



Урок 4

Аутентификация и регистрация пользователя

Загрузка данных в базу из файлов. Модель пользователя. Процедура аутентификации. Формы в Django. Механизм CRUD при работе с моделями.

[Загрузка данных в БД из файла](#)

[Создание своей модели пользователя](#)

[Аутентификация пользователя](#)

[Работа с диспетчером URL](#)

[Создание контроллеров](#)

[Формы Django](#)

[Работа с формами Django в шаблонах](#)

[Подготовка к реализации механизма CRUD для модели пользователя](#)

[Регистрация пользователя](#)

[Редактирование пользователя](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Загрузка данных в БД из файла

Django предоставляет дополнительные инструменты для управления приложениями в проекте. Создадим в папке с проектом **mainapp** структуру папок:

```
management/commands/
```

И в ней — файл **fill_db.py**.

management/commands/fill_db.py

```
from django.core.management.base import BaseCommand
from mainapp.models import ProductCategory, Product
from django.contrib.auth.models import User

import json, os

JSON_PATH = 'mainapp/json'

def load_from_json(file_name):
    with open(os.path.join(JSON_PATH, file_name + '.json'), 'r') as infile:
        return json.load(infile)

class Command(BaseCommand):
    def handle(self, *args, **options):
        categories = load_from_json('categories')

        ProductCategory.objects.all().delete()
        for category in categories:
            new_category = ProductCategory(**category)
            new_category.save()

        products = load_from_json('products')

        Product.objects.all().delete()
        for product in products:
            category_name = product["category"]
            # Получаем категорию по имени
            _category = ProductCategory.objects.get(name=category_name)
            # Заменяем название категории объектом
            product['category'] = _category
            new_product = Product(**product)
            new_product.save()

        # Создаем суперпользователя при помощи менеджера модели
        super_user = User.objects.create_superuser('django',
            'django@geekshop.local', 'geekbrains')
```

Из названия папок понятно, что речь идет о дополнительных командах. Синтаксис запуска простой:

```
python manage.py fill_db
```

Именно так теперь будем быстро заполнять БД данными. Традиционно рекомендуем создать файл **fill_db.bat**.

По сути, мы просто создали класс **Command(BaseCommand)**, унаследованный от Django-класса **BaseCommand**, и в нем — реализацию метода **handle()**. Остальной код повторяет действия с моделями, которые мы делали в консоли на прошлом уроке. Не забывайте, что сначала создаются записи категорий, а уже потом — записи товаров в категориях. Загрузку объектов JSON мы проходили на курсе Python. Стоит только отметить особенность при программном создании суперпользователя — используем метод менеджера модели **create_superuser()** и передаем ему те же аргументы, что и при работе в консоли. Здесь используем уже существующую в Django модель пользователя **User** — просто импортируем ее.

Создание своей модели пользователя

Несмотря на то, что в Django уже есть модель пользователя **User**, в большинстве случаев приходится создавать свою. Вы можете добавить свои методы и атрибуты (например, аватарку или возраст). Создадим новое приложение в проекте — **authnapp**. В этом приложении на базе класса **AbstractUser** создаем модель **ShopUser**.

authnapp/models.py

```
from django.db import models
from django.contrib.auth.models import AbstractUser

class ShopUser(AbstractUser):
    avatar = models.ImageField(upload_to='users_avatars', blank=True)
    age = models.PositiveIntegerField(verbose_name = 'возраст')
```

Кроме этого надо в файле настроек проекта добавить имя приложения в список **INSTALLED_APPS** и создать константу:

```
AUTH_USER_MODEL = 'authnapp.ShopUser'
```

Это необходимо, чтобы Django вместо модели **User** использовал в приложении аутентификации нашу модель.

Теперь последовательность миграций изменится:

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

Скорректируем код создания суперпользователя в файле **fill_db.py**:

```
from authapp.models import ShopUser

super_user = ShopUser.objects.create_superuser('django',
'django@geekshop.local', 'geekbrains', age=33)
```

Проект усложняется — будьте внимательны.

