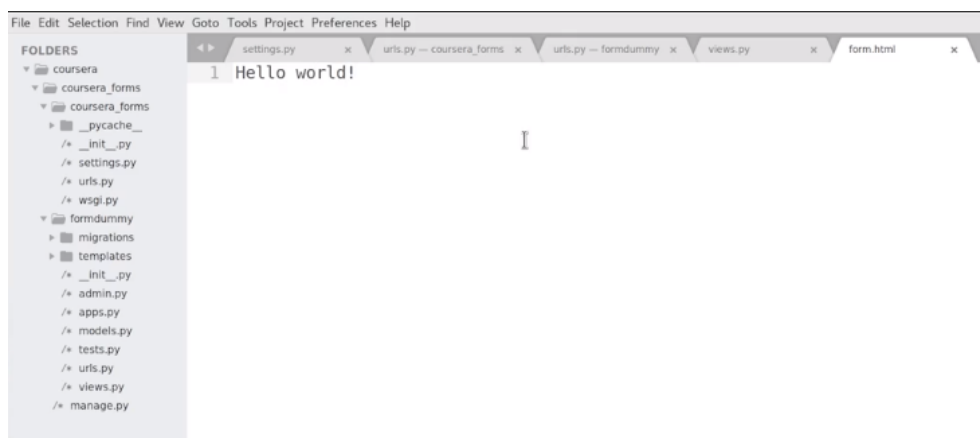
Как мы помним, первым аргументом должен быть self в классе. Вторым аргументом у View является request объект. Мы его здесь тоже указываем. Render — это функция, которая позволяет вам **отрендерить** шаблон. Первым аргументом в render передается request, вторым — указание на имя шаблона. В нашем случае это будет form.html. Третьим аргументом передается context. Context — это словарь, в котором мы передаем все необходимые для рендера шаблона переменные.



```
1 from django.shortcuts import render
2
3 from django.views import View
4
5
6 class FormDummyView(View):
7
8     def get(self, request):
9         return render(request, 'form.html', {})
10
```

Рис. 6

Итак, теперь нам нужно создать наш шаблон. Перед этим надо сохранить то, что мы уже сделали. Шаблоны все помещаются обычно в одну папку в рамках приложения либо в рамках проекта. Мы поместим в папку в рамках приложения. Назовем ее `templates`, как она обычно называется. И в этой папке создадим наш `form.html`. Итак, в нашем шаблоне мы напишем традиционный Hello world, просто чтобы проверить, что всё работает правильно.



```
1 Hello world!
```

Рис. 7

Давайте попробуем запустить наш сервер. Делается это достаточно стандартной командой, `runserver`. Как мы помним, DEF server запускается на `localhost` на порту 8000. Берем любой удобный для нас браузер и открываем `localhost` порт 8000. Все отображается.



Попробуем создать простую форму. Форма создается через тег `form`. Создадим тег `form` и сразу же его закроем. Если мы сейчас отрендерим наш HTML-файл, то ничего не увидим, потому что у нас нет внутри формы никаких инпутов, то есть элементов, благодаря которым мы можем получить данные от пользователя.



```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <title>Ter form</title>
6   </head>
7   <body>
8     <h1>Форма отзыва</h1>
9     <form>
10    </form>
11  </body>
12 </html>
```

Рис. 8

Но перед тем как создать инпуты, давайте разберемся до конца с формой. В форме мне нужно указать несколько параметров.

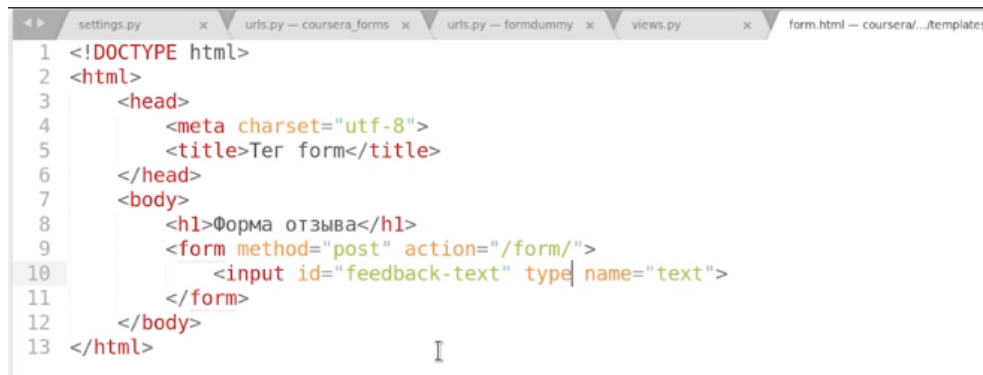
В частности, это метод. Указываем, каким методом мы будем отправлять наши данные из формы. Также нужно указать, куда мы будем отправлять наши данные. В данном случае мы будем отправлять их по тому же ресурсу `form`. А так как мы отправляем их по тому же ресурсу, этот параметр можно и опустить.



```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <title>Ter form</title>
6   </head>
7   <body>
8     <h1>Форма отзыва</h1>
9     <form method="post" action="/form/">
10    |
11  </form>
12 </body>
13 </html>
```

Рис. 9

Теперь мы создадим инпуты. Давайте создадим `input` для ввода нашего отзыва. Будет называться `text`. `id` элементу мы присвоим `feedback-text`. Также мы укажем `name` — это параметр, который указывает, под каким именем поле будет отправлено на сервер. Назовем его просто `text`. И укажем параметр `type`, который определяет тип инпута.

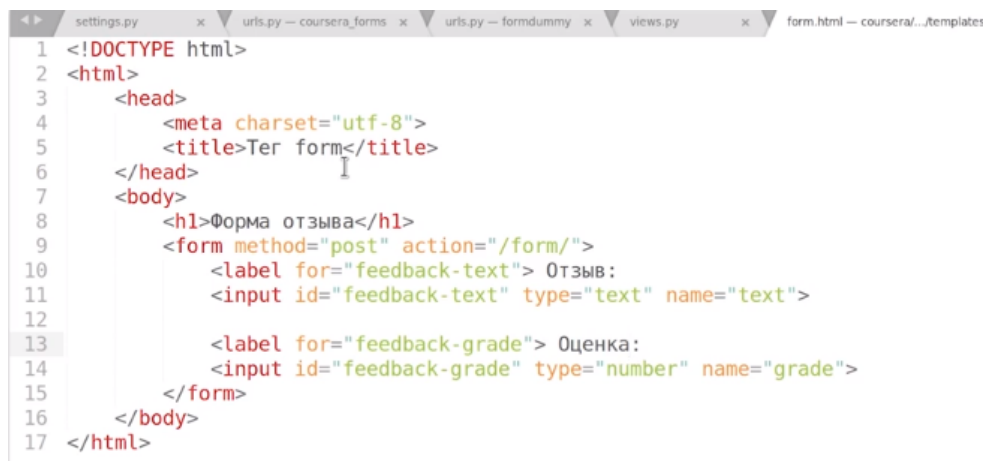


```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <title>Ter form</title>
6   </head>
7   <body>
8     <h1>Форма отзыва</h1>
9     <form method="post" action="/form/">
10      <input id="feedback-text" type="text" name="text">
11    </form>
12  </body>
13 </html>
```

Рис. 10

В качестве типа инпута может выступать текст, может выступать число, может выступать дата, цвет, дата и время, и прочее.

Итак, мы создали `input feedback-text`. Если мы отрендерим форму, можно увидеть, что у нас есть `input`, но не написано, что конкретно это такое. Для того чтобы подписать его, существует тег `label`. `label` нужно указать в параметре `for` `id` инпута, для которого он существует. Пишем название Отзыв.



```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <title>Ter form</title>
6   </head>
7   <body>
8     <h1>Форма отзыва</h1>
9     <form method="post" action="/form/">
10      <label for="feedback-text"> Отзыв:
11      <input id="feedback-text" type="text" name="text">
12    </form>
13    <label for="feedback-grade"> Оценка:
14    <input id="feedback-grade" type="number" name="grade">
15  </form>
16 </body>
17 </html>
```

Рис. 11

Итак, помимо того, что мы хотим получать отзыв от пользователя, мы хотим еще получать оценку в виде количества звезд. Для этого мы создадим `input` с типом `number`, назовем его `feedback-grade`. Подпишем поле как Оценка.

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <title>Ter form</title>
6   </head>
7   <body>
8     <h1>Форма отзыва</h1>
9     <form method="post" action="/form/">
10      <label for="feedback-text"> Отзыв:
11      <input id="feedback-text" type="text" name="text">
12
13      <label for="feedback-grade"> Оценка:
14      <input id="feedback-grade" type="number" name="grade">
15
16      <label for="feedback-image"> Фотография:
17      <input id="feedback-image" type="file" name="image">
18    </form>
19  </body>
20 </html>

```

Рис. 12

Создадим еще input для загрузки файлов. Мы хотим загружать к нашим отзывам еще и фотографии. Назовем его feedback-image. Type у такого input должен быть file.

Мы можем заполнить данными всю нашу форму, но непонятно, как ее отправить. Для того чтобы отправить форму, есть специальный input, он называется submit. И name ему не обязательно.

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <title>Ter form</title>
6   </head>
7   <body>
8     <h1>Форма отзыва</h1>
9     <form method="post" action="/form/">
10      <label for="feedback-text"> Отзыв:
11      <input id="feedback-text" type="text" name="text">
12
13      <label for="feedback-grade"> Оценка:
14      <input id="feedback-grade" type="number" name="grade">
15
16      <label for="feedback-image"> Фотография:
17      <input id="feedback-image" type="file" name="image">
18
19      <input id="feedback-submit" type="file" name="image">
20    </form>
21  </body>
22 </html>

```

Рис. 13

Для того чтобы отправлять файлы из этой формы, нужно сделать еще одну небольшую хитрость, нужно указать параметр enctype. Enctype — это указание на формат кодирования информации отправляемой формы.

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <title>Ter form</title>
6   </head>
7   <body>
8     <h1>Форма отзыва</h1>
9     <form method="post" enctype="multipart/form-data" action="/form/">
10       <label for="feedback-text"> Отзыв:
11       <input id="feedback-text" type="text" name="text">
12
13       <label for="feedback-grade"> Оценка:
14       <input id="feedback-grade" type="number" name="grade">
15
16       <label for="feedback-image"> Фотография:
17       <input id="feedback-image" type="file" name="image">
18
19       <input id="feedback-submit" type="submit">
20     </form>
21   </body>
22 </html>

```

Рис. 14

Проверяем и видим, что есть 403-я ошибка от Django. Это ошибка CSRF verification failed. Для того чтобы обойти эту ошибку, нам нужно сделать следующее. Почему эта ошибка возникает и что это такое, мы разберем чуть позже.

