

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ
К ЛАБОРАТОРНЫМ ЗАНЯТИЯМ

Содержание

Лабораторная работа №1.....	3
Основы языка Python.....	
Установка и запуск веб-фреймворка django.....	
Исследование административного интерфейса django.....	
Лабораторная работа №2.....	12
Создание первой web-страницы с простым текстом.....	
Создание первого html-шаблона.....	
Настройка обработки статичных файлов для django.....	
Лабораторная работа №3.....	20
Создание первой модели данных и её регистрация в административном приложении django.....	
Динамическое генерирование шаблона для вывода всех экземпляров этой модели.....	
Лабораторная работа №4.....	25
Создание страницы определенной записи.....	
Верстка обеих страниц в соответствии с макетом.....	
Лабораторная работа №5.....	29
Создание формы и представления для нового поста.....	
Лабораторная работа №6.....	32
Создание формы, шаблона и представления для авторизации.....	
Создание формы, шаблона и представления для регистрации.....	
Лабораторная работа №7.....	34
Изучение основ JavaScript, создание простейших функций и использование базовых операторов.....	
Работа с элементами DOM с помощью JavaScript.....	
Лабораторная работа №8.....	43
Изучение библиотеки jQuery, добавление подсветки для наведенного поста и эффекта для картинки-логотипа.....	
Лабораторная работа №9.....	47

Дальнейшее изучение библиотеки jQuery, добавление эффекта параллакса для главной страницы блога.....	
Список использованной литературы.....	52

Лабораторная работа №1.

Основы языка Python.

Перед началом работы необходимо изучить вторую часть главы 1 учебного пособия [1].

Установка.

С официального сайта www.python.org скачайте интерпретатор языка Python версии 2.7 (неважно, какая цифра в версии будет стоять третьей, на момент написания этих строк, самой свежей версией был выпуск 2.7.6, но можно скачать как более ранние, так и более поздние, например, подойдет и 2.7.3, и 2.7.7). Для ОС Windows установочную программу можно скачать по следующей ссылке: <https://www.python.org/download/releases/2.7.6>

Скачав файл, можно приступить к установке, два раза кликнув по программе установки.

После установки необходимо добавить скрипты языка Python в переменную путей в ОС Windows. Для этого в Windows 7 необходимо перейти в **Свойства моего компьютера** > **Дополнительные параметры системы** > **Переменные среды**, в окошке **«Системные переменные»** найти переменную **PATH** и *добавить в конец* этой переменной следующее значение:

«;C:\Python27;C:\Python27\Scripts;» (начальный символ точки с запятой отделяет предыдущие значения переменной от добавленных). Естественно, чтобы все заработало, необходимо, чтобы Python был установлен в папку **C:\Python27**. В противном случае необходимо указать путь до директории, куда был на самом деле установлен Python.

В корне одного из своих дисков создайте директорию для всех лабораторных работ с подходящим названием на английском языке (например, с названием “programming”, “workflow” или “web-tutorial”). Внимание, наличие нелатинских символов в пути может привести к появлению **ошибок**, поэтому настоятельно не рекомендуется создавать рабочие папки в директориях, где есть нелатинские символы в названиях одного из родителей!

В этой рабочей директории создайте папку с названием “**lab1**”, а внутри этой папки файл “**helloworld.py**”. В последних версиях Windows, программа Проводник не позволяет самому пользователю задавать тип файла, потому что все системные расширения скрыты. Чтобы показать тип файлов, в любой открытой папке нажмите клавишу «Alt», которая заставляет появиться меню вверх папки. Выберите в меню пункт «Сервис», а в открывшемся окне перейдите на вкладку «Вид». Там внизу списка найдите пункт «Скрыть системные расширения» или подобный ему, и уберите с этого пункта галочку.

Откройте редактор кода на языке Python под названием IDLE (Пуск > Все программы > Python 2.7 > IDLE (Python GUI)). Напишите следующую строку в файл

```
print "Hello, world!"
```

Чтобы запустить программу необходимо открыть командную строку (в Windows (Пуск > Все программы > в поле поиска набрать “**cmd**”). В командной строке с помощью команды **cd** (от англ. “change directory”) переместиться в папку, где хранится скрипт. Команда **cd** принимает через пробел аргумент в виде папки, куда нужно перейти. Так что команда для терминала будет выглядеть примерно так:

```
cd C:\path\to\your\lab1
```

,где вместо “**\path\to\your\lab1**” будет находиться путь до вашей папки **lab1**. Также команде **cd** можно передать в качестве аргумента две точки подряд, что означает переход к папке верхнего уровня:

```
C:\work>cd ..  
C:\>
```

Теперь команды будут выполняться в корне диска **C:**, а не в папке **C:\work**.

Когда команда перехода к папке **lab1** выполнена, можно запустить скрипт с помощью команды **python**, которая в качестве аргумента принимает имя файла.

Так что команда

```
python helloworld.py
```

Выведет надпись:

```
Hello, world!
```

Поздравления! Первая программа на питоне запущена успешно. Теперь настало время закрепить полученные в первой главе учебника знания по базовому использованию Python.

Для начала создайте в той же папке **lab1** файл под названием **groupmates.py**. В этом файле создайте список студентов своей факультета (нет, конечно, же не надо описывать всех подряд, для примера достаточно пяти человек из двух-трех групп).

```
groupmates = [  
    {  
        "name": u"Василий",  
        "group": "912-2",  
        "age": 19,  
        "marks": [4, 3, 5, 5, 4]  
    },  
    {  
        "name": u"Анна",  
        "group": "912-1",  
        "age": 18,  
        "marks": [3, 2, 3, 4, 3]  
    },  
    {  
        "name": u"Георгий",  
        "group": "912-2",  
        "age": 19,  
        "marks": [3, 5, 4, 3, 5]  
    },  
    {  
        "name": u"Валентина",  
        "group": "912-1",  
        "age": 18,  
        "marks": [5, 5, 5, 4, 5]  
    }  
]
```

```
}  
]
```

Обязательно заметьте, что

- созданный список сохраняется в переменную `groupmates`;
- каждый элемент списка (иными словами, каждый студент) представляет из себя словарь с четырьмя парами ключ:значение;
- для каждого студента определены: имя, группа, возраст, оценки;
- оценки из себя представляют просто список нескольких чисел от трех до пяти;
- каждое имя студента представляет из себя строку Юникода.

Теперь можно приступить к написанию первой функции - форматированный вывод списка студентов в виде таблицы. Функция будет называться **print_students** и в качестве параметров будет принимать список студентов.

```
def print_students(students):  
    print u"Имя студента".ljust(15), \  
          u"Группа".ljust(8), \  
          u"Возраст".ljust(8), \  
          u"Оценки".ljust(20)  
    for student in students:  
        print \  
            student["name"].ljust(15), \  
            student["group"].ljust(8), \  
            str(student["age"]).ljust(8), \  
            str(student["marks"]).ljust(20)  
    print "\n"
```

Первой операцией **print** на экран выводится заглавие будущей таблицы. Здесь использовался метод строки **ljust()**, который добавляет справа такое число пробелов, что длина строки становится равной переданному **ljust()** аргументу. Без применения этого метода, столбики таблицы не будут находиться ровно друг под другом (попробуйте обязательно создать аналог функции **print_students** без использования **ljust** и посмотрите, какой выведется результат). Символ обратной косой черты говорит о том, что текущая операция **print** продолжится с новой строки. Применение косой черты здесь вынужденное - без неё перечисление всех аргументов для **print** не поместилось бы в одну обозримую на стандартном мониторе строку.

Второй операцией запускается цикл, где построчно выводится вся информация для каждого студента. Здесь также используются метод **ljust()** и символ обратной косой черты. Также необходимо заметить, что нельзя использовать **ljust()** для нестрокового значения (такими значениями как раз являются число в поле **age** и список в поле **marks**), так что пришлось применить операцию приведения к типу **str()** этих двух значений.

Последней операцией выводится просто пустая строка, чтобы новая таблица, если мы будем её добавлять, выглядела отдельно от текущей.

Окончательный вид файл **groupmates.py**:

```
#coding:utf-8
groupmates = [
    {
        "name": u"Василий",
        "group": "912-2",
        "age": 19,
        "marks": [4, 3, 5, 5, 4]
    },
    {
        "name": u"Анна",
        "group": "912-1",
        "age": 18,
        "marks": [3, 2, 3, 4, 3]
    },
    {
        "name": u"Георгий",
        "group": "912-2",
```



```

        "age": 19,
        "marks": [3, 5, 4, 3, 5]
    },
    {
        "name": u"Валентина",
        "group": "912-1",
        "age": 18,
        "marks": [5, 5, 5, 4, 5]
    }
]

def print_students(students):
    print u"Имя студента".ljust(15), \
          u"Группа".ljust(8), \
          u"Возраст".ljust(8), \
          u"Оценки".ljust(20)
    for student in students:
        print student["name"].ljust(15), \
              student["group"].ljust(8), \
              str(student["age"]).ljust(8), \
              str(student["marks"]).ljust(20)
    print "\n"

print_students(groupmates)

```

Задание.

- Напишите функцию фильтрации студентов по средней оценке (так чтобы функция возвращала всех студентов выше заданного в параметрах функции среднего балла). Примерная схема работы функции: создание пустого массива, куда будут добавляться все студенты, прошедшие фильтрацию; запуск цикла, в каждой итерации которого необходимо считать среднюю оценку текущего студента и сравнивать с тем значением, что передано в качестве параметра.

Установка и запуск веб-фреймворка django.

Перед началом работы над этой частью лабораторной работы, внимательно прочтите первую часть главы 1 и главу 2 учебного пособия [1].

С официального сайта проекта django скачайте дистрибутив фреймворка, версия django должна быть 1.4.x, где x - любое число. Версия 1.4.13 доступна по следующей ссылке: <https://www.djangoproject.com/download/1.4.13/tarball/>. Затем необходимо скачанный файл разархивировать с помощью любой доступной вам программы (будь то WinRar или 7-zip). С помощью командной строки зайдите в разархивированную папку с проектом django, там в числе прочего должен находиться файл **setup.py**. Теперь через команду **python** необходимо запустить этот файл, передав ему дополнительным параметром ключевое слово “**install**”. Иными словами в папке с проектом django необходимо выполнить команду:

```
python setup.py install
```

После этого в командную строку будут выведено множество строк, описывающих текущий этап установки веб-фреймворка. Когда команда будет выполнена, необходимо протестировать установленный проект. Для этого через командную строку зайдите в оболочку Python (достаточно просто команды **python**, без указания аргументов). Должны появиться три символа “>”, приглашающие к введению инструкций на языке Python:

```
C:\work>python
>>>
```

Теперь все выполняемые команды будут обрабатываться интерпретатором Python. Попробуйте импортировать библиотеку django:

```
>>> import django
```

Если после введенной команды не появилось никаких ошибок, значит фреймворк был установлен успешно, поздравления! Можно также проверить текущую версию django с помощью команды:

```
>>> django.get_version()
```

Теперь настало время создать свой первый проект на django. Для этого в директории lab1 с помощью командной строки выполните команду (не забудьте перед этим выйти из интерпретатора **python** с помощью функции «**exit()**»)

```
django-admin.py startproject admin_learning
```

Эта команда создаст папку `admin_learning`, в которой находится основа будущего проекта, структура которой показана ниже:

```
admin_learning/  
  __init__.py  
  manage.py  
  settings.py  
  urls.py
```

- **__init__.py**: Файл необходим для того, чтобы Python рассматривал данный каталог как пакет, т.е., как группу модулей. Это пустой файл и обычно вам не требуется добавлять что-либо в него.
- **manage.py**: Это утилита командной строки, которая позволяет вам взаимодействовать с проектом различными методами. Наберите `python manage.py help` для получения информации о возможностях утилиты. Вы не должны изменять содержимое данного файла, он создан в данном каталоге в целях удобства.
- **settings.py**: Настройки для текущего проекта Django. Посмотрите на содержимое файла, чтобы иметь представление о типах доступных параметров и их значениях по умолчанию.
- **urls.py**: Описания URL для текущего проекта Django, так сказать «оглавление» для вашего сайта.

Эти файлы и составляют ядро Django-проекта, которое обозначено красным прямоугольником в базовой схеме, изображенной на рисунке 1.1.