Аутентификация пользователя

Рассмотрим следующие задачи:

- работа с диспетчером url-адреса, принадлежащего вызову контроллеров механизма аутентификации;
- создание контроллеров;
- создание форм;
- создание шаблонов.

Кратко напомним о [типах запросов](#), которые могут приходить серверу:

- GET — то, что приходит в адресной строке после символа «?» и представляет собой пары «ключ»=«значение», разделенные символом «&;»;
- POST — то, что приходит в теле http-запроса и представляет, по сути, ассоциативный массив, или словарь (в терминах Python).

При помощи GET передаем обычно название страницы или идентификатор объекта (объем ограничен, невозможно скрыть данные).

При помощи POST — логин, пароль, файлы и другие данные (объем ограничен только настройками сервера).

Работа с диспетчером URL

Добавим строку в список **urlpatterns** (основной файл **urls.py**):

```
path('auth/', include('authapp.urls', namespace='auth'))
```

Все остальное пропишем в диспетчере URL-приложения **authapp**:

authapp/urls.py

```
from django.urls import path

import authapp.views as authapp

app_name = 'authapp'

urlpatterns = [
    path('login/', authapp.login, name='login'),
    path('logout/', authapp.logout, name='logout'),
]
```

Обратите внимание что необходимо указывать **app_name**. Это требование функции **include**

Какой адрес необходимо прописать, чтобы сработал контроллер **login()** из файла **'authapp/views.py'**?

Ответ:

```
127.0.0.1:8000/auth/login/
```

Если не смогли правильно ответить — повторите материал предыдущего урока.

Создание контроллеров

Напишем контроллеры в приложении **authapp**.

authapp/views.py

```
from django.shortcuts import render, HttpResponseRedirect
from authapp.forms import ShopUserLoginForm
from django.contrib import auth
from django.urls import reverse

def login(request):
    title = 'Вход'

    login_form = ShopUserLoginForm(data=request.POST)
    if request.method == 'POST' and login_form.is_valid():
        username = request.POST['username']
        password = request.POST['password']

        user = auth.authenticate(username=username, password=password)
        if user and user.is_active:
            auth.login(request, user)
            return HttpResponseRedirect(reverse('main'))

    content = {'title': title, 'login_form': login_form}
    return render(request, 'authapp/login.html', content)

def logout(request):
    auth.logout(request)
    return HttpResponseRedirect(reverse('main'))
```

Рассмотрим контроллер **login**. Здесь мы воспользовались механизмом форм Django. Форма генерируется автоматически на основе соответствующей модели (в данном случае **ShopUser**).

Не вдаваясь в подробности, будем считать, что **ShopUserLoginForm(data=request.POST)** возвращает html-код классической формы для логина. Если запрос выполнен методом GET, форма будет пустая (так как **request.POST** равен **None**). Помещаем ее в контекст и рендерим шаблон **'authapp/login.html'** (создадим его позже).

Если запрос выполнен методом **POST**, форма будет заполнена данными, которые ввел пользователь на странице. Метод формы **is_valid()** проверяет их корректность в соответствии с атрибутами модели. Далее получаем из словаря POST-данных логин и пароль и вызываем встроенную в Django функцию аутентификации:

```
auth.authenticate(username=username, password=password)
```

В случае успеха она вернет объект пользователя в переменную **user**. Если пользователь активен (проверяем атрибут модели **is_active**) — вызываем функцию **auth.login(request, user)**. Она пропишет пользователя в объект запроса **request**. Все, залогинились!