```

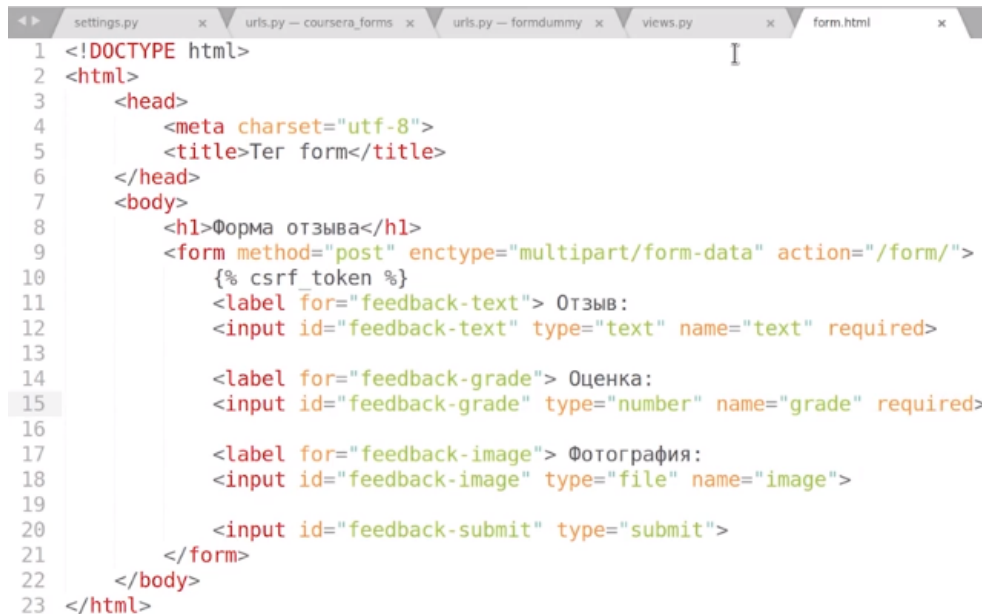
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <title>Ter form</title>
6   </head>
7   <body>
8     <h1>Форма отзыва</h1>
9     <form method="post" enctype="multipart/form-data" action="/form/">
10       {% csrf_token %}
11       <label for="feedback-text"> Отзыв:
12       <input id="feedback-text" type="text" name="text">
13
14       <label for="feedback-grade"> Оценка:
15       <input id="feedback-grade" type="number" name="grade">
16
17       <label for="feedback-image"> Фотография:
18       <input id="feedback-image" type="file" name="image">
19
20       <input id="feedback-submit" type="submit">
21     </form>
22   </body>
23 </html>

```

Рис. 15

## 1.3. Клиентская валидация данных

Мы с вами создали форму. Эта форма принимает какие-то данные, но она ещё далеко не готова. Например, некоторые поля должны быть обязательными. Для того чтобы сделать поле обязательным, нужно в input ввести параметр required.

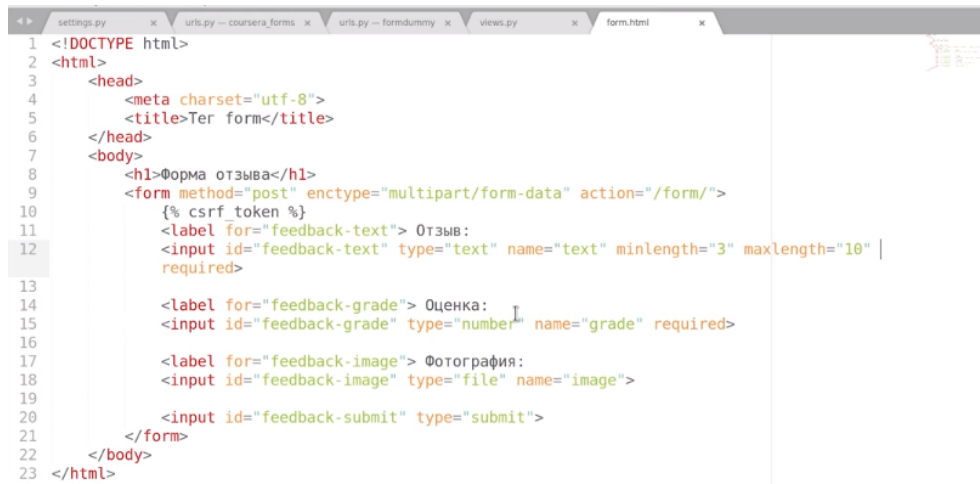


```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <title>Ter form</title>
6   </head>
7   <body>
8     <h1>Форма отзыва</h1>
9     <form method="post" enctype="multipart/form-data" action="/form/">
10       {% csrf_token %}
11       <label for="feedback-text"> Отзыв:
12       <input id="feedback-text" type="text" name="text" required>
13
14       <label for="feedback-grade"> Оценка:
15       <input id="feedback-grade" type="number" name="grade" required>
16
17       <label for="feedback-image"> Фотография:
18       <input id="feedback-image" type="file" name="image">
19
20       <input id="feedback-submit" type="submit">
21     </form>
22   </body>
23 </html>
```

Рис. 16

Браузер выделяет эти поля, которые мы не заполнили, красным цветом и просит их заполнить. У нас могут быть какие-то ограничения по длине отзыва или же по оценке. Допустим, длина отзыва не более 2000 знаков, или длина отзыва не меньше 100 знаков.

Для того чтобы ограничить эти величины, мы можем использовать параметр min и max length. Вводим параметр min length. Устанавливаем его на значении 3. Отзыв не может быть короче трех символов. И max length, мы хотим получать отзывы не более десяти символов.



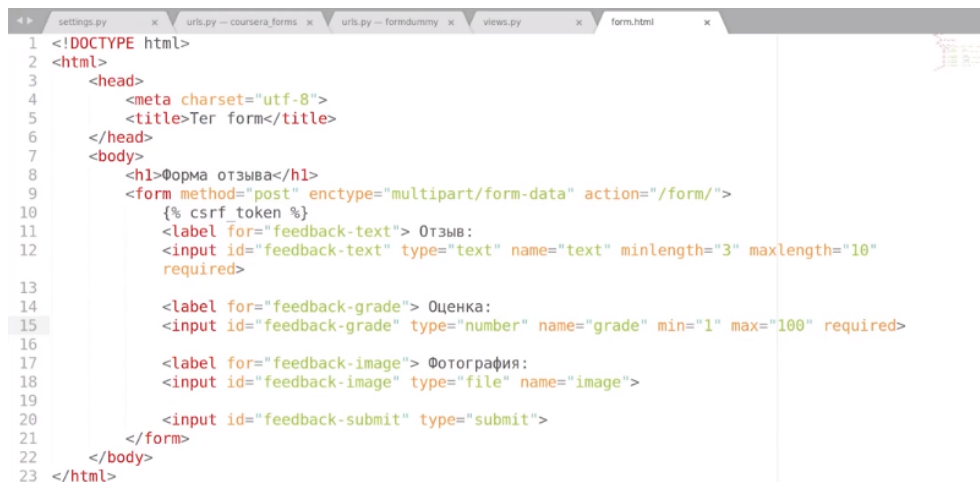
```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <title>Ter form</title>
6   </head>
7   <body>
8     <h1>Форма отзыва</h1>
9     <form method="post" enctype="multipart/form-data" action="/form/">
10      {% csrf.token %}
11      <label for="feedback-text"> Отзыв:
12      <input id="feedback-text" type="text" name="text" minlength="3" maxlength="10"
13      required>
14
15      <label for="feedback-grade"> Оценка:
16      <input id="feedback-grade" type="number" name="grade" required>
17
18      <label for="feedback-image"> Фотография:
19      <input id="feedback-image" type="file" name="image">
20
21      <input id="feedback-submit" type="submit">
22    </form>
23  </body>
24 </html>

```

Рис. 17

Обновляем страницу, пробуем написать отзыв длинее десяти символов. У нас не получается. Итак, помимо отзывов у нас есть ещё и оценка, которая выражается числом. Число ограничивается параметрами min и max. Давайте min установим в единицу, max установим в 100. Ставим нашу оценку в 1 балл. Пробуем ее снизить, и у нас ничего не получается. Пробуем отправить корректный набор параметров, и форма отправляется.



```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <title>Ter form</title>
6   </head>
7   <body>
8     <h1>Форма отзыва</h1>
9     <form method="post" enctype="multipart/form-data" action="/form/">
10      {% csrf.token %}
11      <label for="feedback-text"> Отзыв:
12      <input id="feedback-text" type="text" name="text" minlength="3" maxlength="10"
13      required>
14
15      <label for="feedback-grade"> Оценка:
16      <input id="feedback-grade" type="number" name="grade" min="1" max="100" required>
17
18      <label for="feedback-image"> Фотография:
19      <input id="feedback-image" type="file" name="image">
20
21      <input id="feedback-submit" type="submit">
22    </form>
23  </body>
24 </html>

```

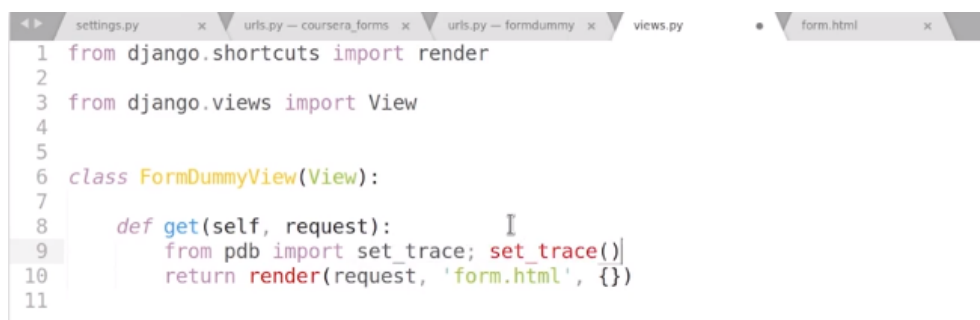
Рис. 18

## 2. Обработка данных на сервере

### 2.1. Прием данных в Django

В django существует объект под названием HttpRequest, в этом объекте передаются все данные, которые отправил пользователь, плюс заголовки, плюс много ещё интересной технической информации.

Давайте попробуем передать какую-то информацию посредством querystring через Query Path параметры и попробуем посмотреть, как она у нас отобразится на стороне django. Для этого мы с вами внедрим точку дебаггинга в методе get view, который он уже создал.