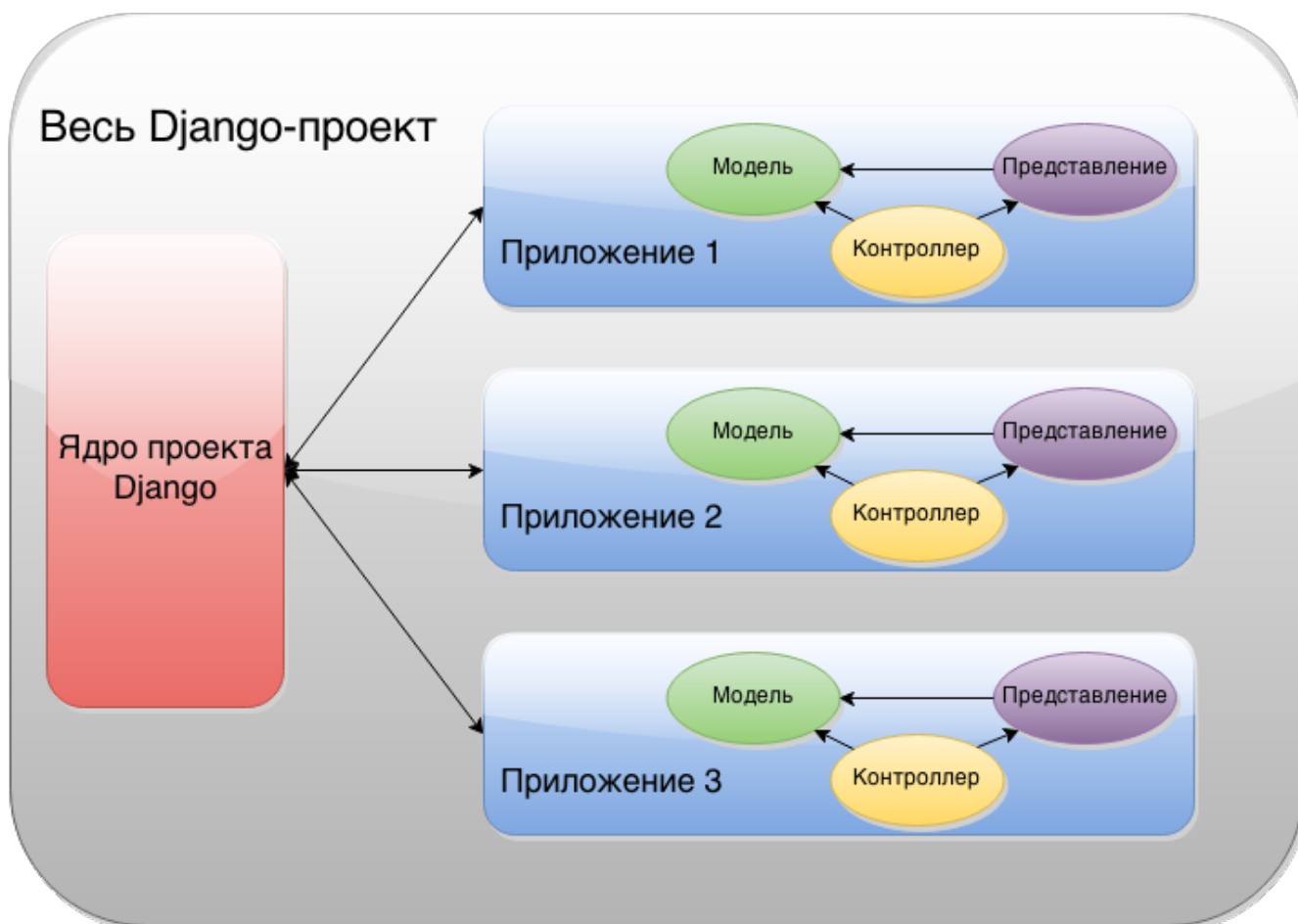


Рисунок 1.1. Структура базового django-проекта с тремя приложениями.

Для проверки правильности установки Django запустите сервер разработки, чтобы посмотреть на созданное приложение в действии.

Сервер разработки Django (также называемый «**runserver**», по имени команды, которая его запускает) — это встроенный лёгкий веб сервер, который вы можете использовать в процессе разработки вашего сайта. Он включен в Django для того, чтобы вы могли быстро приступить к разработке вашего сайта без траты времени на конфигурирование вашего боевого веб сервера (т.е., Apache) раньше времени. Этот сервер разработки отслеживает изменения в вашем коде и автоматически перезагружает его, помогая видеть вносимые вами изменения без перезагрузки веб сервера.

Для запуска сервера перейдите в каталог `admin_learning` (“**cd admin_learning**”), если вы ещё не сделали этого, и выполните команду:

```
python manage.py runserver
```

Вы увидите нечто подобное этому:

```
Validating models...  
0 errors found.
```

```
Django version 1.0, using settings 'admin_learning.settings'  
Development server is running at http://127.0.0.1:8000/  
Quit the server with CONTROL-C.
```

Команда запускает сервер локально на порту 8000. Сервер принимает только локальные соединения с вашего компьютера. Теперь он запущен, посетите страницу <http://127.0.0.1:8000/> с помощью браузера. Вы увидите страницу «Welcome to Django». Всё работает! [2]

Задание:

- Откройте файл **settings.py** через редактор исходного кода IDLE (он уже установлен вместе с интерпретатором Python, как его открыть, объяснялось выше). Внимательно изучите все существующие там переменные и переведите описание каждой из них. Зубрить их нет необходимости, но примерно представлять совокупность всех настроек по умолчанию вам нужно.
- В файле **settings.py** настройте два параметра для базы данных. В переменной-словаре **DATABASES** есть ключ **default**, по которому содержится ещё один внутренний словарь с ключами **ENGINE** и **NAME**. Установите следующие значения этим параметрам:
 - **ENGINE**: `'django.db.backends.sqlite3'`
 - **NAME**: `'db_admin_learning'`
- Теперь для вашего проекта в качестве базы данных подключена **sqlite3**, а название новосозданной базы будет **“db_admin_learning”**.

Исследование административного интерфейса django.

Чтобы войти в административный интерфейс django необходимо обладать правами суперпользователя, а для этого в свою очередь необходимо создать все нужные таблицы в базе данных (потому что все данные о пользователях, их никнеймы, пароли и др. содержатся в соответствующих таблицах баз данных). Чтобы создать таблицы, достаточно в папке `admin_learning` выполнить через командную строку выполнить команду:

```
python manage.py syncdb
```

Во время выполнения команда спросит вас о том, нужно ли создавать суперпользователя, на что вам нужно ответить утвердительно (набрать слово “yes” и нажать клавишу Enter). После этого будет предложено придумать никнейм, указать вашу электронную почту и создать пароль. Когда вы выполните указанные шаги, таблицы в базе данных будут созданы, и в вашем проекте появится один суперпользователь.

Однако этого ещё недостаточно, необходимо указать django, что в текущем проекте надо разрешить административный интерфейс. Для этого в файле настроек **settings.py** нужно раскомментировать элемент у кортежа **INSTALLED_APPS** - 'django.contrib.admin'. Найдите этот элемент в кортеже и уберите стоящую в начале строки решетку “#”. Теперь переменная **INSTALLED_APPS** в вашем файле настроек должен выглядеть примерно так:

```
INSTALLED_APPS = (  
    'django.contrib.auth'  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.sites',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'django.contrib.admin',  
)
```

Затем пройдите в файл **urls.py** и там раскомментируйте несколько строк:

```
# from django.contrib import admin  
# admin.autodiscover()
```

В этом же файле **urls.py** внутри вызова функции `urlpatterns = patterns()` раскомментируйте параметр

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

Теперь необходимо запустить сервер, если он не был запущен до этого, командой:

```
python manage.py runserver
```

и в браузере пройти по адресу <http://127.0.0.1:8000/admin/>.

Должно появиться окно входа, как на рисунке 1.2.

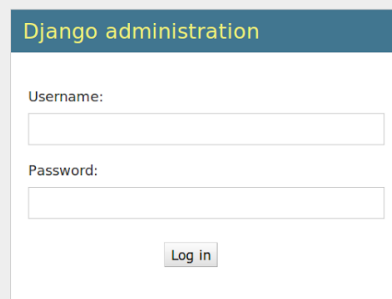


Рисунок 1.2. Окно входа в панель административного интерфейса.

Сюда необходимо ввести данные, которые вы указали при создании суперюзера. Когда вы успешно войдете в аккаунт суперюзера, перед вами откроется главная страница административной панели, через которую вы можете управлять вашими приложениями, редактируя существующие записи в базе данных или генерируя новые. Скриншот этой странице вы можете увидеть на рисунке 1.3.

Site administration

Auth		
Groups	Add	Change
Users	Add	Change

Recent Actions
My Actions
None available

Рисунок 1.3. Скриншот страницы административного интерфейса Django.

Пройдите по ссылке **Users**, чтобы увидеть список всех зарегистрированных на вашем сайте пользователей.

Задание:

- изучите интерфейс административного приложения django;
- через интерфейс административного приложения создайте нового пользователя с правами суперюзера;
- через интерфейс административного приложения создайте нового пользователя без прав суперюзера;
- через интерфейс административного приложения «забаньте» одного из пользователей (сделайте пользователя «неактивным»).

Лабораторная работа №2.

Создание первой web-страницы с простым текстом.

С помощью команды

```
django-admin.py startproject helloworld
```

, которая была описана выше, создайте в папке **lab2** проект под названием **helloworld**. Теперь в папке **helloworld** выполните команду

```
python manage.py startapp flatpages
```

, которая создаст внутри проекта папку **flatpages**. Как было описано в теории, любой реальный джанго проект состоит из одного или нескольких приложений, каждое из которых выполняет свои узконаправленные задачи. В данном случае приложение **flatpages** будет обрабатывать запросы на статичные html-страницы и возвращать их пользователю.

Для новосозданного проекта необходимо в первую очередь задать базовые настройки - указать предпочитаемую базу данных, разрешить административное приложение (как это сделать, описано в первой лабораторной работе) и добавить приложение **flatpages** в проект. В качестве настроек базы данных укажите:

```
ENGINE: 'django.db.backends.sqlite3'  
NAME: 'db_helloworld'
```

Чтобы добавить приложение **flatpages** в проект, необходимо в кортеж **INSTALLED_APPS**, который определяется в файле **settings.py**, добавить в конец элемент - строку “**flatpages**”. Так что теперь, не учитывая комментариев, кортеж установленных приложений будет выглядеть примерно так:

```
INSTALLED_APPS = (  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.sites',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'django.contrib.admin',  
    'flatpages'
```

)

Для будущей страницы создайте новый адрес в файле **urls.py**:

```
urlpatterns = patterns('',
    url(r'^$', 'flatpages.views.home', name='home'),
    url(r'^admin/', include(admin.site.urls)),
)
```

- так теперь должен примерно выглядеть код файла, не включая операторов импортирования в начале. Первый аргумент, который был передан функции **patterns**, говорит о том, что адрес страницы будет самым простым - он будет пустым, а второй аргумент указывает, что функция представления, которая будет обрабатывать запросы по этому адресу, находится в файле **flatpages.views** и называется **home**. Необходимо это представление написать, оно в будущем будет генерировать ответ. Для этого в файле **flatpages/views.py** создайте функцию **home**.

```
#coding:utf-8
from django.http import HttpResponse

def home(request):
    return HttpResponse(u'Привет, Мир!', mimetype="text/plain")
```

Сначала происходит импортирование специального класса **HttpResponse**, любая функция представления должна возвращать переменные, где хранятся представители именно этого класса или класса, наследующего **HttpResponse**. В данном случае функция представления **home** является типичными представителем представлений в django - она принимает на входе объект запроса и возвращает на выходе объект ответа. В нашем случае в теле объекта ответа будет находиться простая строка **'Привет, Мир!'**, а тип ответа указан самый банальный - просто текст. Теперь можно зайти на страницу по адресу <http://127.0.0.1:8000/> и увидеть надпись

Привет, Мир!

Первая страница была создана, теперь её можно немного усложнить, добавив заголовок и немного текста.

Задание:

- Сделайте так, чтобы по адресу <http://127.0.0.1:8000/hello/> тоже возвращался тот же самый текст.
- Уберите указание типа возвращаемого ответа (если напрямую классу **HttpResponse** не указать тип ответа, будет установлено значение по умолчанию). Сравните полученные результаты.