Остается вернуться на главную страницу — для этого используем функцию `HttpResponseRedirect()`. Можно просто передать ей url-адрес в виде строки, но мы сделали интереснее — воспользовались функцией `reverse()`. Она возвращает адрес по его имени в диспетчере URL. Если вам эта схема кажется сложной — пропишите:

```
HttpResponseRedirect('/')
```

Замечание:

1. Аутентификация считается успешной, если объект пользователя появился в `request`.
2. Если диспетчер URL Django не находит имя, которое мы передали в `reverse()` — будет ошибка.

Сейчас уже должно стать очевидным, что для выхода из системы достаточно просто удалить объект пользователя из `request`. Вы догадались, что делает функция `auth.logout(request)`? Правильно!

Формы Django

Вы уже представляете роль форм в Django. Это мощный механизм, избавляющий от большого количества рутины. Главное в Django — модель, а остальное является ее следствием. Дальше вы в этом не раз убедитесь.

Создадим файл `forms.py` в приложении `authapp`.

`authapp/forms.py`

```
from django.contrib.auth.forms import AuthenticationForm

from .models import ShopUser

class ShopUserLoginForm(AuthenticationForm):
    class Meta:
        model = ShopUser
        fields = ('username', 'password')

    def __init__(self, *args, **kwargs):
        super(ShopUserLoginForm, self).__init__(*args, **kwargs)
        for field_name, field in self.fields.items():
            field.widget.attrs['class'] = 'form-control'
```

Как читать этот код? Видим класс `ShopUserLoginForm`, унаследованный от Django класса `AuthenticationForm`. Модель на базе, на которой строится форма, прописывается в атрибуте `model` класса `Meta`. Имена атрибутов модели, которые необходимо вывести на странице, — в атрибуте `fields`.

В этом коде используется один лайфхак: всем полям формы добавляется значение `'form-control'` http-атрибута `'class'`. Это решает одну из самых больших проблем при работе с формами в Django — их оформления при помощи css-стилей. Зная Python, можно легко разобраться в коде этого приема.

Замечание: за html-представление элементов формы отвечают [виджеты](#) (`field.widget`). Изучите материал более подробно. Это позволит использовать максимум возможностей Django-форм.

Работа с формами Django в шаблонах

Как в **mainapp**, создадим в приложении **authapp** структуру папок **'templates/authapp/'**. И в ней — два шаблона: базовый **base.html** и страницу входа **login.html**.

authapp/templates/authapp/base.html

```
<!DOCTYPE html>
{% load staticfiles %}
<html>
<head>
    <meta charset="utf-8">
    <title>
        {% block title %}
            {{ title|title }}
        {% endblock %}
    </title>
    {% block css %}
        <link rel="stylesheet" type="text/css" href="{% static
'css/bootstrap.min.css' %}">
        <link rel="stylesheet" type="text/css" href="{% static 'css/style.css'
%}">
        <link rel="stylesheet" href="{% static
'fonts/font-awesome/css/font-awesome.css' %}">
    {% endblock %}
    {% block js %}
    {% endblock %}
</head>
<body>
    <div class="container">
        <div class="login">
            <div class="h2 text-center head">
                Вход в систему
            </div>

            {% block content %}

            {% endblock %}
        </div>
    </div>
</body>
</html>
```

С базовым шаблоном все понятно. Прежде чем создавать страницу входа, поговорим о способах вывода формы, которую добавили в контроллере:

- воспользоваться одним из [методов](#): **as_p** (через теги `<p>`), **as_table** (как таблицу), **as_ul** (в виде списка ``);
- поработать с формой как со списком полей и вывести их в цикле.

Первый вариант — самый быстрый и простой, но менее гибкий.

authapp/templates/authapp/login.html

```
{% extends 'authapp/base.html' %}
{% load staticfiles %}

{% block content %}
    <form class="form-horizontal" action="{% url 'auth:login' %}" method="post">
        {% csrf_token %}
        {{ login_form.as_p }}
        <br>
        <input class="form-control" type="submit" value="войти">
    </form>

{% endblock %}
```

Пока остановимся на варианте с методом **as_p()**. Можете самостоятельно попробовать остальные варианты — это несложно.