```
1 from django.shortcuts import render
2
3 from django.views import View
4
5
6 class FormDummyView(View):
7
8     def get(self, request):
9         from pdb import set_trace; set_trace()
10        return render(request, 'form.html', {})
11
```

Рис. 19

Давайте сразу загрузим нашу форму с параметром hello. Наша страница перестает загружаться достаточно быстро. Это совершенно нормальная ситуация, когда мы внедрили дебаггер, потому что выполнение request respon cycle'a останавливается именно на точке дебаггинга.

Давайте посмотрим, что у нас представляет собой объект request, а именно что находится у него внутри. Нас интересует то, что было передано в query параметра. Для того, чтобы получить информацию с query параметра, нам достаточно обратиться к полю get. Как мы видим, в querystring был передан параметр hello со значением world.



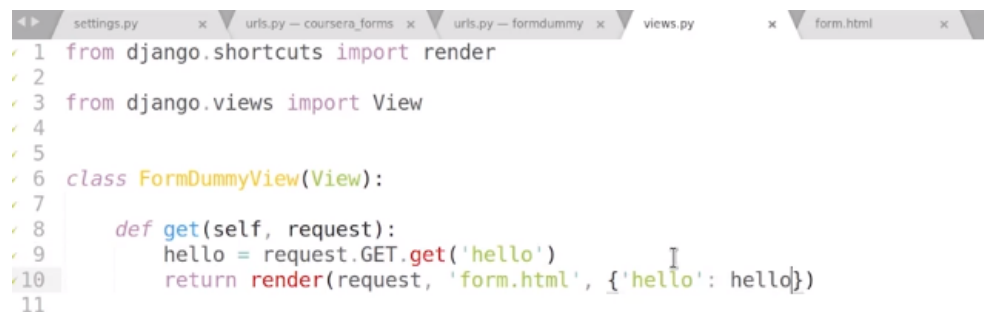
```

'XDG_SESSION_TYPE': 'x11',
'XDG_VTNR': '2',
'XMODIFIERS': '@im=ibus',
'ZEITGEIST_DATA_PATH': '/home/droppoint/.local/share/zeitgeist',
'_': '/home/droppoint/venv/coursera/bin/python',
'wsgi.errors': <_io.TextIOWrapper name='<stderr>' mode='w' encoding='
'wsgi.file_wrapper': <class 'wsgiref.util.FileWrapper'>,
'wsgi.input': <_io.BufferedReader name=6>,
'wsgi.multiprocess': False,
'wsgi.multithread': True,
'wsgi.run_once': False,
'wsgi.url_scheme': 'http',
'wsgi.version': (1, 0)},
'method': 'GET',
'path': '/form/',
'path_info': '/form/',
'resolver_match': ResolverMatch(func=formdummy.views.FormDummyView, args=(), kwargs=None, app_names=[], namespaces=[]),
'session': <django.contrib.sessions.backends.db.SessionStore object at 0x7faf66727c80>,
'user': <SimpleLazyObject: <function AuthenticationMiddleware.process_request:
at 0x7faf66727c80>>}>
(Pdb) request.GET
<QueryDict: {'hello': ['world']}>
(Pdb) █

```

Рис. 20

Таким образом, мы можем получить значения из `querystring`. Но давайте попробуем сделать что-то более интересное, а именно отобразить полученную информацию. Для этого мы получаем параметр из поля `get`, помним, что он называется `hello`, передаем его в контекст, то есть, эту переменную теперь можно использовать в шаблоне. Добавляем ее в шаблон, сразу под форму, чтобы ее было видно. Сохраняем. Проверяем, что у нас сервер перезапустился. Загружаем нашу страницу, и справа от формы у нас отобразился `world`.



```

1 from django.shortcuts import render
2
3 from django.views import View
4
5
6 class FormDummyView(View):
7
8     def get(self, request):
9         hello = request.GET.get('hello')
10        return render(request, 'form.html', {'hello': hello})
11

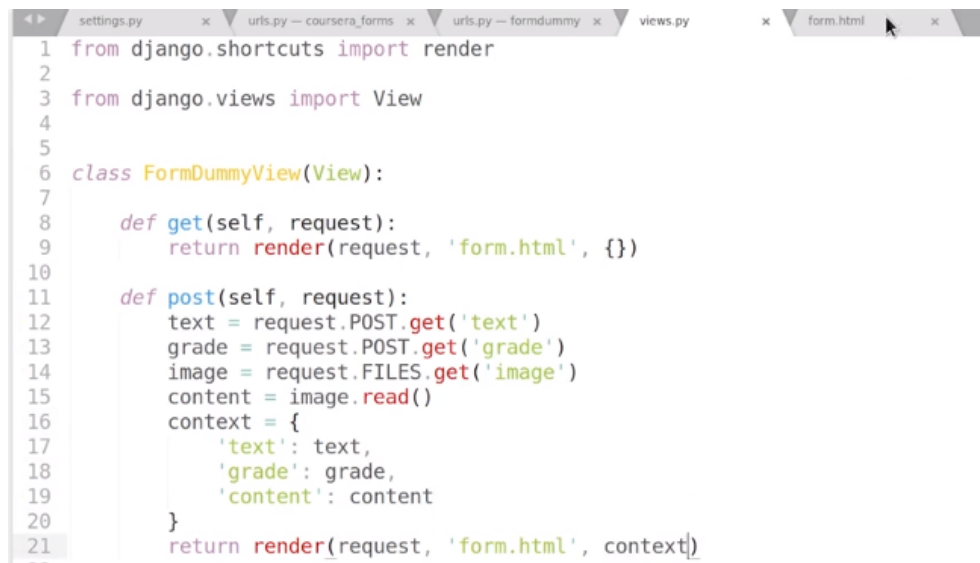
```

Рис. 21

Для того чтобы обработать отправляемую форму, нам необходимо реализовать метод `post` в нашем `view`. Давайте это и сделаем. Пока что мы копируем `render` из метода `get`, хотя по-хорошему после обработки формы осуществляется обычно `redirect`. Итак, мы должны получить из формы текст, оценку и наше изображение. Текст мы получаем из `request`, данные, передаваемые в методе `post`, находятся в поле `post` объекта `request`, мы оттуда их и забираем. Для переменной `grade` мы сделаем ровно то же самое. И передаваемый файл мы должны будем достать из поля `files`. Далее мы должны, так как `request files` — это тоже словарь, передать название поля, которое мы передаем, это `image`. Итак, мы получили три переменные — `text`, `grade` и `image`. Допустим, мы



хотим получить контент из файла `image`, для этого нам нужно сделать `image.read`, потому что в `request.files` располагаются объекты, которые очень похожи на стандартные питоновские файлы. Теперь все эти переменные, которые мы получили, мы засовываем в контекст. Повторяем это для всех остальных переменных. Контекст мы передаем в рендер.



```
1 from django.shortcuts import render
2
3 from django.views import View
4
5
6 class FormDummyView(View):
7
8     def get(self, request):
9         return render(request, 'form.html', {})
10
11     def post(self, request):
12         text = request.POST.get('text')
13         grade = request.POST.get('grade')
14         image = request.FILES.get('image')
15         content = image.read()
16         context = {
17             'text': text,
18             'grade': grade,
19             'content': content
20         }
21         return render(request, 'form.html', context)
```

Рис. 22

## 2.2. Валидация данных

О некоторых параметрах клиент может и не знать, или же вы можете не контролировать клиента вообще. Для этого нужно проводить проверку данных на серверной стороне. Или же данные могут быть заведомо зловредными, то есть некто может пытаться использовать ваш сайт для каких-то своих корыстных целей.

Поэтому всегда на больших проектах используется серверная валидация совместно с клиентской валидацией и никогда не ограничивается только клиентской валидацией. Иногда клиентская валидация не используется вообще — используется только серверная.

Мы должны валидировать то, что данные, приходящие от пользователей, не содержат каких-то инъекций, будь это инъекции кода или SQL инъекция. Также мы должны проверять, что данные пришли к нам из того источника, откуда мы ожидаем. То есть мы должны идентифицировать клиентское приложение. Это делается посредством ключей либо посредством специальных CSRF-токенов, но об этом мы поговорим в других разделах курса. Также мы должны проверять то, что данные, приходящие к нам от клиента, являются валидными для нас данными.

Мы получаем данные о каком-нибудь пользователе, например о каком-нибудь школьнике. Мы хотим получить его фамилию, имя и возраст. Понятно, что имя и фамилия должны быть как-то ограничены в длине. Мы можем прописать ограничение на то, что длина строки не может быть, скажем, свыше 100 символов. Либо мы можем прописать ограничение на возраст. Мы точно знаем, что возраст человека не может быть отрицательной величиной. Либо мы можем прописать ограничение на наличие определенных полей. То есть мы знаем, что у каждого человека

должно быть имя, фамилия. Такого рода ограничения позволяют вашему приложению работать штатно. Вы получаете те данные, которые ожидаете, а те данные, которые вы не ожидаете, на эти данные вы делаете ошибку и отправляете эту ошибку клиенту, чтобы он уже что-то с ней сделал.

Итак, каким образом мы можем провести серверную валидацию в Django? Для этого есть несколько инструментов. Начнем, естественно, с Django Forms, так как этот инструмент встроен непосредственно в Django, он позволяет нам рендерить формы, которые мы уже видели ранее, принимать данные из этих форм, валидировать их по различным параметрам либо с использованием своих каких-то валидаторов и непосредственно передавать эти данные в модель или в объект, или получать из них словарь, в зависимости от того, что необходимо вашему приложению.

```
from django import forms
class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField
        (widget=forms.Textarea)
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False)
```

К плюсам этого инструмента можно отнести то, что не нужно устанавливать никаких дополнительных зависимостей. Но это является и одновременно минусом — Django Forms поддерживается только в Django. Использовать их где-то в другом проекте достаточно сложно или можно сказать, что даже невозможно.

К плюсам можно отнести также, что данные, обработанные Django Forms, достаточно удобно ложатся в модели, и достаточно удобно строить валидацию на базе python-функций или классов. Следующий инструмент, который мы рассмотрим, это jsonschema. Достаточно часто применяется frontend-разработчиками при клиентской валидации перед отправкой данных на сервер. Он также может применяться в виде python-пакета на серверной стороне. То, что вы видите, это jsonschema, это json-документ, который, скажем так, показывает, как валидировать другой json-документ, которым являются пользовательские данные.