Создание первого html-шаблона.

Для выполнения следующего задания внимательно прочтите главу 3 учебного пособия [1].

Создайте папку **templates** внутри вашего приложения **flatpages**, внутри которого создайте файл **index.html**. Теперь структура папки **flatpages** будет выглядеть примерно следующим образом:

```
flatpages/
  templates/
    index.html
  __init__.py
  models.py
  tests.py
  views.py
```

Внутри файла **index.html** добавьте код:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Привет, Мир!</title>
  </head>
  <body>
    <h1>Привет, Мир!</h1>
    <h2>Это учебный сайт, с его помощью будут изучены технологии
python/django, html/css.</h2>
    <h3>Как видите, здесь используются заголовки различных
уровней.</h3>
    <p>Здесь есть маркированный список:</p>
    <ul>
      <li>Элемент 1;</li>
      <li>элемент 2;</li>
      <li>элемент 3;</li>
      <li>последний элемент.</li>
    </ul>
    <p>И нумерованный список:</p>
    <ol>
      <li>Элемент 1;</li>
```

```

        <li>элемент 2;</li>
        <li>элемент 3;</li>
        <li>последний элемент.</li>
    </ol>
    <p>И даже таблица:</p>
    <table border="1">
        <thead>
            <tr>
                <th>Столбик 1</th>
                <th>Столбик 2</th>
                <th>Столбик 3</th>
            </tr>
        </thead>
        <tr>
            <td>Строка 1 Столбец 1</td>
            <td>Строка 1 Столбец 2</td>
            <td>Строка 1 Столбец 3</td>
        </tr>
        <tr>
            <td>Строка 2 Столбец 1</td>
            <td>Строка 2 Столбец 2</td>
            <td>Строка 2 Столбец 3</td>
        </tr>
        <tr>
            <td>Строка 3 Столбец 1</td>
            <td>Строка 3 Столбец 2</td>
            <td>Строка 3 Столбец 3</td>
        </tr>
    </table>
</body>
</html>

```

Теперь необходимо созданный html-файл подключить к функции-представлению **home**, чтобы вместо простого текста в ответ на запрос приходил html-документ. Для этого в файле **flatpages/views.py** добавьте операцию импортирования:

```
from django.shortcuts import render
```

Функция **render** позволяет создавать html-документы из различных файлов, вставляя на нужное место необходимые переменные. Но о переменных речь идти будет позже, пока что просто нужно подключить внешний html-файл, который был только что создан. Перепишите представление **home** следующим образом:

```
def home(request):
    return render(request, 'index.html', {})
```

Как видно из примера, функция **render** принимает три параметра: объект запроса, который является аргументом функции-представления, имя шаблона, который будет найден в директории **templates** приложения и словарь, куда можно сохранить переменные для использования их в шаблонах, но об этом аргументе речь пойдет позже. Сейчас, когда представление и шаблон готовы можно посмотреть на внесенные изменения, для этого достаточно перезагрузить страницу с адресом <http://127.0.0.1:8000/>.

Задание:

- Добавьте к созданной таблице две строки и один столбец.
- Уберите у созданной таблицы все границы.
- Сделайте заголовки списков (нумерованного и маркированного) подзаголовками четвертого уровня.
- Создайте абсолютно такой же шаблон, только поменяв название на “**static_handler.html**”. В следующей главе лабораторной работы изменяйте именно новосозданный шаблон.

Настройка обработки статичных файлов для django.

Работа статичных файлов описана в главе 4 учебного пособия [1].

Внимательно прочтите её перед следующим заданием.

Теперь нужно приукрасить документ, придав ему некоторый стиль в соответствии с макетом. Для начала нужно подключить внешний css-файл, который должен храниться в папке **static** проекта. Создайте папку **static** с директорией **css**, а внутри этой директории добавьте пустой файл **index.css**.

Структура приложения **flatpages** будет выглядеть примерно так:

```
flatpages/  
  static/  
    index.css  
  templates/  
    index.html
```

```
static_handler.html
__init__.py
models.py
tests.py
views.py
```

После этого в файл страницы **static_handler.html** внутрь тега **<head>** вставьте тег подключения CSS-скрипта (сразу после тега **<title>**).

```
<link rel="stylesheet" href="{ STATIC_URL }css/index.css">
```

В шаблоне использовалась переменная **STATIC_URL**, которая автоматически добавляется к контексту любого шаблона, если для рендеринга пользоваться функцией **render**. Как указывалось в теоретическом пособии, любая переменная, используемая в шаблоне, должна быть обрамлена двумя фигурными скобками с обеих сторон.

Теперь укажите несколько стилей для созданного сайта:

```
body {
    background: #1abc9c;
    font-family: Tahoma, Arial, sans-serif;
    color: #333;
}
table {
    border-collapse: collapse;
}
p, h4 {
    font-size: 22px;
    margin-bottom: 0;
}
ul, ol {
    margin: 0
}
table tr td {
    padding: 5px
}
```

Несколько пояснений:

body {...} - все, что находится на месте многоточия в фигурных скобках будет применено к тегу **<body>**.

Свойство **background** задает фоновый цвет для элемента, в данном случае цвет задан в формате HEX (то есть три идущих подряд двузначных шестнадцатеричных числа, каким бы страшным не показалось это определение, здесь совсем ничего сложного).

Свойство **font-family** задает семейство шрифтов, которые будут применены к элементам. Желательно каждому элементу указывать как минимум несколько семейств, потому что в случае, если первый шрифт не подключается или не поддерживается браузером пользователя, будет произведена попытка подключить второй шрифт. Если подключение и второго шрифта потерпит неудачу, настанет черед третьего и так далее. По этой причине список семейств должен заканчиваться тем шрифтом, который точно присутствует в браузере (в противном случае браузер применит стиль по своему усмотрению), и здесь есть два варианта - либо шрифт без засечек **sans-serif**, либо шрифт с засечками **serif**, ведь оба этих семейства поддерживаются абсолютно всеми популярными браузерами. В данном случае по дизайну страницы должен использоваться шрифт без засечек.

Свойство **color** задает цвет шрифта.

Свойства **border-collapse**, **font-size**, **margin** и **padding** и дальнейшие применяемые свойства остаются на самостоятельное изучение.

Когда перед фигурными скобками указано несколько элементов через запятую, значит стиль применится ко всем перечисленным элементам.

Пробел при перечислении элементов указывает на вложенность, грубо говоря, элемент до пробела является родителем элемента после пробела. Соответственно, выражение **table tr td** означает: применить стиль к элементам **td**, которые находятся внутри **tr**, которые в свою очередь находятся внутри тега **table**.

Также к шаблону необходимо будет добавить картинку - логотип вашего будущего сервиса, которую вы можете найти по имени **dpb-logo.png** в файлах, прикрепленных к данному учебному пособию. Для этого после третьего заголовка (тега **<h3>**) сразу добавьте следующую строку:

```

```

В итоге должен получиться сайт, изображенный на рисунке 2.1.

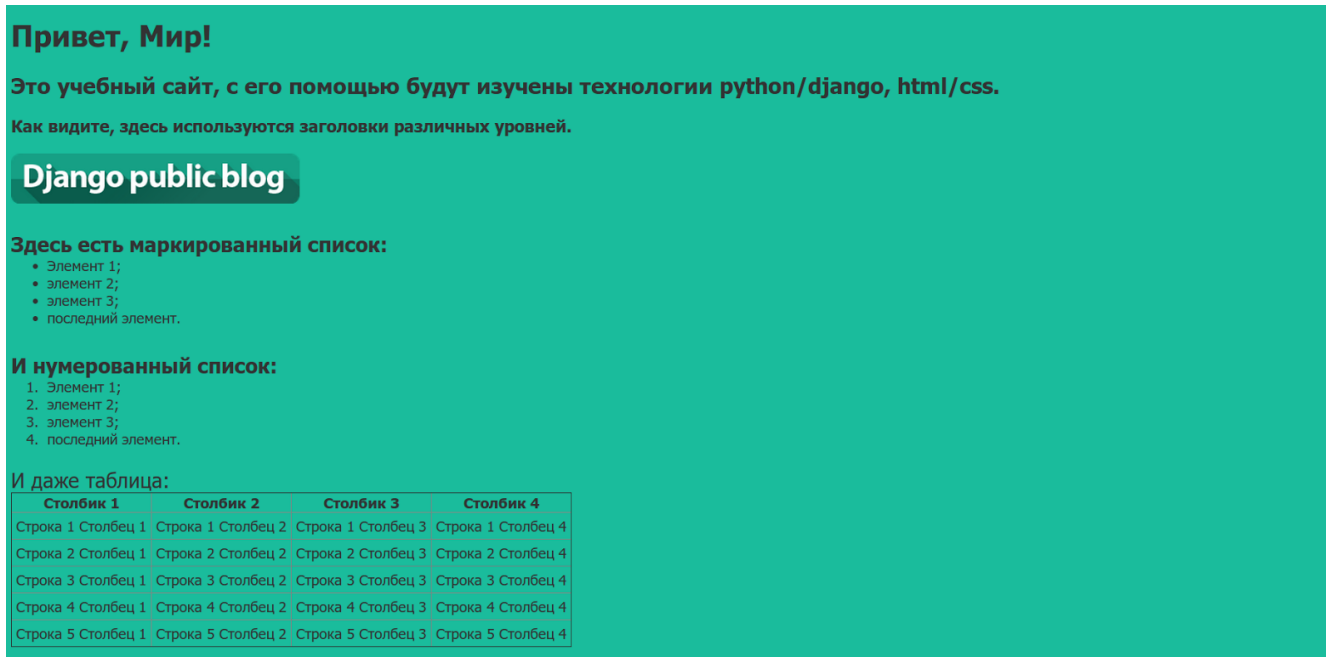


Рисунок 2.1. Пример созданной верстки для первой HTML-странички.

Задание:

- Установите для заголовка первого уровня шрифт с засечками.
- Сделайте картинку высотой 30px.
- Измените шрифт для подзаголовков четвертого уровня с 22px на 14px.
- Сделайте ширину таблицы равно 100% экрана.

Лабораторная работа №3.

Создание первой модели данных и её регистрация в административном приложении django.

Работа моделей была подробно описана в части первой главы 5 учебного пособия [1].

С помощью команды

```
django-admin.py startproject blog
```

создайте в папке **lab3** проект под названием **blog**. Теперь в папке **blog** выполните команду

```
python manage.py startapp articles
```

Это создаст в вашем проекте blog новое приложение. Выполните базовую настройку проекта, как это было проделано в начале предыдущей лабораторной работы.

Откройте файл **articles/models.py**, в котором будут храниться будущие модели статей. Напишите в этом файле следующее:

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

class Article(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey(User)
    text = models.TextField()
    created_date = models.DateField(auto_now_add=True)

    def __unicode__(self):
        return "%s: %s" % (self.author.username, self.title)

    def get_excerpt(self):
return self.text[:140] + "..." if len(self.text) > 140 else self.text
```

Будущая модель статей будет иметь четыре поля, из которых в одном (время создания) значение будет устанавливаться автоматически. Также в одном поле присутствует ссылка на соседнюю таблицу, где будут храниться записи о зарегистрированных пользователях.

Метод `__unicode__` ответственен за отображение текущего экземпляра модели. В том числе этот метод вызывается, когда экземпляр модели будет передан функции (или оператору) `print`. Например, при отсутствии метода `__unicode__`, следующая команда:

```
print Article.objects.get(id=1)
```

выведет строку:

```
Article object
```

Такой вывод малоинформативен, особенно по сравнению с тем случаем, когда метод `__unicode__` был реализован способом, описанным в примере выше:

```
print Article.objects.get(id=1)
```

который выведет:

```
Konstantin: Первая статья
```

Метод же `get_excerpt` будет использоваться в административном интерфейсе, который будет сейчас создан.