Видим классические для формы элементы: атрибуты **action** и **method** (конечно, **POST**). Кнопка «Войти» — это обычный элемент `<input>` типа **submit**. Теперь о новом:

- `{% url 'auth:login' %}` — динамическая гиперссылка на адрес контроллера аутентификации. Двоеточием разделены пространство имен **auth** и адрес в нем — **login**;
- `{% csrf_token %}` — токен системы защиты от [межсайтовой подделки запроса CSRF](#). Обновляется при каждой перезагрузке страницы и передается в виде `<input type="hidden">` (обязательно проверьте это в консоли разработчика браузера).

Замечание: если на первых этапах вам будет сильно мешать система защиты CSRF, можете отключить ее на время с помощью константы **MIDDLEWARE** файла настроек проекта и закомментировать элемент списка `'django.middleware.csrf.CsrfViewMiddleware'`.

Теперь можно проверить, как все работает. Запускаем сервер Django. В адресной строке браузера вводим:

```
127.0.0.1:8000/auth/login/
```

Должна появиться форма входа на сайт. Пробуем ввести неправильный логин или пароль. Увидим сообщение об ошибке. Вводим правильную пару <логин> — <пароль> (в нашем проекте это **django** — **geekbrains**). Окажемся на главной странице. Как проверить, что вход выполнен? Попробуйте зайти в админку — должно получиться.

Представляете, сколько рутины выполняет за нас Django? Формирование html-элементов формы, проверка валидности данных, вывод сообщений об ошибках, хеширование пароля, выполнение запроса в БД и так далее.

Еще нужно добавить в главное меню пункт «Войти» и вывести имя пользователя в случае успешного входа. После входа элемент меню «Войти» должен автоматически переименоваться в «Выйти». Вспомним, где находится шаблон главного меню. Мы делали его как подшаблон — значит где-то в **includes** в папке с шаблонами приложения **mainapp**.

mainapp/templates/mainapp/inc_menu.html

```
{% if user.is_authenticated %}
<li>
  <a href="{% url 'auth:edit' %}">
    {{ user.first_name|default:'Пользователь' }}
  </a>
</li>
{% endif %}
<li>
  {% if user.is_authenticated %}
    <a href="{% url 'auth:logout' %}">ВЫЙТИ</a>
  {% else %}
    <a href="{% url 'auth:login' %}">ВОЙТИ</a>
  {% endif %}
</li>
```

Объект пользователя **всегда** есть в шаблонах в переменной **user**. Если почитать [документацию](#), увидим, что у этого объекта есть атрибут **is_authenticated**, который можно использовать в условном операторе. В этом коде мы использовали шаблонный фильтр **default** — он позволяет задать значение по умолчанию, если выводимая переменная не задана. Очень удобно — можно избавиться от громоздкого условного оператора.

Подготовка к реализации механизма CRUD для модели пользователя

В любом Django-проекте при работе с моделями приходится решать классические задачи — создание объектов (**Create**), вывод их на экран (**Read**), корректировка данных (**Update**) и удаление ненужных (**Delete**). Эти действия принято сокращенно называть [CRUD](#) (**Create, Read, Update, Delete**).

В коде шаблона **inc_menu.html** мы уже перебросили мостик к реализации этого механизма при работе с моделью пользователя в проекте:

```
{% url 'auth:edit' %}
```

Но если прямо сейчас запустить сервер Django — увидим ошибку, так как нет адреса с именем **edit** в диспетчере URL приложения **authapp**. Решим эту проблему:

authapp/urls.py

```
...
path('register/', authapp.register, name='register'),
path('edit/', authapp.edit, name='edit'),
...
```

Сразу добавили адрес для контроллера **register()** регистрации пользователя в системе (действие **Create**). Удаление пока не будем рассматривать — реализуем его позже в админке.