```
{
  "title": "Person",
  "type": "object",
  "properties": {
    "rstName": {
      "type": "string"
    },
    "lastName": {
      "type": "string"
    },
    "age": {
      "description": "Age in years",
      "type": "integer",
      "minimum": 0
    }
  },
  "required": ["rstName", "lastName"]
}
```

Здесь мы видим, что мы ожидаем некоторые обязательные поля, они описаны в поле `required`, и по некоторым полям мы валидируем непосредственно значения. Допустим, поле `age` (возраст), как мы уже говорили, не может быть меньше нуля. К плюсам этого инструмента можно отнести то, что он достаточно прост в использовании. Схемы между backend-ом и frontend-ом могут быть одинаковые, вы можете их использовать повторно. К минусам можно отнести то, что затруднено расширение этого инструмента новыми валидаторами и то, что это непосредственно чистый валидатор. То есть вы не можете на базе `jsonschema` построить формы в HTML, вы не можете прокинуть данные сразу непосредственно в модель. Только валидация только словарных объектов.

Следующий инструмент, который мы рассмотрим, это `marshmallow`. Это простые python-классы, в которых мы описываем поля, и непосредственно в полях мы описываем валидаторы, которыми их следует провалидировать.

```
class AlbumSchema(Schema):
    title = elds.Str()
    release_date = elds.Date()
    artist = elds.Nested(ArtistSchema())
```

Также очень часто используются, как и `jsonschema`, при построении API. К плюсам этого инструмента можно отнести то, что он агностичен по отношению к используемым фреймворкам, как и `jsonschema`. К минусам можно отнести то, что это дополнительная зависимость. И к особенностям можно отнести то, что как и `jsonform`, он написан целиком на Python и достаточно легко расширяется python-функциями в плане валидации.

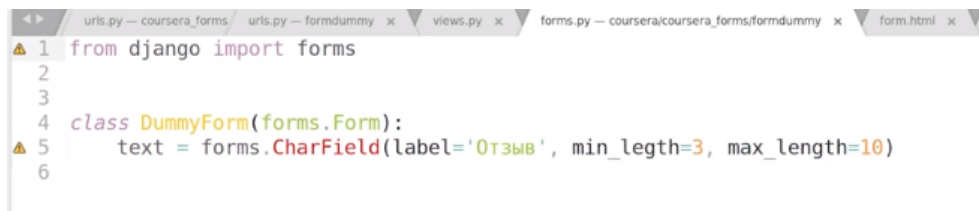
## 2.3. Использование форм в Django

Итак, для начала создадим файл Forms.py в нашем приложении formdummy.

Импортируем из Django.forms наши формы. Создадим класс формы, назовем ее DummyForm от наследуемого класса Form. И теперь мы должны описать поля, которые у нас будут в этой форме. Как мы помним, их у нас три: это текст отзыва, оценка и, соответственно, изображение. Поля описываются примерно, как и модели, через переменные класса. Назовем переменную text = forms. Для текста у нас есть специальное поле CharField, мы указываем label для этого поля. Назовем его Отзыв и указываем параметр max\_length, который, как мы помним, ограничивает максимальную длину. Ограничиваем ее по-прежнему десятью символами, минимальную длину по-прежнему ограничиваем тремя.

Вообще class form можно отрендерить в HTML. Для того чтобы это сделать, нам необходимо импортировать в наш View нашу форму, import DummyForm.

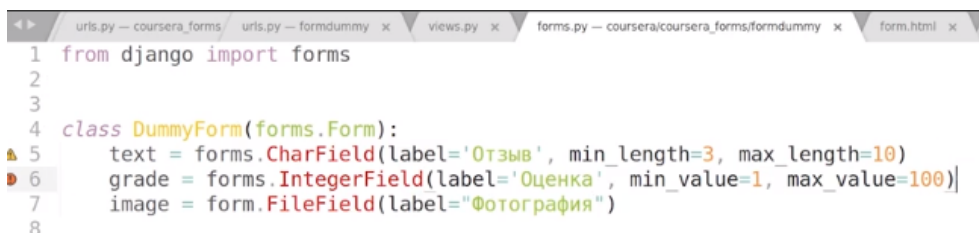
Теперь мы инициализируем нашу форму, передаем ее в контекст, и в нашем шаблоне мы должны ее поставить на место. Объект Form рендерится не в полную форму с тегом form, а рендерится непосредственно только input, и поэтому мы закомментируем эти инпуты, вместо них поставим форму. Так, давайте попробуем загрузить. Итак, мы видим, что у нас отобразилась форма с одним элементом.



```
1 from django import forms
2
3
4 class DummyForm(forms.Form):
5     text = forms.CharField(label='Отзыв', min_length=3, max_length=10)
6
```

Рис. 23

Теперь давайте добавим оставшиеся. Как мы помним, это у нас поле grade. Поле grade у нас является целым числом, поэтому поле IntegerField, название Оценка и, соответственно, параметры для ограничения min\_value и max\_value. Соответственно, как мы помним, от 1 до 100. И также у нас еще есть image файл.

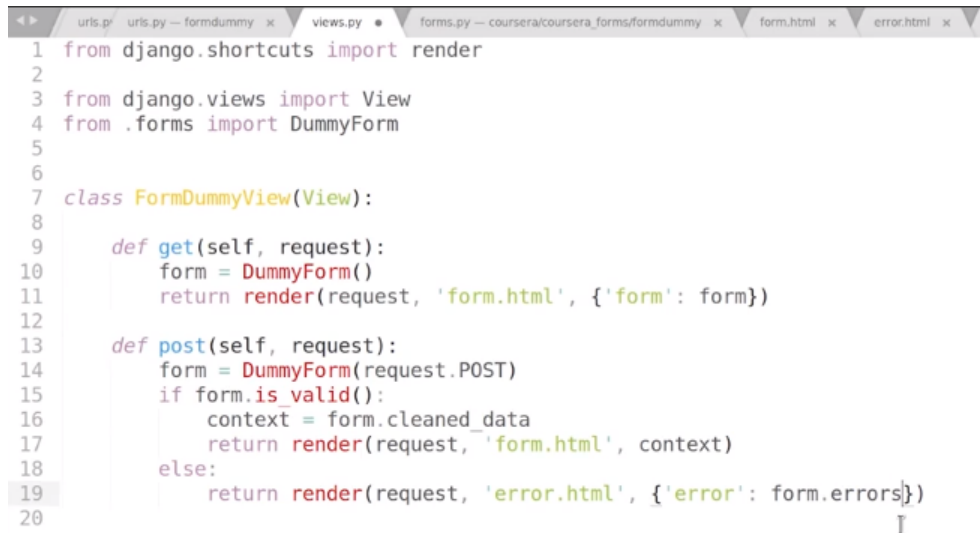


```
1 from django import forms
2
3
4 class DummyForm(forms.Form):
5     text = forms.CharField(label='Отзыв', min_length=3, max_length=10)
6     grade = forms.IntegerField(label='Оценка', min_value=1, max_value=100)
7     image = form.FileField(label='Фотография')
8
```

Рис. 24

Помимо того, что мы можем очень просто рендерить форму, мы можем еще осуществлять валидацию этой формы достаточно простым способом. Для этого нам нужно заполнить нашу форму

следующим образом, POST. Таким образом, данные из POST запроса попадут непосредственно в форму. Нам достаточно будет проверить, является ли форма валидной. Если она является, то мы можем отрендерить тот шаблон, который у нас уже и так был с данными context. При этом context у нас будет заполняться уже из формы. Если же форма не проходит валидацию, то мы отобразим некий шаблон с ошибкой.



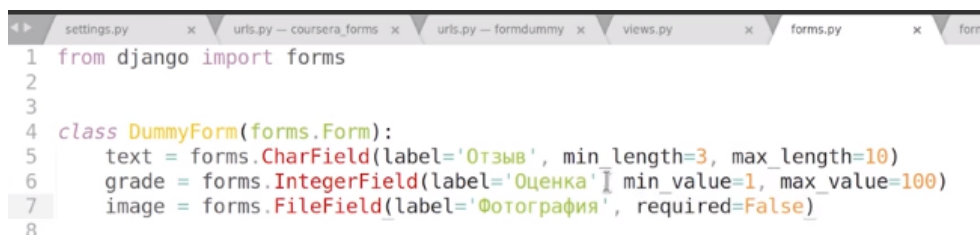
```
1 from django.shortcuts import render
2
3 from django.views import View
4 from .forms import DummyForm
5
6
7 class FormDummyView(View):
8
9     def get(self, request):
10         form = DummyForm()
11         return render(request, 'form.html', {'form': form})
12
13     def post(self, request):
14         form = DummyForm(request.POST)
15         if form.is_valid():
16             context = form.cleaned_data
17             return render(request, 'form.html', context)
18         else:
19             return render(request, 'error.html', {'error': form.errors})
20
```

Рис. 25

Давайте создадим отдельный шаблон error.html, где крупными буквами напишем Ошибка.

Итак, мы с вами создали форму, создали все три поля в ней, она у нас валидно отображается. Давайте попробуем отправить только отзыв и оценку. Поле с фотографией стало обязательным, хотя такого эффекта мы не ожидали. Это всё потому, что в поле Forms каждое из описанных полей является по умолчанию обязательным.

Давайте поле Фотография сделаем необязательным. Для этого нужно параметру required установить значение False.



```
1 from django import forms
2
3
4 class DummyForm(forms.Form):
5     text = forms.CharField(label='Отзыв', min_length=3, max_length=10)
6     grade = forms.IntegerField(label='Оценка', min_value=1, max_value=100)
7     image = forms.FileField(label='Фотография', required=False)
8
```

Рис. 26

Перезагрузим форму, установим значение и осуществим отправку. Отправка была успешно осуществлена.

Давайте попробуем сделать так, чтобы наша форма была невалидной. Для этого давайте опишем новое правило валидации на стороне сервера для поля `text`. Делается это следующим образом. Мы описываем метод `clean_text` и ожидаем, что в нашем поле `clean_text` есть значение, например, `abc`. Напишем сопровождающее сообщение, например: `Вы не о том пишете`. Попробуем спровоцировать отказ формы. Возвращаемся на нашу форму, пишем наш отзыв. Как видим, букв `abc` здесь нет, поэтому должен быть спровоцирован отказ. Нажимаем Отправить запрос, отобразится наш шаблон с ошибкой.