В директории `articles` создайте файл `admin.py`, который будет ответственен за настройку страницы записей в административном приложении. В первой лабораторной административное приложение уже было исследовано, в частности необходимо было разобраться со страницей управления пользователями, которая имела множество функциональных возможностей. Чтобы повторить хотя бы часть из этого функционала, в файл `admin.py` напишите следующий код:

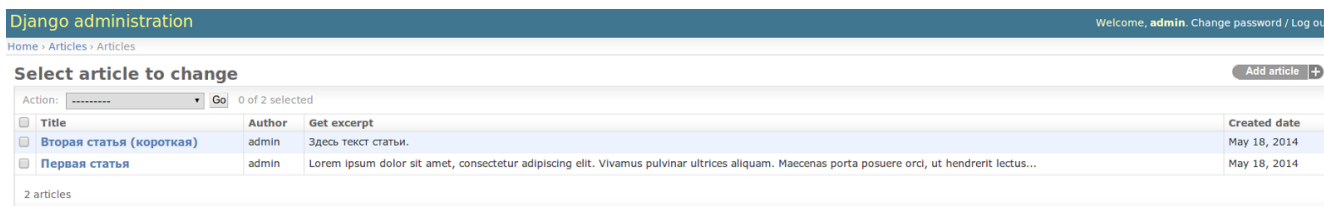
```
from django.contrib import admin
from models import Article

class ArticleAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'get_excerpt', 'created_date')

admin.site.register(Article, ArticleAdmin)
```

Класс **ArticleAdmin** нужен для того, чтобы в [декларативном стиле](#) описать то, каким образом модель **Article** должна отображаться в административной панели (в данном случае просто описывается, какие поля необходимо использовать при отображении списка сразу нескольких статей). Естественно, это описание не должно содержаться в коде самой модели, потому что сама суть модели (например, поля модели и её методы) не надо смешивать с тем, как эта модель должна отображаться на страницах панели.

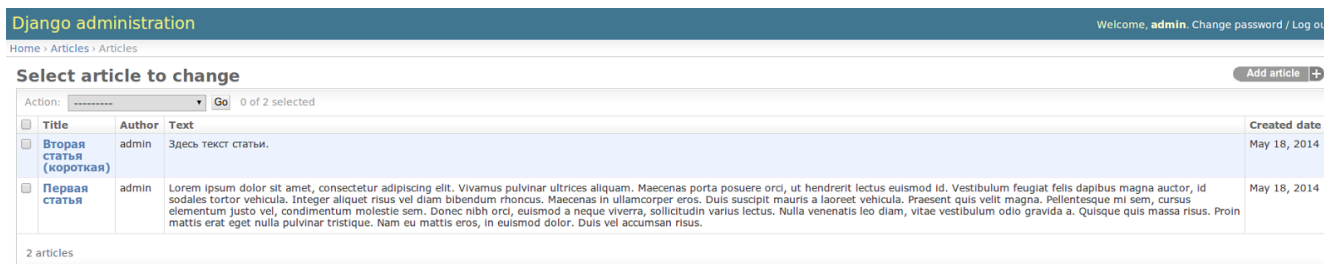
Вот здесь и пригодился созданный ранее метод `get_excerpt`, который позволяет в списке всех статей приводить не целиком текст всей статьи, а показывать лишь первые 140 символов. Различия между использованием метода `get_excerpt` и его неиспользованием показаны на рисунках 3.1 и 3.2.



The screenshot shows the Django administration interface for the 'Articles' app. The header includes 'Django administration' and a welcome message for 'admin'. The breadcrumb trail is 'Home > Articles > Articles'. The main heading is 'Select article to change'. Below it, there's a table with two columns: 'Title' and 'Text'. The first row is 'Вторая статья (короткая)' with the text 'Здесь текст статьи.'. The second row is 'Первая статья' with a long Lorem Ipsum text. The 'Created date' column shows 'May 18, 2014' for both. The table is titled '2 articles'.

Title	Author	Get excerpt	Created date
Вторая статья (короткая)	admin	Здесь текст статьи.	May 18, 2014
Первая статья	admin	Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus pulvinar ultrices aliquam. Maecenas porta posuere orci, ut hendrerit lectus...	May 18, 2014

Рисунок 3.1. Список записей с использованием метода `get_excerpt`.



The screenshot shows the Django administration interface for the 'Articles' app, similar to the previous one but without the 'Get excerpt' column. The table has columns 'Title', 'Author', 'Text', and 'Created date'. The first row is 'Вторая статья (короткая)' with the text 'Здесь текст статьи.'. The second row is 'Первая статья' with a long Lorem Ipsum text. The 'Created date' column shows 'May 18, 2014' for both. The table is titled '2 articles'.

Title	Author	Text	Created date
Вторая статья (короткая)	admin	Здесь текст статьи.	May 18, 2014
Первая статья	admin	Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus pulvinar ultrices aliquam. Maecenas porta posuere orci, ut hendrerit lectus euismod id. Vestibulum feugiat felis dapibus magna auctor, id sodales tortor vehicula. Integer aliquet risus vel diam bibendum rhoncus. Maecenas in ullamcorper eros. Duis suscipit mauris a laoreet vehicula. Praesent quis velit magna. Pellentesque mi sem, cursus elementum justo vel, condimentum molestie sem. Donec nibh orci, euismod a neque viverra, sollicitudin varius lectus. Nulla venenatis leo diam, vitae vestibulum odio gravida a. Quisque quis massa risus. Proin mattis erat eget nulla pulvinar tristique. Nam eu mattis eros, in euismod dolor. Duis vel accumsan risus.	May 18, 2014

Рисунок 3.2. Список записей без использования метода `get_excerpt`.

В конце файла был произведен вызов функции **admin.site.register()**, которой были переданы два параметра - модель статей и класс, описывающий, как модель должна отображаться в административном интерфейсе. Эта функция объявляет, что данная модель должна быть добавлена в административный интерфейс.

Задание:

- Когда модель зарегистрирована, можно создавать новые записи, достаточно перейти в административную панель, находящуюся по адресу <http://127.0.0.1:8000/admin/> (если вы настроили все так, как django устанавливает по умолчанию). Перейдите по вкладке Articles и кликните по кнопке “Add article” справа наверху. Заполните поля Title, Author и Text значениями по своему усмотрению. Создайте ещё две статьи подобным образом.
- С помощью программы управления базами данных sqlite3 (например, программы SQLiteManager) откройте файл базы данных текущего проекта, который хранится в папке проекта с именем, объявленным в настройках проекта в переменной «**DATABASES.NAME**». Найдите созданные в предыдущем пункте задания экземпляры записей. Измените текст одной записи и название статьи для другой.

Динамическое генерирование шаблона для вывода всех экземпляров этой модели.

Принципы работы с шаблонами были описаны в главе 3 учебного пособия [1], а про то, как создавать представления и конфигурировать URL, можно прочитать в части второй главы 5. Перед началом лабораторной работы внимательно ознакомьтесь с указанным материалом.

В директории **articles** создайте папку **templates**, внутри которой создайте файл **archive.html**.

В файле шаблона в качестве названия странице (тег **<title>**) укажите фразу “Архив всех статей”.

Затем в тег **<body>** добавьте два тега **<div>**:

```

<body>
  <div class="header">
  </div>
  <div class="archive">
  </div>
</body>

```

Как верно подметил внимательный читатель, у первого тега **div** установлен класс **header**, а у второго класс **archive**. Это помогает, во-первых, делать верстку понятной (потому что классы играют роль имен для каждого элемента, из названия класса становится понятно, для чего существует текущий элемент), а во-вторых, отличать нужные блоки друг от друга при установке стилей.

Вторая причина особенно важна, потому что простой стиль

```
div {background: green}
```

будет присвоен абсолютно всем тегам **div**, что частно совсем противоречит внешнему виду сайта.

Внутри **div**-а с классом **header** добавьте картинку-логотип проекта

```

```

Внутри **div**-а с классом **archive** нужно добавить шаблон для отображения одной статьи. Как известно, у каждой записи есть: название, имя автора, время создания и текст. Поэтому для каждого поля у записи нужно создать определенный элемент в разметке страницы. Пусть шаблон одного поста выглядит примерно следующим образом:

```

<div class="one-post">
  <h2 class="post-title">{{ post.title }}</h2>
  <div class="article-info">
    <div class="article-author">{{ post.author.username }}</div>
    <div class="article-created-date">{{ post.created_date }}</div>
  </div>
  <p class="article-text">{{ post.get_excerpt }}</p>
</div>

```

Как можно заметить из разметки, поле название было помещено в тег `<h2>` с классом **post-title**, поля имени автора и времени создания были помещены в один общий тег `<div>`, потому что в будущем эти два поля будут визуально находиться на одной строке (только поле имени будет слева, а поле времени создания будет справа). Поле же текста (а точнее выдержки из текста, длиной 140 символов) было обрамлено тегом **p**, который означает один абзац текста (p - сокращенный вариант английского слова “*paragraph*”). Решение отображать не весь текст статьи обосновано тем, что некоторые посты могут по размерам занимать несколько страниц, что, естественно, недопустимо при отображении списка сразу многих экземпляров.

Однако проблема сейчас заключается в том, что созданная разметка подходит для отображения одной статьи, а не нескольких. Необходимо добавить цикл, который бы повторялся столько раз, сколько статей передано в контекст шаблона. Для этого существует шаблонный тег `{% for item in list %}`, который по своему поведению и внешнему виду максимально повторяет цикл **for** языка Python. В данном случае роль массива будет играть переменная **posts**, передаваемая в шаблон. Соответственно, чтобы разметка поддерживала отображение сразу многих записей, достаточно добавить строку начала цикла и строку его завершения:

```
{% for post in posts %}
    <div class="one-post">
        <h2 class="post-title">{{ post.title }}</h2>
        <div class="article-info">
            <div class="article-author">{{ post.author.username }}</div>
            <div class="article-created-
date">{{ post.created_date }}</div>
        </div>
        <p class="article-text">{{ post.get_excerpt }}</p>
    </div>
{% endfor %}
```

Теперь шаблон полностью готов.

В файле **articles/views.py** создайте представление **archive**, которое будет возвращать html-страницу со всеми написанными постами к текущему проекту.

```
from models import Article
from django.shortcuts import render
```

```
def archive(request):
    return render(request, 'archive.html', {"posts":
Article.objects.all()})
```

В отличии от предыдущей лабораторной работы, здесь был использован последний аргумент функции **render**. Именно в этом аргументе было указано, что в шаблоне должна быть определена переменная с именем **posts**, а значение этой переменной должно быть равно массиву всех записей, сохраненных в базе данных. Именно благодаря этой переменной стал возможным вывод всех записей в шаблонном теге **{% for %}**.

Осталось лишь настроить **url**, по которому будут отображаться все статьи проекта. С этим заданием вы способны справиться самостоятельно, указав **django**, что при заходе пользователя на главную страницу нужно отображать список всех записей.

Задание:

- Вновь откройте файл базы данных **sqlite3**, где хранятся экземпляры статей к текущему проекту. Добавьте новую запись в блог непосредственно через менеджер базы данных (не забудьте зафиксировать транзакцию, иными словами, сохранить все внесенные изменения).

Лабораторная работа №4.

Создание страницы определенной записи.

Если у вас возникнут какие-либо вопросы по данной лабораторной работе, внимательно перечитайте часть 2 главы 5 учебного пособия [1].

В отличие от создания страницы для всех постов, реализация страницы определенного поста пройдет в обратном порядке. Сначала необходимо будет создать адрес будущей страницы, затем написать функцию представление для неё и в конечном итоге реализовать шаблон.

Адрес страницы для одной записи обязательно должен нести в себе какое-то ключевое слово, чтобы можно было среди всей таблицы базы данных найти интересующий пользователя экземпляр. Для этого можно использовать **id** сообщений, которые устанавливаются автоматически при создании каждого отдельного объекта сообщений. В итоге адрес страницы для статьи с **id=5** должен выглядеть так: <http://127.0.0.1:8000/article/5/>.

Для того, чтобы было легко извлекать данные из адресов, по которым приходят запросы на сервер, в django используются регулярные выражения. Они уже использовались в предыдущих проектах, но были очень простыми - там присутствовали лишь символы начала регулярного выражения “^” и его завершения “\$”. В итоге получавшийся URL

```
url(r'^$', 'articles.views.archive', name='archive')
```

означал, что адрес не продолжается после указания хоста и порта соединения (в данном случае хостом и портом являются “127.0.0.1:8000/”).

Для начала в папке **lab4** создайте новый проект под названием **blog**, настройте его и перенесите приложение **articles** из предыдущего проекта (для переноса приложения просто скопируйте его папку проекта, добавьте название приложения в **INSTALLED_APPS** и добавьте необходимые настройки адресов в файл **urls.py**).