При обновлении страницы снова видим ошибку, но уже другую. Цель этих шагов — понять, какие действия нужно предпринять в Django, чтобы реализовать новый функционал.

Итак, ошибка: нет контроллеров. Создадим их:

authapp/views.py

```
...

def register(request):
    return HttpResponseRedirect(reverse('main'))

def edit(request):
    return HttpResponseRedirect(reverse('main'))
```

Пока это контроллеры-заглушки. Логику пропишем позже. Не забываем указать стили для новых страниц в файле **styles.css**. Теперь все должно работать.

Регистрация пользователя

Рассмотрим кратко механизм регистрации (**Create**) пользователя. Первый вопрос — где разместить гиперссылку? Можно на странице входа в систему:

authapp/templates/authapp/register.html

```
{% extends 'authapp/base.html' %}
{% load staticfiles %}

{% block content %}
    <form ...>
    ...
</form>
<button class="btn btn-round form-control">
    <a href="{% url 'auth:register' %}" class="">
        зарегистрироваться
    </a>
</button>
{% endblock %}
```

Надеемся, что динамический url-адрес **'auth:register'** не вызывает вопросов. Если с ним не все ясно, изучите более подробно материал, изложенный выше.

Создаем форму регистрации на базе django-класса **UserCreationForm**:

authapp/forms.py

```
...
from django.contrib.auth.forms import UserCreationForm
...

class ShopUserRegisterForm(UserCreationForm):
    class Meta:
        model = ShopUser
        fields = ('username', 'first_name', 'password1', 'password2', 'email',
                  'age', 'avatar')

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        for field_name, field in self.fields.items():
            field.widget.attrs['class'] = 'form-control'
            field.help_text = ''

    def clean_age(self):
        data = self.cleaned_data['age']
        if data < 18:
            raise forms.ValidationError("Вы слишком молоды!")

        return data
```

Здесь мы явно указали кортеж **fields** с отображаемыми полями формы и применили еще два лайфхака:

- очистили атрибуты **help_text** полей формы, чтобы не было стандартного справочного текста Django (он лишний и громоздкий);
- * добавили [валидатор](#) **clean_age** для атрибута **age** модели, проверяющий возраст пользователя (это материал повышенной сложности, можете пока его пропустить и не делать валидацию).

Теперь у нас есть точка входа на странице и класс формы регистрации — можно заняться контроллером.

authapp/views.py

```
...
from authapp.forms import ShopUserRegisterForm
...
def register(request):
    title = 'регистрация'

    if request.method == 'POST':
        register_form = ShopUserRegisterForm(request.POST, request.FILES)

        if register_form.is_valid():
            register_form.save()
            return HttpResponseRedirect(reverse('auth:login'))
    else:
        register_form = ShopUserRegisterForm()

    content = {'title': title, 'register_form': register_form}

    return render(request, 'authapp/register.html', content)
```

Не забывайте импортировать формы!

Код очень похож на контроллер входа в систему. Из нового — второй аргумент конструктора формы:

```
request.FILES
```

Он необходим, чтобы работала загрузка медиафайлов с формы — это словарь, содержащий информацию о переданных файлах. Если вы работали с PHP, то можете провести аналогию с глобальным массивом `$_FILES`. Но это только полпути — необходимо еще прописать атрибут `enctype="multipart/form-data"` для формы регистрации:

authapp/templates/authapp/register.html

```
{% extends 'authapp/base.html' %}
{% load staticfiles %}

{% block content %}
    <form class="form-horizontal" action="{% url 'auth:register' %}"
method="post" enctype="multipart/form-data">
        {% csrf_token %}
        {{ register_form.as_p }}
        <br>
        <input class="form-control" type="submit" value="зарегистрироваться">
    </form>
{% endblock %}
```

Теперь можно регистрироваться в системе.