```
1 from django import forms
2
3
4 class DummyForm(forms.Form):
5     text = forms.CharField(label='Отзыв', min_length=3, max_length=10)
6     grade = forms.IntegerField(label='Оценка', min_value=1, max_value=100)
7     image = forms.FileField(label='Фотография', required=False)
8
9     def clean_text(self):
10         if "abc" not in self.cleaned_data['text']:
11             raise forms.ValidationError('Вы не о том пишете')
12
```

Рис. 27

## 2.4. Использование сторонних валидаторов (jsonschema)

JSON Schema подразумевает, что есть некий документ, это может быть JSON, либо это может быть словарь в PYTHON, по которому мы валидируем другие JSON-документы, то есть данные, которые нам приходят от пользователя.

Давайте попробуем, собственно, реализовать валидацию этих данных пользователя. Для этого мы с вами на базе того View, что у нас уже был, создадим ещё один, потому что первый нам ещё понадобится. Назовём его `SchemaView`. `SchemaView` у нас будет имитировать точку API, поэтому метод `get` ей не понадобится, потому что в методе `get`, насколько вы помните, у нас возвращается HTML форма. Здесь мы будем сразу принимать данные в JSON-документе.

`SchemaView` у нас есть, давайте сразу подменим тот View, что у нас был, на новый. Создадим тот самый документ, по которому мы будем валидировать JSON, который к нам приходит от пользователей. Назовём файл `schema.py`, а лучше `schemas`, потому что их будет несколько. Нажимаем Сохранить.

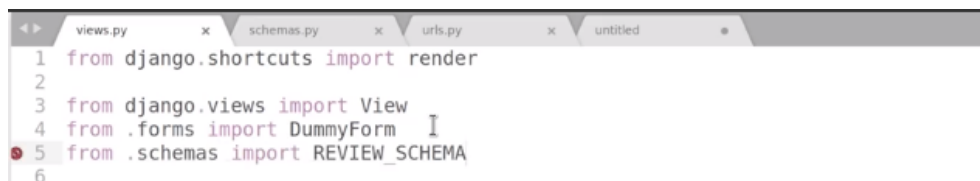


```
1 REVIEW_SCHEMA = {
2     '$schema': 'http://json-schema.org/schema#',
3     'type': 'object',
4     'properties': {
5         'feedback': {
6             'type': 'string',
7             'minLength': 3,
8             'maxLength': 10,
9         },
10        'grade': {
11            'type': 'integer',
12            'minimum': 1,
13            'maximum': 100,
14        },
15    },
16    'required': ['feedback', 'grade'],
17 }
18
```

Рис. 28

В словаре есть указание на то, что это JSON Schema, и ссылка на стандарт. Также мы указываем на то, что внутри корневого документа у нас JSON-объект. И в этом JSON-объекте должны быть поля `feedback`, который является строкой с длиной от трёх символов до десяти, и поле `grade`, которое является числом, которого значения могут принимать от 1 до 100. Оба поля обязательны к заполнению, поэтому они находятся в поле `required`.

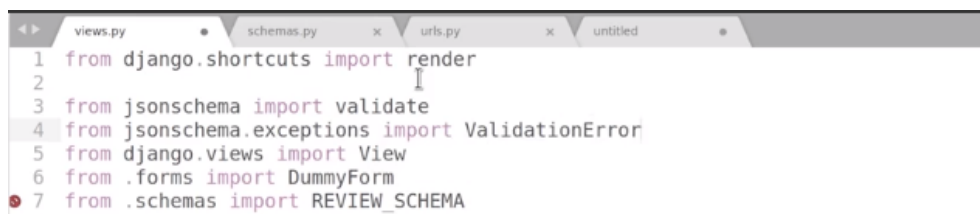
Теперь давайте её импортируем.



```
1 from django.shortcuts import render
2
3 from django.views import View
4 from .forms import DummyForm
5 from .schemas import REVIEW_SCHEMA
6
```

Рис. 29

Давайте добавим, собственно, валидатор. Это основной метод, который отвечает, собственно, за валидацию. И мы добавим ещё исключение `ValidationError` для того, чтобы реагировать на ошибки. Так как нам будет приходиться от пользователя JSON-документ, нам нужно будет его декодировать. То есть нам нужно ещё `import json`, стандартная библиотека. В точке, которую мы делаем, мы будем получать наш документ, декодировать и валидировать его.



```
1 from django.shortcuts import render
2
3 from jsonschema import validate
4 from jsonschema.exceptions import ValidationError
5 from django.views import View
6 from .forms import DummyForm
7 from .schemas import REVIEW_SCHEMA
```

Рис. 30



Итак, первое действие — это декодирование нашего JSON-документа. Итак, в момент декодирования у нас может произойти ошибка `JSONDecodeError`, на которую нам тоже нужно отреагировать. Пока мы не будем ничего здесь возвращать. И мы хотим провалидировать документ. Для этого есть метод `validate`. В метод `validate` мы передаём наш декодируемый документ и указываем, относительно какой схемы его валидировать. И, соответственно, реагируем на ошибку `ValidationError`.

```
24
25 class SchemaView(View):
26
27     def post(self, request):
28         try:
29             document = json.loads(request.body)
30             validate(document, REVIEW_SCHEMA)
31         except json.JSONDecodeError:
32             pass
33         except ValidationError:
34
```

Рис. 31

Итак, допустим, пользователь прислал нам совершенно валидный документ, он прошёл валидацию, нам нужно как-то увидеть, что нам, собственно, возвращается. Поэтому мы сразу после валидации вернём его в неизменном виде. Для этого мы сделаем `return JsonResponse`, передадим туда наш документ, и статус такого ответа будет 201. В случае, если наш JSON не валиден, мы возвращаем `JsonResponse`, в этом `JsonResponse` мы возвращаем человекочитаемую ошибку, а именно `Invalid JSON`. И статус-код 400, потому что ошибки валидации пользовательских данных — это ошибки группы 400 по HTTP.

```
class SchemaView(View):
    def post(self, request):
        try:
            document = json.loads(request.body)
            validate(document, REVIEW_SCHEMA)
            return JsonResponse(document, status=201)
        except json.JSONDecodeError:
            return JsonResponse({'errors': 'Invalid JSON'}, status=400)
        except ValidationError:
            pass
```

Рис. 32

Последняя ошибка, которую нам надо обработать, это `Validation Error`. В данном случае мы передадим то, что нам вернёт наш валидатор в качестве ошибки.



```
25 class SchemaView(View):
26
27     def post(self, request):
28         try:
29             document = json.loads(request.body)
30             validate(document, REVIEW_SCHEMA)
31             return JsonResponse(document, status=201)
32         except json.JSONDecodeError:
33             return JsonResponse({'errors': 'Invalid JSON'}, status=400)
34         except ValidationError as exc:
35             return JsonResponse({'errors': exc.message}, status=400)
36
```

Рис. 33

Импортируем из `django.http` наш `JsonResponse`.

Запускаем на 8000-м порту. Всё запускается. Запрос мы будем посылать посредством командной строки утилиты `curl`. Мы при помощи метода `curl` отправляем запрос на наш сервер на точку `form`, внутри этого запроса, в теле запроса, мы передаём вот такой документ, и в `header`'ах указываем то, что это `application/json`, то есть JSON-документ. И передаём это естественным методом `POST`. У нас возникла ошибка, которая говорит нам о том, что у нас нет CSRF-токенов. Совершенно валидное замечание. Тем более, что мы его тут действительно не передавали. Так как это у нас точка API и CSRF-токены передаются там достаточно сложным способом или там используются другие методы защиты от CSRF-атак, мы пока отключим эту защиту. На продакшене, естественно, так не делайте.

Итак, давайте импортируем специальные декораторы, которые отключат нам защиту. И, собственно, мы эту защиту вот здесь и отключим.

```
views.py  schemas.py  urls.py  untitled
1 import json
2 from jsonschema import validate
3 from jsonschema.exceptions import ValidationError
4 from django.http import JsonResponse
5 from django.shortcuts import render
6 from django.views import View
7 from django.views.decorators.csrf import csrf_exempt
8 from django.utils.decorators import method_decorator
9 from .forms import DummyForm
10 from .schemas import REVIEW_SCHEMA
11
```

Рис. 34

Нам возвращается 201-й ответ, который говорит о том, что всё прошло штатно.

Попробуем спровоцировать ошибки. Давайте попробуем для начала отправить невалидный документ. Для этого достаточно стереть одну скобочку, и мы получаем ответ 400, и в теле ответа мы видим человекочитаемую ошибку, которая говорит нам, что наш JSON-документ не валиден.

```

:"hello"' http://127.0.0.1:8000/form/
Note: Unnecessary use of -X or --request, POST is already inferred.
* Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to 127.0.0.1 (127.0.0.1) port 8000 (#0)
> POST /form/ HTTP/1.1
> Host: 127.0.0.1:8000
> User-Agent: curl/7.55.1
> Accept: */*
> Content-Type: application/json
> Content-Length: 30
>
* upload completely sent off: 30 out of 30 bytes
< HTTP/1.1 400 Bad Request
< Date: Thu, 18 Jan 2018 15:52:35 GMT
< Server: WSGIServer/0.2 CPython/3.6.3
< Content-Type: application/json
< X-Frame-Options: SAMEORIGIN
< Content-Length: 26
<
* Connection #0 to host 127.0.0.1 left intact
{"errors": "Invalid JSON"}

```

Рис. 35

Теперь давайте попробуем спровоцировать ошибку, чтобы JSON Schema вернула нам ошибку. Итак, здесь поставим 420, вместо 42-х, как мы помним, у нас верхний предел — это 100. И, собственно, мы видим, что нам возвращается 400-й ответ.

```


>
* upload completely sent off: 32 out of 32 bytes
< HTTP/1.1 400 Bad Request
< Date: Thu, 18 Jan 2018 15:53:07 GMT
< Server: WSGIServer/0.2 CPython/3.6.3
< Content-Type: application/json
< X-Frame-Options: SAMEORIGIN
< Content-Length: 52
<
* Connection #0 to host 127.0.0.1 left intact
{"errors": "420 is greater than the maximum of 100"}droppoint@Partlabs-Mail:~$
droppoint@Partlabs-Mail:~$
droppoint@Partlabs-Mail:~$
droppoint@Partlabs-Mail:~$

```

Рис. 36

## 2.5. Использование сторонних валидаторов (marshmallow)

Marshmallow очень похож на django forms. То есть для того чтобы что-то свалидировать, нужно описать класс с данными, по которыми мы, собственно, потом будем валидировать. Перейдем в schemas.py. Импортируем класс Schema и fields.