Итак, для того, чтобы можно было прочитать значение **id** интересующего пользователя поста, следует в качестве адреса указать следующее регулярное выражение (естественно, внутри файла **urls.py**):

```
url(
    r'^article/(?P<article_id>\d+)$',
    'articles.views.get_article',
    name='get_article'
)
```

В качестве первого аргумента переданно регулярное выражение, которое возвращает именованную группу **article_id** (именованной группой как раз называют переменные внутри регулярного выражения, которые имеют следующий синтаксис: “(?P<name> ...)”, где **name** вы должны указать сами, а вместо троеточия подставить нужный вам шаблон). Заметьте, что именованная группа идет после символов “**article/**”, и состоит она только из цифр, продолжающихся до завершения адреса. Соответственно, если хоть одно из условий не подходит (например, адрес начинается с “**artikle/**” вместо “**article/**”, или та комбинация, что идет после “**article/**” состоит не только из цифр, но и из латинских букв), то регулярное выражение не подойдет и представление “**articles.views.get_article**” не запустится.

После того, как в регулярном выражении была объявлена именованная группа, в представление можно добавлять именованный аргумент, куда и будет передано значение из адреса, по которому перешел пользователь. Так что создайте в файле **articles/views.py** помимо уже написанной в предыдущей функции **archive** ещё и функцию **get_article**:

```
from django.http import Http404

def get_article(request, article_id):
    try:
        post = Article.objects.get(id=article_id)
        return render(request, 'article.html', {"post": post})
    except Article.DoesNotExist:
        raise Http404
```

Как видите, здесь использовался синтаксис обработки исключений, который помогает вручную обрабатывать возникающие ошибки, главное, чтобы ошибка была именно той, что указано после ключевого слова “**except**”. В данном случае, если возникла ошибка **Article.DoesNotExist**, что говорит об отсутствии элемента с указанным **id**, который искался через функцию **get()**, возбуждается другая ошибка, уже системная, которая будет обработана автоматически платформой django - ошибка **Http404**. В случае возникновения исключения **Http404**, django просто вернет стандартную страницу ошибки 404 (“Страница не найдена”), которую хоть раз, да видел, пожалуй, всякий активный пользователь Интернета. Однако не забудьте эту ошибку **Http404** сначала импортировать, как было сделано в первой строке примера.

Задание:

- Теперь, когда есть страница определенной записи, можно на странице списка постов каждый заголовок сделать ссылкой. Сделайте так, чтобы при клике по названию происходил переход на страницу указанной записи. Для этого достаточно воспользоваться тегом **<a>**, которому в атрибуте **href** указать адрес перехода.

Верстка обеих страниц в соответствии с макетом

Теперь, когда в шаблон “**article.html**” была успешно передана переменная “**post**”, можно позаботиться об её отображении в формате html. Для этого в папке **articles/templates/** создайте файл “**article.html**”. Содержимое его вы можете определить сами, оно будет ещё проще, чем описанная в прошлой лабораторной работе страница вывода всех записей, ведь здесь не будет необходимости создавать цикл, запись-то всего одна.

Теперь самым сложным будет создать правильные стили. Макет страницы одной записи называется **lab4_item** (рисунок 4.1), а макет главной страницы со списком постов называется **lab4_main** (рисунок 4.2). Внешне эти макеты представлены на рисунках ниже.



Рисунок 4.1. Макет страницы с одним экземпляром статьи.



Рисунок 4.2. Макет страницы со списком всех статей.

Итак, для того, что придать внешние стили странице, нужно для начала подключить css-файл. Создайте файл **article.css** и добавьте его в директорию **articles/static/css/**. Теперь подключите этот файл к обоим html-шаблонам (“**articles/templates/archive.html**” и “**articles/templates/article.html**”) по описанному во второй лабораторной работе способу.

Чтобы можно было задавать стили, специфичные для одной из страниц (например, только для **article.html**) тегу **<body>** каждой страницы дайте уникальный класс - для **archive.html** это будет класс **archive**, а для страницы **article.html** будет класс **article**. Теперь в описании стилей можно указать селектор наподобие

```
.article h1 {}
```

и стиль применится ко всем заголовкам первого уровня на странице одной записи, но не на странице архива.

Сначала задайте общие стили страницы - тегу **<body>** укажите:

```
background: #1abc9c;
font-family: Tahoma, Arial, sans-serif;
color: #ffffff;
```

Для того, чтобы горизонтально выровнять картинку-логотип, ей достаточно задать определенную ширину и автоматические боковые отступы (также тегам **img** нужно явно указывать, что это блочный элемент, с помощью стиля **display: block**).

```
.header img {
    display: block;
    width: 318px;
    margin-left: auto;
    margin-right: auto;
}
```

Теперь картинка должна встать по центру. Примерно по такому же принципу необходимо поместить по центру весь остальной контент страницы:

```
.archive {
    width: 960px;
    margin-left: auto;
    margin-right: auto;
```

```
}
```

После того, как вы сделали названия постов на странице **archive** ссылками, у них установился синий цвет. Чтобы это исправить, достаточно явно указать белый цвет шрифта:

```
.post-title a {  
    color: #ffffff;  
}
```

Для того, чтобы имя автора и время создания поста находились на одном горизонтальном уровне, имени автора можно задать обтекание слева (не забыв указать ширину меньшую, чем вся страница, иначе времени создания просто негде будет поместиться справа!)

```
.article-author {  
    width: 50%;  
    float: left;  
}
```

Однако, время создания не находится в самой правой части экрана, для исправления нужно лишь сделать выравнивание текста по правому краю:

```
.article-created-date {  
    text-align: right;  
}
```

Задание:

- Создайте стили, которые соответствуют макету, для страницы определенной записи. Не забудьте добавить и стилизовать ссылку “**Все записи**”.

Лабораторная работа №5.

Создание формы и представления для нового поста.

Основные принципы работы Django с формами описаны в части первой главы 6 учебного пособия [1], прочтите её и запомните.

Создайте папку **lab5**, в неё скопируйте прошлый проект **blog**. В файле **urls.py** создайте ещё один адрес: “**article/new/**”, который будет обрабатываться функцией-представлением **create_post**. Создайте в файле представлений наряду с функциями **archive** и **get_article** новую функцию **create_post** (на которую и ссылается только что написанный урл). Эта функция должна будет при первом обращении к странице просто возвращать html-код с формой создания новой записи (html-шаблон для этого будет написан чуть позже), а в случае, если к функции пользователь обратился во второй и следующий разы, при этом заполнив форму с названием и текстом нового поста, представление должно сохранять отправленные через форму данные в БД. Иными словами, сейчас вам необходимо создать представление, которое выполняет ту же роль, что и страница добавления статьи в административном интерфейсе (эта страница при запущенном проекте должна быть доступна по ссылке <http://localhost:8000/admin/articles/article/add/>).

В первую очередь в представлении проверьте, авторизован ли пользователь на сайте, потому что у каждого поста есть поля автора, которое не может быть корректно установлено, если автор не авторизован. Соответственно, всю работу необходимо проводить, только если данная проверка была успешной пройдена:

```
def create_post(request):
    if not request.user.is_anonymous():
        # Здесь будет основной код представления
    else:
        raise Http404
```

Если пользователь неанонимен, то сначала проверьте, что сейчас программе необходимо сделать: просто вернуть страницу с формой, или пользователь в свое время уже форму получил, заполнил её и сейчас от программы требуется сохранить новую запись. Для этого нужно знать метод, с помощью которого отправлен запрос, и если это метод **POST**, то необходимо проверить полученные данные (если не заполнено одно из полей, вернуть обратно страницу с формой и уведомить об ошибке) и, если проверка пройдена, создать в БД новую запись.

```
if request.method == "POST":
    # обработать данные формы, если метод POST
    form = {
        'text': request.POST["text"],
        'title': request.POST["title"]
    }
    # в словаре form будет храниться информация, введенная пользователем
    if form["text"] and form["title"]:
        # если поля заполнены без ошибок
        Article.objects.create(text=form["text"],
                               title=form["title"],
                               author=request.user)
        return redirect('get_article', article_id=article.id)
        # перейти на страницу поста
    else:
        # если введенные данные некорректны
        form['errors'] = u"Не все поля заполнены"
        return render(request, 'create_post.html', {'form': form})
else:
    # просто вернуть страницу с формой, если метод GET
    return render(request, 'create_post.html', {})
```

Для того, чтобы перейти на главную страницу, в django из представления можно вернуть результат функции **redirect()**, которой в качестве параметра достаточно передать имя **url**-а (имя для каждого адреса определяется в файле **urls.py** в параметре **name**, для страницы определенной записи url-адрес как раз назывался “**get_article**” и имел параметр **article_id**) и указать, как заполнить именованные группы в регулярном выражении.

Когда функция-представление была создана, можно приступить к реализации шаблона с формой.

```
<!DOCTYPE html>
```

```

<html>
  <head>
    <title>Django Public Blog - создать статью</title>
  </head>
  <body>
    <div class="content">
      <h1>Написать статью</h1>
      <form method="POST">{% csrf_token %}
        <input type="text" name="title" placeholder="Название
статьи" value="{{ form.title }}">
        <textarea name="text" placeholder="Текст
статьи">{{ form.text }}</textarea>
        <input type="submit" value="Сохранить">
      </form>
      {{ form.errors }}
    </div>
  </body>
</html>

```

Тег `form` нужен для объединения нескольких полей ввода, чтобы при отправлении в запрос добавить все данные, которые вбил пользователь. Этот тег становится особенно важным, когда на одной странице присутствуют сразу несколько полей ввода, которые нужно отправлять независимо, например, если сразу есть и поля авторизации где-то справа сверху (поля логина и пароля) и поля, например, добавления комментария внизу страницы. Если пользователь оставляет комментарий, на сервер не надо слать данные авторизации (которые к тому же вряд ли заполнены, ведь не будет пользователь просто так вводить лишнюю информацию) и, наоборот, если человек желает авторизоваться, незачем ждать, что он одновременно напишет комментарий к статье.

Внутри тега форм сразу же помещается специальный шаблонный тег `{% csrf_token %}`, который добавляет специальное скрытое поле в форму, это поле легко увидеть, если посмотреть содержимое формы через консоль отладки, например, в Google Chrome (для её открытия нажмите при работе с браузером клавишу f12). Затем следуют два самых главных элемента - `<input>` для ввода названия текста и `<textarea>` для ввода самой записи. Эти два тега отличаются в первую очередь тем, что `<textarea>` позволяет вводить текст на нескольких строках, `<input>` же в свою очередь всегда имеет однострочную форму. К тому же значение по умолчанию для тега `<input>` необходимо указывать в атрибуте `value`, а для тега `<textarea>` просто написав любой текст между открывающим тегом и закрывающим. Также в форму добавлена кнопка для отправки данных на сервер, а после перечисления всех полей указываются все допущенные пользователем ошибки (естественно, если эти ошибки имели место быть). После создания шаблона можно приступать к тестированию, создайте несколько статей через вашу форму.

Задание:

- Создайте стили, подключив css-файл к шаблону, которые соответствуют макету **lab5_creation_form**.
- Добавьте проверку на то, что введенное для нового поста название уникально.

Лабораторная работа №6.

Создание формы, шаблона и представления для авторизации.

Перед выполнением этой лабораторной работы внимательно ознакомьтесь со всей главой 6 учебного пособия [1].

Система авторизации/регистрации в общих чертах для платформы django уже создана создателями, и код приложения находится в пакете **django.contrib.auth**.

Например, сама модель пользователя находится в модуле **django.contrib.auth.models.User**. Для большего понимания можно прямо в коде платформы django посмотреть, что создатели в модель пользователя вложили, к тому же код там довольно простой и интуитивно понятный.