Замечание: если в проекте не работает загрузка мета-файлов (например, изображений) — возможные причины:

- ошибка в контроллере — забыли аргумент конструктора формы `request.FILES`;
- ошибка в форме — забыли атрибут `enctype="multipart/form-data"`;
- ошибки в файле конфигурации.

Не забывайте прописывать стили для новых элементов HTML на страницах!

Если этот материал будет для вас сложным — можете вернуться к нему позже, а пока можете создавать пользователей через консоль.

Редактирование пользователя

Совместим просмотр (**Read**) и редактирование (**Update**) объекта пользователя. Получим что-то наподобие личного кабинета.

Точка входа в контроллер была сделана раньше, через уже существующий элемент основного меню, выводящий имя пользователя (адрес `{% url 'auth:edit' %}`). Дальше идем проторенной дорогой: создаем форму, контроллер, шаблоны.

Форма редактирования на базе django-класса `UserChangeForm`:

`authapp/forms.py`

```
...
from django.contrib.auth.forms import UserChangeForm
...

class ShopUserEditForm(UserChangeForm):
    class Meta:
        model = ShopUser
        fields = ('username', 'first_name', 'email', 'age', 'avatar',
'password')

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        for field_name, field in self.fields.items():
            field.widget.attrs['class'] = 'form-control'
            field.help_text = ''
            if field_name == 'password':
                field.widget = forms.HiddenInput()

    def clean_age(self):
        data = self.cleaned_data['age']
        if data < 18:
            raise forms.ValidationError("Вы слишком молоды!")

        return data
```

Здесь использовали еще один полезный маневр — **спрятали поле с паролем**:

```
if field_name == 'password':  
    field.widget = forms.HiddenInput()
```

Контроллер:

authapp/views.py

```
...  
from authapp.forms import ShopUserEditForm  
...  
def edit(request):  
    title = 'редактирование'  
  
    if request.method == 'POST':  
        edit_form = ShopUserEditForm(request.POST, request.FILES,  
instance=request.user)  
        if edit_form.is_valid():  
            edit_form.save()  
            return HttpResponseRedirect(reverse('auth:edit'))  
    else:  
        edit_form = ShopUserEditForm(instance=request.user)  
  
    content = {'title': title, 'edit_form': edit_form}  
  
    return render(request, 'authapp/edit.html', content)
```

Особенность конструктора формы редактирования в Django — необходимо передать сам объект в именованном аргументе:

```
instance=request.user
```

Шаблон.

authapp/templates/authapp/edit.html

```
{% extends 'authapp/base.html' %}
{% load staticfiles %}

{% block content %}
    <form class="form-horizontal" action="{% url 'auth:edit' %}" method="post"
    enctype="multipart/form-data">
        {% csrf_token %}
        {{ edit_form.as_p }}
        <input class="form-control" type="submit" value="сохранить">
    </form>
    <button class="btn btn-round form-control last">
        <a href="{% url 'main' %}" class="">
            на главную
        </a>
    </button>
    <div class="user_avatar"></div>
{% endblock %}
```

Обратите внимание, как выводим аватарку пользователя:

```

```

Остается только настроить стили.

Практическое задание

1. Создать модель пользователя в проекте. Обязательно добавить поле с изображением и возрастом. Выполнить настройки в файле конфигурации.
2. Организовать загрузку данных в БД из файла.
3. Реализовать механизм аутентификации в проекте.
4. Реализовать механизм регистрации пользователя.
5. Организовать просмотр и редактирование данных пользователем.
6. * Разобраться с механизмом валидации данных формы и создать свои валидаторы.

Дополнительные материалы

Все то, о чем сказано в методичке, но подробнее:

1. [Команды управления приложениями.](#)

2. [User model](#).
3. [Своя модель пользователя в Django](#).
4. [Формы Django](#).
5. [Валидация данных формы](#).

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [GET- и POST-запросы](#).
2. [Межсайтовая подделка запроса \(CSRF\)](#).
3. [CRUD](#).