```

views.py x schemas.py • untyped •
1 from marshmallow import Schema, fields
2
3

```

Рис. 37

И мы пишем нашу схему, которая будет называться ReviewSchema. Наследуется от Schema. Так, поля, которые будет проверять наша схема, описываются так же, как в Django forms, в виде переменных класса.

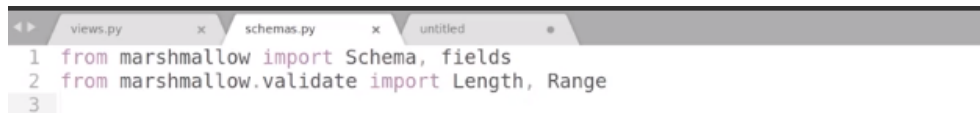
У нас есть два поля: это feedback и grade. Grade является числом. В Marshmallow есть несколько простых валидаторов, в частности, есть валидатор длины, то есть, что длина не короче и не длиннее определенного значения. И есть валидатор числа, тоже по верхней и по нижней границе.

Итак, для того чтобы задействовать валидатор, нужно передать его в скобках в параметре validate. Итак, валидатор для числа называется Range. Как помним, нижняя граница 1, а верхняя 100. Валидатор для строки называется length. И он от 3 до 10.

```
22
23 class ReviewSchema(Schema):
24     feedback = fields.Str(validate=Length(3, 10))
25     grade = fields.Int(validate=Range(1, 100))
26
```

Рис. 38

Добавляем соответствующую строчку в import.



```
views.py x schemas.py x untitled
1 from marshmallow import Schema, fields
2 from marshmallow.validate import Length, Range
3
```

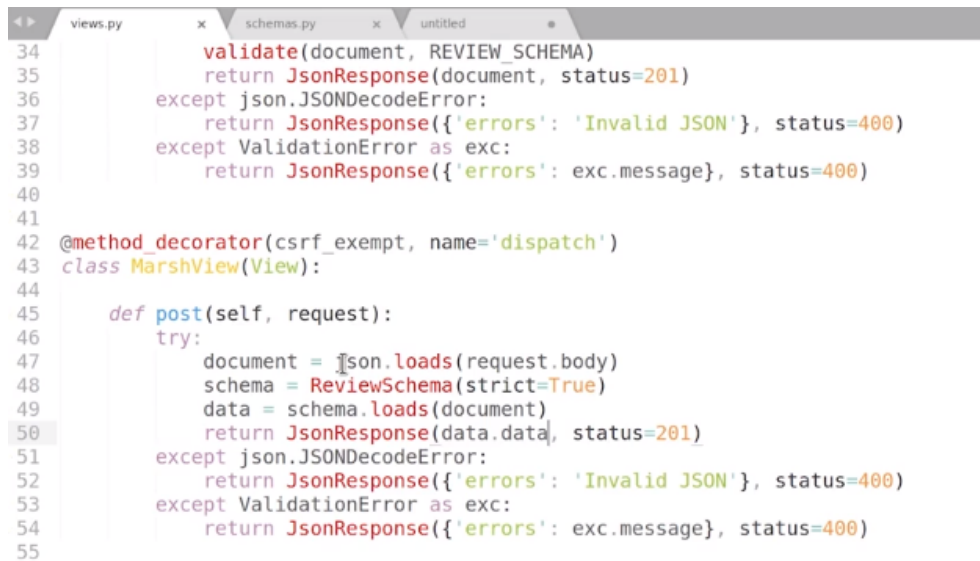
Рис. 39

В нашем MarshView мы убираем то, что осталось от JSON Schema. Импортируем нашу новую ReviewSchema. Инициализируем ее. И передаем ей режим strict=True. Этот режим заставит нашу схему при загрузке, если схема заметит невалидные данные, поднять исключение.

```
1
2 @method_decorator(csrf_exempt, name='dispatch')
3 class MarshView(View):
4
5     def post(self, request):
6         try:
7             schema = ReviewSchema(strict=True)
8             return JsonResponse(document, status=201)
9         except json.JSONDecodeError:
10            return JsonResponse({'errors': 'Invalid JSON'}, status=400)
11        except ValidationError as exc:
12            return JsonResponse({'errors': exc.message}, status=400)
13
```

Рис. 40

Итак, давайте загрузим наш документ.



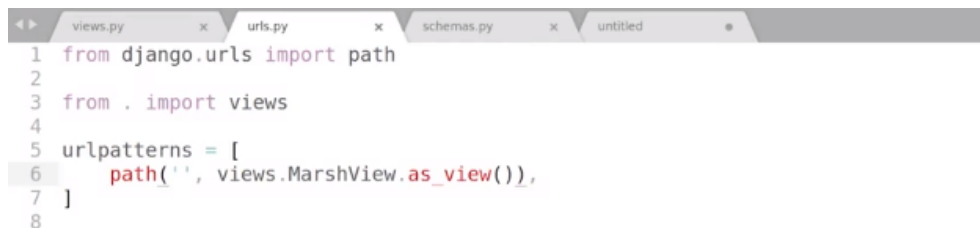
```

34     validate(document, REVIEW_SCHEMA)
35     return JsonResponse(document, status=201)
36 except json.JSONDecodeError:
37     return JsonResponse({'errors': 'Invalid JSON'}, status=400)
38 except ValidationError as exc:
39     return JsonResponse({'errors': exc.message}, status=400)
40
41
42 @method_decorator(csrf_exempt, name='dispatch')
43 class MarshView(View):
44
45     def post(self, request):
46         try:
47             document = json.loads(request.body)
48             schema = ReviewSchema(strict=True)
49             data = schema.loads(document)
50             return JsonResponse(data.data, status=201)
51 except json.JSONDecodeError:
52     return JsonResponse({'errors': 'Invalid JSON'}, status=400)
53 except ValidationError as exc:
54     return JsonResponse({'errors': exc.message}, status=400)
55

```

Рис. 41

Теперь давайте зарегистрируем наш View в urls, в MarshView.



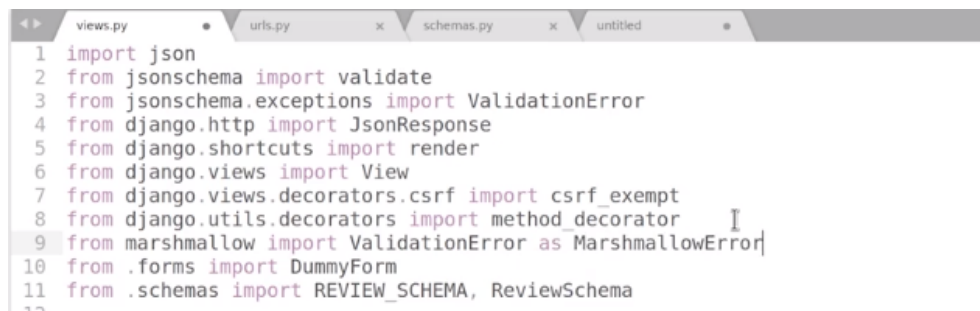
```

1 from django.urls import path
2
3 from . import views
4
5 urlpatterns = [
6     path('', views.MarshView.as_view()),
7 ]
8

```

Рис. 42

Давайте попробуем запустить сервер. И, соответственно, отправим запрос. Немного исправим обработчик. Мы забыли то, что ValidationError у нас идет из jsonschema, у Marshmallow соответственно другое исключение, которое называется точно так же. Поэтому мы его импортируем. Так как у нас названия конфликтуют между этими двумя исключениями, мы импортируем его под другим именем.