Для регистрации пользователя достаточно воспользоваться методом **create_user** (этот метод можно найти в объявлении модели User в исходном коде фреймворка). Использовать эту функцию можно следующим образом:

```
from django.contrib.auth.models import User

User.objects.create_user("Vasya", "vasya@mail.ru", "moy_parol")
```

Как видно из примера, в этот метод можно передать никнейм, email и пароль. Главной задачей теперь становится получение данных от пользователя - однако, решить её очень просто, создав страницу с формой регистрации и должным образом обработав значения. В принципе задача по сложности не превосходит создание нового поста, так что конкретная реализация оставляется на самостоятельную работу. Единственной возможной трудностью может стать проверка на то, был ли уже зарегистрирован какой-либо пользователь под текущим никнеймом, чтобы это проверить, достаточно попробовать его найти в БД с помощью метода **get**, который вызовет исключение, если объекта не существует:

```
try:
    User.objects.get(username=username)
    # если юзер существует, то ошибки не произойдет и
```

```
# программа удачно доберется до следующей строки
print "Такой юзер уже есть"
except User.DoesNotExist:
    print "Такого юзера ещё нет"
```

Задание:

- Создайте шаблон и настройте адрес для отображения страницы регистрации.
- Создайте представление, которое обрабатывает поступающие запросы и регистрирует новых пользователей. Не забудьте сделать проверку на то, что отправленные поля непусты, а введенное имя пользователя уникально.
- Сверстайте страницу в соответствии с макетом **lab6_registration_form**.
- Добавьте в шапку страниц всех записей и страниц для определенных статей ссылку на регистрацию в верхнем правом углу (стиль ссылке сделать точно такой же, как и ссылки “**Все статьи**” на собственных страницах постов)

Создание формы, шаблона и представления для регистрации.

Для того, чтобы аутентифицировать (аутентификация - процедура проверки подлинности, иными словами проверка, подходит ли пароль для указанного аккаунта) существующего пользователя, можно воспользоваться функцией **authenticate** из **django.contrib.auth**. Эта функция принимает именные параметры и сверяет их с теми данными, что лежат в БД, а потом возвращает объект **User**, если данные верны, в противном же случае возвращает **None**:

```
from django.contrib.auth import authenticate

user = authenticate(username="Vasya", password="Secret")
```

После аутентификации, если всё получилось, то можно смело авторизовать пользователя с помощью функции **login**, которая находится все в том же пакете, в **django.contrib.auth**. Эта функция принимает два параметра (текущий объект запроса и объект пользователя):

```
from django.contrib.auth import login  
  
login(request, user)
```

Если же попытка аутентификации провалилась и в переменную **user** было сохранено значение **None**, пользователю нужно вернуть шаблон с теми же формами входа, только внизу указав ошибку *«Нет аккаунта с таким сочетанием никнейма и пароля»*.

Задание:

- Создайте шаблон и настройте адрес для отображения страницы авторизации.
- Создайте представление, которое обрабатывает поступающие запросы и авторизует пользователей. Не забудьте сделать проверку на то, что отправленные поля непусты, логин и пароль соответствуют одному из аккаунтов вашего проекта.
- Сверстайте страницу в соответствии с макетом **lab6_authorization_form**.

Лабораторная работа №7.

Изучение основ JavaScript, создание простейших функций и использование базовых операторов.

Основы работы JavaScript описаны в главе 7 учебного пособия [1].

Внимательно ознакомьтесь с материалами перед началом выполнения лабораторной работы.

JavaScript - язык программирования с динамической типизацией, очень широко распространен в веб-разработке. Сценарии на языке JavaScript загружаются браузером с сервера и выполняются на компьютере пользователя, соответственно для многих задач при работе на JavaScript не нужно отсылать лишних запросов на сервер и ждать ответа для обработки. К тому же, в JavaScript есть широкие возможности обмена информацией с сервером без перезагрузки веб-страницы, что активно используется во многих современных приложениях, например в Gmail или на сайте компании 2GIS. Но все же чаще всего данный язык используется для создания скриптов различных анимаций, манипулирования элементами DOM.

В синтаксисе JavaScript отсупы не играют решающего значения в отличие от языка Python, однако в JavaScript ни в коем случае нельзя забывать открывать и закрывать блоки кода для циклов, условий, функций и так далее. Переменные в JavaScript можно объявлять в любом месте программы, однако при первом использовании обязательно указывайте оператор **var**, без которого будет создана одна из глобальных переменных, которые часто приводят к непредсказуемым ошибкам в коде.

Для просмотра JavaScript-кода веб-страницы, достаточно в панели разработчика браузера Google Chrome (или в аналогичных панелях других браузеров) перейти на вкладку Source и затем выбрать файл, исходный код которого вас интересует.

Итак, пора приступать к созданию первого сценария на JavaScript!

Скопируйте созданный в прошлой лабораторной работе проект в папку **lab7**, после чего в папке **articles/static/** рядом с папкой **css/** создайте папку **js/**. Внутри этой директории создайте файл **helloworld.js**. В итоге для приложения **articles** у вас должна быть следующая файловая структура:

```
articles/  
  static/  
    css/  
      article.css  
    js/  
      helloworld.js  
    img/  
      dpb-logo.png  
  templates/  
    archive.html  
    article.html  
    create_post.html  
    login.html  
    registration.html  
  __init__.py  
  admin.py  
  models.py  
  tests.py  
  views.py
```

Чтобы подключить JavaScript-файл к веб-странице, достаточно вставить следующий тег в html-шаблон:

```
<script src="{{ STATIC_URL }}js/helloworld.js"></script>
```

Тег этот может находиться в любом месте файла, только не за пределами базовых тегов **<head>** и **<body>**, желательно скрипты подключать ближе к закрывающемуся **<body>**, потому что таким образом загрузка сценариев не будет мешать загрузке остального, главного контента, такого, как текст, картинки, стили и шрифты.

Подключите файл **helloworld.js** к html-шаблону **archive.html**. Запустите проект и пройдите по адресу <http://127.0.0.1:8000/>. Запустите панель отладчика, перейдите на вкладку Network и убедитесь, что js-файл есть в списке загруженных файлов, как это показано на рисунке 7.1.

Name	Method	Status	Type	Initiator	Size	Time	Timeline
Path		Text			Content	Latency	
localhost	GET	200 OK	text/html	Other	3.4 KB 3.3 KB	71 ms 71 ms	
article.css /static/css	GET	200 OK	text/css	localhost/:5 Parser	586 B 402 B	18 ms 17 ms	
dpb-logo.png /static/img	GET	304 NOT MODIFI	image/png	localhost/:9 Parser	128 B 6.6 KB	27 ms 26 ms	
helloworld.js /static/js	GET	200 OK	applicatio...	localhost/:68 Parser	196 B 0 B	25 ms 25 ms	

4 requests | 4.3 KB transferred | 188 ms (load: 249 ms, DOMContentLoaded: 248 ms)

Рисунок 7.1. Файл *hellowrold.js* появился в списке загруженных файлов.

Когда файл был успешно загружен, можно начинать писать его исходный код.

Создайте список студентов вашего факультета (опять же, не нужно указывать всех, достаточно пяти-шести человек):

```
var groupmates = [
  {
    "name": "Василий",
    "group": "912-2",
    "age": 19,
    "marks": [4, 3, 5, 5, 4]
  },
  {
    "name": "Анна",
    "group": "912-1",
    "age": 18,
    "marks": [3, 2, 3, 4, 3]
  },
  {
    "name": "Георгий",
    "group": "912-2",
    "age": 19,
    "marks": [3, 5, 4, 3, 5]
  },
  {
    "name": "Валентина",
    "group": "912-1",
    "age": 18,
    "marks": [5, 5, 5, 4, 5]
  }
];
```

Все параметры для каждого студента остались ровно теми же, что и в первой лабораторной работе (если честно, даже значение по каждому студенту изменены не были). Как видите, синтаксис и обычных, и ассоциированных массивов практически ничем не отличается от синтаксиса языка Python. Главными отличиями являются ключевое слово **var** перед названием переменной и точка с запятой после оператора присваивания. Чтобы можно было увидеть значение переменной (иными словами, для вывода переменной на экран), воспользуйтесь методом **console.log()**, который принимает через запятую аргументы, которые вы желаете увидеть:

```
console.log(groupmates);
```

Теперь перезагрузите страницу и в панели разработки Google Chrome (клавиша f12) пройдите на вкладку Console. Там вы сможете увидеть свернутый вариант вашего массива, изображенный на рисунке 7.2.



Рисунок 7.2. Сжатое изображение массива *groupmates*.

Чтобы изучить содержимое массива подробнее, кликните прямо перед массивом по серой стрелке, указывающей направо, и разверните массив, как показано на рисунке 7.3.



Рисунок 7.3. Полное изображение массива *groupmates*.

Создайте функцию, которая будет выводить в виде таблицы содержимое массива **groupmates** (вид таблицы должен напоминать тот, что использовался в первой лабораторной работе).

```
var rpad = function(str, length) {
    // js не поддерживает добавление нужного количества символов
    // справа от строки то есть аналога ljust из языка Python здесь нет
    str = str.toString(); // преобразование в строку
    while (str.length < length)
        str = str + ' '; // добавление пробела в конец строки
    return str; // когда все пробелы добавлены, вернуть строку
};

var printStudents = function(students){
    console.log(
        rpad("Имя студента", 15),
        rpad("Группа", 8),
        rpad("Возраст", 8),
        rpad("Оценки", 20)
    );
    // был выведен заголовок таблицы
    for (var i = 0; i<=students.length-1; i++){
        // в цикле выводится каждый экземпляр студента
        console.log(
            rpad(students[i]['name'], 15),
            rpad(students[i]['group'], 8),
            rpad(students[i]['age'], 8),
            rpad(students[i]['marks'], 20)
        );
    }
    console.log('\n'); // добавляется пустая строка в конце вывода
};
printStudents(groupmates);
```

Как было указано в комментарии, язык JavaScript не имеет собственных средств форматирования строки через добавление пробелов, поэтому была реализована новая функция **rpad()**. Затем была создана функция **printStudents**, которая и выводит список всех пользователей. Обратите внимание, что в языке **python** слова в названии переменных разделяются нижним подчеркиванием (**print_students**), а в языке JavaScript принята так называемая “верблюжья нотация”, то есть имена разделяются заглавными буквами (**printStudents**).

Задание:

- Напишите функцию, которая фильтрует студентов по группе. Функция должна возвращать только тех студентов, что учатся в указанной группе.

Работа с элементами DOM с помощью JavaScript.

Главные возможности языка JavaScript по манипулированию элементами DOM описаны в главе 8 учебного пособия [1].

С помощью JavaScript можно внести немного динамичности самой странице, иными словами можно манипулировать элементами html-разметки, перемещая их, изменяя свойства или трансформируя. Добавьте к вашей главной странице возможность скрыть информацию по каждому посту после нажатия на клавишу “свернуть”. Дизайн новой главной страницы показан на рисунке 7.4.

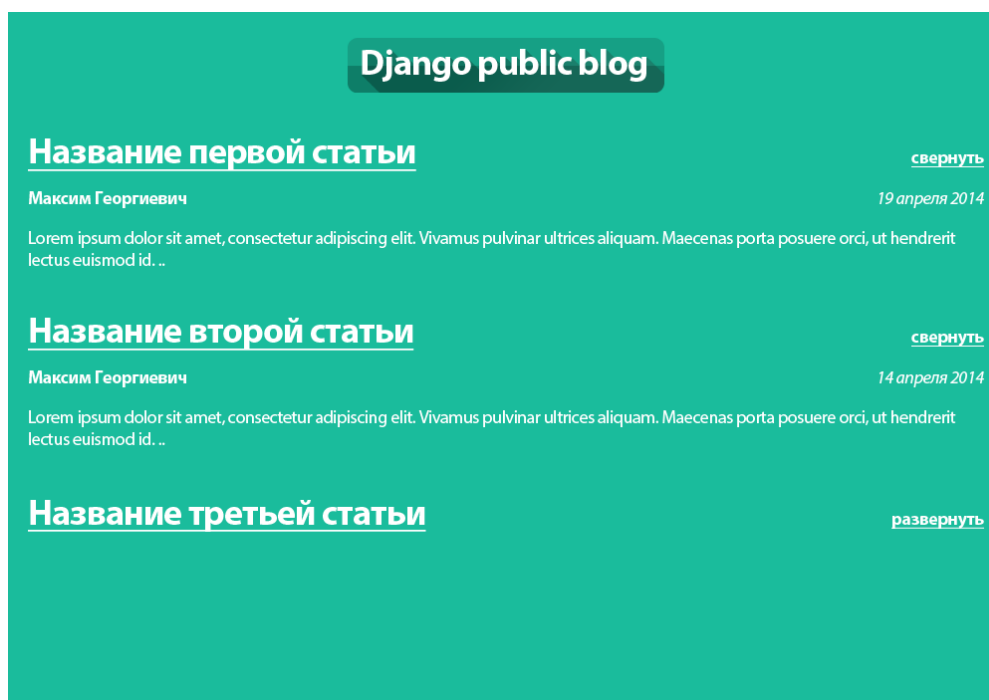


Рисунок 7.4. Дизайн списка постов с возможностью сворачивания информации.