```

1 import json
2 from jsonschema import validate
3 from jsonschema.exceptions import ValidationError
4 from django.http import JsonResponse
5 from django.shortcuts import render
6 from django.views import View
7 from django.views.decorators.csrf import csrf_exempt
8 from django.utils.decorators import method_decorator
9 from marshmallow import ValidationError as MarshmallowError
10 from .forms import DummyForm
11 from .schemas import REVIEW_SCHEMA, ReviewSchema
12

```

Рис. 43

Мы передали 420 в документе, поэтому мы получили ошибку 400. Ну давайте попробуем передать невалидный документ. Берем отсюда 0. Нам вернется правильная ошибка. А теперь давайте попробуем задействовать так называемый Happy Path, то есть когда все так, как и должно быть. Получим 201-ый ответ.

## 3. Аутентификация, авторизация, сессии

### 3.1. Аутентификация и авторизация

**Аутентификация** — пользователь тот, за кого себя выдает. **Авторизация** — пользователь имеет право сделать то, что он запрашивает.

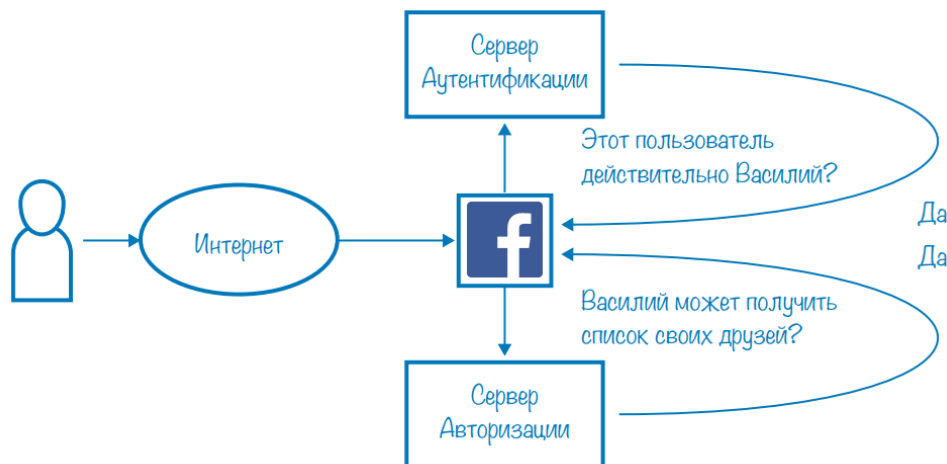


Рис. 44

Аутентификация может быть нескольких типов:

- Многоразовый пароль.
- Одноразовый пароль.
- Аппаратный ключ.
- Строгая взаимная аутентификация.

Строгая взаимная аутентификация – на каждом из устройств генерируется надежный, длинный ключ. Каждое из них может подключиться друг к другу, только зная эти ключи. Ключи передаются обычно через сторонний канал. Этот способ самый надежный, но самый сложный. Аутентификация по многоразовому паролю – самый простой, но самый слабый из методов. Реализация:

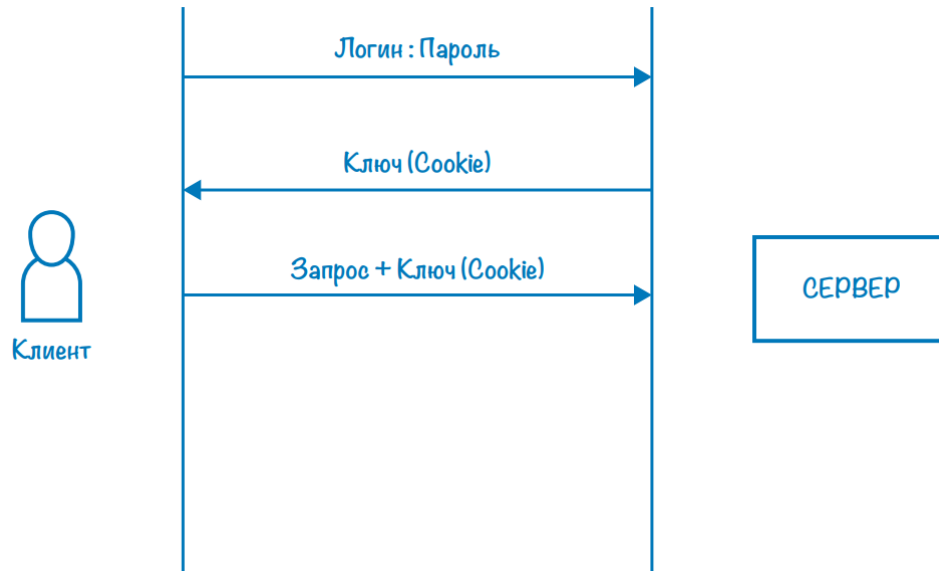


Рис. 45

## 3.2. Аутентификация пользователей в Django

Django предоставляет нам большой набор инструментов для того, чтобы обеспечить аутентификацию, авторизацию пользователей.

У нас есть некий `FormDummyView`, в котором отображаются формы, и мы можем посылать данные в эту форму. Давайте защитим этот View и не будем его показывать неаутентифицированным пользователям.

Для того чтобы настроить аутентификацию, нам нужно зайти в настройки проекта. В настройках проекта должно быть включено приложение Django contrib auth, должны быть включены `MIDDLEWARE`, `sessionmiddleware` и `authenticationmiddleware`. И так же должен быть включен контекст-процессор `auth`. По умолчанию все эти вещи включены, поэтому ничего дополнительно нам делать не нужно.

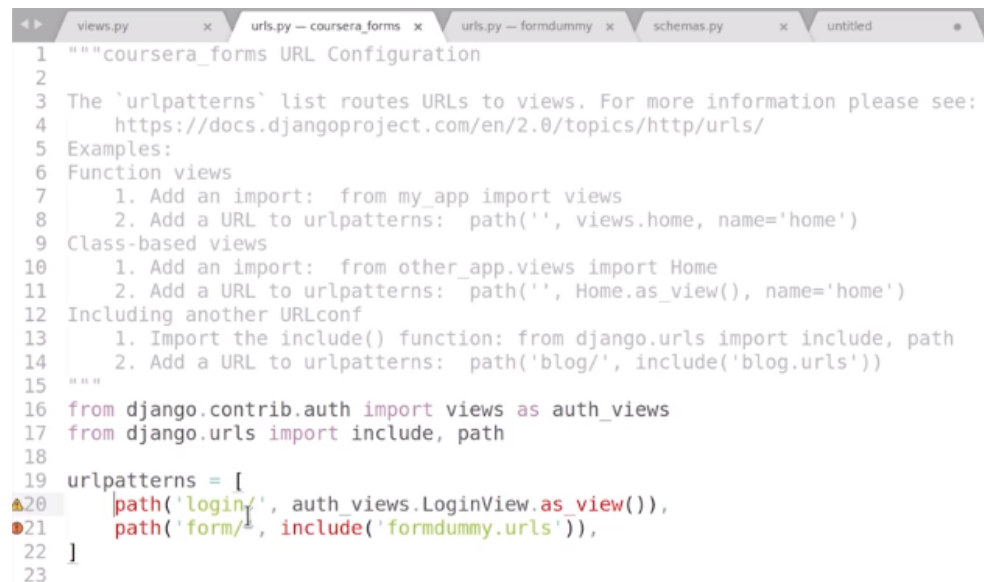
Также для работы аутентификации нужна модель пользователя. До сих пор мы с вами действовали без использования базы данных, но теперь давайте воспользуемся этой базой данных, чтобы хранить там пользователей. Итак, для того, чтобы заполнить базу данных, которая по умолчанию в Django используется `SQL lite 3`, нам нужно использовать команду `migrate`. Накладываются миграции, которые создают необходимые нам таблицы.

Далее нам необходимо создать пользователя, из-под которого мы будем логиниться в наш проект. Давайте откроем консоль нашего проекта. Делается это посредством команды `manage.py shell`, и из приложения мы импортируем модель пользователя. Он, собственно, так и называется, `user`. Итак, создадим юзера через специальный метод.

```
Python 3.6.3 (default, Oct 3 2017, 21:45:48)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_user('hello', 'hello@example.org', 'world')
>>> user.save(0)
... user.save(0)
    File "<console>", line 2
          user.save(0)
            ^
SyntaxError: invalid syntax
>>> user.save()
```

Рис. 46

Далее нам нужно зарегистрировать view из приложения Django contrib auth, для того чтобы мы могли логиниться с собственного view с формой логина. Делается это внутри urls файла проекта. Мы используем специальный view под названием login view.

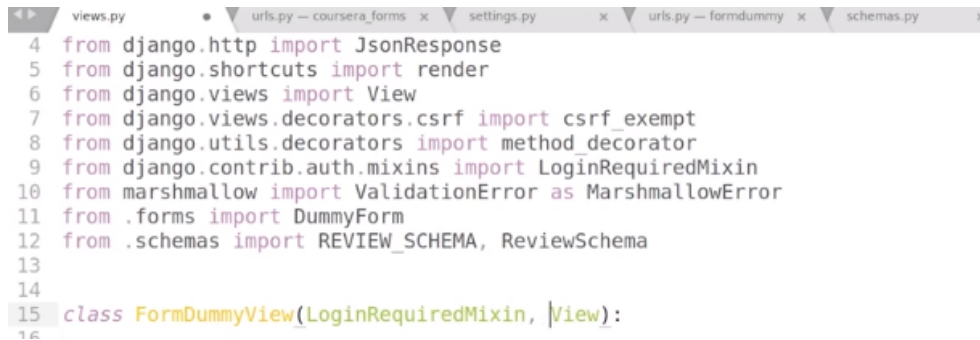


```
1 """coursera_forms URL Configuration
2
3 The `urlpatterns` list routes URLs to views. For more information please see:
4     https://docs.djangoproject.com/en/2.0/topics/http/urls/
5 Examples:
6 Function views
7     1. Add an import:  from my_app import views
8     2. Add a URL to urlpatterns:  path('', views.home, name='home')
9 Class-based views
10    1. Add an import:  from other_app.views import Home
11    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
12 Including another URLconf
13    1. Import the include() function: from django.urls import include, path
14    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
15 """
16 from django.contrib.auth import views as auth_views
17 from django.urls import include, path
18
19 urlpatterns = [
20     path('login/', auth_views.LoginView.as_view()),
21     path('form/', include('formdummy.urls')),
22 ]
23
```

Рис. 47

Форма логина у нас будет по пути, логин в корне нашего проекта. Для того чтобы неавторизованных пользователей отправляло на эту точку, на эту страницу, нам необходимо добавить настройку login url, которая будет называться login. Итак, чтобы защитить нашу view от того, чтобы неавторизованные, не аутентифицированные пользователи не попали на нее, нам необходимо добавить mixin.





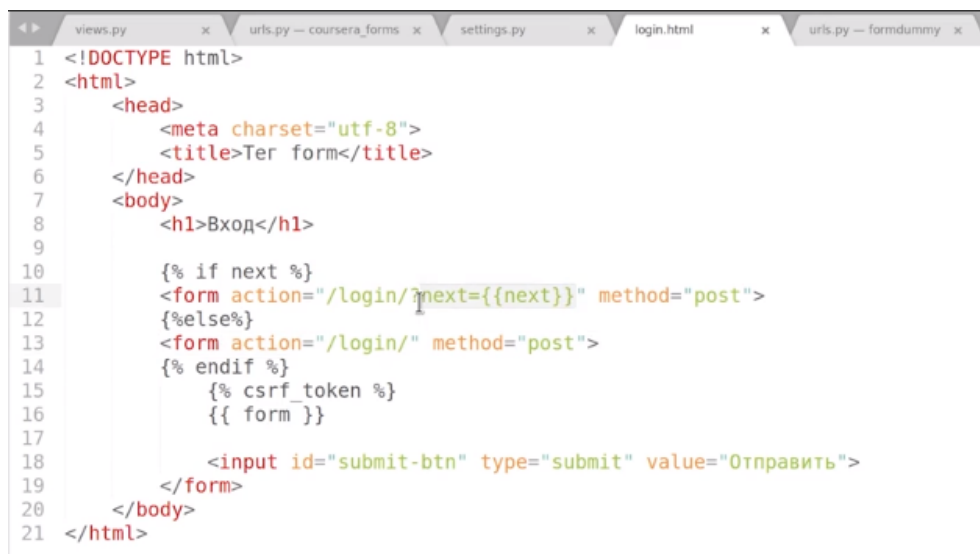
```

4 from django.http import JsonResponse
5 from django.shortcuts import render
6 from django.views import View
7 from django.views.decorators.csrf import csrf_exempt
8 from django.utils.decorators import method_decorator
9 from django.contrib.auth.mixins import LoginRequiredMixin
10 from marshmallow import ValidationError as MarshmallowError
11 from .forms import DummyForm
12 from .schemas import REVIEW_SCHEMA, ReviewSchema
13
14
15 class FormDummyView(LoginRequiredMixin, View):

```

Рис. 48

Итак, осталась небольшая еще вещь. Нам необходимо добавить шаблон для нашей формы логина, потому что по умолчанию в Django он не предоставляется. Итак, давайте создадим папку templates в корне проекта, и в этой папке мы создадим папку registration, и внутри этой папки создадим файл login.html, который и будет, собственно, нашим шаблоном. Это - стандартная форма, которая отправляет данные на точку логин, и в случае, если есть страница, на которую мы хотим перейти, она отправляет гет-параметр next ещё дополнительно.




```

1 <!DOCTYPE html>
2 <html>
3     <head>
4         <meta charset="utf-8">
5         <title>Ter form</title>
6     </head>
7     <body>
8         <h1>Вход</h1>
9
10        {% if next %}
11        <form action="/login/?next={{next}}" method="post">
12        {%else%}
13        <form action="/login/" method="post">
14        {% endif %}
15            {% csrf_token %}
16            {{ form }}
17
18            <input id="submit-btn" type="submit" value="Отправить">
19        </form>
20    </body>
21 </html>

```

Рис. 49

Также нужно еще изменить немного настройки шаблонов, чтобы джанго-проект начал видеть наш шаблон. Для этого нужно добавить вот такую строчку в настройку dirs в настройки templates.



```

55 TEMPLATES = [
56     {
57         'BACKEND': 'django.template.backends.django.DjangoTemplates',
58         'DIRS': [os.path.join(BASE_DIR, 'templates')],
59         'APP_DIRS': True,
60         'OPTIONS': {
61             'context_processors': [

```

Рис. 50



Теперь нас пустит на страницу с формой только после авторизации.

### 3.3. Улучшаем проект

Начнем с того, что мы возьмем наше приложение `formdummy` и переименуем его, потому что оно не полностью отражает цель нашего приложения, мы назовем его `feed back`. Так, приложение `feed back`, для того, чтобы оно по прежнему правильно работало, мы должны переименовать `config`, мы должны изменить название нашего приложения в `setting spy`, также в `URLS`, мы еще переименуем `view`. Первое, что мы сделаем после того, как мы переименовали приложение, мы создадим модель, в котором мы будем хранить все наши отзывы. Т.е. пользователь приносит отзыв, передает нам данные, мы эти данные сохраняем в базе данных, для этого нужна как мы помним модель. Так, мы создадим модель с названием `feed back`. Желательно классом писать для чего это модель или класс, потому что не всегда понятно из названия класса для чего она нужна. Итак, обратите внимание на параметр `verbose_name`, он задает человекочитаемое имя для поля. Одно из полей, а именно автор, это поле `foreign key`, то есть это ссылка на другую модель, в нашем случае эту модель пользователя, то есть мы будем привязывать каждую из наших отзывов к пользователю, который его сделал и, соответственно, так как это ссылка на внешнюю модель, мы задаем сразу поведение при удалении этой внешней модели, то есть при удалении пользователя у нас все отзывы для этого пользователя также будут удалены.



```
1 from django.db import models
2
3
4 class Feedback(models.Model):
5     """Отзыв о чем угодно."""
6     text = models.CharField(verbose_name='Отзыв', max_length=5000)
7     grade = models.IntegerField(verbose_name='Оценка')
8     author = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
9     subject = models.CharField(verbose_name='Объект', max_length=100)
10
```

Рис. 51

Модель готова.

После того, как мы сделали нашу модель, давайте приступим к созданию нашей `view`, которая будет собственно создавать наши отзывы. В Django есть так называемый класс `best views`, есть разные их ответвления, мы в частности будем использовать `create view`, который сам по себе генерирует необходимую форму и позволяет на базе этой формы принимать данные и как-то их обрабатывать.

Импортируем нашу модель, так, задаем модель, задаем набор полей, которые будут у нас показаны в нашей форме.

```

17 class FeedbackCreateView(LoginRequiredMixin, CreateView):
18     model = Feedback
19     fields = ['text', 'grade', 'subject']

```

Рис. 52

Итак, мы добавили в нашу модель, на базе которой она строится view, мы добавили поля, теперь нам нужно указать success URL. Этот параметр указывает, куда мы перейдем в случае, если данные будут приняты. Мы хотим возвращать пока что наших пользователей обратно на нашу форму. И мы должны сделать функцию, благодаря которой будет проставляться автор отзыва. Для этого нам надо переопределить метод форм valid, которая запускается для проверки валидности данных нашей формы.

```

class FeedbackCreateView(LoginRequiredMixin, CreateView):
    model = Feedback
    fields = ['text', 'grade', 'subject']
    success_url = '/feedback/add'

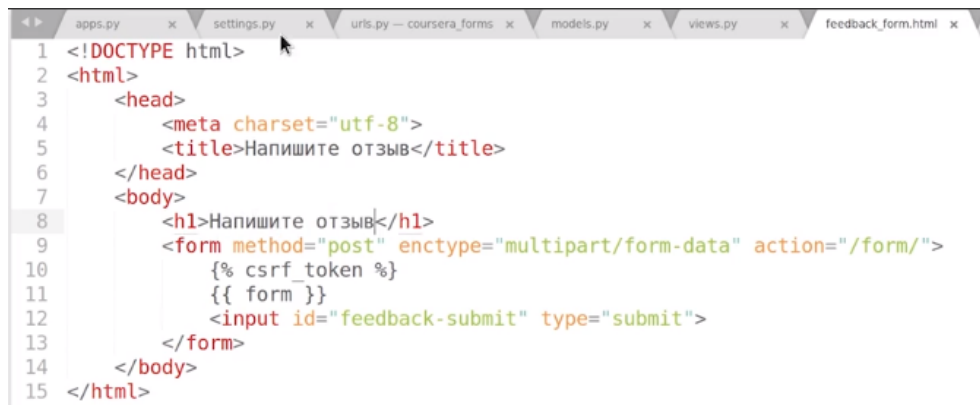
    def form_valid(self, form):
        form.instance.author = self.request.user
        return super().form_valid(form)

```

Рис. 53

Зарегистрируем view.

Теперь нам нужно добавить еще шаблон. Создаем feedback\_form.html в папке template.



```

1 <!DOCTYPE html>
2 <html>
3     <head>
4         <meta charset="utf-8">
5         <title>Напишите отзыв</title>
6     </head>
7     <body>
8         <h1>Напишите отзыв</h1>
9         <form method="post" enctype="multipart/form-data" action="/form/">
10             {% csrf_token %}
11             {{ form }}
12             <input id="feedback-submit" type="submit">
13         </form>
14     </body>
15 </html>

```

Рис. 54

Сервер запустился, давайте попробуем. Нас перебросит на форму логина. Давайте попробуем залогиниться. Как видим, у нас появляется форма, в которой мы можем написать наш отзыв и как видим, есть наши verbose\_name из модели.

Итак, нажимаем отправить запрос, и нас перебрасывает обратно на нашу форму, что означает, что наши данные были успешно приняты и сохранены. Почему перебрасывают обратно на форму? Как я уже говорил, есть параметр success URL, которое указывают, куда нужно возвращать пользователя после того, как данные были успешно введены.

## 3.4. Авторизация в Django

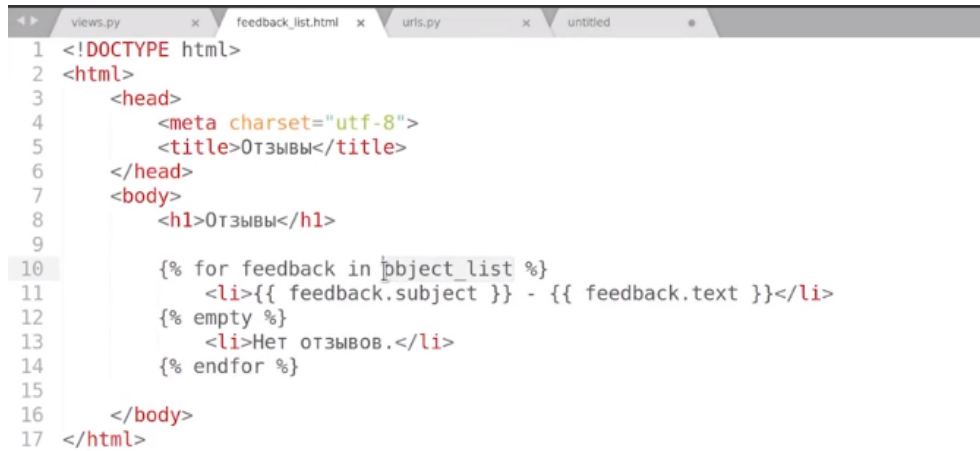
Итак, для того чтобы нам создать view, который у нас будет выдавать список наших отзывов, мы должны создать новый view. Он будет называться `FeedbackListView`. Он будет наследоваться от `LoginRequiredMixin`, для того чтобы только аутентифицированные пользователи могли на него зайти, и будет наследоваться от `ListView`. Также наш view должен знать, для какой модели он работает. Поэтому предоставим ему эти данные. И теперь собственно самое интересное. Наша магия, а именно метод, мы переопределяем метод `get_queryset`, который и будет определять, какие данные выдать одному типу пользователей, какие данные выдать другому типу пользователей. Итак, если наш пользователь, а пользователя мы можем узнать из объекта `request`, нахождение пользователя там обеспечивает `LoginRequiredMixin`, если наш пользователь имеет свойство `is_staff`, то мы предоставляем ему полный список наших отзывов. В том числе не его отзывы, а отзывы других пользователей. Но если пользователь не имеет этого флага, то он получает только свои отзывы. Как помним, привязывает отзыв к пользователю поле `author`. И импортируем `ListView`.

```
class FeedbackListView(LoginRequiredMixin, ListView):  
    model = Feedback  
  
    def get_queryset(self):  
        if self.request.user.is_staff:  
            return Feedback.objects.all()  
        return Feedback.objects.filter(author=self.request.user)
```

Рис. 55

Осталось его зарегистрировать и создать для него шаблон. Давайте его зарегистрируем. Назовем его `feedback-list`. Мы его зарегистрировали и теперь давайте создадим шаблон. Шаблон по соглашению должен лежать в папке `templates`. А должен он называться `feedback_list`, вы можете назвать его как угодно, но в таком случае вам придется переопределить путь для шаблонов в свойствах view.

Самый основной объект, который передается нам в этот шаблон, это список под названием `object_list`. В нем хранится выборка из `get_queryset`. В данном случае мы проходимся по этой выборке, показываем в списке наши отзывы, а именно, о чем был отзыв и сам текст этого отзыва. И в случае, если никаких отзывов нет, то мы просто показываем просто слова «Нет отзывов», и все.



```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <title>Отзывы</title>
6   </head>
7   <body>
8     <h1>Отзывы</h1>
9
10    {% for feedback in object_list %}
11      <li>{{ feedback.subject }} - {{ feedback.text }}</li>
12    {% empty %}
13      <li>Нет отзывов.</li>
14    {% endfor %}
15
16  </body>
17 </html>
```

Рис. 56

Итак, давайте попробуем запустить сервер. Посмотрим вообще, работает ли это для обычного пользователя. Откроем браузер, заходим на feedback.

Давайте попробуем создать другого пользователя и посмотреть, покажутся ли нам те же самые отзывы. По идее не должны, потому что это другой пользователь. Итак, мы создаем нового пользователя.

Итак, пока сохраним пользователя, не предоставляем ему каких-то дополнительных привилегий. Запустим наш сервер. Перезапустим наш браузер, чтобы залогиниться под новым пользователем. Итак, как видим, пользователю не показывается никаких отзывов. Но давайте попробуем предоставить нашему пользователю администраторские права, в данном случае просто привилегированные. И теперь давайте попробуем запустить снова наш сервер. Просто обновим страницу, и как видим, теперь нам показываются отзывы других пользователей, а именно отзывы пользователей hello.