Как можно заметить из дизайна, при нажатии на кнопку “свернуть” у статьи должны скрывать имя автора, время создания и цитата, а текст самой кнопки должен сменить с “свернуть” на “развернуть”.

Итак, для начала добавьте к каждой статье в html-разметке кнопку “свернуть” в том месте, какое указано в макете, также добавьте каждой кнопке класс “**fold-button**”. После этого создайте файл **fold-post.js** в папке **articles/static/js/**.

Создайте функцию, которая будет вызываться каждый раз, как пользователь кликает по кнопке, и выводить в консоль сообщение. Для этого в js-файле введите следующий код:

```
var foldBtns = document.getElementsByClassName("fold-button");

for (var i = 0; i<foldBtns.length; i++){
    foldBtns[i].addEventListener("click", function(event) {
        console.log("you clicked ", event.target);
    });
}
```

Сначала у глобальной переменной **document** вызывается метод **getElementsByClassName**, который ищет по структуре DOM все элементы с указанным классом и возвращает их массив. Затем остается лишь в цикле **for** пройти по всем элементам этого массива и каждому элементу добавить обработчик события “**click**” с помощью метода **addEventListener**, который принимает два аргумента: название события и функцию, которая будет при каждом событии вызываться.

Заметьте, что функция-обработчик может принимать аргумент (имя которого не имеет значения), и именно в этот аргумент будет вноситься объект, содержащий основную информацию о событии (ссылку на сам элемент, по которому кликнули, координаты курсора в момент события, какие были зажаты в этот момент клавиши и много других важных значений).

Теперь с помощью панели разработки в Google Chrome поставьте точку останова (breakpoint) на 5-ой строке, щелкнув по номеру строки, эта область выделена на рисунке 7.5.

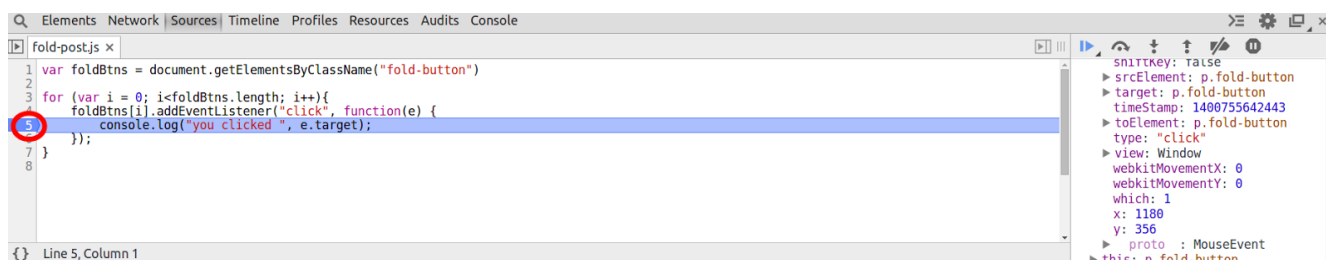


Рисунок 7.5. Создание точки останова в панели разработки Google Chrome.

После этого кликните по кнопке “свернуть”, чтобы запустить тот участок кода, где был установлен breakpoint (именно поэтому точку останова надо было сделать на пятой строке, ведь это непосредственно исходный код функции-обработчика, который выполняется при каждом клике). Исполняемый код должен остановиться как раз на пятой строке, подсветив её синим цветом и затемнив весь остальной экран страницы. Теперь вам доступна отладка вашего JavaScript-кода в пошаговом режиме. С деталями работы в панели разработки вы можете познакомиться из многочисленных источников в сети Интернет. Сейчас же попробуйте заставить программу продолжить исполнять ваши инструкции. Справа вверху нажмите кнопку “Продолжить выполнение скрипта” (“resume script execution”). Теперь снова нажмите на кнопку “свернуть”, чтобы панель снова перешла в режим пошаговой отладки. Обратите внимание на правую колонку, выделенную на рисунке 7.6 красным цветом, где помимо кнопок управления выполнением скрипта, перечислены переменные “Scope variables”. Найдите и исследуйте значение переменной **event**, которая является аргументом функции-обработчика.



Рисунок 7.6. Исследование переменных для текущей точки выполнения.

Теперь пора приступить к реализации исчезновения и появления информации о статье. Поместите следующий код внутри функции-обработчика:

```

foldBtns[i].addEventListener("click", function(e) {
    event.target
        .parentElement
        .getElementsByClassName('article-author')[0]
        .style.display = "none";
    event.target
        .parentElement
        .getElementsByClassName('article-created-date')[0]
        .style.display = "none";
    event.target
        .parentElement
        .getElementsByClassName('article-text')[0]
        .style.display = "none";
    e.target.innerHTML = "развернуть";
});

```

Заметьте, что операция по установке стиля **display** в значение **none** занимает несколько довольно много места, поэтому было принято решение разбить операцию на несколько строк. Сначала в операторе с помощью атрибута **parentElement** определяется родитель элемента по которому произошел клик, затем внутри этого родителя производится поиск всех элементов с классом “**article-author**” с помощью метода **getElementsByClassName()**. Этот метод возвращает массив элементов, потому что по умолчанию один и тот же класс может иметь несколько элементов (даже если в узле DOM-а, где производится поиск, есть всего один элемент с указанным классом, метод все равно вернет список, хоть и с одним элементом). В соответствии с созданной вами html-разметкой внутри родителя **one-post** только у одного блока есть такой класс, значит метод возвращает список с одним элементом, доступ к которому можно получить по нулевому номеру. Теперь осталось этому элементу с помощью атрибутов **style** и **display** указать значение **none**. Именно css-свойство “**display: none**” позволяет скрывать элемент на странице.

Теперь осталось только у элемента, по которому был произведен клик, сменить значение текста, эта операция выполняется с помощью установки атрибута **innerHTML** в нужное положение.

После реализации исчезновения нужно суметь как-то вернуть исходное состояние при повторном клике на кнопку. Так как при любой ситуации функция-обработчик, созданная вами ранее, будет вызвана, нужно именно в ней как-то вычислить, что раньше пользователем информация о записи была скрыта и сейчас нужно вернуть её на место. Лучшее место, которое позволяет сохранить состояние поста (скрыта информация или нет), это классы. Сделайте так, чтобы при нажатии на кнопку “свернуть”, самой кнопке добавлялся класс **folded**:

```
e.target.className = "fold-button folded";
```

Не забывайте, что у кнопки уже есть класс **fold-button**, который не следует удалять, поэтому через пробел был добавлен описывающий текущее состояние класс. Однако, раз у кнопки, которая уже была нажата, есть дополнительный класс, значит вы можете в начале функции-обработчика проверять наличие этого класса, и если он есть, то не скрывать информацию о записи, а наоборот показывать её. Для этого добавьте проверку условием **if** и в случае, если у кнопки есть класс **folded**, сделайте вновь видимой скрытую информацию:

```
foldBtns[i].addEventListener("click", function(e) {
    if (e.target.className == "fold-button folded"){
        e.target.innerHTML = "свернуть";
        e.target.className = "fold-button";
        var displayState = "block";
    }
    else{
        e.target.innerHTML = "развернуть";
        e.target.className = "fold-button folded";
        var displayState = "none";
    }
    event.target
        .parentElement
        .getElementsByClassName('article-author')[0]
        .style.display = displayState;
    event.target
        .parentElement
        .getElementsByClassName('article-created-date')[0]
        .style.display = displayState;
    event.target
        .parentElement
        .getElementsByClassName('article-text')[0]
        .style.display = displayState;
```

```
});
```

Обратите внимание, что в блоке условия **if** изменяются только состояния самой кнопки (устанавливается новый текст и добавляется класс) и создается переменная **displayState**, которая содержит значение для свойства **display**. Если надо информацию скрыть, **displayState** установится в “**none**”, в противном случае **displayState** установится в “**block**” (свойство **display** имеет лишь несколько возможных состояний, проверьте их значения на сайте *htmlbook.ru*).

Задание:

- Сделайте другую реализацию функции-обработчика, которая уместилась бы в гораздо меньшее количество строк кода. Предложенный выше вариант довольно громоздок, однако он хорошо иллюстрирует некоторые возможности манипулирования DOM-ом с помощью JavaScript. Но весь описанный выше функционал можно реализовать с помощью CSS-классов и их изменения через функцию-обработчик. Сделайте так, чтобы класс **folded** добавлялся не кнопке, по которой кликнули, а родителю всего поста (это тот самый элемент, у которого в html-разметке установлен класс “**one-post**”). В таком случае свойство **display** можно изменять более элегантным путем, через CSS-стили:

```
.one-post.folded .article-author{
    /* данный стиль применится только для элементов класса
    .article-author, у которых родитель с классом one-post
    имеет также класс folded */
    display: none;
}
```

- По аналогии через CSS-стили установите исчезновение остальных элементов поста. В функции-обработчике останется только изменять текст кнопки и менять класс для элементов **one-post**.

Лабораторная работа №8.

Изучение библиотеки jQuery, добавление подсветки для наведенного поста и эффекта для картинки-логотипа.

Описание принципов работы библиотеки jQuery находится в главе 9 учебного пособия [1]. Внимательно прочтите её перед выполнением данной лабораторной работы.

Библиотека jQuery, официальный сайт который можно найти по адресу <http://jquery.com/>, создана для того, чтобы уменьшить количество шаблонного кода на JavaScript. В библиотеке уже решены многие особо распространенные проблемы, например:

- добавление класса к элементу, не изменяя при этом уже существующий список классов (как вы помните из предыдущей лабораторной, при добавлении класса следует обрабатывать уже существующий список);
- добавление эффектов с помощью одной функции;
- добавление функции-обработчика сразу на массив многих элементов, что помогает избежать написания лишнего цикла;
- поиск родственных элементов, например, поиск родителей, потомков, причем с уточнением некоторых правил (чтобы найти родителя с классом “one-post”, достаточно вызвать метод `parents(“one-post”)`).

Со страницы <http://jquery.com/download/> скачайте последнюю версию, кликнув по ссылке “*Download the compressed, production jQuery 1.x*”, где x - любое число. Возможно, вместо скачивания, браузер просто откроет окно с исходным кодом библиотеки, тогда вернитесь на страницу скачивания и кликните по ссылке “*Download*” не левой кнопкой, а правой и в контекстном меню выберете “Сохранить ссылку как...”.

Подключите библиотеку к вашей странице со списком всех записей тем же способом, что и ранее, только сейчас подключение библиотеки должно стоять раньше, чем подключение остальных js-файлов, иначе к моменту исполнения вашего исходного кода, библиотека ещё не будет доступна.

```
<script src="{{ STATIC_URL }}js/jquery.js"></script>
<script src="{{ STATIC_URL }}js/helloworld.js"></script>
<script src="{{ STATIC_URL }}js/fold-post.js"></script>
</body>
```

- примерно так должен заканчиваться ваш тег **<body>**. Теперь создайте ещё один пока что пустой js-файл с названием **“highlight-post.js”**. Именно здесь будет написан функционал, подсвечивающий статью, на которую пользователь навел курсор. Подключите и этот файл, поставив его четвертым.

Для создания эффекта подсветки будет необходим фон черного цвета, который будет изменять свою прозрачность от 100% до 80% (иными словами, от полной невидимости до небольшой). Сначала каждому элементу класса **“one-post”** добавьте в конце разметки блок с классом **“one-post-shadow”**:

```
<div class="one-post">
  <div class="post-managment">
    <h2 class="post-title"><a href="/article/
{{ post.id }}">{{ post.title }}</a></h2>
    <p class="fold-button">Свернуть</p>
  </div>
  <div class="article-info">
    <div class="article-author">{{ post.author.username }}</div>
    <div class="article-created-date">{{ post.created_date }}</div>
  </div>
  <p class="article-text">{{ post.get_excerpt }}</p>
  <div class="one-post-shadow"></div>
</div>
```

Теперь этому элементу задайте такой стиль, чтобы блок занимал всю высоту и всю ширину родительского элемента и находился на заднем плане от содержимого:

```
.one-post-shadow {
  position: absolute;
  top: 0;
  left: 0;
  height: 100%;
  width: 100%;
```

```
background: black;
z-index: -1; /* это свойство «отодвигает» элемент на задний план */
}
```

Теперь список с записями должен иметь черный фон, как на рисунке 8.1.



Рисунок 8.1. Список записей после добавления фона для каждого поста.

Чтобы скрыть фон, задайте ему свойство “**opacity: 0**”, которое сделает элемент полностью прозрачным.

Теперь, когда верстка для эффекта создана, осталось создать динамику. Откройте пока ещё пустой файл “**highlight-post.js**”. Добавьте туда следующий код:

```
$(document).ready(function() {
    $('.one-post').hover(function(event) {
        console.log("Навели");
    }, function(event) {
        console.log("Вывели");
    });
});
```

Первая строка нужна для того, чтобы код функции запустился только после того, как DOM будет полностью прогружен, в противном случае имеет место большой риск, что вы попытаетесь работать с элементами, которых ещё не существует. Во второй строке сначала с помощью команды:

```
$('.one-post')
```

производится поиск всех элементов с классом “**one-post**”, а затем с помощью метода **hover** к этим элементам добавляются два обработчика (это две функции, которые передаются методу **hover** через запятую), первый из которых будет вызываться, когда на элемент навели курсор, а второй будет вызываться, когда курсор убрали с элемента. Попробуйте теперь обновить страницу и навести на какую-либо из статей. После этого добавьте в функцию обработчик вывод в консоль того элемента, на который был наведен курсор:

```
$(document).ready(function() {
    $('.one-post').hover(function(event) {
        console.log("Навели");
        console.log(event.currentTarget);
    }, function(event) {
        console.log("Вывели");
    });
});
```

Обратите внимание, что ранее у объекта события **event** использовался атрибут **target**, а сейчас был выведен атрибут **currentTarget**. Попробуйте теперь поэкспериментировать и вывести в консоль просто атрибут **target**, чтобы увидеть разницу. Определите, на какие элементы обычно указывает **event.target**, а на какие **event.currentTarget**.

Теперь добавьте сам эффект анимации:

```
$(document).ready(function() {
    $('.one-post').hover(function(event) {
        $(event.currentTarget).find('.one-post-shadow').animate({opacity:
'0.1'}, 300);
    }, function(event) {
        $(event.currentTarget).find('.one-post-shadow').animate({opacity: '0'},
300);
    });
});
```

С помощью **\$(event.currentTarget)** элемент, на который навели курсором (ведь именно он хранится в переменной **event.currentTarget**) получает все возможности библиотеки jQuery. Теперь можно воспользоваться jQuery-методом **find(selector)**, который отыскивает среди потомков текущего элемента всех потомков, которые подходят под описание, переданное в переменной **selector** (естественно, **selector** - всего лишь пример, на его месте может быть просто строка, как в текущем случае, а может быть какая-нибудь другая переменная). Именно тому элементу, который был определен с помощью **find('.one-post-shadow')** и будет сделана анимация, которая описана следующим образом **animate({opacity: '0.1'}, 300)**, что означает: изменить непрозрачность до 0.1 за 300 миллисекунд.

Во второй функции, которая передана через запятую от первой и которая, как говорилось выше, выполняется, когда курсор уходит с элемента, непрозрачность будет возвращена на предыдущий уровень в значение 0.

Задание:

- Сделайте так, чтобы при наведении на картинку-логотип она увеличивалась в размерах (ширина становилась больше на 20px, а высота увеличивалась пропорционально).

Лабораторная работа №9.

Дальнейшее изучение библиотеки jQuery, добавление эффекта параллакса для главной страницы блога.

В прошлых лабораторных работах вы получили довольно основательный опыт работы с языком JavaScript и библиотекой jQuery. Теперь вам следует выполнить действительно сложную анимацию, которая пользуется довольно большой популярностью на современных веб-сайтах — добавить на страницу эффект параллакса. Этим эффектом называют неравномерное движение нескольких слоев, которые находятся друг под другом. Причем визуально отдаленные слои движутся со скоростью меньшей, чем слои, которые визуально находятся ближе.

Для начала создайте папку lab9, в которую скопируйте проект из прошлой лабораторной работы. Затем добавьте в верстку вашей страницы со списком всех постов небольшие изменения в соответствии с макетом **lab9_for_parallax**, который представлен на рисунке 9.1.

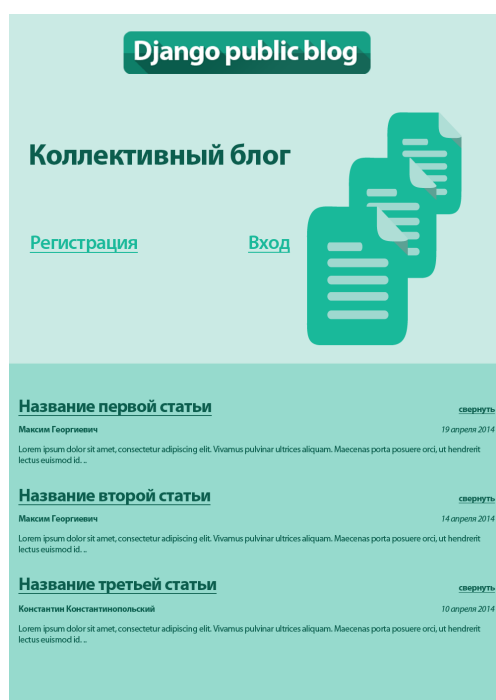


Рисунок 9.1. Макет нового дизайна для страницы со списком всех статей.

В верстке страницы нужно будет:

- изменить цвет фона для основной части страницы (там, где находится сам список статей).
- Изменить цвет шрифта для текстов.
- Добавить отдельную шапку для страницы. В шапке должны присутствовать:
 - Картинка-логотип, которая не претерпела изменений.
 - Фраза «Коллективный блог», которая лаконично характеризует ваш сервис.
 - Ссылка на регистрацию и ссылка на авторизацию.
 - Три одинаковые картинки, которым вы через CSS-стили должны указать разные размеры. Для картинки на переднем плане укажите ширину в 200px, для картинки посередине 175px, а для последней картинки 150px.
- У шапки должен быть свой собственный цвет фона, который вы можете узнать из макета.

В итоге HTML-разметка одной только шапки может принять примерно следующую форму:

```
<div class="header">
  
  <div class="header-text-area">
    <p class="service-title">Коллективный блог</p>
    <a href="/register/" class="register-link">Регистрация</a>
    <a href="/login/" class="register-link">Вход</a>
  </div>
  <div class="icons-for-parallax">
    
    
```

```

        
    </div>
</div>

```

Для того, чтобы поместить блок **.header-text-area** слева от блока **.icons-for-parallax**, не забудьте первому присвоить стиль **float:left** и указать значение ширины в соответствии с макетом (если ширину не указать, блок **.header-text-area** хоть и будет обтекаем, но он не позволит какому-либо другому элементу поместиться справа от себя, так как по умолчанию займет всю доступную ему ширину страницы).

Для того, чтобы разместить картинки одна под другой, и ещё при этом сдвинуть их относительно друг друга, можете применить следующие стили:

```

.icons-for-parallax {
    margin-top: 100px;
    margin-bottom: 100px;
    height: 530px;
    position: relative;
}
.icons-for-parallax img {
    /* здесь находятся общие для всех картинок стили */
    position: absolute;
    top: 0;
}
.icon-for-parallax-first {
    margin-top: 200px;
    z-index: 3; /* эта картинка на переднем плане, у неё больший z-index */
}
.icon-for-parallax-second {
    width: 175px; /* уменьшить ширину для картинки, чтобы визуально */
                /* находилась дальше для пользователя */
    margin-top: 100px; /* эта картинка слегка смещена вправо и вниз */
    margin-left: 80px; /* за счет левой и верхней границ */
    z-index: 2;
}
.icon-for-parallax-third{
    width: 150px;
    margin-left: 160px;
    z-index: 1; /* эта картинка на заднем плане, у неё меньший z-index */
}

```

После внесения всех изменений в верстку, добавьте в папку **articles/static/js/** новый файл **parallax.js**. Именно в этом файле будет содержаться исходный код скрипта, который изменяет координаты иконок при прокрутке страницы.

Примерный ход работы скрипта:

- инициировать начальные значения, то есть установить высоту прокрученной области в 0 пикселей, сохранить в переменную **\$parallaxElements** все DOM-элементы, на которые будет добавлена анимация. Эта переменная нужна для того, чтобы затем каждый раз при прокрутке страницы не искать внутри DOM нужные для скрипта элементы, ведь обход всего документа обычно занимает значительное время.
- Каждый раз при прокрутке считать количество проскролленных (от англ. «to scroll» — прокручивать) пикселей.
- После подсчета пикселей нужно запустить цикл, который обойдет все элементы в переменной **\$parallaxElements**, подсчитает для каждого координаты и установит их.

Для выполнения всех шагов начните со стандартной для каждого файла, где подключен jQuery, операции: ожидания события ready для документа:

```
$(document).ready(function() {  
    // код будет здесь  
});
```

Теперь можно быть уверенным, что код запустится только после окончательно загрузки DOM. Для шагов инициализации, описанных выше, добавьте в свой файл следующий исходный код:

```
var scrolled = 0;  
var $parallaxElements = $('.icons-for-parallax img');
```

Чтобы можно было при каждой прокрутке выполнять какие-либо операции, на событие **scroll** объекта **window** добавьте функцию-обработчик:


```
$(window).scroll(function() {
    // код, который нужно выполнять при
    // каждой прокрутке, должен быть здесь
});
```

Как вы помните, внутри этой функции вас сначала следует узнать количество прокрученных пикселей, для чего используйте следующую функцию так, как показано ниже.

```
scrolled = $(window).scrollTop(); // Обновление значения текущей
прокрутки
```

Естественно, эту операцию нужно выполнять внутри функции-обработчика события **scroll**. Затем там же, в этой функции запустите цикл по всем элементам внутри переменной **\$parallaxElements**.

```
for (var i = 0; i < $parallaxElements.length; i++){
    // здесь вам следует описать манипуляции для каждого
    // элемента из переменной $parallaxElements
}
```

Внутри цикла сначала посчитайте на сколько пикселей вам нужно визуально опустить текущий элемент.

```
yPosition = (scrolled * 0.15*(i + 1));
// подсчет нужного количества пикселей
// для текущего элемента, обратите внимание, что-либо
// здесь переменная зависит от значения i
```

Осталось только установить уже подсчитанные координаты для текущего элемента:

```
$parallaxElements.eq(i).css({ top: yPosition });
```

метод **eq()**, позволяет из всего списка элементов, который хранится в jQuery-переменной, выбрать именно тот элемент, позиция которого совпадает с переданным методу числом. В данном случае вы в цикле последовательно из списка берете 0-ой элемент, затем 1-ый, и, наконец, 2-ой. Для каждого из них вы устанавливаете координату, которая напрямую зависит от номера позиции, ведь в операции подсчета пикселей также фигурирует переменная **i**.

В конечном итоге код вашего скрипта должен иметь примерно следующий вид:

```
$(document).ready(function() {  
    var yPosition;  
    var scrolled = 0;  
    var $parallaxElements = $('.icons-for-parallax img');  
    $(window).scroll(function() {  
        scrolled = $(window).scrollTop();  
        for (var i = 0; i < $parallaxElements.length; i++) {  
            yPosition = (scrolled * 0.15*(i + 1));  
            $parallaxElements.eq(i).css({ top: yPosition });  
        }  
    });  
});
```

После написания этого сценария, пройдите на главную страницу вашего проекта со списком всех записей и обновите страницу. Убедитесь, что иконки статей действительно перемещаются с разной скоростью.

Задание:

- Настройте также эффект параллакса для картинки-логотипа вашего сервиса. Параметры для скорости перемещения задайте самостоятельно.

Список использованной литературы

1. Создание веб-приложений. Учебное пособие. А. В. Титков, С. А. Черепанов. Томский государственный университет систем управления и радиоэлектроники, 101 стр. - 2014.
2. Глава 2, раздел 5: DjangoBook по-русски | Django на русском [Электронный ресурс] // Руслан Попов, Дмитрий Косточко. URL: <http://djbook.ru/ch02s05.html> (дата обращения: 17.05.2014).